Contents

1 The basis of Higher-Order Logic 31
  1.1 Primitive logic ........................................... 32
    1.1.1 Core syntax .................................... 32
    1.1.2 Defined connectives and quantifiers .......... 32
    1.1.3 Additional concrete syntax .................... 33
    1.1.4 Axioms and basic definitions ................. 34
  1.2 Fundamental rules .................................. 35
    1.2.1 Equality ........................................ 35
    1.2.2 Congruence rules for application ............. 36
    1.2.3 Equality of booleans – iff .................. 36
    1.2.4 True (1) ...................................... 37
    1.2.5 Universal quantifier (1) ....................... 37
    1.2.6 False ......................................... 37
    1.2.7 Negation ....................................... 38
    1.2.8 Implication .................................... 38
    1.2.9 Disjunction (1) ................................ 39
    1.2.10 Derivation of iffI .............................. 39
    1.2.11 True (2) ...................................... 40
    1.2.12 Universal quantifier (2) ....................... 40
    1.2.13 Existential quantifier ......................... 40
    1.2.14 Conjunction .................................. 40
    1.2.15 Disjunction (2) ................................ 41
    1.2.16 Classical logic ................................ 41
    1.2.17 Unique existence ............................... 42
    1.2.18 Classical intro rules for disjunction and existential quantifiers ......................... 42
    1.2.19 Intuitionistic Reasoning ....................... 43
    1.2.20 Atomizing meta-level connectives ............ 44
    1.2.21 Atomizing elimination rules ................... 45
  1.3 Package setup ...................................... 46
    1.3.1 Sledgehammer setup ............................. 46
1.3.2 Classical Reasoner setup ........................................ 46
1.3.3 THE: definite description operator .............................. 48
1.3.4 Simplifier ......................................................... 49
1.3.5 Generic cases and induction .................................... 59
1.3.6 Coherent logic .................................................. 62
1.3.7 Reorienting equalities ........................................... 62
1.4 Other simple lemmas and lemma duplicates .................... 63
1.5 Basic ML bindings .................................................. 64

2 NO-MATCH simproc .................................................. 66
2.1 Code generator setup ............................................... 67
  2.1.1 Generic code generator preprocessor setup .................. 67
  2.1.2 Equality ....................................................... 68
  2.1.3 Generic code generator foundation ............................ 68
  2.1.4 Generic code generator target languages .................... 70
  2.1.5 Evaluation and normalization by evaluation ................ 71
2.2 Counterexample Search Units ..................................... 72
  2.2.1 Quickcheck ................................................... 72
  2.2.2 Nitpick setup ................................................ 72
2.3 Preprocessing for the predicate compiler ....................... 72
2.4 Legacy tactics and ML bindings .................................. 72

3 Abstract orderings ................................................... 73
3.1 Abstract ordering .................................................. 73
3.2 Syntactic orders ................................................... 75
3.3 Quasi orders ....................................................... 76
3.4 Partial orders ..................................................... 77
3.5 Linear (total) orders ............................................. 80
3.6 Reasoning tools setup ............................................. 82
3.7 Bounded quantifiers ............................................... 87
3.8 Transitivity reasoning ............................................. 89
3.9 Monotonicity ....................................................... 95
3.10 min and max – fundamental ...................................... 98
3.11 (Unique) top and bottom elements ............................... 98
3.12 Dense orders ...................................................... 100
3.13 Wellorders ......................................................... 102
3.14 Order on bool .................................................... 103
3.15 Order on -⇒ - .................................................... 104
3.16 Order on unary and binary predicates .......................... 106
3.17 Name duplicates .................................................. 107
### Groups, also combined with orderings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Dynamic facts</td>
<td>108</td>
</tr>
<tr>
<td>4.2 Abstract structures</td>
<td>108</td>
</tr>
<tr>
<td>4.3 Generic operations</td>
<td>111</td>
</tr>
<tr>
<td>4.4 Semigroups and Monoids</td>
<td>112</td>
</tr>
<tr>
<td>4.5 Groups</td>
<td>116</td>
</tr>
<tr>
<td>4.6 (Partially) Ordered Groups</td>
<td>120</td>
</tr>
<tr>
<td>4.7 Support for reasoning about signs</td>
<td>122</td>
</tr>
<tr>
<td>4.8 Canonically ordered monoids</td>
<td>135</td>
</tr>
<tr>
<td>4.9 Tools setup</td>
<td>138</td>
</tr>
</tbody>
</table>

### Abstract lattices

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Abstract semilattice</td>
<td>139</td>
</tr>
<tr>
<td>5.2 Syntactic infimum and supremum operations</td>
<td>141</td>
</tr>
<tr>
<td>5.3 Concrete lattices</td>
<td>142</td>
</tr>
<tr>
<td>5.3.1 Intro and elim rules</td>
<td>142</td>
</tr>
<tr>
<td>5.3.2 Equational laws</td>
<td>143</td>
</tr>
<tr>
<td>5.3.3 Strict order</td>
<td>146</td>
</tr>
<tr>
<td>5.4 Distributive lattices</td>
<td>147</td>
</tr>
<tr>
<td>5.5 Bounded lattices and boolean algebras</td>
<td>147</td>
</tr>
<tr>
<td>5.6 min/max as special case of lattice</td>
<td>153</td>
</tr>
<tr>
<td>5.7 Uniqueness of inf and sup</td>
<td>155</td>
</tr>
<tr>
<td>5.8 Lattice on bool</td>
<td>155</td>
</tr>
<tr>
<td>5.9 Lattice on - ⇒ -</td>
<td>156</td>
</tr>
<tr>
<td>5.10 Lattice on unary and binary predicates</td>
<td>157</td>
</tr>
</tbody>
</table>

### Set theory for higher-order logic

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Sets as predicates</td>
<td>158</td>
</tr>
<tr>
<td>6.2 Subsets and bounded quantifiers</td>
<td>161</td>
</tr>
<tr>
<td>6.3 Basic operations</td>
<td>167</td>
</tr>
<tr>
<td>6.3.1 Subsets</td>
<td>167</td>
</tr>
<tr>
<td>6.3.2 Equality</td>
<td>168</td>
</tr>
<tr>
<td>6.3.3 The empty set</td>
<td>168</td>
</tr>
<tr>
<td>6.3.4 The universal set – UNIV</td>
<td>169</td>
</tr>
<tr>
<td>6.3.5 The Powerset operator – Pow</td>
<td>170</td>
</tr>
<tr>
<td>6.3.6 Set complemen</td>
<td>170</td>
</tr>
<tr>
<td>6.3.7 Binary intersection</td>
<td>171</td>
</tr>
<tr>
<td>6.3.8 Binary union</td>
<td>171</td>
</tr>
<tr>
<td>6.3.9 Set difference</td>
<td>172</td>
</tr>
<tr>
<td>6.3.10 Augmenting a set – insert</td>
<td>172</td>
</tr>
<tr>
<td>6.3.11 Singletons, using insert</td>
<td>173</td>
</tr>
<tr>
<td>6.3.12 Image of a set under a function</td>
<td>175</td>
</tr>
<tr>
<td>6.3.13 Some rules with if</td>
<td>177</td>
</tr>
<tr>
<td>6.4 Further operations and lemmas</td>
<td>178</td>
</tr>
</tbody>
</table>
6.4.1 The “proper subset” relation .................. 178
6.4.2 Derived rules involving subsets ................. 179
6.4.3 Equalities involving union, intersection, inclusion, etc. 180
6.4.4 Monotonicity of various operations .............. 190
6.4.5 Inverse image of a function .................. 191
6.4.6 Singleton sets .................................. 192
6.4.7 Getting the contents of a singleton set .......... 193
6.4.8 Least value operator ............................ 193
6.4.9 Monad operation ............................... 194
6.4.10 Operations for execution ...................... 194

7 HOL type definitions ................................. 198

8 Notions about functions ................................ 200
  8.1 The Identity Function id ......................... 201
  8.2 The Composition Operator f ◦ g .................... 201
  8.3 The Forward Composition Operator fcomp .......... 202
  8.4 Mapping functions ................................ 203
  8.5 Injectivity and Bijectivity ....................... 203
  8.6 Function Updating ............................... 214
  8.7 override-on ....................................... 216
  8.8 swap ............................................. 216
  8.9 Inversion of injective functions .................. 218
  8.10 Cantor’s Paradox ................................ 219
  8.11 Monotonic functions over a set .................. 219
  8.12 Setup ........................................... 221
    8.12.1 Proof tools ................................ 221
    8.12.2 Functorial structure of types ............... 221

9 Complete lattices ..................................... 222
  9.1 Syntactic infimum and supremum operations ........ 222
  9.2 Abstract complete lattices ....................... 223
  9.3 Complete lattice on bool ......................... 234
  9.4 Complete lattice on - ⇒ - ........................ 235
  9.5 Complete lattice on unary and binary predicates .. 236
  9.6 Complete lattice on - set ....................... 237
    9.6.1 Inter ........................................ 238
    9.6.2 Intersections of families .................... 239
    9.6.3 Union ........................................ 241
    9.6.4 Unions of families ........................... 242
    9.6.5 Distributive laws ............................. 245
  9.7 Injections and bijections .......................... 246
    9.7.1 Complement ................................. 248
    9.7.2 Miniscoping and maxiscoping ................. 248
10 Wrapping Existing Freely Generated Type’s Constructors 250

11 Knaster-Tarski Fixpoint Theorem and inductive definitions 251
   11.1 Least fixed points ............................................. 251
   11.2 General induction rules for least fixed points .......... 252
   11.3 Greatest fixed points ........................................... 253
   11.4 Coinduction rules for greatest fixed points ............. 254
   11.5 Even Stronger Coinduction Rule, by Martin Coen .... 256
   11.6 Rules for fixed point calculus .............................. 257
   11.7 Inductive predicates and sets .............................. 259
   11.8 The Schroeder-Bernstein Theorem ........................... 259
   11.9 Inductive datatypes and primitive recursion ........... 261

12 Cartesian products 261
   12.1 bool is a datatype .............................................. 262
   12.2 The unit type .................................................. 262
   12.3 The product type ................................................. 265
      12.3.1 Type definition ........................................... 265
      12.3.2 Tuple syntax .............................................. 267
      12.3.3 Code generator setup .................................... 269
      12.3.4 Fundamental operations and properties ............. 269
      12.3.5 Derived operations ................................... 276
   12.4 Simproc for rewriting a set comprehension into a pointfree expression .............................................. 286
   12.5 Lemmas about disjointness .................................. 287
   12.6 Inductively defined sets ..................................... 287
   12.7 Legacy theorem bindings and duplicates ................ 288

13 The Disjoint Sum of Two Types 288
   13.1 Construction of the sum type and its basic abstract operations .... 288
   13.2 Projections .................................................... 291
   13.3 The Disjoint Sum of Sets .................................... 292

14 Rings 293
   14.1 Semirings and rings ............................................ 293
   14.2 Abstract divisibility .......................................... 295
   14.3 Towards integral domains ..................................... 301
   14.4 (Partial) Division .............................................. 306
   14.5 Quotient and remainder in integral domains ............. 328
   14.6 Interlude: basic tool support for algebraic and arithmetic calculations .............................................. 330
   14.7 Ordered semirings and rings ................................ 330
   14.8 Dioids ............................................................ 347
15 Natural numbers

15.1 Type \textit{ind} \hfill 347
15.2 Type \textit{nat} \hfill 348
15.3 Arithmetic operators \hfill 351
  15.3.1 Addition \hfill 353
  15.3.2 Difference \hfill 354
  15.3.3 Multiplication \hfill 354
15.4 Orders on \textit{nat} \hfill 355
  15.4.1 Operation definition \hfill 355
  15.4.2 Introduction properties \hfill 357
  15.4.3 Elimination properties \hfill 358
  15.4.4 Inductive (?) properties \hfill 358
  15.4.5 Monotonicity of Addition \hfill 362
  15.4.6 \textit{min} and \textit{max} \hfill 365
  15.4.7 Additional theorems about (\leq) \hfill 366
  15.4.8 More results about difference \hfill 372
  15.4.9 Monotonicity of multiplication \hfill 374
15.5 Natural operation of natural numbers on functions \hfill 376
15.6 Kleene iteration \hfill 378
15.7 Embedding of the naturals into any \textit{semiring-1: of-nat} \hfill 380
15.8 The set of natural numbers \hfill 384
15.9 Further arithmetic facts concerning the natural numbers \hfill 386
  15.9.1 Greatest operator \hfill 393
15.10 Monotonicity of \textit{funpow} \hfill 394
15.11 The divides relation on \textit{nat} \hfill 394
15.12 Aliasses \hfill 396
15.13 Size of a datatype value \hfill 398
15.14 Code module namespace \hfill 398

16 Fields

16.1 Division rings \hfill 399
16.2 Fields \hfill 405
16.3 Ordered fields \hfill 410

17 Finite sets

17.1 Predicate for finite sets \hfill 425
  17.1.1 Choice principles \hfill 426
  17.1.2 Finite sets are the images of initial segments of natural numbers \hfill 427
17.2 Finiteness and common set operations \hfill 428
17.3 Further induction rules on finite sets \hfill 436
17.4 Class \textit{finite} \hfill 439
17.5 A basic fold functional for finite sets \hfill 440
  17.5.1 From \textit{fold-graph} to \textit{fold} \hfill 441
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.5.2</td>
<td>Liftings to <em>comp-fun-commute</em> etc.</td>
<td>446</td>
</tr>
<tr>
<td>17.5.3</td>
<td>Expressing set operations via <em>fold</em></td>
<td>447</td>
</tr>
<tr>
<td>17.6</td>
<td>Locales as mini-packages for fold operations</td>
<td>451</td>
</tr>
<tr>
<td>17.6.1</td>
<td>The natural case</td>
<td>451</td>
</tr>
<tr>
<td>17.6.2</td>
<td>With idempotency</td>
<td>452</td>
</tr>
<tr>
<td>17.7</td>
<td>Finite cardinality</td>
<td>452</td>
</tr>
<tr>
<td>17.7.1</td>
<td>Cardinality of image</td>
<td>463</td>
</tr>
<tr>
<td>17.7.2</td>
<td>Pigeonhole Principles</td>
<td>464</td>
</tr>
<tr>
<td>17.7.3</td>
<td>Cardinality of sums</td>
<td>465</td>
</tr>
<tr>
<td>17.7.4</td>
<td>Relating injectivity and surjectivity</td>
<td>466</td>
</tr>
<tr>
<td>17.8</td>
<td>Infinite Sets</td>
<td>468</td>
</tr>
<tr>
<td>17.9</td>
<td>The finite powerset operator</td>
<td>470</td>
</tr>
<tr>
<td>18</td>
<td>Relations – as sets of pairs, and binary predicates</td>
<td>471</td>
</tr>
<tr>
<td>18.1</td>
<td>Fundamental</td>
<td>472</td>
</tr>
<tr>
<td>18.1.1</td>
<td>Relations as sets of pairs</td>
<td>472</td>
</tr>
<tr>
<td>18.1.2</td>
<td>Conversions between set and predicate relations</td>
<td>472</td>
</tr>
<tr>
<td>18.2</td>
<td>Properties of relations</td>
<td>474</td>
</tr>
<tr>
<td>18.2.1</td>
<td>Reflexivity</td>
<td>474</td>
</tr>
<tr>
<td>18.2.2</td>
<td>Irreflexivity</td>
<td>476</td>
</tr>
<tr>
<td>18.2.3</td>
<td>Asymmetry</td>
<td>476</td>
</tr>
<tr>
<td>18.2.4</td>
<td>Symmetry</td>
<td>476</td>
</tr>
<tr>
<td>18.2.5</td>
<td>Antisymmetry</td>
<td>477</td>
</tr>
<tr>
<td>18.2.6</td>
<td>Transitivity</td>
<td>478</td>
</tr>
<tr>
<td>18.2.7</td>
<td>Totality</td>
<td>480</td>
</tr>
<tr>
<td>18.2.8</td>
<td>Single valued relations</td>
<td>481</td>
</tr>
<tr>
<td>18.3</td>
<td>Relation operations</td>
<td>482</td>
</tr>
<tr>
<td>18.3.1</td>
<td>The identity relation</td>
<td>482</td>
</tr>
<tr>
<td>18.3.2</td>
<td>Diagonal: identity over a set</td>
<td>482</td>
</tr>
<tr>
<td>18.3.3</td>
<td>Composition</td>
<td>483</td>
</tr>
<tr>
<td>18.3.4</td>
<td>Converse</td>
<td>485</td>
</tr>
<tr>
<td>18.3.5</td>
<td>Domain, range and field</td>
<td>489</td>
</tr>
<tr>
<td>18.3.6</td>
<td>Image of a set under a relation</td>
<td>492</td>
</tr>
<tr>
<td>18.3.7</td>
<td>Inverse image</td>
<td>494</td>
</tr>
<tr>
<td>18.3.8</td>
<td>Powerset</td>
<td>494</td>
</tr>
<tr>
<td>18.3.9</td>
<td>Expressing relation operations via <em>Finite-Set.fold</em></td>
<td>494</td>
</tr>
<tr>
<td>19</td>
<td>Reflexive and Transitive closure of a relation</td>
<td>496</td>
</tr>
<tr>
<td>19.1</td>
<td>Reflexive closure</td>
<td>497</td>
</tr>
<tr>
<td>19.2</td>
<td>Reflexive-transitive closure</td>
<td>498</td>
</tr>
<tr>
<td>19.3</td>
<td>Transitive closure</td>
<td>503</td>
</tr>
<tr>
<td>19.4</td>
<td>Symmetric closure</td>
<td>511</td>
</tr>
<tr>
<td>19.5</td>
<td>The power operation on relations</td>
<td>511</td>
</tr>
<tr>
<td>19.6</td>
<td>Bounded transitive closure</td>
<td>520</td>
</tr>
</tbody>
</table>
19.7 Acyclic relations ............................................. 521
19.8 Setup of transitivity reasoner ............................... 522

20 Well-founded Recursion ...................................... 524
  20.1 Basic Definitions ........................................ 524
  20.2 Basic Results ........................................... 525
    20.2.1 Minimal-element characterization of well-foundedness 526
    20.2.2 Well-foundedness of transitive closure ............... 526
    20.2.3 Well-foundedness of image .......................... 529
  20.3 Well-Foundedness Results for Unions .................... 529
  20.4 Well-Foundedness of Composition ........................ 532
  20.5 Acyclic relations ....................................... 533
    20.5.1 Wellfoundedness of finite acyclic relations ......... 534
  20.6 nat is well-founded .................................... 534
  20.7 Accessible Part ......................................... 535
  20.8 Tools for building wellfounded relations .............. 538
    20.8.1 Lexicographic combinations ......................... 538
    20.8.2 Bounded increase must terminate ................. 542

21 Well-Founded Recursion Combinator ........................ 543
  21.0.1 Well-founded recursion via genuine fixpoints ....... 544
  21.1 Wellfoundedness of same-fst .......................... 544

22 Orders as Relations ........................................ 545
  22.1 Orders on a set ....................................... 545
  22.2 Orders on the field ................................... 546
  22.3 Relations given by a predicate and the field .......... 548
  22.4 Orders on a type ..................................... 548
  22.5 Order-like relations .................................. 549
    22.5.1 Auxiliaries ....................................... 549
    22.5.2 The upper and lower bounds operators ............ 550
  22.6 Variations on Well-Founded Relations ................... 553
    22.6.1 Characterizations of well-foundedness ............ 553
    22.6.2 Characterizations of well-foundedness ............ 555

23 Hilbert’s Epsilon-Operator and the Axiom of Choice ........ 557
  23.1 Hilbert’s epsilon ...................................... 557
  23.2 Hilbert’s Epsilon-operator ............................. 557
  23.3 Axiom of Choice, Proved Using the Description Operator 558
  23.4 Function Inverse ....................................... 559
  23.5 Other Consequences of Hilbert’s Epsilon ............... 568
  23.6 An aside: bounded accessible part ..................... 570
  23.7 More on injections, bijections, and inverses .......... 572
  23.8 Specification package – Hilbertized version ........... 575
23.9 Complete Distributive Lattices – Properties depending on
Hilbert Choice ............................................. 575

24 Zorn’s Lemma ........................................... 583
24.1 Zorn’s Lemma for the Subset Relation ................. 584
  24.1.1 Results that do not require an order .......... 584
  24.1.2 Hausdorff’s Maximum Principle .............. 590
  24.1.3 Results for the proper subset relation ....... 590
  24.1.4 Zorn’s lemma ..................................... 591
24.2 Zorn’s Lemma for Partial Orders ................... 592
24.3 Other variants of Zorn’s Lemma .................... 593
24.4 The Well Ordering Theorem ...................... 596

25 Well-Order Relations as Needed by Bounded Natural Func-
tors ..................................................... 602
25.1 Auxiliaries ........................................... 603
25.2 Well-founded induction and recursion adapted to non-strict
well-order relations ................................. 603
25.3 The notions of maximum, minimum, supremum, successor
and order filter ......................................... 604
  25.3.1 Properties of max2 .............................. 605
  25.3.2 Existence and uniqueness for isMinim and well-definedness
    of minim ............................................. 606
  25.3.3 Properties of minim ............................. 607
  25.3.4 Properties of successor ......................... 608
  25.3.5 Properties of order filters ..................... 610
  25.3.6 Other properties ................................. 612

26 Well-Order Embeddings as Needed by Bounded Natural Func-
tors ..................................................... 614
26.1 Auxiliaries ........................................... 614
26.2 (Well-order) embeddings, strict embeddings, isomorphisms
    and order-compatible functions ................. 615
26.3 Given any two well-orders, one can be embedded in the other 623
26.4 Uniqueness of embeddings ......................... 629
26.5 More properties of embeddings, strict embeddings and iso-
morphisms ............................................. 631

27 Constructions on Wellorders as Needed by Bounded Natural
Functors .................................................. 636
27.1 Restriction to a set ................................ 636
27.2 Order filters versus restrictions and embeddings .... 638
27.3 The strict inclusion on proper ofilters is well-founded .... 641
27.4 Ordering the well-orders by existence of embeddings .... 642
27.5 $<\circ$ is well-founded ........................................... 653
27.6 Copy via direct images ....................................... 654
27.7 Bounded square .................................................. 657

28 Cardinal-Order Relations as Needed by Bounded Natural Functors 671
28.1 Cardinal orders .................................................. 671
28.2 Cardinal of a set .................................................. 673
28.3 Cardinals versus set operations on arbitrary sets .......... 678
28.4 Cardinals versus set operations involving infinite sets .. 687
28.5 The cardinal $\omega$ and the finite cardinals ................. 694
  28.5.1 First as well-orders ........................................ 694
  28.5.2 Then as cardinals ......................................... 696
28.6 The successor of a cardinal .................................. 696
28.7 Regular cardinals .............................................. 701
28.8 Others ............................................................ 704

29 Cardinal Arithmetic as Needed by Bounded Natural Functors 705
29.1 Zero .............................................................. 706
29.2 (In)finite cardinals ............................................ 707
29.3 Binary sum ...................................................... 708
29.4 One ............................................................... 710
29.5 Two ............................................................... 710
29.6 Family sum ....................................................... 710
29.7 Product .......................................................... 711
29.8 Exponentiation .................................................. 712

30 Function Definition Base 719
31 Definition of Bounded Natural Functors 720
32 Composition of Bounded Natural Functors 726
33 Registration of Basic Types as Bounded Natural Functors 730
34 Shared Fixpoint Operations on Bounded Natural Functors 734
35 Least Fixpoint (Datatype) Operation on Bounded Natural Functors 741
36 MESON Proof Method 748
  36.1 Negation Normal Form ....................................... 748
  36.2 Pulling out the existential quantifiers ..................... 748
  36.3 Lemmas for Forward Proof .................................. 749
45 Definition of Quotient Types

45.1 Quotient Predicate

45.2 lemmas for regularisation of ball and bex

45.3 Bounded abstraction

45.4 Bex1-rel quantifier

45.5 Various respects and preserve lemmas

45.6 Quotient composition

45.7 Quotient3 to Quotient

45.8 ML setup

45.9 Methods / Interface

46 Lifting of BNFs

47 Chain-complete partial orders and their fixpoints

48 Datatype option

49 Partial Function Definitions

50 Reconstructing external resolution proofs for propositional logic

51 Function Definitions and Termination Proofs
52 The Integers as Equivalence Classes over Pairs of Natural Numbers 926

52.1 Definition of integers as a quotient type 927
52.2 Integers form a commutative ring 927
52.3 Integers are totally ordered 928
52.4 Ordering properties of arithmetic operations 929
52.5 Embedding of the Integers into any ring-1: of-int 930
52.6 Magnitude of an Integer, as a Natural Number: nat 936
52.7 Lemmas about the Function of-nat and Orderings 940
52.8 Cases and induction 941

52.8.1 Binary comparisons 943
52.8.2 Comparisons, for Ordered Rings 943
52.9 The Set of Integers 944
52.10 sum and prod 948
52.11 Setting up simplification procedures 948
52.12 More Inequality Reasoning 949
52.13 The functions nat and int 950
52.14 Induction principles for int 952
52.15 Intermediate value theorems 955
52.16 Products and 1, by T. M. Rasmussen 955
52.17 The divides relation 956
52.18 Finiteness of intervals 961
52.19 Configuration of the code generator 961
52.20 Duplicates 964

53 Big infimum (minimum) and supremum (maximum) over finite (non-empty) sets 965

53.1 Generic lattice operations over a set 965
53.1.1 Without neutral element 965
53.1.2 With neutral element 968
53.2 Lattice operations on finite sets 971
53.3 Infimum and Supremum over non-empty sets 971
53.4 Minimum and Maximum over non-empty sets 974
53.5 Arg Min 983
53.6 Arg Max 985

54 Division in euclidean (semi) rings 986

54.1 Euclidean (semi) rings with explicit division and remainder 986
54.2 Euclidean (semi) rings with cancel rules 990
54.3 Uniquely determined division 998
54.4 Euclidean division on nat 1003
54.5 Euclidean division on int 1016
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>54.6</td>
<td>Special case: euclidean rings containing the natural numbers</td>
<td>1022</td>
</tr>
<tr>
<td>54.7</td>
<td>Code generation</td>
<td>1028</td>
</tr>
<tr>
<td>55</td>
<td>Parity in rings and semirings</td>
<td>1028</td>
</tr>
<tr>
<td>55.1</td>
<td>Ring structures with parity and <code>even/odd</code> predicates</td>
<td>1028</td>
</tr>
<tr>
<td>55.2</td>
<td>Special case: euclidean rings containing the natural numbers</td>
<td>1032</td>
</tr>
<tr>
<td>55.3</td>
<td>Instance for <code>nat</code></td>
<td>1034</td>
</tr>
<tr>
<td>55.4</td>
<td>Parity and powers</td>
<td>1037</td>
</tr>
<tr>
<td>55.5</td>
<td>Instance for <code>int</code></td>
<td>1041</td>
</tr>
<tr>
<td>55.6</td>
<td>Abstract bit structures</td>
<td>1041</td>
</tr>
<tr>
<td>56</td>
<td>More on quotient and remainder</td>
<td>1060</td>
</tr>
<tr>
<td>56.1</td>
<td>More on division</td>
<td>1060</td>
</tr>
<tr>
<td>56.1.1</td>
<td>General Properties of <code>div</code> and <code>mod</code></td>
<td>1063</td>
</tr>
<tr>
<td>56.1.2</td>
<td>Laws for <code>div</code> and <code>mod</code> with Unary Minus</td>
<td>1064</td>
</tr>
<tr>
<td>56.1.3</td>
<td>Monotonicity in the First Argument (Dividend)</td>
<td>1065</td>
</tr>
<tr>
<td>56.1.4</td>
<td>Monotonicity in the Second Argument (Divisor)</td>
<td>1065</td>
</tr>
<tr>
<td>56.1.5</td>
<td>Splitting Rules for <code>div</code> and <code>mod</code></td>
<td>1067</td>
</tr>
<tr>
<td>56.1.6</td>
<td>Computing <code>div</code> and <code>mod</code> with shifting</td>
<td>1068</td>
</tr>
<tr>
<td>56.1.7</td>
<td>Quotients of Signs</td>
<td>1069</td>
</tr>
<tr>
<td>56.1.8</td>
<td>Further properties</td>
<td>1071</td>
</tr>
<tr>
<td>56.2</td>
<td>Numeral division with a pragmatic type class</td>
<td>1074</td>
</tr>
<tr>
<td>56.2.1</td>
<td>Dedicated simproc for calculation</td>
<td>1083</td>
</tr>
<tr>
<td>56.2.2</td>
<td>Code generation</td>
<td>1084</td>
</tr>
<tr>
<td>56.3</td>
<td>Lemmas of doubtful value</td>
<td>1085</td>
</tr>
<tr>
<td>57</td>
<td>Combination and Cancellation Simprocs for Numeral Expressions</td>
<td>1086</td>
</tr>
<tr>
<td>58</td>
<td>Semiring normalization</td>
<td>1092</td>
</tr>
<tr>
<td>59</td>
<td>Groebner bases</td>
<td>1096</td>
</tr>
<tr>
<td>59.1</td>
<td>Groebner Bases</td>
<td>1096</td>
</tr>
<tr>
<td>60</td>
<td>Set intervals</td>
<td>1097</td>
</tr>
<tr>
<td>60.1</td>
<td>Various equivalences</td>
<td>1099</td>
</tr>
<tr>
<td>60.2</td>
<td>Logical Equivalences for Set Inclusion and Equality</td>
<td>1100</td>
</tr>
<tr>
<td>60.3</td>
<td>Two-sided intervals</td>
<td>1101</td>
</tr>
<tr>
<td>60.3.1</td>
<td>Emptyness, singletons, subset</td>
<td>1102</td>
</tr>
<tr>
<td>60.4</td>
<td>Infinite intervals</td>
<td>1108</td>
</tr>
<tr>
<td>60.4.1</td>
<td>Intersection</td>
<td>1110</td>
</tr>
<tr>
<td>60.5</td>
<td>Intervals of natural numbers</td>
<td>1111</td>
</tr>
<tr>
<td>60.5.1</td>
<td>The Constant <code>lessThan</code></td>
<td>1111</td>
</tr>
<tr>
<td>60.5.2</td>
<td>The Constant <code>greaterThan</code></td>
<td>1112</td>
</tr>
<tr>
<td>60.5.3</td>
<td>The Constant <code>atLeast</code></td>
<td>1112</td>
</tr>
</tbody>
</table>
62.5 Integer division and modulo for Z3 ........................................ 1163
62.6 Extra theorems for veriT reconstruction .............................. 1164
62.7 Setup ................................................................................. 1166
62.8 Configuration ................................................................. 1166
62.9 General configuration options ......................................... 1167
62.10 Certificates .................................................................... 1167
62.11 Tracing ............................................................................. 1168
62.12 Schematic rules for Z3 proof reconstruction ................. 1168

63 Sledgehammer: Isabelle–ATP Linkup .............................. 1171

64 Numeric types for code generation onto target language numerals only ....................................................... 1171
64.1 Type of target language integers ................................... 1172
64.2 Code theorems for target language integers ..................... 1179
64.3 Serializer setup for target language integers .................. 1186
64.4 Type of target language naturals ...................................... 1188
64.5 Inductive representation of target language naturals .......... 1194
64.6 Code refinement for target language naturals ................... 1194

65 Setup for Lifting/Transfer for the set type .................. 1197
65.1 Relator and predicator properties ................................... 1197
65.2 Quotient theorem for the Lifting package ....................... 1198
65.3 Transfer rules for the Transfer package ......................... 1199
65.3.1 Unconditional transfer rules ....................................... 1199
65.3.2 Rules requiring bi-unique, bi-total or right-total relations ......................................................... 1200

66 The datatype of finite lists ................................................. 1203
66.1 Basic list processing functions ....................................... 1204
66.1.1 List comprehension ....................................................... 1211
66.1.2 \texttt{[]} and \texttt{(#)} ......................................................... 1219
66.1.3 \texttt{length} ................................................................ 1220
66.1.4 \texttt{@} – append .......................................................... 1223
66.1.5 \texttt{map} .................................................................. 1226
66.1.6 \texttt{rev} .................................................................... 1228
66.1.7 \texttt{set} ..................................................................... 1230
66.1.8 \texttt{filter} .................................................................. 1233
66.1.9 List partitioning .......................................................... 1236
66.1.10 \texttt{concat} ................................................................. 1237
66.1.11 (!) ........................................................................ 1238
66.1.12 \texttt{list-update} ........................................................ 1241
66.1.13 \texttt{last} and \texttt{butlast} ............................................. 1243
66.1.14 \texttt{take} and \texttt{drop} ................................................ 1245
66.6.1 Counterparts for set-related operations . . . . . . . . . 1344
66.6.2 Optimizing by rewriting . . . . . . . . . . . . . . . . . 1348
66.6.3 Pretty lists . . . . . . . . . . . . . . . . . . . . . . . . 1351
66.6.4 Use convenient predefined operations . . . . . . . . . 1352
66.6.5 Implementation of sets by lists . . . . . . . . . . . . . 1353
66.7 Setup for Lifting/Transfer . . . . . . . . . . . . . . . . . . . 1355
   66.7.1 Transfer rules for the Transfer package . . . . . . . . 1355

67 Sum and product over lists . . . . . . . . . . . . . . . . . . . . . 1360
   67.1 List summation . . . . . . . . . . . . . . . . . . . . . . . . 1361
   67.2 Further facts about List.n-lists . . . . . . . . . . . . . . . 1367
   67.3 Tools setup . . . . . . . . . . . . . . . . . . . . . . . . . . 1367
   67.4 List product . . . . . . . . . . . . . . . . . . . . . . . . . . 1368

68 A HOL random engine . . . . . . . . . . . . . . . . . . . . . . . 1369
   68.1 Auxiliary functions . . . . . . . . . . . . . . . . . . . . . . 1369
   68.2 Random seeds . . . . . . . . . . . . . . . . . . . . . . . . . . 1369
   68.3 Base selectors . . . . . . . . . . . . . . . . . . . . . . . . . . 1370
   68.4 ML interface . . . . . . . . . . . . . . . . . . . . . . . . . . . 1372

69 Maps . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1373
   69.1 empty . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1374
   69.2 map-upd . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1374
   69.3 map-of . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1375
   69.4 map-option related . . . . . . . . . . . . . . . . . . . . . . . 1378
   69.5 map-comp related . . . . . . . . . . . . . . . . . . . . . . . . 1378
   69.6 ++ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1379
   69.7 restrict-map . . . . . . . . . . . . . . . . . . . . . . . . . . . 1380
   69.8 map-ups . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1381
   69.9 dom . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1383
   69.10 ran . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1385
   69.11 map-le . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1387
   69.12 Various . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1388

70 Finite types as explicit enumerations . . . . . . . . . . . . . . . . 1390
   70.1 Class enum . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1390
   70.2 Implementations using enum . . . . . . . . . . . . . . . . . . . 1391
      70.2.1 Unbounded operations and quantifiers . . . . . . . . . . 1391
      70.2.2 An executable choice operator . . . . . . . . . . . . . . . 1391
      70.2.3 Equality and order on functions . . . . . . . . . . . . . . 1392
      70.2.4 Operations on relations . . . . . . . . . . . . . . . . . . . . 1393
      70.2.5 Bounded accessible part . . . . . . . . . . . . . . . . . . . . 1393
   70.3 Default instances for enum . . . . . . . . . . . . . . . . . . . . 1395
   70.4 Small finite types . . . . . . . . . . . . . . . . . . . . . . . . . . 1400
70.5 Closing up ........................................... 1414

71 Character and string types .......................... 1414
  71.1 Strings as list of bytes ............................ 1414
    71.1.1 Bytes as datatype ............................. 1414
  71.2 Strings as dedicated type for target language code generation 1423
    71.2.1 Logical specification .......................... 1423
    71.2.2 Syntactic representation ...................... 1424
    71.2.3 Operations ................................. 1425
    71.2.4 Executable conversions ....................... 1427
    71.2.5 Technical code generation setup ............. 1428
    71.2.6 Code generation utility ...................... 1431
    71.2.7 Finally ......................................... 1431

72 Reflecting Pure types into HOL ..................... 1431

73 Predicates as enumerations ........................... 1433
  73.1 The type of predicate enumerations (a monad) ........ 1433
  73.2 Emptiness check and definite choice ............... 1437
  73.3 Derived operations ............................... 1440
  73.4 Implementation .................................. 1442

74 Lazy sequences ...................................... 1448
  74.1 Type of lazy sequences ............................ 1448
  74.2 Code setup .................................... 1452
  74.3 Generator Sequences ................................ 1453
    74.3.1 General lazy sequence operation ............ 1453
    74.3.2 Small lazy typeclasses ...................... 1453
  74.4 With Hit Bound Value ............................. 1454

75 Depth-Limited Sequences with failure element ....... 1455
  75.1 Depth-Limited Sequence ........................... 1455
  75.2 Positive Depth-Limited Sequence .................. 1457
  75.3 Negative Depth-Limited Sequence .................. 1458
  75.4 Negation ........................................ 1459

76 Term evaluation using the generic code generator ..... 1460
  76.1 Term representation ............................... 1460
    76.1.1 Terms and class term-of ..................... 1460
    76.1.2 Syntax .................................... 1461
  76.2 Tools setup and evaluation ....................... 1461
  76.3 Dedicated term-of instances ..................... 1462
  76.4 Generic reification ............................... 1463
  76.5 Diagnostic ....................................... 1463
77 A simple counterexample generator performing random testing

77.1 Catching Match exceptions
77.2 The random class
77.3 Fundamental and numeric types
77.4 Complex generators
77.5 Deriving random generators for datatypes
77.6 Code setup

78 The Random-Predicate Monad

79 Various kind of sequences inside the random monad

80 A simple counterexample generator performing exhaustive testing

80.1 Basic operations for exhaustive generators
80.2 Exhaustive generator type classes
  80.2.1 A smarter enumeration scheme for functions over finite datatypes
80.3 Bounded universal quantifiers
80.4 Fast exhaustive combinators
80.5 Continuation passing style functions as plus monad
80.6 Defining generators for any first-order data type
80.7 Defining generators for abstract types

81 A compiler for predicates defined by introduction rules

81.1 Set membership as a generator predicate

82 Counterexample generator performing narrowing-based testing

82.1 Counterexample generator
  82.1.1 Code generation setup
  82.1.2 Narrowing’s deep representation of types and terms
  82.1.3 From narrowing’s deep representation of terms to HOL.Code-Evaluation’s terms
  82.1.4 Auxiliary functions for Narrowing
  82.1.5 Narrowing’s basic operations
  82.1.6 Narrowing generator type class
  82.1.7 class is-testable
  82.1.8 Defining a simple datatype to represent functions in an incomplete and redundant way
  82.1.9 Setting up the counterexample generator
82.2 Narrowing for sets
82.3 Narrowing for integers
88.2.5 Order filters ........................................ 1600
88.3 Sequentially ........................................... 1602
88.4 Increasing finite subsets ............................... 1603
88.5 The cofinite filter ..................................... 1604
  88.5.1 Product of filters ................................ 1605
88.6 Limits .................................................. 1610
88.7 Limits to at-top and at-bot ............................ 1613
88.8 Setup 'a filter for lifting and transfer ............... 1616

89 Conditionally-complete Lattices .......................... 1627

90 Factorial Function, Rising Factorials .................. 1643
  90.1 Factorial Function .................................. 1643
  90.2 Pochhammer’s symbol: generalized rising factorial . 1647
  90.3 Misc .................................................. 1652

91 Binomial Coefficients and Binomial Theorem .......... 1652
  91.1 Binomial coefficients ............................... 1653
  91.2 The binomial theorem (courtesy of Tobias Nipkow): .. 1656
  91.3 Generalized binomial coefficients .................... 1659
    91.3.1 Summation on the upper index .................. 1670
  91.4 More on Binomial Coefficients .................... 1676
  91.5 Executable code .................................... 1681

92 Main HOL .................................................. 1682

93 Archimedean Fields, Floor and Ceiling Functions ...... 1683
  93.1 Class of Archimedean fields ....................... 1685
  93.2 Existence and uniqueness of floor function ........ 1686
  93.3 Floor function .................................... 1687
  93.4 Ceiling function .................................. 1693
    93.4.1 Ceiling with numerals ......................... 1694
    93.4.2 Addition and subtraction of integers .......... 1695
  93.5 Negation ........................................... 1697
  93.6 Natural numbers .................................... 1697
  93.7 Frac Function ..................................... 1697
  93.8 Rounding to the nearest integer ................... 1699

94 Rational numbers ......................................... 1700
  94.1 Rational numbers as quotient ...................... 1701
    94.1.1 Construction of the type of rational numbers ... 1701
    94.1.2 Representation and basic operations .......... 1701
    94.1.3 Function normalize ........................... 1706
    94.1.4 Various ........................................ 1709
    94.1.5 The ordered field of rational numbers ........ 1709
94.1.6 Rationals are an Archimedean field ........................................ 1713
94.2 Linear arithmetic setup ............................................................ 1714
94.3 Embedding from Rationals to other Fields .................................... 1714
94.4 The Set of Rational Numbers ................................................... 1717
94.5 Implementation of rational numbers as pairs of integers ................. 1719
94.6 Setup for Nitpick ................................................................. 1723
94.7 Float syntax .............................................................................. 1724
94.8 Hiding implementation details .................................................... 1724

95 Development of the Reals using Cauchy Sequences .......................... 1725
95.1 Preliminary lemmas ..................................................................... 1725
95.2 Sequences that converge to zero ................................................ 1726
95.3 Cauchy sequences ...................................................................... 1727
95.4 Equivalence relation on Cauchy sequences ....................................... 1732
95.5 The field of real numbers .......................................................... 1732
95.6 Positive reals ............................................................................. 1735
95.7 Completeness ............................................................................. 1738
95.8 Supremum of a set of reals .......................................................... 1743
95.9 Hiding implementation details .................................................... 1744
95.10 More Lemmas ........................................................................... 1744
95.11 Embedding numbers into the Reals ............................................. 1744
95.12 Embedding the Naturals into the Reals ......................................... 1746
95.13 The Archimedean Property of the Reals ....................................... 1747
95.14 Rationals ................................................................................... 1747
95.15 Density of the Rational Reals in the Reals .................................... 1750
95.16 Numerals and Arithmetic .......................................................... 1750
95.17 Simprules combining $x + y$ and $0$ .............................................. 1750
95.18 Lemmas about powers ............................................................... 1751
95.19 Density of the Reals ................................................................... 1751
95.20 Archimedean properties and useful consequences .......................... 1751
95.21 Floor and Ceiling Functions from the Reals to the Integers ............ 1753
95.22 Exponentiation with floor ......................................................... 1756
95.23 Implementation of rational real numbers ..................................... 1757
95.24 Setup for Nitpick ...................................................................... 1759
95.25 Setup for SMT .......................................................................... 1760
95.26 Setup for Argo .......................................................................... 1760

96 Topological Spaces ........................................................................ 1760
96.1 Topological space ........................................................................ 1760
96.2 Hausdorff and other separation properties .................................... 1763
96.3 Generators for topologies ............................................................ 1765
96.4 Order topologies ......................................................................... 1765
96.5 Setup some topologies ............................................................... 1768
96.5.1 Boolean is an order topology .................................................. 1768
96.5.2 Topological filters ............................................. 1770
96.5.3 Tendsto ........................................................... 1775
96.5.4 Rules about Lim .................................................... 1782
96.6 Limits on sequences .................................................. 1784
96.7 Monotone sequences and subsequences .................................. 1784
  96.7.1 Definition of subsequence. ........................................ 1785
  96.7.2 Subsequence (alternative definition, (e.g. Hoskins) ........ 1786
  96.7.3 Increasing and Decreasing Series ............................... 1792
96.8 First countable topologies ........................................... 1792
96.9 Function limit at a point ............................................ 1796
  96.9.1 Relation of LIM and LIMSEQ .................................... 1798
96.10 Continuity ............................................................ 1801
  96.10.1 Continuity on a set ............................................. 1801
  96.10.2 Continuity at a point ......................................... 1806
  96.10.3 Open-cover compactness ...................................... 1810
96.11 Finite intersection property ..................................... 1812
96.12 Connectedness ....................................................... 1816

97 Linear Continuum Topologies ........................................ 1821
  97.1 Intermediate Value Theorem ..................................... 1824
97.2 Uniform spaces ....................................................... 1829
  97.2.1 Totally bounded sets ........................................... 1830
  97.2.2 Cauchy filter ................................................... 1831
  97.2.3 Uniformly continuous functions ............................... 1833

98 Product Topology ...................................................... 1834
  98.1 Product is a topological space .................................. 1834
  98.1.1 Continuity of operations ..................................... 1836
  98.1.2 Connectedness of products ................................... 1838
  98.1.3 Separation axioms .............................................. 1839
  98.2 A generic notion of the convex, affine, conic hull, or closed
        "hull".................................................................. 1841

99 Modules ................................................................. 1843
  99.1 Locale for additive functions ................................... 1843

100 Subspace .............................................................. 1845

101 Span: subspace generated by a set .................................. 1846

102 Dependent and independent sets .................................... 1850

103 Representation of a vector on a specific basis .................... 1853

104 Vector Spaces ........................................................ 1863
105 Vector Spaces and Algebras over the Reals 1899
105.1 Real vector spaces .......................... 1899
105.2 Embedding of the Reals into any real-algebra-1: of-real 1904
105.3 The Set of Real Numbers .......................... 1906
105.4 Ordered real vector spaces ...................... 1908
105.5 Real normed vector spaces ....................... 1913
105.6 Metric spaces .................................. 1921
105.7 Class instances for real numbers ................. 1925
105.8 Extra type constraints ........................... 1926
105.9 Sign function .................................. 1927
105.10 Bounded Linear and Bilinear Operators ......... 1928
105.11 Filters and Limits on Metric Space ............... 1935
105.11.1 Limits of Sequences ......................... 1937
105.11.2 Limits of Functions .......................... 1938
105.12 Complete metric spaces ......................... 1939
105.13 Cauchy sequences .............................. 1939
105.13.1 Cauchy Sequences are Convergent ............ 1944
105.14 The set of real numbers is a complete metric space 1945

106 Limits on Real Vector Spaces 1948
106.1 Filter going to infinity norm ............... 1948
106.1.1 Boundedness ................................ 1949
106.1.2 Bounded Sequences .......................... 1950
106.1.3 A Few More Equivalence Theorems for Boundedness 1952
106.1.4 Upper Bounds and Lubs of Bounded Sequences ...... 1953
106.1.5 Polynomial function extremal theorem, from HOL Light 1955
106.2 Convergence to Zero ............................ 1957
106.2.1 Distance and norms ........................... 1960
106.3 Topological Monoid ............................. 1962
106.3.1 Topological group ............................ 1963
106.3.2 Linear operators and multiplication .......... 1965
106.3.3 Inverse and division .......................... 1971
106.4 Relate at, at-left and at-right ............... 1978
106.5 Floor and Ceiling .............................. 1990
106.6 Limits of Sequences ............................. 1991
106.7 Convergence on sequences ...................... 1995
106.8 Power Sequences ............................... 2000
106.9 Limits of Functions ............................. 2001
106.10 Continuity .................................. 2003
106.11 Uniform Continuity ............................ 2005
106.12 Nested Intervals and Bisection – Needed for Compactness 2006
106.13 Boundedness of continuous functions .......... 2009
### 10 Infinite Series

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>107.1</td>
<td>Definition of infinite summability</td>
<td>2013</td>
</tr>
<tr>
<td>107.2</td>
<td>Infinite summability on topological monoids</td>
<td>2014</td>
</tr>
<tr>
<td>107.3</td>
<td>Infinite summability on ordered, topological monoids</td>
<td>2017</td>
</tr>
<tr>
<td>107.4</td>
<td>Infinite summability on topological monoids</td>
<td>2019</td>
</tr>
<tr>
<td>107.5</td>
<td>Infinite summability on real normed vector spaces</td>
<td>2020</td>
</tr>
<tr>
<td>107.6</td>
<td>Infinite summability on real normed algebras</td>
<td>2024</td>
</tr>
<tr>
<td>107.7</td>
<td>Infinite summability on real normed fields</td>
<td>2024</td>
</tr>
<tr>
<td>107.8</td>
<td>Telescoping</td>
<td>2026</td>
</tr>
<tr>
<td>107.9</td>
<td>Infinite summability on Banach spaces</td>
<td>2027</td>
</tr>
<tr>
<td>107.10</td>
<td>The Ratio Test</td>
<td>2029</td>
</tr>
<tr>
<td>107.11</td>
<td>Cauchy Product Formula</td>
<td>2031</td>
</tr>
<tr>
<td>107.12</td>
<td>Series on reals</td>
<td>2033</td>
</tr>
</tbody>
</table>

### 10.8 Differentiation

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>108.1</td>
<td>Frechet derivative</td>
<td>2040</td>
</tr>
<tr>
<td>108.1.1</td>
<td>Limit transformation for derivatives</td>
<td>2045</td>
</tr>
<tr>
<td>108.2</td>
<td>Continuity</td>
<td>2046</td>
</tr>
<tr>
<td>108.3</td>
<td>Composition</td>
<td>2047</td>
</tr>
<tr>
<td>108.4</td>
<td>Uniqueness</td>
<td>2052</td>
</tr>
<tr>
<td>108.5</td>
<td>Differentiability predicate</td>
<td>2053</td>
</tr>
<tr>
<td>108.6</td>
<td>Vector derivative</td>
<td>2056</td>
</tr>
<tr>
<td>108.7</td>
<td>Derivatives</td>
<td>2059</td>
</tr>
<tr>
<td>108.8</td>
<td>Local extrema</td>
<td>2065</td>
</tr>
<tr>
<td>108.9</td>
<td>Rolle's Theorem</td>
<td>2068</td>
</tr>
<tr>
<td>108.10</td>
<td>Mean Value Theorem</td>
<td>2071</td>
</tr>
<tr>
<td>108.10.1</td>
<td>A function is constant if its derivative is 0 over an interval.</td>
<td>2073</td>
</tr>
<tr>
<td>108.10.2</td>
<td>A function with positive derivative is increasing</td>
<td>2075</td>
</tr>
<tr>
<td>108.11</td>
<td>Generalized Mean Value Theorem</td>
<td>2079</td>
</tr>
<tr>
<td>108.12</td>
<td>Hopitals rule</td>
<td>2081</td>
</tr>
</tbody>
</table>

### 10.9 Nth Roots of Real Numbers

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>109.1</td>
<td>Existence of Nth Root</td>
<td>2089</td>
</tr>
<tr>
<td>109.2</td>
<td>Nth Root</td>
<td>2090</td>
</tr>
<tr>
<td>109.3</td>
<td>Square Root</td>
<td>2097</td>
</tr>
<tr>
<td>109.4</td>
<td>Square Root of Sum of Squares</td>
<td>2101</td>
</tr>
</tbody>
</table>

### 10.10 Power Series, Transcendental Functions etc.

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>110.1</td>
<td>More facts about binomial coefficients</td>
<td>2107</td>
</tr>
<tr>
<td>110.2</td>
<td>Properties of Power Series</td>
<td>2110</td>
</tr>
<tr>
<td>110.3</td>
<td>Alternating series test / Leibniz formula</td>
<td>2114</td>
</tr>
<tr>
<td>110.4</td>
<td>Term-by-Term Differentiability of Power Series</td>
<td>2118</td>
</tr>
</tbody>
</table>
11.0.5 The Derivative of a Power Series Has the Same Radius of
Convergence ........................................... 2123
11.0.6 Derivability of power series .......................... 2127
11.0.7 Exponential Function ............................... 2133
  11.0.7.1 Properties of the Exponential Function ........ 2135
  11.0.7.2 Properties of the Exponential Function on Reals 2138
11.0.8 Natural Logarithm .................................. 2141
  11.0.8.1 A couple of simple bounds ...................... 2155
11.0.9 The general logarithm ............................... 2156
11.0.10 Sine and Cosine .................................. 2172
11.0.11 Properties of Sine and Cosine ...................... 2176
11.0.12 Deriving the Addition Formulas .................... 2177
11.0.13 The Constant Pi .................................. 2182
11.0.14 More Corollaries about Sine and Cosine ........... 2196
11.0.15 Tangent ........................................... 2200
11.0.16 Cotangent ......................................... 2208
11.0.17 Inverse Trigonometric Functions ................... 2209
11.0.18 Prove Totality of the Trigonometric Functions .. 2217
11.0.19 Machin’s formula ................................. 2221
11.0.20 Introducing the inverse tangent power series .... 2223
11.0.21 Basics about polynomial functions: products, extremal
behaviour and root counts ............................ 2234
11.0.22 Hyperbolic functions ............................... 2240
  11.0.23.1 More specific properties of the real functions 2245
  11.0.23.2 Limits ........................................ 2250
  11.0.23.3 Properties of the inverse hyperbolic functions 2251
11.0.24 Improvis for root and power literals ............... 2257

11.1 Complex Numbers: Rectangular and Polar Representation 2261
11.1.1 Addition and Subtraction ........................... 2262
11.1.2 Multiplication and Division ........................ 2262
11.1.3 Scalar Multiplication ............................. 2263
11.1.4 Numerals, Arithmetic, and Embedding from R ....... 2264
11.1.5 The Complex Number \( i \) .......................... 2266
11.1.6 Vector Norm ....................................... 2267
11.1.7 Absolute value .................................... 2270
11.1.8 Completeness of the Complexes .................... 2270
11.1.9 Complex Conjugation .............................. 2272
11.1.10 Basic Lemmas .................................... 2275
11.1.11 Polar Form for Complex Numbers .................. 2277
  11.1.11.1 \( \cos \theta + i \sin \theta \) ....................... 2277
  11.1.11.2 \( 2(\cos \theta + i \sin \theta) \) ..................... 2279
  11.1.11.3 Complex exponential .......................... 2279
11.11. Complex argument ........................................ 2281
11.12. Complex n-th roots ........................................ 2282
11.13. Square root of complex numbers ......................... 2286

11.2. MacLaurin and Taylor Series .......................... 2288
  11.2.1. Maclaurin’s Theorem with Lagrange Form of Remainder ... 2289
  11.2.2. More Convenient ”Bidirectional” Version .................. 2293
  11.2.3. Version for Exponential Function ......................... 2295
  11.2.4. Version for Sine Function ................................ 2296
  11.2.5. Maclaurin Expansion for Cosine Function ............... 2297

11.3. Taylor series ............................................ 2300

11.4. Comprehensive Complex Theory .......................... 2303
1 The basis of Higher-Order Logic

theory HOL
imports Pure Tools.Code-Generator
keywords
  try solve-direct quickcheck print-coercions print-claset
  print-induct-rules :: diag and
  quickcheck-params :: thy-decl
begin

ML-file ⟨~/src/Tools/misc-legacy.ml⟩
ML-file ⟨~/src/Tools/try.ml⟩
ML-file ⟨~/src/Tools/quickcheck.ml⟩
ML-file ⟨~/src/Tools/solve-direct.ml⟩
ML-file ⟨~/src/Tools/IsaPlanner/zipper.ml⟩
ML-file ⟨~/src/Tools/IsaPlanner/isand.ml⟩
ML-file ⟨~/src/Tools/IsaPlanner/rw-inst.ml⟩
ML-file ⟨~/src/Provers/hypsubst.ml⟩
ML-file ⟨~/src/Provers/splitter.ml⟩
ML-file ⟨~/src/Provers/classical.ml⟩
ML-file ⟨~/src/Provers/blast.ml⟩
ML-file ⟨~/src/Tools/eqsubst.ml⟩
ML-file ⟨~/src/Tools/quantifier1.ml⟩
ML-file ⟨~/src/Tools/atomize-elim.ml⟩
ML-file ⟨~/src/Tools/cong-tac.ml⟩
ML-file ⟨~/src/Tools/intuitionistic.ml⟩ setup ⟨Intuitionistic.method-setup binding iprover⟩
ML-file ⟨~/src/Tools/project-rule.ml⟩
ML-file ⟨~/src/Tools/subtyping.ml⟩
ML-file ⟨~/src/Tools/case-product.ml⟩

ML ⟨Plugin-Name.declare-setup binding (extraction)⟩

ML ⟨
  Plugin-Name.declare-setup binding (quickcheck-random);
  Plugin-Name.declare-setup binding (quickcheck-exhaustive);
  Plugin-Name.declare-setup binding (quickcheck-bounded-forall);
  Plugin-Name.declare-setup binding (quickcheck-full-exhaustive);
  Plugin-Name.declare-setup binding (quickcheck-narrowing);
⟩

ML ⟨
  Plugin-Name.define-setup binding (quickcheck)
  [plugin (quickcheck-exhaustive),
   plugin (quickcheck-random),
   plugin (quickcheck-bounded-forall),
   plugin (quickcheck-full-exhaustive),
   plugin (quickcheck-narrowing)]
⟩
1.1 Primitive logic

The definition of the logic is based on Mike Gordon’s technical report [2] that describes the first implementation of HOL. However, there are a number of differences. In particular, we start with the definite description operator and introduce Hilbert’s $\varepsilon$ operator only much later. Moreover, axiom $(P \rightarrow Q) \rightarrow (Q \rightarrow P) \rightarrow (P = Q)$ is derived from the other axioms. The fact that this axiom is derivable was first noticed by Bruno Barras (for Mike Gordon’s line of HOL systems) and later independently by Alexander Maletzky (for Isabelle/HOL).

1.1.1 Core syntax

```plaintext
setup : Axclass.class axiomatization (binding (type, []));
default-sort type
setup : Object-Logic.add-base-sort sort (type);

setup : Proofterm.set-preproc (Proof-Rewrite-Rules.standard-preproc []);

axiomatization where fun-arity: OFCLASS('a ⇒ 'b, type-class)
instance fun :: (type, type) type by (rule fun-arity)

axiomatization where itself-arity: OFCLASS('a itself, type-class)
instance itself :: (type) type by (rule itself-arity)

typedecl bool

judgment Trueprop :: bool ⇒ prop ((·) 5)

axiomatization implies :: [bool, bool] ⇒ bool (infixr ⇒ 25)
  and eq :: ['a, 'a] ⇒ bool
  and The :: ('a ⇒ bool) ⇒ 'a

notation (input)
eq (infixl = 50)
notation (output)
eq (infix = 50)
```

The input syntax for $eq$ is more permissive than the output syntax because of the large amount of material that relies on infixl.

1.1.2 Defined connectives and quantifiers

```plaintext
definition True :: bool
  where True ≡ ((\(\lambda x::bool. x) = (\lambda x. x))
```
THEORY "HOL"

definition All :: ('a ⇒ bool) ⇒ bool (binder ∀ 10)
  where All P ≡ (P = (λx. True))

definition Ex :: ('a ⇒ bool) ⇒ bool (binder ∃ 10)
  where Ex P ≡ ∀ Q. (∀ x. P x → Q) → Q

definition False :: bool
  where False ≡ (∀ P. P)

definition Not :: bool ⇒ bool (¬ 40)
  where not-def: ¬ P ≡ P → False

definition conj :: [bool, bool] ⇒ bool (infixr ∧ 35)
  where and-def: P ∧ Q ≡ ∀ R. (P → Q → R) → R

definition disj :: [bool, bool] ⇒ bool (infixr ∨ 30)
  where or-def: P ∨ Q ≡ ∀ R. (P → R) → (Q → R) → R

definition Ex1 :: ('a ⇒ bool) ⇒ bool
  where Ex1 P ≡ ∃ x. P x ∧ (∀ y. P y → y = x)

1.1.3 Additional concrete syntax

syntax (ASCII)
  -Ex1 :: pttrn ⇒ bool ⇒ bool ((∃ EX! -/ -) [0, 10] 10)

syntax (input)
  -Ex1 :: pttrn ⇒ bool ⇒ bool ((∃ !-/-) [0, 10] 10)

translations \∃ x. P ⇔ CONST Ex1 (λx. P)

print-translation
  [Syntax-Trans.preserve-binder-abs-tr' const-syntax (Ex1) syntax-const (-Ex1)]
  \( → \) to avoid eta-contraction of body

syntax
  -Not-Ex :: idts ⇒ bool ⇒ bool ((¬ -) [0, 10] 10)
  -Not-Ex1 :: pttrn ⇒ bool ⇒ bool ((¬ !-/-) [0, 10] 10)

translations
  \¬ x. P ⇔ (\∥ x. P)
  \¬ !x. P ⇔ (\∃! x. P)

abbreviation not-equal :: ['a, 'a] ⇒ bool (infix ≠ 50)
  where x ≠ y ≡ ¬ (x = y)

notation (ASCII)
  Not (~ - [40] 40) and
  conj (infixr & 35) and
THEORY "HOL"

disj (infixr | 30) and
implies (infixr \rightarrow 25) and
not-equal (infix \sim 50)

abbreviation (iff)
iff :: [bool, bool] \Rightarrow bool (infixr \leftrightarrow 25)
where A \leftrightarrow B \equiv A = B

syntax -The :: [pttrn, bool] \Rightarrow 'a ((\lambda x. x) [0, 10] 10)
translations THE x. P \Leftarrow FONT The (\lambda x. P)
print-translation
\langle \langle \text{const-syntax} \langle \text{The} \rangle, \text{fn - }\Rightarrow \text{fn Abs abs} \rangle \Rightarrow
let val (x, t) = Syntax-Trans.atomic-abs-tr' abs
in Syntax.const \text{syntax-const} \langle \text{The} \rangle $x$ $t$ end\rangle
\rangle
— To avoid eta-contraction of body

nonterminal letbinds and letbind
syntax
 -bind :: [pttrn, 'a] \Rightarrow letbind ((\lambda -/- -) 10)
   :: letbind \Rightarrow letbinds (-)
 -binds :: [letbind, letbinds] \Rightarrow letbinds (/-/-)
 -Let :: [letbinds, 'a] \Rightarrow 'a ((let -/- in -)) [0, 10] 10)

nonterminal case-syn and cases-syn
syntax
 -case-syntax :: ['a, cases-syn] \Rightarrow 'b ((case - of/- -) 10)
 -case1 :: ['a, 'b] \Rightarrow case-syn ((\lambda -/-) 10)
   :: case-syn \Rightarrow cases-syn (-)
 -case2 :: [case-syn, cases-syn] \Rightarrow cases-syn (-/-)

syntax (ASCII)
 -case1 :: ['a, 'b] \Rightarrow case-syn ((\lambda -/-) 10)

notation (ASCII)
 All (binder ALL 10) and
 Ex (binder EX 10)

notation (input)
 All (binder ! 10) and
 Ex (binder ? 10)

1.1.4 Axioms and basic definitions

axiomatization where
refl: t = (t::'a) and
subst: s = t \Rightarrow P s \Rightarrow P t and
ext: (\forall x::'a. (f x ::'b = g x) \Rightarrow (\lambda x. f x) = (\lambda x. g x))
— Extensionality is built into the meta-logic, and this rule expresses a related property. It is an eta-expanded version of the traditional rule, and similar to the ABS rule of HOL and
the-eq-trivial: \((\text{T}HE \ x. \ x = a) = (\text{a}::\text{a})\)

axiomatization where
impl: \((P \implies Q) \implies P \implies Q\)
mp: \([P \implies Q; P] \implies Q\)

True-or-False: \((P = \text{True}) \lor (P = \text{False})\)

definition \(\text{If} :: \text{bool} \Rightarrow \text{a} \Rightarrow \text{a} \Rightarrow \text{a} ((\text{if} \ (-)/ \text{then} \ (-)/ \text{else} \ (-)) \ [0, 0, 10] \ 10)\)
where \(\text{If} \ P \ x \ y \equiv (\text{T}HE \ z::\text{a}. \ (P = \text{True} \implies z = x) \land (P = \text{False} \implies z = y))\)

definition \(\text{Let} :: \text{a} \Rightarrow (\text{a} \Rightarrow \text{b}) \Rightarrow \text{b}\)
where \(\text{Let} \ s \ f \equiv f \ s\)

translations
- \(\text{-Let} \ (-\text{binds} \ b \ \text{bs}) \ e \equiv \text{-Let} b \ (-\text{Let} \ \text{bs} \ e)\)
- \(\text{let} \ x = a \ \text{in} \ e \equiv \text{CONST Let} \ a \ (\lambda x. \ e)\)

axiomatization \(\text{undefined} :: \text{a}\)

class \(\text{default} = \text{fixes} \ Default :: \text{a}\)

1.2 Fundamental rules

1.2.1 Equality

lemma \(\text{sym}:: s = t \implies t = s\)
by (erule subst) (rule refl)

lemma \(\text{ssubst}:: t = s \implies P s \implies P t\)
by (drule sym) (erule subst)

lemma \(\text{trans}:: [r = s; s = t] \implies r = t\)
by (erule subst)

lemma \(\text{trans-sym}:: [\text{Pure.elim?}]: r = s \implies t = s \implies r = t\)
by (rule trans [OF - sym])

lemma \(\text{meta-eq-to-obj-eq}::\)
assumes \(A \equiv B\)
shows \(A = B\)
unfolding \text{assms} by (rule refl)

Useful with \text{erule} for proving equalities from known equalities.

lemma \(\text{box-equals}:: [a = b; a = c; b = d] \implies c = d\)
apply (rule trans)
apply (rule trans)
apply (rule sym)
apply assumption+
done

For calculational reasoning:

**lemma** forw-subst: \( a = b \implies P \ b \implies P \ a \)
by (rule ssubst)

**lemma** back-subst: \( P \ a \implies a = b \implies P \ b \)
by (rule subst)

### 1.2.2 Congruence rules for application

Similar to \textit{AP-THM} in Gordon’s HOL.

**lemma** fun-cong: \( (f :: \; \texttt{‘}a \Rightarrow \texttt{‘}b) = g \implies f \ x = g \ x \)
apply (erule subst)
apply (rule refl)
done

Similar to \textit{AP-TERM} in Gordon’s HOL and FOL’s \textit{subst-context}.

**lemma** arg-cong: \( x = y \implies f \ x = f \ y \)
apply (erule subst)
apply (rule refl)
done

**lemma** arg-cong2: \([\ a = b; \ c = d\] \implies f \ a \ c = f \ b \ d\)
apply (erule ssubst)+
apply (rule refl)
done

**lemma** cong: \([f = g; \ (x::\texttt{‘}a) = y]\) \implies f \ x = g \ y
apply (erule subst)+
apply (rule refl)
done

ML \( \texttt{fun cong-tac ctxt = Cong-Tac.cong-tac ctxt @\{thm cong\}}\)

### 1.2.3 Equality of booleans – iff

**lemma** iffD2: \( [P = Q; \ Q] \implies P \)
by (erule subst)

**lemma** rev-iffD2: \([Q; \ P = Q]\) \implies P
by (erule iffD2)

**lemma** iffD1: \( Q = P \implies Q \implies P \)
by (drule sym) (rule iffD2)

**lemma** rev-iffD1: \( Q \implies Q = P \implies P \)
by (drule sym) (rule rev-iffD2)
THEORY "HOL"

lemma `iffE`:
  assumes major: \( P = Q \)
  and minor: \([P \rightarrow Q; Q \rightarrow P] \Rightarrow R\)
  shows R
  by (iprover intro: minor impI major [THEN iffD2] major [THEN iffD1])

1.2.4 True (1)

lemma `TrueI`:
  unfolding True-def by (rule refl)

lemma `eqTrueE`:
  \( P = \text{True} \Rightarrow P \)
  by (erule iffD2) (rule TrueI)

1.2.5 Universal quantifier (1)

lemma `spec`:
  \( \forall x ::'a. P x \Rightarrow P x \)
  apply (unfold All-def)
  apply (rule eqTrueE)
  apply (erule fun-cong)
  done

lemma `allE`:
  assumes major: \( \forall x. P x \)
  and minor: \( P x \Rightarrow R \)
  shows R
  by (iprover intro: minor major [THEN spec])

lemma `all-dupE`:
  assumes major: \( \forall x. P x \)
  and minor: \( [P x; \forall x. P x] \Rightarrow R \)
  shows R
  by (iprover intro: minor major major [THEN spec])

1.2.6 False

Depends upon `spec`; it is impossible to do propositional logic before quantifiers!

lemma `FalseE`:
  \( False \Rightarrow P \)
  apply (unfold False-def)
  apply (erule spec)
  done

lemma `False-neq-True`:
  \( False = True \Rightarrow P \)
  by (erule eqTrueE [THEN FalseE])
1.2.7 Negation

**lemma notI**:  
*assumes* \( P \Rightarrow \text{False} \)  
*shows* \( \neg P \)  
*apply* (unfold not-def)  
*apply* (iprover intro: impI assms)  
done

**lemma False-not-True**: \( \text{False} \neq \text{True} \)  
*apply* (rule notI)  
*apply* (erule False-neq-True)  
done

**lemma True-not-False**: \( \text{True} \neq \text{False} \)  
*apply* (rule notI)  
*apply* (drule sym)  
*apply* (erule False-neq-True)  
done

**lemma notE**: \[ \neg P ; P \] = \( \Rightarrow R \)  
*apply* (unfold not-def)  
*apply* (erule mp [\( \text{THEN} \) FalseE])  
*apply* assumption  
done

**lemma notI2**:  
\((P \Rightarrow \neg P) \Rightarrow (P \Rightarrow P) \Rightarrow \neg P\)  
*by* (erule notE [\( \text{THEN} \) notI]) (erule meta-mp)

1.2.8 Implication

**lemma impE**:  
*assumes* \( P \Rightarrow Q \) \( Q \Rightarrow R \)  
*shows* \( R \)  
*by* (iprover intro: assms mp)

Reduces \( Q \) to \( P \Rightarrow Q \), allowing substitution in \( P \).

**lemma rev-mp**: \[ [P ; P \Rightarrow Q] \Rightarrow Q \]  
*by* (iprover intro: mp)

**lemma contrapos-nn**:  
*assumes* major: \( \neg Q \)  
*and minor:* \( P \Rightarrow Q \)  
*shows* \( \neg P \)  
*by* (iprover intro: notI minor major [\( \text{THEN} \) notE])

Not used at all, but we already have the other 3 combinations.

**lemma contrapos-pn**:  
*assumes* major: \( Q \)  
*and minor:* \( P \Rightarrow \neg Q \)
shows $\neg P$
by (iprover intro: notI minor major notE)

**lemma** not-sym: $t \neq s \implies s \neq t$
by (erule contrapos-nn) (erule sym)

**lemma** eq-neq-eq-imp-neq: $[x = a; a \neq b; b = y] \implies x \neq y$
by (erule subst, erule ssubst, assumption)

### 1.2.9 Disjunction (1)

**lemma** disjE:
assumes major: $P \lor Q$
and minorP: $P \implies R$
and minorQ: $Q \implies R$
shows $R$
by (iprover intro: minorP minorQ impI
major [unfolded or-def, THEN spec, THEN mp, THEN mp])

### 1.2.10 Derivation of iffI

In an intuitionistic version of HOL iffI needs to be an axiom.

**lemma** iffI:
assumes $P \implies Q$ and $Q \implies P$
shows $P = Q$
proof (rule disjE[of True-or-False[of P]])
assume 1: $P = True$
note Q = assms(1)[OF eqTrueE[OF this]]
from 1 show ?thesis
proof (rule ssubst)
from True-or-False[of Q] show $True = Q$
proof (rule disjE)
assume $Q = True$
thus ?thesis by (rule sym)
next
assume $Q = False$
with Q have False by (rule rev-iffD1)
thus ?thesis by (rule FalseE)
qed
qed

next
assumes 2: $P = False$
thus ?thesis
proof (rule ssubst)
from True-or-False[of Q] show $False = Q$
proof (rule disjE)
assume $Q = True$
from 2 assms(2)[OF eqTrueE[OF this]] have False by (rule iffD1)
thus ?thesis by (rule FalseE)
next
  assume $Q = False$
  thus $\text{thesis by (rule sym)}$
  qed
qed

1.2.11 True (2)

lemma \(eqTrueI\): $P \implies P = True$
  by (iprover intro: iffI TrueI)

1.2.12 Universal quantifier (2)

lemma \(allI\):
  assumes $\forall x::'a. P x$
  shows $\forall x. P x$
  unfolding \(All-def\) by (iprover intro: ext eqTrueI assms)

1.2.13 Existential quantifier

lemma \(exI\): $P x \implies \exists x::'a. P x$
  unfolding \(Ex-def\) by (iprover intro: allI allE impI mp)

lemma \(exE\):
  assumes major: $\exists x::'a. P x$
  and minor: $\forall x. P x \implies Q$
  shows $Q$
  by (rule major \[unfolded Ex-def, THEN spec, THEN mp\]) (iprover intro: impI [THEN allI] minor)

1.2.14 Conjunction

lemma \(conjI\): $[P; Q] \implies P \land Q$
  unfolding \(and-def\) by (iprover intro: impI [THEN allI] mp)

lemma \(conjunct1\): $[P \land Q] \implies P$
  unfolding \(and-def\) by (iprover intro: impI dest: spec mp)

lemma \(conjunct2\): $[P \land Q] \implies Q$
  unfolding \(and-def\) by (iprover intro: impI dest: spec mp)

lemma \(conjE\):
  assumes major: $P \land Q$
  and minor: $[P; Q] \implies R$
  shows $R$
  apply (rule minor)
  apply (rule major \[THEN conjunct1\])
  apply (rule major \[THEN conjunct2\])
done
lemma `context-conjI`:
  assumes `P P \Rightarrow Q`
  shows `P \land Q`
  by (iprover intro: conjI assms)

1.2.15 Disjunction (2)

lemma `disjI1`: `P \Rightarrow P \lor Q`
  unfolding `or_def` by (iprover intro: allI impI mp)

lemma `disjI2`: `Q \Rightarrow P \lor Q`
  unfolding `or_def` by (iprover intro: allI impI mp)

1.2.16 Classical logic

lemma `classical`:
  assumes `prem: \neg P \Rightarrow P`
  shows `P`
  apply (rule True-or-False [THEN disjE, THEN eqTrueE])
    apply assumption
  apply (rule notI [THEN prem, THEN eqTrueI])
  apply (erule subst)
  apply assumption
  done

```
lemmas `ccontr` = `FalseE` [THEN classical]
```

`notE` with premises exchanged; it discharges `\neg R` so that it can be used to make elimination rules.

lemma `rev-notE`:
  assumes `premp: P`
    and `premnot: \neg R \Rightarrow \neg P`
  shows `R`
  apply (rule ccontr)
  apply (erule notE [OF premnot premp])
  done

Double negation law.

lemma `notnotD`: `\neg\neg P \Rightarrow P`
  apply (rule classical)
  apply (erule notE)
  apply assumption
  done

lemma `contrapos-pp`:
  assumes `p1: Q`
    and `p2: \neg P \Rightarrow \neg Q`
  shows `P`
  by (iprover intro: classical p1 p2 notE)
1.2.17 Unique existence

lemma \textit{exI}:
\begin{itemize}
  \item assumes $P \, a \land x. \, P \, x \Rightarrow x = a$
  \item shows $\exists!x. \, P \, x$
\end{itemize}
unfolding \textit{Ex1-def} by (iprover intro: assms \textit{exI} \textit{conjI} allI \textit{impI})

Sometimes easier to use: the premises have no shared variables. Safe!

lemma \textit{ex-ex1I}:
\begin{itemize}
  \item assumes \textit{ex-prem}: $\exists x. \, P \, x$
  \item and \textit{eq}: $\forall x, y. \, [P \, x; \, P \, y] \Rightarrow x = y$
  \item shows $\exists!x. \, P \, x$
\end{itemize}
by (iprover intro: \textit{ex-prem} [THEN \textit{exE}] \textit{ex1I} \textit{eq})

lemma \textit{ex1E}:
\begin{itemize}
  \item assumes \textit{major}: $\exists!x. \, P \, x$
  \item and \textit{minor}: $\forall x. \, [P \, x; \, \forall y. \, P \, y \Rightarrow y = x] \Rightarrow R$
  \item shows $R$
\end{itemize}
by (rule \textit{major} [unfolded \textit{Ex1-def}, THEN \textit{exE}])
apply (rule \textit{conjE})
apply (iprover intro: \textit{minor})
done

lemma \textit{ex1-implies-ex}: $\exists!x. \, P \, x \Rightarrow \exists x. \, P \, x$
apply (erule \textit{ex1E})
apply (rule \textit{exI})
apply assumption
done

1.2.18 Classical intro rules for disjunction and existential quantifiers

lemma \textit{disjCI}:
\begin{itemize}
  \item assumes $\neg \, Q \Rightarrow P$
  \item shows $P \lor Q$
\end{itemize}
by (rule classical) (iprover intro: assms \textit{disjI1} \textit{disjI2} \textit{notI} \textit{elim: notE})

lemma \textit{excluded-middle}: $\neg \, P \lor P$
by (iprover intro: \textit{disjCI})

Case distinction as a natural deduction rule. Note that $\neg \, P$ is the second case, not the first.

lemma \textit{case-split} [case-names True False]:
\begin{itemize}
  \item assumes \textit{prem1}: $P \Rightarrow Q$
  \item and \textit{prem2}: $\neg \, P \Rightarrow Q$
  \item shows $Q$
\end{itemize}
apply (rule \textit{excluded-middle} [THEN \textit{disjE}])
apply (erule \textit{prem2})
apply (erule \textit{prem1})
done

Classical implies ($\rightarrow$) elimination.

**lemma** `impCE`:

assumes major: $P \rightarrow Q$
and minor: $\neg P \Rightarrow R$ $Q \Rightarrow R$

shows $R$

apply (rule excluded-middle [of $P$, THEN disjE])
apply (iprover intro: minor major [THEN mp])+

done

This version of $\rightarrow$ elimination works on $Q$ before $P$. It works best for those cases in which $P$ holds "almost everywhere". Can’t install as default: would break old proofs.

**lemma** `impCE'`:

assumes major: $P \rightarrow Q$
and minor: $Q \Rightarrow R$ $\neg P \Rightarrow R$

shows $R$

apply (rule excluded-middle [of $P$, THEN disjE])
apply (iprover intro: minor major [THEN mp])+

done

Classical $\leftrightarrow$ elimination.

**lemma** `iffCE`:

assumes major: $P = Q$
and minor: $[P; Q] \Rightarrow R$ $[\neg P; \neg Q] \Rightarrow R$

shows $R$

by (rule major [THEN iffE]) (iprover intro: minor elim: impCE notE)

**lemma** `exCI`:

assumes $\forall x. \neg P x \Rightarrow P a$

shows $\exists x. P x$

by (rule ccontr) (iprover intro: assms exI allI notI notE [of $\exists x. P x$])

### 1.2.19 Intuitionistic Reasoning

**lemma** `impE'`:

assumes 1: $P \rightarrow Q$
and 2: $Q \Rightarrow R$
and 3: $P \rightarrow Q \Rightarrow P$

shows $R$

proof –
from 3 and 1 have $P$ .
with 1 have $Q$ by (rule impE)
with 2 show $R$ .
qed

**lemma** `allE'`:

assumes 1: $\forall x. P x$
and 2: \( P x \implies \forall x. P x \implies Q \)
shows \( Q \)
proof –
from 1 have \( P x \) by (rule spec)
from this and 1 show \( Q \) by (rule 2)
qed

lemma \textit{notE}:
assumes 1: \( \neg P \)
and 2: \( \neg P \implies P \)
shows \( R \)
proof –
from 2 and 1 have \( P \).
with 1 show \( R \) by (rule notE)
qed

lemma \textit{TrueE}:
\( \text{True} \implies P \implies P \).

lemma \textit{notFalseE}:
\( \neg \text{False} \implies P \implies P \).

lemmas \[ \text{Pure elim}! \] = disjE iffE FalseE conjE exE TrueE notFalseE
and \[ \text{Pure intro}! \] = iffI conjI implI TrueI notI allI refl
and \[ \text{Pure elim 2} \] = allE notE' impE'
and \[ \text{Pure intro} \] = exI disjI2 disjI1

lemmas \[ \text{trans} \] = trans
and \[ \text{sym} \] = sym not-sym
and \[ \text{Pure elim?} \] = iffD1 iffD2 impE

1.2.20 Atomizing meta-level connectives

axiomatization where
\( \text{eq-reflection: } x = y \implies x \equiv y \) — admissible axiom

lemma \textit{atomize-all} [\textit{atomize}]: \( \forall x. P x \equiv \text{Trueprop} (\forall x. P x) \)
proof
assume \( \forall x. P x \)
then show \( \forall x. P x \).
next
assume \( \forall x. P x \)
then show \( \forall x. P x \) by (rule allE)
qued

lemma \textit{atomize-imp} [\textit{atomize}]: \( A \implies B \equiv \text{Trueprop} (A \rightarrow B) \)
proof
assume \( r: A \implies B \)
show \( A \implies B \) by (rule impI) (rule \( r \))
next
assume \( A \implies B \) and \( A \)
then show \( B \) by (rule mp)
theory "HOL"

lemma atomize-not: (A \rightarrow False) \equiv Trueprop (\neg A)
proof
  assume r: A \rightarrow False
  show \neg A by (rule notI) (rule r)
next
  assume \neg A and A
  then show False by (rule notE)
qed

lemma atomize-eq [atomize, code]: (x \equiv y) \equiv Trueprop (x = y)
proof
  assume x \equiv y
  show x = y by (unfold \langle x \equiv y \rangle) (rule refl)
next
  assume x = y
  then show x \equiv y by (rule eq-reflection)
qed

lemma atomize-conj [atomize]: (A \&\& B) \equiv Trueprop (A \land B)
proof
  assume conj: A \&\& B
  show A \land B
  proof (rule conjI)
    from conj show A by (rule conjunctionD1)
    from conj show B by (rule conjunctionD2)
  qed
next
  assume conj: A \land B
  show A \&\& B
  proof
    from conj show A ..
    from conj show B ..
  qed
qed

lemmas [symmetric, rulify] = atomize-all atomize-imp
  and [symmetric, defn] = atomize-all atomize-imp atomize-eq

1.2.21 Atomizing elimination rules

lemma atomize-exL[atomize-elim]: (\forall x. P x \rightarrow Q) \equiv ((\exists x. P x) \rightarrow Q)
  by rule iprover+

lemma atomize-conjL[atomize-elim]: (A \rightarrow B \rightarrow C) \equiv (A \land B \rightarrow C)
  by rule iprover+

lemma atomize-disjL[atomize-elim]: ((A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C) \equiv ((A \lor
\[ B \Rightarrow C \Rightarrow C \]

by rule iprover+

**lemma** atomize-elimL[atomize-elim]: \((\land B. (A \Rightarrow B) \Rightarrow B) \equiv \text{Trueprop } A \).

### 1.3 Package setup

ML-file ⟨Tools/hologic.ML⟩

ML-file ⟨Tools/rewrite-hol-proof.ML⟩

setup ⟨Proofterm.set-preproc (Proof-Rewrite-Rules.standard-preproc Rewrite-HOL-Proof.rews)⟩

#### 1.3.1 Sledgehammer setup

Theorems blacklisted to Sledgehammer. These theorems typically produce clauses that are prolific (match too many equality or membership literals) and relate to seldom-used facts. Some duplicate other rules.

**named-theorems** no-atp theorems that should be filtered out by Sledgehammer

#### 1.3.2 Classical Reasoner setup

**lemma** imp-elim: \( P \Rightarrow Q \Rightarrow (\neg R \Rightarrow P) \Rightarrow (Q \Rightarrow R) \Rightarrow R \)

by (rule classical) iprover

**lemma** swap: \( \neg P \Rightarrow (\neg R \Rightarrow P) \Rightarrow R \)

by (rule classical) iprover

**lemma** thin-refl: \([ x = x; \text{PROP } W ] \Rightarrow \text{PROP } W \).

ML ⟨
structure Hypsubst = Hypsubst
{
  val dest-eq = HOLogic.dest-eq
  val dest-Trueprop = HOLogic.dest-Trueprop
  val dest-imp = HOLogic.dest-imp
  val eq-reflection = @{thm eq-reflection}
  val rev-eq-reflection = @{thm meta-eq-to-obj-eq}
  val imp-intr = @{thm impI}
  val rev-mp = @{thm rev-mp}
  val subst = @{thm subst}
  val sym = @{thm sym}
  val thin-refl = @{thm thin-refl};
};
open Hypsubst;
structure Classical = Classical
{
  val imp-elim = @{thm imp-elim}
  val not-elim = @{thm notE}
}⟩
val swap = @ {thm swap}
val classical = @ {thm classical}
val sizef = Drule.size-of-thm
val hyp-subst-tacs = [Hypsubst.hyp-subst-tac] 
);

structure Basic-Classical: BASIC-CLASSICAL = Classical;
open Basic-Classical;

setup 
(*prevent substitution on bool*)
let
  fun non-bool-eq (const-name HOL.eq, Type (_, [T, -])) = T <> typ (bool)
  | non-bool-eq - = false;
  fun hyp-subst-tac' ctxt =
    SUBGOAL (fn (goal, i) =>
      if Term.exists-Const non-bool-eq goal
      then Hypsubst.hyp-subst-tac ctxt i
      else no-tac);
  in
    Context-Rules.addSWrapper (fn ctxt =>
      fn tac =>
        hyp-subst-tac' ctxt ORELSE'
    tac)
  end

declare iffI [intro!]
and notI [intro!]
and impI [intro!]
and disjCI [intro!]
and conjI [intro!]
and TrueI [intro!]
and refl [intro!]

declare iffCE [elim!]
and FalseE [elim!]
and impCE [elim!]
and disjE [elim!]
and conjE [elim!]

declare ex-exII [intro!]
and allI [intro!]
and exI [intro]

declare exE [elim!]
allE [elim]

ML (val HOL-cs = classet-of context)
lemma contrapos-np: \( \neg Q \implies (\neg P \implies Q) \implies P \)
apply (erule swap)
apply (erule (1) meta-mp)
done

declare ex-ex1I [rule del, intro! 2]
and ex1I [intro]

declare ext [intro]

lemmas [intro?] = ext
and [elim?] = ex1-implies-ex

Better than ex1E for classical reasoner: needs no quantifier duplication!

lemma alt-ex1E [elim!]:
assumes major: \( \exists ! x. P x \)
and prem: \( \forall x. [P x; \forall y y'. P y \land P y' \implies y = y'] \implies R \)
shows R
apply (rule ex1E [OF major])
apply (rule prem)
apply assumption
apply (rule allI)+
apply (tactic \langle eresolve-tac context [Classical.dup-elim context @{thm allE}] \rangle)
apply iprover
done

ML:
structure Blast = Blast
{
  structure Classical = Classical
  val Trueprop-const = dest-Const const Trueprop
  val equality-name = const-name HOL.eq
  val not-name = const-name Not
  val notE = @{thm notE}
  val ccontr = @{thm ccontr}
  val hyp-subst-tac = Hypsubst.blast-hyp-subst-tac
};
val blast-tac = Blast.blast-tac;

1.3.3 THE: definite description operator

lemma the-equality [intro]:
assumes P a
and \( \forall x. P x \implies x = a \)
shows (THE x. P x) = a
by (blast intro: asms trans [OF arg-cong [where f=The] the-eq-trivial])
THEORY “HOL”

lemma theI:
  assumes P a and \( \forall x. P x \implies x = a \)
  shows P (THE x. P x)
  by (iprover intro: assms the-equality [THEN ssubst])

lemma theI': \( \exists! x. P x \implies P (\text{THE} x. P x) \)
  by (blast intro: theI)

Easier to apply than theI: only one occurrence of \( P \).

lemma theI2:
  assumes P a \( \forall x. P x \implies x = a \forall x. P x \implies Q x \)
  shows Q (THE x. P x)
  by (iprover intro: assms theI)

lemma theI2:
  assumes \( \exists! x. P x \forall x. P x \implies Q x \)
  shows Q (THE x. P x)
  by (iprover intro: assms(2) theI2[where P=P and Q=Q] ex1E[OF assms(1)]
              elim: allE impE)

lemma the1-equality [elim?): \( \exists! x. P x; P a \implies (\text{THE} x. P x) = a \)
  by blast

lemma the-sym-eq-trivial: (THE y. x = y) = x
  by blast

1.3.4 Simplifier

lemma eta-contract-eq: \((\lambda s. f s) = f \ldots\)

lemma simp-thms:
  shows not-not: (\neg \neg P) = P
  and Not-eq-iff: \((\neg P) = (\neg Q)) = (P = Q) \)
  and
      \( (P \neq Q) = (P = (\neg Q)) \)
      \( (P \lor \neg P) = \text{True} \quad (\neg P \lor P) = \text{True} \)
      \( (x = x) = \text{True} \)
  and not-True-eq-False [code]: (\neg \text{True}) = \text{False}
  and not-False-eq-True [code]: (\neg \text{False}) = \text{True}
  and
      \( (\neg P) \neq P \quad P \neq (\neg P)) \)
      \( (\text{True} = P) = P \)
  and eq-True: \((P = \text{True}) = P \)
  and (False = P) = (\neg P)
  and eq-False: \((P = \text{False}) = (\neg P) \)
  and
      \( (\text{True} \rightarrow P) = P \quad (\text{False} \rightarrow P) = \text{True} \)
      \( (P \rightarrow \text{True}) = \text{True} \quad (P \rightarrow P) = \text{True} \)
(\(P \rightarrow \text{False}\)) = (\(\neg P\)) (\(P \rightarrow \neg P\)) = (\(\neg P\))

\((P \land \text{True}) = P\) (\(\text{True} \land P\) = P)

\((P \land \text{False}) = \text{False}\) (\(\text{False} \land P\) = False)

\((P \land P) = P\) (\(P \land (P \land Q)\) = (\(P \land Q\))

\((P \land \neg P) = \text{False}\) (\(\neg P \land P\) = False)

\((P \lor \text{True}) = \text{True}\) (\(\text{True} \lor P\) = True)

\((P \lor \text{False}) = P\) (\(P \lor (P \lor Q)\) = (\(P \lor Q\) and

\((\forall x. P) = P\) (\(\exists x. P\) = P) \exists x. x = t \exists x. t = x

and

\(\land P. (\exists x. x = t \land P x) = P t\)

\(\land P. (\exists x. t = x \land P x) = P t\)

\(\land P. (\forall x. x = t \rightarrow P x) = P t\)

\(\land P. (\forall x. t = x \rightarrow P x) = P t\)

\((\forall x. x \neq t) = \text{False}\) (\(\forall x. t \neq x\) = False)

by (blast, blast, blast, blast, blast, iprover+)

lemma disj-absorb: \(A \lor A \longleftrightarrow A\)

by blast

lemma disj-left-absorb: \(A \lor (A \lor B) \longleftrightarrow A \lor B\)

by blast

lemma conj-absorb: \(A \land A \longleftrightarrow A\)

by blast

lemma conj-left-absorb: \(A \land (A \land B) \longleftrightarrow A \land B\)

by blast

lemma eq-ac:

shows eq-commute: \(a = b \longleftrightarrow b = a\)

and iff-left-commute: \((P \longleftrightarrow (Q \leftrightarrow R)) \longleftrightarrow (Q \leftrightarrow (P \leftrightarrow R))\)

and iff-assoc: ((\(P \longleftrightarrow Q\) \(\leftrightarrow R\) \longleftrightarrow (\(P \leftrightarrow (Q \leftrightarrow R))\)

by (iprover, blast+)

lemma neq-commute: \(a \neq b \longleftrightarrow b \neq a\) by iprover

lemma conj-ccoms:

shows conj-commute: \(P \land Q \longleftrightarrow Q \land P\)

and conj-left-commute: \(P \land (Q \land R) \longleftrightarrow Q \land (P \land R)\) by iprover+

lemma conj-assoc: \((P \land Q) \land R \longleftrightarrow P \land (Q \land R)\) by iprover

lemmas conj-ac = conj-commute conj-left-commute conj-assoc

lemma disj-ccoms:

shows disj-commute: \(P \lor Q \longleftrightarrow Q \lor P\)

and disj-left-commute: \(P \lor (Q \lor R) \longleftrightarrow Q \lor (P \lor R)\) by iprover+

lemma disj-assoc: \((P \lor Q) \lor R \longleftrightarrow P \lor (Q \lor R)\) by iprover
lemmas disj-ac = disj-commute disj-left-commute disj-assoc

lemma conj-disj-distribL: $P \land (Q \lor R) \iff P \land Q \lor P \land R$ by iprover
lemma conj-disj-distribR: $(P \lor Q) \land R \iff P \land R \lor Q \land R$ by iprover

lemma disj-conj-distribL: $P \lor (Q \land R) \iff (P \lor Q) \land (P \lor R)$ by iprover
lemma disj-conj-distribR: $(P \land Q) \lor R \iff (P \land R) \lor (Q \land R)$ by iprover

lemma imp-conjR: $(P \rightarrow (Q \land R)) = ((P \rightarrow Q) \land (P \rightarrow R))$ by iprover
lemma imp-conjL: $(P \land Q) \rightarrow R = (P \rightarrow (Q \rightarrow R))$ by iprover
lemma imp-disjL: $(P \lor Q) \rightarrow R = ((P \rightarrow R) \land (Q \rightarrow R))$ by iprover

These two are specialized, but imp-disj-not1 is useful in Auth/Yahalom.

lemma imp-disj-not1: $(P \rightarrow Q \lor R) \iff (\neg Q \rightarrow P \rightarrow R)$ by blast
lemma imp-disj-not2: $(P \rightarrow Q \lor R) \iff (\neg R \rightarrow P \rightarrow Q)$ by blast

lemma imp-disj1: $(P \rightarrow Q) \lor R \iff (P \rightarrow Q \lor R)$ by blast
lemma imp-disj2: $(Q \lor (P \rightarrow R)) \iff (P \rightarrow Q \lor R)$ by blast

lemma imp-cong: $(P = P') \Rightarrow (P' = (Q = Q')) \Rightarrow ((P \rightarrow Q) \iff (P' \rightarrow Q'))$
  by iprover

lemma de-Morgan-disj: $\neg (P \lor Q) \iff \neg P \land \neg Q$ by iprover
lemma de-Morgan-conj: $\neg (P \land Q) \iff \neg P \lor \neg Q$ by blast
lemma not-imp: $\neg (P \rightarrow Q) \iff P \land \neg Q$ by blast
lemma not-if$: P \neq Q \iff (P \iff \neg Q)$ by blast
lemma disj-not1: $P \lor Q \iff (P \rightarrow Q)$ by blast
lemma disj-not2: $P \land Q \iff (Q \rightarrow P)$ by blast — changes orientation :-(
lemma imp-conv-disj: $(P \rightarrow Q) \iff (\neg P) \lor Q$ by blast
lemma disj-imp: $P \lor Q \iff \neg P \rightarrow Q$ by blast

lemma iff-cone-conj-imp: $(P \rightarrow Q) \iff (P \rightarrow Q) \land (Q \rightarrow P)$ by iprover

lemma cases-simp: $(P \rightarrow Q) \land (\neg P \rightarrow Q) \iff Q$
— Avoids duplication of subgoals after if-split, when the true and false
— cases boil down to the same thing.
  by blast

lemma not-all: $\neg (\forall x. P x) \iff (\exists x. \neg P x)$ by blast
lemma imp-all: $(\forall x. P x) \rightarrow Q \iff (\exists x. P x \rightarrow Q)$ by blast
lemma not-ex: $\neg (\exists x. P x) \iff (\forall x. P x)$ by iprover
lemma imp-ex: $(\exists x. P x) \rightarrow Q \iff (\forall x. P x \rightarrow Q)$ by iprover
lemma all-not-ex: $(\forall x. P x) \iff \neg (\exists x. \neg P x)$ by blast

declare All-def [no-atp]

lemma ex-disj-distrib: $(\exists x. P x \lor Q x) \iff (\exists x. P x) \lor (\exists x. Q x)$ by iprover
lemma all-conj-distrib: \((\forall x. \, P \, x \land Q \, x) \iff (\forall x. \, P \, x) \land (\forall x. \, Q \, x)\) by iprover

The \(\land\) congruence rule: not included by default! May slow rewrite proofs down by as much as 50%

lemma conj-cong: \(P = P' \Rightarrow (P' \Rightarrow Q = Q') \Rightarrow (P \land Q) = (P' \land Q')\)
by iprover

lemma rev-conj-cong: \(Q = Q' \Rightarrow (Q' \Rightarrow P = P') \Rightarrow (P \land Q) = (P' \land Q')\)
by iprover

The | congruence rule: not included by default!

lemma disj-cong: \(P = P' \Rightarrow (\neg P' \Rightarrow Q = Q') \Rightarrow (P \lor Q) = (P' \lor Q')\)
by blast

if-then-else rules

lemma if-True [code]: \((\text{if True then } x \text{ else } y) = x\)
unfolding If-def by blast

lemma if-False [code]: \((\text{if False then } x \text{ else } y) = y\)
unfolding If-def by blast

lemma if-P: \(P \Rightarrow (\text{if } P \text{ then } x \text{ else } y) = x\)
unfolding If-def by blast

lemma if-not-P: \(\neg P \Rightarrow (\text{if } P \text{ then } x \text{ else } y) = y\)
unfolding If-def by blast

lemma if-split: \(P \ (\text{if } Q \text{ then } x \text{ else } y) = ((Q \rightarrow P \ x) \land (\neg Q \rightarrow P \ y))\)
apply (rule case-split [of \(Q\)])
apply (simplesubst if-P)
pref 3
apply (simplesubst if-not-P)
apply blast+
done

lemma if-split-asm: \(P \ (\text{if } Q \text{ then } x \text{ else } y) = (\neg ((Q \land \neg P \ x) \lor (\neg Q \land \neg P \ y)))\)
by (simplesubst if-split) blast

lemmas if-splits [no-atp] = if-split if-split-asm

lemma if-cancel: \((\text{if } c \text{ then } x \text{ else } y) = x\)
by (simplesubst if-split) blast

lemma if-eq-cancel: \((\text{if } x = y \text{ then } y \text{ else } x) = x\)
by (simplesubst if-split) blast

lemma if-bool-eq-conj: \((\text{if } P \text{ then } Q \text{ else } R) = ((P \rightarrow Q) \land (\neg P \rightarrow R))\)
— This form is useful for expanding if\(s\) on the RIGHT of the \(\Rightarrow\) symbol.
by (rule if-split)

lemma if-bool-eq-disj: \((if \, P \, then \, Q \, else \, R) = (\neg (P \wedge Q) \lor (\neg P \wedge R))\)

— And this form is useful for expanding if’s on the LEFT.

by (simplesubst if-split) blast

lemma Eq-TrueI: \(P \implies P \equiv True\) unfolding atomize_eq by iprover
lemma Eq-FalseI: \(\neg P \implies P \equiv False\) unfolding atomize_eq by iprover

let rules for simproc

lemma Let-folded: \(f \, x \equiv g \, x \implies Let \, x \, f \equiv Let \, x \, g\)

by (unfold Let-def)

lemma Let-unfold: \(f \, x \equiv g \implies Let \, x \, f \equiv g\)

by (unfold Let-def)

The following copy of the implication operator is useful for fine-tuning con- 
gruence rules. It instructs the simplifier to simplify its premise.

definition simp-implies :: prop \Rightarrow prop \Rightarrow prop \ (infixr =simp=> 1)

where simp-implies \(\equiv (\Rightarrow)\)

lemma simp-impliesI:

assumes PQ: \(PROP \, P \implies PROP \, Q\)

shows \(PROP \, P =simp=> PROP \, Q\)

apply (unfold simp-implies-def)

apply (rule PQ)

apply assumption

done

lemma simp-impliesE:

assumes PQ: \(PROP \, P =simp=> PROP \, Q\)

and P: \(PROP \, P\)

and QR: \(PROP \, Q \implies PROP \, R\)

shows \(PROP \, R\)

apply (rule QR)

apply (rule PQ [unfolded simp-implies-def])

apply (rule P)

done

lemma simp-implies-cong:

assumes PP’: \(PROP \, P \equiv PROP \, P’\)

and P’QQ’: \(PROP \, P’ \implies (PROP \, Q \equiv PROP \, Q’)\)

shows \((PROP \, P =simp=> PROP \, Q) \equiv (PROP \, P’ =simp=> PROP \, Q’)\)

unfolding simp-implies-def

proof (rule equal-intr-rule)

assume PQ: \(PROP \, P \implies PROP \, Q\)

and P’: \(PROP \, P’\)

from PP’ [symmetric] and P’ have \(PROP \, P\)
by (rule equal-elim-rule1)
then have $PROP\ Q$ by (rule $PQ$)
with $P'Q'\ [OF\ P']$ show $PROP\ Q'$ by (rule equal-elim-rule1)

next
assume $P'Q'$: $PROP\ P' \Longrightarrow PROP\ Q'$
and $P$: $PROP\ P$
from $PP'$ and $P$ have $P'$: $PROP\ P'$ by (rule equal-elim-rule1)
then have $PROP\ Q'$ by (rule $P'Q'$)
with $P'Q'$ [OF $P'$; symmetric] show $PROP\ Q$
by (rule equal-elim-rule1)

qed

lemma uncurry:
assumes $P \longrightarrow Q \longrightarrow R$
shows $P \land Q \longrightarrow R$
using assms by blast

lemma iff-allI:
assumes $\forall x. P\ x = Q\ x$
shows $\forall x. P \ x = (\forall x. Q \ x)$
using assms by blast

lemma iff-exI:
assumes $\forall x. P\ x = Q\ x$
shows $\exists x. P \ x = (\exists x. Q \ x)$
using assms by blast

lemma all-comm: $(\forall x \ y. P\ x \ y) = (\forall y \ x. P \ y \ x)$
by blast

lemma ex-comm: $(\exists x \ y. P\ x \ y) = (\exists y \ x. P \ y \ x)$
by blast

ML-file ⟨Tools/simpdata.ML⟩
ML ⟨open Simpdata⟩

setup \
  map-theory-simpset (put-simpset HOL-basic-ss) #>
  Simplifier.method-setup Splitter.split-modifiers
}

simproc-setup defined-Ex ($\exists x. P\ x) = (K\ Quantifier1.rearrange-ex$
simproc-setup defined-All ($\forall x. P\ x) = (K\ Quantifier1.rearrange-all$

Simproc for proving $(y = x) \equiv False$ from premise $\neg (x = y)$:

simproc-setup neq $(x = y) = (fn - \Rightarrow)$
let
  val neq-to-EQ-False = @{thm not-sym} RS @{thm Eq-False1};
  fun is-neq eq lhs rhs thm =
THEORY "HOL"

(case Thm.prop-of thm of
 - $(\text{Not} \ (eq' \ l' \ r')) =>
   \text{Not} = \text{HOLogic.Not andalso eq' = eq andalso}
   r' aconv lhs andalso l' aconv rhs
 | - => false);
fun proc ss ct =
  (case Thm.term-of ct of
   eq $ lhs $ rhs =>
    (case find-first (is-neq eq lhs rhs) (Simplifier.prems-of ss) of
     SOME thm => SOME (thm RS neq-to-EQ-False)
     | NONE => NONE)
 | - => NONE);
in proc end

simproc-setup let-simp (Let x f) = :
let
  fun count-loose (Bound i) k = if i >= k then 1 else 0
  | count-loose (s $ t) k = count-loose s k + count-loose t k
  | count-loose (Abs (-, -, t)) k = count-loose t (k + 1)
  | count-loose - - = 0;
  fun is-trivial-let (Const (\text{const-name} (\text{Let}), -) $ x $ t) =
    (case t of
     Abs (-, -, t') => count-loose t' 0 <= 1
     | - => true);
in fn - => fn cxt ==> fn ct =>>
  if is-trivial-let (Thm.term-of ct)
  then SOME @\{thm Let-def\} (*no or one occurrence of bound variable*)
  else
    let (*Norbert Schirmer's case*)
      val t = Thm.term-of ct;
      val (t', cxt') = yield-singleton (Variable.import-terms false) t cxt;
    in Option.map (hd o Variable.export cxt' cxt o single)
     (case t' of Const (\text{const-name} (\text{Let}),-) $ x $ f =>> (* x and f are already
      in normal form *)
      if is-Free x orelse is-Bound x orelse is-Const x
      then SOME @\{thm Let-def\}
      else
        let val n = case f of (Abs (x, -, -)) => x | - => x;
            val cx = Thm.cterm-of cxt x;
            val xT = Thm.typ-of-cterm cx;
            val cf = Thm.cterm-of cxt f;
            val fx-g = Simplifier.rewrite cxt (Thm.apply cf cx);
            val (- $ - $ g) = Thm.prop-of fx-g;
            val g' = abstract-over (x, g);
            val abs-g' = Abs (n, xT, g');

in
  if g aconv g' then
    let
      val rl =
      infer-instantiate ctxt [(((f, 0), cf), ((x, 0), cx))] @{thm Let-unfold};
      in SOME (rl OF [fx-g]) end
  else if (Envir.beta-eta-contract f) aconv (Envir.beta-eta-contract abs-g')
  then NONE (*avoid identity conversion*)
  else let
      val g' x = abs-g' $ x;
      val g-g' x = Thm.symmetric (Thm.beta-conversion false (Thm.cterm-of ctxt g' x));
      val rl =
      @{thm Let-folded} |> infer-instantiate ctxt
      [((f, 0), Thm.cterm-of ctxt f),
       ((x, 0), cx),
       ((g, 0), Thm.cterm-of ctxt abs-g')];
      in SOME (rl OF [Thm.transitive fx-g g-g' x]) end
    end
  end |
  - => NONE)
end

lemma True-implies-equals: (True \implies PROP P) \equiv PROP P
proof
  assume True \implies PROP P
  from this [OF TrueI] show PROP P .
next
  assume PROP P
  then show PROP P .
qed

lemma implies-True-equals: (PROP P \implies True) \equiv Trueprop True
by standard (intro TrueI)

lemma False-implies-equals: (False \implies P) \equiv Trueprop True
by standard simp-all

lemma implies-False-swap: 
NO-MATCH (Trueprop False) P \implies (False \implies PROP P \implies PROP Q) \equiv (PROP P \implies False \implies PROP Q)
by (rule swap-prems-eq)

lemma ex-simps:
\[ \forall P. Q. (\exists x. P x \land Q) = (\exists x. P x) \land Q \]
\( \forall P \ Q. (\exists x. \ P \land Q \ x) = (P \land (\exists x. \ Q \ x)) \)
\( \forall P \ Q. (\exists x. \ P \land Q \ x) = (P \land (\exists x. \ Q \ x)) \)
\( \forall P \ Q. (\exists x. \ P \lor Q \ x) = (P \lor (\exists x. \ Q \ x)) \)
\( \forall P \ Q. (\exists x. \ P \rightarrow Q \ x) = ((\forall x. \ P \rightarrow Q) \rightarrow Q) \)
\( \forall P \ Q. (\exists x. \ P \rightarrow Q \ x) = (P \rightarrow (\exists x. \ Q \ x)) \)

— Miniscoping: pushing in existential quantifiers.

**by** \((\text{iprover} \mid \text{blast})+\)

**lemma** all-simps:
\( \forall P \ Q. (\forall x. \ P \land Q \ x) = ((\forall x. \ P \ x) \land Q) \)
\( \forall P \ Q. (\forall x. \ P \land Q \ x) = (P \land (\forall x. \ Q \ x)) \)
\( \forall P \ Q. (\forall x. \ P \lor Q \ x) = ((\forall x. \ P \ x) \lor Q) \)
\( \forall P \ Q. (\forall x. \ P \lor Q \ x) = (P \lor (\forall x. \ Q \ x)) \)
\( \forall P \ Q. (\forall x. \ P \rightarrow Q \ x) = ((\forall x. \ P \ x) \rightarrow Q) \)
\( \forall P \ Q. (\forall x. \ P \rightarrow Q \ x) = (P \rightarrow (\forall x. \ Q \ x)) \)

— Miniscoping: pushing in universal quantifiers.

**by** \((\text{iprover} \mid \text{blast})+\)

**lemmas** \([\text{simp}] = \)

\text{triv-forall-equality} — prunes params
\text{True-implies-equals} \implies \text{True-equals} — prune True in asms
\text{False-implies-equals} — prune False in asms
\text{if-True}
\text{if-False}
\text{if-cancel}
\text{if-eq-cancel}
\text{imp-disjL} — In general it seems wrong to add distributive laws by default: they might cause exponential blow-up. But \text{imp-disjL} has been in for a while and cannot be removed without affecting existing proofs. Moreover, rewriting by \((P \lor Q \rightarrow R) = ((P \rightarrow R) \land (Q \rightarrow R))\) might be justified on the grounds that it allows simplification of \(R\) in the two cases.
\text{conj-assoc}
\text{disj-assoc}
\text{de-Morgan-conj}
\text{de-Morgan-disj}
\text{imp-disj1}
\text{imp-disj2}
\text{not-imp}
\text{disj-not1}
\text{not-all}
\text{not-ex}
\text{cases-simp}
\text{the-eq-trivial}
\text{the-sym-eq-trivial}
\text{ez-simps}
\text{all-simps}
\text{simp-thms}

**lemmas** \([\text{cong}] = \text{imp-cong simp-implies-cong} \)
lemmas [split] = if-split

ML (val HOL-ss = simpset-of context)

Simplifies $x$ assuming $c$ and $y$ assuming $\neg c$.

lemma if-cong:
  assumes $b = c$
  and $c \implies x = u$
  and $\neg c \implies y = v$
  shows $(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v)$
  using assms by simp

Prevents simplification of $x$ and $y$: faster and allows the execution of functional programs.

lemma if-weak-cong [cong]:
  assumes $b = c$
  shows $(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } x \text{ else } y)$
  using assms by (rule arg-cong)

Prevents simplification of $t$: much faster

lemma let-weak-cong:
  assumes $a = b$
  shows $(\text{let } x = a \text{ in } t x) = (\text{let } x = b \text{ in } t x)$
  using assms by (rule arg-cong)

To tidy up the result of a simproc. Only the RHS will be simplified.

lemma eq-cong2:
  assumes $u' = u$
  shows $(t \equiv u) \equiv (t \equiv u')$
  using assms by simp

lemma if-distrib: $f (\text{if } c \text{ then } x \text{ else } y) = (\text{if } c \text{ then } f x \text{ else } f y)$
  by simp

lemma if-distribR: $(\text{if } b \text{ then } f \text{ else } g) x = (\text{if } b \text{ then } f x \text{ else } g x)$
  by simp

lemma all-if-distrib: $(\forall x. \text{if } x = a \text{ then } P x \text{ else } Q x) \iff P a \land (\forall x. x \neq a \implies Q x)$
  by auto

lemma ex-if-distrib: $(\exists x. \text{if } x = a \text{ then } P x \text{ else } Q x) \iff P a \lor (\exists x. x \neq a \land Q x)$
  by auto

lemma if-if-eq-conj: $(\text{if } P \text{ then } \text{if } Q \text{ then } x \text{ else } y \text{ else } y) = (\text{if } P \land Q \text{ then } x \text{ else } y)$
  by simp
As a simplification rule, it replaces all function equalities by first-order equalities.

lemma fun-eq-iff: \( f = g \leftrightarrow (\forall x. f x = g x) \)
by auto

1.3.5 Generic cases and induction

Rule projections:

ML:

structure Project-Rule = Project-Rule
(
val conjunct1 = @{thm conjunct1}
val conjunct2 = @{thm conjunct2}
val mp = @{thm mp}
);

context
begin

qualified definition induct-forall P ≡ ∀ x. P x
qualified definition induct-implies A B ≡ A → B
qualified definition induct-equal x y ≡ x = y
qualified definition induct-conj A B ≡ A ∧ B
qualified definition induct-true ≡ True
qualified definition induct-false ≡ False

lemma induct-forall-eq: (\( \forall x. P x \)) ≡ Trueprop (induct-forall (\( \lambda x. P x \))
by (unfold atomize-all induct-forall-def)

lemma induct-implies-eq: (A ⇒ B) ≡ Trueprop (induct-implies A B)
by (unfold atomize-imp induct-implies-def)

lemma induct-equal-eq: (x ≡ y) ≡ Trueprop (induct-equal x y)
by (unfold atomize-eq induct-equal-def)

lemma induct-conj-eq: (A & B) ≡ Trueprop (induct-conj A B)
by (unfold atomize-conj induct-conj-def)

lemmas induct-atomize' = induct-forall-eq induct-implies-eq induct-conj-eq
lemmas induct-atomize = induct-atomize' induct-equal-eq
lemmas induct-rulify'[symmetric] = induct-atomize'
lemmas induct-rulify [symmetric] = induct-atomize
lemmas induct-rulify-fallback =
  induct-forall-def induct-implies-def induct-equal-def induct-conj-def
  induct-true-def induct-false-def

lemma induct-forall-conj: induct-forall (\( \lambda x. induct-conj (A x) (B x) \)) =
lemma induct-implies-conj: \( \text{induct-implies } C\ (\text{induct-conj } A\ B) = \text{induct-conj } (\text{induct-implies } C\ A)\ (\text{induct-implies } C\ B) \)
by (unfold induct-implies-def induct-conj-def) iprove

lemma induct-conj-curry: \( (\text{induct-conj } A\ B) = \Rightarrow\ PROP\ C \) \(\equiv\ (A = \Rightarrow\ B = \Rightarrow\ PROP\ C)\)
proof
assume \( r: \text{induct-conj } A\ B = \Rightarrow\ PROP\ C \)
assume \( ab: A\ B \)
show PROP C by (rule r) (simp add: induct-conj-def ab)
next
assume \( r: A = \Rightarrow\ B = \Rightarrow\ PROP\ C \)
assume \( ab: \text{induct-conj } A\ B \)
show PROP C by (rule r) (simp-all add: ab [unfolded induct-conj-def])
qed

lemmas induct-conj = induct-forall-conj induct-implies-conj induct-conj-curry

lemma induct-trueI: \( \text{induct-true} \)
by (simp add: induct-true-def)

Method setup.

ML-file ⟨~{/src/Tools/induct.ML}⟩

structure Induct = Induct
{
  val cases-default = @\{thm case-split\}
  val atomize = @\{thms induct-atomize\}
  val rulify = @\{thms induct-rulify\}
  val rulify-fallback = @\{thms induct-rulify-fallback\}
  val equal-def = @\{thm induct-equal-def\}
  fun dest-def (Const (const-name \(\approx\) induct-equal, |-) $ t $ u) = SOME (t, u)
    | dest-def - = NONE
  fun trivial-tac ctxt = match-tac ctxt @\{thms induct-trueI\}
)}

ML-file ⟨~{/src/Tools/induction.ML}⟩

declaration ⟨
fn - => Induct.map-simpset (fn ss => ss
  addsimpprocs
  [Simplifier.make-simproc context swap-induct-false
    \{lhss = \{term \(\approx\) induct-false \(\Rightarrow\) PROP P \(\Rightarrow\) PROP Q\},
           proc = fn - => fn - => fn ct =>>
    (case Thm.term_of ct of}
- \$ (P as \$ const\{induct-false\}) \$ (- \$ Q \$-) =>
  if P <> Q then SOME Drule.swap-prems-eq else NONE
| - => NONE},
Simplifier.make-simproc context induct-equal-conj-curry
{lhss = \{term induct-conj P Q => PROP R\},
proc = fn - => fn - => fn ct =>>
(case Thm.term-of ct of
  - \$ (- \$ P \$) \$ - =>>
  let
    fun is-conj (const induct-conj) $ P $ Q =
      is-conj P andalso is-conj Q
    | is-conj (Const (const-name induct-equal), -) $ - -$ = false
    | is-conj const (induct-true) = true
    | is-conj const (induct-false) = true
    | is-conj - = false
    in if is-conj P then SOME @{thm induct-conj-curry} else NONE end
  | - => NONE}\}
| Simplifier.set-mksimps (fn ctxt =>>
  Simpldata.mksimps Simpldata.mksimps-pairs ctxt #>
  map (rewrite-rule ctxt (map Thm.symmetric @{thms induct-rulify-fallback})))))

Pre-simplification of induction and cases rules

lemma [induct-simp]: \(\forall x.\text{ induct-equal } x t \implies PROP P x\) \(\equiv\) PROP P t
unfolding induct-equal-def
proof
  assume r: \(\forall x. x = t \implies PROP P x\)
  show PROP P t by (rule r \[OF refl\])
next
fix x
assume PROP P t t = x
then show PROP P x by simp
qed

lemma [induct-simp]: \(\forall x.\text{ induct-equal } t x \implies PROP P x\) \(\equiv\) PROP P t
unfolding induct-equal-def
proof
  assume r: \(\forall x. t = x \implies PROP P x\)
  show PROP P t by (rule r \[OF refl\])
next
fix x
assume PROP P t t = x
then show PROP P x by simp
qed

lemma [induct-simp]: \(\text{induct-false} \implies P\) \(\equiv\) Trueprop induct-true
unfolding induct-false-def induct-true-def
by (iprover intro: equal-intr-rule)
lemma \([\text{induct-simp}]: \text{induct-true} \Rightarrow \text{PROP P} \equiv \text{PROP P}\)
proof
  unfolding \text{induct-true-def}
  assume \text{True} \Rightarrow \text{PROP P}
  then show \text{PROP P} using \text{TrueI}.
next
  assume \text{PROP P}
  then show \text{PROP P}.
qed

lemma \([\text{induct-simp}]: \text{PROP P} \Rightarrow \text{induct-true} \equiv \text{Trueprop induct-true}\)
unfolding \text{induct-true-def}
by (iprover intro: equal-intr-rule)

lemma \([\text{induct-simp}]: (\forall x::'a::{}. \text{induct-true}) \equiv \text{Trueprop induct-true}\)
unfolding \text{induct-true-def}
by (iprover intro: equal-intr-rule)

lemma \([\text{induct-simp}]: \text{induct-implies induct-true P} \equiv P\)
by (simp add: \text{induct-implies-def induct-true-def})

lemma \([\text{induct-simp}]: x = x \leftrightarrow \text{True}\)
by (rule simp-thms)

end

ML-file \(\sim\!/src/Tools/induct-tacs.ML\)

1.3.6 Coherent logic

ML-file \(\sim\!/src/Tools/coherent.ML\)

ML |
structure Coherent = Coherent
{ 
  val atomize-elimL = @\{thm atomize-elimL\};
  val atomize-exL = @\{thm atomize-exL\};
  val atomize-conjL = @\{thm atomize-conjL\};
  val atomize-disjL = @\{thm atomize-disjL\};
  val operator-names = [\text{const-name} (HOL.disj), \text{const-name} (HOL.conj), \text{const-name} (Ex)];
};

1.3.7 Reorienting equalities

ML |
signature REORIENT-PROC =
sig
  val add : (term -> bool) -> theory -> theory
  val proc : morphism -> Proof.context -> cterm -> thm option
end;
structure Reorient-Proc : REORIENT-PROC =
struct
structure Data = Theory-Data
(type T = ((term $-> bool) $* stamp) list;
val empty = [];
val extend = I;
fun merge data : T = Library.merge (eq-snd (op =)) data;
fun matches thy t = exists (fn (m, -) => m t) (Data.get thy);
val meta-reorient = @{thm eq-commute [THEN eq-reflection]};
fun proc phi ctxt ct =
  let
    val thy = Proof-Context.theory-of ctxt;
in
  case Thm.term-of ct of
    (- $ t $ u) => if matches thy u then NONE else SOME meta-reorient
    | - => NONE
  end;
end;
⟩

1.4 Other simple lemmas and lemma duplicates

lemma all-cong1: $(\forall x. P x = P' x) \implies (\forall x. P x) = (\forall x. P' x)$
  by auto

lemma ex-cong1: $(\forall x. P x = P' x) \implies (\exists x. P x) = (\exists x. P' x)$
  by auto

lemma all-cong: $(\forall x. Q x \implies P x = P' x) \implies (\forall x. Q x \implies P x) = (\forall x. Q x \implies P' x)$
  by auto

lemma ex-cong: $(\forall x. Q x \implies P x = P' x) \implies (\exists x. Q x \wedge P x) = (\exists x. Q x \wedge P' x)$
  by auto

lemma ex1-eq [iff]: $\exists! x. x = t \iff \exists! x. t = x$
  by blast+

lemma choice-eq: $(\forall x. \exists! y. P x y) = (\exists! f. \forall x. P x (f x))$
  apply (rule iffI)
  apply (rule-lac a = \x. THE y. P x y in exI)
  apply (fast dest!: theI')
  apply (fast intro: the1-equality [symmetric])
apply (erule ex1E)
apply (rule allI)
apply (rule ex1I)
apply (erule spec)
apply (erule-tac
 x = \lambda z. if z = x then y else f z in allE)
apply (erule impE)
apply (rule allI)
apply (case-tac xa = x)
apply (drule-tac [3] x = x in fun-cong)
apply simp-all
done

lemmas eq-sym-conv = eq-commute

lemma nnf-simps:
(¬ (P ∧ Q)) = (¬ P ∨ ¬ Q)
(¬ (P ∨ Q)) = (¬ P ∧ ¬ Q)
(P → Q) = (¬ P ∨ Q)
(P = Q) = ((P ∧ Q) ∨ (¬ P ∧ ¬ Q))
(¬ (P = Q)) = ((P ∧ ¬ Q) ∨ (¬ P ∧ Q))
(¬ ¬ P) = P
by blast+

1.5 Basic ML bindings

ML:
val FalseE = @{thm FalseE}
val Let-def = @{thm Let-def}
val TrueI = @{thm TrueI}
val allE = @{thm allE}
val allI = @{thm allI}
val all-dupE = @{thm all-dupE}
val arg-cong = @{thm arg-cong}
val box-equals = @{thm box-equals}
val ccontr = @{thm ccontr}
val classical = @{thm classical}
val conjE = @{thm conjE}
val conjI = @{thm conjI}
val conjunct1 = @{thm conjunct1}
val conjunct2 = @{thm conjunct2}
val disjCI = @{thm disjCI}
val disjE = @{thm disjE}
val disjI1 = @{thm disjI1}
val disjI2 = @{thm disjI2}
val eq-reflection = @{thm eq-reflection}
val ex1E = @{thm ex1E}
val ex1I = @{thm ex1I}
val ex1-implies-ex = @{thm ex1-implies-ex}
val exE = @{thm exE}
locale cnf
begin

lemma clause2raw-notE: \[ [ P; \neg P ] \implies \text{False} \] by auto

lemma clause2raw-not-disj: \[ [ \neg P; \neg Q ] \implies \neg (P \lor Q) \] by auto

lemma clause2raw-not-not: \[ P \implies \neg \neg P \] by auto

lemma iff-refl: \((P::\text{bool}) = P\) by auto

lemma iff-trans: \[ \{ P = Q \} \implies (P = Q) \] by auto

lemma conj-cong: \[ \{ P = P'; Q = Q' \} \implies (P = Q = R) \] by auto

lemma disj-cong: \[ \{ P = P'; Q = Q' \} \implies (P \lor Q) \] by auto

lemma make-nnf-imp: \[ \{ \neg(P) = P'; Q = Q' \} \implies (P = (P' \lor Q')) \] by auto

lemma make-nnf-iff: \[ \{ P = P'; \neg(P) = NP; Q = Q' = \neg(Q) \} \implies (P = Q = (P' \lor Q')) \] by auto

lemma make-nnf-not-false: \((\neg\text{False}) = \text{True}\) by auto

lemma make-nnf-not-true: \((\neg\text{True}) = \text{False}\) by auto

lemma make-nnf-not-conj: \[ \{ \neg(P) = P'; \neg(Q) = Q' \} \implies (\neg(P \lor Q)) \] by auto

lemma make-nnf-not-disj: \[ \{ \neg(P) = P'; \neg(Q) = Q' \} \implies (\neg(P \lor Q)) \] by auto

⟩
lemma make-nnf-not-imp: \[P = P'; (\neg Q) = Q'\] \implies ((P \implies Q)) = (P' \land Q') by auto
lemma make-nnf-not-if: \[P = P'; (\neg P) = NP; Q = Q'; (\neg Q) = NQ\] \implies (P = Q) = ((P' \lor Q') \land (NP \lor NQ)) by auto
lemma make-nnf-not-not: \[P = P' \implies (\neg \neg P) = P'\] by auto

lemma simp-TF-conj-True-l: \[P = True; Q = Q'\] \implies (P \land Q) = Q' by auto
lemma simp-TF-conj-True-r: \[P = P'; Q = True\] \implies (P \land Q) = P' by auto
lemma simp-TF-conj-False-l: \[P = False \implies (P \land Q) = False\] by auto
lemma simp-TF-conj-False-r: \[Q = False \implies (P \land Q) = False\] by auto
lemma simp-TF-disj-True-l: \[P = True \implies (P \lor Q) = True\] by auto
lemma simp-TF-disj-True-r: \[Q = True \implies (P \lor Q) = True\] by auto
lemma simp-TF-disj-False-l: \[P = False; Q = Q'\] \implies (P \lor Q) = Q' by auto
lemma simp-TF-disj-False-r: \[P = P'; Q = False\] \implies (P \lor Q) = P' by auto

lemma make-cnfx-disj-conj-r: \[(P \lor (Q \land R)) = (P \lor Q) \land (P \lor R)\] by auto
lemma make-cnfx-disj-conj-l: \[(P \land (Q \lor R)) = (P \land Q) \lor (P \land R)\] by auto
lemma make-cnfx-disj-ex-l: \[(\exists x : bool. P x) \lor Q) = (\exists x. P x \lor Q)\] by auto
lemma make-cnfx-disj-ex-r: \[(P \lor (\exists x : bool. Q x)) \lor (Q \lor x)\] by auto
lemma make-cnfx-newlit: \[(P \lor Q) = (\exists x. (P \lor Q) \land (Q \lor x))\] by auto
lemma make-cnfx-ex-cong: \[(\forall x : bool. P x = Q x) \implies (\exists x. P x) = (\exists x. Q x)\] by auto

lemma weakening-thm: \[P; Q\] \implies Q by auto
lemma cnftr-eq-imp: \[P = Q; P\] \implies Q by auto
end

ML-file (Tools/cnf.ML)

2 \textit{NO-MATCH} simproc

The simplification procedure can be used to avoid simplification of terms of a certain form.

definition \textit{NO-MATCH} :: ('a => 'b => bool)
where \textit{NO-MATCH} pat val = \textit{NO-MATCH} pat val

lemma \textit{NO-MATCH}-cong[cong]; \textit{NO-MATCH} pat val = \textit{NO-MATCH} pat val
by (rule refl)
**THEORY “HOL”**

**declare** [[coercion-args NO-MATCH − −]]

**simproc-setup** NO-MATCH (NO-MATCH pat val) = (fn - => fn ctxt => fn ct =>)

\[
\text{let}
\]

\[
\begin{array}{l}
\text{val thy = Proof-Context.theory-of ctxt} \\
\text{val dest-binop = Term.dest-comb => apfst (Term.dest-comb => snd)} \\
\text{val m = Pattern.matches thy (dest-binop (Thm. term-of ct))} \\
\text{in if m then NONE else SOME @{thm NO-MATCH-def} end}
\end{array}
\]

This setup ensures that a rewrite rule of the form NO-MATCH pat val \(\Rightarrow t\) is only applied, if the pattern pat does not match the value val.

Tagging a premise of a simp rule with ASSUMPTION forces the simplifier not to simplify the argument and to solve it by an assumption.

**definition** ASSUMPTION :: bool => bool

**where** ASSUMPTION A \equiv A

**lemma** ASSUMPTION-cong[cong]: ASSUMPTION A = ASSUMPTION A

by (rule refl)

**lemma** ASSUMPTION-I: A \(\Rightarrow\) ASSUMPTION A

by (simp add: ASSUMPTION-def)

**lemma** ASSUMPTION-D: ASSUMPTION A \(\Rightarrow\) A

by (simp add: ASSUMPTION-def)

**setup**:

\[
\text{let}
\]

\[
\begin{array}{l}
\text{val assm-sol = mk-solver ASSUMPTION (fn ctxt =>} \\
\text{resolve-tac ctxt [@{thm ASSUMPTION-I}] THEN’} \\
\text{resolve-tac ctxt (Simplifier.prems-of ctxt))} \\
\text{in} \\
\text{map-theory-simpset (fn ctxt => Simplifier.addSolver (ctxt,assm-sol))} \\
\text{end}
\end{array}
\]

**2.1 Code generator setup**

**2.1.1 Generic code generator preprocessor setup**

**lemma** conj-left-cong: P \(\leftrightarrow\) Q \(\Rightarrow\) P \(\land\) R \(\leftrightarrow\) Q \(\land\) R

by (fact arg-cong)

**lemma** disj-left-cong: P \(\leftrightarrow\) Q \(\Rightarrow\) P \(\lor\) R \(\leftrightarrow\) Q \(\lor\) R

by (fact arg-cong)

**setup**:

\[
\text{Code-Preproc.map-pre (put-simpset HOL-basic-ss) #>}
\]
2.1.2 Equality

class equal =
  fixes equal :: 'a ⇒ 'a ⇒ bool
  assumes equal-eq: equal x y ←→ x = y
begin

lemma equal: equal = (=)
  by (rule ext equal-eq)+

lemma equal-refl: equal x x ←→ True
  unfolding equal by rule+

lemma eq-equal: (=) ≡ equal
  by (rule eq-reflection) (rule ext, rule ext, rule sym, rule equal-eq)

end

declare eq-equal [symmetric, code-post]
declare eq-equal [code]

setup {
Code-Preproc.map-pre (fn ctxt =>
  ctxt addsimprocs
  [Simplifier.make-simproc context equal
    {lhss = [term ⟨HOL.eq⟩],
     proc = fn - => fn - => fn ct =>>
     (case Thm.term_of ct of
       Const (_, Type ⟨type-name (fun, [Type -, -])⟩) => SOME @(thm eq-equal)
       | _ => NONE)}])
}

2.1.3 Generic code generator foundation

Datatype bool
code-datatype True False

lemma [code]:
  shows False ∧ P ←→ False
  and True ∧ P ←→ P
  and P ∧ False ←→ False
  and P ∧ True ←→ P
  by simp-all
lemma [code]:
  shows False ∨ P ←→ P
  and True ∨ P ←→ True
  and P ∨ False ←→ P
  and P ∨ True ←→ True
by simp-all

lemma [code]:
  shows (False → P) ←→ True
  and (True → P) ←→ P
  and (P → False) ←→ ¬ P
  and (P → True) ←→ True
by simp-all

More about prop

lemma [code nbe]:
  shows (True ⇒ PROP Q) ≡ PROP Q
  and (PROP Q ⇒ True) ≡ Trueprop True
  and (P ⇒ R) ≡ Trueprop (P → R)
by (auto intro!: equal-intr-rule)

lemma Trueprop-code [code]: Trueprop True ≡ Code-Generator.holds
by (auto intro!: equal-intr-rule holds)

declare Trueprop-code [symmetric, code-post]

Equality

declare simp-thms(6) [code nbe]

instantiation itself :: (type) equal
begin

definition equal-itself :: 'a itself ⇒ 'a itself ⇒ bool
  where equal-itself x y ←→ x = y

instance
by standard (fact equal-itself-def)
end

lemma equal-itself-code [code]: equal TYPE('a) TYPE('a) ←→ True
by (simp add: equal)

setup (Sign.add-const-constraint (const-name equal, SOME typ ('a::type ⇒ 'a ⇒ bool)):

lemma equal-alias-cert: OFCLASS('a, equal-class) ≡ ((=) :: 'a ⇒ 'a ⇒ bool) ≡ equal)
proof
  assume PROP ?ofclass
  show PROP ?equal
  by (tactic :ALLGOALS (resolve-tac context [Thm.unconstrainT @{thm eq-equal}])))
    (fact (PROP ?ofclass))
next
  assume PROP ?equal
  show PROP ?ofclass proof
  qed (simp add: (PROP ?equal))
qed

setup (Sign.add-const-constraint (const-name "equal", SOME typ ("a::equal ⇒ 'a ⇒ bool")));
setup (Nbe.add-const-alias @{thm equal-alias-cert});

Cases
lemma Let-case-cert:
  assumes CASE ≡ (λx. Let x f)
  shows CASE x ≡ f x
  using assms by simp-all
setup
  Code.declare-case-global @{thm Let-case-cert} #>
  Code.declare-undefined-global const-name (undefined)
}

declare [[code abort: undefined]]

2.1.4 Generic code generator target languages

*type* bool

*code-printing*

| *type-constructor* bool → |
| (SML) bool and (OCaml) bool and (Haskell) Bool and (Scala) Boolean |
| constant True → |
| (SML) true and (OCaml) true and (Haskell) True and (Scala) true |
| constant False → |
| (SML) false and (OCaml) false and (Haskell) False and (Scala) false |

*code-reserved* SML
  bool true false

*code-reserved* OCaml
  bool

*code-reserved* Scala
  Boolean
code-printing

constant Not \rightarrow
\begin{align*}
(SML) & \text{not} & (OCaml) & \text{not} & (Haskell) & \text{not} & (Scala) & \text{!} - \\
\end{align*}
\begin{align*}
| \text{constant HOL.conj} \rightarrow & \quad \text{(SML) infixl 1 andalso} & (OCaml) & \text{infixl} 2 & (Haskell) & \text{infixr} 3 & (Scala) & \text{infixl} 3 & \\
\end{align*}
\begin{align*}
| \text{constant HOL.disj} \rightarrow & \quad \text{(SML) infixl 0 orelse} & (OCaml) & \text{infixl} 2 & (Haskell) & \text{infixl} 2 & (Scala) & \text{infixl} 1 & \\
\end{align*}
\begin{align*}
| \text{constant HOL.implies} \rightarrow & \quad \text{(SML) !((if (\text{-})/ \text{then (\text{-})}/ \text{else true})} & (OCaml) & \text{!(if (\text{-})/ \text{then (\text{-})}/ \text{else true})} & (Haskell) & \text{!(if (\text{-})/ \text{then (\text{-})}/ \text{else True})} & (Scala) & \text{!(if ((\text{-})}/ (\text{-})/ \text{else true})} & \\
\end{align*}
\begin{align*}
| \text{constant If} \rightarrow & \quad \text{(SML) !(if (\text{-})/ \text{then (\text{-})}/ \text{else (\text{-})})} & (OCaml) & \text{!(if (\text{-})/ \text{then (\text{-})}/ \text{else (\text{-})})} & (Haskell) & \text{!(if (\text{-})/ \text{then (\text{-})}/ \text{else (\text{-})})} & (Scala) & \text{!(if ((\text{-})}/ (\text{-})/ \text{else (\text{-})})} & \\
\end{align*}

Using built-in Haskell equality.

code-printing

constant undefined \rightarrow
\begin{align*}
(SML) & \text{!((raise/ Fail/ undefined)} & (OCaml) & \text{failwith/ undefined} & (Haskell) & \text{error/ undefined} & (Scala) & \text{sys.error(undefined)} & \\
\end{align*}

2.1.5 Evaluation and normalization by evaluation

method-setup eval = \\
\begin{align*}
\text{let} & \\
\end{align*}
fun eval-tac ctxt = 
  let val conv = Code-Runtime.dynamic-holds-conv ctxt 
  in 
    CONVERSION (Conv.params-conv ~1 (K (Conv.concl-conv ~1 conv))) ctxt 
    THEN' 
    resolve-tac ctxt [TrueI] 
  end 
  in 
    Scan.succeed (SIMPLE-METHOD' o eval-tac) 
  end 
› solve goal by evaluation

method-setup normalization = ⟨
Scan.succeed (fn ctxt => 
  SIMPLE-METHOD'
  (CHANGED-PROP o
   (CONVERSION (Nbe.dynamic-conv ctxt)
    THEN-ALL-NEW (TRY o resolve-tac ctxt [TrueI]))))
› solve goal by normalization

2.2 Counterexample Search Units

2.2.1 Quickcheck

quickcheck-params [size = 5, iterations = 50]

2.2.2 Nitpick setup

named-theorems nitpick-unfold alternative definitions of constants as needed by Nitpick 
  and nitpick-simp equational specification of constants as needed by Nitpick 
  and nitpick-psimp partial equational specification of constants as needed by Nitpick 
  and nitpick-choice-spec choice specification of constants as needed by Nitpick

declare if-bool-eq-conj [nitpick-unfold, no-atp] 
  and if-bool-eq-disj [no-atp]

2.3 Preprocessing for the predicate compiler

named-theorems code-pred-def alternative definitions of constants for the Predicate Compiler 
  and code-pred-inline inlining definitions for the Predicate Compiler 
  and code-pred-simp simplification rules for the optimisations in the Predicate Compiler

2.4 Legacy tactics and ML bindings

ML ⟨
(* combination of (spec RS spec RS ... (j times) ... spec RS mp) *)
  local
fun wrong-prem (Const (const-name ⟨All⟩, -) $ Abs (_, _, t)) = wrong-prem t
| wrong-prem (Bound _) = true
| wrong-prem _ = false;

val filter-right = filter (not o wrong-prem o HOLogic.dest_Trueprop o hd o Thm.prems_of);

fun smp i = funpow i (fn m => filter-right ([spec] RL m)) [mp];
in
fun smp-tac ctxt j = EVERY'[dresolve-tac ctxt (smp j), assume-tac ctxt];
end;

local
val nnf-ss = simpset_of (put-simpset HOL-basic-ss context addsimps @{thms simp-thms nnf-simps});
in
fun nnf-conv ctxt = Simplifier.rewrite (put-simpset nnf-ss ctxt);
end

hide-const (open) eq equal

end

3 Abstract orderings

theory Orderings
imports HOL
keywords print-orders :: diag
begin

ML-file (~~/src/Provers/order.ML)

3.1 Abstract ordering

locale ordering =
fixes less-eq :: 'a ⇒ 'a ⇒ bool (infix ≤ 50)
and less :: 'a ⇒ 'a ⇒ bool (infix < 50)
assumes strict-iff-order: a < b ⇔ a ≤ b ∧ a ≠ b
assumes refl: a ≤ a — not iff: makes problems due to multiple (dual) interpretations
and antisym: a ≤ b ⇒ b ≤ a ⇒ a = b
and trans: a ≤ b ⇒ b ≤ c ⇒ a ≤ c
begin

lemma strict-implies-order:
a < b ⇒ a ≤ b
by (simp add: strict-iff-order)

lemma strict-implies-not-eq:
a < b ⇒ a ≠ b
by (simp add: strict-iff-order)

lemma not-eq-order-implies-strict:
  \( a \neq b \Rightarrow a \leq b \Rightarrow a < b \)
by (simp add: strict-iff-order)

lemma order-iff-strict:
  \( a \leq b \iff a < b \vee a = b \)
by (auto simp add: strict-iff-order refl)

lemma irrefl: — not iff: makes problems due to multiple (dual) interpretations
  \( \neg a < a \)
by (simp add: strict-iff-order)

lemma asym:
  \( a < b \Rightarrow b < a \Rightarrow False \)
by (auto simp add: strict-iff-order intro: antisym)

lemma strict-trans1:
  \( a \leq b \Rightarrow b < c \Rightarrow a < c \)
by (auto simp add: strict-iff-order intro: trans antisym)

lemma strict-trans2:
  \( a < b \Rightarrow b \leq c \Rightarrow a < c \)
by (auto simp add: strict-iff-order intro: trans antisym)

lemma strict-trans:
  \( a < b \Rightarrow b < c \Rightarrow a < c \)
by (auto intro: strict-trans1 strict-implies-order)
end

Alternative introduction rule with bias towards strict order

lemma ordering-strict1:
  fixes less-eq (infix \( \leq \))
  and less (infix <)
  assumes less-eq-less: \( \forall a b. a \leq b \iff a < b \vee a = b \)
  assumes asym: \( \forall a b. a < b \Rightarrow \neg b < a \)
  assumes irrefl: \( \forall a. \neg a < a \)
  assumes trans: \( \forall a b c. a < b \Rightarrow b < c \Rightarrow a < c \)
  shows ordering less-eq less
proof
  fix a b
  show \( a < b \iff a \leq b \land a \neq b \)
    by (auto simp add: less-eq-less asym irrefl)
next
  fix a
  show \( a \leq a \)
    by (auto simp add: less-eq-less)
next
fix \(a\), \(b\), \(c\)
assume \(a \leq b\) and \(b \leq c\) then show \(a \leq c\)
by (auto simp add: less-eq-less intro: trans)
next
fix \(a\), \(b\)
assume \(a \leq b\) and \(b \leq a\) then show \(a = b\)
by (auto simp add: less-eq-less asym)
qed

lemma ordering-dualI:
fixes less-eq (infix \(\leq\)) \((\leq)\)
and less (infix \(<\)) \((<)\)
assumes ordering \((\lambda a\ b.\ b \leq a)\) \((\lambda a\ b.\ b < a)\)
shows ordering less-eq less
proof –
from assms interpret ordering \(\lambda a\ b.\ b \leq a\) \(\lambda a\ b.\ b < a\).
show \(?thesis\)
by standard (auto simp: strict-iff-order refl intro: antisym trans)
qed

locale ordering-top = ordering +
fixes top :: `'a\ (\top)\`
assumes extremum [simp]: \(a \leq \top\)
begin

lemma extremum-uniqueI:
\(\top \leq a \implies a = \top\)
by (rule antisym) auto

lemma extremum-unique:
\(\top \leq a \iff a = \top\)
by (auto intro: antisym)

lemma extremum-strict [simp]:
\(\neg (\top < a)\)
using extremum [of \(a\)] by (auto simp add: order-iff-strict intro: asym irrefl)

lemma not-eq-extremum:
\(a \neq \top \iff a < \top\)
by (auto simp add: order-iff-strict intro: not-eq-order-implies-strict extremum)

end

3.2 Syntactic orders

class ord =
fixes less-eq :: `'a \Rightarrow 'a \Rightarrow bool
and less :: `'a \Rightarrow 'a \Rightarrow bool
THEORY “Orderings”

begin

notation
less-eq ('(≤')) and
less-eq ('(− ≤ −) [51, 51] 50) and
less ('(<') and
less ('(− < −) [51, 51] 50)

abbreviation (input)
greater-eq (infix ≥ 50)
where $x \geq y \equiv y \leq x$

abbreviation (input)
greater (infix > 50)
where $x > y \equiv y < x$

notation (ASCII)
less-eq ('(≤=)') and
less-eq ('(− <= −) [51, 51] 50)

notation (input)
greater-eq (infix ≥= 50)

end

3.3 Quasi orders

class preorder = ord +
assumes less-le-not-le: $x < y \iff x \leq y \land \neg (y \leq x)$

and order-refl [iff]: $x \leq x$

and order-trans: $x \leq y \Rightarrow y \leq z \Rightarrow x \leq z$

begin

Reflexivity.

lemma eq-refl: $x = y \Rightarrow x \leq y$
— This form is useful with the classical reasoner.
by (erule ssubit) (rule order-refl)

lemma less-irrefl [iff]: $\neg x < x$
by (simp add: less-le-not-le)

lemma less-imp-le: $x < y \Longrightarrow x \leq y$
by (simp add: less-le-not-le)

Asymmetry.

lemma less-not-sym: $x < y \Longrightarrow \neg (y < x)$
by (simp add: less-le-not-le)

lemma less-asym: $x < y \Longrightarrow (\neg P \Longrightarrow y < x) \Longrightarrow P$
by (drule less-not-sym, erule contrapos-np) simp

Transitivity.

lemma less-trans: \( x < y \implies y < z \implies x < z \)
by (auto simp add: less-le-not-le intro: order-trans)

lemma le-less-trans: \( x \leq y \implies y < z \implies x < z \)
by (auto simp add: less-le-not-le intro: order-trans)

lemma less-le-trans: \( x < y \implies y \leq z \implies x < z \)
by (auto simp add: less-le-not-le intro: order-trans)

Useful for simplification, but too risky to include by default.

lemma less-imp-not-less: \( x < y \implies \neg (y < x) \leftrightarrow True \)
by (blast elim: less-asym)

lemma less-imp-triv: \( x < y \implies (y < x \rightarrow P) \leftrightarrow True \)
by (blast elim: less-asym)

Transitivity rules for calculational reasoning

lemma less-asym': \( a < b \implies b < a \implies P \)
by (rule less-asym)

Dual order

lemma dual-preorder:
  class preorder \((\geq) (>)\)
  by standard (auto simp add: less-le-not-le intro: order-trans)

end

3.4 Partial orders

class order = preorder +
  assumes antisym: \( x \leq y \implies y \leq x \implies x = y \)
begin

lemma less-le: \( x < y \leftrightarrow x \leq y \land x \neq y \)
by (auto simp add: less-le-not-le intro: antisym)

sublocale order: ordering less-eq less + dual-order: ordering greater-eq greater
proof
  interpret ordering less-eq less
  by standard (auto intro: antisym order-trans simp add: less-le)

  show ordering less-eq less
  by (fact ordering-axioms)

  then show ordering greater-eq greater
  by (rule ordering-dualf)
qed
THEORY "Orderings"

Reflexivity.

**lemma** le-less: \( x \leq y \iff x < y \lor x = y \)

— NOT suitable for iff, since it can cause PROOF FAILED.

by (fact order.order-iff-strict)

**lemma** le-imp-less-or-eq: \( x \leq y \implies x < y \lor x = y \)
by (simp add: less-le)

Useful for simplification, but too risky to include by default.

**lemma** less-imp-not-eq: \( x < y \implies (x = y) \iff False \)
by auto

**lemma** less-imp-not-eq2: \( x < y \implies (y = x) \iff False \)
by auto

Transitivity rules for calculational reasoning

**lemma** neq-le-trans: \( a \neq b \implies a \leq b \implies a < b \)
by (fact order.not-eq-order-implies-strict)

**lemma** le-neq-trans: \( a \leq b \implies a \neq b \implies a < b \)
by (rule order.not-eq-order-implies-strict)

Asymmetry.

**lemma** eq-iff: \( x = y \iff x \leq y \land y \leq x \)
by (blast intro: antisym)

**lemma** antisym-conv: \( y \leq x \implies x \leq y \iff x = y \)
by (blast intro: antisym)

**lemma** less-imp-neq: \( x < y \implies x \neq y \)
by (fact order.strict-implies-not-eq)

**lemma** antisym-conv1: \( \neg x < y \implies x \leq y \iff x = y \)
by (simp add: local.le-less)

**lemma** antisym-conv2: \( x \leq y \implies \neg x < y \iff x = y \)
by (simp add: local.less-le)

**lemma** leD: \( y \leq x \implies \neg x < y \)
by (auto simp: less-le antisym)

Least value operator

**definition** (in ord)

Least :: (\'a \Rightarrow bool) \Rightarrow \'a (binder LEAST 10) where

Least \( P = (\text{THE } x. \ P x \land (\forall y. \ P y \implies x \leq y)) \)

**lemma** Least-equality: assumes \( P x \)
and \( \bigwedge y. P y \Rightarrow x \leq y \)
shows \( \text{Least } P = x \)

unfolding Least-def by (rule the-equality)
(blast intro: assms antisym)+

lemma LeastI2-order:
assumes \( P x \)
and \( \bigwedge y. P y \Rightarrow x \leq y \)
and \( \bigwedge x. P x \Rightarrow \forall y. P y \Rightarrow x \leq y \Rightarrow Q x \)
shows \( Q \) (Least \( P \))
unfolding Least-def by (rule theI2)
(blast intro: assms antisym)+

lemma Least-ex1:
assumes \( \exists ! x. P x \wedge (\forall y. P y \Rightarrow x \leq y) \)
shows Least1I: \( P \) (Least \( P \)) and Least1-le: \( P z \Rightarrow \text{Least } P \leq z \)
using theI'[OF assms]
unfolding Least-def
by auto

Greatest value operator

definition Greatest :: (‘a ⇒ bool) ⇒ ‘a (binder GREATEST 10) where
Greatest \( P = (\text{THE } x. P x \wedge (\forall y. P y \Rightarrow x \geq y)) \)

lemma GreatestI2-order:
[ P x; 
\[ y. P y \Rightarrow x \geq y; \]
\[ x. [ P x; \forall y. P y \Rightarrow x \geq y ] \Rightarrow Q x ] \]
⇒ Q (Greatest \( P \))
unfolding Greatest-def
by (rule theI2) (blast intro: antisym)+

lemma Greatest-equality:
[ P x; \[ y. P y \Rightarrow x \geq y ] \Rightarrow \text{Greatest } P = x
unfolding Greatest-def
by (rule the-equality) (blast intro: antisym)+

end

lemma ordering-orderI1:
fixes less-eq (infix \( \leq \) 50)
and less (infix < 50)
assumes ordering less-eq less
shows class.order less-eq less
proof –
from assms interpret ordering less-eq less .
show \?thesis
by standard (auto intro: antisym trans simp add: refl strict-iff-order)
qed
lemma order-strictI:
  fixes less (infix ⊏)
  and less-eq (infix ⊑)
  assumes \( \forall a \ b. \ a \sqsubseteq b \leftrightarrow a \sqsubseteq b \lor a = b \)
  assumes \( \forall a \ b. \ a \sqsubseteq b \implies \neg b \sqsubseteq a \)
  assumes \( \forall a. \ \neg a \sqsubseteq a \)
  assumes \( \forall a \ b \ c. \ a \sqsubseteq b \implies b \sqsubseteq c \implies a \sqsubseteq c \)
  shows class.order less-eq less
  by (rule ordering-orderI) (rule ordering-strictI, (fact assms)+)

context order
begin

Dual order

lemma dual-order:
  class.order (≥) (>)
  using dual-order.ordering-axioms by (rule ordering-orderI)

end

3.5 Linear (total) orders

class linorder = order +
  assumes linear: \( x \leq y \lor y \leq x \)
begin

lemma less-linear: \( x < y \lor x = y \lor y < x \)
unfolding less-le using less-le linear by blast

lemma le-less-linear: \( x \leq y \lor y < x \)
by (simp add: le-less less-linear)

lemma le-cases [case-names le ge]:
  \( x \leq y \implies P \) \( \implies (y \leq x \implies P) \implies P \)
using linear by blast

lemma (in linorder) le-cases3:
  \( [[x \leq y; y \leq z]] 
      \implies P; \ [y \leq x; x \leq z] \implies P; \ [x \leq z; z \leq y] \implies P; \)
  \( [z \leq y; y \leq x] \implies P; \ [y \leq z; z \leq x] \implies P; \ [z \leq x; x \leq y] \implies P \) \( \implies P \)
by (blast intro: le-cases)

lemma linorder-cases [case-names less equal greater]:
  \( x < y \implies P \) \( \implies (x = y \implies P) \implies (y < x \implies P) \implies P \)
using less-linear by blast

lemma linorder-wlog [case-names le sym]:
  \( \forall a \ b. \ a \leq b \implies P \ a \ b \) \( \implies (\forall a \ b. \ P \ b \ a \implies P \ a \ b) \implies P \ a \ b \)
by (cases rule: le-cases[of a b]) blast+
lemma not-less: \( \neg x < y \iff y \leq x \)

unfolding less-le
using linear by (blast intro: antisym)

lemma not-less-iff-gr-or-eq: \( \neg (x < y) \iff (x > y \lor x = y) \)

by (auto simp add: not-less le-less)

lemma not-le: \( \neg x \leq y \iff y < x \)

unfolding less-le
using linear by (blast intro: antisym)

lemma neq-iff: \( x \neq y \iff x < y \lor y < x \)

by (cut-tac \( x = x \) and \( y = y \) in less-linear, auto)

lemma neqE: \( x \neq y \implies (x < y \implies R) \implies (y < x \implies R) \implies R \)

by (simp add: neq-iff) blast

lemma antisym-conv3: \( \neg y < x \implies \neg x < y \iff x = y \)

by (blast intro: antisym dest: not-less [THEN iffD1])

lemma leI: \( \neg x < y \implies y \leq x \)

unfolding not-less.

lemma not-le-imp-less: \( \neg y \leq x \implies x < y \)

unfolding not-le.

lemma linorder-less-wlog[case-names less refl sym]:

\[
\begin{align*}
\forall a b. a < b & \implies P a b; \\
\forall a. P a a & \implies P a b; \\
\forall a b. P b a & \implies P a b \\
\end{align*}
\]

using antisym-conv3 by blast

Dual order

lemma dual-linorder:

\[ \text{class.linorder} (\geq) (\leq) \]

by (rule class.linorder.intro, rule dual-order) (unfold-locales, rule linear)

end

Alternative introduction rule with bias towards strict order

lemma linorder-strictI:

fixes less-eq (infix \( \leq \) 50)

and less (infix \( < \) 50)

assumes \( \text{class.order} \ less-eq \ less \)

assumes trichotomy: \( \forall a b. a < b \lor a = b \lor b < a \)

shows \( \text{class.linorder} \ less-eq \ less \)

proof –

interpret order less-eq less

by (fact \( \text{class.order} \ less-eq \ less \))

show \?thesis
proof
fix a b
show a \leq b \lor b \leq a
using trichotomy by (auto simp add: le-less)
qed

3.6 Reasoning tools setup

ML ⟨signature ORDERS =
sig
val print-structures: Proof.context -> unit
val order-tac: Proof.context -> thm list -> int -> tactic
val add-struct: string * term list -> string -> attribute
val del-struct: string * term list -> attribute
end;
structure Orders: ORDERS =
struct
(* context data *)
fun struct-eq ((s1: string, ts1), (s2, ts2)) =
s1 = s2 andalso eq-list (op aconv) (ts1, ts2);
structure Data = Generic-Data
(  type T = ((string * term list) * Order-Tac.less-arith) list;
  (* Order structures:
     identifier of the structure, list of operations and record of theorems
     needed to set up the transitivity reasoner,
     identifier and operations identify the structure uniquely. *)
  val empty = [];
  val extend = I;
  fun merge data = AList.join struct-eq (K fst) data);
fun print-structures ctxt =
let
  val structs = Data.get (Context.Proof ctxt);
  fun pretty-term t = Pretty.block
    [Pretty.quote (Syntax.pretty-term ctxt t), Pretty.brk 1,
     Pretty.str ::, Pretty.brk 1,
     Pretty.quote (Syntax.pretty-typ ctxt (type-of t))];
  fun pretty-struct ((s, ts), -) = Pretty.block
    [Pretty.str s, Pretty.str ::, Pretty.brk 1,
     Pretty.enclose ( ) (Pretty.breaks (map pretty-term ts))];
in
Pretty.writeln (Pretty.big-list order structures: (map pretty-struct structs))
end;

val _ =
Outer-Syntax.command command-keyword (print-orders)
  print order structures available to transitivity reasoner
  (Scan.succeed (Toplevel.keep (print-structures o Toplevel.context-of)));

(* tactics *)
fun struct-tac ((s, ops), thms) ctxt facts =
  let
    val [eq, le, less] = ops;
    fun decomp thy (const⟨Trueprop⟩ $ t) =
      let
        val excluded t =
          (* exclude numeric types: linear arithmetic subsumes transitivity *)
        let
          val T = type-of t
        in
          T = HOLogic.natT orelse T = HOLogic.intT orelse T = HOLogic.realT
        end;
        fun rel (bin-op $ t1 $ t2) =
          if excluded t1 then NONE
          else if Pattern.matches thy (eq, bin-op) then SOME (t1, =, t2)
          else if Pattern.matches thy (le, bin-op) then SOME (t1, <=, t2)
          else if Pattern.matches thy (less, bin-op) then SOME (t1, <, t2)
          else NONE
        | rel _ = NONE;
        fun dec (Const ⟨const-name Not, -⟩ $ t) =
          (case rel t of NONE =>
            NONE |
            SOME (t1, rel, t2) => SOME (t1, ~ rel, t2))
          | dec x = rel x;
          in
          dec t end
      | decomp _ _ = NONE;
    in
      (case s of
        order => Order-Tac.partial-tac decomp thms ctxt facts
        linorder => Order-Tac.linear-tac decomp thms ctxt facts
        _ => error (Unknown order kind "quote s " encountered in transitivity reasoner))
    end

fun order-tac ctxt facts =
  FIRST' (map (fn s => CHANGED o struct-tac s ctxt facts) (Data.get (Context.Proof ctxt)));
THEORY "Orderings"

(* attributes *)

fun add-struct s tag = Thm.declaration-attribute (fn thm => Data.map (AList.map-default struct-eq (s, Order-Tac.empty TrueI) (Order-Tac.update tag thm)));
fun del-struct s = Thm.declaration-attribute (fn - => Data.map (AList.delete struct-eq s));

end;

attribute-setup order = ⟨Scan.lift ((Args.add -- Args.name >> (fn (-, s) => SOME s) || Args.del >> K NONE) --] Args.colon (* FIXME || Scan.succeed true *) ) -- Scan.lift Args.name -- Scan.repeat ArgsTERM
  |> (fn ((SOME tag, n), ts) => Orders.add-struct (n, ts) tag
       | ((NONE, n), ts) => Orders.del-struct (n, ts))⟩

method-setup order = ⟨Scan.succeed (fn ctxt => SIMPLE-METHOD' (Orders.order-tac ctxt []))⟩

Declarations to set up transitivity reasoner.

context order begin

declare less-irrefl [THEN notE, order add less-reflE: order (=) :: 'a ⇒ 'a ⇒ bool (<=) (<)]

declare order-refl [order add le-refl: order (=) :: 'a ⇒ 'a ⇒ bool (<=) (<=)]

declare less-imp-le [order add less-imp-le: order (=) :: 'a ⇒ 'a ⇒ bool (<=) (<=)]

declare antisym [order add eqI: order (=) :: 'a ⇒ 'a ⇒ bool (<=) (<=)]

declare eq-refl [order add eqD1: order (=) :: 'a ⇒ 'a ⇒ bool (<=) (<=)]

declare sym [THEN eq-refl, order add eqD2: order (=) :: 'a ⇒ 'a ⇒ bool (<=) (<=)]

declare less-trans [order add less-trans: order (=) :: 'a ⇒ 'a ⇒ bool (<=) (<=)]
declare less-le-trans [order add le-trans: order (=) :: 'a => 'a => bool (<=) (<)]

declare le-less-trans [order add le-less-trans: order (=) :: 'a => 'a => bool (<=) (<)]

declare order-trans [order add le-trans: order (=) :: 'a => 'a => bool (<=) (<)]

declare le-neq-trans [order add le-neq-trans: order (=) :: 'a => 'a => bool (<=) (<)]

declare neq-le-trans [order add neq-le-trans: order (=) :: 'a => 'a => bool (<=) (<)]

declare less-imp-neq [order add less-imp-neq: order (=) :: 'a => 'a => bool (<=) (<)]

declare eq-neq-eq-imp-neq [order add eq-neq-eq-imp-neq: order (=) :: 'a => 'a => bool (<=) (<)]

declare not-sym [order add not-sym: order (=) :: 'a => 'a => bool (<=) (<)]

declare neq-le-trans [order add neq-le-trans: order (=) :: 'a => 'a => bool (<=) (<)]

end

class context linorder
begin

declare [[order del: order (=) :: 'a => 'a => bool (<=) (<)]]

declare less-irrefl [THEN notE, order add less-reflE: linorder (=) :: 'a => 'a => bool (<=) (<)]

declare order-refl [order add le-refl: linorder (=) :: 'a => 'a => bool (<=) (<)]

declare less-imp-le [order add less-imp-le: linorder (=) :: 'a => 'a => bool (<=) (<)]

declare not-less [THEN iffD2, order add not-lessI: linorder (=) :: 'a => 'a => bool (<=) (<)]

declare not-le [THEN iffD2, order add not-leI: linorder (=) :: 'a => 'a => bool (<=) (<)]

declare not-less [THEN iffD1, order add not-lessD: linorder (=) :: 'a => 'a => bool (<=) (<)]

declare not-le [THEN iffD1, order add not-leD: linorder (=) :: 'a => 'a => bool (<=) (<)]]
THEORY “Orderings"

```plaintext
declare antisym [order add eqI: linorder (=) :: 'a => 'a => bool (<=) (<)]]
declare eq-refl [order add eqD1: linorder (=) :: 'a => 'a => bool (<=) (<)]]
declare sym [THEN eq-refl, order add eqD2: linorder (=) :: 'a => 'a => bool (<=) (<)]]
declare less-trans [order add less-trans: linorder (=) :: 'a => 'a => bool (<=) (<)]]
declare less-le-trans [order add less-le-trans: linorder (=) :: 'a => 'a => bool (<=) (<)]]
declare le-less-trans [order add le-less-trans: linorder (=) :: 'a => 'a => bool (<=) (<)]]
declare order-trans [order add le-trans: linorder (=) :: 'a => 'a => bool (<=) (<)]]
declare le-neq-trans [order add le-neq-trans: linorder (=) :: 'a => 'a => bool (<=) (<)]]
declare neq-le-trans [order add neq-le-trans: linorder (=) :: 'a => 'a => bool (<=) (<)]]
declare less-imp-neq [order add less-imp: linorder (=) :: 'a => 'a => bool (<=) (<)]]
declare eq-neq-eq-imp-neq [order add eq-neq-eq-imp-neq: linorder (=) :: 'a => 'a => booo (<=) (<)]]
declare not-sym [order add not-sym: linorder (=) :: 'a => 'a => bool (<=) (<)]]
end

setup ::
    map-theory-simpset (fn ctxt0 => ctxt0 addSolver
    mk-solver Transitivity (fn ctxt => Orders.order-tac ctxt (Simplifier.prems-of ctxt))))

(*Adding the transitivity reasoners also as safe solvers showed a slight speed-up, but the reasoning strength appears to be not higher (at least no breaking of additional proofs in the entire HOL distribution, as of 5 March 2004, was observed).*

ML ::
local
    fun prp t thm = Thm.prop-of thm = t; (* FIXME proper aconv! ? *)
in
```
fun antisym-le-simproc ctxt ct =
  (case Thm.term-of ct of
   (le as Const (_, T)) $ r $ s =>
   (let
     val prems = Simplifier.prems-of ctxt;
     val less = Const (const-name:less:, T);
     val t = HOLogic.mk_Trueprop(le $ s $ r);
     in
     (case find-first (prp t) prems of
      NONE =>
      let val t = HOLogic.mk_Trueprop(HOLogic.Not $(less $ r $ s)) in
      (case find-first (prp t) prems of
       NONE => SOME thm =>> SOME(mk-meta-eq(thm RS @{thm antisym-conv1})))
      end
      | SOME thm =>> SOME (mk-meta-eq (thm RS @{thm linorder-classantisym-conv}))
      end handle THM =>> NONE)
    | - =>> NONE)
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
THEORY “Orderings”

syntax

\[-\text{Ex-less} :: \text{idt}, 'a, bool] => bool \quad ((3\text{EX} \preceq \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{All-less-eq} :: \text{idt}, 'a, bool] => bool \quad ((3\text{ALL} \preceq \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{Ex-less-eq} :: \text{idt}, 'a, bool] => bool \quad ((3\text{EX} \preceq \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{All-greater} :: \text{idt}, 'a, bool] => bool \quad ((3\text{ALL} \rightarrow \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{Ex-greater} :: \text{idt}, 'a, bool] => bool \quad ((3\text{EX} \rightarrow \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{All-greater-eq} :: \text{idt}, 'a, bool] => bool \quad ((3\text{ALL} \rightarrow= \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{Ex-greater-eq} :: \text{idt}, 'a, bool] => bool \quad ((3\text{EX} \rightarrow= \cdot \cdot \cdot) [0, 0, 10] 10)\]

\[-\text{All-neq} :: \text{idt}, 'a, bool] => bool \quad ((3\text{ALL} \sim \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{Ex-neq} :: \text{idt}, 'a, bool] => bool \quad ((3\text{EX} \sim \cdot \cdot \cdot) [0, 0, 10] 10)\]

syntax (input)

\[-\text{All-less} :: \text{idt}, 'a, bool] => bool \quad ((3\forall \preceq \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{Ex-less} :: \text{idt}, 'a, bool] => bool \quad ((3\exists \preceq \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{All-less-eq} :: \text{idt}, 'a, bool] => bool \quad ((3\forall \leq \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{Ex-less-eq} :: \text{idt}, 'a, bool] => bool \quad ((3\exists \leq \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{All-greater} :: \text{idt}, 'a, bool] => bool \quad ((3\forall \rightarrow \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{Ex-greater} :: \text{idt}, 'a, bool] => bool \quad ((3\exists \rightarrow \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{All-greater-eq} :: \text{idt}, 'a, bool] => bool \quad ((3\forall \rightarrow= \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{Ex-greater-eq} :: \text{idt}, 'a, bool] => bool \quad ((3\exists \rightarrow= \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{All-neq} :: \text{idt}, 'a, bool] => bool \quad ((3\forall \sim \cdot \cdot \cdot) [0, 0, 10] 10)\]
\[-\text{Ex-neq} :: \text{idt}, 'a, bool] => bool \quad ((3\exists \sim \cdot \cdot \cdot) [0, 0, 10] 10)\]

translations

\[\forall x < y. \quad P \Rightarrow \forall x. x < y \quad \rightarrow \quad P\]
\[\exists x < y. \quad P \Rightarrow \exists x. x < y \land P\]
\[\forall x \leq y. \quad P \Rightarrow \forall x. x \leq y \quad \rightarrow \quad P\]
\[\exists x \leq y. \quad P \Rightarrow \exists x. x \leq y \land P\]
\[\forall x > y. \quad P \Rightarrow \forall x. x > y \quad \rightarrow \quad P\]
\[\exists x > y. \quad P \Rightarrow \exists x. x > y \land P\]
\[\forall x \geq y. \quad P \Rightarrow \forall x. x \geq y \quad \rightarrow \quad P\]
\[\exists x \geq y. \quad P \Rightarrow \exists x. x \geq y \land P\]
\[\forall x \neq y. \quad P \Rightarrow \forall x. x \neq y \quad \rightarrow \quad P\]
\[\exists x \neq y. \quad P \Rightarrow \exists x. x \neq y \land P\]

print-translation:

let

val All-binder = Mixfix.binder-name const-syntext: All;;
THEORY “Orderings”

val Ex-binder = Mixfix.binder-name const-syntax (Ex);
val impl = const-syntax (HOL.implies);
val conj = const-syntax (HOL.conj);
val less = const-syntax (less);
val less-eq = const-syntax (less-eq);

val trans =
[(All-binder, impl, less),
 (syntax-const (-All-less), syntax-const (-All-greater)),
 (All-binder, impl, less-eq),
 (syntax-const (-All-less-eq), syntax-const (-All-greater-eq)),
 (Ex-binder, conj, less),
 (syntax-const (-Ex-less), syntax-const (-Ex-greater)),
 (Ex-binder, conj, less-eq),
 (syntax-const (-Ex-less-eq), syntax-const (-Ex-greater-eq))];

fun matches-bound v t =
  (case t of
    Const (syntax-const (-bound), -) \Free (v', -) => v = v'
  | _ => false);
fun contains-var v = Term.exists-subterm (fn Free (x, -) => x = v | _ => false);
fun mk x c n P = Syntax.const c $ Syntax.Trans.mark-bound-body x $ n $ P;

fun tr' q = (q, fn _ =>
  (fn (Const (syntax-const (-bound), -) $ Free (v, T),
    Const (c, -) $ (Const (d, -) $ t $ u) $ P) =>
      (case AList.lookup (=) trans (q, c, d) of
        NONE => raise Match
      | SOME (l, g) =>
          if matches-bound v t andalso not (contains-var v u) then mk (v, T) t u P
        else if matches-bound v u andalso not (contains-var v t) then mk (v, T) t u P
        else raise Match)
  | _ => raise Match));
in [tr' All-binder, tr' Ex-binder] end

3.8 Transitivity reasoning
context ord
begin

lemma ord-le-eq-trans: a ≤ b ==> b = c ==> a ≤ c
  by (rule subst)

lemma ord-eq-le-trans: a = b ==> b ≤ c ==> a ≤ c
  by (rule ssubst)

lemma ord-less-eq-trans: a < b ==> b = c ==> a < c

by (rule subst)

lemma ord-eq-less-trans: \( a = b \implies b < c \implies a < c \)
by (rule sssubst)

end

lemma order-less-subst2: (\( a::'a::order \)) < b ==> f b < (c::'c::order) ==> 
(\( !x. x < y ==> f x < f y \)) ==> f a < c
proof -
  assume r: \( !x. x < y ==> f x < f y \)
  assume a < b hence f a < f b by (rule r)
also assume f b < c
finally (less-trans) show \( \text{thesis} \).
qed

lemma order-less-subst1: (\( a::'a::order \)) < f b ==> (b::'b::order) < c ==> 
(\( !x. x < y ==> f x < f y \)) ==> a < f c
proof -
  assume r: \( !x. x < y ==> f x < f y \)
  assume a < f b
also assume b < c hence f b < f c by (rule r)
finally (less-trans) show \( \text{thesis} \).
qed

lemma order-le-less-subst2: (\( a::'a::order \)) <= b ==> f b < (c::'c::order) ==> 
(\( !x. x <= y ==> f x <= f y \)) ==> f a < c
proof -
  assume r: \( !x. x <= y ==> f x <= f y \)
  assume a <= b hence f a <= f b by (rule r)
also assume f b < c
finally (le-less-trans) show \( \text{thesis} \).
qed

lemma order-le-less-subst1: (\( a::'a::order \)) <= f b ==> (b::'b::order) < c ==> 
(\( !x. x < y ==> f x < f y \)) ==> a < f c
proof -
  assume r: \( !x. x < y ==> f x < f y \)
  assume a <= f b
also assume b < c hence f b < f c by (rule r)
finally (le-less-trans) show \( \text{thesis} \).
qed

lemma order-less-le-subst2: (\( a::'a::order \)) < b == f b <= (c::'c::order) ==> 
(\( !x. x < y ==> f x < f y \)) ==> f a < c
proof -
  assume r: \( !x. x < y ==> f x < f y \)
  assume a < b hence f a < f b by (rule r)
also assume f b <= c
finaly (less-le-trans) show ?thesis .

qed

lemma order-less-le-subst1: (a::'a::order) < f b ==> (b::'b::order) <= c ==> (! x y. x <= y ==> f x <= f y) ==> a < f c

proof –
  assume r: ! x y. x <= y ==> f x <= f y
  assume a < f b
  also assume b <= c hence f b <= f c by (rule r)
  finally (less-le-trans) show ?thesis .

qed

lemma order-subst1: (a::'a::order) <= f b ==> (b::'b::order) <= c ==> (! x y. x <= y ==> f x <= f y) ==> a <= f c

proof –
  assume r: ! x y. x <= y ==> f x <= f y
  assume a <= f b
  also assume b <= c hence f b <= f c by (rule r)
  finally (order-trans) show ?thesis .

qed

lemma order-subst2: (a::'a::order) <= b ==> f b <= c ==> (! x y. x <= y ==> f x <= f y) ==> f a <= f c

proof –
  assume r: ! x y. x <= y ==> f x <= f y
  assume a <= b hence f a <= f b by (rule r)
  also assume f b <= c
  finally (order-trans) show ?thesis .

qed

lemma ord-le-eq-subst: a <= b ==> f b = c ==> (! x y. x <= y ==> f x <= f y) ==> f a <= c

proof –
  assume r: ! x y. x <= y ==> f x <= f y
  assume a <= b hence f a <= f b by (rule r)
  also assume f b = c

qed

lemma ord-eq-le-subst: a = f b ==> b <= c ==> (! x y. x <= y ==> f x <= f y) ==> a <= f c

proof –
  assume r: ! x y. x <= y ==> f x <= f y
  assume a = f b
  also assume b <= c hence f b <= f c by (rule r)

qed

lemma ord-less-eq-subst: a < b ==> f b = c ==>
THEORY "Orderings"

(!!(x y. x < y ==> f x < f y) ==> f a < c)

proof -
  assume r: !!(x y. x < y ==> f x < f y)
  assume a < b hence f a < f b by (rule r)
  also assume f b = c
  finally (ord-less-eq-trans) show ?thesis .
qed

lemma ord-eq-less-subst: a = f b ==> b < c ==> (!!(x y. x < y ==> f x < f y) ==> a < f c)

proof -
  assume r: !!(x y. x < y ==> f x < f y)
  assume a = f b
  also assume b < c hence f b < f c by (rule r)
  finally (ord-eq-less-trans) show ?thesis .
qed

Note that this list of rules is in reverse order of priorities.

lemmas [trans] =
  order-less-subst2
  order-less-subst1
  order-le-less-subst2
  order-le-less-subst1
  order-less-le-subst2
  order-less-le-subst1
  order-subst2
  order-subst1
  order-le-eq-subst
  ord-eq-le-subst
  ord-less-eq-subst
  ord-eq-less-subst
  forw-subst
  back-subst
  rev-mp
  mp

lemmas (in order) [trans] =
  neq-le-trans
  le-neq-trans

lemmas (in preorder) [trans] =
  less-trans
  less-asym'
  le-less-trans
  less-le-trans
  order-trans

lemmas (in order) [trans] =
  antisym
lemmas (in ord) [trans] =
ord-le-eq-trans
ord-eq-le-trans
ord-less-eq-trans
ord-eq-less-trans

lemmas [trans] =
trans

lemmas order-trans-rules =
order-less-subst2
order-less-subst1
order-le-less-subst2
order-le-less-subst1
order-less-le-subst2
order-less-le-subst1
order-subst2
order-subst1
ord-le-eq-subst
ord-eq-le-subst
ord-less-eq-subst
ord-eq-less-subst
forw-subst
back-subst
rev-mp
mp
neq-le-trans
le-neq-trans
less-trans
less-asym'
le-less-trans
less-le-trans
order-trans
antisym
ord-le-eq-trans
ord-eq-le-trans
ord-less-eq-trans
ord-eq-less-trans
trans

These support proving chains of decreasing inequalities $a \preceq b \preceq c \ldots$ in Isar proofs.

lemma xtl [no-atp]:
\[ a = b \Rightarrow b > c \Rightarrow a > c \]
\[ a > b \Rightarrow b = c \Rightarrow a > c \]
\[ a = b \Rightarrow b \geq c \Rightarrow a \geq c \]
\[ a \geq b \Rightarrow b = c \Rightarrow a \geq c \]
\[ (x::'a::order) \geq y \Rightarrow y \geq x \Rightarrow x = y \]
lemmas

THEORY "Orderings" 94

(x::'a::order) ≥ y ==> y ≥ z ==> x ≥ z
(x::'a::order) > y ==> y ≥ z ==> x > z
(x::'a::order) ≥ y ==> y > z ==> x > z
(a::'a::order) > b ==> b > a ==> P
(x::'a::order) > y ==> y > z ==> x > z
(a::'a::order) ≥ b ==> a ≠ b ==> a > b
(a::'a::order) ≠ b ==> a ≥ b ==> a > b

a = f b ==> b > c ==> (∀x y. x > y ==> f x > f y) ==> a > f c
a > b ==> f b = c ==> (∀x y. x > y ==> f x > f y) ==> f a > c
a = f b ==> b ≥ c ==> (∀x y. x ≥ y ==> f x ≥ f y) ==> a ≥ f c
a ≥ b ==> f b = c ==> (∀x y. x ≥ y ==> f x ≥ f y) ==> f a ≥ c

by auto

lemma xt2 [no-atp]:
  (a::'a::order) >= f b ==> b >= c ==> (!!x y. x >= y ==> f x >= f y) ==> a >= f c
by (subgoal-tac f b >= f c, force, force)

lemma xt3 [no-atp]: (a::'a::order) >= b ==> (f b::'b::order) >= c ==> (!!x. x >= y ==> f x >= f y) ==> a >= f c
by (subgoal-tac f a >= f b, force, force)

lemma xt4 [no-atp]: (a::'a::order) > f b ==> (b::'b::order) >= c ==> (!!x. x >= y ==> f x >= f y) ==> a > f c
by (subgoal-tac f b > f c, force, force)

lemma xt5 [no-atp]: (a::'a::order) > b ==> (f b::'b::order) >= c ==> (!!x. x >= y ==> f x >= f y) ==> f a > c
by (subgoal-tac f a > f b, force, force)

lemma xt6 [no-atp]: (a::'a::order) >= f b ==> b > c ==> (!!x. x > y ==> f x > f y) ==> a > f c
by (subgoal-tac f b > f c, force, force)

lemma xt7 [no-atp]: (a::'a::order) > b ==> (f b::'b::order) > c ==> (!!x. x > y ==> f x > f y) ==> f a > c
by (subgoal-tac f a > f b, force, force)

lemma xt8 [no-atp]: (a::'a::order) > f b ==> (b::'b::order) > c ==> (!!x. x > y ==> f x > f y) ==> a > f c
by (subgoal-tac f b > f c, force, force)

lemma xt9 [no-atp]: (a::'a::order) > b ==> (f b::'b::order) > c ==> (!!x. x > y ==> f x > f y) ==> f a > c
by (subgoal-tac f a > f b, force, force)

lemmas xtrans = xt1 xt2 xt3 xt4 xt5 xt6 xt7 xt8 xt9
3.9 Monotonicity

context order

begin

definition mono :: ('a ⇒ 'b::order) ⇒ bool where
  mono f ⟷ (∀ x y. x ≤ y ⟹ f x ≤ f y)

lemma monoI [intro?]:
  fixes f :: 'a ⇒ 'b::order
  shows (∀ x y. x ≤ y ⟹ f x ≤ f y) ⟹ mono f
  unfolding mono-def by iprover

lemma monoD [dest?] :
  fixes f :: 'a ⇒ 'b::order
  shows mono f ⟹ x ≤ y ⟹ f x ≤ f y
  unfolding mono-def by iprover

lemma monoE:
  fixes f :: 'a ⇒ 'b::order
  assumes mono f
  assumes x ≤ y
  obtains f x ≤ f y
  proof
  from assms show f x ≤ f y by (simp add: mono-def)
  qed

definition antimono :: ('a ⇒ 'b::order) ⇒ bool where
  antimono f ⟷ (∀ x y. x ≤ y ⟹ f x ≥ f y)

lemma antimonoI [intro?]:
  fixes f :: 'a ⇒ 'b::order
  shows (∀ x y. x ≤ y ⟹ f x ≥ f y) ⟹ antimono f
  unfolding antimono-def by iprover

lemma antimonoD [dest?] :
  fixes f :: 'a ⇒ 'b::order
  shows antimono f ⟹ x ≤ y ⟹ f x ≥ f y
  unfolding antimono-def by iprover

lemma antimonoE:
  fixes f :: 'a ⇒ 'b::order
  assumes antimono f
  assumes x ≤ y
  obtains f x ≥ f y
  proof
  from assms show f x ≥ f y by (simp add: antimono-def)
  qed

definition strict-mono :: ('a ⇒ 'b::order) ⇒ bool where
THEORY “Orderings”

strict-mono \( f \leftrightarrow (\forall x y. x < y \rightarrow f x < f y) \)

lemma strict-monoI [intro?]:
assumes \( \forall x y. x < y \Rightarrow f x < f y \)
shows strict-mono \( f \)
using assms unfolding strict-mono-def by auto

lemma strict-monoD [dest?]:
strict-mono \( f \Rightarrow x < y \Rightarrow f x < f y \)
unfolding strict-mono-def by auto

lemma strict-mono-mono [dest?]:
assumes strict-mono \( f \)
shows mono \( f \)
proof (rule monoI)
fix \( x \ y \)
assume \( x \leq y \)
show \( f x \leq f y \)
proof (cases \( x = y \))
case True then show ?thesis by simp
next
case False with \( x \leq y \) have \( x < y \) by simp
with assms strict-monoD have \( f x < f y \) by auto
then show ?thesis by simp
qed
qed

end

context linorder
begin

lemma mono-invE:
fixes \( f : 'a \Rightarrow 'b : order \)
assumes mono \( f \)
assumes \( f x < f y \)
obtains \( x \leq y \)
proof
show \( x \leq y \)
proof (rule ccontr)
assume \( \neg x \leq y \)
then have \( y \leq x \) by simp
with \( \neg mono \ f \) obtain \( f y \leq f x \) by (rule monoE)
with \( f x < f y \) show False by simp
qed
qed

lemma mono-strict-invE:
fixes \( f : 'a \Rightarrow 'b : order \)
assumes \( \text{mono } f \)
assumes \( f \ x < f \ y \)
obtains \( x < y \)
proof
show \( x < y \)
proof (rule ccontr)
  assume \( \neg x < y \)
  then have \( y \leq x \) by simp
  with \( \text{mono } f \) obtain \( f \ y \leq f \ x \) by (rule monoE)
  with \( f \ x < f \ y \) show False by simp
qed

definition (in Orderings)
lemma strict-mono-eq:
assumes strict-mono \( f \)
shows \( f \ x = f \ y \longleftrightarrow x = y \)
proof
assume \( f \ x = f \ y \)
show \( x = y \)
proof (cases \( x \ y \) rule: linorder-cases)
  case \( \text{less} \) with assms strict-monoD have \( f \ x < f \ y \) by auto
  with \( f \ x = f \ y \) show ?thesis by simp
next
  case equal then show ?thesis.
next
  case \( \text{greater} \) with assms strict-monoD have \( f \ y < f \ x \) by auto
  with \( f \ x = f \ y \) show ?thesis by simp
qed
qed simp

lemma strict-mono-less-eq:
assumes strict-mono \( f \)
shows \( f \ x \leq f \ y \longleftrightarrow x \leq y \)
proof
assume \( x \leq y \)
with assms strict-mono monoD show \( f \ x \leq f \ y \) by auto
next
assume \( f \ x \leq f \ y \)
show \( x \leq y \)
proof (rule ccontr)
  assume \( \neg x \leq y \) then have \( y < x \) by simp
  with assms strict-monoD have \( f \ y < f \ x \) by auto
  with \( f \ x \leq f \ y \) show False by simp
qed
qed

lemma strict-mono-less:
assumes strict-mono \( f \)
shows \( f \ x < f \ y \longleftrightarrow x < y \)
using assms
by (auto simp add: less-le Orderings.less-le strict-mono-eq strict-mono-less-eq)
THEORY "Orderings"

end

3.10 min and max – fundamental

definition (in ord) min :: 'a ⇒ 'a ⇒ 'a where
\( \text{min } a \; b = (\text{if } a \leq b \text{ then } a \text{ else } b) \)

definition (in ord) max :: 'a ⇒ 'a ⇒ 'a where
\( \text{max } a \; b = (\text{if } a \leq b \text{ then } b \text{ else } a) \)

lemma min-absorb1: \( x \leq y \implies \text{min } x \; y = x \)
by (simp add: min-def)

lemma max-absorb2: \( x \leq y \implies \text{max } x \; y = y \)
by (simp add: max-def)

lemma min-absorb2: \( y : 'a :: \text{order} \leq x \implies \text{min } x \; y = y \)
by (simp add: min-def)

lemma max-absorb1: \( y : 'a :: \text{order} \leq x \implies \text{max } x \; y = x \)
by (simp add: max-def)

lemma max-min-same [simp]:
\[
\text{fixes } x \; y : 'a :: \text{linorder}
\]
\[
\text{shows } \text{max } x \; (\text{min } x \; y) = x \; \text{max } (\text{min } x \; y) \; x = x \; \text{max } (\text{min } x \; y) \; y = y \; \text{max } y
\]
by(auto simp add: max-def min-def)

3.11 (Unique) top and bottom elements

class bot =
    fixes bot :: 'a (\bot)

class order-bot = order + bot +
    assumes bot-least: \bot \leq a
begin

sublocale bot: ordering-top greater-eq greater bot
    by standard (fact bot-least)

lemma le-bot:
\( a \leq \bot \implies a = \bot \)
by (fact bot.extremum-uniqueI)

lemma bot-unique:
\( a \leq \bot \iff a = \bot \)
by (fact bot.extremum-unique)

lemma not-less-bot:
THEORY “Orderings”

¬a<⊥
by (fact bot.extremum-strict)

lemma bot-less:
a≠⊥↔⊥<a
by (fact bot.not-eq-extremum)

lemma max-bot[simp]: max bot x = x
by (simp add: max-def bot-unique)

lemma max-bot2[simp]: max x bot = x
by (simp add: max-def bot-unique)

lemma min-bot[simp]: min bot x = bot
by (simp add: min-def bot-unique)

lemma min-bot2[simp]: min x bot = bot
by (simp add: min-def bot-unique)

deductivemethod end

class top =
  fixes top :: 'a (⊤)

class order-top = order + top +
  assumes top-greatest: a ≤ ⊤
begin

subclass top: ordering-top less-eq less top
  by standard (fact top-greatest)

lemma top-le:
  ⊤ ≤ a ⇒ a = ⊤
by (fact top.extremum-uniqueI)

lemma top-unique:
  ⊤ ≤ a ⇒ a = ⊤
by (fact top.extremum-unique)

lemma not-top-less:
  ¬⊤<a
by (fact top.extremum-strict)

lemma less-top:
  a≠⊤↔a<⊤
by (fact top.not-eq-extremum)

lemma max-top[simp]: max top x = top
by (simp add: max-def top-unique)
lemma max-top2[simp]: max x top = top
by(simp add: max-def top-unique)

lemma min-top[simp]: min x x = x
by(simp add: min-def top-unique)

lemma min-top2[simp]: min x top = x
by(simp add: min-def top-unique)

end

3.12 Dense orders

class dense-order = order +
  assumes dense: x < y ⇒ (∃z. x < z ∧ z < y)

class dense-linorder = linorder + dense-order
begin

lemma dense-le:
  fixes y z :: 'a
  assumes \(\forall x. x < y \Rightarrow x \leq z\)
  shows y \leq z
proof (rule contr)
  assume ¬ ?thesis
  hence z < y by simp
  from dense[OF this]
  obtain x where x < y and z < x by safe
  moreover have x \leq z using assms[OF ⟨x < y⟩].
  ultimately show False by auto
qed

lemma dense-le-bounded:
  fixes x y z :: 'a
  assumes x < y
  assumes *: \(\forall w. \[ x < w ; w < y \] \Rightarrow w \leq z\)
  shows y \leq z
proof (rule dense-le)
  fix w assume w < y
  from dense[OF ⟨x < y⟩] obtain u where x < u u < y by safe
  from linear[of u w]
  show w \leq z
  proof (rule dense-le)
    assume u \leq w
    from less-le-trans[OF ⟨x < w ; u \leq w⟩] ⟨w < y⟩
    show w \leq z by (rule *)
  next
    assume w \leq u
  qed

from \( w \leq w \ast [OF (x < w) (u < y)] \)
show \( w \leq z \) by (rule order-trans)
Qed
Qed

lemma dense-ge:
fixes \( y \), \( z :: 'a \)
assumes \( \forall x. z < x \Longrightarrow y \leq x \)
shows \( y \leq z \)
proof (rule ccontr)
assume \(~ \theta \)
hence \( z < y \) by simp
from dense [OF this]
obtain \( x \) where \( x < y \) and \( z < x \) by safe
moreover have \( y \leq x \) using assms [OF \( z < x \)] .
ultimately show \( \text{False} \) by auto
Qed

lemma dense-ge-bounded:
fixes \( x, y, z :: 'a \)
assumes \( z < x \)
assumes \( \ast : \forall w. \left[ z < w ; w < x \right] \Longrightarrow y \leq w \)
shows \( y \leq z \)
proof (rule dense-ge)
fix \( w \)
assume \( z < w \)
from dense [OF \( z < x \)] obtain \( u \) where \( z < u \) \( u < x \) by safe
from linear [of \( u \) \( w \)]
show \( y \leq w \)
proof (rule disjE)
assume \( w \leq u \)
from \( z < w \) le-less-trans [OF \( w \leq u \) \( u < x \)]
show \( y \leq w \) by (rule \( \ast \))
next
assume \( u \leq w \)
from \( \ast [OF (z < w) (u < x)] \) \( u \leq w \)
show \( y \leq w \) by (rule order-trans)
Qed
Qed

end

class no-top = order +
assumes gt-ex: \( \exists y. x < y \)

class no-bot = order +
assumes lt-ex: \( \exists y. y < x \)

class unbounded-dense-linorder = dense-linorder + no-top + no-bot
3.13 Wellorders

class wellorder = linorder +
  assumes less-induct [case-names less]: \( (\forall x. (\forall y. y < x \implies P y) \implies P x) \implies P a \)
begin

lemma wellorder-Least-lemma:
  fixes k :: 'a
  assumes P k
  shows LeastI: \( P (LEAST x. P x) \) and Least-le: \( (LEAST x. P x) \leq k \)
proof
  have \( P (LEAST x. P x) \land (LEAST x. P x) \leq k \)
  using assms proof (induct k rule: less-induct)
  case (less x) then have \( P x \)
  by simp
  show ?case proof (rule classical)
  assume assm: \( \neg (P (LEAST a. P a) \land (LEAST a. P a) \leq x) \)
  have \( \forall y. P y \implies x \leq y \)
  proof (rule classical)
    fix y
    assume P y and x \leq y
    with less have \( P (LEAST a. P a) \) and \( (LEAST a. P a) \leq y \)
    by (auto simp add: not-le)
    with assm have \( x < (LEAST a. P a) \) and \( (LEAST a. P a) \leq y \)
    by auto
    then show \( x \leq y \)
    by auto
    qed
  qed
  with \( P x \) have Least: \( (LEAST a. P a) = x \)
  by (rule Least-equality)
  with \( P x \) show ?thesis by simp
  qed
  qed
  then show \( P (LEAST x. P x) \) and \( (LEAST x. P x) \leq k \)
  by auto
  qed

— The following 3 lemmas are due to Brian Huffman

lemma LeastI-ex: \( \exists x. P x \implies P (Least P) \)
  by (erule exE) (erule LeastI)

lemma LeastI2:
  \( P a \implies (\forall x. P x \implies Q x) \implies Q (Least P) \)
  by (blast intro: LeastI)

lemma LeastI2-ex:
  \( \exists a. P a \implies (\forall x. P x \implies Q x) \implies Q (Least P) \)
  by (blast intro: LeastI-ex)

lemma LeastI2-wellorder:
  assumes P a
  and \( \forall a. \ [ P a; \forall b. P b \implies a \leq b \] \implies Q a \)
shows $Q \ (\text{Least } P)$

proof (rule LeastI2-order)

show $P \ (\text{Least } P)$ using $\langle P \ a \rangle$ by (rule LeastI)

next

fix $y$ assume $P \ y$ thus $\text{Least } P \leq y$ by (rule Least-le)

next

fix $x$ assume $P \ x \ \forall \ y. \ P \ y \rightarrow x \leq y$ thus $Q \ x$ by (rule assms(2))

qed

lemma LeastI2-wellorder-ex:

assumes $\exists \ x. \ P \ x$

and $\forall \ a. \ [ P \ a; \ \forall \ b. \ P \ b \rightarrow a \leq b ] \rightarrow Q \ a$

shows $Q \ (\text{Least } P)$

using assms by clarify (blast intro: LeastI2-wellorder)

lemma not-less-Least: $k < (\text{LEAST } x. \ P \ x) \rightarrow \neg P \ k$

apply (simp add: not-le [symmetric])

apply (erule contrapos-nn)

apply (erule Least-le)

done

lemma exists-least-iff: $(\exists \ n. \ P \ n) \longleftrightarrow (\exists \ n. \ P \ n \land (\forall \ m < n. \ \neg P \ m))$ (is $?\text{lhs} \longleftrightarrow ?\text{rhs}$)

proof

assume $?\text{rhs}$ thus $?\text{lhs}$ by blast

next

assume $H$: $?\text{lhs}$ then obtain $n$ where $n: P \ n$ by blast

let $?x = \text{Least } P$

{ fix $m$ assume $m: m < ?x$

  from not-less-Least[OF $m$] have $\neg P \ m$ . }

with LeastI-ex[OF $H$] show $?\text{rhs}$ by blast

qed

end

3.14 Order on bool

instantiation bool :: \{order-bot, order-top, linorder\}

begin

definition le-boole-def [simp]: $P \leq Q \longleftrightarrow P \rightarrow Q$

definition [simp]: $(P::\text{bool}) < Q \longleftrightarrow \neg P \land Q$

definition [simp]: $\bot \longleftrightarrow \text{False}$
definition
[simp]: \top \iff \text{True}

instance proof
qed auto
end

lemma le-boolI: \ (P \to Q) \to P \leq Q
by simp

lemma le-boolI': P \to Q \to P \leq Q
by simp

lemma le-boolE: P \leq Q \to P \to (Q \to R) \to R
by simp

lemma le-boolD: P \leq Q \to P \to Q
by simp

lemma bot-boolE: \bot \to P
by simp

lemma top-boolI: \top
by simp

lemma [code]:
False \leq b \iff \text{True}
True \leq b \iff b
False < b \iff b
True < b \iff \text{False}
by simp-all

3.15 Order on - \to -

instantiation fun :: (type, ord) ord
begin

definition
le-fun-def: f \leq g \iff (\forall x. f x \leq g x)

definition
(f::'a \Rightarrow 'b) < g \iff f \leq g \& \& (g \leq f)

instance ..
end

instance fun :: (type, preorder) preorder proof
qed (auto simp add: le-fun-def less-fun-def intro: order-trans antisym)

instance fun :: (type, order) order proof
qed (auto simp add: le-fun-def intro: antisym)

instantiation fun :: (type, bot) bot begin

definition ⊥ = (λx. ⊥)

instance ..
end

instantiation fun :: (type, order-bot) order-bot begin

lemma bot-apply [simp, code]:
⊥ x = ⊥
by (simp add: bot-fun-def)

instance proof
qed (simp add: le-fun-def)
end

instantiation fun :: (type, top) top begin

definition [no-atp]: ⊤ = (λx. ⊤)

instance ..
end

instantiation fun :: (type, order-top) order-top begin

lemma top-apply [simp, code]:
⊤ x = ⊤
by (simp add: top-fun-def)

instance proof
qed (simp add: le-fun-def)
end
lemma le-funI: \( (\forall x. f x \leq g x) \implies f \leq g \)
unfolding le-fun-def by simp

lemma le-funE: \( f \leq g \implies (f x \leq g x \implies P) \implies P \)
unfolding le-fun-def by simp

lemma le-funD: \( f \leq g \implies f x \leq g x \)
by (rule le-funE)

lemma mono-compose: mono \( Q \implies mono \ (\lambda x. Q i (f x)) \)
unfolding mono-def le-fun-def by auto

3.16 Order on unary and binary predicates

lemma predicate1I:
assumes \( PQ: \forall x. P x \implies Q x \)
shows \( P \leq Q \)
apply (rule le-funI)
apply (rule le-boolI)
apply (rule PQ)
apply assumption
done

lemma predicate1D:
\( P \leq Q \implies P x \implies Q x \)
apply (erule le-funE)
apply (erule le-boolE)
apply assumption+
done

lemma rev-predicate1D:
\( P x \implies P \leq Q \implies Q x \)
by (rule predicate1D)

lemma predicate2I:
assumes \( PQ: \forall x y. P x y \implies Q x y \)
shows \( P \leq Q \)
apply (rule le-funI)+
apply (rule le-boolI)
apply (rule PQ)
apply assumption
done

lemma predicate2D:
\( P \leq Q \implies P x y \implies Q x y \)
apply (erule le-funE)+
apply (erule le-boolE)
apply assumption+
done

**lemma** rev-predicate2D:

\[ P \ x \ y \implies P \leq Q \implies Q \ x \ y \]

by (rule predicate2D)

**lemma** bot1E [no-atp]: \[ \bot \ x \implies P \]

by (simp add: bot-fun-def)

**lemma** bot2E: \[ \bot \ x \ y \implies P \]

by (simp add: bot-fun-def)

**lemma** top1I: \[ \top \ x \]

by (simp add: top-fun-def)

**lemma** top2I: \[ \top \ x \ y \]

by (simp add: top-fun-def)

### 3.17 Name duplicates

**lemmas** order-eq-refl = preorder-class.eq-refl

**lemmas** order-less-irrefl = preorder-class.less-irrefl

**lemmas** order-less-imp-le = preorder-class.less-imp-le

**lemmas** order-less-not-sym = preorder-class.less-not-sym

**lemmas** order-less-asym = preorder-class.less-asym

**lemmas** order-less-trans = preorder-class.less-trans

**lemmas** order-le-less-trans = preorder-class.le-less-trans

**lemmas** order-less-le-trans = preorder-class.less-le-trans

**lemmas** order-less-imp-not-less = preorder-class.less-imp-not-less

**lemmas** order-less-imp-triv = preorder-class.less-imp-triv

**lemmas** order-less-asym' = preorder-class.less-asym'

**lemmas** order-less-le = order-class.less-le

**lemmas** order-le-less = order-class.le-less

**lemmas** order-le-imp-less-or-eq = order-class.le-imp-less-or-eq

**lemmas** order-less-imp-not-eq = order-class.less-imp-not-eq

**lemmas** order-less-imp-not-eq2 = order-class.less-imp-not-eq2

**lemmas** order-neq-le-trans = order-class.neq-le-trans

**lemmas** order-le-neq-trans = order-class.le-neq-trans

**lemmas** order-antisym = order-class.antisym

**lemmas** order-eq-iff = order-class.eq-iff

**lemmas** order-antisym-conv = order-class.antisym-conv

**lemmas** linorder-linear = linorder-class.linear

**lemmas** linorder-less-linear = linorder-class.less-linear

**lemmas** linorder-le-less-linear = linorder-class.le-less-linear

**lemmas** linorder-le-cases = linorder-class.le-cases

**lemmas** linorder-not-less = linorder-class.not-less

**lemmas** linorder-not-le = linorder-class.not-le
4 Groups, also combined with orderings

theory Groups
  imports Orderings
begin

4.1 Dynamic facts

named-theorems ac-simps associativity and commutativity simplification rules
  and algebra-simps algebra simplification rules for rings
  and algebra-split-simps algebra simplification rules for rings, with potential goal splitting
  and field-simps algebra simplification rules for fields
  and field-split-simps algebra simplification rules for fields, with potential goal splitting

The rewrites accumulated in algebra-simps deal with the classical algebraic structures of groups, rings and family. They simplify terms by multiplying everything out (in case of a ring) and bringing sums and products into a canonical form (by ordered rewriting). As a result it decides group and ring equalities but also helps with inequalities.

Of course it also works for fields, but it knows nothing about multiplicative inverses or division. This is catered for by field-simps.

Facts in field-simps multiply with denominators in (in)equations if they can be proved to be non-zero (for equations) or positive/negative (for inequalities). Can be too aggressive and is therefore separate from the more benign algebra-simps.

Collections algebra-split-simps and field-split-simps correspond to algebra-simps and field-simps but contain more aggressive rules that may lead to goal splitting.

4.2 Abstract structures

These locales provide basic structures for interpretation into bigger structures; extensions require careful thinking, otherwise undesired effects may occur due to interpretation.

locale semigroup =
  fixes f :: 'a ⇒ 'a ⇒ 'a (infixl ∗ 70)
  assumes assoc [ac-simps]: a ∗ b ∗ c = a ∗ (b ∗ c)
locale abel-semigroup = semigroup +
  assumes commute [ac-simps]: \( a * b = b * a \)
begin

lemma left-commute [ac-simps]: \( b * (a * c) = a * (b * c) \)
proof
  have \((b * a) * c = (a * b) * c\)
    by (simp only: commute)
  then show \(?thesis\)
    by (simp only: assoc)
qed

end

locale monoid = semigroup +
  fixes \( z : 'a (1) \)
  assumes left-neutral [simp]: \( 1 * a = a \)
  assumes right-neutral [simp]: \( a * 1 = a \)

locale comm-monoid = abel-semigroup +
  fixes \( z : 'a (1) \)
  assumes comm-neutral: \( a * 1 = a \)
begin

sublocale monoid
  by standard (simp-all add: commute comm-neutral)
end

locale group = semigroup +
  fixes \( z : 'a (1) \)
  fixes inverse :: 'a ⇒ 'a
  assumes group-left-neutral: \( 1 * a = a \)
  assumes left-inverse [simp]: inverse a * a = 1
begin

lemma left-cancel: \( a * b = a * c \iff b = c \)
proof
  assume \( a * b = a * c \)
  then have \( inverse a * (a * b) = inverse a * (a * c) \)
    by simp
  then have \( (inverse a * a) * b = (inverse a * a) * c \)
    by (simp only: assoc)
  then show \( b = c \)
    by (simp add: group-left-neutral)
qed simp

sublocale monoid
proof
  fix \( a \)
  have \( inverse a * a = 1 \)
    by simp
then have \( \text{inverse } a \ast (a \ast 1) = \text{inverse } a \ast a \)
by (simp add: group-left-neutral assoc [symmetric])
with left-cancel show \( a \ast 1 = a \)
by (simp only: left-cancel)
qed (fact group-left-neutral)

lemma inverse-unique:
assumes \( a \ast b = 1 \)
shows \( \text{inverse } a = b \)
proof
from assms have \( \text{inverse } a \ast (a \ast b) = \text{inverse } a \)
by simp
then show \( ?\text{thesis} \)
by (simp add: assoc [symmetric])
qed

lemma inverse-neutral [simp]: \( \text{inverse } 1 = 1 \)
by (rule inverse-unique) simp

lemma inverse-inverse [simp]: \( \text{inverse } (\text{inverse } a) = a \)
by (rule inverse-unique) simp

lemma right-inverse [simp]: \( a \ast \text{inverse } a = 1 \)
proof
have \( a \ast \text{inverse } a = \text{inverse } (\text{inverse } a) \ast \text{inverse } a \)
by simp
also have \( \ldots \ast 1 \)
by (rule left-inverse)
then show \( ?\text{thesis} \) by simp
qed

lemma inverse-distrib-swap: \( \text{inverse } (a \ast b) = \text{inverse } b \ast \text{inverse } a \)
proof (rule inverse-unique)
have \( a \ast b \ast (\text{inverse } b \ast \text{inverse } a) = a \ast (b \ast \text{inverse } b) \ast \text{inverse } a \)
by (simp only: assoc)
also have \( \ldots = 1 \)
by simp
finally show \( a \ast b \ast (\text{inverse } b \ast \text{inverse } a) = 1 \).
qed

lemma right-cancel: \( b \ast a = c \ast a \iff b = c \)
proof
assume \( b \ast a = c \ast a \)
then have \( b \ast a \ast \text{inverse } a = c \ast a \ast \text{inverse } a \)
by simp
then show \( b = c \)
by (simp add: assoc)
qed simp
4.3 Generic operations

class zero =
  fixes zero :: 'a (0)

class one =
  fixes one :: 'a (1)

hide-const (open) zero one

lemma Let-0 [simp]: Let 0 f = f 0
  unfolding Let-def ..

lemma Let-1 [simp]: Let 1 f = f 1
  unfolding Let-def ..

setup :
  Reorient-Proc.add
    (fn Const(const-name⟨Groups.zero⟩, _) => true
     | Const(const-name⟨Groups.one⟩, _) => true
     | _ => false)

simproc-setup reorient-zero (0 = x) = Reorient-Proc.proc
simproc-setup reorient-one (1 = x) = Reorient-Proc.proc

typed-print-translation :
  let
    fun tr' c = (c, fn ctxt => fn T => fn ts =>
      if null ts andalso Printer.type-emphasis ctxt T then
        Syntax_const syntax-const ⟨-constrain⟩ $ Syntax.const c $ Syntax-Phases.term-of-typ ctxt T
      else raise Match);
  in map tr' [const-syntax⟨Groups.one⟩, const-syntax⟨Groups.zero⟩] end

— show types that are presumably too general

class plus =
  fixes plus :: 'a ⇒ 'a ⇒ 'a (infixl + 65)

class minus =
  fixes minus :: 'a ⇒ 'a ⇒ 'a (infixl - 65)

class uminus =
  fixes uminus :: 'a ⇒ 'a (- - [81] 80)

class times =
fixes times :: 'a ⇒ 'a ⇒ 'a  (infixl * 70)

4.4 Semigroups and Monoids

class semigroup-add = plus +
assumes add-assoc [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
(a + b) + c = a + (b + c)
begin
sublocale add: semigroup plus
  by standard (fact add-assoc)
end

hide-fact add-assoc

class ab-semigroup-add = semigroup-add +
assumes add-commute [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
a + b = b + a
begin
sublocale add: abel-semigroup plus
  by standard (fact add-commute)
declare add.left-commute [algebra-simps, algebra-split-simps, field-simps, field-split-simps]
lemmas add-ac = add.assoc add.commute add.left-commute

end

hide-fact add-commute

lemmas add-ac = add.assoc add.commute add.left-commute

class semigroup-mult = times +
assumes mult-assoc [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
(a * b) * c = a * (b * c)
begin
sublocale mult: semigroup times
  by standard (fact mult-assoc)
end

hide-fact mult-assoc

class ab-semigroup-mult = semigroup-mult +
assumes mult-commute [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
a * b = b * a
begin

sublocale mult: abel-semigroup times
  by standard (fact mult-commute)

declare mult.left-commute [algebra-simps, algebra-split-simps, field-simps, field-split-simps]

lemmas mult-ac = mult.assoc mult.commute mult.left-commute

end

hide-fact mult-commute

lemmas mult-ac = mult.assoc mult.commute mult.left-commute

class monoid-add = zero + semigroup-add +
  assumes add-0-left: 0 + a = a
    and add-0-right: a + 0 = a
begin

subclass add: monoid plus 0
  by standard (fact add-0-left add-0-right)+

end

lemma zero-reorient: 0 = x ↔ x = 0
  by (fact eq-commute)

class comm-monoid-add = zero + ab-semigroup-add +
  assumes add-0: 0 + a = a
begin

subclass monoid-add
  by standard (simp-all add: add-0 add.commute [of - 0])

subclass add: comm-monoid plus 0
  by standard (simp add: ac-simps)

end

class monoid-mult = one + semigroup-mult +
  assumes mult-1-left: 1 * a = a
    and mult-1-right: a * 1 = a
begin

subclass mult: monoid times 1
  by standard (fact mult-1-left mult-1-right)+

end
lemma one-reorient: \(1 = x \iff x = 1\)
by (fact eq-commute)

class comm-monoid-mult = one + ab-semigroup-mult +
assumes mult-1: \(1 \ast a = a\)
begin

subclass monoid-mult
by standard (simp-all add: mult-1 mult.commute [of - 1])

sublocale mult: comm-monoid times 1
by standard (simp add: ac-simps)

end

class cancel-semigroup-add = semigroup-add +
assumes add-left-imp-eq: \(a + b = a + c \implies b = c\)
assumes add-right-imp-eq: \(b + a = c + a \implies b = c\)
begin

lemma add-left-cancel [simp]: \(a + b = a + c \iff b = c\)
by (blast dest: add-left-imp-eq)

lemma add-right-cancel [simp]: \(b + a = c + a \iff b = c\)
by (blast dest: add-right-imp-eq)

end

class cancel-ab-semigroup-add = ab-semigroup-add + minus +
assumes diff-diff-cancel-left'[simp]: \((a + b) - a = b\)
assumes diff-diff-add [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
\(a - b - c = a - (b + c)\)
begin

lemma add-diff-cancel-right'[simp]: \((a + b) - b = a\)
using add-diff-cancel-left'[of b a] by (simp add: ac-simps)

subclass cancel-semigroup-add
proof
fix \(a \ b \ c:: 'a\)
assume \(a + b = a + c\)
then have \(a + b - a = a + c - a\)
  by simp
then show \(b = c\)
  by simp
next
fix \(a \ b \ c:: 'a\)
assume \(b + a = c + a\)
then have \( b + a - a = c + a - a \)
  by simp
then show \( b = c \)
  by simp

qed

lemma add-diff-cancel-left [simp]: \((c + a) - (c + b) = a - b\)
  unfolding diff-diff-add [symmetric] by simp

lemma add-diff-cancel-right [simp]: \((a + c) - (b + c) = a - b\)
  using add-diff-cancel-left [symmetric] by (simp add: ac-simps)

lemma diff-right-commute: \(a - c - b = a - b - c\)
  by (simp add: diff-diff-add add.commute)

end

class cancel-comm-monoid-add = cancel-ab-semigroup-add + comm-monoid-add
begin

lemma diff-zero [simp]: \(a - 0 = a\)
  using add-diff-cancel-right' [of a 0] by simp

lemma diff-cancel [simp]: \(a - a = 0\)
  proof
    have \((a + 0) - (a + 0) = 0\)
      by (simp only: add-diff-cancel-left diff-zero)
    then show ?thesis by simp
  qed

lemma add-implies-diff:
  assumes \(c + b = a\)
  shows \(c = a - b\)
  proof
    from assms have \((b + c) - (b + 0) = a - b\)
      by (simp add: add.commute)
    then show \(c = a - b\) by simp
  qed

lemma add-cancel-right-right [simp]: \(a = a + b \iff b = 0\)
  (is \(?P \iff ?Q\))
  proof
    assume \(?Q\)
    then show \(?P\) by simp
  next
    assume \(?P\)
    then have \(a - a = a + b - a\) by simp
    then show \(?Q\) by simp
  qed
lemma add-cancel-right-left [simp]: \( a = b + a \iff b = 0 \)
using add-cancel-right-right [of a b] by (simp add: ac-simps)

lemma add-cancel-left-right [simp]: \( a + b = a \iff b = 0 \)
by (auto dest: sym)

lemma add-cancel-left-left [simp]: \( b + a = a \iff b = 0 \)
by (auto dest: sym)

end

class comm-monoid-diff = cancel-comm-monoid-add +
assumes zero-diff [simp]: \( 0 - a = 0 \)
begin

lemma diff-add-zero [simp]: \( a - (a + b) = 0 \)
proof 
  have \( a - (a + b) = (a + 0) - (a + b) \)
  by simp
  also have \( \ldots = 0 \)
  by (simp only: add-diff-cancel-left zero-diff)
  finally show \( ?thesis \).
  qed

end

4.5 Groups

class group-add = minus + uminus + monoid-add +
assumes left-minus: \( - a + a = 0 \)
assumes add-uminus-conv-diff [simp]: \( a + (- b) = a - b \)
begin

lemma diff-conv-add-uminus: \( a - b = a + (- b) \)
by simp

sublocale add: group plus 0 uminus
by standard (simp-all add: left-minus)

lemma minus-unique: \( a + b = 0 \implies - a = b \)
by (fact add.inverse-unique)

lemma minus-zero: \( - 0 = 0 \)
by (fact add.inverse-neutral)

lemma minus-minus: \( - (- a) = a \)
by (fact add.inverse-inverse)
THEORY “Groups”

lemma right-minus: \( a + - a = 0 \)
    by (fact add.right-inverse)

lemma diff-self [simp]: \( a - a = 0 \)
    using right-minus [of a] by simp

subclass cancel-semigroup-add
    by standard (simp-all add: add.left-cancel add.right-cancel)

lemma minus-add-cancel [simp]: \( -a + (a + b) = b \)
    by (simp add: add.assoc [symmetric])

lemma add-minus-cancel [simp]: \( a + (-a + b) = b \)
    by (simp only: diff-cone-add-uminus add.assoc) simp

lemma diff-add-cancel [simp]: \( a - b + b = a \)
    by (simp only: diff-cone-add-uminus add.assoc) simp

lemma add-diff-cancel [simp]: \( a + b - b = a \)
    by (simp only: diff-cone-add-uminus add.assoc) simp

lemma minus-add: \( -a + b - b + a = \)
    by (fact add.inverse-distrib-swap)

lemma right-minus-eq [simp]: \( a - b = 0 \iff a = b \)
    proof
        assume \( a - b = 0 \)
        have \( a = (a - b) + b \) by (simp add: add.assoc)
        also have \( ... = b \) using \( a - b = 0 \) by simp
        finally show \( a = b \).
    next
        assume \( a = b \)
        then show \( a - b = 0 \) by simp
    qed

lemma eq-iff-diff-eq-0: \( a = b \iff a - b = 0 \)
    by (fact right-minus-eq [symmetric])

lemma diff-0 [simp]: \( 0 - a = -a \)
    by (simp only: diff-cone-add-uminus add-0-left)

lemma diff-0-right [simp]: \( a - 0 = a \)
    by (simp only: diff-cone-add-uminus minus-zero add-0-right)

lemma diff-minus-eq-add [simp]: \( a - b = a + b \)
    by (simp only: diff-cone-add-uminus minus-minus)

lemma neg-equal-iff-equal [simp]: \( -a = -b \iff a = b \)
    proof
assume $-a = -b$
then have $-(a) = -(b)$ by simp
then show $a = b$ by simp
next
assume $a = b$
then show $-a = -b$ by simp
qed

lemma neg-equal-0-iff-equal [simp]: $-a = 0$ $\iff$ $a = 0$
  by (subst neg-equal-iff-equal [symmetric]) simp

lemma neg-0-equal-iff-equal [simp]: $0 = -a$ $\iff$ $0 = a$
  by (subst neg-equal-iff-equal [symmetric]) simp

The next two equations can make the simplifier loop!

lemma equation-minus-iff: $a = -b$ $\iff$ $b = -a$
proof
  have $- (a) = -b$ $\iff$ $a = b$
    by (rule neg-equal-iff-equal)
  then show ?thesis
    by (simp add: eq-commute)
qed

lemma minus-equation-iff: $-a = b$ $\iff$ $-b = a$
proof
  have $a = - (b)$ $\iff$ $a = -b$
    by (rule neg-equal-iff-equal)
  then show ?thesis
    by (simp add: eq-commute)
qed

lemma eq-neg-iff-add-eq-0: $a = -b$ $\iff$ $a + b = 0$
proof
  assume $a = -b$
  then show $a + b = 0$ by simp
next
assume $a + b = 0$
moreover have $a + (b + -b) = (a + b) + -b$
  by (simp only: add.assoc)
ultimately show $a = -b$
  by simp
qed

lemma add-eq-0-iff2: $a + b = 0$ $\iff$ $a = -b$
  by (fact eq-neg-iff-add-eq-0 [symmetric])

lemma neg-eq-iff-add-eq-0: $- a = b$ $\iff$ $a + b = 0$
  by (auto simp add: add-eq-0-iff2)
lemma add-eq-0-iff: \( a + b = 0 \iff b = -a \)
by (auto simp add: neg-eq-iff-add-eq-0 [symmetric])

lemma minus-diff-eq [simp]: \( -(a - b) = b - a \)
by (simp only: neg-eq-iff-add-eq-0 diff-conv-add-uminus add.assoc minus-add-cancel)

simp

lemma add-diff-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
\( a + (b - c) = (a + b) - c \)
by (simp only: add.assoc)

lemma diff-add-eq-diff-diff-swap:
\( a - (b + c) = a - c - b \)
by (simp only: diff-conv-add-uminus add.assoc minus-add-cancel)

lemma diff-eq-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
\( a - b = c \iff a = c + b \)
by auto

lemma diff-eq-diff-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
\( a - b = c - d \implies a = b \iff c = d \)
by (simp only: eq-iff-diff-eq-0 [of a b] eq-iff-diff-eq-0 [of c d])

end

class ab-group-add = minus + uminus + comm-monoid-add +
assumes ab-left-minus: \( -a + a = 0 \)
assumes ab-diff-conv-uminus: \( a - b = a + (-b) \)
begin
subclass group-add
by standard (simp-all add: ab-left-minus ab-diff-conv-uminus)
subclass cancel-comm-monoid-add
proof
fix \( a \ b \ c :: 'a \)
have \( b + a - a = b \)
  by simp
then show \( a + b - a = b \)
  by (simp add: ac-simps)
show \( a - b - c = a - (b + c) \)
  by (simp add: algebra-simps)
qed
lemma uminus-add-conv-diff [simp]: \(- a + b = b - a\)
  by (simp add: add.commute)

lemma minus-add-distrib [simp]: \(- (a + b) = - a + - b\)
  by (simp add: algebra-simps)

lemma diff-add-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]:
  \((a - b) + c = (a + c) - b\)
  by (simp add: algebra-simps)

lemma minus-diff-commute:
  \(- b - a = - a - b\)
  by (simp only: diff-cone-add-uminus add.commute)
end

4.6 (Partially) Ordered Groups

The theory of partially ordered groups is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society, 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press, 1963

Most of the used notions can also be looked up in

- [http://www.mathworld.com](http://www.mathworld.com) by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer

class ordered-ab-semigroup-add = order + ab-semigroup-add +
  assumes add-left-mono: \(a \leq b \implies c + a \leq c + b\)
begnin

lemma add-right-mono: \(a \leq b \implies a + c \leq b + c\)
  by (simp add: add.commute [of - c] add-left-mono)

non-strict, in both arguments

lemma add-mono: \(a \leq b \implies c \leq d \implies a + c \leq b + d\)
  apply (erule add-right-mono [THEN order-trans])
  apply (simp add: add.commute add-left-mono)
  done

end

Strict monotonicity in both arguments
class strict-ordered-ab-semigroup-add = ordered-ab-semigroup-add +
assumes add-strict-mono: \(a < b \implies c < d \implies a + c < b + d\)

class ordered-cancel-ab-semigroup-add =
  ordered-ab-semigroup-add + cancel-ab-semigroup-add
begin

lemma add-strict-left-mono: \(a < b \implies c + a < c + b\)
  by (auto simp add: less-le add-left-mono)

lemma add-strict-right-mono: \(a < b \implies a + c < b + c\)
  by (simp add: add.commute [of - c] add-strict-left-mono)

subclass strict-ordered-ab-semigroup-add
  apply standard
  apply (erule add-strict-right-mono [THEN less-trans])
  apply (erule add-left-mono)
  done

lemma add-less-le-mono: \(a < b \implies c \leq d \implies a + c < b + d\)
  apply (erule add-strict-right-mono [THEN less-le-trans])
  apply (erule add-left-mono)
  done

lemma add-le-less-mono: \(a \leq b \implies c < d \implies a + c < b + d\)
  apply (erule add-right-mono [THEN le-less-trans])
  apply (erule add-strict-left-mono)
  done

end

class ordered-ab-semigroup-add-imp-le = ordered-cancel-ab-semigroup-add +
assumes add-le-imp-le-left: \(c + a \leq c + b \implies a \leq b\)
begin

lemma add-less-imp-less-left:
  assumes less: \(c + a < c + b\)
  shows \(a < b\)
proof –
  from less have le: \(c + a \leq c + b\)
    by (simp add: order-le-less)
  have \(a \leq b\)
    apply (insert le)
    apply (erule add-le-imp-le-left)
    apply (insert le)
    apply (erule add-le-imp-le-left)
    apply assumption
    done
  moreover have \(a \neq b\)
proof (rule ccontr)
  assume ¬ thesis
  then have a = b by simp
  then have c + a = c + b by simp
  with less show False by simp
qed
ultimately show a < b
  by (simp add: order-le-less)
qed

lemma add-less-imp-less-right: a + c < b + c ⇒ a < b
by (rule add-less-imp-less-left [of c]) (simp add: add.commute)

lemma add-less-cancel-left [simp]: c + a < c + b ←→ a < b
by (blast intro: add-less-imp-less-left add-strict-left-mono)

lemma add-less-cancel-right [simp]: a + c < b + c ←→ a < b
by (blast intro: add-less-imp-less-right add-strict-right-mono)

lemma add-le-cancel-left [simp]: c + a ≤ c + b ←→ a ≤ b
apply auto
  apply (drule add-le-imp-le-left)
  apply (simp-all add: add-left-mono)
done

lemma add-le-cancel-right [simp]: a + c ≤ b + c ←→ a ≤ b
by (simp add: add.commute [of a c] add.commute [of b c])

lemma add-le-imp-le-right: a + c ≤ b + c ⇒ a ≤ b
by simp

lemma max-add-distrib-left: max x y + z = max (x + z) (y + z)
unfolding max-def by auto

lemma min-add-distrib-left: min x y + z = min (x + z) (y + z)
unfolding min-def by auto

lemma max-add-distrib-right: x + max y z = max (x + y) (x + z)
unfolding max-def by auto

lemma min-add-distrib-right: x + min y z = min (x + y) (x + z)
unfolding min-def by auto
end

4.7 Support for reasoning about signs

class ordered-comm-monoid-add = comm-monoid-add + ordered-ab-semigroup-add
begin
lemma add-nonneg-nonneg [simp]: \(0 \leq a \Rightarrow 0 \leq b \Rightarrow 0 \leq a + b\)
using add-mono[of \(0\) \(a\) \(0\) \(b\)] by simp

lemma add-nonpos-nonpos: \(a \leq 0 \Rightarrow b \leq 0 \Rightarrow a + b \leq 0\)
using add-mono[of \(a\) \(0\) \(b\) \(0\)] by simp

lemma add-nonneg-eq-0-iff: \(0 \leq x \Rightarrow 0 \leq y \Rightarrow x + y = 0 \iff x = 0 \land y = 0\)
using add-left mono[of \(0\) \(y\) \(x\)] add-right mono[of \(0\) \(x\) \(y\)] by auto

lemma add-nonpos-eq-0-iff: \(x \leq 0 \Rightarrow y \leq 0 \Rightarrow x + y = 0 \iff x = 0 \land y = 0\)
using add-left mono[of \(y\) \(0\) \(x\)] add-right mono[of \(0\) \(x\) \(y\)] by auto

lemma add-increasing: \(0 \leq a \Rightarrow b \leq c \Rightarrow b \leq a + c\)
using add-mono[of \(0\) \(a\) \(b\) \(c\)] by simp

lemma add-increasing2: \(0 \leq c \Rightarrow b \leq a \Rightarrow b \leq a + c\)
by (simp add: add-increasing add commute[of \(a\)])

lemma add-decreasing: \(a \leq 0 \Rightarrow c \leq b \Rightarrow a + c \leq b\)
using add-mono[of \(a\) \(0\) \(c\) \(b\)] by simp

lemma add-decreasing2: \(c \leq 0 \Rightarrow a \leq b \Rightarrow a + c \leq b\)
using add-mono[of \(a\) \(b\) \(c\) \(0\)] by simp

lemma add-pos-nonneg: \(0 < a \Rightarrow 0 \leq b \Rightarrow 0 < a + b\)
using less-le-trans[of \(0\) \(a\) \(a + b\)] by (simp add: add-increasing2)

lemma add-pos-pos: \(0 < a \Rightarrow 0 < b \Rightarrow 0 < a + b\)
by (intro add-pos-nonneg less-imp-le)

lemma add-nonneg-pos: \(0 < a \Rightarrow 0 < b \Rightarrow 0 < a + b\)
using add-pos-nonneg[of \(b\) \(a\)] by (simp add: add-commute)

lemma add-neg-nonneg: \(a < 0 \Rightarrow b \leq 0 \Rightarrow a + b < 0\)
using le-less-trans[of \(a\) \(a + b\) \(0\)] by (simp add: add-decreasing2)

lemma add-neg-neg: \(a < 0 \Rightarrow b < 0 \Rightarrow a + b < 0\)
by (intro add-neg-nonpos less-imp-le)

lemma add-nonpos-neg: \(a \leq 0 \Rightarrow b < 0 \Rightarrow a + b < 0\)
using add-neg-nonpos[of \(b\) \(a\)] by (simp add: add-commute)

lemmas add-sign-intros =
add-pos-nonneg add-pos-pos add-nonneg-pos add-nonneg-nonneg
add-neg-nonneg add-neg-pos add-neg-neg add-nonneg-pos add-nonpos-neg add-nonpos-pos
end

class strict-ordered-comm-monoid-add = comm-monoid-add + strict-ordered-ab-semigroup-add
begin

lemma pos-add-strict: \( 0 < a \Rightarrow b < c \Rightarrow b < a + c \)
  using add-strict-mono \([\text{of } 0 \ a \ b \ c]\) by simp

end

class ordered-cancel-comm-monoid-add = ordered-comm-monoid-add + cancel-ab-semigroup-add
begin

subclass ordered-cancel-ab-semigroup-add ..
subclass strict-ordered-comm-monoid-add ..

lemma add-strict-increasing: \( 0 < a \Rightarrow b \leq c \Rightarrow b < a + c \)
  using add-less-le-mono \([\text{of } 0 \ a \ b \ c]\) by simp

lemma add-strict-increasing2: \( 0 \leq a \Rightarrow b < c \Rightarrow b < a + c \)
  using add-le-less-mono \([\text{of } 0 \ a \ b \ c]\) by simp

end

class ordered-ab-semigroup-monoid-add-imp-le = monoid-add + ordered-ab-semigroup-add-imp-le
begin

lemma add-less-same-cancel1 \([\text{simp}]: b + a < b \iff a < 0 \)
  using add-less-cancel-left \([\text{of } -] \) by simp

lemma add-less-same-cancel2 \([\text{simp}]: a + b < b \iff a < 0 \)
  using add-less-cancel-right \([\text{of } 0] \) by simp

lemma less-add-same-cancel1 \([\text{simp}]: a < a + b \iff 0 < b \)
  using add-less-cancel-left \([\text{of } 0] \) by simp

lemma less-add-same-cancel2 \([\text{simp}]: a < b + a \iff 0 < b \)
  using add-less-cancel-right \([\text{of } 0] \) by simp

lemma add-le-same-cancel1 \([\text{simp}]: b + a \leq b \iff a \leq 0 \)
  using add-le-cancel-left \([\text{of } -] \) by simp

lemma add-le-same-cancel2 \([\text{simp}]: a + b \leq b \iff a \leq 0 \)
  using add-le-cancel-right \([\text{of } 0] \) by simp

lemma le-add-same-cancel1 \([\text{simp}]: a \leq a + b \iff 0 \leq b \)
  using add-le-cancel-left \([\text{of } 0] \) by simp

lemma le-add-same-cancel2 \([\text{simp}]: a \leq b + a \iff 0 \leq b \)
using add-le-cancel-right [of 0] by simp

subclass cancel-comm-monoid-add
  by standard auto

subclass ordered-cancel-comm-monoid-add
  by standard
end

class ordered-ab-group-add = ab-group-add + ordered-ab-semigroup-add
begin

subclass ordered-cancel-ab-semigroup-add ..

subclass ordered-ab-semigroup-monoid-add-imp-le
proof
  fix a b c :: 'a
  assume c + a ≤ c + b
  then have (−c) + (c + a) ≤ (−c) + (c + b)
    by (rule add-left-mono)
  then have (((−c) + c) + a ≤ ((−c) + c) + b
    by (simp only: add.assoc)
  then show a ≤ b by simp
qed

lemma max-diff-distrib-left: max x y − z = max (x − z) (y − z)
  using max-add-distrib-left [of x y − z] by simp

lemma min-diff-distrib-left: min x y − z = min (x − z) (y − z)
  using min-add-distrib-left [of x y − z] by simp

lemma le-imp-neg-le:
  assumes a ≤ b
  shows − b ≤ − a
proof –
  from assms have − a + a ≤ − a + b
    by (rule add-left-mono)
  then have 0 ≤ − a + b
    by simp
  then have 0 + (− b) ≤ (− a + b) + (− b)
    by (rule add-right-mono)
  then show ?thesis
    by (simp add: algebra-simps)
qed

lemma neg-le-iff-le [simp]: − b ≤ − a ↔ a ≤ b
proof
  assume − b ≤ − a
then have $- ( - a ) \leq - ( - b )$
  by (rule le-imp-neg-le)
then show $a \leq b$
  by simp
next
  assume $a \leq b$
  then show $- b \leq - a$
    by (rule le-imp-neg-le)
qed

lemma neg-le-0-iff-le [simp]: $- a \leq 0 \iff 0 \leq a$
  by (subst neg-le-iff-le [symmetric]) simp

lemma neg-0-le-iff-le [simp]: $0 \leq - a \iff a \leq 0$
  by (subst neg-le-iff-le [symmetric]) simp

lemma neg-less-iff-less [simp]: $- b < - a \iff a < b$
  by (auto simp add: less-le)

lemma neg-less-0-iff-less [simp]: $0 < - a \iff a < 0$
  by (subst neg-less-iff-less [symmetric]) simp

lemma neg-0-less-iff-less [simp]: $0 < - a \iff a < 0$
  by (subst neg-less-iff-less [symmetric]) simp

The next several equations can make the simplifier loop!

lemma less-minus-iff: $a < - b \iff b < - a$
proof -
  have $- ( - a ) < - b \iff b < - a$
    by (rule neg-less-iff-less)
  then show ?thesis by simp
qed

lemma minus-less-iff: $- a < b \iff - b < a$
proof -
  have $- a < - ( - b ) \iff - b < a$
    by (rule neg-less-iff-less)
  then show ?thesis by simp
qed

lemma le-minus-iff: $a \leq - b \iff b \leq - a$
proof -
  have $mm: - ( - a ) < - b \Rightarrow - ( - b ) < - a$
    for $a \ b :: 'a$
    by (simp only: minus-less-iff)
  have $- ( - a ) \leq - b \iff b \leq - a$
    apply (auto simp only: le-less)
    apply (drule mm)
    apply (simp-all)
    apply (drule mm[simplified], assumption)
  qed
done
then show \(?\text{thesis}\) by simp
qed

lemma \textit{minus-le-iff}: \(-a \leq b \iff -b \leq a\)
by (auto simp add: le-less minus-less-iff)

lemma \textit{diff-less-0-iff-less [simp]}: \(a - b < 0 \iff a < b\)
proof
  have \(a - b < 0 \iff a + (-b) < b + (-b)\)
  by simp
also have \(\ldots \iff a < b\)
  by (simp only: add-less-cancel-right)
finally show \(?\text{thesis}\).
qed

lemmas \textit{less-iff-diff-less-0} = diff-less-0-iff-less [symmetric]

lemma \textit{diff-less-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]}: \(a - b < c \iff a < c + b\)
apply (subt less-iff-diff-less-0 [of a])
apply (rule less-iff-diff-less-0 [of - c, \text{THEN ssubst}])
apply (simp add: algebra-simps)
done

lemma \textit{less-diff-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]}: \(a < c - b \iff a + b < c\)
apply (subt less-iff-diff-less-0 [of a + b])
apply (subt less-iff-diff-less-0 [of a])
apply (simp add: algebra-simps)
done

lemma \textit{diff-gt-0-iff-gt [simp]}: \(a - b > 0 \iff a > b\)
by (simp add: less-diff-eq)

lemma \textit{diff-le-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]}: \(a - b \leq c \iff a \leq c + b\)
by (auto simp add: le-less diff-less-eq)

lemma \textit{le-diff-eq [algebra-simps, algebra-split-simps, field-simps, field-split-simps]}: \(a \leq c - b \iff a + b \leq c\)
by (auto simp add: le-less less-diff-eq)

lemma \textit{diff-le-0-iff-le [simp]}: \(a - b \leq 0 \iff a \leq b\)
by (simp add: algebra-simps)

lemmas \textit{le-iff-diff-le-0} = diff-le-0-iff-le [symmetric]

lemma \textit{diff-ge-0-iff-ge [simp]}: \(a - b \geq 0 \iff a \geq b\)
by (simp add: le-diff-eq)

**lemma** diff-eq-diff-less: \( a - b = c - d \implies a < b \iff c < d \)
by (auto simp only: less_iff_diff_less_0 [of a b] less_iff_diff_less_0 [of c d])

**lemma** diff-eq-diff-less-eq: \( a - b = c - d \implies a \leq b \iff c \leq d \)
by (auto simp only: le_iff_diff_le_0 [of a b] le_iff_diff_le_0 [of c d])

**lemma** diff-mono: \( a \leq b \implies d \leq c \implies a - c \leq b - d \)
by (simp add: field_simps add_mono)

**lemma** diff-left-mono: \( b \leq a \implies c - a \leq c - b \)
by (simp add: field_simps)

**lemma** diff-right-mono: \( a \leq b \implies a - c \leq b - c \)
by (simp add: field_simps)

**lemma** diff-strict-mono: \( a < b \implies d < c \implies a - c < b - d \)
by (simp add: field_simps add_strict_mono)

**lemma** diff-strict-left-mono: \( b < a \implies c - a < c - b \)
by (simp add: field_simps)

**lemma** diff-strict-right-mono: \( a < b \implies a - c < b - c \)
by (simp add: field_simps)

end

locale group-cancel
begin

**lemma** add1: \( A::\alpha::comm_monoid_add \equiv k + a \implies A + b \equiv k + (a + b) \)
by (simp only: ac_simps)

**lemma** add2: \( B::\alpha::comm_monoid_add \equiv k + b \implies a + B \equiv k + (a + b) \)
by (simp only: ac_simps)

**lemma** sub1: \( A::\alpha::ab_group_add \equiv k + a \implies A - b \equiv k + (a - b) \)
by (simp only: add_diff_eq)

**lemma** sub2: \( B::\alpha::ab_group_add \equiv k + b \implies a - B \equiv - k + (a - b) \)
by (simp only: minus_add_diff_conv_add_uminus ac_simps)

**lemma** neg1: \( A::\alpha::ab_group_add \equiv k + a \implies - A \equiv - k + - a \)
by (simp only: minus_add_distrib)

**lemma** rule0: \( a::\alpha::comm_monoid_add \equiv a + 0 \)
by (simp only: add_0_right)
ML-file (Tools/group-cancel.ML):

simproc-setup group-cancel-add \( (a + b :: 'a :: ab-group-add) = \)
\( \langle fn \, phi \Rightarrow fn \, ss \Rightarrow \text{try Group-Cancel.cancel-add-conv} \rangle \)

simproc-setup group-cancel-diff \( (a - b :: 'a :: ab-group-add) = \)
\( \langle fn \, phi \Rightarrow fn \, ss \Rightarrow \text{try Group-Cancel.cancel-diff-conv} \rangle \)

simproc-setup group-cancel-eq \( (a = (b :: 'a :: ab-group-add)) = \)
\( \langle fn \, phi \Rightarrow fn \, ss \Rightarrow \text{try Group-Cancel.cancel-eq-conv} \rangle \)

simproc-setup group-cancel-le \( (a \leq (b :: 'a :: ordered-ab-group-add)) = \)
\( \langle fn \, phi \Rightarrow fn \, ss \Rightarrow \text{try Group-Cancel.cancel-le-conv} \rangle \)

simproc-setup group-cancel-less \( (a < (b :: 'a :: ordered-ab-group-add)) = \)
\( \langle fn \, phi \Rightarrow fn \, ss \Rightarrow \text{try Group-Cancel.cancel-less-conv} \rangle \)

class linordered-ab-semigroup-add =
\( \text{linorder + ordered-ab-semigroup-add} \)

class linordered-cancel-ab-semigroup-add =
\( \text{linorder + ordered-cancel-ab-semigroup-add} \)

begin

subclass linordered-ab-semigroup-add ..

subclass ordered-ab-semigroup-add-imp-le

proof
  fix \( a \, b \, c :: 'a \)
  assume \( \text{le1: } c + a \leq c + b \)
  show \( a \leq b \)
  proof (rule ccontr)
    assume \( \ast : \neg ?\text{thesis} \)
    then have \( b \leq a \) by (simp add: linorder-not-le)
    then have \( c + b \leq c + a \) by (rule add-left-mono)
    with \( \text{le1} \) have \( a = b \)
    apply
    apply (erule antisym)
    apply simp-all
    done
    with \( \ast \) show False
    by (simp add: linorder-not-le [symmetric])
  qed

qed

end
class linordered-ab-group-add = linorder + ordered-ab-group-add
begin

subclass linordered-cancel-ab-semigroup-add ..

lemma equal-neg-zero [simp]: a = − a ↔ a = 0
proof
  assume a = 0
  then show a = − a by simp
next
  assume A: a = − a
  show a = 0
  proof (cases 0 ≤ a)
    case True
    with A have 0 ≤ − a by auto
    with le-minus-iff have a ≤ 0 by simp
    with True show ?thesis by (auto intro: order-trans)
  next
    case False
    then have B: a ≤ 0 by auto
    with A have − a ≤ 0 by auto
    with B show ?thesis by (auto intro: order-trans)
  qed
qed

lemma neg-equal-zero [simp]: − a = a ↔ a = 0
by (auto dest: sym)

lemma neg-less-eq-nonneg [simp]: − a ≤ a ↔ 0 ≤ a
proof
  assume *: − a ≤ a
  show 0 ≤ a
  proof (rule classical)
    assume ¬ ?thesis
    then have a < 0 by auto
    with * have − a < 0 by (rule le-less-trans)
    then show ?thesis by auto
  qed
next
  assume *: 0 ≤ a
  then have − a ≤ 0 by (simp add: minus-le-iff)
  from this * show − a ≤ a by (rule order-trans)
qed

lemma neg-less-pos [simp]: − a < a ↔ 0 < a
by (auto simp add: less-le)

lemma less-eq-neg-nonpos [simp]: a ≤ − a ↔ a ≤ 0
using neg-less-eq-nonneg [of − a] by simp
lemma less-neg-neg [simp]: \( a < -a \iff a < 0 \)
  using neg-less-pos [of \(-a\)] by simp

lemma double-zero [simp]: \( a + a = 0 \iff a = 0 \)
proof
  assume \( a + a = 0 \)
  then have \( -a = a \) by (rule minus-unique)
  then show \( a = 0 \) by (simp only: neg-equal-zero)
next
  assume \( a = 0 \)
  then show \( a + a = 0 \) by simp
qed

lemma double-zero-sym [simp]: \( 0 = a + a \iff a = 0 \)
apply (rule iffI)
  apply (drule sym)
  apply simp-all
done

lemma zero-less-double-add-iff-zero-less-single-add [simp]: \( 0 < a + a \iff 0 < a \)
proof
  assume \( 0 < a + a \)
  then have \( 0 - a < a \) by (simp only: diff-less-eq)
  then have \( -a < a \) by simp
  then show \( 0 < a \) by simp
next
  assume \( 0 < a \)
  with this have \( 0 + 0 < a + a \)
    by (rule add-strict-mono)
  then show \( 0 < a + a \) by simp
qed

lemma zero-le-double-add-iff-zero-le-single-add [simp]: \( 0 \leq a + a \iff 0 \leq a \)
proof
  have \( \neg a + a < 0 \iff \neg a < 0 \)
    by (simp add: not-less)
  then show ?thesis by simp
qed

lemma double-add-less-zero-iff-single-add-less-zero [simp]: \( a + a < 0 \iff a < 0 \)
proof
  have \( \neg a + a < 0 \iff \neg a < 0 \)
    by (simp add: not-less)
  then show ?thesis by simp
qed

lemma double-add-le-zero-iff-single-add-le-zero [simp]: \( a + a \leq 0 \iff a \leq 0 \)
proof
  have \( \neg a + a \leq 0 \iff \neg a \leq 0 \)
    by (simp add: not-le)
  then show ?thesis by simp
qed
lemma minus-max-eq-min: \(-\max x y = \min (-x) (-y)\)
by (auto simp add: max-def min-def)

lemma minus-min-eq-max: \(-\min x y = \max (-x) (-y)\)
by (auto simp add: max-def min-def)

end

class abs =
fixes abs :: 'a ⇒ 'a (| - |)

class sgn =
fixes sgn :: 'a ⇒ 'a

class ordered-ab-group-add-abs = ordered-ab-group-add + abs +
assumes abs-ge-zero [simp]: \(|a| \geq 0\)
and abs-ge-self: \(a \leq |a|\)
and abs-leI: \(a \leq b \implies -a \leq b \implies |a| \leq b\)
and abs-minus-cancel [simp]: \(|-a| = |a|\)
and abs-triangle-ineq: \(|a + b| \leq |a| + |b|\)
begin

lemma abs-minus-le-zero: \(-|a| \leq 0\)
unfolding neg-le-0-iff-le by simp

lemma abs-of-nonneg [simp]:
assumes nonneg: \(0 \leq a\)
shows \(|a| = a\)
proof (rule antisym)
show \(a \leq |a|\) by (rule abs-ge-self)
from nonneg le-imp-neg-le have \(-a \leq 0\) by simp
from this nonneg have \(-a \leq a\) by (rule order-trans)
then show \(|a| \leq a\) by (auto intro: abs-leI)
qed

lemma abs-idempotent [simp]: \(||a|| = |a|\)
by (rule antisym) (auto intro!: abs-leI order-trans[of \(-|a|\) 0 |a|])

lemma abs-eq-0 [simp]: \(|a| = 0 \iff a = 0\)
proof -
have \(|a| = 0 \implies a = 0\)
proof (rule antisym)
assume zero: \(|a| = 0\)
with abs-ge-self show \(a \leq 0\) by auto
from zero have \(|-a| = 0\) by simp
with abs-ge-self [of \(-a\)] have \(-a \leq 0\) by auto
with neg-le-0-iff-le show \(0 \leq a\) by auto
qed
then show \( \text{thesis by auto} \)

qed

lemma abs-zero \[ simp \]: \(|0| = 0 \)
  by simp

lemma abs-0-eq \[ simp \]: \(0 = |a| \iff a = 0 \)
proof
  have \(0 = |a| \iff |a| = 0\) by (simp only: eq-ac)
  then show \( \text{thesis by simp}\)
qed

lemma abs-le-zero-iff \[ simp \]: \(|a| \leq 0 \iff a = 0 \)
proof
  assume \( |a| \leq 0 \)
  then have \(|a| = 0 \implies 0 \leq a \)
    using order.trans by blast
  then show \( a = 0 \) by simp
next
  assume \( a = 0 \)
  then show \( |a| \leq 0 \) by simp
qed

lemma abs-le-self-iff \[ simp \]: \(|a| \leq a \iff 0 \leq a \)
proof
  have \(0 \leq |a| \)
    using abs-ge-zero by blast
  then have \(|a| \leq a \implies 0 \leq a \)
    using order.trans by blast
  then show \( \text{thesis} \)
    using abs-of-nonneg eq-refl by blast
qed

lemma zero-less-abs-iff \[ simp \]: \(0 < |a| \iff a \neq 0 \)
by (simp add: less-le)

lemma abs-not-less-zero \[ simp \]: \(\neg \ |a| < 0 \)
proof
  have \(x \leq y \implies \neg\ y < x\) for \(x \ y\) by auto
  then show \( \text{thesis by simp} \)
qed

lemma abs-ge-minus-self: \(- a \leq |a| \)
proof
  have \(- a \leq |a|\) by (rule abs-ge-self)
  then show \( \text{thesis by simp} \)
qed

lemma abs-minus-commute: \(|a - b| = |b - a| \)
proof

have \(|a - b| = |-(a - b)|\)
by (simp only: abs-minus-cancel)
also have \(\ldots = |b - a|\) by simp
finally show \(?thesis\).
qed

lemma abs-of-pos: \(0 < a \implies |a| = a\)
by (rule abs-of-nonpos) (rule less-imp-le)

lemma abs-of-nonpos [simp]:
assumes \(a \leq 0\)
shows \(|a| = -a\)
proof –
let \(?b = -a\)
have \(-?b \leq 0 \implies |-(?b) = -(?-b)\)
unfolding abs-minus-cancel [of \(?b\)]
unfolding neg-le-0-iff-le [of \(?b\)]
unfolding minus-minus by (rule abs-of-nonneg)
then show \(?thesis\) using assms by auto
qed

lemma abs-of-neg: \(a < 0 \implies |a| = -a\)
by (rule abs-of-nonpos) (rule less-imp-le)

lemma abs-le-D1: \(|a| \leq b \implies a \leq b\)
using abs-ge-self by (blast intro: order-trans)

lemma abs-le-D2: \(|a| \leq b \implies -a \leq b\)
using abs-le-D1 [of \(-a\)] by simp

lemma abs-le-iff: \(|a| \leq b \iff a \leq b \land -a \leq b\)
by (blast intro: abs-leI dest: abs-le-D1 abs-le-D2)

lemma abs-triangle-ineq2: \(|a| - |b| \leq |a - b|\)
proof –
have \(|a| = |b + (a - b)|\)
by (simp add: algebra-simps)
then have \(|a| \leq |b| + |a - b|\)
by (simp add: abs-triangle-ineq)
then show \(?thesis\)
by (simp add: algebra-simps)
qed

lemma abs-triangle-ineq2-sym: \(|a| - |b| \leq |b - a|\)
by (simp only: abs-minus-commute [of \(b\)] abs-triangle-ineq2)

lemma abs-triangle-ineq3: \(||a| - |b|| \leq |a - b|\)
by (simp add: abs-le-iff abs-triangle-ineq2 abs-triangle-ineq2-sym)
lemma abs-triangle-ineq4: \(|a - b| \leq |a| + |b|

proof –
  have \(|a - b| = |a + - b|\)
  by (simp add: algebra-simps)
  also have \(\ldots \leq |a| + |- b|\)
  by (rule abs-triangle-ineq)
  finally show \(?\text{thesis}\) by simp
qed

lemma abs-diff-triangle-ineq: \(|a + b - (c + d)| \leq |a - c| + |b - d|\n
proof –
  have \(|a + b - (c + d)| = |(a - c) + (b - d)|\)
  by (simp add: algebra-simps)
  also have \(\ldots \leq |a - c| + |b - d|\)
  by (rule abs-triangle-ineq)
  finally show \(?\text{thesis}\).
qed

lemma abs-add-abs [simp]: ||a| + |b|| = |a| + |b|
(is \(?L = ?R\))

proof (rule antisym)
  show \(?L \geq ?R\) by (rule abs-ge-self)
  have \(?L \leq ||a|| + ||b||\) by (rule abs-triangle-ineq)
  also have \(\ldots = ?R\) by simp
  finally show \(?L \leq ?R\).
qed

end

4.8 Canonically ordered monoids

Canonically ordered monoids are never groups.
class canonically-ordered-monoid-add = comm-monoid-add + order +
  assumes le-iff-add: a ≤ b ←→ (∃ c. b = a + c)
begin

lemma zero-le[simp]: 0 ≤ x
  by (auto simp: le-iff-add)

lemma le-zero-eq[simp]: n ≤ 0 ←→ n = 0
  by (auto intro: antisym)

lemma not-less-zero[simp]: ¬ n < 0
  by (auto simp)

lemma zero-less-iff-neq-zero: 0 < n ←→ n ≠ 0
  by (auto simp)

This theorem is useful with blast

lemma gr-zeroI: (n = 0 ⟹ False) ⟹ 0 < n
  by (rule zero-less-iff-neq-zero[THEN iffD2]) iprover

lemma not-gr-zero[simp]: ¬ 0 < n ←→ n = 0
  by (simp add: zero-less-iff-neq-zero)

subclass ordered-comm-monoid-add
proof qed (auto simp: le-iff-add add-ac)

lemma gr-implies-not-zero: m < n ⟹ n ≠ 0
  by auto

lemma add-eq-0-iff-both-eq-0[simp]: x + y = 0 ←→ x = 0 ∧ y = 0
  by (intro add-nonneg-eq-0-iff zero-le)

lemma zero-eq-add-iff-both-eq-0[simp]: 0 = x + y ←→ x = 0 ∧ y = 0
  unfolding eq-commute[of 0]

lemma less-eqE:
  assumes ⟨a ≤ b⟩
  obtains c where ⟨b = a + c⟩
  using assms by (auto simp add: le-iff-add)

lemma lessE:
  assumes ⟨a < b⟩
  obtains c where ⟨b = a + c⟩ and ⟨c ≠ 0⟩
  proof –
  from assms have ⟨a ≤ b⟩ ⟨a ≠ b⟩
    by simp-all
  from ⟨a ≤ b⟩ obtain c where ⟨b = a + c⟩
    by (rule less-eqE)
  moreover have ⟨c ≠ 0⟩ using ⟨a ≠ b⟩ ⟨b = a + c⟩
by auto
ultimately show \(\text{thesis}\)
  by (rule that)
qed

lemmas zero-order = zero-le le-zero eq not-less-zero zero-less-iff-neq-zero not-gr-zero
— This should be attributed with \(\text{iff}\), but then blast fails in Set.

end

class ordered-cancel-comm-monoid-diff =
canonically-ordered-monoid-add + comm-monoid-diff + ordered-ab-semigroup-add-imp-le
begin

context
fixes a b :: 'a
assumes le: a \leq b
begin

lemma add-diff-inverse: \(a + (b - a) = b\)
  using le by (auto simp add: le-iff-add)

lemma add-diff-assoc: \(c + (b - a) = c + b - a\)
  using le by (auto simp add: le-iff-add add.left-commute [of c])

lemma add-diff-assoc2: \(b - a + c = b + c - a\)
  using le by (auto simp add: le-iff-add add.assoc)

lemma diff-add-assoc: \(c + b - a = c + (b - a)\)
  using le by (simp add: add.commute add-diff-assoc)

lemma diff-add-assoc2: \(b + c - a = b - a + c\)
  using le by (simp add: add.commute add-diff-assoc)

lemma diff-diff-right: \(c - (b - a) = c + a - b\)
  by (simp add: add-diff-inverse add-diff-cancel-left [of a c b - a, symmetric]
  add.commute)

lemma diff-add: \(b - a + a = b\)
  by (simp add: add.commute add-diff-inverse)

lemma le-add-diff: \(c \leq b + c - a\)
  by (auto simp add: add.commute diff-add-assoc2 le-iff-add)

lemma le-imp-diff-is-add: \(a \leq b \implies b - a = c \iff b = c + a\)
  by (auto simp add: add.commute add-diff-inverse)

lemma le-diff-conv2: \(c \leq b - a \iff c + a \leq b\)
(is ?P \iff ?Q)
proof
assume \( ?P \)
then have \( c + a \leq b - a + a \)
  by (rule add-right-mono)
then show \( ?Q \)
  by (simp add: add-diff-inverse add.commute)
next
assume \( ?Q \)
then have \( a + c \leq a + (b - a) \)
  by (simp add: add-diff-inverse add.commute)
then show \( ?P \) by simp
qed
end

4.9 Tools setup

lemma add-mono-thms-linordered-semiring:
  fixes \( i \) \( j \) \( k \) :: 'a::ordered-ab-semigroup-add
  shows \( i \leq j \wedge k \leq l \Longrightarrow i + k \leq j + l \)
  and \( i = j \wedge k \leq l \Longrightarrow i + k \leq j + l \)
  and \( i \leq j \wedge k = l \Longrightarrow i + k \leq j + l \)
  and \( i = j \wedge k = l \Longrightarrow i + k = j + l \)
  by (rule add-mono, clarify+)

lemma add-mono-thms-linordered-field:
  fixes \( i \) \( j \) \( k \) :: 'a::ordered-cancel-ab-semigroup-add
  shows \( i < j \wedge k = l \Longrightarrow i + k < j + l \)
  and \( i = j \wedge k < l \Longrightarrow i + k < j + l \)
  and \( i < j \wedge k \leq l \Longrightarrow i + k < j + l \)
  and \( i \leq j \wedge k < l \Longrightarrow i + k < j + l \)
  and \( i < j \wedge k < l \Longrightarrow i + k < j + l \)
  by (auto intro: add-strict-right-mono add-strict-left-mono
      add-less-le-mono add-le-less-mono add-strict-mono)

end

5 Abstract lattices

theory Lattices
imports Groups
begin

THEORY “Lattices” 138
5.1 Abstract semilattice

These locales provide a basic structure for interpretation into bigger structures; extensions require careful thinking, otherwise undesired effects may occur due to interpretation.

```isar
locale semilattice = abel-semigroup + 
  assumes idem [simp]: a ∗ a = a
begin

lemma left-idem [simp]: a ∗ (a ∗ b) = a ∗ b
  by (simp add: assoc [symmetric])

lemma right-idem [simp]: (a ∗ b) ∗ b = a ∗ b
  by (simp add: assoc)
end

locale semilattice-neutr = semilattice + comm-monoid

locale semilattice-order = semilattice + 
  fixes less-eq :: 'a ⇒ 'a ⇒ bool (infix ≤ 50) 
  and less :: 'a ⇒ 'a ⇒ bool (infix < 50) 
  assumes order-iff: a ≤ b ←→ a = a ∗ b 
  and strict-order-iff: a < b ←→ a = a ∗ b ∧ a ≠ b
begin

lemma orderI: a = a ∗ b =⇒ a ≤ b 
  by (simp add: order-iff)

lemma orderE:
  assumes a ≤ b 
  obtains a = a ∗ b 
  using assms by (unfold order-iff)

sublocale ordering less-eq less 
proof
  show a < b ←→ a ≤ b ∧ a ≠ b for a b 
    by (simp add: order-iff strict-order-iff)
next
  show a ≤ a for a 
    by (simp add: order-iff)
next
  fix a b 
  assume a ≤ b b ≤ a 
  then have a = a ∗ b a ∗ b = b 
    by (simp-all add: order-iff commute)
  then show a = b by simp
next
  fix a b c
```
assume \( a \leq b \leq c \)
then have \( a = a \cdot b \cdot b = b \cdot c \)
  by (simp-all add: order-iff commute)
then have \( a = a \cdot (b \cdot c) \)
  by simp
then have \( a = (a \cdot b) \cdot c \)
  by (simp add: assoc)
with \( a = a \cdot b \) [symmetric] have \( a = a \cdot c \) by simp
then show \( a \leq c \) by (rule orderI)
qed

lemma cobounded1 [simp]: \( a \cdot b \leq a \)
  by (simp add: order-iff commute)

lemma cobounded2 [simp]: \( a \cdot b \leq b \)
  by (simp add: order-iff)

lemma boundedI:
  assumes \( a \leq b \) and \( a \leq c \)
  shows \( a \leq b \cdot c \)
proof (rule orderI)
  from assms obtain \( a \cdot b = a \) and \( a \cdot c = a \)
    by (auto elim!: orderE)
  then show \( a = a \cdot (b \cdot c) \)
    by (simp add: assoc [symmetric])
qed

lemma boundedE:
  assumes \( a \leq b \cdot c \)
  obtains \( a \leq b \) and \( a \leq c \)
  using assms by (blast intro: trans cobounded1 cobounded2)

lemma bounded-iff [simp]: \( a \leq b \cdot c \rightleftharpoons a \leq b \land a \leq c \)
  by (blast intro: boundedI elim: boundedE)

lemma strict-boundedE:
  assumes \( a < b \cdot c \)
  obtains \( a < b \) and \( a < c \)
  using assms by (auto simp add: commute strict-iff-order elim: orderE intro!: that)+

lemma coboundedI1: \( a \leq c \implies a \cdot b \leq c \)
  by (rule trans) auto

lemma coboundedI2: \( b \leq c \implies a \cdot b \leq c \)
  by (rule trans) auto

lemma strict-coboundedI1: \( a < c \implies a \cdot b < c \)
  using irrefl
by (auto intro: not-eq-order-implies-strict coboundedI1 strict-implies-order elim: strict-boundedE)

lemma strict-coboundedI2: \( b < c \implies a \ast b < c \)
using strict-coboundedI1 \([of b c a]\) by (simp add: commute)

lemma mono: \( a \leq c \implies b \leq d \implies a \ast b \leq c \ast d \)
by (blast intro: boundedI coboundedI1 coboundedI2)

lemma absorb1: \( a \leq b \implies a \ast b = a \)
by (rule antisym) (auto simp: refl)

lemma absorb2: \( b \leq a \implies a \ast b = b \)
by (rule antisym) (auto simp: refl)

lemma absorb-iff1: \( a \leq b \iff a \ast b = a \)
using order-iff by auto

lemma absorb-iff2: \( b \leq a \iff a \ast b = b \)
using order-iff by (auto simp add: commute)

end

locale semilattice-neutr-order = semilattice-neutr + semilattice-order begin

sublocale ordering-top less-eq less 1
by standard (simp add: order-iff)
end

Passive interpretations for boolean operators

lemma semilattice-neutr-and:
semilattice-neutr HOL.conj True
by standard auto

lemma semilattice-neutr-or:
semilattice-neutr HOL.disj False
by standard auto

5.2 Syntactic infimum and supremum operations

class inf =
fixes inf :: 'a \Rightarrow 'a \Rightarrow 'a (infixl \&\& 70)

class sup =
fixes sup :: 'a \Rightarrow 'a \Rightarrow 'a (infixl \lor 65)
5.3 Concrete lattices

class semilattice-inf = order + inf +
  assumes inf-le1 [simp]: x \land y \leq x
  and inf-le2 [simp]: x \land y \leq y
  and inf-greatest: x \leq y \implies x \leq z \implies x \leq y \land z

class semilattice-sup = order + sup +
  assumes sup-ge1 [simp]: x \leq x \lor y
  and sup-ge2 [simp]: y \leq x \lor y
  and sup-least: y \leq x \implies z \leq x \implies y \lor z \leq x

begin

Dual lattice.

lemma dual-semilattice: class.semilattice-inf sup greater-eq greater
  by (rule class.semilattice-inf.intro, rule dual-order)
    (unfold-locales, simp-all add: sup-least)

end

class lattice = semilattice-inf + semilattice-sup

5.3.1 Intro and elim rules

context semilattice-inf

begin

lemma le-infI1: a \leq x \implies a \land b \leq x
  by (rule order-trans) auto

lemma le-infI2: b \leq x \implies a \land b \leq x
  by (rule order-trans) auto

lemma le-infI: x \leq a \implies x \leq b \implies x \leq a \land b
  by (fact inf-greatest)

lemma le-infE: x \leq a \land b \implies (x \leq a \implies x \leq b \implies P) \implies P
  by (blast intro: order-trans inf-le1 inf-le2)

lemma le-inf iff: x \leq y \land z \iff x \leq y \land x \leq z
  by (blast intro: le-infE elim: le-infE)

lemma le-inf iff: x \leq y \iff x \land y = x

lemma inf-mono: a \leq c \implies b \leq d \implies a \land b \leq c \land d
  by (fast intro: inf-greatest le-infI1 le-infI2)

lemma mono-inf: mono f \implies f (A \land B) \leq f A \land f B
  for f :: 'a \Rightarrow 'b::semilattice-inf
  by (auto simp add: mono-def intro: Lattices.inf-greatest)
THEORY "Lattices"

end

context semilattice-sup
begin

lemma le-supI1: x ≤ a ⇒ x ≤ a ⊔ b
  by (rule order-trans) auto

lemma le-supI2: x ≤ b ⇒ x ≤ a ⊔ b
  by (rule order-trans) auto

lemma le-supI: a ≤ x ⇒ b ≤ x ⇒ a ⊔ b ≤ x
  by (fact sup-least)

lemma le-supE: a ⊔ b ≤ x ⇒ (a ≤ x ⇒ b ≤ x ⇒ P) ⇒ P
  by (blast intro: order-trans sup-least)

lemma le-infI: a ⊔ b ≤ x ≤ z ⇒ y ≤ z ≤ y ≤ z
  by (blast intro: le-infI1 le-infI2 simp add: le-inf-iff)

lemma le-inf-supp: x ≤ y ≤ x ⊔ y = y
  by (auto intro: le-infI1 le-infI2 antisym dest: eq-iff simp add: le-inf-iff)

lemma sup-mono: a ≤ c ⇒ b ≤ d ⇒ a ⊔ b ≤ c ⊔ d
  by (fast intro: sup-least)

lemma mono-sup: mono f ⇒ f A ⊔ f B ≤ f (A ⊔ B)
  by (auto simp add: mono-def intro: Lattices.sup-least)

end

5.3.2 Equational laws

context semilattice-inf
begin

sublocale inf: semilattice inf

proof
  fix a b c
  show (a ⊓ b) ⊓ c = a ⊓ (b ⊓ c)
    by (rule antisym) (auto intro: le-infI1 le-infI2 simp add: le-inf-iff)
  show a ⊓ b = b ⊓ a
    by (rule antisym) (auto simp add: le-inf-iff)
  show a ⊓ a = a
    by (rule antisym) (auto simp add: le-inf-iff)

qed

sublocale inf: semilattice-order inf less-eq less
by standard (auto simp add: le_iff_inf less_le)

lemma inf-assoc: \((x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)\)
  by (fact inf_assoc)

lemma inf-commute: \((x \sqcap y) = (y \sqcap x)\)
  by (fact inf_commute)

lemma inf-left-commute: \(x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z)\)
  by (fact inf_left_commute)

lemma inf-idem: \(x \sqcap x = x\)
  by (fact inf_idem)

lemma inf-left-idem: \(x \sqcap (x \sqcap y) = x \sqcap y\)
  by (fact inf_left_idem)

lemma inf-right-idem: \(x \sqcap y \sqcap y = x \sqcap y\)
  by (fact inf_right_idem)

lemma inf-absorb1: \(x \leq y \implies x \sqcap y = x\)
  by (rule antisym) auto

lemma inf-absorb2: \(y \leq x \implies x \sqcap y = y\)
  by (rule antisym) auto

lemmas inf-aci = inf-commute inf-assoc inf-left-commute inf-left-idem

end

context semilattice-sup
begin

sublocale sup: semilattice sup
proof
  fix \(a\) \(b\) \(c\)
  show \((a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)\)
    by (rule antisym) (auto intro: le-supI1 le-supI2 simp add: le-sup_iff)
  show \(a \sqcup b = b \sqcup a\)
    by (rule antisym) (auto simp add: le-sup_iff)
  show \(a \sqcup a = a\)
    by (rule antisym) (auto simp add: le-sup_iff)
qed

sublocale sup: semilattice-order sup greater-eq greater
by standard (auto simp add: le_iff_sup sup.commute less_le)

lemma sup-assoc: \((x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)\)
  by (fact sup_assoc)
lemma sup-commute: \( (x \sqcup y) = (y \sqcup x) \)  
by (fact sup.commute)

lemma sup-left-commute: \( x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z) \)  
by (fact sup.left-commute)

lemma sup-idem: \( x \sqcap x = x \)  
by (fact sup.idem)

lemma sup-left-idem [simp]: \( x \sqcup (x \sqcap y) = x \sqcap y \)  
by (fact sup.left-idem)

lemma sup-absorb1: \( y \leq x \implies x \sqcup y = x \)  
by (rule antisym) auto

lemma sup-absorb2: \( x \leq y \implies x \sqcup y = y \)  
by (rule antisym) auto

lemmas sup-aci = sup-commute sup-associ sup-left-commute sup-left-idem

end

context lattice
begin

lemma dual-lattice: class.lattice sup (\geq) (> \,) inf  
by (rule class.lattice.intro,  
rule dual-semilattice,  
rule class.semilattice-sup.intro,  
rule dual-order)  
(anfold-locales, auto)

lemma inf-sup-absorb [simp]: \( x \sqcap (x \sqcup y) = x \)  
by (blast intro: antisym inf-le1 inf-greatest sup-ge1)

lemma sup-inf-absorb [simp]: \( x \sqcup (x \sqcap y) = x \)  
by (blast intro: antisym sup-ge1 sup-least inf-le1)

lemmas inf-sup-aci = inf-aci sup-aci

lemmas inf-sup-ord = inf-le1 inf-le2 sup-ge1 sup-ge2

Towards distributivity.

lemma distrib-sup-le: \( x \sqcup (y \sqcap z) \leq (x \sqcup y) \sqcap (x \sqcup z) \)  
by (auto intro: le-infI1 le-infI2 le-supI1 le-supI2)

lemma distrib-inf-le: \( (x \sqcap y) \sqcup (x \sqcap z) \leq x \sqcap (y \sqcup z) \)  
by (auto intro: le-infI1 le-infI2 le-supI1 le-supI2)
If you have one of them, you have them all.

**Lemma distrib-imp1:**

**Assumes** distr: \( \forall x \ y \ z. \ x \cap (y \cup z) = (x \cap y) \cup (x \cap z) \)

**Shows** \( x \cup (y \cap z) = (x \cup y) \cap (x \cup z) \)

**Proof—**

- **Have** \( x \cup (y \cap z) = (x \cup (x \cap z)) \cup (y \cap z) \)
  - by **simp**
- **Also have** \( x \cup (z \cap (x \cup y)) \)
  - by (simp add: distr inf-commute sup-assoc del: sup-inf-absorb)
- **Also have** \( ((x \cup y) \cap x) \cup ((x \cup y) \cap z) \)
  - by (simp add: inf-commute)
- **Also have** \( (x \cup y) \cap (x \cup z) \)
  - by (simp add: distr)

**Finally show** ?thesis .

**Qed**

**Lemma distrib-imp2:**

**Assumes** distr: \( \forall x \ y \ z. \ x \cup (y \cap z) = (x \cup y) \cap (x \cup z) \)

**Shows** \( x \cap (y \cup z) = (x \cap y) \cup (x \cap z) \)

**Proof—**

- **Have** \( x \cap (y \cup z) = (x \cap (x \cup z)) \cap (y \cup z) \)
  - by **simp**
- **Also have** \( x \cap (z \cup (x \cap y)) \)
  - by (simp add: distr sup-commute inf-assoc del: inf-sup-absorb)
- **Also have** \( ((x \cap y) \cup x) \cap ((x \cap y) \cup z) \)
  - by (simp add: sup-commute)
- **Also have** \( (x \cup y) \cup (x \cap z) \)
  - by (simp add: distr)

**Finally show** ?thesis .

**Qed**

**End**

### 5.3.3 Strict order

**Context** semilattice-inf

**Begin**

**Lemma** less-infI1: \( a < x \Rightarrow a \cap b < x \)

**By** (auto simp add: less-le inf-absorb1 intro: le-infI1)

**Lemma** less-infI2: \( b < x \Rightarrow a \cap b < x \)

**By** (auto simp add: less-le inf-absorb2 intro: le-infI2)

**End**

**Context** semilattice-sup

**Begin**

**Lemma** less-supI1: \( x < a \Rightarrow x < a \cup b \)

**Using** dual-semilattice
by (rule semilattice-inf.less-infI1)

lemma less-supI2: \( x < b \Rightarrow x < a \sqcup b \)
  using dual-semilattice
  by (rule semilattice-inf.less-infI2)

end

5.4 Distributive lattices

class distrib-lattice = lattice +
  assumes sup-inf-distrib1: \( x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z) \)

context distrib-lattice
begin

lemma sup-inf-distrib2: \( (y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x) \)
  by (simp add: sup-commute sup-inf-distrib1)

lemma inf-sup-distrib1: \( x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z) \)
  by (rule distrib-imp2 [OF sup-inf-distrib1])

lemma inf-sup-distrib2: \( (y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x) \)
  by (simp add: inf-commute inf-sup-distrib1)

lemma dual-distrib-lattice: class.distrib-lattice sup (≥) (>)
  by (rule class.distrib-lattice.intro, rule dual-lattice)
  (unfold-locales, fact inf-sup-distrib1)

lemmas sup-inf-distrib = sup-inf-distrib1 sup-inf-distrib2
lemmas inf-sup-distrib = inf-sup-distrib1 inf-sup-distrib2
lemmas distrib = sup-inf-distrib1 sup-inf-distrib2 inf-sup-distrib1 inf-sup-distrib2

end

5.5 Bounded lattices and boolean algebras

class bounded-semilattice-inf-top = semilattice-inf + order-top
begin

sublocale inf-top: semilattice-neutr inf top
  + inf-top: semilattice-neutr-order inf top less-eq less

proof
  show \( x \sqcap \top = x \) for \( x \)
    by (rule inf-absorb1) simp

qed

end
THEORY "Lattices"

class bounded-semilattice-sup-bot = semilattice-sup + order-bot
begin

sublocale sup-bot: semilattice-neutr sup bot
+ sup-bot: semilattice-neutr-order sup bot greater-eq greater
proof
  show $x \sqcup \bot = x$ for $x$
  by (rule sup-absorb1) simp
qed

end

class bounded-lattice-bot = lattice + order-bot
begin

subclass bounded-semilattice-sup-bot ..

lemma inf-bot-left [simp]: $\bot \sqcap x = \bot$
  by (rule inf-absorb1) simp

lemma inf-bot-right [simp]: $x \sqcap \bot = \bot$
  by (rule inf-absorb2) simp

lemma sup-bot-left: $\bot \sqcup x = x$
  by (fact sup-bot.left-neutral)

lemma sup-bot-right: $x \sqcup \bot = x$
  by (fact sup-bot.right-neutral)

lemma sup-eq-bot-iff [simp]: $x \sqcup y = \bot \longleftrightarrow x = \bot \land y = \bot$
  by (simp add: eq-iff)

lemma bot-eq-sup-iff [simp]: $\bot = x \sqcup y \longleftrightarrow x = \bot \land y = \bot$
  by (simp add: eq-iff)

end

class bounded-lattice-top = lattice + order-top
begin

subclass bounded-semilattice-inf-top ..

lemma sup-top-left [simp]: $\top \sqcup x = \top$
  by (rule sup-absorb1) simp

lemma sup-top-right [simp]: $x \sqcup \top = \top$
  by (rule sup-absorb2) simp
lemma inf-top-left: $\top \sqcap x = x$
  by (fact inf-top.left-neutral)

lemma inf-top-right: $x \sqcap \top = x$
  by (fact inf-top.right-neutral)

lemma inf-eq-top-iff [simp]: $x \sqcap y = \top \iff x = \top \land y = \top$
  by (simp add: eq_iff)

end

class bounded-lattice = lattice + order-bot + order-top
begin
subclass bounded-lattice-bot ..
subclass bounded-lattice-top ..

lemma dual-bounded-lattice:
  class bounded-lattice sup greater-eq greater inf $\top \bot$
  by unfold-locales (auto simp add: less-le-not-le)
end

class boolean-algebra = distrib-lattice + bounded-lattice + minus + uminus +
assumes inf-compl-bot: $x \sqcap -x = \bot$
  and sup-compl-top: $x \sqcup -x = \top$
assumes diff-eq: $x - y = x \sqcap -y$
begin

lemma dual-boolean-algebra:
  class boolean-algebra ($\lambda x y. x \sqcup -y$) uminus sup greater-eq greater inf $\top \bot$
  by (rule class:boolean-algebra.intro,
    rule dual-bounded-lattice,
    rule dual-distrib-lattice)
  (unfold-locales, auto simp add: inf-compl-bot sup-compl-top diff-eq)

lemma compl-inf-bot [simp]: $-x \sqcap x = \bot$
  by (simp add: inf-commute inf-compl-bot)

lemma compl-sup-top [simp]: $-x \sqcup x = \top$
  by (simp add: sup-commute sup-compl-top)

lemma compl-unique:
  assumes $x \sqcap y = \bot$
  and $x \sqcup y = \top$
  shows $-x = y$
  proof
    have $(x \sqcap -x) \sqcup (-x \sqcap y) = (x \sqcap y) \sqcup (-x \sqcap y)$
      using inf-compl-bot assms(1) by simp
    then have $(-x \sqcap x) \sqcup (-x \sqcap y) = (y \sqcap x) \sqcup (y \sqcap -x)$
by \((\text{simp add: inf-commute})\)
then have \(-x \sqcap (x \sqcup y) = y \sqcap (x \sqcup -x)\)
by \((\text{simp add: inf-sup-distrib1})\)
then have \(-x \sqcap T = y \sqcap T\)
using \text{sup-compl-top} \text{assms(2)} by \text{simp}
then show \(-x = y\) by \text{simp}
qed

\textbf{lemma} \textit{double-compl} \texttt{[simp]}: \(-(-x) = x\)
using \text{compl-inf-bot} \text{compl-sup-top} by \text{rule compl-unique}

\textbf{lemma} \textit{compl-eq-compl-iff} \texttt{[simp]}: \(-x = -y \iff x = y\)
proof
assume \(-x = -y\)
then have \(-(x) = -(y)\) by \text{rule arg-cong}
then show \(x = y\) by \text{simp}
next
assume \(x = y\)
then show \(-x = -y\) by \text{simp}
qed

\textbf{lemma} \textit{compl-bot-eq} \texttt{[simp]}: \(-\bot = \top\)
proof
- from \text{sup-compl-top} have \(\bot \sqcup -\bot = T\).
then show \(?\text{thesis}\) by \text{simp}
qed

\textbf{lemma} \textit{compl-top-eq} \texttt{[simp]}: \(-\top = \bot\)
proof
- from \text{inf-compl-bot} have \(\top \sqcap -\top = \bot\).
then show \(?\text{thesis}\) by \text{simp}
qed

\textbf{lemma} \textit{compl-inf} \texttt{[simp]}: \(-(x \sqcap y) = -x \sqcup -y\)
proof \textit{(rule compl-unique)}
have \((x \sqcap y) \sqcap (-x \sqcup -y) = (y \sqcap (x \sqcap -x)) \sqcup (x \sqcap (y \sqcap -y))\)
by \text{simp only: inf-sup-distrib inf-aci}
then show \((x \sqcap y) \sqcap (-x \sqcup -y) = \bot\)
by \text{simp add: inf-compl-bot}
next
have \((x \sqcap y) \sqcup (-x \sqcup -y) = (-y \sqcup (x \sqcup -x)) \sqcap (-x \sqcap (y \sqcup -y))\)
by \text{simp only: sup-inf-distrib sup-aci}
then show \((x \sqcap y) \sqcup (-x \sqcup -y) = T\)
by \text{simp add: sup-compl-bot}
qed

\textbf{lemma} \textit{compl-sup} \texttt{[simp]}: \(-(x \sqcup y) = -x \sqcap -y\)
using \text{dual-boolean-algebra}
by \text{rule boolean-algebra.compl-inf}
lemma compl-mono:
  assumes $x \leq y$
  shows $-y \leq -x$
proof -
  from assms have $x \sqcup y = y$ by (simp only: le-iff-sup)
  then have $-(x \sqcup y) = -y$ by simp
  then have $-x \sqcap -y = -y$ by simp
  then show \(?\text{thesis}\) by (simp only: le-iff-inf)
qed

lemma compl-le-compl-iff [simp]: $-x \leq -y \iff y \leq x$
by (auto dest: compl-mono)

lemma compl-le-swap1:
  assumes $y \leq -x$
  shows $x \leq -y$
proof -
  from assms have $-(-x) \leq -y$ by (simp only: compl-le-compl-iff)
  then show ?thesis by simp
qed

lemma compl-le-swap2:
  assumes $-y \leq x$
  shows $-x \leq -y$
proof -
  from assms have $-x \leq -(y)$ by (simp only: compl-le-compl-iff)
  then show ?thesis by simp
qed

lemma compl-less-compl-iff: $-x < -y \iff y < x$
by (auto simp add: less-le)

lemma compl-less-swap1:
  assumes $y < -x$
  shows $x < -y$
proof -
  from assms have $-(-x) < -y$ by (simp only: compl-less-compl-iff)
  then show ?thesis by simp
qed

lemma compl-less-swap2:
  assumes $-y < x$
  shows $-x < y$
proof -
  from assms have $-x < -(y)$
  by (simp only: compl-less-compl-iff)
  then show ?thesis by simp
qed

lemma sup-cancel-left1: sup (sup x a) (sup (− x) b) = top
by (simp add: inf-sup-aci sup-compl-top)

lemma sup-cancel-left2: sup (sup (− x) a) (sup x b) = top
by (simp add: inf-sup-aci sup-compl-top)

lemma inf-cancel-left1: inf (inf x a) (inf (− x) b) = bot
by (simp add: inf-sup-aci inf-compl-bot)

lemma inf-cancel-left2: inf (inf (− x) a) (inf x b) = bot
by (simp add: inf-sup-aci inf-compl-bot)

declare inf-compl-bot [simp]
and sup-compl-top [simp]

lemma sup-compl-top-left1 [simp]: sup (− x) (sup x y) = top
by (simp add: sup-assoc[symmetric])

lemma sup-compl-top-left2 [simp]: sup x (sup (− x) y) = top
using sup-compl-top-left1[of − x y] by simp

lemma inf-compl-bot-left1 [simp]: inf (− x) (inf x y) = bot
by (simp add: inf-assoc[symmetric])

lemma inf-compl-bot-left2 [simp]: inf x (inf (− x) y) = bot
using inf-compl-bot-left1[of − x y] by simp

lemma inf-compl-bot-right [simp]: inf x (inf y (− x)) = bot
by (subst inf-left-commute) simp

end

locale boolean-algebra-cancel
begin

lemma sup1: (A::'a::semilattice-sup) ≡ sup k a =⇒ sup A b = sup k (sup a b)
by (simp only: ac-simps)

lemma sup2: (B::'a::semilattice-sup) ≡ sup k b =⇒ sup a B = sup k (sup a b)
by (simp only: ac-simps)

lemma sup0: (a::'a::bounded-semilattice-sup-bot) ≡ sup a bot
by simp

lemma inf1: (A::'a::semilattice-inf) ≡ inf k a =⇒ inf A b = inf k (inf a b)
by (simp only: ac-simps)
lemma inf2: \((B::'a::semilattice-inf) \equiv \inf k b \Rightarrow \inf a B \equiv \inf k (\inf a b)\)
by (simp only: ac-simps)

lemma inf0: \((a::'a::bounded-semilattice-inf-top) \equiv \inf a top\)
by simp
end

ML-file (Tools/boolean-algebra-cancel.ML)

simproc-setup boolean-algebra-cancel-sup (sup a b::'a::boolean-algebra) =
(fn phi => fn ss => try Boolean-Algebra-Cancel.cancel-sup-conv)

simproc-setup boolean-algebra-cancel-inf (inf a b::'a::boolean-algebra) =
(fn phi => fn ss => try Boolean-Algebra-Cancel.cancel-inf-conv)

5.6 \(\text{min}/\text{max}\) as special case of lattice

context linorder
begin

sublocale min: semilattice-order min less-eq less
+ max: semilattice-order max greater-eq greater
by standard (auto simp add: min-def max-def)

lemma min-le-iff-disj: \(\min x y \leq z \iff x \leq z \vee y \leq z\)
unfolding min-def using linear by (auto intro: order-trans)

lemma le-max-iff-disj: \(z \leq \max x y \iff z \leq x \vee z \leq y\)
unfolding max-def using linear by (auto intro: order-trans)

lemma min-less-iff-disj: \(\min x y < z \iff x < z \vee y < z\)
unfolding min-def le-less using less-linear by (auto intro: less-trans)

lemma less-max-iff-disj: \(z < \max x y \iff z < x \vee z < y\)
unfolding max-def le-less using less-linear by (auto intro: less-trans)

lemma min-less-iff-conj [simp]: \(z < \min x y \iff z < x \land z < y\)
unfolding min-def le-less using less-linear by (auto intro: less-trans)

lemma max-less-iff-conj [simp]: \(\max x y < z \iff x < z \land y < z\)
unfolding max-def le-less using less-linear by (auto intro: less-trans)

lemma min-max-distrib1: \(\min (\max b c) a = \max (\min b a) (\min c a)\)
by (auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro: antisym)

lemma min-max-distrib2: \(\min a (\max b c) = \max (\min a b) (\min a c)\)
by (auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro:
lemma max-min-distrib1: \( \max (\min b c) \ a = \min (\max b a) (\max c a) \)
  by (auto simp add: min-def max-def not-le dest: le-less-trans less-less-trans intro: antisym)

lemma max-min-distrib2: \( \max a (\min b c) = \min (\max a b) (\max a c) \)
  by (auto simp add: min-def max-def not-le dest: le-less-trans less-trans intro: antisym)

lemmas min-max-distrib = min-max-distrib1 min-max-distrib2 max-min-distrib1 max-min-distrib2

lemma split-min [no-atp]: \( P (\min i j) \leftrightarrow (i \leq j \rightarrow P i) \land (\sim i \leq j \rightarrow P j) \)
  by (simp add: min-def)

lemma split-max [no-atp]: \( P (\max i j) \leftrightarrow (i \leq j \rightarrow P j) \land (\sim i \leq j \rightarrow P i) \)
  by (simp add: max-def)

lemma split-min-lin [no-atp]:
  \( P (\min a b) \leftrightarrow (b = a \rightarrow P a) \land (a < b \rightarrow P b) \land (b < a \rightarrow P a) \)
  by (cases a b rule: linorder-cases) (auto simp add: min.absorb1 min.absorb2)

lemma split-max-lin [no-atp]:
  \( P (\max a b) \leftrightarrow (b = a \rightarrow P a) \land (a < b \rightarrow P b) \land (b < a \rightarrow P a) \)
  by (cases a b rule: linorder-cases) (auto simp add: max.absorb1 max.absorb2)

lemma min-of-mono: mono \( f \Rightarrow \min (f m) (f n) = f (\min m n) \) for \( f :: 'a \Rightarrow 'b::linorder \)
  by (auto simp: mono-def Orderings.min-def min-def intro: Orderings.antisym)

lemma max-of-mono: mono \( f \Rightarrow \max (f m) (f n) = f (\max m n) \) for \( f :: 'a \Rightarrow 'b::linorder \)
  by (auto simp: mono-def Orderings.max-def max-def intro: Orderings.antisym)

end

lemma max-of-antimono: antimono \( f \Rightarrow \max (f x) (f y) = f (\min x y) \)
  and min-of-antimono: antimono \( f \Rightarrow \min (f x) (f y) = f (\max x y) \)
  for \( f :: 'a::linorder \Rightarrow 'b::linorder \)
  by (auto simp: antimono-def Orderings.max-def min-def intro!: antisym)

lemma inf-min: inf = (\( \min :: 'a::{semilattice-inf,linorder} \Rightarrow 'a \Rightarrow 'a \))
  by (auto intro: antisym simp add: min-def fun-eq-iff)

lemma sup-max: sup = (\( \max :: 'a::{semilattice-sup,linorder} \Rightarrow 'a \Rightarrow 'a \))
  by (auto intro: antisym simp add: max-def fun-eq-iff)
5.7 Uniqueness of inf and sup

**Lemma (in semilattice-inf) inf-unique:**

**Fixes** $f$ (**infixl** $\triangle$ 70)

**Assumes** le1: $\forall x y. x \triangle y \leq x$

and le2: $\forall x y. x \triangle y \leq y$

and greatest: $\forall x y z. x \leq y \implies x \leq z \implies x \leq y \triangle z$

**Show** $x \cap y = x \triangle y$

**Proof** (**rule antisym**)

show $x \cap y \leq x \triangle y$

by (**rule le-infI**) (**rule le1**, **rule le2**)

have leI: $\forall x y z. x \leq z \implies y \leq z \implies x \leq y \triangle z$

by (**blast intro: greatest**)  

show $x \cap y \leq x \triangle y$

by (**rule leI**), simp-all

**Qed**

**Lemma (in semilattice-sup) sup-unique:**

**Fixes** $f$ (**infixl** $\triangledown$ 70)

**Assumes** ge1: $\forall x y. x \leq y \triangledown y$

and ge2: $\forall x y. y \leq x \triangledown y$

and least: $\forall x y z. y \leq x \implies z \leq x \implies y \triangledown z \leq x$

**Show** $x \uplus y = x \triangledown y$

**Proof** (**rule antisym**)

show $x \uplus y \leq x \triangledown y$

by (**rule le-supI**) (**rule ge1**, **rule ge2**)

have leI: $\forall x y z. x \leq z \implies y \leq z \implies x \triangledown y \leq z$

by (**blast intro: least**)  

show $x \triangledown y \leq x \uplus y$

by (**rule leI**), simp-all

**Qed**

5.8 Lattice on bool

**Instantiation** bool :: boolean-algebra

**Begin**

**Definition** bool-Compl-def [simp]: $\text{uminus} = \text{Not}$

**Definition** bool-diff-def [simp]: $A - B \iff A \land \neg B$

**Definition** [simp]: $P \cap Q \iff P \land Q$

**Definition** [simp]: $P \cup Q \iff P \lor Q$

**Instance** by **standard auto**

**End**

**Lemma** sup-boolI1: $P \implies P \cup Q$
by simp

lemma sup-boolI2: Q ⊢ P ⊔ Q
  by simp

lemma sup-boolE: P ⊔ Q ⊢ (P ⊢ R) ⊢ (Q ⊢ R) ⊢ R
  by auto

5.9 Lattice on - ⇒ -

instantiation fun :: (type, semilattice-sup) semilattice-sup
begin

definition f ⊔ g = (λx. f x ⊔ g x)

lemma sup-apply [simp, code]: (f ⊔ g) x = f x ⊔ g x
  by (simp add: sup-fun-def)

instance by standard (simp-all add: le-fun-def)
end

instantiation fun :: (type, semilattice-inf) semilattice-inf
begin

definition f ⊓ g = (λx. f x ⊓ g x)

lemma inf-apply [simp, code]: (f ⊓ g) x = f x ⊓ g x
  by (simp add: inf-fun-def)

instance by standard (simp-all add: le-fun-def)
end

instance fun :: (type, lattice) lattice ..

instance fun :: (type, distrib-lattice) distrib-lattice
  by standard (rule ext, simp add: sup-inf-distrib1)

instance fun :: (type, bounded-lattice) bounded-lattice ..

instantiation fun :: (type, uminus) uminus
begin

definition fun-Compl-def: − A = (λx. − A x)

lemma uminus-apply [simp, code]: (− A) x = − (A x)
  by (simp add: fun-Compl-def)
instance ..

end

**instantiation** fun :: (type, minus) minus

**begin**

**definition** fun-diff-def: \( A - B = (\lambda x. A x - B x) \)

**lemma** minus-apply [simp, code]: \((A - B) x = A x - B x\)

  by (simp add: fun-diff-def)

**instance ..**

end

**instance** fun :: (type, boolean-algebra) boolean-algebra

  by standard (rule ext, simp-all add: inf-compl-bot sup-compl-top diff-eq)+

### 5.10 Lattice on unary and binary predicates

**lemma** inf1I: \( A x \Longrightarrow B x \Longrightarrow (A \cap B) x \)

  by (simp add: inf-fun-def)

**lemma** inf2I: \( A x y \Longrightarrow B x y \Longrightarrow (A \cap B) x y \)

  by (simp add: inf-fun-def)

**lemma** inf1E: \( (A \cap B) x \Longrightarrow (A x \Longrightarrow B x \Longrightarrow P) \Longrightarrow P \)

  by (simp add: inf-fun-def)

**lemma** inf2E: \( (A \cap B) x y \Longrightarrow (A x y \Longrightarrow B x y \Longrightarrow P) \Longrightarrow P \)

  by (simp add: inf-fun-def)

**lemma** inf1D1: \( (A \cap B) x \Longrightarrow A x \)

  by (rule inf1E)

**lemma** inf2D1: \( (A \cap B) x y \Longrightarrow A x y \)

  by (rule inf2E)

**lemma** inf1D2: \( (A \cap B) x \Longrightarrow B x \)

  by (rule inf1E)

**lemma** inf2D2: \( (A \cap B) x y \Longrightarrow B x y \)

  by (rule inf2E)

**lemma** sup1I1: \( A x \Longrightarrow (A \sqcup B) x \)

  by (simp add: sup-fun-def)
lemma sup2I1: A x y ⇒ (A ∪ B) x y
  by (simp add: sup-fun-def)

lemma sup1I2: B x ⇒ (A ∪ B) x
  by (simp add: sup-fun-def)

lemma sup2I2: B x y ⇒ (A ∪ B) x y
  by (simp add: sup-fun-def)

lemma sup1E: (A ∪ B) x ⇒ (A x ⇒ P) ⇒ (B x ⇒ P) ⇒ P
  by (simp add: sup-fun-def) iprover

lemma sup2E: (A ∪ B) x y ⇒ (A x y ⇒ P) ⇒ (B x y ⇒ P) ⇒ P
  by (simp add: sup-fun-def) iprover

Classical introduction rule: no commitment to A vs B.

lemma sup1CI: (¬ B x ⇒ A x) ⇒ (A ∪ B) x
  by (auto simp add: sup-fun-def)

lemma sup2CI: (¬ B x y ⇒ A x y) ⇒ (A ∪ B) x y
  by (auto simp add: sup-fun-def)

end

6  Set theory for higher-order logic

theory Set
  imports Lattices
begin

6.1  Sets as predicates

typedec 'a set

axiomatization Collect :: ('a ⇒ bool) ⇒ 'a set — comprehension
  and member :: 'a ⇒ 'a set ⇒ bool — membership
where mem-Collect-eq [iff, code-unfold]: member a (Collect P) = P a
  and Collect-mem-eq [simp]: Collect (λx. member x A) = A

notation member ('(∈)'), and
  member (/(⁻/ ∈ -)) [51, 51] 50

abbreviation not-member
  where not-member x A ≡ (x ∈ A) — non-membership
notation not-member ('(∉')) and
  not-member (/(⁻/ ∉ -)) [51, 51] 50
notation (ASCII)
  member ("\{\}") and
  member ("\{-\}") [51, 51] 50) and
  not-member ("\~{\}") and
  not-member ("\{-\}") [51, 51] 50)

Set comprehensions

syntax
-\text手上 possesses 
  pttrn \Rightarrow bool \Rightarrow \'a set \ ((1\{-\}/-))

translations
\{x, P\} \leftarrow \text{CONST Collect} (\lambda x. P)

syntax (ASCII)
-\textCollect :: pstt => \'a set \Rightarrow \'a set \ ((1\{-\}/\-))

syntax
-\textCollect :: pttrn \Rightarrow \'a set \Rightarrow \'a set \ ((1\{-\}/\-))

translations
\{p:A. P\} \leftarrow \text{CONST Collect} (\lambda x. P)

lemma \textCollectI: \text \rightarrow \mathit{a} \in \{x. P x\}
  \text by simp

lemma \textCollectD: \mathit{a} \in \{x. P x\} \Rightarrow \mathit{P} \mathit{a}
  \text by simp

lemma \textCollect-cong: \text \& \text \Rightarrow \text
  \text by simp

Simproc for pulling \text \rightarrow \mathit{a} \in \{x. \ldots \& x = t \& \ldots\} to the front (and simultaneously for \text =\text):

\text Simproc-setup defined-Collect (\{x. P x \& Q x\}) = \langle
  fn \text => Quantifier1.rearrange-Collect
  (fn ctxt =>
    resolve-tac ctxt @\{thms Collect-cong\} 1 THEN
    resolve-tac ctxt @\{thms iffI\} 1 THEN
    ALLGOALS
      (EVERY' [REPEAT-DETERM o erevolve-tac ctxt @\{thms conjE\},
        DEPTH-SOLVE-1 o (assume-tac ctxt ORELSE' resolve-tac ctxt @\{thms conjI\}])))
  \rangle

lemmas \textCollectE = CollectD [elim-format]

lemma \text set-eql:
  assumes \text \rightarrow \mathit{A} \equiv \mathit{B}
  \text
  from \text \& \text have \{x. \, x \in \mathit{A}\} = \{x. \, x \in \mathit{B}\}
THEORY "Set"

by simp
then show ?thesis by simp
qed

lemma set-eq-iff: $A = B \iff (\forall x. x \in A \iff x \in B)$
by (auto intro: set-eqI)

lemma Collect-eqI:
assumes $\forall x. P x = Q x$
shows $\text{Collect } P = \text{Collect } Q$
using assms by (auto intro: set-eqI)

Lifting of predicate class instances
instantiation set :: (type) boolean-algebra
begin

definition less-eq-set
where $A \leq B \iff (\lambda x. \text{member } x A) \leq (\lambda x. \text{member } x B)$

definition less-set
where $A < B \iff (\lambda x. \text{member } x A) < (\lambda x. \text{member } x B)$

definition inf-set
where $A \cap B = \text{Collect } ((\lambda x. \text{member } x A) \cap (\lambda x. \text{member } x B))$

definition sup-set
where $A \cup B = \text{Collect } ((\lambda x. \text{member } x A) \cup (\lambda x. \text{member } x B))$

definition bot-set
where $\bot = \text{Collect } \bot$

definition top-set
where $\top = \text{Collect } \top$

definition uminus-set
where $- A = \text{Collect } (- (\lambda x. \text{member } x A))$

definition minus-set
where $A - B = \text{Collect } ((\lambda x. \text{member } x A) - (\lambda x. \text{member } x B))$

instance
by standard
(simp-all add: less-eq-set-def less-set-def inf-set-def sup-set-def
bot-set-def top-set-def uminus-set-def minus-set-def
less-le-not-le sup-inf-distrib1 diff-eq set-eqI fun-eq-iff
del: inf-apply sup-apply bot-apply top-apply minus-apply uminus-apply)

end

Set enumerations
abbreviation empty :: 'a set (\{\})
  where \{\} ≡ bot

definition insert :: 'a ⇒ 'a set ⇒ 'a set
  where insert-compr: insert a B = \{x. x = a ∨ x ∈ B\}
syntax
  -Finset :: args ⇒ 'a set (\{\})
translations
  \{x, xs\} ⇌ CONST insert x \{xs\}
  \{x\} ⇌ CONST insert x \{}

6.2 Subsets and bounded quantifiers
abbreviation subset :: 'a set ⇒ 'a set ⇒ bool
  where subset ≡ less
abbreviation subset-eq :: 'a set ⇒ 'a set ⇒ bool
  where subset-eq ≡ less-eq
notation
  subset ("\subseteq") and
  subset ((/- ⊆ -) [51, 51] 50) and
  subset-eq ("\subseteq") and
  subset-eq ((/- ⊆ -) [51, 51] 50)
abbreviation (input)
  supset :: 'a set ⇒ 'a set ⇒ bool where
  supset ≡ greater
abbreviation (input)
  supset-eq :: 'a set ⇒ 'a set ⇒ bool where
  supset-eq ≡ greater-eq
notation
  supset ("\supset") and
  supset ((/- ⊃ -) [51, 51] 50) and
  supset-eq ("\supset") and
  supset-eq ((/- ⊃ -) [51, 51] 50)
notation (ASCII output)
  supset ("<") and
  supset ((/- < -) [51, 51] 50) and
  supset-eq ("<") and
  supset-eq ((/- <= -) [51, 51] 50)
definition Ball :: 'a set ⇒ ('a ⇒ bool) ⇒ bool
  where Ball A P ⟷ (∀x. x ∈ A ⟹ P x) — bounded universal quantifiers
THEORY "Set"

definition Bex :: 'a set ⇒ ('a ⇒ bool) ⇒ bool
where Bex A P ⇒ (∃ x. x ∈ A ∧ P x) — bounded existential quantifiers

syntax (ASCII)
-Ball :: pattrn ⇒ 'a set ⇒ bool ⇒ bool ((3ALL (-/-)/ -) [0, 0, 10] 10)
-Bex :: pattrn ⇒ 'a set ⇒ bool ⇒ bool ((3EX (-/-)/ -) [0, 0, 10] 10)
-Bex1 :: pattrn ⇒ 'a set ⇒ bool ⇒ bool ((3EX! (-/-)/ -) [0, 0, 10] 10)
-Bleast :: id ⇒ 'a set ⇒ bool ⇒ 'a ((3LEAST (-/-)/ -) [0, 0, 10] 10)

syntax (input)
-Ball :: pattrn ⇒ 'a set ⇒ bool ⇒ bool ((∃! (-/-)/ -) [0, 0, 10] 10)
-Bex :: pattrn ⇒ 'a set ⇒ bool ⇒ bool ((∃? (-/-)/ -) [0, 0, 10] 10)
-Bex1 :: pattrn ⇒ 'a set ⇒ bool ⇒ bool ((∃?! (-/-)/ -) [0, 0, 10] 10)

syntax
-Ball :: pattrn ⇒ 'a set ⇒ bool ⇒ bool ((3∀ (-/-)/ -) [0, 0, 10] 10)
-Bex :: pattrn ⇒ 'a set ⇒ bool ⇒ bool ((3∃ (-/-)/ -) [0, 0, 10] 10)
-Bex1 :: pattrn ⇒ 'a set ⇒ bool ⇒ bool ((3∃! (-/-)/ -) [0, 0, 10] 10)
-Bleast :: id ⇒ 'a set ⇒ bool ⇒ 'a ((3LEAST (-/-)/ -) [0, 0, 10] 10)

translations
∀ x∈A. P ⇒ CONST Ball A (λx. P)
∃ x∈A. P ⇒ CONST Bex A (λx. P)
∃!x∈A. P ⇒ ∃!x. x ∈ A ∧ P
          LEAST x:A. P ⇒ LEAST x. x ∈ A ∧ P

syntax (ASCII output)
-setlessAll :: [idt, 'a, bool] ⇒ bool ((3∀<-/ -) [0, 0, 10] 10)
-setlessEx :: [idt, 'a, bool] ⇒ bool ((3∃<-/ -) [0, 0, 10] 10)
-setleAll :: [idt, 'a, bool] ⇒ bool ((3∀<=-/ -) [0, 0, 10] 10)
-setleEx :: [idt, 'a, bool] ⇒ bool ((3∃<=-/ -) [0, 0, 10] 10)
-setleEx1 :: [idt, 'a, bool] ⇒ bool ((3∃!<=-/ -) [0, 0, 10] 10)

translations
∀ A⊆B. P ⇒ ∀ A. A ⊆ B → P
∃ A⊆B. P ⇒ ∃ A. A ⊆ B ∧ P
∀ A ⊈ B. P ⇒ ∀ A. A ⊈ B → P
∃ A ⊈ B. P ⇒ ∃ A. A ⊈ B ∧ P
∃! A ⊈ B. P ⇒ ∃! A. A ⊈ B ∧ P
print-translation :

let
  val All-binder = Mixfix.binder-name const-syntax All;
  val Ex-binder = Mixfix.binder-name const-syntax Ex;
  val impl = const-syntax HOL.implies;
  val conj = const-syntax HOL.conj;
  val sbset = const-syntax subset;
  val sbset-eq = const-syntax subset-eq;

  val trans =
    [((All-binder, impl, sbset), syntax-const -setlessAll),
     ((All-binder, impl, sbset-eq), syntax-const -setleAll),
     ((Ex-binder, conj, sbset), syntax-const -setlessEx),
     ((Ex-binder, conj, sbset-eq), syntax-const -setleEx)];

  fun mk v (v', T) c n P =
    if v = v' andalso not (Term.exists-subterm (fn Free (x, -) => x = v | - => false) n)
      then Syntax.const c $ Syntax-Trans.mark-bound-body (v', T) $ n $ P
    else raise Match;

  fun tr' q = (q, fn - =>
    (fn [Const (syntax-const -bound, -) $ Free (v, Type (type-name (set), -)),
         Const (c, -) $
           (Const (d, -) $ (Const (syntax-const -bound, -) $ Free (v', T)) $ n)$ P] =>
             case AList.lookup (=) trans (q, c, d) of
               NONE => raise Match
             | SOME l => mk v (v', T) l n P)
     | - => raise Match));

  fun nvars (Const (syntax-const -idts, -) $ - $ idts) = nvars idts + 1
     | nvars - = 1;

  fun setcompr-tr ctxt [e, idts, b] =
    let
      val ex-tr = snd (Syntax-Trans.mk-binder-tr (EX, const-syntax Ex));

      fun nvars (Const (syntax-const -idts, -) $ - $ idts) = nvars idts + 1
          | nvars - = 1;

      fun setcompr-tr ctxt [e, idts, b] =
        let
          val ex-tr = snd (Syntax-Trans.mk-binder-tr (EX, const-syntax Ex));
val eq = Syntax.const const-syntax(HOL.eq) $ Bound (nvars idts) $ e;
val P = Syntax.const const-syntax(HOL.conj) $ eq $ b;
val exP = ex-tr ctxt [idts, P];
in Syntax.const const-syntax(Collect) $ absdummy dummyT exP end;
in [(syntax-const(-Setcompr), setcompr-tr)] end

print-translation :
[Syntax-Trans.preserve-binder-abs2-tr' const-syntax(Ball), syntax-const(-Ball),
Syntax-Trans.preserve-binder-abs2-tr' const-syntax(Bex), syntax-const(-Bex)]

— to avoid eta-contraction of body

print-translation :
let
val ex-tr' = snd (Syntax-Trans.mk-binder-tr' (const-syntax(Ex), DUMMY));

fun setcompr-tr' ctxt [Abs (abs as (_, _, P))] = let
  fun check (Const (const-syntax(Ex), _) $ Abs (_, _, P), n) = check (P, n + 1)
    | check (Const (const-syntax(HOL.conj), _) $
      (Const (const-syntax(HOL.eq), _) $ Bound m $ e) $ P, n) =
      n > 0 andalso m = n andalso not (loose-bvar1 (P, n)) andalso
      subset (=) (0 upto (n - 1), add-loose-bnos (e, 0, []))
    | check _ = false;

  fun tr' (- $ abs) = let val - $ idts $ (- $ (- $ e) $ Q) = ex-tr' ctxt [abs]
    in Syntax.const syntax-const(-Setcompr) $ e $ idts $ Q end;
  in if check (P, 0) then tr' P else
    let
      val (x as - $ Free(xN, _), t) = Syntax-Trans.atomic-abs-tr' abs;
      val M = Syntax.const syntax-const(-Coll) $ x $ t;
      in case t of
        Const (const-syntax(HOL.conj), _) $
          (Const (const-syntax(Set.member), _) $
            (Const (syntax-const(-bound), _) $ Free (yN, _)) $ A) $ P =>
          if xN = yN then Syntax.const syntax-const(-Collect) $ x $ A $ P else M
        _ => M
    end
  end;
in [(const-syntax(Collect), setcompr-tr')] end

simproc-setup defined-Bex ($\exists x \in A. \ P x \land Q x$) = 
(fn - => Quantifier1.rearrange-bex
(fn ctxt =>
  unfold-tac ctxt @{thms Bex-def} THEN
  Quantifier1.prove-one-point-ex-tac ctxt)
)

simproc-setup defined-All ($\forall x \in A. \ P x \longrightarrow Q x$) = 
(fn - => Quantifier1.rearrange-ball
(fn ctxt =>
  unfold-tac ctxt @{thms Ball-def} THEN
  Quantifier1.prove-one-point-all-tac ctxt)
)

lemma ballI [intro!]: ($\forall x. \ x \in A \implies P x$) \implies \forall x \in A. \ P x
by (simp add: Ball-def)

lemmas strip = impI allI ballI

lemma bspec [dest?]: $\forall x \in A. \ P x \implies x \in A \implies P x$
by (simp add: Ball-def)

Gives better instantiation for bound:

setup :
  map-theory-claset (fn ctxt =>
    ctxt addbefore (bspec, fn ctxt' =>
      dresolve-tac ctxt' @{thms bspec} THEN'
      assume-tac ctxt'))

ML :
structure Simpdata =
  struct
    open Simpdata;
    val mksimps-pairs = [(const-name Ball, @{thms bspec})] @ mksimps-pairs;
  end;

open Simpdata;

declaration (fn - => Simplifier.map-ss (Simplifier.set-mksims (mksimps mksims-pairs)))

lemma ballE [elim]: $\forall x \in A. \ P x \implies (P x \implies Q) \implies (x \notin A \implies Q) \implies Q$
unfolding Ball-def by blast

lemma bexI [intro]: $P x \implies x \in A \implies \exists x \in A. \ P x$
— Normally the best argument order: $P x$ constrains the choice of $x \in A.$
unfolding Bex-def by blast

lemma rev-bexI [intro?]: $x \in A \implies P x \implies \exists x \in A. \ P x$
— The best argument order when there is only one $x \in A$.

**unfolding** **Bex-def** by blast

**lemma** **bexCI**: \((\forall x \in A. \neg P x \implies P a) \implies a \in A \implies \exists x \in A. P x\)

**unfolding** **Bex-def** by blast

**lemma** **bxE** [elim!]: \(\exists x \in A. P x \implies (\forall x \in A \implies P x \implies Q) \implies Q\)

**unfolding** **Bex-def** by blast

**lemma** **ball-triv** [simp]: \((\forall x \in A. P) \iff ((\exists x. x \in A) \implies P)\)

— Trivial rewrite rule.

by (**simp add**: **Ball-def**)

**lemma** **bex-triv** [simp]: \(\exists x \in A. P \iff ((\exists x. x \in A) \land P)\)

— Dual form for existentials.

by (**simp add**: **Bex-def**)

**lemma** **bex-triv-one-point1** [simp]: \(\exists x \in A. x = a \iff a \in A\)

by blast

**lemma** **bex-triv-one-point2** [simp]: \(\exists x \in A. a = x \iff a \in A\)

by blast

**lemma** **bex-one-point1** [simp]: \((\exists x \in A. a \land P x) \iff a \in A \land P a\)

by blast

**lemma** **bex-one-point2** [simp]: \((\exists x \in A. a = x \land P x) \iff a \in A \land P a\)

by blast

**lemma** **ball-one-point1** [simp]: \((\forall x \in A. a = x \implies P x) \iff (a \in A \implies P a)\)

by blast

**lemma** **ball-one-point2** [simp]: \((\forall x \in A. x = a \implies P x) \iff (a \in A \implies P a)\)

by blast

**lemma** **ball-conj-distrib**: \((\forall x \in A. P x \land Q x) \iff (\forall x \in A. P x) \land (\forall x \in A. Q x)\)

by blast

**lemma** **bex-disj-distrib**: \((\exists x \in A. P x \lor Q x) \iff (\exists x \in A. P x) \lor (\exists x \in A. Q x)\)

by blast

**Congruence rules**

**lemma** **ball-cong**:

\[
[ A = B; \\forall x. x \in B \implies P x \iff Q x ] \implies \\
(\forall x \in A. P x) \iff (\forall x \in B. Q x)
\]

by (**simp add**: **Ball-def**)

**lemma** **ball-cong-simp** [cong]:

\[
[ A = B; \\forall x. x \in B =simp=> P x \iff Q x ] \implies 
\]
THEORY “Set”

\[(\forall x \in A. \ P x) \longleftrightarrow (\forall x \in B. \ Q x)\]
by (simp add: simp-implies-def Ball-def)

lemma bex-cong:
\[\begin{array}{l}
A = B; \ \land x. \ x \in B \implies P x \longleftrightarrow Q x \implies
(\exists x \in A. \ P x) \longleftrightarrow (\exists x \in B. \ Q x)
\end{array}\]
by (simp add: Bex-def cong: conj-cong)

lemma bex-cong-simp:
\[\begin{array}{l}
A = B; \ \land x. \ x \in B = \implies P x \longleftrightarrow Q x
\implies
(\exists x \in A. \ P x) \longleftrightarrow (\exists x \in B. \ Q x)
\end{array}\]
by (simp add: simp-implies-def Bex-def cong: conj-cong)

lemma bex1-def:
\[\begin{array}{l}
(\exists! x \in X. \ P x) \longleftrightarrow (\exists x \in X. \ P x) \land (\forall x \in X. \ \forall y \in X. \ P x \longleftrightarrow P y \longleftrightarrow x = y)
\end{array}\]
by auto

6.3 Basic operations

6.3.1 Subsets

lemma subsetI [intro!]: \(\land x. \ x \in A \implies x \in B\) \(\implies A \subseteq B\)
by (simp add: less-eq-set-def le-fun-def)

Map the type \(\text{’a set} \Rightarrow \text{anything}\) to just \(\text{’a}\); for overloading constants whose first argument has type \(\text{’a set}\).

lemma subsetD [elim, intro?]: \(A \subseteq B \implies c \in A \implies c \in B\)
by (simp add: less-eq-set-def le-fun-def)
— Rule in Modus Ponens style.

lemma rev-subsetD [intro?, no-atp]: \(c \in A \implies A \subseteq B \implies c \in B\)
— The same, with reversed premises for use with erule – cf. \[\begin{array}{l}
?P; \ ?P \longrightarrow ?Q
\implies ?Q
\end{array}\]
by (rule subsetD)

lemma subsetCE [elim, no-atp]: \(A \subseteq B \implies (c \notin A \longrightarrow P) \longrightarrow (c \in B \longrightarrow P)\)
\(\implies P\)
— Classical elimination rule.
by (auto simp add: less-eq-set-def le-fun-def)

lemma subset-eq: \(A \subseteq B \longleftrightarrow (\forall x \in A. \ x \in B)\)
by blast

lemma contra-subsetD [no-atp]: \(A \subseteq B \implies c \notin B \implies c \notin A\)
by blast

lemma subset-refl: \(A \subseteq A\)
by (fact order-refl)
lemma subset-trans: \( A \subseteq B \implies B \subseteq C \implies A \subseteq C \)
by (fact order-trans)

lemma subset-not-subset-eq [code]: \( A \subset B \iff A \subseteq B \land \sim B \subseteq A \)
by (fact less-le-not-le)

lemma eq-mem-trans: \( a = b \implies b \in A \implies a \in A \)
by simp

lemmas basic-trans-rules [trans] =
order-trans-rules rev-subsetD subsetD eq-mem-trans

6.3.2 Equality

lemma subset-antisym [intro!]: \( A \subseteq B \implies B \subseteq A \implies A = B \)
— Anti-symmetry of the subset relation.
by (iprover intro: set-eqI subsetD)

Equality rules from ZF set theory – are they appropriate here?

lemma equalityD1: \( A = B \implies A \subseteq B \)
by simp

lemma equalityD2: \( A = B \implies B \subseteq A \)
by simp

Be careful when adding this to the claset as subset-empty is in the simpset:
\( A = \{\} \) goes to \( \{\} \subseteq A \) and \( A \subseteq \{\} \) and then back to \( A = \{\} \!\)

lemma equalityE: \( A = B \implies (A \subseteq B \implies B \subseteq A \implies P) \implies P \)
by simp

lemma equalityCE [elim]: \( A = B \implies (c \in A \implies c \in B \implies P) \implies (c \notin A \implies c \notin B \implies P) \implies P \)
by blast

lemma eqset-imp-iff: \( A = B \implies x \in A \iff x \in B \)
by simp

lemma eqelem-imp-iff: \( x = y \implies x \in A \iff y \in A \)
by simp

6.3.3 The empty set

lemma empty-def: \( \{\} = \{x. \text{False}\} \)
by (simp add: bot-set-def bot-fun-def)

lemma empty-iff [simp]: \( c \in \{\} \iff \text{False} \)
by (simp add: empty-def)
lemma emptyE [clm!]: $a \in \{\} \implies P$
  by simp

lemma empty-subsetI [iff]: $\{\} \subseteq A$
  — One effect is to delete the ASSUMPTION $\{\} \subseteq A$
  by blast

lemma equals0I: $(\forall y. y \in A \implies False) \implies A = \{\}$
  by blast

lemma equals0D: $A = \{\} \implies a \notin A$
  — Use for reasoning about disjointness: $A \cap B = \{\}$
  by blast

lemma ball-empty [simp]: $\ball \{\} P \iff True$
  by (simp add: Ball-def)

lemma bex-empty [simp]: $\bex \{\} P \iff False$
  by (simp add: Bex-def)

6.3.4 The universal set – UNIV

abbreviation UNIV :: 'a set
  where UNIV $\equiv$ top

lemma UNIV-def: $\text{UNIV} = \{x. \text{True}\}$
  by (simp add: top-set-def top-fun-def)

lemma UNIV-I [simp]: $x \in \text{UNIV}$
  by (simp add: UNIV-def)

declare UNIV-I [intro] — unsafe makes it less likely to cause problems

lemma UNIV-witness [intro?]: $\exists x. x \in \text{UNIV}$
  by simp

lemma subset-UNIV: $A \subseteq \text{UNIV}$
  by (fact top-greatest)

Eta-contracting these two rules (to remove $P$) causes them to be ignored
because of their interaction with congruence rules.

lemma ball-UNIV [simp]: $\ball \text{UNIV} P \iff \text{All} P$
  by (simp add: Ball-def)

lemma bex-UNIV [simp]: $\bex \text{UNIV} P \iff \text{Ex} P$
  by (simp add: Bex-def)

lemma UNIV-eq-I: $(\forall x. x \in A) \implies \text{UNIV} = A$
  by auto
**THEORY “Set”**

**lemma** UNIV-not-empty [iff]: \( \text{UNIV} \neq \{ \} \)
by (blast elim: equalityE)

**lemma** empty-not-UNIV[simp]: \( \{ \} \neq \text{UNIV} \)
by blast

### 6.3.5 The Powerset operator – Pow

**definition** Pow :: 'a set ⇒ 'a set set
where Pow-def: Pow A = \{ B. B ⊆ A \}

**lemma** Pow-iff [iff]: \( A \in \text{Pow } B \iff A \subseteq B \)
by (simp add: Pow-def)

**lemma** PowI: \( A \subseteq B \Longrightarrow A \in \text{Pow } B \)
by (simp add: Pow-def)

**lemma** PowD: \( A \in \text{Pow } B \Longrightarrow A \subseteq B \)
by (simp add: Pow-def)

**lemma** Pow-bottom: \( \{ \} \in \text{Pow } B \)
by simp

**lemma** Pow-top: \( A \in \text{Pow } A \)
by simp

**lemma** Pow-not-empty: Pow A \( \neq \{ \} \)
using Pow-top by blast

### 6.3.6 Set complement

**lemma** Compl-iff [simp]: \( c \in - A \iff c \notin A \)
by (simp add: fun-Compl-def uminus-set-def)

**lemma** ComplI [intro!]: \( (c \in A \Longrightarrow False) \Longrightarrow c \in - A \)
by (simp add: fun-Compl-def uminus-set-def) blast

This form, with negated conclusion, works well with the Classical prover. Negated assumptions behave like formulae on the right side of the notional turnstile . . .

**lemma** ComplD [dest!]: \( c \in - A \Longrightarrow c \notin A \)
by simp

**lemmas** ComplE = ComplD [elim-format]

**lemma** Compl-eq: \( - A = \{ x. \neg x \in A \} \)
by blast
6.3.7 Binary intersection

abbreviation \( \text{inter} :: 'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set} \) (infixl \( \cap \) 70)
where \( (\cap) \equiv \text{inf} \)

notation (ASCII)
inter \( \text{(infixl Int 70)} \)

lemma Int-def: \( A \cap B = \{ x. \ x \in A \land x \in B \} \)
by (simp add: inf-set-def inf-fun-def)

lemma Int-iff \( \text{[simp]} \): \( c \in A \cap B \leftrightarrow c \in A \land c \in B \)
unfolding Int-def by blast

lemma IntI \( \text{[intro]} \): \( c \in A \Rightarrow c \in B \Rightarrow c \in A \cap B \)
by simp

lemma IntD1: \( c \in A \cap B \Rightarrow c \in A \)
by simp

lemma IntD2: \( c \in A \cap B \Rightarrow c \in B \)
by simp

lemma IntE \( \text{[elim]} \): \( c \in A \cap B \Rightarrow (c \in A \Rightarrow c \in B \Rightarrow P) \Rightarrow P \)
by simp

lemma mono-Int: \( \text{mono } f \Rightarrow f (A \cap B) \subseteq f A \cap f B \)
by (fact mono-inf)

6.3.8 Binary union

abbreviation \( \text{union} :: 'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set} \) (infixl \( \cup \) 65)
where union \( \equiv \text{sup} \)

notation (ASCII)
union \( \text{(infixl Un 65)} \)

lemma Un-def: \( A \cup B = \{ x. \ x \in A \lor x \in B \} \)
by (simp add: sup-set-def sup-fun-def)

lemma Un-iff \( \text{[simp]} \): \( c \in A \cup B \leftrightarrow c \in A \lor c \in B \)
unfolding Un-def by blast

lemma UnI1 \( \text{[intro]} \): \( c \in A \Rightarrow c \in A \cup B \)
by simp

lemma UnI2 \( \text{[intro]} \): \( c \in B \Rightarrow c \in A \cup B \)
by simp

Classical introduction rule: no commitment to \( A \) vs. \( B \).
lemma UnCI [intro!]: \( c \notin B \rightarrow c \in A \rightarrow c \in A \cup B \)
by auto

lemma UnE [elim!]: \( c \in A \cup B \rightarrow (c \in A \rightarrow P) \rightarrow (c \in B \rightarrow P) \rightarrow P \)
 unfolding Un-def by blast

lemma insert-def: insert \( a \) \( B \) = \{ \( x \). \( x = a \) \}\( \cup B \)
 by (simp add: insert-compr Un-def)

lemma mono-Un: mono \( f \) = \( f (A \cup B) \subseteq f (A \cup B) \)
 by (fact mono-sup)

6.3.10 Augmenting a set – insert

lemma insert-iff [simp]: \( a \in insert b A \leftrightarrow a = b \lor a \in A \)
 unfolding insert-def by blast

lemma insertI1: \( a \in insert a B \)
 by simp

lemma insertI2: \( a \in B \rightarrow a \in insert b B \)
 by simp

lemma insertE [elim!]: \( a \in insert b A \rightarrow (a = b \rightarrow P) \rightarrow (a \in A \rightarrow P) \rightarrow P \)
 unfolding insert-def by blast
lemma insertCI [intro!]: \((a \notin B \Rightarrow a = b) \Rightarrow a \in \text{insert } b \ B\)
— Classical introduction rule.

by auto

lemma subset-insert-iff: \(A \subseteq \text{insert } x \ B \iff (\text{if } x \in A \text{ then } A - \{x\} \subseteq B \text{ else } A \subseteq B)\)

by auto

lemma set-insert:
assumes \(x \in A\)
obtains \(B\) where \(A = \text{insert } x \ B\) and \(x \notin B\)
proof
show \(A = \text{insert } x \ (A - \{x\})\) using assms by blast
show \(x \notin A - \{x\}\) by blast
qed

lemma insert-ident: \(x \notin A \Longrightarrow x \notin B \Longrightarrow \text{insert } x \ A = \text{insert } x \ B\)
by auto

lemma insert-eq-iff:
assumes \(a \notin A\) \(b \notin B\)
s shows \(\text{insert } a \ A = \text{insert } b \ B \iff (\text{if } a = b \text{ then } A = B \text{ else } \exists \ C. \ A = \text{insert } b \ C \land b \notin \ C \land B = \text{insert } a \ C \land a \notin \ C)\)
(is \(?L \leftrightarrow \ ?R\))
proof
show \(?R\) if \(?L\)
proof (cases \(a = b\))
case True
with assms \(?L\) show \(?R\)
by (simp add: insert-ident)
next
case False
let \(?C = A - \{b\}\)
have \(A = \text{insert } b \ ?C \land b \notin \ ?C \land B = \text{insert } a \ ?C \land a \notin \ ?C\)
using assms \(?L\) \(\langle a \neq b\rangle\) by auto
then show \(?R\) using \(?a \neq b\rangle\) by auto
qed
show \(?L\) if \(?R\)
using that by (auto split: if-splits)
qed

lemma insert-UNIV: \(\text{insert } x \ \text{UNIV} = \text{UNIV}\)
by auto

6.3.11 Singletons, using insert

lemma singletonI [intro!]: \(a \in \{a\}\)
— Redundant? But unlike \(\text{insertCI}\), it proves the subgoal immediately!
by (rule insertII)

**lemma** singletonD [dest!]: \( b \in \{a\} \implies b = a \)
by blast

**lemmas** singletonE = singletonD [elim-format]

**lemma** singleton-iff: \( b \in \{a\} \iff b = a \)
by blast

**lemma** singleton-inject [dest!]: \( \{a\} = \{b\} \implies a = b \)
by blast

**lemma** singleton-insert-inj-iff [iff]: \( \{b\} = \text{insert } a \ A \iff a = b \land A \subseteq \{b\} \)
by blast

**lemma** subset-singletonD: \( A \subseteq \{x\} \implies A = \{} \lor A = \{x\} \)
by fast

**lemma** subset-singleton-iff: \( X \subseteq \{a\} \iff X = \{} \lor X = \{a\} \)
by blast

**lemma** singleton-conv [simp]: \( \{x. \ x = a\} = \{a\} \)
by blast

**lemma** singleton-conv2 [simp]: \( \{x. \ a = x\} = \{a\} \)
by blast

**lemma** Diff-single-insert: \( A - \{x\} \subseteq B \implies A \subseteq \text{insert } x \ B \)
by blast

**lemma** subset-Diff-insert: \( A \subseteq B - \text{insert } x \ C \iff A \subseteq B - C \land x \notin A \)
by blast

**lemma** doubleton-eq-iff: \( \{a, b\} = \{c, d\} \iff a = c \land b = d \lor a = d \land b = c \)
by (blast elim: equalityE)

**lemma** Un-singleton-iff: \( A \cup B = \{x\} \iff A = \{} \land B = \{x\} \lor A = \{x\} \land B = \{} \)
by auto

**lemma** singleton-Un-iff: \( \{x\} = A \cup B \iff A = \{} \land B = \{x\} \lor A = \{x\} \land B = \{} \)
by auto
6.3.12 Image of a set under a function

Frequently $b$ does not have the syntactic form of $f x$.

**definition** image :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'b set  (**infixr** ' 90)

where $f \cdot A = \{ y. \exists x \in A. y = f x \}$

**lemma** image-eqI [simp, intro]: $b = f x \implies x \in A \implies b \in f \cdot A$

**unfolding** image-def by blast

**lemma** imageI: $x \in A \implies f x \in f \cdot A$

by (rule image-eqI) (rule refl)

**lemma** rev-image-eqI: $x \in A \implies b = f x \implies b \in f \cdot A$

— This version’s more effective when we already have the required $x$.

by (rule image-eqI)

**lemma** imageE [elim!]:

assumes $b \in (\lambda x. f x) \cdot A$ — The eta-expansion gives variable-name preservation.

obtains $x$ where $b = f x$ and $x \in A$

using assms unfolding image-def by blast

**lemma** Compr-image-eq: $\{ x \in f \cdot A. P x \} = f \cdot \{ x \in A. P (f x) \}$

by auto

**lemma** image-Un: $f \cdot (A \cup B) = f \cdot A \cup f \cdot B$

by blast

**lemma** image-iff: $z \in f \cdot A \iff (\exists x \in A. z = f x)$

by blast

**lemma** image-subsetI: $(\forall x. x \in A \implies f x \in B) \implies f \cdot A \subseteq B$

— Replaces the three steps subsetI, imageE, hypsubst, but breaks too many existing proofs.

by blast

**lemma** image-subset-iff: $f \cdot A \subseteq B \iff (\forall x \in A. f x \in B)$

— This rewrite rule would confuse users if made default.

by blast

**lemma** subset-imageE:

assumes $B \subseteq f \cdot A$

obtains $C$ where $C \subseteq A$ and $B = f \cdot C$

**proof**

from assms have $B = f \cdot \{ a \in A. f a \in B \}$ by fast

moreover have $\{ a \in A. f a \in B \} \subseteq A$ by blast

ultimately show thesis by (blast intro: that)

qed

**lemma** subset-image-iff: $B \subseteq f \cdot A \iff (\exists AA \subseteq A. B = f \cdot AA)$
by (blast elim: subset-imageE)

**lemma** image-ident [simp]: \((\lambda x. x)' Y = Y\)
by blast

**lemma** image-empty [simp]: \(f' \{\} = \{\}\)
by blast

**lemma** image-insert [simp]: \(f' insert a B = insert (f a) (f' B)\)
by blast

**lemma** image-constant: \(x \in A \implies (\lambda x. c)' A = \{c\}\)
by auto

**lemma** image-constant-conv: \((\lambda x. c)' A = (if A = \{\} then \{\} else \{c\})\)
by auto

**lemma** image-image: \(f' (g' A) = (\lambda x. f(g x))' A\)
by blast

**lemma** insert-image [simp]: \(x \in A \implies insert (f x) (f' A) = f' A\)
by blast

**lemma** image-is-empty [iff]: \(f' A = \{\} \iff A = \{\}\)
by blast

**lemma** empty-is-image [iff]: \(\{\} = f' A \iff A = \{\}\)
by blast

**lemma** image-Collect: \(f' \{x. P x\} = \{f x | x. P x\}\)
— NOT suitable as a default simp rule: the RHS isn’t simpler than the LHS,
with its implicit quantifier and conjunction. Also image enjoys better equational
properties than does the RHS.
by blast

**lemma** if-image-distrib [simp]:
\((\lambda x. if P x then f x else g x)' S = f' (S \cap \{x. P x\}) \cup g' (S \cap \{x. \neg P x\})\)
by auto

**lemma** image-cong:
\(f' M = g' N\) if \(M = N \land x. x \in N \implies f x = g x\)
using that by (simp add: image-def)

**lemma** image-cong-simp [cong]:
\(f' M = g' N\) if \(M = N \land x. x \in N \implies f x = g x\)
using that image-cong [of M N f g] by (simp add: simp-implies-def)

**lemma** image-Int-subset: \(f' (A \cap B) \subseteq f' A \cap f' B\)
by blast
lemma image-diff-subset: \( f \setminus A - f \setminus B \subseteq f \setminus (A - B) \)
by blast

lemma Setcompr-eq-image: \( \{ f \, x \mid x \in A \} = f \setminus A \)
by blast

lemma setcompr-eq-image: \( \{ f \, x \mid x \in A \} = f \setminus \{ x \mid P \} \)
by auto

lemma ball-imageD: \( \forall x \in f \setminus A. P x \Rightarrow \forall x \in A. P (f x) \)
by simp

lemma bex-imageD: \( \exists x \in f \setminus A. P x \Rightarrow \exists x \in A. P (f x) \)
by auto

lemma image-add-0 [simp]: \( (+) (0 \, :: \text{comm-monoid-add}) \setminus S = S \)
by auto

Range of a function – just an abbreviation for image!

abbreviation range :: \( (\forall x \cdot \text{fun}) \Rightarrow \text{set} \)
where range \( f \equiv f \setminus \text{UNIV} \)

lemma range-eqI: \( b = f x \Rightarrow b \in \text{range} \, f \)
by simp

lemma rangeI: \( f x \in \text{range} \, f \)
by simp

lemma rangeE [elim?]: \( b \in \text{range} (\lambda x. f x) \Rightarrow (\forall x. b = f x \Rightarrow P) \Rightarrow P \)
by (rule imageE)

lemma full-SetCompr-eq: \( \{ u. \, \exists x. u = f x \} = \text{range} \, f \)
by auto

lemma range-composition: range (\lambda x. f \, (g x)) = f \setminus \text{range} \, g
by auto

lemma range-constant [simp]: range (\lambda x. x) = \{ x \}
by (simp add: image-constant)

lemma range-eq-singletonD: \( \text{range} \, f = \{ a \} \Rightarrow f x = a \)
by auto

6.3.13 Some rules with if

Elimination of \( \{ x. \ldots \land x = t \land \ldots \} \).

lemma Collect-conv-if: \( \{ x. x = a \land P \} = (if P \ a \ then \ \{ a \} \ else \ \{ \}) \)
by auto

lemma Collect-conv-if2: \{x. \ a = x \land \ P x\} = (if \ P \ a \ then \ \{a\} \ else \ \{\})
  by auto

Rewrite rules for boolean case-splitting: faster than if-split [split].

lemma if-split-eq1: (if \ Q \ then \ x \ else \ y) = b \iff (Q \imp x = b) \land (\neg Q \imp y = b)
  by (rule if-split)

lemma if-split-eq2: a = (if \ Q \ then \ x \ else \ y) \iff (Q \imp a = x) \land (\neg Q \imp a = y)
  by (rule if-split)

Split ifs on either side of the membership relation. Not for [simp] – can cause goals to blow up!

lemma if-split-mem1: (if \ Q \ then \ x \ else \ y) \in b \iff (Q \imp x \in b) \land (\neg Q \imp y \in b)
  by (rule if-split)

lemma if-split-mem2: a \in (if \ Q \ then \ x \ else \ y) \iff (Q \imp a \in x) \land (\neg Q \imp a \in y)
  by (rule if-split [where \ P = \lambda S. \ a \in S])

lemmas split-ifs = if-bool-eq-conj if-split-eq1 if-split-eq2 if-split-mem1 if-split-mem2

6.4 Further operations and lemmas

6.4.1 The “proper subset” relation

lemma psubsetI [intro!]: A \subseteq B \imp A \neq B \equiv A \subset B
  unfolding less-le by blast

lemma psubsetE [elim!]: A \subseteq B \imp (A \subseteq B \equiv \neg B \subseteq A \equiv R) \equiv R
  unfolding less-le by blast

lemma psubset-insert-iff:
  A \subseteq insert x B \iff (if \ x \in B \ then A \subseteq B \ else if \ x \in A \ then A - \{x\} \subseteq B \ else A \subseteq B)
  by (auto simp add: less-le subset-insert-iff)

lemma psubset-eq: A \subseteq B \iff A \subseteq B \land A \neq B
  by (simp only: less-le)

lemma psubset-imp-subset: A \subseteq B \equiv A \subseteq B
  by (simp add: psubset-eq)

lemma psubset-trans: A \subseteq B \equiv B \subseteq C \equiv A \subseteq C
  unfolding less-le by (auto dest: subset-antisym)
lemma \( psubsetD \): \( A \subseteq B \Rightarrow c \in A \Rightarrow c \in B \)
unfolding less-le by (auto dest: subsetD)

lemma \( psubset-subset-trans \): \( A \subseteq B \Rightarrow B \subseteq C \Rightarrow A \subseteq C \)
by (auto simp add: psubset-eq)

lemma \( subset-psubset-trans \): \( A \subseteq B \Rightarrow B \subset C \Rightarrow A \subset C \)
by (auto simp add: psubset-eq)

lemma \( psubset-imp-ex-mem \): \( A \subseteq B \Rightarrow \exists b. b \in B - A \)
unfolding less-le by blast

lemma atomize-ball: \( \forall x. x \in A \Rightarrow P x \) \( \equiv \) \( \forall x \in A. P x \)
by (simp only: Ball-def atomize-all atomize-imp)

lemmas [symmetric, rulify] = atomize-ball
and [symmetric, defn] = atomize-ball

lemma image-Pow-mono: \( f ' A \subseteq B \Rightarrow image f ' Pow A \subseteq Pow B \)
by blast

lemma image-Pow-surj: \( f ' A = B \Rightarrow image f ' Pow A = Pow B \)
by (blast elim: subset-imageE)

6.4.2 Derived rules involving subsets.

insert.

lemma subset-insertI: \( B \subseteq insert a B \)
by (rule subsetI) (erule insertI2)

lemma subset-insertI2: \( A \subseteq B \Rightarrow A \subseteq insert b B \)
by blast

lemma subset-insert: \( x \not\in A \Rightarrow A \subseteq insert x B \iff A \subseteq B \)
by blast

Finite Union – the least upper bound of two sets.

lemma Un-upper1: \( A \subseteq A \cup B \)
by (fact sup-ge1)

lemma Un-upper2: \( B \subseteq A \cup B \)
by (fact sup-ge2)

lemma Un-least: \( A \subseteq C \Rightarrow B \subseteq C \Rightarrow A \cup B \subseteq C \)
by (fact sup-least)

Finite Intersection – the greatest lower bound of two sets.
lemma Int-lower1: \( A \cap B \subseteq A \)
by (fact inf-le1)

lemma Int-lower2: \( A \cap B \subseteq B \)
by (fact inf-le2)

lemma Int-greatest: \( C \subseteq A \implies C \subseteq B \implies C \subseteq A \cap B \)
by (fact inf-greatest)

Set difference.

lemma Diff-subset[simp]: \( A - B \subseteq A \)
by blast

lemma Diff-subset-conv: \( A - B \subseteq C \iff A \subseteq B \cup C \)
by blast

6.4.3 Equalities involving union, intersection, inclusion, etc.

\{\}\.

lemma Collect-const [simp]: \( \{s.\ P\} = (if \ P\ then\ UNIV\ else\ \{\}) \)
— supersedes Collect-False-empty
by auto

lemma subset-empty [simp]: \( A \subseteq \{\} \iff A = \{\} \)
by (fact bot-unique)

lemma not-psubset-empty [iff]: \( \neg( A < \{\}) \)
by (fact not-less-bot)

lemma Collect-subset [simp]: \( \{x\in A.\ P\ x\} \subseteq A \) by auto

lemma Collect-empty-eq [simp]: Collect P = { } \iff (\forall x. \neg P \ x)
by blast

lemma empty-Collect-eq [simp]: \{\} = Collect P \iff (\forall x. \neg P \ x)
by blast

lemma Collect-neg-eq: \( \{x. \neg P \ x\} = -\{x. P \ x\} \)
by blast

lemma Collect-disj-eq: \( \{x. P \ x \vee Q \ x\} = \{x. P \ x\} \cup \{x. Q \ x\} \)
by blast

lemma Collect-imp-eq: \( \{x. P \ x \implies Q \ x\} = -\{x. P \ x\} \cup \{x. Q \ x\} \)
by blast

lemma Collect-conj-eq: \( \{x. P \ x \land Q \ x\} = \{x. P \ x\} \cap \{x. Q \ x\} \)
by blast
lemma Collect-mono-iff: Collect P ⊆ Collect Q ←→ (∀ x. P x −→ Q x)
  by blast

insert.

lemma insert-is-Un: insert a A = {a} ∪ A
  — NOT SUITABLE FOR REWRITING since {a} ≡ insert a {}
  by blast

lemma insert-not-empty [simp]: insert a A ≠ {}
  and empty-not-insert [simp]: {} ≠ insert a A
  by blast+

lemma insert-absorb: a ∈ A ⇒ insert a A = A
  — [simp] causes recursive calls when there are nested inserts
  — with quadratic running time
  by blast

lemma insert-absorb2 [simp]: insert x (insert x A) = insert x A
  by blast

lemma insert-commute: insert x (insert y A) = insert y (insert x A)
  by blast

lemma insert-subset [simp]: insert x A ⊆ B ←→ x ∈ B ∧ A ⊆ B
  by blast

lemma mk-disjoint-insert: a ∈ A ⇒ ∃ B. A = insert a B ∧ a /∈ B
  — use new B rather than A − {a} to avoid infinite unfolding
  by (rule exI [where x = A − {a}]) blast

lemma insert-Collect: insert a (Collect P) = {u. u ≠ a −→ P u}
  by auto

lemma insert-inter-insert [simp]: insert a A ∩ insert a B = insert a (A ∩ B)
  by blast

lemma insert-disjoint [simp]:
  insert a A ∩ B = {}
  insert a A ∩ B = {}

lemma disjoint-insert [simp]:
  B ∩ insert a A = {}
  B ∩ insert a A = {}

Int
lemma Int-absorb: $A \cap A = A$
  by (fact inf-idem)

lemma Int-left-absorb: $A \cap (A \cap B) = A \cap B$
  by (fact inf-left-idem)

lemma Int-commute: $A \cap B = B \cap A$
  by (fact inf-commute)

lemma Int-left-commute: $A \cap (B \cap C) = B \cap (A \cap C)$
  by (fact inf-left-commute)

lemma Int-assoc: $(A \cap B) \cap C = A \cap (B \cap C)$
  by (fact inf-assoc)

lemmas Int-ac = Int-assoc Int-left-absorb Int-commute Int-left-commute

— Intersection is an AC-operator

lemma Int-absorb1: $B \subseteq A \implies A \cap B = B$
  by (fact inf-absorb2)

lemma Int-absorb2: $A \subseteq B \implies A \cap B = A$
  by (fact inf-absorb1)

lemma Int-empty-left: $\{\} \cap B = \{\}$
  by (fact inf-bot-left)

lemma Int-empty-right: $A \cap \{\} = \{\}$
  by (fact inf-bot-right)

lemma disjoint-eq-subset-Compl: $A \cap B = \{\} \iff A \subseteq -B$
  by blast

lemma disjoint-iff-not-equal: $A \cap B = \{\} \iff (\forall x \in A. \ \forall y \in B. \ x \neq y)$
  by blast

lemma Int-UNIV-left: $\text{UNIV} \cap B = B$
  by (fact inf-top-left)

lemma Int-UNIV-right: $A \cap \text{UNIV} = A$
  by (fact inf-top-right)

lemma Int-Un-distrib: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
  by (fact inf-sup-distrib1)

lemma Int-Un-distrib2: $(B \cup C) \cap A = (B \cap A) \cup (C \cap A)$
  by (fact inf-sup-distrib2)

lemma Int-UNIV [simp]: $A \cap B = \text{UNIV} \iff A = \text{UNIV} \land B = \text{UNIV}$
by (fact inf-eq-top-iff)

lemma Int-subset-iff [simp]: \( C \subseteq A \cap B \iff C \subseteq A \land C \subseteq B \)
by (fact le-inf-iff)

lemma Int-Collect: \( x \in A \cap \{ x. \ P x \} \iff x \in A \land P x \)
by blast

Un.

lemma Un-absorb: \( A \cup A = A \)
by (fact sup-idem)

lemma Un-left-absorb: \( A \cup (A \cup B) = A \cup B \)
by (fact sup-left-idem)

lemma Un-commute: \( A \cup B = B \cup A \)
by (fact sup-commute)

lemma Un-left-commute: \( A \cup (B \cup C) = B \cup (A \cup C) \)
by (fact sup-left-commute)

lemma Un-assoc: \( (A \cup B) \cup C = A \cup (B \cup C) \)
by (fact sup-assoc)

lemmas Un-ac = Un-assoc Un-left-absorb Un-commute Un-left-commute
— Union is an AC-operator

lemma Un-absorb1: \( A \subseteq B \implies A \cup B = B \)
by (fact sup-absorb2)

lemma Un-absorb2: \( B \subseteq A \implies A \cup B = A \)
by (fact sup-absorb1)

lemma Un-empty-left: \( \emptyset \cup B = B \)
by (fact sup-bot-left)

lemma Un-empty-right: \( A \cup \emptyset = A \)
by (fact sup-bot-right)

lemma Un-UNIV-left: \( \text{UNIV} \cup B = \text{UNIV} \)
by (fact sup-top-left)

lemma Un-UNIV-right: \( A \cup \text{UNIV} = \text{UNIV} \)
by (fact sup-top-right)

lemma Un-insert-left [simp]: \( \text{insert a B} \cup C = \text{insert a} \ (B \cup C) \)
by blast

lemma Un-insert-right [simp]: \( A \cup (\text{insert a B}) = \text{insert a} \ (A \cup B) \)
by blast

lemma Int-insert-left: \( (\text{insert } a \ B) \cap C = (\text{if } a \in C \text{ then } \text{insert } a \ (B \cap C) \text{ else } B \cap C) \)
  by auto

lemma Int-insert-left-if0 [simp]: \( a \notin C \implies (\text{insert } a \ B) \cap C = B \cap C \)
  by auto

lemma Int-insert-left-if1 [simp]: \( a \in C \implies (\text{insert } a \ B) \cap C = \text{insert } a \ (B \cap C) \)
  by auto

lemma Int-insert-right: \( A \cap (\text{insert } a \ B) = (\text{if } a \in A \text{ then } \text{insert } a \ (A \cap B) \text{ else } A \cap B) \)
  by auto

lemma Int-insert-right-if0 [simp]: \( a \notin A \implies A \cap (\text{insert } a \ B) = A \cap B \)
  by auto

lemma Int-insert-right-if1 [simp]: \( a \in A \implies A \cap (\text{insert } a \ B) = \text{insert } a \ (A \cap B) \)
  by auto

lemma Un-Int-distrib: \( A \cup (B \cap C) = (A \cup B) \cap (A \cup C) \)
  by (fact sup-inf-distrib1)

lemma Un-Int-distrib2: \( (B \cap C) \cup A = (B \cup A) \cap (C \cup A) \)
  by (fact sup-inf-distrib2)

lemma Un-Int-crazy: \( (A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A) \)
  by blast

lemma subset-Un-eq: \( A \subseteq B \iff A \cup B = B \)
  by (fact le-iff-sup)

lemma Un-empty [iff]: \( A \cup B = \{\} \iff A = \{\} \land B = \{\} \)
  by (fact sup-eq-bot-iff)

lemma Un-subset-iff [simp]: \( A \cup B \subseteq C \iff A \subseteq C \land B \subseteq C \)
  by (fact le-sup-iff)

lemma Un-Diff-Int: \( (A - B) \cup (A \cap B) = A \)
  by blast

lemma Diff-Int2: \( A \cap C - B \cap C = A \cap C - B \)
  by blast

lemma subset-UnE:
assumes $C \subseteq A \cup B$

obtains $A' B'$ where $A' \subseteq A B' \subseteq B C = A' \cup B'$

proof

show $C \cap A \subseteq A C \cap B \subseteq B C = (C \cap A) \cup (C \cap B)$

using assms by blast+

qed

Set complement

lemma Compl-disjoint [simp]: $A \cap -A = {}$

by (fact inf-compl-bot)

lemma Compl-disjoint2 [simp]: $-A \cap A = {}$

by (fact compl-inf-bot)

lemma Compl-partition: $A \cup -A = \text{UNIV}$

by (fact sup-compl-top)

lemma Compl-partition2: $-A \cup A = \text{UNIV}$

by (fact compl-sup-top)

lemma double-complement: $-(-A) = A$ for $A :: 'a set$

by (fact double-compl)

lemma Compl-Un: $-(A \cup B) = (-A) \cap (-B)$

by (fact compl-sup)

lemma Compl-Int: $-(A \cap B) = (-A) \cup (-B)$

by (fact compl-inf)

lemma subset-Compl-self-eq: $A \subseteq -A \iff A = {}$

by blast

lemma Un-Int-assoc-eq: $(A \cap B) \cup C = A \cap (B \cup C) \iff C \subseteq A$

— Halmos, Naive Set Theory, page 16.

by blast

lemma Compl-UNIV-eq: $-\text{UNIV} = {}$

by (fact compl-top-eq)

lemma Compl-empty-eq: $\{\} = \text{UNIV}$

by (fact compl-bot-eq)

lemma Compl-subset-Compl-iff [iff]: $A \subseteq -B \iff B \subseteq A$

by (fact compl-le-compl-iff)

lemma Compl-eq-Compl-iff [iff]: $A = -B \iff A = B$

for $A B :: 'a set$

by (fact compl-eq-compl-iff)
lemma Compl-insert: $- insert x A = (- A) - \{x\}$
  by blast

Bounded quantifiers.
The following are not added to the default simpset because (a) they duplicate
the body and (b) there are no similar rules for Int.

lemma ball-Un: $(\forall x \in A \cup B. P x) \iff (\forall x\in A. P x) \land (\forall x\in B. P x)$
  by blast

lemma bex-Un: $(\exists x \in A \cup B. P x) \iff (\exists x\in A. P x) \lor (\exists x\in B. P x)$
  by blast

Set difference.

lemma Diff-eq: $A - B = A \cap (- B)$
  by blast

lemma Diff-eq-empty-iff [simp]: $A - B = \{\} \iff A \subseteq B$
  by blast

lemma Diff-cancel [simp]: $A - A = \{\}$
  by blast

lemma Diff-idemp [simp]: $(A - B) - B = A - B$
  for $A B :: 'a set$
  by blast

lemma Diff-triv: $A \cap B = \{\} \Rightarrow A - B = A$
  by (blast elim: equalityE)

lemma empty-Diff [simp]: $\{\} - A = \{\}$
  by blast

lemma Diff-empty [simp]: $A - \{\} = A$
  by blast

lemma Diff-UNIV [simp]: $A - UNIV = \{\}$
  by blast

lemma Diff-insert0 [simp]: $x \notin A \Rightarrow A - insert x B = A - B$
  by blast

lemma Diff-insert: $A - insert a B = A - B - \{a\}$
  — NOT SUITABLE FOR REWRITING since $\{a\} \equiv insert a 0$
  by blast

lemma Diff-insert2: $A - insert a B = A - \{a\} - B$
  — NOT SUITABLE FOR REWRITING since $\{a\} \equiv insert a 0$
  by blast
lemma insert-Diff-if: insert x A − B = (if x ∈ B then A − B else insert x (A − B))
  by auto

lemma insert-Diff1 [simp]: x ∈ B ⊢ insert x A − B = A − B
  by blast

lemma insert-Diff-single [simp]: insert a (A − {a}) = insert a A
  by blast

lemma insert-Diff: a ∈ A ⊢ insert a (A − {a}) = A
  by blast

lemma Diff-insert-absorb: x /∈ A ⊢ (insert x A) − {x} = A
  by auto

lemma Diff-disjoint [simp]: A ∩ (B − A) = {}
  by blast

lemma Diff-partition: A ⊆ B ⊢ A ∪ (B − A) = B
  by blast

lemma double-diff: A ⊆ B ⊢ B ⊆ C ⊢ B − (C − A) = A
  by blast

lemma Un-Diff-cancel [simp]: A ∪ (B − A) = A ∪ B
  by blast

lemma Un-Diff-cancel2 [simp]: (B − A) ∪ A = B ∪ A
  by blast

lemma Diff-Un: A − (B ∪ C) = (A − B) ∩ (A − C)
  by blast

lemma Diff-Int: A − (B ∩ C) = (A − B) ∪ (A − C)
  by blast

lemma Diff-Diff-Int: A − (A − B) = A ∩ B
  by blast

lemma Un-Diff: (A ∪ B) − C = (A − C) ∪ (B − C)
  by blast

lemma Int-Diff: (A ∩ B) − C = A ∩ (B − C)
  by blast

lemma Diff-Int-distrib: C ∩ (A − B) = (C ∩ A) − (C ∩ B)
  by blast
lemma Diff-Int-distrib2: \[(A - B) \cap C = (A \cap C) - (B \cap C)\]
by blast

lemma Diff-Compl [simp]: \[A - (-B) = A \cap B\]
by auto

lemma Compl-Diff-eq [simp]: \[-(A - B) = -A \cup B\]
by blast

lemma subset-Compl-singleton [simp]: \[A \subseteq -\{b\} \iff b \notin A\]
by blast

Quantification over type \(\text{bool}\).

lemma bool-induct: \[P \text{ True} \implies P \text{ False} \implies P \text{ x}\]
by (cases x) auto

lemma all-bool-eq: \[(\forall b. P \text{ b}) \iff P \text{ True} \land P \text{ False}\]
by (auto intro: bool-induct)

lemma bool-contrapos: \[P \text{ x} \implies \neg P \text{ False} \implies P \text{ True}\]
by (cases x) auto

lemma ex-bool-eq: \[(\exists b. P \text{ b}) \iff P \text{ True} \lor P \text{ False}\]
by (auto intro: bool-contrapos)

lemma UNIV-bool: \[\text{UNIV} = \{\text{False, True}\}\]
by (auto intro: bool-induct)

Pow

lemma Pow-empty [simp]: \[\text{Pow } \{\} = \{\{\}\}\]
by (auto simp add: Pow-def)

lemma Pow-singleton-iff [simp]: \[\text{Pow } X = \{Y\} \iff X = \{\} \land Y = \{\}\]
by blast

lemma Pow-insert: \[\text{Pow } (\text{insert } a \text{ A}) = \text{Pow } A \cup (\text{insert } a \setminus \text{Pow } A)\]
by (blast intro: image-eqI [where \(?x = u - \{a\} \text{ for } u\)]

lemma Pow-Compl: \[\text{Pow } (-A) = \{-B \mid B. A \in \text{Pow } B\}\]
by (blast intro: exI [where \(?x = -u \text{ for } u\)]

lemma Pow-UNIV [simp]: \[\text{Pow } \text{UNIV} = \text{UNIV}\]
by blast

lemma Un-Pow-subset: \[\text{Pow } A \cup \text{Pow } B \subseteq \text{Pow } (A \cup B)\]
by blast
lemma Pow-Int-eq [simp]: \( \text{Pow} (A \cap B) = \text{Pow} A \cap \text{Pow} B \)
  by blast

Miscellany.

lemma set-eq-subset: \( A = B \iff A \subseteq B \land B \subseteq A \)
  by blast

lemma subset-iff: \( A \subseteq B \iff (\forall t. t \in A \rightarrow t \in B) \)
  by blast

lemma subset-iff-psubset-eq: \( A \subseteq B \iff A \subset B \lor A = B \)
  unfolding less-le by blast

lemma all-not-in-conv [simp]: \( (\forall x. x \notin A) \iff A = \{\} \)
  by blast

lemma ex-in-conv: \( (\exists x. x \in A) \iff A \neq \{\} \)
  by blast

lemma ball-simps [simp, no-atp]:
  \( \bigwedge x P Q. (\forall x \in A. P x \lor Q) \iff (\forall x \in A. P x) \lor Q \)
  \( \bigwedge x P Q. (\forall x \in A. P \lor Q x) \iff (P \lor (\forall x \in A. Q x)) \)
  \( \bigwedge x P Q. (\forall x \in A. P \rightarrow Q x) \iff (P \rightarrow (\forall x \in A. Q x)) \)
  \( \bigwedge x P Q. (\forall x \in A. P x \rightarrow Q) \iff ((\exists x \in A. P x) \rightarrow Q) \)
  \( \bigwedge P. (\forall x \in \text{UNIV}. P x) \iff \text{True} \)
  \( \bigwedge a B P. (\forall x \in \text{insert} a B. P x) \iff (P a \land (\forall x \in B. P x)) \)
  \( \bigwedge P Q. (\forall x \in \text{Collect} Q. P x) \iff (\forall x. Q x \rightarrow P x) \)
  \( \bigwedge A P f. (\forall x \in f' A. P x) \iff (\forall x \in A. P (f x)) \)
  \( \bigwedge A P. (\neg (\forall x \in A. P x)) \iff (\exists x \in A. \neg P x) \)
  by auto

lemma bex-simps [simp, no-atp]:
  \( \bigwedge x P Q. (\exists x \in A. P x \land Q) \iff (\exists x \in A. P x) \land Q \)
  \( \bigwedge x P Q. (\exists x \in A. P \land Q x) \iff (P \land (\exists x \in A. Q x)) \)
  \( \bigwedge P. (\exists x \in \text{UNIV}. P x) \iff (\exists x. P x) \)
  \( \bigwedge a B P. (\exists x \in \text{insert} a B. P x) \iff (P a \lor (\exists x \in B. P x)) \)
  \( \bigwedge P Q. (\exists x \in \text{Collect} Q. P x) \iff (\exists x. Q x \land P x) \)
  \( \bigwedge A P f. (\exists x \in f' A. P x) \iff (\exists x \in A. P (f x)) \)
  \( \bigwedge A P. (\neg (\exists x \in A. P x)) \iff (\forall x \in A. \neg P x) \)
  by auto

lemma ex-image-cong-iff [simp, no-atp]:
  \( (\exists x. x \in f' A) \iff A \neq \{\} \)
  \( (\exists x. x \in f' A \land P x) \iff (\exists x \in A. P (f x)) \)
  by auto
6.4.4 Monotonicity of various operations

**lemma** image-mono: \( A \subseteq B \implies f \cdot A \subseteq f \cdot B \)
by blast

**lemma** Pow-mono: \( A \subseteq B \implies \operatorname{Pow} A \subseteq \operatorname{Pow} B \)
by blast

**lemma** insert-mono: \( C \subseteq D \implies \operatorname{insert} a C \subseteq \operatorname{insert} a D \)
by blast

**lemma** Un-mono: \( A \subseteq C \implies B \subseteq D \implies A \cup B \subseteq C \cup D \)
by (\textit{fact sup-mono})

**lemma** Int-mono: \( A \subseteq C \implies B \subseteq D \implies A \cap B \subseteq C \cap D \)
by (\textit{fact inf-mono})

**lemma** Diff-mono: \( A \subseteq C \implies D \subseteq B \implies A \setminus B \subseteq C \setminus D \)
by blast

**lemma** Compl-anti-mono: \( A \subseteq B \implies \neg B \subseteq \neg A \)
by (\textit{fact compl-mono})

Monotonicity of implications.

**lemma** in-mono: \( A \subseteq B \implies x \in A \implies x \in B \)
by (rule \textit{impl}) (erule subsetD)

**lemma** conj-mono: \( P_1 \implies Q_1 \implies P_2 \implies Q_2 \implies (P_1 \land P_2) \implies (Q_1 \land Q_2) \)
by \textit{iprover}

**lemma** disj-mono: \( P_1 \implies Q_1 \implies P_2 \implies Q_2 \implies (P_1 \lor P_2) \implies (Q_1 \lor Q_2) \)
by \textit{iprover}

**lemma** imp-mono: \( Q_1 \implies P_1 \implies P_2 \implies Q_2 \implies (P_1 \implies P_2) \implies (Q_1 \implies Q_2) \)
by \textit{iprover}

**lemma** imp-refl: \( P \implies P \ldots \)

**lemma** not-mono: \( Q \implies P \implies \neg P \implies \neg Q \)
by \textit{iprover}

**lemma** ex-mono: \( (\forall x. P x \implies Q x) \implies (\exists x. P x) \implies (\exists x. Q x) \)
by \textit{iprover}

**lemma** all-mono: \( (\forall x. P x \implies Q x) \implies (\forall x. P x) \implies (\forall x. Q x) \)
by \textit{iprover}

**lemma** Collect-mono: \( (\forall x. P x \implies Q x) \implies \operatorname{Collect} P \subseteq \operatorname{Collect} Q \)
THEORY “Set”

by blast

lemma Int-Collect-mono: A ⊆ B ⇒ (∀x. x ∈ A ⇒ P x → Q x) ⇒ A ∩ Collect P ⊆ B ∩ Collect Q
by blast

lemmas basic-monos =
subset-refl imp-refl disj-mono conj-mono ex-mono Collect-mono in-mono

lemma eq-to-mono: a = b ⇒ c = d ⇒ b → d ⇒ a → c
by iprover

6.4.5 Inverse image of a function

definition vimage :: (′a ⇒ ′b) ⇒ ′b set ⇒ ′a set (infixr − ′)
where f −′ B ≡ {x. f x ∈ B}

lemma vimage-eq [simp]: a ∈ f −′ B ⇐⇒ f a ∈ B
unfolding vimage-def by blast

lemma vimage-singleton-eq: a ∈ f −′ {b} ⇐⇒ f a = b
by simp

lemma vimageI [intro]: f a = b ⇒ b ∈ B ⇒ a ∈ f −′ B
unfolding vimage-def by blast

lemma vimageI2: f a ∈ A ⇒ a ∈ f −′ A
unfolding vimage-def by fast

lemma vimageE [elim!]: a ∈ f −′ B ⇒ (∀x. f a = x ⇒ x ∈ B ⇒ P) ⇒ P
unfolding vimage-def by blast

lemma vimageD: a ∈ f −′ A ⇒ f a ∈ A
unfolding vimage-def by fast

lemma vimage-empty [simp]: f −′ {} = {}
by blast

lemma vimage-Compl: f −′ (− A) = − (f −′ A)
by blast

lemma vimage-Un [simp]: f −′ (A ∪ B) = (f −′ A) ∪ (f −′ B)
by blast

lemma vimage-Int [simp]: f −′ (A ∩ B) = (f −′ A) ∩ (f −′ B)
by fast

lemma vimage-Collect-eq [simp]: f −′ Collect P = {y. P (f y)}
by blast
lemma vimage-Collect: \( \forall x. \ P\ (f\ x) = Q\ x \implies f\ -\ '\ (\text{Collect}\ P) = \text{Collect}\ Q \)  
by blast

lemma vimage-insert: \( f\ -\ '\ (\text{insert}\ a\ B) = (f\ -\ '\ \{a\}) \cup (f\ -\ '\ B) \)  
— NOT suitable for rewriting because of the recurrence of \( \{a\} \).  
by blast

lemma vimage-Diff: \( f\ -\ '\ (A\ -\ B) = (f\ -\ '\ A) - (f\ -\ '\ B) \)  
by blast

lemma vimage-UNIV [simp]: \( f\ -\ '\ \text{UNIV} = \text{UNIV} \)  
by blast

lemma vimage-mono: \( A \subseteq B \implies f\ -\ '\ A \subseteq f\ -\ '\ B \)  
— monotonicity  
by blast

lemma vimage-image-eq: \( f\ -\ '\ (f\ '\ A) = \{y. \exists x \in A. f\ x = f\ y\} \)  
by (blast intro: sym)

lemma image-vimage-subset: \( f\ '\ (f\ -\ '\ A) \subseteq A \)  
by blast

lemma image-vimage-eq [simp]: \( f\ '\ (f\ -\ '\ A) = A \cap \text{range}\ f \)  
by blast

lemma image-subset-iff-subset-vimage: \( f\ '\ A \subseteq B \iff A \subseteq f\ -\ '\ B \)  
by blast

lemma vimage-const [simp]: \( ((\lambda x.\ c)\ -\ '\ A) = (if\ c\ \in\ A\ then\ \text{UNIV}\ else\ \{\})\)  
by auto

lemma vimage-if [simp]: \( ((\lambda x. \text{if}\ x\ \in\ B\ then\ c\ else\ d)\ -\ '\ A) = \)  
(\text{if}\ c\ \in\ A\ \text{then}\ (if\ d\ \in\ A\ then\ \text{UNIV}\ else\ B)\)  
else if\ d\ \in\ A\ \text{then}\ \text{UNIV}\ else\ \{\})\)  
by (auto simp add: vimage-def)

lemma vimage-inter-cong: \( \bigwedge w.\ w \in S \implies f\ w = g\ w \implies f\ -\ '\ y \cap S = g\ -\ '\ y \cap S \)  
by auto

lemma vimage-ident [simp]: \( (\lambda x.\ x)\ -\ '\ Y = Y \)  
by blast

6.4.6 Singleton sets

definition \text{is-singleton} :: \\ 'a\ set\ \Rightarrow\ \text{bool} 
where \text{is-singleton}\ A \leftrightarrow (\exists x.\ A = \{x\})
lemma \textit{is-singletonI} [simp, intro!:]: \textit{is-singleton} \{x\}

unfolding \textit{is-singleton-def} by simp

lemma \textit{is-singletonI'}: \(A \neq \{\} \implies (\forall x \ y. \ x \in A \implies y \in A \implies x = y) \implies \textit{is-singleton} A\)

unfolding \textit{is-singleton-def} by blast

lemma \textit{is-singletonE}: \(\textit{is-singleton} A \implies (\forall x. \ A = \{x\} \implies P) \implies P\)

unfolding \textit{is-singleton-def} by blast

6.4.7 Getting the contents of a singleton set

definition \textit{the-elem} :: 'a set \Rightarrow 'a

where \textit{the-elem} \(X = (\text{THE} \ x. \ X = \{x\})\)

lemma \textit{the-elem-eq} [simp]: \textit{the-elem} \(\{x\} = x\)

by (simp add: \textit{the-elem-def})

lemma \textit{is-singleton-the-elem}: \(\textit{is-singleton} A \iff A = \{\textit{the-elem} A\}\)

by (auto simp: \textit{is-singleton-def})

lemma \textit{the-elem-image-unique}:

assumes \(A \neq \{\}\)

and \(*\): \(\forall y. \ y \in A \implies f y = f x\)

shows \(\textit{the-elem} (f \circ A) = f x\)

unfolding \textit{the-elem-def}

proof (rule the1-equality)

from \(\langle A \neq \{\}\rangle\) obtain \(y\) where \(y \in A\) by auto

with \(*\) have \(f x = f y\) by simp

with \(\langle y \in A\rangle\) have \(f x \in f \circ A\) by blast

with \(*\) show \(f \circ A = \{f x\}\) by auto

then show \(\exists! x. \ f \circ A = \{x\}\) by auto

qed

6.4.8 Least value operator

lemma \textit{Least-mono}: \(\text{mono} f \implies \exists x \in S. \ \forall y \in S. \ x \leq y \implies (\text{LEAST} y. \ y \in f \circ S) = f (\text{LEAST} x. \ x \in S)\)

for \(f :: 'a::order \Rightarrow 'b::order\)

— Courtesy of Stephan Merz

apply clarify

apply (erule-tac \(P = \lambda x. \ x \in S\) in \textit{LeastI2-order})

apply fast

apply (rule \textit{LeastI2-order})

apply (auto elim: \textit{monoD} intro!: order-antisym)

done
6.4.9 Monad operation

definition bind :: 'a set ⇒ ('a ⇒ 'b set) ⇒ 'b set
where bind A f = {x. ∃B ∈ f[A]. x ∈ B}

hide-const (open) bind

lemma bind-bind: Set.bind (Set.bind A B) C = Set.bind A (λx. Set.bind (B x) C)
for A :: 'a set
by (auto simp: bind-def)

lemma empty-bind [simp]: Set.bind {} f = {}
by (simp add: bind-def)

lemma nonempty-bind-const: A ≠ {} ⇒ Set.bind A (λ-. B) = B
by (auto simp: bind-def)

lemma bind-const: Set.bind A (λ-. B) = (if A = {} then {} else B)
by (auto simp: bind-def)

lemma bind-singleton-conv-image: Set.bind A (λx. {f x}) = f ' A
by (auto simp: bind-def)

6.4.10 Operations for execution

definition is-empty :: 'a set ⇒ bool
where [code-abbrev]: is-empty A ←→ A = {}

hide-const (open) is-empty

definition remove :: 'a ⇒ 'a set ⇒ 'a set
where [code-abbrev]: remove x A = A - {x}

hide-const (open) remove

lemma member-remove [simp]: x ∈ Set.remove y A ←→ x ∈ A ∧ x ≠ y
by (simp add: remove-def)

definition filter :: ('a ⇒ bool) ⇒ 'a set ⇒ 'a set
where [code-abbrev]: filter P A = {a ∈ A. P a}

hide-const (open) filter

lemma member-filter [simp]: x ∈ Set.filter P A ←→ x ∈ A ∧ P x
by (simp add: filter-def)

instantiation set :: (equal) equal
begin
definition HOL.equal A B ⟷ A ⊆ B ∧ B ⊆ A

instance by standard (auto simp add: equal-set-def)

end

Misc

definition pairwise :: ('a ⇒ bool) ⇒ 'a set ⇒ bool
where pairwise R S ⟷ (∀ x ∈ S. ∀ y ∈ S. x ≠ y ⟹ R x y)

lemma pairwise-alt: pairwise R S ⟷ (∀ x∈S. ∀ y∈S−{x}. R x y)
by (auto simp add: pairwise-def)

lemma pairwise-trivial [simp]: pairwise (λ x y. x ≠ y) I
by (auto simp: pairwise-def)

lemma pairwiseI [intro?]:
  pairwise R S if ⋀ x y. x ∈ S ⇒ y ∈ S ⇒ x ≠ y ⇒ R x y
using that by (simp all add: pairwise-def)

lemma pairwiseD:
  R x y and R y x
if pairwise R S x ∈ S and y ∈ S and x ≠ y
using that by (simp all add: pairwise-def)

lemma pairwise-empty [simp]: pairwise P {} 
by (simp add: pairwise-def)

lemma pairwise-singleton [simp]: pairwise P {A}
by (simp add: pairwise-def)

lemma pairwise-insert:
  pairwise r (insert x s) ⟷ (∀ y. y ∈ s ∧ y ≠ x ⟹ r x y ∧ r y x) ∧ pairwise r s
by (force simp: pairwise-def)

lemma pairwise-subset: pairwise P S ⟹ T ⊆ S ⟹ pairwise P T
by (force simp: pairwise-def)

lemma pairwise-mono: [pairwise P A; ⋀ x y. P x y ⟹ Q x y; B ⊆ A] ⟹ pairwise Q B
by (fastforce simp: pairwise-def)

lemma pairwise-imageI:
  pairwise P (f ' A)
if ⋀ x y. x ∈ A ⇒ y ∈ A ⇒ x ≠ y ⇒ f x ≠ f y ⇒ P (f x) (f y)
using that by (auto intro: pairwiseI)

lemma pairwise-image: pairwise r (f ' s) ⟷ pairwise (λ x y. (f x ≠ f y) ⟹ r (f x) (f y)) s
by (force simp: pairwise-def)

definition disjoint :: 'a set ⇒ 'a set ⇒ bool
where disjoint A B ←→ A ∩ B = {}

lemma disjoint-self-iff-empty [simp]: disjoint S S ←→ S = {}
  by (auto simp: disjoint-def)

lemma disjoint-if: disjoint A B ←→ (∀ x. ¬ (x ∈ A ∧ x ∈ B))
  by (force simp: disjoint-def)

lemma disjoint-sym: disjoint A B =⇒ disjoint B A
  using disjoint-if by blast

lemma disjoint-empty1 [simp]: disjoint {} A
and disjoint-empty2 [simp]: disjoint A {} by (auto simp: disjoint-def)

lemma disjoint-insert1 [simp]: disjoint (insert a X) Y ←→ a /∈ Y ∧ disjoint X Y
by (simp add: disjoint-def)

lemma disjoint-insert2 [simp]: disjoint Y (insert a X) ←→ a /∈ Y ∧ disjoint Y X
by (simp add: disjoint-def)

lemma disjoint-subset1: [disjoint X Y; Z ⊆ X] =⇒ disjoint Z Y
by (auto simp: disjoint-def)

lemma disjoint-subset2: [disjoint X Y; Z ⊆ Y] =⇒ disjoint X Z
by (auto simp: disjoint-def)

lemma disjoint-Un1 [simp]: disjoint (A ∪ B) C =⇒ disjoint A C ∧ disjoint B C
by (auto simp: disjoint-def)

lemma disjoint-Un2 [simp]: disjoint C (A ∪ B) =⇒ disjoint C A ∧ disjoint C B
by (auto simp: disjoint-def)

lemma disjoint-image-subset: [pairwise disjoint A; ∀ X. X ∈ A ⇒ f X ⊆ X] =⇒
  pairwise disjoint (f ' A)
  unfolding disjoint-def pairwise-def by fast

lemma Int-empty1: (∀ x. x ∈ A =⇒ x ∈ B =⇒ False) =⇒ A ∩ B = {}
by blast

lemma in-image-insert-iff:
  assumes ∀ C. C ∈ B =⇒ x /∈ C
  shows A ∈ insert x ' B =⇒ x ∈ A ∧ A − {x} ∈ B (is ?P =⇒ ?Q)
proof
  assume ?P then show ?Q
  using assms by auto
next
THEORY “Set”

assume \(?Q\)
then have \(x \in A\) and \(A - \{x\} \in B\)
  by simp-all
from \(\langle A - \{x\} \in B; \text{ have insert } x (A - \{x\}) \in \text{ insert } x \cdot B \rangle\)
  by (rule imageI)
also from \(x \in A\).
have \(\text{ insert } x (A - \{x\}) = A\)
  by auto
finally show \(?P\).
qed

hide-const (open) member not-member

lemmas equalityI = subset-antisym
lemmas set-mp = subsetD
lemmas set-rev-mp = rev-subsetD

ML :
val Ball-def = @{thm Ball-def}
val Bex-def = @{thm Bex-def}
val CollectD = @{thm CollectD}
val CollectE = @{thm CollectE}
val CollectI = @{thm CollectI}
val Collect-conj-eq = @{thm Collect-conj-eq}
val Collect-mem-eq = @{thm Collect-mem-eq}
val IntD1 = @{thm IntD1}
val IntD2 = @{thm IntD2}
val IntE = @{thm IntE}
val IntI = @{thm IntI}
val Int-Collect = @{thm Int-Collect}
val UNIV-I = @{thm UNIV-I}
val UNIV-witness = @{thm UNIV-witness}
val UnE = @{thm UnE}
val UnI1 = @{thm UnI1}
val UnI2 = @{thm UnI2}
val ballE = @{thm ballE}
val ballI = @{thm ballI}
val bexCI = @{thm bexCI}
val bexE = @{thm bexE}
val bexI = @{thm bexI}
val bex-triv = @{thm bex-triv}
val bspec = @{thm bspec}
val contra-subsetD = @{thm contra-subsetD}
val equalityCE = @{thm equalityCE}
val equalityD1 = @{thm equalityD1}
val equalityD2 = @{thm equalityD2}
val equalityE = @{thm equalityE}
val equalityI = @{thm equalityI}
val imageE = @{thm imageE}
7 HOL type definitions

theory Typedef
imports Set
keywords
typedef :: thy-goal-defn and
morphisms :: quasi-command
begin

locale type-definition =
  fixes Rep and Abs and A
  assumes Rep: Rep x ∈ A
  and Rep-inverse: Abs (Rep x) = x
  and Abs-inverse: y ∈ A ⇒ Rep (Abs y) = y
  — This will be axiomatized for each typedef!
begin

lemma Rep-inject: Rep x = Rep y ↔ x = y
proof
  assume Rep x = Rep y
  then have Abs (Rep x) = Abs (Rep y) by (simp only:)
  moreover have Abs (Rep x) = x by (rule Rep-inverse)
  moreover have Abs (Rep y) = y by (rule Rep-inverse)
  ultimately show x = y by simp

end
next
  assume \( x = y \)
  then show \( \text{Rep} \ x = \text{Rep} \ y \) by (simp only:)
qed

lemma Abs-inject:
  assumes \( x \in A \) and \( y \in A \)
  shows \( \text{Abs} \ x = \text{Abs} \ y \rightleftharpoons x = y \)
proof
  assume \( \text{Abs} \ x = \text{Abs} \ y \)
  then have \( \text{Rep} \ (\text{Abs} \ x) = \text{Rep} \ (\text{Abs} \ y) \) by (simp only:)
  moreover from \( \langle x \in A \rangle \) have \( \text{Rep} \ (\text{Abs} \ x) = x \) by (rule Abs-inverse)
  moreover from \( \langle y \in A \rangle \) have \( \text{Rep} \ (\text{Abs} \ y) = y \) by (rule Abs-inverse)
  ultimately show \( x = y \) by simp
next
  assume \( x = y \)
  then show \( \text{Abs} \ x = \text{Abs} \ y \) by (simp only:)
qed

lemma Rep-cases [cases set]:
  assumes \( y \in A \)
  and \( \text{hyp}: \forall x. \ y = \text{Rep} \ x \Rightarrow P \)
  shows \( P \)
proof (rule hyp)
  from \( y \in A \) have \( \text{Rep} \ (\text{Abs} \ y) = y \) by (rule Abs-inverse)
  then show \( y = \text{Rep} \ (\text{Abs} \ y) \) ..
qed

lemma Abs-cases [cases type]:
  assumes \( r: \forall y. \ x = \text{Abs} \ y \Rightarrow \ y \in A \Rightarrow P \)
  shows \( P \)
proof (rule r)
  have \( \text{Abs} \ (\text{Rep} \ x) = x \) by (rule Rep-inverse)
  then show \( x = \text{Abs} \ (\text{Rep} \ x) \) ..
  show \( \text{Rep} \ x \in A \) by (rule Rep)
qed

lemma Rep-induct [induct set]:
  assumes \( y: y \in A \)
  and \( \text{hyp}: \forall x. \ P \ (\text{Rep} \ x) \)
  shows \( \text{P} \ y \)
proof
  have \( P \ (\text{Rep} \ (\text{Abs} \ y)) \) by (rule hyp)
  moreover from \( y \) have \( \text{Rep} \ (\text{Abs} \ y) = y \) by (rule Abs-inverse)
  ultimately show \( P \ y \) by simp
qed

lemma Abs-induct [induct type]:
  assumes \( r: \forall y. \ y \in A \Rightarrow P \ (\text{Abs} \ y) \)
shows $P \ x$
proof
  have $\Rep \ x \in A$ by (rule $\Rep$)
  then have $P \ (\Abs \ (\Rep \ x))$ by (rule $r$)
  moreover have $\Abs \ (\Rep \ x) = x$ by (rule $\Rep$-inverse)
  ultimately show $P \ x$ by simp
qed

lemma $\Rep$-range: $\range \ \Rep = A$
proof
  show $\range \ \Rep \subseteq A$ using $\Rep$ by (auto simp add: image-def)
  show $A \subseteq \range \ \Rep$
proof
    fix $x$ assume $x \in A$
    then have $x = \Rep \ (\Abs \ x)$ by (rule $\Abs$-inverse [symmetric])
    then show $x \in \range \ \Rep$ by (rule range-eqI)
  qed
qed

lemma $\Abs$-image: $\Abs \ A = \UNIV$
proof
  show $\Abs \ A \subseteq \UNIV$ by (rule subset-UNIV)
  show $\UNIV \subseteq \Abs \ A$
proof
    fix $x$
    have $x = \Abs \ (\Rep \ x)$ by (rule $\Rep$-inverse [symmetric])
    moreover have $\Rep \ x \in A$ by (rule $\Rep$)
    ultimately show $x \in \Abs \ A$ by (rule image-eqI)
  qed
qed

ML-file ⟨Tools/typedef.ML⟩
end

end

8 Notions about functions

theory Fun
  imports Set
  keywords functor :: thy-goal-defn
begin

lemma apply-inverse: $f \ x = u \Longrightarrow (\forall x. \ P \ x \Longrightarrow g \ (f \ x) = x) \Longrightarrow P \ x \Longrightarrow x = g$
  $u$
  by auto

Uniqueness, so NOT the axiom of choice.
lemma uniq-choice: \( \forall x. \exists! y. Q x y \implies \exists f. \forall x. Q x (f x) \)
by (force intro: theI)

lemma b-uniq-choice: \( \forall x \in S. \exists! y. Q x y \implies \exists f. \forall x \in S. Q x (f x) \)
by (force intro: theI)

8.1 The Identity Function \( id \)
definition id :: 'a ⇒ 'a
where \( id = (\lambda x. x) \)

lemma id-apply [simp]: \( id x = x \)
by (simp add: id-def)

lemma image-id [simp]: \( \text{image id} = id \)
by (simp add: id-def fun-eq-iff)

lemma vimage-id [simp]: \( \text{vimage id} = id \)
by (simp add: id-def fun-eq-iff)

lemma eq-id-iff: \( (\forall x. f x = x) \iff f = id \)
by auto

code-printing
constant id ↦ (Haskell) id

8.2 The Composition Operator \( f \circ g \)
definition comp :: ('b ⇒ 'c) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'c (infixl ◦ 55)
where \( f \circ g = (\lambda x. f (g x)) \)

notation (ASCII)
\( \circ \quad \text{infixl 55} \)

lemma comp-apply [simp]: \( (f \circ g) x = f (g x) \)
by (simp add: comp-def)

lemma comp-assoc: \( f \circ (g \circ h) = (f \circ g) \circ h \)
by (simp add: fun-eq-iff)

lemma id-comp [simp]: \( id \circ g = g \)
by (simp add: fun-eq-iff)

lemma comp-id [simp]: \( f \circ id = f \)
by (simp add: fun-eq-iff)

lemma comp-eq-dest: \( a \circ b = c \circ d \implies a \ (b \ v) = c \ (d \ v) \)
by (simp add: fun-eq-iff)

lemma comp-eq-elim: \( a \circ b = c \circ d \implies ((\forall v. a \ (b \ v) = c \ (d \ v)) \implies R) \implies R \)
by (simp add: fun-eq-iff)

lemma comp-eq-dest-lhs: \( a \circ b = c \implies a (b v) = c v \)
  by clarsimp

lemma comp-eq-id-dest: \( a \circ b = id \circ c \implies a (b v) = c v \)
  by clarsimp

lemma image-comp: \( \lambda (g' r) = (f \circ g)' r \)
  by auto

lemma vimage-comp: \( f' (g' x) = (g \circ f)' x \)
  by auto

lemma image-eq-imp-comp: \( f' A = g' B \implies (h \circ f)' A = (h \circ g)' B \)
  by (auto simp: comp-def elim!: equalityE)

lemma image-bind: \( f' (\text{Set.bind} A g) = \text{Set.bind} A (('f \circ g)') \)
  by (auto simp add: Set.bind-def)

lemma bind-image: \( \text{Set/bind} (f' A) g = \text{Set/bind} A (g \circ f) \)
  by (auto simp add: Set.bind-def)

lemma (in group-add) minus-comp-minus [simp]: \( \text{uminus} \circ \text{uminus} = id \)
  by (simp add: fun-eq-iff)

lemma (in boolean-algebra) minus-comp-minus [simp]: \( \text{uminus} \circ \text{uminus} = id \)
  by (simp add: fun-eq-iff)

code-printing
constant comp :: (SML) infixl 5 o and (Haskell) infixr 9.

8.3 The Forward Composition Operator \( f\circ\cdot \)

definition fcomp :: \( ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'c \) (infixl \( \circ > \) 60)
  where \( f \circ g = (\lambda x. g (f x)) \)

lemma fcomp-apply [simp]: \( f \circ g) x = g (f x) \)
  by (simp add: fcomp-def)

lemma fcomp-assoc: \( f \circ g) \circ h = f \circ (g \circ h) \)
  by (simp add: fcomp-def)

lemma id-fcomp [simp]: \( id \circ g = g \)
  by (simp add: fcomp-def)

lemma fcomp-id [simp]: \( f \circ id = f \)
  by (simp add: fcomp-def)
lemma fcomp-comp: fcomp f g = comp g f
  by (simp add: ext)

code-printing
  constant fcomp ⇀ (Eval) infixl 1 #>

no-notation fcomp (infixl ◦ > 60)

8.4 Mapping functions

definition map-fun :: ('c ⇒ 'a) ⇒ ('b ⇒ 'd) ⇒ ('a ⇒ 'b ⇒ 'd)
  where map-fun f g h = g ◦ h ◦ f

lemma map-fun-apply [simp]: map-fun f g h x = g (h (f x))
  by (simp add: map-fun-def)

8.5 Injectivity and Bijectivity

definition inj-on :: ('a ⇒ 'b) ⇒ 'a set ⇒ bool — injective
  where inj-on f A = (∀ x y ∈ A. f x = f y → x = y)

definition bij-betw :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'b set ⇒ bool — bijective
  where bij-betw f A B = inj-on f A ∧ f ' A = B

A common special case: functions injective, surjective or bijective over the
entire domain type.

abbreviation inj :: ('a ⇒ 'b) ⇒ bool
  where inj f = inj-on f UNIV

abbreviation surj :: ('a ⇒ 'b) ⇒ bool
  where surj f = range f = UNIV

translations — The negated case:
  ¬ CONST surj f ⇄ CONST range f ≠ CONST UNIV

abbreviation bij :: ('a ⇒ 'b) ⇒ bool
  where bij f = bij-betw f UNIV UNIV

lemma inj-def: inj f = (∀ x y. f x = f y → x = y)
  unfolding inj-on-def by blast

lemma injI: (∀ x y. f x = f y → x = y) → inj f
  unfolding inj-def by blast

theorem range-ex1-eq: inj f =⇒ b ∈ range f =⇒ (∃! x. b = f x)
  unfolding inj-def by blast

lemma injD: inj f =⇒ f x = f y =⇒ x = y
  by (simp add: inj-def)
lemma inj-on-eq-iff: inj-on f A \implies x \in A \implies y \in A \implies f x = f y \iff x = y
by (auto simp: inj-on-def)

lemma inj-on-cong: (\forall a. a \in A \implies f a = g a) \implies inj-on f A \iff inj-on g A
by (auto simp: inj-on-def)

lemma inj-on-strict-subset: inj-on f B \implies A \subset B \implies f ' A \subset f ' B
unfolding inj-on-def by blast

lemma inj-compose: inj f \implies inj g \implies inj (f \circ g)
by (simp add: inj-def)

lemma inj-fun: inj f \implies inj (\lambda x y. f x)
by (simp add: inj-def fun-eq-iff)

lemma inj-eq: inj f \implies f x = f y \iff x = y
by (simp add: inj-on-eq-iff)

lemma inj-on-id[simp]: inj-on id A
by (simp add: inj-on-def)

lemma inj-on-id2[simp]: inj-on (\lambda x. x) A
by (simp add: inj-on-def)

lemma inj-on-Int: inj-on f A \lor inj-on f B \implies inj-on f (A \cap B)
unfolding inj-on-def by blast

lemma surj-id: surj id
by simp

lemma bij-id[simp]: bij id
by (simp add: bij-betw-def)

lemma bij-uminus: bij (uminus :: 'a::ab-group-add)
unfolding bij-betw-def inj-on-def
by (force intro: minus-minus [symmetric])

lemma inj-on1 [intro?]: (\forall x y. x \in A \implies y \in A \implies f x = f y \implies x = y) \implies inj-on f A
by (simp add: inj-on-def)

lemma inj-on-inverseI: (\forall x. x \in A \implies g (f x) = x) \implies inj-on f A
by (auto dest: arg-cong [of concl: g] simp add: inj-on-def)

lemma inj-onD: inj-on f A \implies f x = f y \implies x \in A \implies y \in A \implies x = y
unfolding inj-on-def by blast

lemma inj-on-subset:
THEORY "Fun"

assumes inj-on f A
and B ⊆ A
shows inj-on f B
proof (rule inj-onI)
  fix a b
  assume a ∈ B and b ∈ B
  with assms have a ∈ A and b ∈ A
    by auto
  moreover assume f a = f b
  ultimately show a = b
    using assms by (auto dest: inj-onD)
qed

lemma comp-inj-on: inj-on f A ⇒ inj-on g (f ′ A) ⇒ inj-on (g ∘ f) A
  by (simp add: comp-def inj-on-def)

lemma inj-on-imageI: inj-on (g ∘ f) A ⇒ inj-on g (f ′ A)
  by (auto simp add: inj-on-def)

lemma inj-on-image-iff:
  ∀x ∈ A. ∀y ∈ A. g (f x) = g (f y) ⇔ g x = g y ⇒ inj-on f A ⇒ inj-on g (f ′ A)
  ⇔ inj-on g A
unfolding inj-on-def by blast

lemma inj-on-contraD: inj-on f A ⇒ x ≠ y ⇒ x ∈ A ⇒ y ∈ A ⇒ f x ≠ f y
unfolding inj-on-def by blast

lemma inj-singleton [simp]: inj-on (λx. {x}) A
  by (simp add: inj-on-def)

lemma inj-on-empty [iff]: inj-on f {}
  by (simp add: inj-on-def)

lemma subset-inj-on: inj-on f B ⇒ A ⊆ B ⇒ inj-on f A
  unfolding inj-on-def by blast

lemma inj-on-Un: inj-on f (A ∪ B) ⇔ inj-on f A ∧ inj-on f B ∧ f ′ (A − B) ∩ f ′ (B − A) = {}
  unfolding inj-on-def by (blast intro: sym)

lemma inj-on-insert [iff]: inj-on f (insert a A) ⇔ inj-on f A ∧ f a ∉ f ′ (A − {a})
  unfolding inj-on-def by (blast intro: sym)

lemma inj-on-diff: inj-on f A ⇒ inj-on f (A − B)
  unfolding inj-on-def by blast

lemma comp-inj-on-iff: inj-on f A ⇒ inj-on f ′ (f ′ A) ⇔ inj-on (f ′ ∘ f) A
  by (auto simp: comp-inj-on inj-on-def)
lemma inj-on-imageI2: inj-on \((f' \circ f)\) \(A\) \(\Longrightarrow\) inj-on \(f\) \(A\)
by \((\text{auto simp: comp-inj-on inj-on-def})\)

lemma inj-img-insertE:
assumes \(\text{inj-on} f A\)
assumes \(x \notin B\) and \(\text{insert } x B = f' A\)
obtains \(x' A'\) where \(x' \notin A'\) and \(\text{insert } x' A = \text{insert } x A'\) and \(x = f x'\) and \(B = f' A'\)
proof
from assms have \(x \in f' A\) by auto
then obtain \(x'\) where \(*\) \(x' \in A\) \(x = f x'\) by auto
then have \(A\) : \(A = \text{insert } x' A - \{x'\}\) by auto
with assms \(*\) have \(B\) : \(B = f' (A - \{x'\})\) by \((\text{auto dest: inj-on-contraD})\)
have \(x' \notin A - \{x'\}\) by simp
from this \(A \langle x = f x' \rangle B\) show ?thesis ..
qed

lemma linorder-inj-onI:
fixes \(A::'a::\text{order set}\)
assumes ne: \(\forall x y. [x < y; x \in A; y \in A] \Rightarrow f x \neq f y\) and lin: \(\forall x y. [\forall x \in A; y \in A] \Rightarrow x \leq y \lor y \leq x\)
shows inj-on \(f A\)
proof \((\text{rule inj-onI})\)
fix \(x y\)
assume eq: \(f x = f y\) and \(x \in A y \in A\)
then show \(x = y\)
using lin \([\text{of } x y]\) ne by \((\text{force simp: dual-order.order-iff-strict})\)
qed

lemma linorder-injI:
assumes \(\forall x y::'a::\text{linorder}. x < y \Rightarrow f x \neq f y\)
shows inj \(f\)
— Courtesy of Stephan Merz
using assms by \((\text{auto intro: linorder-inj-onI linear})\)

lemma inj-on-image-Pow: inj-on \(f A\) \(\Longrightarrow\) inj-on \((\text{image } f)\) \((\text{Pow } A)\)
unfolding Pow-def inj-on-def by blast

lemma bij-betw-image-Pow: bij-betw \(f A B\) \(\Longrightarrow\) bij-betw \((\text{image } f)\) \((\text{Pow } A)\) \((\text{Pow } B)\)
by \((\text{auto simp add: bij-betw-def inj-on-image-Pow image-Pow-surj})\)

lemma surj-def: \(\text{surj } f \longleftrightarrow (\forall y. \exists x. y = f x)\)
by auto

lemma surjI:
assumes \(\forall x. g (f x) = x\)
shows \( \text{surj} \) \( g \)
using assms \([\text{symmetric}]\) by auto

lemma \( \text{surjD} \): \( \text{surj} \ f \implies \exists x. \ y = f \ x \)
by (simp add: \( \text{surj-def} \))

lemma \( \text{surjE} \): \( \text{surj} \ f \implies (\forall x. \ y = f \ x \implies C) \implies C \)
by (simp add: \( \text{surj-def} \)) blast

lemma \( \text{comp-surj} \): \( \text{surj} \ f \implies \text{surj} \ g \implies \text{surj} \ (g \circ f) \)
using image-comp \([\text{of g f UNIV}]\) by simp

lemma \( \text{bij-betw-imageI} \): \( \text{inj-on} \ f \ A \implies f ' A = B \implies \text{bij-betw} \ f \ A \ B \)
unfolding \( \text{bij-betw-def} \) by clarify

lemma \( \text{bij-betw-imp-surj-on} \): \( \text{bij-betw} \ f \ A \ B \implies f ' A = B \)
unfolding \( \text{bij-betw-def} \) by clarify

lemma \( \text{bij-betw-empty1} \): \( \text{bij-betw} \ f \ \{\} \ A \implies A = \{\} \)
unfolding \( \text{bij-betw-def} \) by blast

lemma \( \text{bij-betw-empty2} \): \( \text{bij-betw} \ f \ \{\} \ A \implies A = \{\} \)
unfolding \( \text{bij-betw-def} \) by blast

lemma \( \text{inj-on-imp-bij-betw} \): \( \text{inj-on} \ f \ A \implies \text{bij-betw} \ f \ A \ (f ' A) \)
unfolding \( \text{bij-betw-def} \) by simp

lemma \( \text{bij-betw-apply} \): \([\text{bij-betw} \ f \ A \ B ; \ a \in A] \implies f a \in B \)
unfolding \( \text{bij-betw-def} \) by auto

lemma \( \text{bij-def} \): \( \text{bij} \ f \iff \text{inj} \ f \land \text{surj} \ f \)
by (rule \( \text{bij-betw-def} \))

lemma \( \text{bijI} \): \( \text{inj} \ f \implies \text{surj} \ f \implies \text{bij} \ f \)
by (rule \( \text{bij-betw-imageI} \))

lemma \( \text{bij-is-inj} \): \( \text{bij} \ f \implies \text{inj} \ f \)
by (simp add: \( \text{bij-def} \))

lemma \( \text{bij-is-surj} \): \( \text{bij} \ f \implies \text{surj} \ f \)
by (simp add: \( \text{bij-def} \))

lemma \( \text{bij-betw-imp-inj-on} \): \( \text{bij-betw} \ f \ A \ B \implies \text{inj-on} \ f \ A \)
by (simp add: \( \text{bij-betw-def} \))

lemma \( \text{bij-betw-trans} \): \( \text{bij-betw} \ f \ A \ B \implies \text{bij-betw} \ g \ B \ C \implies \text{bij-betw} \ (g \circ f) \ A \ C \)
by (auto simp add: bij-betw-def comp-inj-on)

lemma bij-comp: bij f ⟷ bij g ⟷ bij (g ∘ f)
  by (rule bij-betw-trans)

lemma bij-betw-comp-iff: bij-betw f A A' ⟷ bij-betw f' A' A'' ⟷ bij-betw (f' ∘ f) A A''
  by (auto simp add: bij-betw-def)

lemma bij-betw-comp-iff2:
  assumes bij': bij-betw f' A' A''
  and img: f' A ≤ A'
  shows bij-betw f A A' ⟷ bij-betw (f' ∘ f) A A''
  using assms
proof (auto simp add: bij-betw-comp-iff)
  assume *: bij-betw (f' ∘ f) A A''
  then show bij-betw f A A'
    using img
proof (auto simp add: bij-betw-def)
  assume inj-on (f' ∘ f) A
  then show inj-on f A
    using inj-on-imageI2 by blast
next
  fix a'
  assume **: a' ∈ A'
  with bij have f' a' ∈ A''
    unfolding bij-betw-def by auto
  with * obtain a where 1: a ∈ A ∧ f (f a) = f' a'
    unfolding bij-betw-def by force
  with img have f a ∈ A'' by auto
  with bij ** 1 have f a = a'
    unfolding bij-betw-def inj-on-def by auto
  with 1 show a' ∈ f' A by auto
  qed
  qed

lemma bij-betw-inv:
  assumes bij-betw f A B
  shows ∃ g. bij-betw g B A
proof –
  have i: inj-on f A and s: f' A = B
    using assms by (auto simp: bij-betw-def)
  let ?P = λa. a ∈ A ∧ f a = b
  let ?g = λb. The (?P b)
  have g: ∀b. ?P b a if P: ?P b a for a b
  proof –
    from that s have cx1: ∃ a. ?P b a by blast
    then have wcx1: ∃!a. ?P b a by (blast dest:inj-onD[OF i])
    then show ?thesis
using the1-equality[OF uex1, OF P] P by simp
qed
have inj-on \?g B
proof (rule inj-onI)
  fix x y
  assume x ∈ B y ∈ B \?g x = \?g y
  from s \langle x ∈ B \rangle obtain a1 where a1: \?P x a1 by blast
  from s \langle y ∈ B \rangle obtain a2 where a2: \?P y a2 by blast
  from g [OF a1] a1 g [OF a2] a2 \langle \?g x = \?g y \rangle show x = y by simp
qed
moreover have \?g ' B = A
proof (auto simp: image-def)
  fix b
  assume b ∈ B
  with s obtain a where P: \?P b a by blast
  with g [OF P] show \?g b ∈ A by auto
next
  fix a
  assume a ∈ A
  with s obtain b where P: \?P b a by blast
  with s have b ∈ B by blast
  with g [OF P] show \exists b ∈ B. a = \?g b by blast
qed
ultimately show \?thesis
  by (auto simp: bij-betw-def)
qed

lemma bij-betw-cong: (\forall a. a ∈ A \implies f a = g a) \implies \text{bij-betw } f A A' = \text{bij-betw } g A A'
  unfolding bij-betw-def inj-on-def by safe force+

lemma bij-betw-id[intro, simp]: \text{bij-betw id } A A
  unfolding bij-betw-def id-def by auto

lemma bij-betw-id-iff: \text{bij-betw id } A B \iff A = B
  by (auto simp add: bij-betw-def)

lemma bij-betw-compose:
  \text{bij-betw } f A B \implies \text{bij-betw } f C D \implies B \cap D = \{} \implies \text{bij-betw } f (A \cup C) (B \cup D)
  unfolding bij-betw-def inj-on-Un image-Un by auto

lemma bij-betw-subset: \text{bij-betw } f A A' \implies B \subseteq A \implies f ' B = B' \implies \text{bij-betw } f B B'
  by (auto simp add: bij-betw-def inj-on-def)

lemma bij-pointE:
  assumes bij f
  obtains x where y = f x and \\langle x'. y = f x' \implies x' = x \rangle
proof –
from assms have inj f by (rule bij-is-inj)
moreover from assms have surj f by (rule bij-is-surj)
then have y ∈ range f by simp
ultimately have ∃!x. y = f x by (simp add: range-ex1-eq)
with that show thesis by blast
qed

lemma surj-image-vimage-eq: surj f ⟹ f −' (f −' A) = A
by simp

lemma surj-vimage-empty:
assumes surj f
shows f −' A = {} ⟷ A = {}
using surj-image-vimage-eq [OF ⟨surj f⟩, of A]
by (intro iffI) fastforce+

lemma inj-vimage-image-eq:
inj f ⟹ f −' (f ' A) = A
unfolding inj-def by blast

lemma vimage-subsetD:
surj f ⟹ f −' B ⊆ A ⟹ B ⊆ f ' A
by (blast intro: sym)

lemma vimage-subsetI:
inj f ⟹ B ⊆ f ' A ⟹ f −' B ⊆ A
unfolding inj-def by blast

lemma vimage-subset-eq:
bij f ⟹ f −' B ⊆ A ⟷ B ⊆ f ' A
by (intro iffI)

lemma inj-on-image-eq-iff:
inj-on f C ⟹ A ⊆ C ⟹ B ⊆ C ⟹ f ' A = f ' B
⟷ A = B
by (fastforce simp: inj-on-def)

lemma inj-on-Un-image-eq-iff:
inj-on f (A ∪ B) ⟹ f ' A = f ' B ⟷ A = B
by (erule inj-on-image-eq-iff) simp-all

lemma inj-on-image-Int:
inj-on f C ⟹ A ⊆ C ⟹ B ⊆ C ⟹ f ' (A ∩ B) = f ' A ∩ f ' B
unfolding inj-on-def by blast

lemma inj-on-image-set-diff:
inj-on f C ⟹ A − B ⊆ C ⟹ B ⊆ C ⟹ f ' (A − B) = f ' A − f ' B
unfolding inj-on-def by blast

lemma image-Int: inj f ⟹ f ' (A ∩ B) = f ' A ∩ f ' B
unfolding inj-def by blast

lemma image-set-diff: inj f ⟹ f ' (A − B) = f ' A − f ' B
unfolding inj-def by blast
lemma inj-on-image-mem-iff: inj-on \( f \) \( B \) \( \implies \) \( a \in B \) \( \implies \) \( A \subseteq B \) \( \implies \) \( f a \in f' A \) \( \iff \) \( a \in A \)
   by (auto simp: inj-on-def)

lemma inj-on-image-mem-iff-alt: inj-on \( f \) \( B \) \( \implies \) \( A \subseteq B \) \( \implies \) \( f a \in f' A \) \( \iff \) \( a \in A \)
   by (blast dest: inj-onD)

lemma inj-image-subset-iff: inj \( f \) \( \implies \) \( f' A \subseteq f' B \) \( \iff \) \( A \subseteq B \)
   by (blast dest: injD)

lemma inj-image-eq-iff: inj \( f \) \( \implies \) \( f' A = f' B \) \( \iff \) \( A = B \)
   by (blast dest: injD)

lemma bij-image-Compl-eq: bij \( f \) \( \implies \) \( f' (- A) = - (f' A) \)
   by (simp add: bij-def inj-image-Compl-subset surj-Compl-image-subset equalityI)

lemma inj-vimage-singleton: inj \( f \) \( \implies \) \( f \; {-}^{-1} \{ a \} \subseteq \{ \text{THE } x. f x = a \} \)
   — The inverse image of a singleton under an injective function is included in a singleton.
   by (simp add: inj-def) (blast intro: the-equality [symmetric])

lemma inj-on-vimage-singleton: inj-on \( f \) \( A \) \( \implies \) \( f \; {-}^{-1} \{ a \} \cap A \subseteq \{ \text{THE } x. x \in A \wedge f x = a \} \)
   by (auto simp add: inj-on-def intro: the-equality [symmetric])

lemma (in ordered-ab-group-add) inj-uminus[simp, intro]: inj-on \( \ominus \) \( A \)
   by (auto intro!: inj-onI)

lemma (in linorder) strict-mono-imp-inj-on: strict-mono \( f \) \( \implies \) inj-on \( f \) \( A \)
   by (auto intro!: inj-onI dest: strict-mono-eq)

lemma bij-betw-byWitness:
   assumes left: \( \forall a \in A. f' (f a) = a \)
   and right: \( \forall a' \in A'. f (f' a') = a' \)
   and inj1: \( f' A \subseteq A' \)
   and inj2: \( f' A' \subseteq A \)
   shows bij-betw \( f \) \( A \) \( A' \)
using assms
unfolding bij-betw-def inj-on-def

proof safe
  fix a b
  assume a ∈ A b ∈ A
  with left have a = f′ (f a) ∧ b = f′ (f b) by simp
  moreover assume f a = f b
  ultimately show a = b by simp
next
  fix a' assume *: a' ∈ A'
  with img2 have f' a' ∈ A by blast
  moreover from * right have a' = f (f' a') by simp
  ultimately show a' ∈ f' A by blast
qed

corollary notIn-Un-bij-betw:
  assumes b /∈ A
  and f b /∈ A'
  and bij-betw f A A'
  shows bij-betw f (A ∪ {b}) (A' ∪ {f b})
proof
  have bij-betw f {b} {f b}
  unfolding bij-betw-def inj-on-def by simp
  with assms show ?thesis
  using bij-betw-combine[of f A A' {b} {f b}] by blast
qed

lemma notIn-Un-bij-betw3:
  assumes b /∈ A
  and f b /∈ A'
  shows bij-betw f A A' = bij-betw f (A ∪ {b}) (A' ∪ {f b})
proof
  assume bij-betw f A A'
  then show bij-betw f (A ∪ {b}) (A' ∪ {f b})
  using assms notIn-Un-bij-betw [of b A f A'] by blast
next
  assume *: bij-betw f (A ∪ {b}) (A' ∪ {f b})
  have f · A = A'
  proof auto
    fix a
    assume **: a ∈ A
    then have f a ∈ A' ∪ {f b}
      using * unfolding bij-betw-def by blast
    moreover
    have False if f a = f b
    proof
      have a = b
      using ** that unfolding bij-betw-def inj-on-def by blast
      with (b /∈ A) ** show ?thesis by blast
qed
ultimately show \( f \, a \in A' \) by blast
next
fix \( a' \)
assume **: \( a' \in A' \)
then have \( a' \in f' (A \cup \{b\}) \)
using * by (auto simp add: bij-betw-def)
then obtain \( a \) where I: \( a \in A \cup \{b\} \land f \, a = a' \) by blast
moreover
have False if \( a = b \) using I ** if \( b \not\in A' \) that by blast
ultimately have \( a \in A \) by blast
with I show \( a' \in f' \, A \) by blast
qed
then show bij-betw f A A'
using * bij-betw-subset[of f A \cup \{b\} - A] by blast
qed

Important examples

context cancel-semigroup-add begin

lemma inj-on-add simp:

inj-on \((+) \, a) A
by (rule inj-onI) simp

lemma inj-add-left:
inj ((+) a)
by simp

lemma inj-on-add' simp:
inj-on \((\lambda b. b + a) \) A
by (rule inj-onI) simp

lemma bij-betw-add simp:
bij-betw \((+) \, a) A B \leftrightarrow (+) \, a' \, A = B
by (simp add: bij-betw-def)

end

context ab-group-add begin

lemma surj-plus simp:
surj \((+) a)
by (auto intro!: range-eqI [of b \((+) \, a \, b - a \) for b]) (simp add: algebra-simps)

lemma inj-diff-right simp:
inj \((\lambda b. b - a)\)
proof —
have \((\text{inj } ((+) (- a)))\)
by (fact inj-add-left)
also have \((+)(-a) = (\lambda b. b - a)\)
by (simp add: fun-eq-iff)
finally show \(?\text{thesis} \).
qed

lemma surj-diff-right [simp]:
surj \((\lambda x. x - a)\)
using surj-plus \([af - a]\) by (simp cong: image-cong-simp)

lemma translation-Compl:
\((+)(-t) = -((+)(a t))\)
proof (rule set-eqI)
fix \(b\)
show \(b \in (+)(-t) \iff b \in -(+)(a t)\)
by (auto simp: image-iff algebra-simps intro: bexI ![of \(-b-a\)])
qed

lemma translation-subtract-Compl:
\((\lambda x. x - a)'(-t) = -((\lambda x. x - a)'t)\)
using translation-Compl \([af - a t]\) by (simp cong: image-cong-simp)

lemma translation-diff:
\((+)(s - t) = ((+)(a t)) - ((+)(a t))\)
by auto

lemma translation-subtract-diff:
\((\lambda x. x - a)'(s - t) = ((\lambda x. x - a)'s) - ((\lambda x. x - a)'t)\)
using translation-diff \([af - a]\) by (simp cong: image-cong-simp)

lemma translation-Int:
\((+)(s \cap t) = ((+)(a t)) \cap ((+)(a t))\)
by auto

lemma translation-subtract-Int:
\((\lambda x. x - a)'(s \cap t) = ((\lambda x. x - a)'s) \cap ((\lambda x. x - a)'t)\)
using translation-Int \([af - a]\) by (simp cong: image-cong-simp)
end

8.6 Function Updating

definition fun-upd :: \(\text{('}a \Rightarrow \text{'}b) \Rightarrow \text{'}a \Rightarrow \text{'}b \Rightarrow \text{'}a \Rightarrow \text{'}b\)
where fun-upd \(f\ a\ b = (\lambda x. \text{if } x = a \text{ then } b \text{ else } f\ x)\)

nonterminal updbinds and updbind

syntax
-updbind :: 'a ⇒ 'a ⇒ updbind
  ((2- :=/ -))
:: updbind ⇒ updbinds
- updbinds: updbind ⇒ updbinds ⇒ updbinds (.,/ -)
-Update :: 'a ⇒ updbinds ⇒ 'a
  (.-/('') [1000, 0] 900)

translations
-Update f (-updbinds b bs) ⇔ -Update (-Update f b) bs
f(x:=y) ⇔ CONST fun-upd f x y

lemma fun-upd-idem-iff: f(x:=y) = f ⇔ f x = y
unfolding fun-upd-def
apply safe
apply (erule subst)
apply (rule-tac [2] ext)
apply auto
done

lemma fun-upd-idem: f x = y ⇒ f(x := y) = f
by (simp only: fun-upd-idem-iff)

lemma fun-upd-triv [iff]: f(x := f x) = f
by (simp only: fun-upd-idem)

lemma fun-upd-apply [simp]: (f(x := y)) z = (if z = x then y else f z)
by (simp add: fun-upd-def)

lemma fun-upd-same: (f(x := y)) x = y
by simp

lemma fun-upd-other: z ≠ x ⇒ (f(x := y)) z = f z
by simp

lemma fun-upd-upd [simp]: f(x := y, x := z) = f(x := z)
by (simp add: fun-eq-iff)

lemma fun-upd-twist: a ≠ c ⇒ (m(a := b))(c := d) = (m(c := d))(a := b)
by (rule ext) auto

lemma inj-on-fun-updI: inj-on f A ⇒ y ∉ f ' A ⇒ inj-on (f(x := y)) A
by (auto simp: inj-on-def)

lemma fun-upd-image: f(x := y) ' A = (if x ∈ A then insert y (f ' (A - {x}))
else f ' A)
by auto

lemma fun-upd-comp: f o (g(x := y)) = (f o g)(x := f y)
THEORY "Fun"

by auto

lemma fun-upd-eqD: f(x := y) = g(x := z) → y = z
  by (simp add: fun-eq-iff split: if-split-asm)

8.7 override-on

definition override-on :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a set ⇒ 'a ⇒ 'b
  where override-on f g A = (λa. if a ∈ A then g a else f a)

lemma override-on-emptyset[simp]: override-on f g {} = f
  by (simp add: override-on-def)

lemma override-on-apply-notin[simp]: a ∉ A ⇒ (override-on f g A) a = f a
  by (simp add: override-on-def)

lemma override-on-apply-in[simp]: a ∈ A ⇒ (override-on f g A) a = g a
  by (simp add: override-on-def)

lemma override-on-insert: override-on f g (insert x X) = (override-on f g X)(x:=g x)
  by (simp add: override-on-def fun-eq-iff)

lemma override-on-insert': override-on f g (insert x X) = (override-on (f(x:=g x)) g X)
  by (simp add: override-on-def fun-eq-iff)

8.8 swap

definition swap :: 'a ⇒ 'a ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'b)
  where swap a b f = f (a := f b, b := f a)

lemma swap-apply [simp]:
  swap a b f a = f b
  swap a b f b = f a
  c ≠ a ⇒ c ≠ b ⇒ swap a b f c = f c
  by (simp-all add: swap-def)

lemma swap-self [simp]: swap a a f = f
  by (simp add: swap-def)

lemma swap-commute: swap a b f = swap b a f
  by (simp add: fun-upd-def swap-def fun-eq-iff)

lemma swap-nilpotent [simp]: swap a b (swap a b f) = f
  by (rule ext) (simp add: fun-upd-def swap-def)

lemma swap-comp-involutory [simp]: swap a b ∘ swap a b = id
  by (rule ext) simp
lemma swap-triple:
  assumes $a \neq c$ and $b \neq c$
  shows $\operatorname{swap} a b (\operatorname{swap} b c (\operatorname{swap} a b f)) = \operatorname{swap} a c f$
  using assms by (simp add: fun-eq-iff swap-def)

lemma comp-swap: $f \circ \operatorname{swap} a b g = \operatorname{swap} a b (f \circ g)$
  by (rule ext) (simp add: fun-upd-def swap-def)

lemma swap-image-eq [simp]:
  assumes $a \in A$ $b \in A$
  shows $\operatorname{swap} a b f : A \rightarrow f : A$
  proof
    have subset: $\forall f. \operatorname{swap} a b f : A \subseteq f : A$
      using assms by (auto simp: image_iff swap-def)
    then have $\operatorname{swap} a b (\operatorname{swap} a b f) : A \subseteq (\operatorname{swap} a b f) : A$
      with subset[of f] show ?thesis by auto
  qed

lemma inj-on-imp-inj-on-swap:
  inj-on $f : A \Rightarrow a \in A \Rightarrow b \in A \Rightarrow$ inj-on $(\operatorname{swap} a b f) : A$
  by (auto simp add: inj-on-def swap-def)

lemma inj-on-swap-iff [simp]:
  assumes $A: a \in A b \in A$
  shows inj-on $(\operatorname{swap} a b f) : A \iff$ inj-on $f : A$
  proof
    assume inj-on $(\operatorname{swap} a b f) : A$
    with $A$ have inj-on $(\operatorname{swap} a b (\operatorname{swap} a b f)) : A$
      by (iprover intro: inj-on-imp-inj-on-swap)
    then show inj-on $f : A$ by simp
  next
    assume inj-on $f : A$
    with $A$ show inj-on $(\operatorname{swap} a b f) : A$
      by (iprover intro: inj-on-imp-inj-on-swap)
  qed

lemma surj-imp-surj-swap: $\operatorname{surj} f \Rightarrow \operatorname{surj} (\operatorname{swap} a b f)$
  by simp

lemma surj-swap-iff [simp]: $\operatorname{surj} (\operatorname{swap} a b f) \iff \operatorname{surj} f$
  by simp

lemma bij-betw-swap-iff [simp]: $x \in A \Rightarrow y \in A \Rightarrow \operatorname{bij-betw} (\operatorname{swap} x y f) : A B \iff \operatorname{bij-betw} f : A B$
  by (auto simp: bij-betw-def)

lemma bij-swap-iff [simp]: $\operatorname{bij} (\operatorname{swap} a b f) \iff \operatorname{bij} f$
  by simp
hide-const (open) swap

8.9 Inversion of injective functions

definition the-inv-into :: 'a set ⇒ ('a ⇒ 'b) ⇒ ('b ⇒ 'a)
where the-inv-into A f = (λx. THE y. y ∈ A ∧ f y = x)

lemma the-inv-into-f-f: inj-on f A ⇒ x ∈ A ⇒ the-inv-into A f (f x) = x
unfolding the-inv-into-def inj-on-def by blast

lemma f-the-inv-into-f: inj-on f A ⇒ y ∈ f A ⇒ f (the-inv-into A f y) = y
apply (simp add: the-inv-into-def)
apply (rule the1I2)
apply (blast dest: inj-onD)
apply blast
done

lemma the-inv-into-into: inj-on f A ⇒ x ∈ f A ⇒ A ⊆ B ⇒ the-inv-into A f x ∈ B
apply (simp add: the-inv-into-def)
apply (rule the1I2)
apply (blast dest: inj-onD)
apply blast
done

lemma the-inv-into-onto [simp]: inj-on f A ⇒ the-inv-into A f A' (f A') = A
by (fast intro: the-inv-into-into the-inv-into-f-f [symmetric])

lemma the-inv-into-f-eq: inj-on f A ⇒ f x = y ⇒ x ∈ A ⇒ the-inv-into A f y = x
apply (erule subst)
apply (erule the-inv-into-f-f)
apply assumption
done

lemma the-inv-into-comp:
inj-on f (g A) ⇒ inj-on g A ⇒ x ∈ g A ⇒ the-inv-into A (f ∘ g) x = (the-inv-into A g ∘ the-inv-into (g A) f) x
apply (rule the-inv-into-f-eq)
apply (fast intro: comp-inj-on)
apply (simp add: f-the-inv-into-f the-inv-into-into)
apply (simp add: the-inv-into-into)
done

lemma inj-on-the-inv-into: inj-on f A ⇒ inj-on (the-inv-into A f) (f A)
by (auto intro: inj-onI simp: the-inv-into-f-f)

lemma bij-betw-the-inv-into: bij-betw f A B ⇒ bij-betw (the-inv-into A f) B A
by (auto simp add: bij-betw-def inj-on-the-inv-into the-inv-into-into)
THEORY "Fun"

abbreviation the-inv :: (a ⇒ b) ⇒ (b ⇒ a)
  where the-inv f ≡ the-inv-into UNIV f

lemma the-inv-f-f: the-inv f (f x) = x if inj f
  using that UNIV-I by (rule the-inv-into-f-f)

8.10 Cantor’s Paradox

theorem Cantors-paradox: ∃ f. f : A = Pow A
proof
  assume ∃ f. f : A = Pow A
  then obtain f where f : f : A = Pow A ..
  let ?X = { a ∈ A. a ∉ f a }
  have ?X ∈ Pow A by blast
  then have ?X ∈ f : A by (simp only: f)
  then obtain x where x ∈ A and f x = ?X by blast
  then show False by blast
qed

8.11 Monotonic functions over a set

definition mono-on f A ≡ ∀ r s. r ∈ A ∧ s ∈ A ∧ r ≤ s → f r ≤ f s

lemma mono-onI:
  (∀ r s. r ∈ A → s ∈ A → r ≤ s → f r ≤ f s) → mono-on f A
  unfolding mono-on-def by simp

lemma mono-onD:
[ mono-on f A; r ∈ A; s ∈ A; r ≤ s ] → f r ≤ f s
  unfolding mono-on-def by simp

lemma mono-imp-mono-on: mono f → mono-on f A
  unfolding mono-def mono-on-def by auto

lemma mono-on-subset: mono-on f A → B ⊆ A → mono-on f B
  unfolding mono-on-def by auto

definition strict-mono-on f A ≡ ∀ r s. r ∈ A ∧ s ∈ A ∧ r < s → f r < f s

lemma strict-mono-onI:
(∀ r s. r ∈ A → s ∈ A → r < s → f r < f s) → strict-mono-on f A
  unfolding strict-mono-on-def by simp

lemma strict-mono-onD:
[ strict-mono-on f A; r ∈ A; s ∈ A; r < s ] → f r < f s
  unfolding strict-mono-on-def by simp

lemma mono-on-greaterD:
  assumes mono-on g A x ∈ A y ∈ A g x > (g y :: linorder) :: linorder

THEORY "Fun"

shows $x > y$
proof (rule contr)
  assume $\neg x > y$
  hence $x \leq y$ by (simp add: not-less)
from assms(1-3) and this have $g x \leq g y$ by (rule mono-onD)
with assms(4) show False by simp
qed

lemma strict-mono-inv:
  fixes $f :: ('a::linorder) \Rightarrow ('b::linorder)$
  assumes strict-mono and surj $f$ and $\forall x. g (f x) = x$
  shows strict-mono $g$
proof
  fix $x$ $y$::'b assume $x < y$
  from $\langle$ surj $f$$\rangle$ obtain $x'$ $y'$ where $[simp]: x = f x'$ $y = f y'$ by blast
  with $\langle$ $x < y$$\rangle$ and (strict-mono $f$) have $x' < y'$ by (simp add: strict-mono-less)
  with $\langle$ inv $\rangle$ show $g x < g y$ by simp
qed

lemma strict-mono-on-imp-inj-on:
  assumes strict-mono-on $f :: (\cdot ::$ linorder $) \Rightarrow (\cdot ::$ preorder $)$ $A$
  shows inj-on $f$ $A$
proof (rule inj-onI)
  fix $x$ $y$ assume $x \in A$ $y \in A$ $f x = f y$
  thus $x = y$
    by (cases $x$ $y$ rule: linorder-cases)
    (auto dest: strict-mono-onD[of assms, of $x$ $y$] strict-mono-onD[OF assms, of $y$ $x$])
qed

lemma strict-mono-on-leD:
  assumes strict-mono-on $f :: (\cdot ::$ linorder $) \Rightarrow (\cdot ::$ preorder $)$ $A$ $x \in A$ $y \in A$ $x \leq y$
  shows $f x \leq f y$
proof (insert le-less-linear[of $y$ $x$], elim disjE)
  assume $x < y$
  with assms have $f x < f y$ by (rule-tac strict-mono-onD[OF assms(1)]) simp-all
  thus ?thesis by (rule less-imp-le)
qed (insert assms, simp)

lemma strict-mono-on-eqD:
  fixes $f :: (\cdot ::$ linorder $) \Rightarrow (\cdot ::$ preorder $)$
  assumes strict-mono-on $f A f x = f y$ $x \in A$ $y \in A$
  shows $y = x$
using assms by (rule-tac linorder-cases[of $x$ $y$]) (auto dest: strict-mono-onD)

lemma strict-mono-on-imp-mono-on:
  strict-mono-on $f :: (\cdot ::$ linorder $) \Rightarrow (\cdot ::$ preorder $)$ $A \Rightarrow$ mono-on $f$ $A$
by (rule mono-onI, rule strict-mono-on-leD)
8.12 Setup

8.12.1 Proof tools

Simplify terms of the form \( f(\ldots, x:=y, \ldots, x:=z, \ldots) \) to \( f(\ldots, x:=z, \ldots) \)

\[
\text{simproc-setup fun-upd2} \quad (f(v := w, x := y)) = (\text{fn} - \Rightarrow
\]

\[
\begin{array}{l}
\text{let} \\
\quad \text{fun gen-fun-upd NONE T - - = NONE} \\
\quad \text{| gen-fun-upd (SOME f) T x y = SOME (Const (const-name \langle \text{fun-upd} \rangle, T)} \\
\text{\$ f \$ x \$ y) } \\
\quad \text{fun dest-fun-T1 (Type (-, T :: Ts)) = T} \\
\quad \text{fun find-double (t as Const (const-name \langle \text{fun-upd} \rangle, T) \$ f \$ x \$ y) =} \\
\quad \text{let} \\
\quad \quad \text{fun find (Const (const-name \langle \text{fun-upd} \rangle, T) \$ g \$ v \$ w) =} \\
\quad \quad \quad \text{if v aconv x then SOME g else gen-fun-upd (find g) T v w} \\
\quad \quad \text{| find t = NONE} \\
\quad \text{in (dest-fun-T1 T, gen-fun-upd (find f) T x y) end} \\
\end{array}
\]

\[
\text{val ss = simpset-of context}
\]

\[
\text{fun proc ctxt ct =} \\
\quad \text{let} \\
\quad \quad \text{val t = Thm.term-of ct} \\
\quad \text{in} \\
\quad \quad \text{(case find-double t of} \\
\quad \quad \quad (T, NONE) => NONE \\
\quad \quad \text{| (T, SOME rhs) =>} \\
\quad \quad \quad SOME (Goal.prove ctxt [] [] (Logic.mk_equals (t, rhs)))} \\
\quad \quad \quad (fn - =>} \\
\quad \quad \quad \text{resolve-tac ctxt [eq-reflection] 1 THEN} \\
\quad \quad \quad \text{resolve-tac ctxt @(thms ext) 1 THEN} \\
\quad \quad \quad \text{simp-tac (put-simpset ss ctxt) 1))} \\
\quad \text{end} \\
\text{in proc end}
\]

8.12.2 Functorial structure of types

ML-file \( \langle \text{Tools/functor.ML} \rangle \)

\[
\text{functor map-fun: map-fun} \\
\quad \text{by (simp-all add: fun-eq-iff)}
\]

\[
\text{functor vimage} \\
\quad \text{by (simp-all add: fun-eq-iff vimage-comp)}
\]

Legacy theorem names

\[
\text{lemmas o-def = comp-def} \\
\text{lemmas o-apply = comp-apply}
\]
lemmas o-assoc = comp-assoc [symmetric]
lemmas id-o = id-comp
lemmas o-id = comp-id
lemmas o-eq-dest = comp-eq-dest
lemmas o-eq-elim = comp-eq-elim
lemmas o-eq-dest-lhs = comp-eq-dest-lhs
lemmas o-eq-id-dest = comp-eq-id-dest
end

9 Complete lattices

theory Complete-Lattices
  imports Fun
begin

9.1 Syntactic infimum and supremum operations

class Inf =
  fixes Inf :: 'a set ⇒ 'a (⨅)

class Sup =
  fixes Sup :: 'a set ⇒ 'a (⨆)

syntax
  -INF1 :: pttrns ⇒ 'b ⇒ 'b ((3INF -. / -. /) [0, 10] 10)
  -INF :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((3INF -∈ / -. /) [0, 0, 10] 10)
  -SUP1 :: pttrns ⇒ 'b ⇒ 'b ((3SUP -. / -. /) [0, 10] 10)
  -SUP :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((3SUP -∈ / -. /) [0, 0, 10] 10)

translations
  ∏ x y. f ≡ ∏ x. ∏ y. f
  ∏ x. f ≡ ∏ (CONST range (λx. f))
  ∏ x∈A. f ≡ CONST Inf ((λx. f) * A)
  ∏ x y. f ≡ ∏ x. ∏ y. f
  ∏ x. f ≡ ∏ (CONST range (λx. f))
  ∏ x∈A. f ≡ CONST Sup ((λx. f) * A)

context Inf
begin

lemma INF-image: ∏ (g o f) = ∏ (g * A)
  by (simp add: image-comp)
lemma INF-identity-eq [simp]: (∏ x ∈ A. x) = ∏ A
    by simp

lemma INF-id [simp]: ∏ (id ' A) = ∏ A
    by simp

lemma INF-cong: A = B \implies (∀ x ∈ B \implies C x = D x) \implies ∏ (C ' A) = ∏ (D ' B)
    by (simp add: image-def)

lemma INF-cong-simp:
A = B \implies (∀ x ∈ B =simp=> C x = D x) \implies ∏ (C ' A) = ∏ (D ' B)
unfolding simp-implies-def by (fact INF-cong)

end

context Sup
begin

lemma SUP-image: ∪ (g ' f ' A) = ∪ ((g o f) ' A)
    by (fact Inf.INF-image)

lemma SUP-identity-eq [simp]: (∪ x ∈ A. x) = ∪ A
    by (fact Inf.INF-identity-eq)

lemma SUP-id [simp]: ∪ (id ' A) = ∪ A
    by (fact Inf.INF-id-eq)

lemma SUP-cong: A = B \implies (∀ x ∈ B =simp=> C x = D x) \implies ∪ (C ' A) = ∪ (D ' B)
    by (fact Inf.INF-cong)

lemma SUP-cong-simp:
A = B \implies (∀ x ∈ B =simp=> C x = D x) \implies ∪ (C ' A) = ∪ (D ' B)
by (fact Inf.INF-cong-simp)

end

9.2 Abstract complete lattices

A complete lattice always has a bottom and a top, so we include them into the following type class, along with assumptions that define bottom and top in terms of infimum and supremum.

class complete-lattice = lattice + Inf + Sup + bot + top +
asumes Inf-lower: \? x ∈ A \implies ∏ A ≤ \? x
    and Inf-greatest: (∀ x ∈ A \implies z ≤ x) \implies z ≤ ∏ A
    and Sup-upper: \? x ∈ A \implies x ≤ ∪ A
    and Sup-least: (∀ x ∈ A \implies x ≤ z) \implies ∪ A ≤ z
and \( \text{Inf-empty} \) [simp]: \( \bigcap \{\} = \top \)
and \( \text{Sup-empty} \) [simp]: \( \bigcup \{\} = \bot \)

begin

subclass bounded-lattice
proof
fix \( a \)
show \( \bot \leq a \) by (auto intro: Sup-least simp only: Sup-empty [symmetric])
show \( a \leq \top \) by (auto intro: Inf-greatest simp only: Inf-empty [symmetric])

qed

lemma dual-complete-lattice: class.complete-lattice Sup Inf sup (\( \geq \)) (\( > \)) inf \( \top \) \( \bot \)
by (auto intro!: class.complete-lattice.intro dual-lattice)
(unfold-locales, (fact Inf-empty Sup-empty Sup-upper Sup-least Inf-lower Inf-greatest)+)

end

context complete-lattice
begin

lemma Sup-eqI:
\( (\forall y. y \in A \Rightarrow y \leq x) \Rightarrow (\forall z. z \in A \Rightarrow z \leq y) \Rightarrow x \leq y) \Rightarrow \bigcup A = x \)
by (blast intro: antisym Sup-least Sup-upper)

lemma Inf-eqI:
\( (\forall i. i \in A \Rightarrow x \leq i) \Rightarrow (\forall y. (\forall i. i \in A \Rightarrow y \leq i) \Rightarrow y \leq x) \Rightarrow \bigcap A = x \)
by (blast intro: antisym Inf-greatest Inf-lower)

lemma SUP-eqI:
\( (\forall i. i \in A \Rightarrow f i \leq x) \Rightarrow (\forall y. (\forall i. i \in A \Rightarrow f i \leq y) \Rightarrow x \leq y) \Rightarrow (\bigcup i \in A. f i) = x \)
using Sup-eqI [of \( f \ A \) \( x \)] by auto

lemma INF-eqI:
\( (\forall i. i \in A \Rightarrow x \leq f i) \Rightarrow (\forall y. (\forall i. i \in A \Rightarrow f i \geq y) \Rightarrow x \geq y) \Rightarrow (\bigcap i \in A. f i) = x \)
using Inf-eqI [of \( f \ A \) \( x \)] by auto

lemma INF-lower: \( i \in A \Rightarrow (\bigcap i \in A. f i) \leq f i \)
using Inf-lower [of \( - f \ A \) \( x \)] by simp

lemma INF-greatest: \( (\forall i. i \in A \Rightarrow u \leq f i) \Rightarrow u \leq (\bigcap i \in A. f i) \)
using Inf-greatest [of \( f \ A \) \( x \)] by auto

lemma SUP-upper: \( i \in A \Rightarrow f i \leq (\bigcup i \in A. f i) \)
using Sup-upper [of \( f \ A \) \( x \)] by simp
**THEORY “Complete-Lattices”**

**lemma SUP-least:** \( \forall i. i \in A \implies f i \leq u \implies (\bigsqcup i \in A. f i) \leq u \)

**using** Sup-least \([of f \cdot A]\) **by** auto

**lemma Inf-lower2:** \( u \in A \implies u \leq v \implies \bigsqcap A \leq v \)

**using** Inf-lower \([of u \cdot A]\) **by** auto

**lemma INF-lower2:** \( i \in A \implies f i \leq u \implies (\bigsqcap i \in A. f i) \leq u \)

**using** INF-lower \([of i \cdot A \cdot f]\) **by** auto

**lemma SUP-upper2:** \( u \in A \implies v \leq u \implies \bigsqcup A \leq v \)

**using** Sup-upper \([of u \cdot A]\) **by** auto

**lemma SUP-upper2:** \( i \in A \implies u \leq f i \implies u \leq (\bigsqcup i \in A. f i) \)

**using** SUP-upper \([of i \cdot A \cdot f]\) **by** auto

**lemma le-Inf-iff:** \( b \leq \bigsqcap A \iff (\forall a \in A. b \leq a) \)

**by** (auto intro: Inf-greatest dest: Inf-lower)

**lemma le-INF-iff:** \( u \leq (\bigsqcap i \in A. f i) \iff (\forall i \in A. u \leq f i) \)

**using** le-Inf-iff \([of - f \cdot A]\) **by** simp

**lemma Sup-le-iff:** \( \bigsqcup A \leq b \iff (\forall a \in A. a \leq b) \)

**by** (auto intro: Sup-least dest: Sup-upper)

**lemma SUP-le-iff:** \( (\bigsqcup i \in A. f i) \leq u \iff (\forall i \in A. f i \leq u) \)

**using** SUP-le-iff \([of f \cdot A]\) **by** simp

**lemma Inf-insert:** \( \bigsqcap (\text{insert } a \cdot A) = a \cap \bigsqcap A \)

**by** (auto intro: le-infI le-infI1 le-infI2 antisym Inf-greatest Inf-lower)

**lemma INF-insert:** \( (\bigsqcap x \in \text{insert } a \cdot A. f x) = f a \cap \bigsqcap (f \cdot A) \)

**by** simp

**lemma Sup-insert:** \( \bigsqcup (\text{insert } a \cdot A. f x) = f a \cup \bigsqcup (f \cdot A) \)

**by** simp

**lemma INF-empty:** \( (\bigsqcap x \in \{}. f x) = \top \)

**by** simp

**lemma SUP-empty:** \( (\bigsqcup x \in \{}. f x) = \bot \)

**by** simp

**lemma Inf-UNIV:** \( \bigsqcap \text{UNIV} = \bot \)

**by** (auto intro!: antisym Inf-lower)
lemma Sup-UNIV [simp]: \( \bigcup \) UNIV = \( \top \)
by (auto intro!: antisym Sup-upper)

lemma Inf-eq-Sup: \( \bigcap \) A = \( \bigcup \) \( \{ b. \ \forall a \in A. \ b \leq a \} \)
by (auto intro: antisym Inf-lower Inf-greatest Sup-upper Sup-least)

lemma Sup-eq-Inf: \( \bigwedge \) A = \( \bigcap \) \( \{ b. \ \forall a \in A. \ a \leq b \} \)
by (auto intro: antisym Inf-lower Inf-greatest Sup-upper Sup-least)

lemma Inf-superset-mono: \( B \subseteq A \implies \bigwedge A \leq \bigwedge B \)
by (auto intro: Inf-greatest)

lemma Sup-subset-mono: \( A \subseteq B \implies \bigwedge A \leq \bigwedge B \)
by (auto intro: Sup-least)

lemma Inf-mono:
assumes \( \bigwedge b. \ b \in B \implies \exists a \in A. \ a \leq b \)
shows \( \bigwedge A \leq \bigwedge B \)
proof (rule Inf-greatest)
fix \( b \) assume \( b \in B \)
with assms obtain \( a \) where \( a \in A \) and \( a \leq b \) by blast
from \( a \in A \) have \( \bigwedge A \leq a \) by (rule Inf-lower)
with \( a \leq b \) show \( \bigwedge A \leq b \) by auto
qed

lemma INF-mono: \( \bigwedge m. \ m \in B \implies \exists n \in A. \ f n \leq g m \) \( \implies \bigwedge n \in B. \ f n \leq \bigwedge n \in B. \ g n \)
using INF-mono \( \{ f \} \ A \{ g \} \) by auto

lemma INF-mono': \( \bigwedge x. \ f x \leq g x \) \( \implies \bigwedge x \in A. \ f x \leq \bigwedge x \in A. \ g x \)
by (rule INF-mono) auto

lemma Sup-mono:
assumes \( \bigwedge a. \ a \in A \implies \exists b \in B. \ a \leq b \)
shows \( \bigwedge A \leq \bigwedge B \)
proof (rule Sup-least)
fix \( a \) assume \( a \in A \)
with assms obtain \( b \) where \( b \in B \) and \( a \leq b \) by blast
from \( b \in B \) have \( b \leq \bigwedge B \) by (rule Sup-upper)
with \( a \leq b \) show \( a \leq \bigwedge B \) by auto
qed

lemma SUP-mono: \( \bigwedge n. \ n \in A \implies \exists m \in B. \ f n \leq g m \) \( \implies \bigwedge n \in B. \ f n \leq \bigwedge n \in B. \ g n \)
using SUP-mono \( \{ f \} \ A \{ g \} \) by auto

lemma SUP-mono': \( \bigwedge x. \ f x \leq g x \) \( \implies \bigwedge x \in A. \ f x \leq \bigwedge x \in A. \ g x \)
by (rule SUP-mono) auto
lemma INF-superset-mono: \( B \subseteq A \implies (\forall x. x \in B \implies f x \leq g x) \implies (\cap x \in A. f x) \leq (\cap x \in B. g x) \)

— The last inclusion is POSITIVE!

by (blast intro: INF-mono dest: subsetD)

lemma SUP-subset-mono: \( A \subseteq B \implies (\forall x. x \in A \implies f x \leq g x) \implies (\cup x \in A. f x) \leq (\cup x \in B. g x) \)

by (blast intro: SUP-mono dest: subsetD)

lemma Inf-less-eq:
  assumes \( \forall v. v \in A \implies v \leq u \)
  and \( A \neq \{\} \)
  shows \( \cap A \leq u \)

proof
  from \( A \neq \{\} \) obtain \( v \in A \) by blast
  moreover from \( \{v \in A\} \) have \( v \leq u \) by blast
  ultimately show \( \chi \)thesis by (rule Inf-lower2)

qed

lemma less-eq-Sup:
  assumes \( \forall v. v \in A \implies u \leq v \)
  and \( A \neq \{\} \)
  shows \( u \leq \cup A \)

proof
  from \( A \neq \{\} \) obtain \( v \in A \) by blast
  moreover from \( \{v \in A\} \) have \( u \leq v \) by blast
  ultimately show \( \chi \)thesis by (rule Sup-upper2)

qed

lemma INF-eq:
  assumes \( \forall i. i \in A \implies \exists j \in B. f i \geq g j \)
  and \( \forall j. j \in B \implies \exists i \in A. g j \geq f i \)
  shows \( \cap (f \cdot A) = \cap (g \cdot B) \)

by (intro antisym INF-greatest) (blast intro: INF-lower2 dest: assms)+

lemma SUP-eq:
  assumes \( \forall i. i \in A \implies \exists j \in B. f i \leq g j \)
  and \( \forall j. j \in B \implies \exists i \in A. g j \leq f i \)
  shows \( \cup (f \cdot A) = \cup (g \cdot B) \)

by (intro antisym SUP-least) (blast intro: SUP-upper2 dest: assms)+

lemma less-eq-Inf-inter: \( \cap A \cup \cap B \leq \cap (A \cap B) \)

by (auto intro: Inf-greatest Inf-lower)

lemma Sup-inter-less-eq: \( \cup (A \cap B) \leq \cup A \cup \cup B \)

by (auto intro: Sup-least Sup-upper)

lemma Inf-union-distrib: \( \cap (A \cup B) = \cap A \cap \cap B \)

by (rule antisym) (auto intro: Inf-greatest Inf-lower le-infI1 le-infI2)
lemma **INF-union**: \( \bigcap \{ i \in A \cup B. M i \} = (\bigcap i \in A. M i) \cap (\bigcap i \in B. M i) \)

by (auto intro!: antisym INF-mono intro: le-infI1 le-infI2 INF-greatest INF-lower)

lemma **Sup-union-distrib**: \( \bigsqcup (A \cup B) = \bigsqcup A \sqcup \bigsqcup B \)

by (rule antisym)

lemma **SUP-union**: \( \bigvee i \in A \cup B. M i = (\bigvee i \in A. M i) \sqcup (\bigvee i \in B. M i) \)

by (auto intro!: antisym SUP-mono intro: le-supI1 le-supI2 SUP-least SUP-upper)

lemma **INF-inf-distrib**: \( d a \in A. f a \sqcap d a \in A. g a = d a \in A. f a \sqcap g a \)

by (rule antisym)

lemma **SUP-sup-distrib**: \( (\bigsqcup a \in A. f a) \sqcup (\bigsqcup a \in A. g a) = (\bigsqcup a \in A. f a \sqcup g a) \)

(is ?L = ?R)

proof (rule antisym)

show ?L \leq ?R

by (auto intro: le-supI1 le-supI2 SUP-upper SUP-mono)

show ?R \leq ?L

by (rule SUP-least) (auto intro: le-supI1 le-supI2 SUP-upper)

qed

lemma **Inf-top-conv** [simp]:

\( \prod A = \top \iff (\forall x \in A. x = \top) \)

\( \top = \prod A \iff (\forall x \in A. x = \top) \)

proof -

show \( \prod A = \top \iff (\forall x \in A. x = \top) \)

proof

assume \( \forall x \in A. x = \top \)

then have \( A = \{ \} \lor A = \{ \top \} \) by auto

then show \( \prod A = \top \) by auto

next

assume \( \prod A = \top \)

show \( \forall x \in A. x = \top \)

proof (rule ccontr)

assume \( \neg (\forall x \in A. x = \top) \)

then obtain \( x \) where \( x \in A \) and \( x \neq \top \) by blast

then obtain \( B \) where \( A = \text{insert } x B \) by blast

with \( \prod A = \top \) \( x \neq \top \) show False by simp

qed

qed

then show \( \top = \prod A \iff (\forall x \in A. x = \top) \) by auto

qed

lemma **INF-top-conv** [simp]:

\( (\prod x \in A. B x) = \top \iff (\forall x \in A. B x = \top) \)

\( \top = (\prod x \in A. B x) \iff (\forall x \in A. B x = \top) \)

using **Inf-top-conv** [of B : A] by simp-all
lemma Sup-bot-conv [simp]:
\[ \bigsqcup A = \bot \iff (\forall x \in A. x = \bot) \]
\[ \bot = \bigsqcup A \iff (\forall x \in A. x = \bot) \]
using dual-complete-lattice
by (rule complete-lattice.Inf-top-conv)+

lemma SUP-bot-conv [simp]:
\[ (\bigsqcup x \in A. B x) = \bot \iff (\forall x \in A. B x = \bot) \]
\[ \bot = (\bigsqcup x \in A. B x) \iff (\forall x \in A. B x = \bot) \]
using Sup-bot-conv [of B ' A] by simp-all

lemma INF-const [simp]: \( A \neq \{\} \Rightarrow (\prod i \in A. f) = f \)
by (auto intro: antisym INF-lower INF-greatest)

lemma SUP-const [simp]: \( A \neq \{\} \Rightarrow (\bigsqcup i \in A. f) = f \)
by (auto intro: antisym SUP-upper SUP-least)

lemma INF-top [simp]: \( (\prod x \in A. \top) = \top \)
by (cases A = \{\}) simp-all

lemma SUP-bot [simp]: \( (\bigsqcup x \in A. \bot) = \bot \)
by (cases A = \{\}) simp-all

lemma INF-commute: \( (\prod i \in A. \prod j \in B. f i j) = (\prod j \in B. \prod i \in A. f i j) \)
by (iprover intro: INF-lower INF-greatest order-trans antisym)

lemma SUP-commute: \( (\bigsqcup i \in A. \bigsqcup j \in B. f i j) = (\bigsqcup j \in B. \bigsqcup i \in A. f i j) \)
by (iprover intro: SUP-upper SUP-least order-trans antisym)

lemma INF-absorb:
assumes k \( \in I \)
shows A \( k \cap (\prod i \in I. A i) = (\prod i \in I. A i) \)
proof –
from assms obtain J where I = insert k J by blast
then show ?thesis by simp
qed

lemma SUP-absorb:
assumes k \( \in I \)
shows A \( k \sqcup (\bigsqcup i \in I. A i) = (\bigsqcup i \in I. A i) \)
proof –
from assms obtain J where I = insert k J by blast
then show ?thesis by simp
qed

lemma INF-inf-const1: \( I \neq \{\} \Rightarrow (\prod i \in I. \inf x (f i)) = \inf x (\prod i \in I. f i) \)
by (intro antisym INF-greatest inf-mono order-refl INF-lower)
(auto intro: INF-lower2 le-infI2 intro!: INF-mono)
lemma **INF-inf-const2**: \( I \neq \{ \} \implies (\bigcap i \in I. \inf (f i) \ x) = \inf (\bigcap i \in I. f i) \ x \)
using **INF-inf-const1** [of \( I \ x f \)] by (simp add: inf-commute)

lemma **INF-constant**: \( \bigcap y \in A. c = (if A = \{ \} then \top else c) \)
by simp

lemma **SUP-constant**: \( \bigsqcup y \in A. c = (if A = \{ \} then \bot else c) \)
by simp

lemma **less-INF-D**:
  assumes \( y < (\bigcap i \in A. f i) \ i \in A \)
  shows \( y < f i \)
proof –
  note \( \langle y < (\bigcap i \in A. f i) \rangle \)
  also have \( (\bigcap i \in A. f i) \leq f i \) using \( \langle i \in A \rangle \)
  by (rule INF-lower)
  finally show \( y < f i \).
qed

lemma **SUP-lessD**:
  assumes \( (\bigsqcup i \in A. f i) < y \ i \in A \)
  shows \( f i < y \)
proof –
  have \( f i \leq (\bigsqcup i \in A. f i) \)
  using \( \langle i \in A \rangle \) by (rule SUP-upper)
  also note \( (\bigsqcup i \in A. f i) < y \)
  finally show \( f i < y \).
qed

lemma **INF-UNIV-bool-expand**: \( \bigsqcap b. A \ b = A \ 
\top \ 
\sqcap A \ \bot \)
by (simp add: UNIV-bool inf-commute)

lemma **SUP-UNIV-bool-expand**: \( \bigsqcup b. A \ b = A \ \bot \ 
\sqcup A \ \top \)
by (simp add: UNIV-bool sup-commute)

lemma **Inf-le-Sup**:
  \( A \neq \{ \} \implies \Inf A \leq \Sup A \)
by (blast intro: Sup-upper2 Inf-lower ex-in-conv)

lemma **INF-le-SUP**:
  \( A \neq \{ \} \implies \inf (f' A) \leq (\bigcup f' A) \)
using Inf-le-Sup [of \( f' A \)] by simp

lemma **INF-eq-const**:
  \( I \neq \{ \} \implies (\bigcap i \in I. f i = x) \implies (\bigcap i \in I. f i) = x \)
by (auto intro: INF-eqI)

lemma **SUP-eq-const**:
  \( I \neq \{ \} \implies (\bigcup i \in I. f i = x) \implies (\bigsqcup f' I) = x \)
by (auto intro: SUP-eqI)

lemma **INF-eq-iff**:
  \( I \neq \{ \} \implies (\bigcap i \in I. f i = c) \implies (\bigcap f ' I) = c \leftrightarrow \)
THEORY “Complete-Lattices” 231

\[ (\forall i \in I. f_i = c) \]
by (auto intro: INF-eq-const INF-lower antisym)

lemma SUP-eq-iff: \[ I \neq \emptyset \implies (\forall i \in I \implies c \leq f_i) \implies \bigsqcup (f \cdot I) = c \iff (\forall i \in I . f_i = c) \]
by (auto intro: SUP-eq-const SUP-upper antisym)

end

context complete-lattice
begin

lemma Sup-Inf-le: \[ \sup (\inf ' {\{ f \cdot A \mid f . (\forall Y \in A . f Y \in Y)\}}) \leq \inf (\sup ' A) \]
by (rule SUP-least, clarify, rule INF-greatest, simp add: INF-lower2 Sup-upper)

end

class complete-distrib-lattice = complete-lattice +
  assumes Inf-Sup-le: \[ \inf (\sup ' A) \leq \sup (\inf ' {\{ f \cdot A \mid f . (\forall Y \in A . f Y \in Y)\}}) \]
begin

lemma Inf-Sup: \[ \inf (\sup ' A) = \sup (\inf ' {\{ f \cdot A \mid f . (\forall Y \in A . f Y \in Y)\}}) \]
by (rule antisym, rule Inf-Sup-le, rule Sup-Inf-le)

subclass distrib-lattice
proof
  fix a b c
  show a \sqcup b \sqcap c = (a \sqcup b) \cap (a \sqcap c)
  proof (rule antisym, simp-all, safe)
    show b \cap c \leq a \sqcup b
      by (rule le-infI1, simp)
    show b \cap c \leq a \sqcap c
      by (rule le-infI2, simp)
    have [simp]: a \sqcap c \leq a \sqcup b \cap c
      by (rule le-infI1, simp)
    have [simp]: b \sqcap a \leq a \sqcap b \cap c
      by (rule le-infI2, simp)
    have [(Sup' \cdot \{\{a, b\}, \{a, c\}\}) = 
      \bigsqcup (\inf ' {\{ f \cdot \{\{a, b\}, \{a, c\}\} \mid f \cdot (\forall Y \in \{\{a, b\}, \{a, c\}\}. f Y \in Y)})
      by (rule Inf-Sup)
    from this show (a \sqcup b) \cap (a \sqcap c) \leq a \sqcup b \cap c
      apply simp
      by (rule SUP-least, safe, simp-all)
  qed
  qed
end

context complete-lattice
begin
context
THEORY “Complete-Lattices”

fixes f :: 'a ⇒ 'b::complete-lattice
assumes mono f
begin

lemma mono-Inf: f (∏ A) ≤ (∏ x∈A. f x)
using ⟨mono f⟩ by (auto intro: complete-lattice-class.INF-greatest Inf-lower dest: monoD)

lemma mono-Sup: (∪ x∈A. f x) ≤ f (∪ A)
using ⟨mono f⟩ by (auto intro: complete-lattice-class.SUP-least Sup-upper dest: monoD)

lemma mono-INF: f (∏ i∈I. A i) ≤ (∏ x∈I. f (A x))
by (intro complete-lattice-class.INF-greatest monoD[OF ⟨mono f⟩] INF-lower)

lemma mono-SUP: (∪ x∈I. f (A x)) ≤ f (∪ i∈I. A i)
by (intro complete-lattice-class.SUP-least monoD[OF ⟨mono f⟩] SUP-upper)
end
end

class complete-boolean-algebra = boolean-algebra + complete-distrib-lattice
begin

lemma uminus-Inf: − (∏ A) = ∪(uminus ' A)
proof (rule antisym)
  show − ∏ A ≤ ∪(uminus ' A)
    by (rule compl-le-swap2, rule INF-greatest, rule compl-le-swap2, rule Sup-upper)
simp
  show ∪(uminus ' A) ≤ − ∏ A
    by (rule Sup-least, rule compl-le-swap1, rule Inf-lower) auto
qed

lemma uminus-INF: − (∏ x∈A. B x) = (∪ x∈A. − B x)
by (simp add: uminus-Inf image-image)

lemma uminus-Sup: − (∪ A) = ∏ (uminus ' A)
proof
  have ∪ A = − (∏ (uminus ' A))
    by (simp add: image-image uminus-INF)
  then show ?thesis by simp
qed

lemma uminus-SUP: − (∪ x∈A. B x) = (∏ x∈A. − B x)
by (simp add: uminus-Sup image-image)
end
class complete-linorder = linorder + complete-lattice
begin

lemma dual-complete-linorder:
class.complete-linorder Sup Inf (≥) (>) inf ⊤ ⊥
by (rule class.complete-linorder.intro, rule dual-complete-lattice, rule dual-linorder)

lemma complete-linorder-inf-min: inf = min
by (auto intro: antisym simp add: min-def fun-eq-iff)

lemma complete-linorder-sup-max: sup = max
by (auto intro: antisym simp add: max-def fun-eq-iff)

lemma Inf-less-iff: (staking) < a ←→ (∃x ∈ staking. x < a)
by (simp add: not-le [symmetric] le-Inf-iff)

lemma less-Sup-iff: a < ⨆i ∈ staking. f i ←→ (∃x ∈ staking. a < f x)
by (simp add: less-Sup-iff [of f ' staking])

lemma less-SUP-iff: a < (∐i ∈ staking. f i) ←→ (∃x ∈ staking. a < f x)
by (simp add: less-Sup-iff [of f ' staking])

lemma Sup-eq-top-iff [simp]: ⨆staking = ⊤ ←→ (∀x < ⊤. ∃i ∈ staking. x < i)
proof
assume *: ⨆staking = ⊤
show (∀x < ⊤. ∃i ∈ staking. x < i)
  unfolding * [symmetric]
proof (intro allI impI)
  fix x
  assume x < ⨆staking
  then show ∃i ∈ staking. x < i
    by (simp add: less-Sup-iff)
qed

next
assume ∀x < ⊤. ∃i ∈ staking. x < i
show ⨆staking = ⊤
proof (rule ccontr)
  assume ⨆staking ≠ ⊤
  with top-greatest [of ⨆staking] have ⨆staking < ⊤
  unfolding le-less by auto
  with * have ⨆staking < ⨆staking
    unfolding less-Sup-iff by auto
  then show False by auto
qed
qed
THEORY "Complete-Lattices"

lemma SUP-eq-top-iff [simp]: (∪i∈A. f i) = ⊤ ←→ (∀x<⊤. ∃i∈A. x < f i)
using Sup-eq-top-iff [of f · A] by simp

lemma Inf-eq-bot-iff [simp]: (∩A = ⊥ ←→ (∀x>⊥. ∃i∈A. i < x)
using dual-complete-linorder
by (rule complete-linorder.Sup-eq-top-iff)

lemma INF-eq-bot-iff [simp]: (∩i∈A. f i) = ⊥ ←→ (∀x>⊥. ∃i∈A. f i < x)
using Inf-eq-bot-iff [of f · A] by simp

lemma Inf-le-iff: ∩A ≤ x ←→ (∀y>x. ∃a∈A. y > a)
proof safe
fix y
assume x ≥ ∩A y > x
then have y > ∩A by auto
then show ∃a∈A. y > a
  unfolding Inf-less-iff .
qed (auto elim!: allE[of - ∩A] simp add: not-le[symmetric] Inf-lower)

lemma INF-le-iff: (∩f · A) ≤ x ←→ (∀y>x. ∃i∈A. y > f i)
using Inf-le-iff [of f · A] by simp

lemma le-Sup-iff: x ≤ ∪A ←→ (∀y<x. ∃a∈A. y < a)
proof safe
fix y
assume x ≤ ∪A y < x
then have y < ∪A by auto
then show ∃a∈A. y < a
  unfolding less-Sup-iff .
qed (auto elim!: allE[of - ∪A] simp add: not-le[symmetric] Sup-upper)

lemma le-SUP-iff: x ≤ ∪(f · A) ←→ (∀y<x. ∃i∈A. y < f i)
using le-Sup-iff [of f · A] by simp

end

9.3 Complete lattice on bool

instantiation bool :: complete-lattice
begin

definition [simp, code]: ∩A ←→ False ∈ A

definition [simp, code]: ∪A ←→ True ∈ A

instance
  by standard (auto intro: bool-induct)

end
lemma not-False-in-image-Ball [simp]: False ∉ P → A ∈ Ball A P
by auto

lemma True-in-image-Bex [simp]: True ∈ P → A ∈ Bex A P
by auto

lemma INF-bool-eq [simp]: (λA f. ∩(f ∈ A)) = Ball
by (simp add: fun-eq-iff)

lemma SUP-bool-eq [simp]: (λA f. ∪(f ∈ A)) = Bex
by (simp add: fun-eq-iff)

instance bool :: complete-boolean-algebra
by (standard, fastforce)

9.4 Complete lattice on - ⇒ -

instantiation fun :: (type, Inf) Inf
begin

definition ∩A = (λx. ∩f∈A. f x)

lemma Inf-apply [simp, code]: (∩A) x = (∩f∈A. f x)
by (simp add: Inf-fun-def)

instance ..
end

instantiation fun :: (type, Sup) Sup
begin

definition ∪A = (λx. ∪f∈A. f x)

lemma Sup-apply [simp, code]: (∪A) x = (∪f∈A. f x)
by (simp add: Sup-fun-def)

instance ..
end

instantiation fun :: (type, complete-lattice) complete-lattice
begin

instance
by standard (auto simp add: le-fun-def intro: INF-lower INF-greatest SUP-upper SUP-least)
9.5 Complete lattice on unary and binary predicates

**Lemma INF1-I**: \( (\forall P. P \in A \Rightarrow P \, a) \Rightarrow (\prod A) \, a \)
by auto

**Lemma INF1-D**: \( (\prod x \in A. B \, x) \, b \Rightarrow a \in A \Rightarrow B \, a \, b \)
by simp

**Lemma INF2-I**: \( (\forall r. r \in A \Rightarrow r \, a \, b) \Rightarrow (\prod A) \, a \, b \)
by auto

**Lemma INF2-D**: \( (\prod x \in A. B \, x) \, b \, c \Rightarrow a \in A \Rightarrow B \, a \, b \, c \)
by simp

**Lemma INF1-E**: 
assumes \( (\prod A) \, a \)
obtains \( P \, a \mid P \notin A \)
using assms by auto

**Lemma INF1-E**: 
assumes \( (\prod x \in A. B \, x) \, b \)
obtains \( B \, a \, b \mid a \notin A \)
using assms by auto

**Lemma INF2-E**: 
assumes \( (\prod A) \, a \, b \)
obtains \( r \, a \, b \mid r \notin A \)
using assms by auto
THEORY “Complete-Lattices”

lemma INF2-E:
assumes (∏x∈A. B x) b c
obtains B a b c | a /
using assms by auto

lemma Sup1-I: P ∈ A ⇒ P a ⇒ (∪ A) a
by auto

lemma SUP1-I: a ∈ A ⇒ B a b ⇒ (∪ x∈A. B x) b
by auto

lemma Sup2-I: r ∈ A ⇒ r a b ⇒ (∪ A) a b
by auto

lemma SUP2-I: a ∈ A ⇒ B a b c ⇒ (∪ x∈A. B x) b c
by auto

lemma Sup1-E:
assumes (∪ A) a
obtains P where P ∈ A and P a
using assms by auto

lemma SUP1-E:
assumes (∪ x∈A. B x) b
obtains x where x ∈ A and B x b
using assms by auto

lemma Sup2-E:
assumes (∪ A) a b
obtains r where r ∈ A r a b
using assms by auto

lemma SUP2-E:
assumes (∪ x∈A. B x) b c
obtains x where x ∈ A B x b c
using assms by auto

9.6 Complete lattice on - set

instantiation set :: (type) complete-lattice
begin

definition ∏ A = {x. ∏((λB. x ∈ B) i A)}
definition ∪ A = {x. ∪((λB. x ∈ B) i A)}

instance
by standard (auto simp add: less-eq-set-def Inf-set-def Sup-set-def le-fun-def)
end

9.6.1 Inter

abbreviation Inter :: 'a set set ⇒ 'a set (\cap)
where \cap S ≡ \bigcap S

lemma Inter-eq: \cap A = \{ x. \forall B ∈ A. x ∈ B \}
proof (rule set-eqI)
  fix x
  have (\forall Q ∈\{ P. \exists B ∈ A. P \iff x ∈ B \}. Q) \iff (\forall B ∈ A. x ∈ B)
    by auto
  then show x ∈ \cap A \iff x ∈ \{ x. \forall B ∈ A. x ∈ B \}
    by (simp add: Inf-set-def image-def)
qed

lemma Inter-iff [simp]: A ∈ \cap C \iff (\forall X ∈ C. A ∈ X)
  by (unfold Inter-eq) blast

lemma InterI [intro!]: (∀X. X ∈ C ⇒ A ∈ X) ⇒ A ∈ \cap C
  by (simp add: Inter-eq)

A "destruct" rule – every X in C contains A as an element, but A ∈ X can
hold when X ∈ C does not! This rule is analogous to spec.

lemma InterD [elim, Pure.elim]: A ∈ \cap C ⇒ X ∈ C ⇒ A ∈ X
  by auto

lemma InterE [elim]: A ∈ \cap C ⇒ (X \notin C ⇒ R) ⇒ (A ∈ X ⇒ R) ⇒ R
  -- "Classical" elimination rule – does not require proving X ∈ C.
  unfolding Inter-eq by blast

lemma Inter-lower: B ∈ A ⇒ \cap A ⊆ B
  by (fact Inf-lower)

lemma Inter-subset: (∀X. X ∈ A ⇒ X ⊆ B) ⇒ A ≠ {} ⇒ \cap A ⊆ B
  by (fact Inf-less-eq)

lemma Inter-greatest: (∀X. X ∈ A ⇒ C ⊆ X) ⇒ C ⊆ \cap A
  by (fact Inf-greatest)

lemma Inter-empty: \cap {} = UNIV
  by (fact Inf-empty)

lemma Inter-UNIV: \cap UNIV = {}
  by (fact Inf-UNIV)

lemma Inter-insert: \cap (insert a B) = a \cap B
  by (fact Inf-insert)
lemma Inter-Un-subset: \( \cap A \cup \cap B \subseteq \cap (A \cap B) \)
by (fact less-eq-Inf-inter)

lemma Inter-Un-distrib: \( (A \cup B) = \cap A \cap \cap B \)
by (fact Inf-anion-distrib)

lemma Inter-UNIV-conv [simp]:
\( \cap A = \text{UNIV} \iff (\forall x \in A. \ x = \text{UNIV}) \)
\( \text{UNIV} = \cap A \iff (\forall x \in A. \ x = \text{UNIV}) \)
by (fact Inf-top-conv)+

lemma Inter-anti-mono: \( B \subseteq A \Rightarrow \cap A \subseteq \cap B \)
by (fact Inf-superset-mono)

9.6.2 Intersections of families

syntax (ASCII)
-INTER1 :: pttrns \Rightarrow 'b set \Rightarrow 'b set \((3\cap -/ -) [0, 10] 10) 
-INTER :: pttrn \Rightarrow 'a set \Rightarrow 'b set \Rightarrow 'b set \((3\cap -\in -/ -) [0, 0, 10] 10) 

syntax (latex output)
-INTER1 :: pttrns \Rightarrow 'b set \Rightarrow 'b set \((3\cap (\langle \text{unbreakable}:\in \rangle -) [0, 10] 10) 
-INTER :: pttrn \Rightarrow 'a set \Rightarrow 'b set \Rightarrow 'b set \((3\cap (\langle \text{unbreakable}:\in \rangle -) [0, 0, 10] 10) 

translations
\( \cap x. f = \cap x. \cap y. f \)
\( \cap x. f = \cap (\text{CONST range} (\lambda x. f)) \)
\( \cap x \in A. f = \text{CONST Inter} ((\lambda x. f) \ ' A) \)

lemma INTER-eq: \( (\cap x \in A. B x) = \{ y. \forall x \in A. y \in B x \} \)
by (auto intro!: INF-eqI)

lemma INT-iff [simp]: \( b \in (\cap x \in A. B x) \iff (\forall x \in A. b \in B x) \)
using Inter-iff [of - B ' A] by simp

lemma INT-I [intro!]: \( (\forall x. x \in A \Rightarrow b \in B x) \Rightarrow b \in (\cap x \in A. B x) \)
by auto

lemma INT-D [elim, Pure.elim]: \( b \in (\cap x \in A. B x) \Rightarrow a \in A \Rightarrow b \in B a \)
by auto

lemma INT-E [elim]: \( b \in (\cap x \in A. B x) \Rightarrow (b \in B a \Rightarrow R) \Rightarrow (a \notin A \Rightarrow R) \Rightarrow R \)
— "Classical" elimination — by the Excluded Middle on \( a \in A \).

by auto

\textbf{lemma} Collect-ball-eq: \( \{ x. \forall y \in A. \ P x y \} = (\bigcap y \in A. \ \{ x. \ P x y \}) \)

by blast

\textbf{lemma} Collect-all-eq: \( \{ x. \forall y. \ P x y \} = (\bigcap y. \ \{ x. \ P x y \}) \)

by blast

\textbf{lemma} \( \text{INT-lower} \): \( a \in A \Rightarrow (\bigcap x \in A. \ B x) \subseteq B a \)

by (fact \text{INF-lower})

\textbf{lemma} \( \text{INT-greatest} \): \( (\bigwedge x. \ x \in A \Rightarrow C \subseteq B x) \Rightarrow C \subseteq (\bigcap x \in A. \ B x) \)

by (fact \text{INF-greatest})

\textbf{lemma} \( \text{INT-empty} \): \( (\bigcap x \in \{ \}. \ B x) = \text{UNIV} \)

by (fact \text{INF-empty})

\textbf{lemma} \( \text{INT-absorb} \): \( k \in I \Rightarrow A k \cap (\bigcap i \in I. \ A i) = (\bigcap i \in I. \ A i) \)

by (fact \text{INF-absorb})

\textbf{lemma} \( \text{INT-subset-iff} \): \( B \subseteq (\bigcap i \in I. \ A i) \iff (\forall i \in I. \ B \subseteq A i) \)

by (fact \text{le-INF-iff})

\textbf{lemma} \( \text{INT-insert [simp]} \): \( (\bigcap x \in \text{insert} \ a \ A. \ B x) = B a \cap (\bigcap B ' A) \)

by blast

\textbf{lemma} \( \text{INT-constant [simp]} \): \( (\bigcap y \in A. \ c) = (\text{if } A = \{ \} \text{ then } \text{UNIV} \text{ else } c) \)

by (fact \text{INF-constant})

\textbf{lemma} \( \text{INTER-UNIV-conv} \):

\( (\text{UNIV} = (\bigcap x \in A. \ B x)) = (\forall x \in A. \ B x = \text{UNIV}) \)

\( ((\bigcap x \in A. \ B x) = \text{UNIV}) = (\forall x \in A. \ B x = \text{UNIV}) \)

by (fact \text{INF-top-conv})

\textbf{lemma} \( \text{INT-bool-eq} \): \( (\bigcap b. \ A b) = A \ True \cap A \ False \)

by (fact \text{INF-UNIV-bool-expand})

\textbf{lemma} \( \text{INT-anti-mono} \): \( A \subseteq B \Rightarrow (\forall x. \ x \in A \Rightarrow f x \subseteq g x) \Rightarrow (\bigcap x \in B. \ f x) \subseteq (\bigcap x \in A. \ g x) \)

— The last inclusion is POSITIVE!

by (fact \text{INF-superset-mono})
lemma **Pow-INT-eq**: $\text{Pow} \left( \bigcap_{x \in A} B \ x \right) = \left( \bigcap_{x \in A} \text{Pow} B \ x \right)$
  
  by blast

lemma **vimage-INT**: $f^{-1} \left( \bigcap_{x \in A} B \ x \right) = \left( \bigcap_{x \in A} f^{-1} B \ x \right)$

  by blast

9.6.3 **Union**

abbreviation **Union** :: 
  'a set set ⇒ 'a set 
where $\bigcup S \equiv \bigcup S$

lemma **Union-eq**: $\bigcup A = \{ x. \exists B \in A. x \in B \}$

proof (rule set-eqI)
  
  fix $x$
  
  have $\exists Q \in \{ P. \exists B \in A. P \iff x \in B \}. Q \iff (\exists B \in A. x \in B)$
  
  by auto
  
  then show $x \in \bigcup A \iff x \in \{ x. \exists B \in A. x \in B \}$
  
  by (simp add: Sup-set-def image-def)

qed

lemma **Union-iff** [simp]: $A \in \bigcup C \iff (\exists X \in C. A \in X)\quad$

by (unfold Union-eq) blast

lemma **UnionI** [intro]: $X \in C \Rightarrow A \in X \Rightarrow A \in \bigcup C$
  
  — The order of the premises presupposes that $C$ is rigid; $A$ may be flexible.

  by auto

lemma **UnionE** [elim!]: $A \in \bigcup C \Rightarrow (\forall X. A \in X \Rightarrow X \in C \Rightarrow R) \Rightarrow R$

  by auto

lemma **Union-upper**: $B \in A \Rightarrow B \subseteq \bigcup A$

  by (fact Sup-upper)

lemma **Union-least**: $(\forall X. X \in A \Rightarrow X \subseteq C) \Rightarrow \bigcup A \subseteq C$

  by (fact Sup-least)

lemma **Union-empty**: $\bigcup \{ \} = \{ \}$

  by (fact Sup-empty)

lemma **Union-UNIV**: $\bigcup \text{UNIV} = \text{UNIV}$

  by (fact Sup-UNIV)

lemma **Union-insert** [simp]: $\bigcup (\text{insert} \ a \ B) = a \cup \bigcup B$

  by (fact Sup-insert)

lemma **Union-Un-distrib** [simp]: $\bigcup (A \cup B) = \bigcup A \cup \bigcup B$

  by (fact Sup-union-distrib)
Lemma Union-Int-subset: \( \bigcup (A \cap B) \subseteq \bigcup A \cap \bigcup B \)
by (fact Sup-inter-less-eq)

Lemma Union-empty-conv: \( \bigcup A = \{\} \) \(\iff\) \((\forall x \in A. \ x = \{\})\)
by (fact Sup-bot-conv)

Lemma Empty-Union-conv: \( \{\} = \bigcup A \) \(\iff\) \((\forall x \in A. \ x = \{\})\)
by (fact Sup-bot-conv)

Lemma subset-Pow-Union: \( A \subseteq \text{Pow}(\bigcup A) \)
by blast

Lemma Union-Pow-eq: \( \bigcup (\text{Pow} \ A) = A \)
by blast

Lemma Union-mono: \( A \subseteq B \implies \bigcup A \subseteq \bigcup B \)
by (fact Sup-subset-mono)

Lemma Union-subsetI: \( (\forall x. \ x \in A \implies \exists y. \ y \in B \land x \subseteq y) \implies \bigcup A \subseteq \bigcup B \)
by blast

Lemma disjoint-inj-on-iff:
\[ \text{inj-on } f (\bigcup A); X \in A; Y \in A \implies \text{disjnt } (f ' X) (f ' Y) \iff \text{disjnt } X Y \]
apply (auto simp: disjoint-def)
using inj-on-eq-iff by fastforce

Lemma disjoint-Union1 simp: disjoint (\bigcup A) B \(\iff\) \((\forall A \in A. \ \text{disjnt } A B)\)
by (auto simp: disjoint-def)

Lemma disjoint-Union2 simp: disjoint B (\bigcup A) \(\iff\) \((\forall A \in A. \ \text{disjnt } B A)\)
by (auto simp: disjoint-def)

9.6.4 Unions of families

Syntax (ASCII)
-UNION1 :: pttrns => 'b set => 'b set ((3UN -./-) [0, 10] 10)
-UNION :: pttrn => 'a set => 'b set => 'b set ((3UN -./-) [0, 0, 10] 10)

Syntax (latex output)
-UNION1 :: pttrns => 'b set => 'b set ((3\cup (\text{unbreakable}.-)/-) [0, 10] 10)
-UNION :: pttrn => 'a set => 'b set => 'b set ((3\cup (\text{unbreakable}.\ -)/-) [0, 0, 10] 10)
translations
\[ \bigcup x. y. f = \bigcup x. \bigcup y. f \]
\[ \bigcup x. f = \bigcup (\texttt{CONST range } (\lambda x. f)) \]
\[ \bigcup x \in A. f = \texttt{CONST Union } ((\lambda x. f) \ ' A) \]

Note the difference between ordinary syntax of indexed unions and intersections (e.g. \( \bigcup a \in A_1. B \)) and their \LaTeX\ rendition: \( \bigcup a \in A_1. B \).

**lemma** disjoint-UN-iff: \( \text{disjoint } A (\bigcup i \in I. B i) \iff (\forall i \in I. \text{disjoint } A (B i)) \)
by (auto simp: disjoint-def)

**lemma** UNION-eq: \( (\bigcup x \in A. B x) = \{ y. \exists x \in A. y \in B x \} \)
by (auto intro!: SUP-eqI)

**lemma** bind-UNION [code]: \( \text{Set. bind } A f = \bigcup (f ' A) \)
by (simp add: bind-def UNION-eq)

**lemma** member-bind [simp]: \( x \in \text{Set. bind } A f \iff x \in \bigcup (f ' A) \)
by (simp add: bind-UNION)

**lemma** Union-SetCompr-eq: \( \bigcup \{ f x \mid x. P x \} = \{ a. \exists x. P x \land a \in f x \} \)
by blast

**lemma** UN-I [intro]: \( a \in A \Longrightarrow b \in B a \Longrightarrow b \in (\bigcup x \in A. B x) \)
— The order of the premises presupposes that \( A \) is rigid; \( b \) may be flexible.
by auto

**lemma** UN-E [elim!]: \( b \in (\bigcup x \in A. B x) \Longrightarrow (\forall x \in A. b \in B x) \)
using UN-iff [of - B ' A] by simp

**lemma** UN-I [intro]: \( a \in A \Longrightarrow b \in B a \Longrightarrow b \in (\bigcup x \in A. B x) \)
by auto

**lemma** UN-upper: \( a \in A \Longrightarrow B a \subseteq (\bigcup x \in A. B x) \)
by (fact SUP-upper)

**lemma** UN-least: \( (\forall x. x \in A \Longrightarrow B x \subseteq C) \Longrightarrow (\bigcup x \in A. B x) \subseteq C \)
by (fact SUP-least)

**lemma** Collect-bex-eq: \( \{ x. \exists y \in A. P x y \} = (\bigcup y \in A. \{ x. P x y \}) \)
by blast

**lemma** UN-insert-distrib: \( u \in A \Longrightarrow (\bigcup x \in A. \text{insert } a (B x)) = \text{insert } a (\bigcup x \in A. B x) \)
by blast

**lemma** UN-empty: \( (\bigcup x \in \{ \}. B x) = \{} \)
by (fact SUP-empty)
lemma `UN-empty2`: $(\bigcup x \in A. \{\}) = \{\}$
  by (fact SUP-bot)

lemma `UN-absorb`: $k \in I \implies A_k \cup (\bigcup i \in I. A_i) = (\bigcup i \in I. A_i)$
  by (fact SUP-absorb)

lemma `UN-insert [simp]`: $(\bigcup x \in \text{insert } a A. B x) = B a \cup (\bigcup i \in B. A_i)$
  by (fact SUP-insert)

lemma `UN-Un [simp]`: $(\bigcup i \in A \cup B. M_i) = (\bigcup i \in A. M_i) \cup (\bigcup i \in B. M_i)$
  by (fact SUP-union)

lemma `UN-Un-flatten`: $(\bigcup x \in (\bigcup y \in A. B y). C x) = (\bigcup y \in A. \bigcup x \in B y. C x)$
  by blast

lemma `UN-subset-iff`: $(\bigcup i \in I. A_i) \subseteq B \iff (\forall i \in I. A_i \subseteq B)$
  by (fact SUP-le-iff)

lemma `UN-constant [simp]`: $(\bigcup y \in A. c) = (\text{if } A = \{\} \text{ then } \{\} \text{ else } c)$
  by (fact SUP-constant)

lemma `UNION-singleton-eq-range`: $(\bigcup x \in A. \{f x\}) = f \upharpoonright A$
  by blast

lemma `image-Union`: $f \upharpoonright \bigcup S = (\bigcup x \in S. f \upharpoonright x)$
  by blast

lemma `UNION-empty-conv`: 
(\{\} = (\bigcup x \in A. B x) \iff (\forall x \in A. B x = \{\})

lemma `Collect-ex-eq`: \{x. \exists y. P x y\} = (\bigcup y. \{x. P x y\})
  by blast

lemma `ball-UN`: $(\forall z \in (\bigcup (B \upharpoonright A). P z) \iff (\forall x \in A. \forall z \in B x. P z)$
  by blast

lemma `bex-UN`: $(\exists z \in (\bigcup (B \upharpoonright A). P z) \iff (\exists x \in A. \exists z \in B x. P z)$
  by blast

lemma `Un-eq-UN`: $A \cup B = (\bigcup b. \text{if } b \text{ then } A \text{ else } B)$
  by safe (auto simp add: if-split-mem2)

lemma `UN-bool-eq`: $(\bigcup b. A b) = (A \text{ True} \cup A \text{ False})$
  by (fact SUP-UNIV-bool-expand)

lemma `UN-Pow-subset`: $(\bigcup x \in A. \text{Pow}(B x)) \subseteq \text{Pow}(\bigcup x \in A. B x)$
  by blast
lemma **UN-mono**: 
\( A \subseteq B \implies (\forall x. x \in A \implies f x \subseteq g x) \implies (\bigcup x \in A. f x) \subseteq (\bigcup x \in B. g x) \) 
by (fact SUP-subset-mono)

lemma **vimage-Union**: \( f^{-1} (\bigcup A) = (\bigcup X \in A. f^{-1} X) \) 
by blast

lemma **vimage-UN**: \( f^{-1} (\bigcup x \in A. B x) = (\bigcup x \in A. f^{-1} B x) \) 
by blast

lemma **vimage-eq-UN**: \( f^{-1} B = \bigcup y \in B. f^{-1} \{y\} \) 
— NOT suitable for rewriting 
by blast

lemma **image-UN**: \( f^{-1} \bigcup (B^{-1} A) = (\bigcup x \in A. f^{-1} B x) \) 
by blast

lemma **UN-singleton [simp]**: \( (\bigcup x \in A. \{x\}) = A \) 
by blast

lemma **inj-on-image**: inj-on f \((\bigcup A) \implies inj-on (\{^{-1}\} f) A\) 
unfolding inj-on-def by blast

9.6.5 Distributive laws

lemma **Int-Union**: \( A \cap \bigcup B = (\bigcup C \in B. A \cap C) \) 
by blast

lemma **Un-Inter**: \( A \cup \bigcap B = (\bigcap C \in B. A \cup C) \) 
by blast

lemma **Int-Union2**: \( \bigcup B \cap A = (\bigcup C \in B. C \cap A) \) 
by blast

lemma **INT-Int-distrib**: \( (\bigcap i \in I. A i \cap B i) = (\bigcap i \in I. A i) \cap (\bigcap i \in I. B i) \) 
by (rule sym) (rule INF-inf-distrib)

lemma **UN-Un-distrib**: \( (\bigcup i \in I. A i \cup B i) = (\bigcup i \in I. A i) \cup (\bigcup i \in I. B i) \) 
by (rule sym) (rule SUP-sup-distrib)

lemma **Int-Inter-image**: \( (\bigcap x \in C. A x \cap B x) = \bigcap (A \{^{-1}\} C) \cap (B \{^{-1}\} C) \) 
by (simp add: INT-Int-distrib)

lemma **Int-Inter-eq**: \( A \cap \bigcap B = (if B={} then A else (\bigcap B \in B. A \cap B)) \) 
by auto
THEORY “Complete-Lattices”

lemma Un-Union-image: (∪ x∈C. A x ∪ B x) = ∪(A ∪ C) ∪ (∪B ∪ C)
— Devlin, Fundamentals of Contemporary Set Theory, page 12, exercise 5:
 by (simp add: UN-Un-distrib)

lemma Un-INT-distrib: B ∪ (∩i∈I. A i) = (∩i∈I. B ⊔ A i)
 by blast

lemma Int-UN-distrib: B ∩ (∪i∈I. A i) = (∪i∈I. B ∩ A i)
— Halmos, Naive Set Theory, page 35.
 by blast

lemma Int-UN-distrib2: (∪i∈I. A i) ∩ (∪j∈J. B j) = (∪i∈I. ∪j∈J. A i ∩ B j)
 by blast

lemma Un-INT-distrib2: (∩i∈I. A i) ⊔ (∪j∈J. B j) = (∩i∈I. ∩j∈J. A i ∪ B j)
 by blast

lemma Union-disjoint: (∪ C ∩ A = {}) ←→ (∀B∈C. B ∩ A = {})
 by blast

lemma SUP-UNION: (∪ x∈(∪y∈A. g y). f x) = (∪ y∈A. ∪x∈g y. f x :: - :: complete-lattice)
 by (rule order-antisym) (blast intro: SUP-least SUP-upper2)+

9.7 Injections and bijections

lemma inj-on-Inter: S ≠ {} ⇒ (∀A. A ∈ S ⇒ inj-on f A) ⇒ inj-on f (∩S)
 unfolding inj-on-def by blast

lemma inj-on-INTER: I ≠ {} ⇒ (∀i. i ∈ I ⇒ inj-on f (A i)) ⇒ inj-on f (∩i∈I. A i)
 unfolding inj-on-def by safe simp

lemma inj-on-UNION-chain:
 assumes chain: ∀i j. i ∈ I ⇒ j ∈ I ⇒ A i ≤ A j ∨ A j ≤ A i
 and inj: ∀i. i ∈ I ⇒ inj-on f (A i)
 shows inj-on f (∪i∈I. A i)
proof
 have x = y
 if *: i ∈ I j ∈ I
 and **: x ∈ A i y ∈ A j
 and ***: f x = f y
 for i j x y
 using chain [OF *]
proof
 assume A i ≤ A j
THEORY "Complete-Lattices"

with ** have \( x \in A \) by auto
with inj ** ** show ?thesis
  by (auto simp add: inj-on-def)
next
  assume \( A j \leq A i \)
  with ** have \( y \in A i \) by auto
  with inj ** ** show ?thesis
  by (auto simp add: inj-on-def)
qed
then show ?thesis
  by (unfold inj-on-def UNION-eq) auto
qed

lemma bij-betw-UNION-chain:
  assumes chain: \( \wedge i j\. i \in I \Longrightarrow j \in I \Longrightarrow A i \leq A j \lor A j \leq A i \)
  and bij: \( \wedge i\. i \in I \Longrightarrow bij\betw f (A i) (A' i) \)
  shows bij\betw f (\( \bigcup i \in I\. A i \)) (\( \bigcup i \in I\. A' i \))
unfolding bij\betw-def
proof safe
  have \( \bigwedge i\. i \in I \Longrightarrow inj\on f (A i) \)
    using bij bij\betw-def[of f] by auto
  then show inj\on f (\( \bigcup (A' \cdot I) \))
    using chain inj\on-UNION-chain[of I A f] by auto
next
  fix \( i \) \( x \)
  assume \(*: i \in I\ x \in A i \)
  with bij have \( f x \in A' i \)
    by (auto simp: bij\betw-def)
  with \(*\) show \( f x \in \bigcup (A' \cdot I) \) by blast
next
  fix \( i \) \( x' \)
  assume \(*: i \in I\ x' \in A' i \)
  with bij have \( \exists x \in A i\. x' = f x \)
    unfolding bij\betw-def by blast
  with \(*\) have \( \exists j \in I\. \exists x \in A j\. x' = f x \)
    by blast
  then show \( x' \in f \cdot \bigcup (A' \cdot I) \)
    by blast
qed

lemma image-INT: inj\on f C \( \Longrightarrow \forall x \in A\. B x \subseteq C \Longrightarrow j \in A \Longrightarrow f \cdot (\bigcap (B \cdot A)) = (\bigcap x \in A\. f \cdot B x) \)
by (auto simp add: inj\on-def) blast

lemma bij-image-INT: bij f \( \Longrightarrow f \cdot (\bigcap (B \cdot A)) = (\bigcap x \in A\. f \cdot B x) \)
by (auto simp: bij\def inj\def surj\def) blast

lemma UNION-fun-upd: \( \bigcup (A(i := B) \cdot J) = \bigcup (A \cdot (J - \{i\})) \cup (if i \in J then
B else {} by (auto simp add: set-eq-iff)

lemma bij-betw-Pow:
assumes bij-betw f A B
shows bij-betw (image f) (Pow A) (Pow B)
proof –
  from assms have inj-on f A by (rule bij-betw-imp-inj-on)
  then have inj-on f (⋃(Pow A)) by simp
  then have inj-on (image f) (Pow A) by (rule inj-on-image)
  moreover from assms have f ‘ A = B by (rule bij-betw-imp-surj-on)
  then have image f ‘ Pow A = Pow B by (rule image-Pow-surj)
  ultimately show thesis by simp
qed

9.7.1 Complement

lemma Compl-INT [simp]: − (∩ x∈A. B x) = (∪ x∈A. −B x)
by blast

lemma Compl-UN [simp]: − (∪ x∈A. B x) = (∩ x∈A. −B x)
by blast

9.7.2 Miniscoping and maxiscoping

Miniscoping: pushing in quantifiers and big Unions and Intersections.

lemma UN-simps [simp]:
  A B C. (∩ x∈C. insert a (B x)) = (if C={} then {} else insert a (∪ x∈C. B x))
  A B C. (∪ x∈C. A x ∪ B) = ((if C={} then {} else ∪ x∈C. A x) ∪ B)
  A B C. (∪ x∈C. A x ∪ B x) = ((if C={} then {} else A ∪ (∪ x∈C. B x))
  A B C. (∪ x∈C. A x ∩ B) = (∪ x∈C. A x) ∩ B
  A B C. (∪ x∈C. A x ∩ B x) = (A ∩ (∪ x∈C. B x))
  A B C. (∪ x∈C. A x − B) = (∪ x∈C. A x) − B
  A B C. (∪ x∈C. A − B x) = (A − (∩ x∈C. B x))
  A B. (∪ x∈A. B x) = (∪ y∈A. ∪ x∈y. B x)
  A B C. (∪ z∈(∪ (B ‘ A)). C z) = (∪ x∈A. ∪ z∈B x. C z)
  A B f. (∪ x∈fA. B x) = (∪ a∈A. B (f a))
by auto

lemma INT-simps [simp]:
  A B C. (∩ x∈C. A x ∩ B) = (if C={} then UNIV else (∩ x∈C. A x) ∩ B)
\[ \forall A \ B \ C \ (\bigcap x \in C. \ A \cap B \ x) = (\text{if } C = \{\} \text{ then } \text{UNIV else } A \cap (\bigcap x \in C. \ B \ x)) \]
\[ \forall A \ B \ C \ (\bigcap x \in C. \ A \times B) = (\text{if } C = \{\} \text{ then } \text{UNIV else } (\bigcap x \in C. \ A \times B)) \]
\[ \forall A \ B \ C \ (\bigcap x \in C. \ A \times B) = (\text{if } C = \{\} \text{ then } \text{UNIV else } A - (\bigcup x \in C. \ B \ x)) \]
\[ \forall A \ B \ C \ (\bigcap x \in C. \ insert a (B \ x)) = (\text{if } C = \{\} \text{ then } \text{UNIV else } (\bigcap x \in C. \ insert a (B \ x))) \]

by auto

**lemma** \text{UN-ball-bex-simps} [simp]:
\[ \forall A \ P \ (\forall x \in \bigcup A. \ P \ x) \iff (\forall y \in A. \exists x \in y. \ P \ x) \]
\[ \forall A \ B \ P \ (\forall x \in (\bigcap (B \ A)). \ P \ x) = (\forall a \in A. \forall x \in B \ a. \ P \ x) \]
\[ \forall A \ P \ (\exists y \in \bigcup A. \ P \ x) \iff (\exists y \in A. \exists x \in y. \ P \ x) \]

by auto

Maxiscoping: pulling out big Unions and Intersections.

**lemma** \text{UN-extend-simps}:
\[ \forall A \ B \ C \ (\bigcap x \in C. \ insert a (B \ x)) = (\text{if } C = \{\} \text{ then } \{a\} \text{ else } (\bigcup x \in C. \ insert a (B \ x))) \]
\[ \forall A \ B \ C \ (\bigcap x \in C. \ A \times B) = (\text{if } C = \{\} \text{ then } \text{UNIV else } (\bigcap x \in C. \ A \times B)) \]
\[ \forall A \ B \ C \ (\bigcap x \in C. \ A \times B) = (\text{if } C = \{\} \text{ then } \text{UNIV else } A - (\bigcup x \in C. \ B \ x)) \]

by auto

**lemma** \text{INT-extend-simps}:
\[ \forall A \ B \ C \ (\bigcap x \in C. \ A \times B) = (\text{if } C = \{\} \text{ then } \text{UNIV - B else } (\bigcap x \in C. \ A \times B)) \]
\[ \forall A \ B \ C \ (\bigcap x \in C. \ insert a (B \ x)) = (\text{if } C = \{\} \text{ then } \text{UNIV else } (\bigcap x \in C. \ insert a (B \ x))) \]
\[ \forall A \ B \ C \ (\bigcap x \in C. \ A \times B) = (\text{if } C = \{\} \text{ then } \text{UNIV else } (\bigcap x \in C. \ A \times B)) \]
\[ \forall A \ B \ C \ (\bigcap x \in C. \ A \times B) = (\text{if } C = \{\} \text{ then } \text{UNIV - B else } (\bigcap x \in C. \ A \times B)) \]

by auto

Finally

**lemmas** \text{mem-simps} =
10 Wrapping Existing Freely Generated Type’s Constructors

theory Ctr-Sugar
imports HOL
keywords
  print-case-translations :: diag and
  free-constructors :: thy-goal
begin

consts
 case-guard :: bool ⇒ 'a ⇒ ('a ⇒ 'b) ⇒ 'b
case-nil :: 'a ⇒ 'b
case-cons :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'b
case-elem :: 'a ⇒ 'b ⇒ 'a ⇒ 'b
case-abs :: ('c ⇒ 'b) ⇒ 'b

declare [[coercion-args case-guard − + −]]
declare [[coercion-args case-cons −]]
declare [[coercion-args case-abs −]]
declare [[coercion-args case-elem − +]]

ML-file ⟨Tools/Ctr-Sugar/case-translation.ML⟩

lemma iffI-np: [x ⇒ ¬ y; ¬ x ⇒ y] ⇒ ¬ x ↔ y
 by (erule iffI) (erule contrapos-np)

lemma iff-contradict:
 ¬ P ⇒ P ↔ Q ⇒ Q ⇒ R
 ¬ Q ⇒ P ↔ Q ⇒ P ⇒ R
 by blast+

ML-file ⟨Tools/Ctr-Sugar/ctr-sugar-util.ML⟩
ML-file ⟨Tools/Ctr-Sugar/ctr-sugar-tactics.ML⟩
ML-file ⟨Tools/Ctr-Sugar/ctr-sugar-code.ML⟩
ML-file ⟨Tools/Ctr-Sugar/ctr-sugar.ML⟩

Coinduction method that avoids some boilerplate compared with coinduct.

ML-file ⟨Tools/coinduction.ML⟩

end
11 Knaster-Tarski Fixpoint Theorem and inductive definitions

theory Inductive
imports Complete-Lattices Ctr-Sugar
keywords
  inductive coinductive inductive-cases inductive-simps :: thy-defn and
  monos and
  print-inductives :: diag and
  old-rep-datatype :: thy-goal and
  primrec :: thy-defn
begin

11.1 Least fixed points

context complete-lattice
begin

definition lfp :: ('a ⇒ 'a) ⇒ 'a
  where lfp f = Inf {u. f u ≤ u}

lemma lfp-lowerbound: f A ≤ A =⇒ lfp f ≤ A
  unfolding lfp-def by (rule Inf-lower) simp

lemma lfp-greatest: (∀u. f u ≤ u =⇒ A ≤ u) =⇒ A ≤ lfp f
  unfolding lfp-def by (rule Inf-greatest) simp

end

lemma lfp-fixpoint:
  assumes mono f
  shows f (lfp f) = lfp f
  unfolding lfp-def
proof (rule order-antisym)
  let ?H = {u. f u ≤ u}
  let ?a = ∩ ?H
  show f ?a ≤ ?a
  proof (rule Inf-lower)
    fix x
    assume x ∈ ?H
    then have ?a ≤ x by (rule Inf-lower)
    with ⟨mono f⟩ have f ?a ≤ f x ..
    also from ⟨x ∈ ?H⟩ have f x ≤ x ..
    finally show f ?a ≤ x ..
  qed
  show ?a ≤ f ?a
  proof (rule Inf-lower)
    from ⟨mono f⟩ and ⟨f ?a ≤ ?a⟩ have f (f ?a) ≤ f ?a ..
    then show f ?a ∈ ?H ..
  qed
lemma lfp-unfold: mono f ⇒ lfp f = f (lfp f)
  by (rule lfp-fixpoint [symmetric])

lemma lfp-const: lfp (λx. t) = t
  by (rule lfp-unfold) (simp add: mono-def)

lemma lfp-eqI: mono F =⇒ F x = x =⇒ (⋀z. F z = z =⇒ x ≤ z) =⇒ lfp F = x
  by (rule antisym) (simp-all add: lfp-lowerbound lfp-unfold [symmetric])

11.2 General induction rules for least fixed points

lemma lfp-ordinal-induct [case-names mono step union]:
  fixes f :: 'a::complete-lattice ⇒ 'a
  assumes mono: mono f
  and P-f: ⋀S. P S =⇒ S ≤ lfp f =⇒ P (f S)
  and P-Union: ⋀M. ∀S∈M. P S =⇒ P (Sup M)
  shows P (lfp f)
proof –
  let ?M = {S. S ≤ lfp f ∧ P S}
  from P-Union have P (Sup ?M) by simp
  also have Sup ?M = lfp f
  proof (rule antisym)
    show Sup ?M ≤ lfp f
      by (blast intro: Sup-least)
    then have f (Sup ?M) ≤ f (lfp f)
      by (rule mono [THEN monoD])
    then have f (Sup ?M) ≤ lfp f
      using mono [THEN lfp-unfold] by simp
    then have f (Sup ?M) ∈ ?M
      using P-Union by simp (intro P-f Sup-least, auto)
    then have f (Sup ?M) ≤ Sup ?M
      by (rule Sup-upper)
    then show lfp f ≤ Sup ?M
      by (rule lfp-lowerbound)
  qed
  finally show ?thesis .
  qed

theory lfp-induct:
  assumes mono: mono f
  and ind: f (inf (lfp f) P) ≤ P
  shows lfp f ≤ P
proof (induct rule: lfp-ordinal-induct)
  case mono
  show ?case by fact
  next
case (step S)  
then show ?case  
  by (intro order-trans[OF - ind] monoD[OF mono]) auto
next
  case (union M)
  then show ?case  
  by (auto intro: Sup-least)
qed

lemma lfp-induct-set:  
assumes lfp: \( a \in \text{lfp } f \)
  and mono: mono \( f \)
  and hyp: \( \forall x. x \in f (\text{lfp } f \cap \{ x. P x \}) \rightarrow P x \)
shows P a
  by (rule lfp-induct [THEN subsetD, THEN CollectD, OF mono - lfp]) (auto intro: hyp)

lemma lfp-ordinal-induct-set:  
assumes mono: mono \( f \)
  and P-f: \( \forall S. P S \rightarrow P (f S) \)
  and P-Union: \( \forall M. \forall S \in M. P S \rightarrow P (\bigcup M) \)
shows P (lfp f)
  using assms by (rule lfp-ordinal-induct)

Definition forms of \text{lfp-unfold} and \text{lfp-induct}, to control unfolding.

lemma def-lfp-unfold: \( h \equiv \text{lfp } f = \Rightarrow \text{mono } f = \Rightarrow h = f h \)
  by (auto intro!: lfp-unfold)

lemma def-lfp-induct: \( A \equiv \text{lfp } f = \Rightarrow \text{mono } f = \Rightarrow f (\text{inf } A P) \leq P \Rightarrow A \leq P \)
  by (blast intro: lfp-induct)

lemma def-lfp-induct-set:
\( A \equiv \text{lfp } f = \Rightarrow \text{mono } f = \Rightarrow a \in A \Rightarrow (\forall x. x \in f (A \cap \{ x. P x \}) \Rightarrow P x) \Rightarrow P a \)
  by (blast intro: lfp-induct-set)

Monotonicity of \text{lfp}!

lemma lfp-mono: \( \forall Z. f Z \leq g Z \Rightarrow \text{lfp } f \leq \text{lfp } g \)
  by (rule lfp-lowerbound [THEN lfp-greatest]) (blast intro: order-trans)

11.3 Greatest fixed points

context complete-lattice
begin

definition gfp :: \( 'a \Rightarrow 'a \Rightarrow 'a \)
  where gfp \( f = \text{Sup } \{ u. u \leq f u \} \)

lemma gfp-upperbound: \( X \leq f X \Rightarrow X \leq \text{gfp } f \)
by (auto simp add: gfp-def intro: Sup-upper)

**lemma gfp-least**: \( (\forall u. u \leq f u \Rightarrow u \leq X) \Rightarrow \text{gfp } f \leq X \)
  
  by (auto simp add: gfp-def intro: Sup-upper)

end

**lemma lfp-le-gfp**: mono \( f \Rightarrow \text{lfp } f \leq \text{gfp } f \)
  
  by (rule gfp-upperbound) (simp add: lfp-fixpoint)

**lemma gfp-fixpoint**:
  
  assumes mono \( f \)
  
  shows \( f (\text{gfp } f) = \text{gfp } f \)
  
  unfolding gfp-def
  
  proof (rule order-antisym)
    
    let \(?H = \{ u. u \leq f u \}\)
    
    let \(?a = \bigcup ?H \)
    
    show \(?a \leq f ?a \)
    
    proof (rule Sup-least)
      
      fix \( x \)
      
      assume \( x \in \ ?H \)
      
      then have \( x \leq f x \).
      
      also from \( (x \in \ ?H) \) have \( x \leq ?a \) by (rule Sup-upper)
      
      with \( \text{mono } f \) have \( f x \leq f ?a \).
      
      finally show \( x \leq f ?a \).
    
    qed
    
    show \( f ?a \leq ?a \)
    
    proof (rule Sup-upper)
      
      from \( \text{mono } f \) and \( \text{\( ?a \leq f ?a \) have } f ?a \leq f f ?a \).
      
      then show \( f ?a \in \ ?H \).
    
    qed
    
    qed

**lemma gfp-unfold**: mono \( f \Rightarrow \text{gfp } f = f (\text{gfp } f) \)
  
  by (rule gfp-fixpoint [symmetric])

**lemma gfp-const**: \( \text{gfp } (\lambda x. t) = t \)
  
  by (rule gfp-unfold) (simp add: mono-def)

**lemma gfp-eqI**: mono \( F \Rightarrow F x = x \Rightarrow (\forall z. F z = z \Rightarrow z \leq x) \Rightarrow \text{gfp } F = x \)
  
  by (rule antisym) (simp-all add: gfp-upperbound gfp-unfold[symmetric])

11.4 Coinduction rules for greatest fixed points

Weak version.

**lemma weak-coinduct**: \( a \in X \Rightarrow X \subseteq f X \Rightarrow a \in \text{gfp } f \)
  
  by (rule gfp-upperbound [THEN subsetD]) auto
lemma weak-coinduct-image: \( a \in X \rightarrow g'X \subseteq f (g'X) \rightarrow g \ a \in \text{gfp f} \)
apply (erule gfp-upperbound [THEN subsetD])
apply (erule imageI)
done

lemma coinduct-lemma: \( X \leq f (\text{sup } X (\text{gfp f})) \rightarrow \text{mono } f \rightarrow \text{sup } X (\text{gfp f}) \leq f (\text{sup } X (\text{gfp f})) \)
apply (rule gfp-unfold [THEN eq-refl])
apply (erule mono-sup)
apply (rule le-supI)
apply assumption
apply (rule order-trans)
apply assumption
apply (rule sup-ge2)
apply assumption
done

Strong version, thanks to Coen and Frost.

lemma coinduct-set: \( \text{mono } f \rightarrow a \in X \rightarrow X \subseteq f (X \cup \text{gfp f}) \rightarrow a \in \text{gfp f} \)
by (rule weak-coinduct[rotated], rule coinduct-lemma)

lemma gfp-fun-UnI2: \( \text{mono } f \rightarrow a \in \text{gfp f} \rightarrow a \in f (X \cup \text{gfp f}) \)
by (blast dest: gfp-fixpoint mono-Un)

lemma gfp-ordinal-induct[case-names mono step union]:
fixes f :: 'a::complete-lattice \Rightarrow 'a
assumes mono: \( \text{mono } f \)
and P-f: \( \forall S. \text{P } S \rightarrow \text{gfp f} \leq S \rightarrow \text{P } (f S) \)
and P-Union: \( \forall M. \forall S \in M. \text{P } S \rightarrow \text{P } (\text{Inf } M) \)
shows \( \text{P } (\text{gfp f}) \)
proof
let \( ?M = \{ S. \text{gfp f} \leq S \wedge P S \} \)
from P-Union have \( \text{P } (\text{Inf } ?M) \) by simp
also have \( \text{Inf } ?M = \text{gfp f} \)
proof (rule antisym)
show \( \text{gfp f} \leq \text{Inf } ?M \)
by (blast intro: Inf-greatest)
then have \( f (\text{gfp f}) \leq f (\text{Inf } ?M) \)
by (rule mono [THEN monoD])
then have \( \text{gfp f} \leq f (\text{Inf } ?M) \)
using mono [THEN gfp-unfold] by simp
then have \( f (\text{Inf } ?M) \in ?M \)
using P-Union by simp (intro P-f Inf-greatest, auto)
then have \( \text{Inf } ?M \leq f (\text{Inf } ?M) \)
by (rule Inf-lower)
then show \( \text{Inf } ?M \leq \text{gfp f} \)
by (rule gfp-upperbound)
qed
finally show ?thesis .

qed

lemma coinduct:
  assumes mono: mono f
  and ind: X ≤ f (sup X (gfp f))
  shows X ≤ gfp f
proof (induct rule: gfp-ordinal-induct)
  case mono
  then show ?case by fact
next
  case (step S)
  then show ?case by (intro order-trans[OF ind -] monoD[OF mono]) auto
next
  case (union M)
  then show ?case by (auto intro: mono Inf-greatest)
qed

11.5 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition X ⊆ f X to one expressed using both lfp and gfp

lemma coinduct3-mono-lemma: mono f =⇒ mono (λx. f x ∪ X ∪ B)
by (iprover intro: subset-refl monoI Un-mono monoD)

lemma coinduct3-lemma:
  X ⊆ f (lfp (λx. f x ∪ X ∪ gfp f)) =⇒ mono f =⇒
  lfp (λx. f x ∪ X ∪ gfp f) ⊆ f (lfp (λx. f x ∪ X ∪ gfp f))
apply (rule subset-trans)
  apply (erule coinduct3-mono-lemma [THEN lfp-unfold [THEN eq-refl]])
apply (rule Un-least [THEN Un-least])
  apply (rule subset-refl, assumption)
apply (rule gfp-unfold [THEN equalityD1, THEN subset-trans], assumption)
apply (rule monoD, assumption)
apply (subst coinduct3-mono-lemma [THEN lfp-unfold], auto)
done

lemma coinduct3: mono f =⇒ a ∈ X =⇒ X ⊆ f (lfp (λx. f x ∪ X ∪ gfp f)) =⇒
a ∈ gfp f
apply (rule coinduct3-lemma [THEN [2] weak-coinduct])
  apply (rule coinduct3-mono-lemma [THEN lfp-unfold, THEN ss subst])
  apply simp-all
done

Definition forms of gfp-unfold and coinduct, to control unfolding.

lemma def-gfp-unfold: A ≡ gfp f =⇒ mono f =⇒ A = f A
by (auto intro!: gfp-unfold)
THEORY "Inductive"

lemma def-coinduct: \( \text{A} \equiv \text{gfp f} \implies \text{mono f} \implies X \leq f (\sup X \text{A}) \implies X \leq \text{A} \)
by (iprover intro!: coinduct)

lemma def-coinduct-set: \( \text{A} \equiv \text{gfp f} \implies \text{mono f} \implies a \in X \implies X \subseteq f (X \cup \text{A}) \implies a \in \text{A} \)
by (auto intro!: coinduct-set)

lemma def-Collect-coinduct: \( \text{A} \equiv \text{gfp (\lambda w. Collect (P w))} = \implies \text{mono (\lambda w. Collect (P w))} = \implies a \in X = \implies (\forall z. z \in X \implies P (X \cup \text{A} z)) = \implies a \in \text{A} \)
by (erule def-coinduct-set) auto

Monotonicity of \text{gfp}!

lemma gfp-mono: \( (\forall Z. f Z \leq g Z) \implies \text{gfp f} \leq \text{gfp g} \)
by (rule gfp-upperbound [THEN gfp-least]) (blast intro: order-trans)

11.6 Rules for fixed point calculus

lemma lfp-rolling:
assumes mono g mono f
shows \( g (\text{lfp (\lambda x. f (g x)))} = \text{lfp (\lambda x. g (f x))} \)
proof (rule antisym)
  have \(*\): mono (\lambda x. f (g x))
  using assms by (auto simp: mono-def)
  show \( \text{lfp (\lambda x. lfp (f x))} \leq \text{lfp (\lambda x. f (g x))} \)
   by (rule lfp-lowerbound) (simp add: lfp-unfold[OF *, symmetric])
  show \( g (\text{lfp (\lambda x. f (g x)))} \leq \text{lfp (\lambda x. g (f x))} \)
   proof (rule lfp-greatest)
     fix u
     assume u: g (f u) \leq u
     then have \( g (\text{lfp (\lambda x. f (g x)))} \leq g (f u) \)
       by (intro assms[THEN monoD] lfp-lowerbound)
     with u show \( g (\text{lfp (\lambda x. f (g x)))} \leq u \)
       by auto
     qed
   qed

lemma lfp-lfp:
assumes f: \( \forall x y w z. x \leq y \implies w \leq z \implies f w \leq f y \)
shows \( \text{lfp (\lambda x. lfp (f x))} = \text{lfp (\lambda x. f x)} \)
proof (rule antisym)
  have \(*\): mono (\lambda x. f x)
    by (blast intro: monoI f)
  show \( \text{lfp (\lambda x. f x)} \leq \text{lfp (\lambda x. f x)} \)
    by (intro lfp-lowerbound) (simp add: lfp-unfold[OF *, symmetric])
  show \( \text{lfp (\lambda x. f x)} \leq \text{lfp (\lambda x. f x)} \)
    by (intro lfp-lowerbound) (simp add: lfp-unfold[OF *, symmetric])
  qed
show \( \text{lfp} (\lambda x. \text{lfp} (f x)) \geq \text{lfp} (\lambda x. f x x) \) \((\text{is } ?F \geq -)\)
proof (intro lfp-lowerbound)
  have \(*: ?F = \text{lfp} (f ?F)\)
    by (rule lfp-unfold) (blast intro: monoI lfp-mono f)
  also have \(\ldots = f ?F (\text{lfp} (f ?F))\)
    by (rule lfp-unfold) (blast intro: monoI lfp-mono f)
finally show \( f ?F ?F \leq ?F \)
  by (simp add: \(*\)[symmetric])
qed

lemma gfp-rolling:
  assumes mono g mono f
  shows \( g (\text{gfp} (\lambda x. f (g x))) = \text{gfp} (\lambda x. g (f x)) \)
proof (rule antisym)
  have \(*: \text{mono} (\lambda x. f (g x))\)
    using assms by (auto simp: mono-def)
  show \( g (\text{gfp} (\lambda x. f (g x))) \leq \text{gfp} (\lambda x. g (f x)) \)
    by (rule gfp-upperbound) (simp add: gfp-unfold[of \*, symmetric])
  show \( \text{gfp} (\lambda x. g (f x)) \leq g (\text{gfp} (\lambda x. f (g x))) \)
    by (rule gfp-least)
  fix \( u \)
  assume \( u: u \leq g (f u) \)
  then have \( g (f u) \leq g (\text{gfp} (\lambda x. f (g x))) \)
    by (intro assms[THEN monoD] gfp-upperbound)
  with \( u \) show \( u \leq g (\text{gfp} (\lambda x. f (g x))) \)
    by auto
qed

lemma gfp-gfp:
  assumes \( f: \forall x y w z. x \leq y \Longrightarrow w \leq z \Longrightarrow f x w \leq f y z \)
  shows \( \text{gfp} (\lambda x. \text{gfp} (f x)) = \text{gfp} (\lambda x. f x x) \)
proof (rule antisym)
  have \(*: \text{mono} (\lambda x. f x x)\)
    by (blast intro: monoI f)
  show \( \text{gfp} (\lambda x. f x x) \leq \text{gfp} (\lambda x. \text{gfp} (f x)) \)
    by (intro gfp-upperbound) (simp add: gfp-unfold[of \*, symmetric])
  show \( \text{gfp} (\lambda x. \text{gfp} (f x)) \leq \text{gfp} (\lambda x. f x x) \) \((\text{is } ?F \leq -)\)
    by (rule antisym)
  have \(*: ?F = \text{gfp} (f ?F)\)
    by (rule gfp-unfold) (blast intro: monoI gfp-mono f)
  also have \(\ldots = f ?F (\text{gfp} (f ?F))\)
    by (rule gfp-unfold) (blast intro: monoI gfp-mono f)
finally show \( ?F \leq f ?F ?F \)
  by (simp add: \(*\)[symmetric])
qed
11.7 Inductive predicates and sets

Package setup.

lemmas basic-monos =
  subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
  Collect-mono in-mono vimage-mono

lemma le-rel-bool-arg-iff: \( X \leq Y \leftrightarrow X \text{ False} \leq Y \text{ False} \land X \text{ True} \leq Y \text{ True} \)

unfolding le-fun-def le-bool-def using bool-induct by auto

lemma imp-conj-iff: \( (P \rightarrow Q) \land P = (P \land Q) \)
  by blast

lemma meta-fun-cong: \( P \equiv Q \Rightarrow P a \equiv Q a \)
  by auto

ML-file ⟨Tools/inductive.ML⟩

lemmas [mono] =
  imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
  imp-mono not-mono
  Ball-def Bex-def
  induct-rulify-fallback

11.8 The Schroeder-Bernstein Theorem

See also:

- $\text{ISABELLE_HOME/src/HOL/ex/Set_Theory.thy}$
- http://planetmath.org/proofofschroederbernsteintheoremusingtarskiknastertheorem
- Springer LNCS 828 (cover page)

theorem Schroeder-Bernstein:

fixes \( f :: 'a \Rightarrow 'b \) and \( g :: 'b \Rightarrow 'a \)
  and \( A :: 'a \text{ set} \) and \( B :: 'b \text{ set} \)

assumes inj1: inj-on \( f \) \( A \) and sub1: \( f ' A \subseteq B \)
  and inj2: inj-on \( g \) \( B \) and sub2: \( g ' B \subseteq A \)

shows \( \exists h. \text{ bij-betw} \ h \ A \ B \)

proof (rule exI, rule bij-betw-imageI)

define \( X \) where \( X = \text{lfp} (\lambda X. A - (g ' (B - (f ' X)))) \)

define \( g' \) where \( g' = \text{the-inv-into} (B - (f ' X)) \ g \)

let \( ?h = \lambda z. \text{ if } z \in X \text{ then } f z \text{ else } g' z \)

have \( X: X = A - (g ' (B - (f ' X))) \)
  unfolding X-def by (rule lfp-unfold) (blast intro: monoI)

then have X-compl: \( A - X = g ' (B - (f ' X)) \)
  using sub2 by blast
from inj? have inj2': inj-on g (B - (f' X))
  by (rule inj-on-subset) auto
with X-compl have *: g' '(A - X) = B - (f' X)
  by (simp add: g'-def)

from X have X-sub: X ⊆ A by auto
from X sub1 have fX-sub: f' X ⊆ B by auto

show ?h' A = B
proof -
  from X-sub have ?h' A = ?h' (X ∪ (A - X)) by auto
also have . . . = ?h' X ∪ ?h' (A - X) by (simp only: image-Un)
also have ?h' X = f' X by auto
also from * have ?h' (A - X) = B - (f' X) by auto
also from fX-sub have f' X ∪ (B - f' X) = B by blast
finally show ?thesis .
qed

show inj-on ?h A
proof -
  from inj1 X-sub have on-X: inj-on f X
    by (rule subset-inj-on)
  have on-X-compl: inj-on g' (A - X)
    unfolding g'-def X-compl
    by (rule inj-on-the-inv-into) (rule inj2')
  have impossible: False if eq: f a = g' b and a: a ∈ X and b: b ∈ A - X for a b
    proof -
      from a have fa: f a ∈ f' X by (rule imageI)
      from b have g' b ∈ g' '(A - X) by (rule imageI)
      with * have g' b ∈ − (f' X) by simp
      with eq fa show False by simp
  qed

show ?thesis
proof (rule inj-onI)
  fix a b
  assume h: ?h a = ?h b
  assume a ∈ A and b ∈ A
  then consider a ∈ X b ∈ X | a ∈ A - X b ∈ A - X
    | a ∈ X b ∈ A - X | a ∈ A - X b ∈ X
    by blast
  then show a = b
  proof cases
    case 1
    with h on-X show ?thesis by (simp add: inj-on-eq-iff)
  next
case 2
  with h on-X-compl show ?thesis by (simp add: inj-on-eq-iff)
next
  case 3
  with h impossible [of a b] have False by simp
  then show ?thesis ..
next
  case 4
  with h impossible [of b a] have False by simp
  then show ?thesis ..
qed
qed
qed

11.9 Inductive datatypes and primitive recursion

Package setup.

ML-file ⟨Tools/Old-Datatype/old-datatype-aux.ML⟩
ML-file ⟨Tools/Old-Datatype/old-datatype-prop.ML⟩
ML-file ⟨Tools/Old-Datatype/old-datatype-data.ML⟩
ML-file ⟨Tools/Old-Datatype/old-rep-datatype.ML⟩
ML-file ⟨Tools/BNF/bnf-fp-rec-sugar-util.ML⟩
ML-file ⟨Tools/Old-Datatype/old-primrec.ML⟩
ML-file ⟨Tools/BNF/bnf-lfp-rec-sugar.ML⟩

Lambda-abstractions with pattern matching:
syntax (ASCII)
- lam-pats-syntax :: cases-syn ⇒ 'a ⇒ 'b ((%-) 10)
syntax
- lam-pats-syntax :: cases-syn ⇒ 'a ⇒ 'b ((λ-) 10)
parse-translation ⇨
  let
    fun fun-tr ctxt [cs] =
    let
      val x = Syntax.free (fst (Name.variant x (Term.declare-term-frees cs Name.context)));
      val ft = Case-Translation.case-tr true ctxt [x, cs];
    in lambda x ft end
  in [(syntax-const (-lam-pats-syntax), fun-tr)] end
end

12 Cartesian products

theory Product-Type
12.1  \textit{bool} is a datatype

\textbf{free-constructors} (discs-sels) case-bool for \textit{True} \textbar\ \textit{False}
\begin{itemize}
\item by auto
\end{itemize}

Avoid name clashes by prefixing the output of \textit{old-rep-datatype} with \textit{old}.

\textbf{setup} \langle \textit{Sign.mandatory-path \textit{old}} \rangle

\textbf{old-rep-datatype} \textit{True} \textit{False} by (auto intro: bool-induct)

\textbf{setup} \langle \textit{Sign.parent-path} \rangle

But erase the prefix for properties that are not generated by \textit{free-constructors}.

\textbf{setup} \langle \textit{Sign.mandatory-path bool} \rangle

\textbf{lemmas} \textit{induct} = \textit{old.bool.induct}

\textbf{lemmas} \textit{inducts} = \textit{old.bool.inducts}

\textbf{lemmas} \textit{rec} = \textit{old.bool.rec}

\textbf{lemmas} \textit{simps} = bool.distinct bool.case bool.rec

\textbf{setup} \langle \textit{Sign.parent-path} \rangle

\textbf{declare} case-split [cases type: bool]
\begin{itemize}
\item — prefer plain propositional version
\end{itemize}

\textbf{lemma} \langle \text{code} \rangle: \textit{HOL.equal} \textit{False} \textit{P} \iff \neg \textit{P}

\textbf{and} \langle \text{code} \rangle: \textit{HOL.equal} \textit{True} \textit{P} \iff \textit{P}

\textbf{and} \langle \text{code} \rangle: \textit{HOL.equal} \textit{P} \textit{False} \iff \neg \textit{P}

\textbf{and} \langle \text{code} \rangle: \textit{HOL.equal} \textit{P} \textit{True} \iff \textit{P}

\textbf{and} \langle \text{code nbe} \rangle: \textit{HOL.equal} \textit{P} \textit{P} \iff \textit{True}

\begin{itemize}
\item by (simp-all add: \textit{equal})
\end{itemize}

\textbf{lemma} \textit{If-case-cert}:
\begin{itemize}
\item \textbf{assumes} \textit{CASE} \equiv (\lambda b. \textit{If} \ b \ \textit{f} \ \textit{g})
\item \textbf{shows} (\textit{CASE} \textit{True} \equiv \textit{f}) \& \& (\textit{CASE} \textit{False} \equiv \textit{g})
\item \textbf{using} \textit{assms} by simp-all
\end{itemize}

\textbf{setup} \langle \textit{Code.declare-case-global @\{thm \textit{If-case-cert}\}} \rangle

\textbf{code-printing}
\begin{itemize}
\item \textbf{constant} \textit{HOL.equal} :: \textit{bool} \Rightarrow \textit{bool} \Rightarrow \textit{bool} \Rightarrow (\textit{Haskell}) \textit{infix} 4 \equiv
\item \textbf{class-instance} \textit{bool} :: \textit{equal} \Rightarrow (\textit{Haskell}) –
\end{itemize}

12.2  The \textit{unit} type

\textbf{typedef} \textit{unit} = \{ \textit{True} \}
by auto

definition Unity :: unit ('()')
  where () = Abs-unit True

lemma unit-eq [no-atp]: u = ()
  by (induct u) (simp add: Unity-def)

Simplification procedure for unit-eq. Cannot use this rule directly — it loops!

simproc-setup unit-eq (x::unit) = 
  fn - => fn - => fn ct =>
  if HOLogic.is-unit (Thm.term_of ct) then NONE
  else SOME (mk-meta-eq @{thm unit-eq})

free-constructors case-unit for ()
  by auto

Avoid name clashes by prefixing the output of old-rep-datatype with old.

setup (Sign.mandatory-path old)

old-rep-datatype () by simp

setup (Sign.parent-path)

But erase the prefix for properties that are not generated by free-constructors.

setup (Sign.mandatory-path unit)

lemmas induct = old.unit.induct
lemmas inducts = old.unit.inducts
lemmas rec = old.unit.rec
lemmas simps = unit.case unit.rec

setup (Sign.parent-path)

lemma unit-all-eq1: (\x::unit. PROP P x) \equiv PROP P ()
  by simp

lemma unit-all-eq2: (\x::unit. PROP P) \equiv PROP P
  by (rule trie forall-equality)

This rewrite counters the effect of simproc unit-eq on \(\lambda u::unit. f\ u\), replacing
it by \(f\) rather than by \(\lambda u. f\ ()\).

lemma unit-abs-eta-conv [simp]; (\lambda u::unit. f ()\) = f
  by (rule ext) simp

lemma UNIV-unit: UNIV = {()}

by auto

instantiation unit :: default
begin

definition default = ()

instance ..
end

instantiation unit :: {complete-boolean-algebra,complete-linorder,wellorder}
begin

definition less-eq-unit :: unit ⇒ unit ⇒ bool
  where (-::unit) ≤ - ←→ True

lemma less-eq-unit [iff]: u ≤ v for u v :: unit
  by (simp add: less-eq-unit-def)

definition less-unit :: unit ⇒ unit ⇒ bool
  where (-::unit) < - ←→ False

lemma less-unit [iff]: ¬ u < v for u v :: unit
  by (simp-all add: less-eq-unit-def less-unit-def)

definition bot-unit :: unit
  where [code-unfold]: ⊥ = ()

definition top-unit :: unit
  where [code-unfold]: ⊤ = ()

definition inf-unit :: unit ⇒ unit ⇒ unit
  where [simp]: - ⊓ - = ()

definition sup-unit :: unit ⇒ unit ⇒ unit
  where [simp]: - ⊔ - = ()

definition Inf-unit :: unit set ⇒ unit
  where [simp]: ⨍- = ()

definition Sup-unit :: unit set ⇒ unit
  where [simp]: ⨆- = ()

definition uminus-unit :: unit ⇒ unit
  where [simp]: - - = ()

declare less-eq-unit-def [abs-def, code-unfold]
less-unit-def [abs-def, code-unfold]
12.3 The product type

12.3.1 Type definition

definition Pair-Rep :: 'a ⇒ 'b ⇒ 'a ⇒ 'b ⇒ bool
where Pair-Rep a b = (λx y. x = a ∧ y = b)

definition prod = {f. ∃ a b. f = Pair-Rep (a::'a) (b::'b)}

typedef ('a, 'b) prod ((×/ -) [21, 20]) 20 = prod :: ('a ⇒ 'b ⇒ bool) set

unfolding prod-def by auto
type-notation (ASCII)

prod (infixr * 20)

definition Pair :: 'a ⇒ 'b ⇒ 'a × 'b

where Pair a b = Abs-prod (Pair-Rep a b)

lemma prod-cases: (\a b. P (Pair a b)) ⇒ P p

by (cases p) (auto simp add: prod-def Pair-def Pair-Rep-def)

free-constructors case-prod for Pair fst snd

proof –

fix P :: bool and p :: 'a × 'b

show (\x1 x2. p = Pair x1 x2 ⇒ P) ⇒ P

by (cases p) (auto simp add: prod-def Pair-def Pair-Rep-def)

next

fix a c :: 'a and b d :: 'b

have Pair-Rep a b = Pair-Rep c d ⟷ a = c ∧ b = d

by (auto simp add: Pair-Rep-def fun-eq-iff)

moreover have Pair-Rep a b ∈ prod and Pair-Rep c d ∈ prod

by (auto simp add: prod-def)

ultimately show Pair a b = Pair c d ⟷ a = c ∧ b = d

by (simp add: Pair-def Abs-prod-inject)

qed

Avoid name clashes by prefixing the output of old-rep-datatype with old.

setup (Sign.mandatory-path old)

old-rep-datatype Pair

by (erule prod-cases) (rule prod.inject)

setup (Sign.parent-path)

But erase the prefix for properties that are not generated by free-constructors.

setup (Sign.mandatory-path prod)

declare old.prod.inject [iff del]

lemmas induct = old.prod.induct
lemmas inducts = old.prod.inducts
lemmas rec = old.prod.rec
lemmas simps = prod.inject prod.case prod.rec

setup (Sign.parent-path)

declare prod.case [nitpick-simp del]
declare old.prod.case-cong-weak [cong del]
declare prod.case-eq-if [mono]
declare prod.split [no-atp]
THEORY "Product-Type"

```plaintext
declare prod.split-asm [no-atp]

prod.split could be declared as [split] done after the Splitter has been speeded up significantly; precompute the constants involved and don’t do anything unless the current goal contains one of those constants.

12.3.2 Tuple syntax

Patterns – extends pre-defined type `pttrn` used in abstractions.

```plaintext
nonterminal (tuple-args and patterns)
syntax
-tuple :: 'a ⇒ tuple-args ⇒ 'a × 'b ((1'(_,/-))
-tuple-arg :: 'a ⇒ tuple-args (-)
-tuple-args :: 'a ⇒ tuple-args ⇒ tuple-args (-/-)
-pattern :: pttrn ⇒ patterns ⇒ pttrn ((/-/-))
:: pttrn ⇒ patterns (-)
-patterns :: pttrn ⇒ patterns ⇒ patterns (-/-)
-unit :: pttrn (′)
translations
(x, y) ⇌ CONST Pair x y
-pattern x y ⇌ CONST Pair x y
-pattns x y ⇌ CONST Pair x y
-tuple x (t-ptrn y z) ⇌ -tuple x (-t-ptrn y z)
λ(x, y, z). b ⇌ CONST case-prod (λx y, zs). b)
λ(x, y). b ⇌ CONST case-prod (λx y). b)
-abs (CONST Pair x y) t → λ(x, y). t
— This rule accommodates tuples in case C ...

print case-prod f as case-prod f and case-prod f as case-prod f
```

```plaintext
typed-print-translation

| let
| fun case-prod-guess-names-tr' T [Abs (x, -, Abs -)] = raise Match
| case-prod-guess-names-tr' T [Abs (x, xT, t)] =
  (case (head-of t) of
    Const (const-syntax case-prod, -) => raise Match
  | - =>
    let
    val (- :: yT :: -) = binder-types (domain-type T) handle Bind => raise Match;
    val (y, t') = Syntax-Trans.atomic-abs-tr' (y, yT, incr-boundvars 1 t $ Bound 0);
    val (x', t'') = Syntax-Trans.atomic-abs-tr' (x, xT, t');
    in
    Syntax.const syntax-const -abs; $
    (Syntax.const syntax-const (-pattern) $ x' $ y) $ t'"
```
THEORY "Product-Type"

raise Match;

val (xT :: yT :: -) = binder-types (domain-type T) handle Bind =>

end)

| case-prod-guess-names-tr' T [t] =
  (case head-of t of
   Const (const-syntax case-prod', -) => raise Match
   | _ =>
     let
     val (xT :: yT :: -) = binder-types (domain-type T) handle Bind =>
     raise Match;
     val (y, t') =
     Syntax-Trans.atomic-abs-tr' (y, yT, incr-boundvars 2 t $ Bound 1 $
     Bound 0);
     val (x', t'') = Syntax-Trans.atomic-abs-tr' (xT, x, t');
     in
     Syntax.const syntax-const (-abs) $
     (Syntax.const syntax-const (-pattern) $ x' $ y) $ t''
   end)
   |
   case-prod-guess-names-tr' - - = raise Match;
   in [(const-syntax case-prod, K case-prod-guess-names-tr')] end

Reconstruct pattern from (nested) case-prods, avoiding eta-contraction of
body; required for enclosing "let", if "let" does not avoid eta-contraction,
which has been observed to occur.

print-translation

let

fun case-prod-tr' [Abs (x, T, t as (Abs abs))] =
  (* case-prod (λx y. t) => λ(x, y) t *)
  let
    val (y, t') = Syntax-Trans.atomic-abs-tr' abs;
    val (x', t'') = Syntax-Trans.atomic-abs-tr' (xT, x, t');
    in
    Syntax.const syntax-const (-abs) $
    (Syntax.const syntax-const (-pattern) $ x' $ y) $ t''
  end
|
  case-prod-tr' [Abs (x, T, (s as Const (const-syntax case-prod', -) $ t))]
  =
  (* case-prod (λx. (case-prod (λy z. t))) => λ(x, y, z). t *)
  let
    val Const (syntax-const (-abs), -) $
    (Const (syntax-const (-pattern), -) $ y $ z) $ t' =
    case-prod-tr' [t];
    val (x', t'') = Syntax-Trans.atomic-abs-tr' (xT, x, t');
    in
    Syntax.const syntax-const (-abs) $
    (Syntax.const syntax-const (-pattern) $ x' $ y) $ t''
  end
|
  case-prod-tr' [Const (const-syntax case-prod', -) $ t] =
  (* case-prod (case-prod (λx y z. t)) => λ((x, y), z). t *)
  case-prod-tr' [(case-prod-tr' [t])]
THEORY “Product-Type”

\begin{verbatim}
(* inner case-prod-tr' creates next pattern *)
| case-prod-tr' [Const (syntax-const (-abs), -)] $ x-y $ Abs abs =
  (* case-prod (\pttrn z. t) \rightarrow \lambda(\pttrn, z). t *)
  let val (z, t) = Syntax-Trans.atomic-abs-tr' abs in
  Syntax.const syntax-const (-pattern) $ x-y $ z $ t end

| case-prod-tr' - = raise Match;
in [((const-syntax (case-prod), K case-prod-tr')] end

12.3.3 Code generator setup

code-printing
type-constructor prod \rightarrow
  (SML) infix 2 *
  and (OCaml) infix 2 *
  and (Haskell) !(((-),/ (-))
  and (Scala) !((-),/ (-))

| constant Pair \rightarrow
  (SML) !(((-),/ (-))
  and (OCaml) !(((-),/ (-))
  and (Haskell) !(((-),/ (-))
  and (Scala) !((-),/ (-))

| class-instance prod :: equal \rightarrow
  (Haskell) --

| constant HOL.equal :: 'a \times 'b \Rightarrow 'a \times 'b \Rightarrow bool \rightarrow
  (Haskell) infix 4 ==
| constant fst \rightarrow (Haskell) fst
| constant snd \rightarrow (Haskell) snd

12.3.4 Fundamental operations and properties

lemma Pair-inject: (a, b) = (a', b') \Rightarrow (a = a' \Rightarrow b = b' \Rightarrow R) \Rightarrow R
  by simp

lemma surj-pair [simp]: \exists x y. p = (x, y)
  by (cases p) simp

lemma fst-eqD: fst (x, y) = a \Rightarrow x = a
  by simp

lemma snd-eqD: snd (x, y) = a \Rightarrow y = a
  by simp

lemma case-prod-unfold [nitpick-unfold]: case-prod = (\lambda c p. c (fst p) (snd p))
  by (simp add: fun-eq-iff split: prod.split)

lemma case-prod-conv [simp, code]: (case (a, b) of (c, d) \Rightarrow f c d) = f a b
  by (fact prod.case)
\end{verbatim}
lemmas surjective-pairing = prod.collapse [symmetric]

lemma prod-eq-iff: s = t =⇒ fst s = fst t ∧ snd s = snd t
  by (cases s, cases t) simp

lemma prod-eqI [intro?]: fst p = fst q =⇒ snd p = snd q =⇒ p = q
  by (simp add: prod-eq-iff)

lemma case-prodI: f a b =⇒ case (a, b) of (c, d) =⇒ f c d
  by (rule prod.case [THEN iffD2])

lemma case-prodD: (case (a, b) of (c, d) =⇒ f c d) =⇒ f a b
  by (rule prod.case [THEN iffD1])

lemma case-prod-Pair [simp]: case-prod Pair = id
  by (simp add: fun-eq-iff split: prod.split)

lemma case-prod-comp: (case x of (a, b) =⇒ (f ∘ g) a b) = f (g (fst x)) (snd x)
  by (cases x simp)

lemma The-case-prod: The (case-prod P) = (THE xy. P (fst xy) (snd xy))
  by (simp add: case-prod-unfold)

lemma cond-case-prod-eq: (λ(x, y). f x y) = f
  — Subsumes the old split-Pair when f is the identity function.
  by (simp add: fun-eq-iff split: prod.split)

lemma split-paired-all (no-atp): (∀ x. PROP P x) = (∀ a b. PROP P (a, b))
proof
  fix a b
  assume ∀ x. PROP P x
  then show PROP P (a, b).
next
  fix x
  assume ∃ a b. PROP P (a, b)
  from ⟨PROP P (fst x, snd x)⟩ show PROP P x by simp
qed

The rule split-paired-all does not work with the Simplifier because it also
affects premises in congruence rules, where this can lead to premises of the
form ∃ a b. . . . = ?P(a, b) which cannot be solved by reflexivity.

lemmas split-tupled-all = split-paired-all unit-all-eq2
ML

(* replace parameters of product type by individual component parameters *)
local (* filtering with exists-paired-all is an essential optimization *)
fun exists-paired-all (Const (const-name (Pair.all), -) $ Abs (_, T, t)) =
  can HOLogic.dest-prodT T orelse exists-paired-all t
| exists-paired-all (t $ u) = exists-paired-all t orelse exists-paired-all u
| exists-paired-all (Abs (_, _, t)) = exists-paired-all t
| exists-paired-all _ = false;
val ss =
simpset-of
(put-simpset HOL-basic-ss context
adssimps [@{thm split-paired-all}, @{thm unit-all-eq2}, @{thm unit-abs-eta-conv}]
adssimps [simpproc (unit-eq)]);
in
fun split-all-tac ctxt = SUBGOAL (fn (t, i) =>
  if exists-paired-all t then safe-full-simp-tac (put-simpset ss ctxt) i else no-tac);

fun unsafe-split-all-tac ctxt = SUBGOAL (fn (t, i) =>
  if exists-paired-all t then full-simp-tac (put-simpset ss ctxt) i else no-tac);

fun split-all ctxt th =
  if exists-paired-all (Thm.prop_of th)
  then full-simplify (put-simpset ss ctxt) th else th;
end;

setup (map-theory-claset (fn ctxt => ctxt addSbefore (split-all-tac, split-all-tac)));

lemma split-paired-All [simp, no-atp]: (∀ x. P x) ←→ (∀ a b. P (a, b))
— [iff] is not a good idea because it makes blast loop
by fast

lemma split-paired-Ex [simp, no-atp]: (∃ x. P x) ←→ (∃ a b. P (a, b))
by fast

lemma split-paired-The [no-atp]: (THE x. P x) = (THE (a, b). P (a, b))
— Can't be added to simpset: loops!
by (simp add: case-prod-eta)

Simplification procedure for cond-case-prod-eta. Using case-prod-eta as a
rewrite rule is not general enough, and using cond-case-prod-eta directly
would render some existing proofs very inefficient; similarly for prod.case-eq-if.

ML

val cond-case-prod-eta-ss =
simpset-of (put-simpset HOL-basic-ss context adssimps @{thms cond-case-prod-eta});
fun Pair-pat k 0 (Bound m) = (m = k)
| Pair-pat k i (Const (const-name (Pair, -) $ Bound m $ t)) =
  i > 0 andalso m = k + i andalso Pair-pat k (i - 1) t
THEORY “Product-Type”

Pair-pat - - - = false;
fun no-args k i (Abs (_, _, t)) = no-args (k + 1) i t
| no-args k i (t $ u) = no-args k i t andalso no-args k i u
| no-args k i (Bound m) = m < k orelse m > k + i
| no-args - - - = true;
fun split-pat tp i (Abs (_, _, t)) = split-pat tp i (Const (const-name case-prod, _) $ Abs (_, _, t)) = split-pat tp (i + 1) t
| split-pat tp i - - - = NONE;
fun metaeq ctxt lhs rhs = mk-meta-eq (Goal.prove ctxt [] [] (HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs))))
| (K (simp-tac (put-simpset cond-case-prod-eta-ss ctxt) 1)));
fun beta-term-pat k i (Abs (_, _, t)) = beta-term-pat k i t
| beta-term-pat k i (t $ u) = Pair-pat k i (t $ u) orelse beta-term-pat k i t andalso beta-term-pat k i u
| beta-term-pat - - - = false;
fun eta-term-pat k i (f $ arg) = no-args k i f andalso Pair-pat k i arg
| eta-term-pat - - - = false;
fun subst arg k i (Abs x T t) = subst arg (k + 1) i t
| subst arg k i (t $ u) = if Pair-pat k i (t $ u) then incr-boundvars k arg
else (subst arg k i t $ subst arg k i u)
| subst arg k i t = t;
in
fun beta-proc ctxt (s as Const (const-name case-prod, _) $ Abs (_, _, t) $ arg) =
(case split-pat beta-term-pat 1 t of
SOME (i, f) => SOME (metaeq ctxt s (subst arg 0 i f))
| NONE => NONE)
| beta-proc - - - = NONE;
fun eta-proc ctxt (s as Const (const-name case-prod, _) $ Abs (_, _, t)) =
(case split-pat eta-term-pat 1 t of
SOME (_, ft) => SOME (metaeq ctxt s (let val f $ _ = ft in f end))
| NONE => NONE)
| eta-proc - - - = NONE;
end;
}
simproc-setup case-prod-beta (case-prod f z) =
(fn - => fn ctxt => fn ct => beta-proc ctxt (Thm.term-of ct))
simproc-setup case-prod-eta (case-prod f) =
(fn - => fn ctxt => fn ct => eta-proc ctxt (Thm.term-of ct));

lemma case-prod-beta': (λ(x,y). f x y) = (λx. f (fst x) (snd x))
by (auto simp: fun-eq-iff)

case-prod used as a logical connective or set former.

These rules are for use with blast; could instead call simp using prod.split
as rewrite.

**lemma case-prodI2:**
\[ \forall p. (\forall a b. p = (a, b) \Rightarrow c a b) \Rightarrow \text{case } p \text{ of } (a, b) \Rightarrow c a b \]
by (simp add: split-tupled-all)

**lemma case-prodI2':**
\[ \forall p. (\forall a b. (a, b) = p \Rightarrow c a b x) \Rightarrow (\text{case } p \text{ of } (a, b) \Rightarrow c a b x) \]
by (simp add: split-tupled-all)

**lemma case-prodE [elim!]:**
\[ (\text{case } p \text{ of } (a, b) \Rightarrow c a b) \Rightarrow (\forall x y. p = (x, y) \Rightarrow c x y \Rightarrow Q) \Rightarrow Q \]
by (induct p) simp

**lemma case-prodE' [elim!]:**
\[ (\text{case } p \text{ of } (a, b) \Rightarrow c a b) \Rightarrow (\forall x y. z = (x, y) \Rightarrow c x y z \Rightarrow Q) \Rightarrow Q \]
by (induct p) simp

**lemma case-prodE2:**
assumes \( q: Q \text{ (case } z \text{ of } (a, b) \Rightarrow P a b) \)
and \( r: \forall x y. z = (x, y) \Rightarrow Q \text{ (P } x y) \Rightarrow R \)
shows \( R \)
proof (rule r)
show \( z = (\text{fst } z, \text{snd } z) \) by simp
then show \( Q \text{ (P } (\text{fst } z) (\text{snd } z)) \)
using \( q \) by (simp add: case-prod-unfold)
qed

**lemma case-prodD':** (case \( a, b \) of \( c, d \) \Rightarrow R c d) \Rightarrow R a b c
by simp

**lemma mem-case-prodI:** \( z \in c a b \Rightarrow z \in \text{case } (a, b) \text{ of } (d, e) \Rightarrow c d e \)
by simp

**lemma mem-case-prodI2 [intro!]:**
\[ \forall p. (\forall a b. p = (a, b) \Rightarrow z \in c a b) \Rightarrow z \in (\text{case } p \text{ of } (a, b) \Rightarrow c a b) \]
by (simp only: split-tupled-all) simp

**declare mem-case-prodI [intro!] — postponed to maintain traditional declaration order!**

**declare case-prodI2' [intro!] — postponed to maintain traditional declaration order!**

**declare case-prodI2 [intro!] — postponed to maintain traditional declaration order!**

**declare case-prodI [intro!] — postponed to maintain traditional declaration order!**

**lemma mem-case-prodE [elim!]:**
assumes \( z \in \text{case-prod } c p \)
obtains \( x y \) where \( p = (x, y) \) and \( z \in c x y \)
using \( \text{assms} \) by (rule case-prodE2)

ML <
fun exists-p-split (Const (const-name `case-prod`,-) $ -$ (Const (const-name `Pair`,-) $ -$)) = true
  | exists-p-split (t $ u) = exists-p-split t orelse exists-p-split u
  | exists-p-split (Abs (_, _, t)) = exists-p-split t
  | exists-p-split _ = false;

in

fun split-conv-tac ctxt = SUBGOAL (fn (t, i) =>
  if exists-p-split t
  then safe-full-simp-tac (put-simpset HOL-basic-ss ctxt addsimps @{thms case-prod-conv})
  else no-tac);

end;

setup (map-theory-claset (fn ctxt => ctxt addSbefore (split-conv-tac, split-conv-tac)))

lemma split-eta-SetCompr [simp, no-atp]: (λ u. ∃ x y. u = (x, y) ∧ P (x, y)) = P by (rule ext) fast

lemma split-eta-SetCompr2 [simp, no-atp]: (λ u. ∃ x y. u = (x, y) ∧ P x y) = case-prod P by (rule ext) fast

lemma split-part [simp]: (λ (a, b). P ∧ Q a b) = (λ ab. P ∧ case-prod Q ab)
  — Allows simplifications of nested splits in case of independent predicates.
  by (rule ext) blast

lemma split-comp-eq:
  fixes f :: 'a ⇒ 'b ⇒ 'c
  and g :: 'd ⇒ 'a
  shows (λu. f (g (fst u)) (snd u)) = case-prod (λx. f (g x))
  by (rule ext) auto

lemma pair-imageI [intro]: (a, b) ∈ A ⇒ f a b ∈ (λ(a, b). f a b) ` A
  by (rule image-eqI [where x = (a, b)]) auto

lemma Collect-const-case-prod[simp]: {(a, b). P} = (if P then UNIV else { })
  by auto

lemma The-split-eq [simp]: (THE (x', y'). x = x' ∧ y = y') = (x, y)
  by blast

lemma case-prod-beta: case-prod f p = f (fst p) (snd p)
  by (fact prod.case-eq-if)
lemma prod-cases3 [cases type]:
  obtains (fields) a b c where y = (a, b, c)
  by (cases y, case-tac b) blast

lemma prod-induct3 [case-names fields, induct type]:
  (\(\forall a b c. \, P (a, b, c)\)) \implies P x
  by (cases x) blast

lemma prod-cases4 [cases type]:
  obtains (fields) a b c d where y = (a, b, c)
  by (cases y, case-tac c) blast

lemma prod-induct4 [case-names fields, induct type]:
  (\(\forall a b c d. \, P (a, b, c, d)\)) \implies P x
  by (cases x) blast

lemma prod-cases5 [cases type]:
  obtains (fields) a b c d e where y = (a, b, c, d)
  by (cases y, case-tac d) blast

lemma prod-induct5 [case-names fields, induct type]:
  (\(\forall a b c d e. \, P (a, b, c, d, e)\)) \implies P x
  by (cases x) blast

lemma prod-cases6 [cases type]:
  obtains (fields) a b c d e f where y = (a, b, c, d, e)
  by (cases y, case-tac e) blast

lemma prod-induct6 [case-names fields, induct type]:
  (\(\forall a b c d e f. \, P (a, b, c, d, e, f)\)) \implies P x
  by (cases x) blast

lemma prod-cases7 [cases type]:
  obtains (fields) a b c d e f g where y = (a, b, c, d, e, f)
  by (cases y, case-tac f) blast

lemma prod-induct7 [case-names fields, induct type]:
  (\(\forall a b c d e f g. \, P (a, b, c, d, e, f, g)\)) \implies P x
  by (cases x) blast

definition internal-case-prod :: ('a ⇒ 'b ⇒ 'c) ⇒ 'a × 'b ⇒ 'c
  where internal-case-prod ≡ case-prod

lemma internal-case-prod-conv: internal-case-prod c (a, b) = c a b
  by (simp only: internal-case-prod-def case-prod-conv)

ML-file ⟨Tools/split-rule.ML⟩
THEORY “Product-Type”

hide-const internal-case-prod

12.3.5 Derived operations

definition curry :: ('a × 'b ⇒ 'c) ⇒ 'a ⇒ 'b ⇒ 'c
where curry = (λc x y. c (x, y))

lemma curry-conv [simp, code]: curry f a b = f (a, b)
by (simp add: curry-def)

lemma curryI [intro!]: f (a, b) ⇒ curry f a b
by (simp add: curry-def)

lemma curryD [dest!]: curry f a b ⇒ f (a, b)
by (simp add: curry-def)

lemma curryE: curry f a b ⇒ (f (a, b) ⇒ Q) ⇒ Q
by (simp add: curry-def)

lemma case-prod-curry [simp]: case-prod (curry f) = f
by (simp add: curry-def case-prod-unfold)

lemma Pair-scomp: Pair x o→ f = f x
by (simp add: fun-eq-iff)

lemma scomp-Pair: x o→ Pair = x
by (simp add: fun-eq-iff)

lemma scomp-scomp: (f o→ g) o→ h = f o→ (λx. g x o→ h)
by (simp add: fun-eq-iff scomp-unfold)

The composition-uncurry combinator.

notation fcomp (infixl o→ 60)

definition scomp :: ('a ⇒ 'b × 'c) ⇒ (′b ⇒ 'c ⇒ 'd) ⇒ 'a ⇒ 'd (infixl o→ 60)
where f o→ g = (λx. case-prod g (f x))

lemma scomp-unfold: scomp = (λf g x. g (fst (f x))) (snd (f x)))
by (simp add: fun-eq-iff scomp-def case-prod-unfold)

lemma scomp-apply [simp]: (f o→ g) x = case-prod g (f x)
by (simp add: scomp-unfold case-prod-unfold)

lemma Pair-scomp: Pair x o→ f = f x
by (simp add: fun-eq-iff)

lemma scomp-Pair: x o→ Pair = x
by (simp add: fun-eq-iff)

lemma scomp-scomp: (f o→ g) o→ h = f o→ (λx. g x o→ h)
by (simp add: fun-eq-iff scomp-unfold)
**THEORY “Product-Type”**

```plaintext
lemma sccomp-fcomp: (f ◦→ g) ◦→ h = f ◦→ (λx. g x ◦→ h)
by (simp add: fun-eq-iff sccomp-unfold fcomp-def)

lemma fcomp-sccomp: (f ◦→ g) ◦→ h = f ◦→ (g ◦→ h)
by (simp add: fun-eq-iff sccomp-unfold)

code-printing
constant sccomp (infixl 3 #→)
no-notation fcomp (infixl ◦→ 60)
no-notation sccomp (infixl ◦→ 60)

map-prod — action of the product functor upon functions.

definition map-prod :: (('a ⇒ 'c) ⇒ ('b ⇒ 'd) ⇒ 'a × 'b ⇒ 'c × 'd)
where map-prod f g = (λ(x, y). (f x, g y))

lemma map-prod-simp [simp, code]: map-prod f g (a, b) = (f a, g b)
by (simp add: map-prod-def)

functor map-prod: map-prod
by (auto simp add: split-paired-all)

lemma fst-map-prod [simp]: fst (map-prod f g x) = f (fst x)
by (cases x) simp-all

lemma snd-map-prod [simp]: snd (map-prod f g x) = g (snd x)
by (cases x) simp-all

lemma fst-comp-map-prod [simp]: fst ◦ map-prod f g = f ◦ fst
by (rule ext) simp-all

lemma snd-comp-map-prod [simp]: snd ◦ map-prod f g = g ◦ snd
by (rule ext) simp-all

lemma map-prod-compose: map-prod (f1 ◦ f2) (g1 ◦ g2) = (map-prod f1 g1 ◦ map-prod f2 g2)
by (rule ext) (simp add: map-prod.compositionality comp-def)

lemma map-prod-ident [simp]: map-prod (λx. x) (λy. y) = (λz. z)
by (rule ext) (simp add: map-prod.identity)

lemma map-prod-image1 [intro]: (a, b) ∈ R → (f a, g b) ∈ map-prod f g ◦ R
by (rule image-eqI) simp-all

lemma prod-fun-imageE [elim!]:
assumes major: c ∈ map-prod f g ◦ R
   and cases: (∀x y. c = (f x, g y) ⇒ (x, y) ∈ R ⇒ P)
shows P
apply (rule major [THEN imageE])
```

apply (case-tac x)
apply (rule cases)
  apply simp-all
done

definition apfst :: ('a ⇒ 'c) ⇒ 'a × 'b ⇒ 'c × 'b
  where apfst f = map-prod f id

definition apsnd :: ('b ⇒ 'c) ⇒ 'a × 'b ⇒ 'a × 'c
  where apsnd f = map-prod id f

lemma apfst-conv [simp, code]: apfst f (x, y) = (f x, y)
  by (simp add: apfst-def)

lemma apsnd-conv [simp, code]: apsnd f (x, y) = (x, f y)
  by (simp add: apsnd-def)

lemma fst-apfst [simp]: fst (apfst f x) = f (fst x)
  by (cases x) simp

lemma fst-comp-apfst [simp]: fst ∘ apfst f = f ∘ fst
  by (simp add: fun-eq-iff)

lemma fst-apsnd [simp]: fst (apsnd f x) = fst x
  by (cases x) simp

lemma fst-comp-apsnd [simp]: fst ∘ apsnd f = fst
  by (simp add: fun-eq-iff)

lemma snd-apfst [simp]: snd (apfst f x) = snd x
  by (cases x) simp

lemma snd-comp-apfst [simp]: snd ∘ apfst f = snd
  by (simp add: fun-eq-iff)

lemma snd-apsnd [simp]: snd (apsnd f x) = f (snd x)
  by (cases x) simp

lemma snd-comp-apsnd [simp]: snd ∘ apsnd f = f ∘ snd
  by (simp add: fun-eq-iff)

lemma apfst-compose: apfst f (apfst g x) = apfst (f ∘ g) x
  by (cases x) simp

lemma apsnd-compose: apsnd f (apsnd g x) = apsnd (f ∘ g) x
  by (cases x) simp

lemma apfst-apsnd [simp]: apfst f (apsnd g x) = (f (fst x), g (snd x))
  by (cases x) simp
lemma apsnd-apfst [simp]: apsnd f (apfst g x) = (g (fst x), f (snd x))
  by (cases x) simp

lemma apfst-id [simp]: apfst id = id
  by (simp add: fun-eq-iff)

lemma apsnd-id [simp]: apsnd id = id
  by (simp add: fun-eq-iff)

lemma apfst-eq-conv [simp]: apfst f x = apfst g x ←→ f (fst x) = g (fst x)
  by (cases x) simp

lemma apsnd-eq-conv [simp]: apsnd f x = apsnd g x ←→ f (snd x) = g (snd x)
  by (cases x) simp

lemma apsnd-apfst-commute: apsnd f (apfst g p) = apfst g (apsnd f p)
  by simp

context begin

local-setup ⟨Local-Theory.map-background-naming (Name-Space.mandatory-path prod)⟩

definition swap :: 'a × 'b ⇒ 'b × 'a
  where swap p = (snd p, fst p)

end

lemma swap-simp [simp]: prod.swap (x, y) = (y, x)
  by (simp add: prod.swap-def)

lemma swap-swap [simp]: prod.swap (prod.swap p) = p
  by (cases p) simp

lemma swap-comp-swap [simp]: prod.swap ∘ prod.swap = id
  by (simp add: fun-eq-iff)

lemma pair-in-swap-image [simp]: (y, x) ∈ prod.swap 'A ←→ (x, y) ∈ A
  by (auto intro!: image-eqI)

lemma inj-swap [simp]: inj-on prod.swap A
  by (rule inj-onI) auto

lemma swap-inj-on: inj-on (λ(i, j). (j, i)) A
  by (rule inj-onI) auto

lemma surj-swap [simp]: surj prod.swap
THEORY “Product-Type”

by (rule surjI [of - prod.swap]) simp

lemma bij-swap [simp]: bij prod.swap
  by (simp add: bij-def)

lemma case-swap [simp]: (case prod.swap p of (y, x) ⇒ f x y) = (case p of (x, y) ⇒ f x y)
  by (cases p) simp

lemma fst-swap [simp]: fst (prod.swap x) = snd x
  by (cases x) simp

lemma snd-swap [simp]: snd (prod.swap x) = fst x
  by (cases x) simp

Disjoint union of a family of sets – Sigma.

definition Sigma :: `'a set ⇒ ('a ⇒ 'b set) ⇒ ('a × 'b) set
  where Sigma A B ≡ ⋃ x ∈ A. ⋃ y ∈ B x. {Pair x y}

abbreviation Times :: `'a set ⇒ 'b set ⇒ ('a × 'b) set (infixr × 80)
  where A × B ≡ Sigma A (λ- B)

hide-const (open) Times

bundle no-Set-Product-syntax begin
  no-notation Product-Type.Times (infixr × 80)
end

bundle Set-Product-syntax begin
  notation Product-Type.Times (infixr × 80)
end

syntax
  -Sigma :: pttrn ⇒ 'a set ⇒ 'b set ⇒ ('a × 'b) set ((3SIGMA -:- / -) [0, 0, 10] 10)

translations
  SIGMA x:A. B ⇔ CONST Sigma A (λx. B)

lemma SigmaI [intro!]: a ∈ A ⇒ b ∈ B a ⇒ (a, b) ∈ Sigma A B
  unfolding Sigma-def by blast

lemma SigmaE [elim!]: c ∈ Sigma A B ⇒ (∀x y. x ∈ A ⇒ y ∈ B x ⇒ c = (x, y) ⇒ P) ⇒ P
  — The general elimination rule.
  unfolding Sigma-def by blast

Elimination of (a, b) ∈ A × B – introduces no eigenvariables.

lemma SigmaD1: (a, b) ∈ Sigma A B ⇒ a ∈ A
  by blast
lemma $\text{SigmaD2}$: \((a, b) \in \Sigma A B \Rightarrow b \in B a\)
   by blast

lemma $\text{SigmaE2}$: \((a, b) \in \Sigma A B \Rightarrow (a \in A \Rightarrow b \in B a \Rightarrow P) \Rightarrow P\)
   by blast

lemma $\text{Sigma-cong}$: \(A = B = \Rightarrow (\forall x. x \in B \Rightarrow C x = D x) = (\Sigma x: A. C x) = (\Sigma x: B. D x)\)
   by auto

lemma $\text{Sigma-mono}$: \(A \subseteq C = \Rightarrow (\forall x. x \in A \Rightarrow B x \subseteq D x) = \Sigma A B \subseteq \Sigma C D\)
   by blast

lemma $\text{Sigma-empty1}$ [simp]: \(\Sigma \emptyset B = \emptyset\)
   by blast

lemma $\text{Sigma-empty2}$ [simp]: \(A \times \emptyset = \emptyset\)
   by blast

lemma $\text{UNIV-Times-UNIV}$ [simp]: \(\text{UNIV} \times \text{UNIV} = \text{UNIV}\)
   by auto

lemma $\text{Compl-Times-UNIV1}$ [simp]: \(- (\text{UNIV} \times A) = \text{UNIV} \times (-A)\)
   by auto

lemma $\text{Compl-Times-UNIV2}$ [simp]: \(- (A \times \text{UNIV}) = (-A) \times \text{UNIV}\)
   by auto

lemma $\text{mem-Sigma-iff}$ [iff]: \((a, b) \in \Sigma A B = \leftrightarrow a \in A \land b \in B a\)
   by blast

lemma $\text{mem-Times-iff}$: \(x \in A \times B = \leftrightarrow \text{fst } x \in A \land \text{snd } x \in B\)
   by (induct x) simp

lemma $\text{Sigma-empty-iff}$: \((\Sigma i: I. X i) = \emptyset = \leftrightarrow (\forall i : I. X i = \emptyset)\)
   by auto

lemma $\text{Times-subset-cancel2}$: \(x \in C = \Rightarrow A \times C \subseteq B \times C = \leftrightarrow A \subseteq B\)
   by blast

lemma $\text{Times-eq-cancel2}$: \(x \in C = \Rightarrow A \times C = B \times C = \leftrightarrow A = B\)
   by (blast elim: equalityE)

lemma $\text{Collect-case-prod-Sigma}$: \((x, y). P x \land Q x y) = (\Sigma x: \text{Collect } P. \text{Collect } (Q x))\)
   by blast

lemma $\text{Collect-case-prod}$ [simp]: \((a, b). P a \land Q b) = \text{Collect } P \times \text{Collect } Q\)
LEMMA Collect-case-prodD: \( x \in \text{Collect} \ (\text{case-prod A}) \implies A \ (\text{fst} \ x) \ (\text{snd} \ x) \)
by auto

LEMMA Collect-case-prod-mono: \( A \leq B \implies \text{Collect} \ (\text{case-prod A}) \subseteq \text{Collect} \ (\text{case-prod B}) \)
by auto (auto elim!: le-funE)

LEMMA Collect-split-mono-strong:
\[ X = \text{fst} \ \cdot \ A \implies Y = \text{snd} \ \cdot \ A \implies \forall a \in X. \ \forall b \in Y. \ P a b \implies Q a b \]
\[ \implies A \subseteq \text{Collect} \ (\text{case-prod P}) \implies A \subseteq \text{Collect} \ (\text{case-prod Q}) \]
by fastforce

LEMMA UN-Times-distrib:
\[ (\bigcup \ (a, \ b)\in A \times B. \ E a \times F b) = \bigcup (E' A) \times \bigcup (F' B) \]
— Suggested by Pierre Chartier
by blast

LEMMA split-paired-Ball-Sigma [simp, no-atp]: \( (\forall z \in \Sigma A B. \ P z) \iff (\forall x \in A. \ \forall y \in B. \ P (x, y)) \)
by blast

LEMMA split-paired-Bex-Sigma [simp, no-atp]: \( (\exists z \in \Sigma A B. \ P z) \iff (\exists x \in A. \ \exists y \in B. \ P (x, y)) \)
by blast

LEMMA Sigma-Un-distrib1: \( \Sigma (I \cup J) C = \Sigma I C \cup \Sigma J C \)
by blast

LEMMA Sigma-Un-distrib2: \( \Sigma i : I. \ A i \cup B i = \Sigma I A \cup \Sigma I B \)
by blast

LEMMA Sigma-Int-distrib1: \( \Sigma (I \cap J) C = \Sigma I C \cap \Sigma J C \)
by blast

LEMMA Sigma-Int-distrib2: \( \Sigma i : I. \ A i \cap B i = \Sigma I A \cap \Sigma I B \)
by blast

LEMMA Sigma-Diff-distrib1: \( \Sigma (I - J) C = \Sigma I C - \Sigma J C \)
by blast

LEMMA Sigma-Diff-distrib2: \( \Sigma i : I. \ A i - B i = \Sigma I A - \Sigma I B \)
by blast

LEMMA Sigma-Union: \( \Sigma (\bigcup X) B = (\bigcup A \in X. \ \Sigma A B) \)
by blast

LEMMA Pair-vimage-Sigma: \( \text{Pair} x \ -\ \cdot \ \Sigma A f = (\text{if} \ x \in A \ \text{then} \ f x \ \text{else} \ \{\}) \)
by auto

Non-dependent versions are needed to avoid the need for higher-order matching, especially when the rules are re-oriented.

lemma Times-Un-distrib1: \((A \cup B) \times C = A \times C \cup B \times C\)
by (fact Sigma-Un-distrib1)

lemma Times-Int-distrib1: \((A \cap B) \times C = A \times C \cap B \times C\)
by (fact Sigma-Int-distrib1)

lemma Times-Diff-distrib1: \((A - B) \times C = A \times C - B \times C\)
by (fact Sigma-Diff-distrib1)

lemma Times-empty [simp]: \(A \times B = \{\} \iff A = \{\} \lor B = \{\}\)
by auto

lemma times-subset-iff: \(A \times C \subseteq B \times D \iff A = \{\} \lor C = \{\} \lor A \subseteq B \land C \subseteq D\)
by blast

lemma times-eq-iff: \(A \times B = C \times D \iff A = C \land B = D \lor (A = \{\} \lor B = \{\}) \land (C = \{\} \lor D = \{\})\)
by auto

lemma fst-image-times [simp]: \(\text{fst } (A \times B) = (\text{if } B = \{\} \text{ then } \{\} \text{ else } A)\)
by force

lemma snd-image-times [simp]: \(\text{snd } (A \times B) = (\text{if } A = \{\} \text{ then } \{\} \text{ else } B)\)
by force

lemma fst-image-Sigma: \(\text{fst } (\Sigma A B) = \{x \in A. B(x) \neq \{\}\}\)
by force

lemma snd-image-Sigma: \(\text{snd } (\Sigma A B) = (\bigcup x \in A. B x)\)
by force

lemma vimage-fst: \(\text{fst }^{-1} A = A \times UNIV\)
by auto

lemma vimage-snd: \(\text{snd }^{-1} A = UNIV \times A\)
by auto

lemma insert-Times-insert [simp]:
insert a A \times insert b B = insert (a,b) (A \times insert b B \cup insert a A \times B)
by blast

lemma vimage-Times: \(f^{-1} (A \times B) = (\text{fst } \circ f) -^{-1} A \cap (\text{snd } \circ f) -^{-1} B\)
proof (rule set-eqI)
show \(x \in f^{-1} (A \times B) \iff x \in (\text{fst } \circ f) -^{-1} A \cap (\text{snd } \circ f) -^{-1} B\) for \(x\)
by (cases f x) (auto split: prod.split)
qed

lemma Times-Int-Times: \( A \times B \cap C \times D = (A \cap C) \times (B \cap D) \)
  by auto

lemma image-paired-Times: 
  \( \lambda(x,y). (f x, g y) \) \( \cdot \) \( (A \times B) \cap C \times D = (A \cap C) \times (B \cap D) \)
  by auto

lemma product-swap: \( \text{prod.swap} \cdot (A \times B) = B \times A \)
  by (auto simp add: set-eq-iff)

lemma swap-product: 
  \( \lambda(i,j). (j,i) \) \( \cdot \) \( (A \times B) \)
  = \( B \times A \)
  by (auto simp add: set-eq-iff)

lemma image-split-eq-Sigma: 
  \( \lambda(x). (f x, g x) \) \( \cdot \) \( A \)
  = \( \Sigma(f \cdot A) (\lambda x. (f \cdot (f - {x}) \cap A)) \)
proof (safe intro: imageI)
  fix a b
  assume *: \( a \in A \) \( b \in A \) and \( \text{eq} \cdot f a = f b \)
  show \( (f b, g a) \in (\lambda x. (f x, g x)) \cdot A \)
    using * eq[symmetric] by auto
qed simp-all

lemma subset-fst-snd: \( A \subseteq (\text{fst} \cdot A \times \text{snd} \cdot A) \)
  by force

lemma inj-on-apfst [simp]: \( \text{inj-on (apfst f)} (A \times \text{UNIV}) \leftrightarrow \text{inj-on} f A \)
  by (auto simp add: inj-on-def)

lemma inj-apfst [simp]: \( \text{inj (apfst f)} \leftrightarrow \text{inj} f \)
  using inj-on-apfst[of f UNIV] by simp

lemma inj-on-apsnd [simp]: \( \text{inj-on (apsnd f)} (\text{UNIV} \times A) \leftrightarrow \text{inj-on} f A \)
  by (auto simp add: inj-on-def)

lemma inj-apsnd [simp]: \( \text{inj (apsnd f)} \leftrightarrow \text{inj} f \)
  using inj-on-apsnd[of f UNIV] by simp

context
begin

qualified definition product :: 'a set \Rightarrow 'b set \Rightarrow ('a \times 'b) set
  where [code-abbrev]: product A B = A \times B

lemma member-product: \( x \in \text{Product-Type.product} A B \leftrightarrow x \in A \times B \)
  by (simp add: product-def)
The following map-prod lemmas are due to Joachim Breitner:

**Lemma map-prod-inj-on:**
- Assumes inj-on \( f \ A \)
- and inj-on \( g \ B \)
- Shows inj-on \((\text{map-prod } f \ g) (A \times B)\)

**Proof** (rule inj-onI)

1. Fix \( x :: 'a \times 'c \)
2. Fix \( y :: 'a \times 'c \)
3. Assume \( x \in A \times B \)
4. Then have \( \text{fst } x \in A \) and \( \text{snd } x \in B \) by auto
5. Assume \( y \in A \times B \)
6. Then have \( \text{fst } y \in A \) and \( \text{snd } y \in B \) by auto
7. Assume \( \text{map-prod } f \ g \ x = \text{map-prod } f \ g \ y \)
8. Then have \( \text{fst } (\text{map-prod } f \ g \ x) = \text{fst } (\text{map-prod } f \ g \ y) \) by auto
9. Then have \( f \ (\text{fst } x) = f \ (\text{fst } y) \) by (cases \( x \), cases \( y \)) auto

**Unfolding** inj-on \( A \) and \( \text{fst } x \in A \) and \( \text{fst } y \in A \) have \( \text{fst } x = \text{fst } y \)
   - by (auto dest: inj-onD)

10. Moreover from \((\text{map-prod } f \ g \ x = \text{map-prod } f \ g \ y)\)
    - Have \( \text{snd } (\text{map-prod } f \ g \ x) = \text{snd } (\text{map-prod } f \ g \ y) \) by auto
    - Then have \( g \ (\text{snd } x) = g \ (\text{snd } y) \) by (cases \( x \), cases \( y \)) auto

**Unfolding** inj-on \( B \) and \( \text{snd } x \in B \) and \( \text{snd } y \in B \) have \( \text{snd } x = \text{snd } y \)
   - by (auto dest: inj-onD)

11. Ultimately show \( x = y \) by (rule prod-eqI)

**QED**

**Lemma map-prod-surj:**

1. Fixes \( f :: 'a \Rightarrow 'b \)
2. and \( g :: 'c \Rightarrow 'd \)
3. Assumes surj \( f \) and surj \( g \)
4. Shows surj \( (\text{map-prod } f \ g) \)

**Unfolding** surj-def

**Proof**

1. Fix \( y :: 'b \times 'd \)
2. From (surj \( f \)) obtain \( a \) where \( \text{fst } y = f \ a \)
   - by (auto elim: surjE)
3. Moreover from (surj \( g \)) obtain \( b \) where \( \text{snd } y = g \ b \)
   - by (auto elim: surjE)
4. Ultimately have \( (\text{fst } y, \text{snd } y) = \text{map-prod } f \ g \ (a,b) \)
   - by auto
5. Then show \( \exists x. \ y = \text{map-prod } f \ g \ x \)
   - by auto

**QED**

**Lemma map-prod-surj-on:**

1. Assumes \( f \cdot A = A' \) and \( g \cdot B = B' \)
2. Shows \( \text{map-prod } f \ g \cdot (A \times B) = A' \times B' \)
unfolding image-def
proof (rule set-eqI, rule iffI)
  fix x :: 'a × 'c
  assume x ∈ {y::'a × 'c. ∃x::'b × 'd∈A × B. y = map-prod f g x}
  then obtain y where y ∈ A × B and x = map-prod f g y
    by blast
  from (image f A = A′ and (∃y ∈ A × B. have f (fst y) ∈ A′)
    by auto
  moreover from (image g B = B′ and (∃y ∈ A × B. have g (snd y) ∈ B′)
    by auto
  ultimately have (f (fst y), g (snd y)) ∈ (A′ × B′)
    by auto
  with (x = map-prod f g y) show x ∈ A′ × B′
    by (cases y) auto
next
  fix x :: 'a × 'c
  assume x ∈ A′ × B′
  then have fst x ∈ A′ and snd x ∈ B′
    by auto
  from (image f A = A′ and (∃x ∈ A′. have fst x ∈ image f A)
    by auto
  then obtain a where a ∈ A and fst x = f a
    by (rule imageE)
  moreover from (image g B = B′ and (∃snd x ∈ B′. obtain b where b ∈ B
    and snd x = g b
      by auto
  ultimately have (fst x, snd x) = map-prod f g (a, b)
    by auto
  moreover from (∃a ∈ A. have (a, b) ∈ A × B
    by auto
  ultimately have (∃y ∈ A × B. x = map-prod f g y)
    by auto
  then show x ∈ {x. ∃y ∈ A × B. x = map-prod f g y}
    by auto
qed

12.4 Simproc for rewriting a set comprehension into a point-free expression

ML-file ⟨Tools/set-comprehension-pointfree.ML⟩

setup:
  Code-Preproc.map-pre (fn ctxt => ctxt addsimprocs
  [Simplifier.make-simproc context set comprehension
   {lhss = [term (Collect P)],
      proc = K Set-Comprehension-Pointfree.code-simproc}])
12.5 Lemmas about disjointness

**lemma disjnt-Times1-iff [simp]:**

\[
\text{disjnt} (C \times A) (C \times B) \longleftrightarrow C = \{\} \lor \text{disjnt } A B
\]

by (auto simp: disjnt-def)

**lemma disjnt-Times2-iff [simp]:**

\[
\text{disjnt} (A \times C) (B \times C) \longleftrightarrow C = \{\} \lor \text{disjnt } A B
\]

by (auto simp: disjnt-def)

**lemma disjnt-Sigma-iff:**

\[
\text{disjnt} (\Sigma A C) (\Sigma B C) \longleftrightarrow (\forall i \in A \cap B. C i = \{\}) \lor \text{disjnt } A B
\]

by (auto simp: disjnt-def)

12.6 Inductively defined sets

**simproc-setup**

\[
\text{Collect-mem} (\text{Collect } t) = \langle fn - => fn ctxt => fn ct =>
\]

(case Thm.term-of ct of
  S as Const (\textbf{const-name}(:Collect), Type (\textbf{type-name}(:fun, [\_, T]))) $ t =>
  let val (u, _, ps) = HOLogic.strip_ptupleabs t in
  (case u of
    (c as Const (\textbf{const-name}(:Set.member), -)) $ q $ S' =>
      (case try (HOLogic.strip_ptuple ps) q of
        NONE => NONE |
        SOME ts =>
          if not (Term.is_open S') andalso
          ts = map Bound (length ps downto 0) then
            let val simp =
              full_simp_tac (put_simpset HOL_basic_ss ctxt
              addsimps [@{thm split_paired_all}, @{thm case_prod_conv}]) 1
            in
              SOME (Goal.prove ctxt [] []
              (Const (\textbf{const-name}(:Pure.eq), T => T => propT) $ S $ S'))
            end
          else NONE)
    | - => NONE)
  end |
  - => NONE)
end
\]}
ML-file ⟨Tools/inductive-set.ML⟩

12.7 Legacy theorem bindings and duplicates

lemmas \( \text{fst-conv} = \text{prod.sel}(1) \)
lemmas \( \text{snd-conv} = \text{prod.sel}(2) \)
lemmas \( \text{split-def} = \text{case-prod-unfold} \)
lemmas \( \text{split-beta}' = \text{case-prod-beta}' \)
lemmas \( \text{split-beta} = \text{prod.case-eq-if} \)
lemmas \( \text{split-conv} = \text{case-prod-conv} \)
lemmas \( \text{split} = \text{case-prod-conv} \)

hide-const (open) \( \text{prod} \)

end

13 The Disjoint Sum of Two Types

theory \( \text{Sum-Type} \)
  imports Typedef Inductive Fun
begin

13.1 Construction of the sum type and its basic abstract operations

definition \( \text{Inl-Rep} :: a \Rightarrow b \Rightarrow \text{bool} \Rightarrow \text{bool} \)
where \( \text{Inl-Rep} a x y p \leftarrow x = a \land p \)
definition \( \text{Inr-Rep} :: b \Rightarrow a \Rightarrow b \Rightarrow \text{bool} \Rightarrow \text{bool} \)
where \( \text{Inr-Rep} b x y p \leftarrow y = b \land \neg p \)
definition \( \text{sum} = \{ f. (\exists a. f = \text{Inl-Rep} (a::a)) \lor (\exists b. f = \text{Inr-Rep} (b::b)) \} \)
typedef \( (a, b) \text{ sum} \) (infixr + 10) = \( \text{sum} :: (a \Rightarrow b \Rightarrow \text{bool} \Rightarrow \text{bool}) \text{ set} \)
  unfolding \( \text{sum-def} \) by auto

lemma \( \text{Inl-RepI}: \text{Inl-Rep} a \in \text{sum} \)
  by (auto simp add: \( \text{sum-def} \))

lemma \( \text{Inr-RepI}: \text{Inr-Rep} b \in \text{sum} \)
  by (auto simp add: \( \text{sum-def} \))

lemma \( \text{inj-on-Abs-sum}: A \subseteq \text{sum} \Rightarrow \text{inj-on Abs-sum} A \)
  by (rule inj-on-inverseI, rule Abs-sum-inverse) auto

lemma \( \text{Inl-Rep-inject}: \text{inj-on Inl-Rep} A \)
proof (rule inj-onI)
  show \( \forall a c. \text{Inl-Rep} a = \text{Inl-Rep} c \Rightarrow a = c \)
    by (auto simp add: \( \text{Inl-Rep-def fun-eq-if} \))
THEORY “Sum-Type”

**lemma** *Inr-Rep-inject*: inj-on *Inr-Rep* *A*
**proof** (rule *inj-onI*)
  - show \( \forall b\ d.\ \text{Inr-Rep}\ b = \text{Inr-Rep}\ d \implies b = d \)
    - by (auto simp add: *Inr-Rep-def* fun-eq-iff)
**qed**

**lemma** *Inl-Rep-not-Inr-Rep*: *Inl-Rep* *a* \(\neq\) *Inr-Rep* *b*
  - by (auto simp add: *Inl-Rep-def* *Inr-Rep-def* fun-eq-iff)

**definition** *Inl* :: ‘*a* \(\Rightarrow\) ‘*a* + ‘*b*
  where *Inl* = *Abs-sum* \(\circ\) *Inl-Rep*

**definition** *Inr* :: ‘*b* \(\Rightarrow\) ‘*a* + ‘*b*
  where *Inr* = *Abs-sum* \(\circ\) *Inr-Rep*

**lemma** *inj-Inl* [simp]: inj-on *Inl* *A*
  - by (auto simp add: *Inl-def* intro: *comp-inj-on* *Inl-Rep-inject* *inj-on-Abs-sum* *Inl-RepI*)

**lemma** *Inl-inject*: *Inl* *x* = *Inl* *y* \(\Rightarrow\) *x* = *y*
  - using *inj-Inl* by (rule *injD*)

**lemma** *inj-Inr* [simp]: inj-on *Inr* *A*
  - by (auto simp add: *Inr-def* intro: *comp-inj-on* *Inr-Rep-inject* *inj-on-Abs-sum* *Inr-RepI*)

**lemma** *Inr-inject*: *Inr* *x* = *Inr* *y* \(\Rightarrow\) *x* = *y*
  - using *inj-Inr* by (rule *injD*)

**lemma** *Inl-not-Inr*: *Inl* *a* \(\neq\) *Inr* *b*
**proof** –
  - have \(\{\text{Inl-Rep}\ a,\ \text{Inr-Rep}\ b\}\ \subseteq\ \text{sum}\)
    - using *Inl-RepI* [of *a*] *Inr-RepI* [of *b*] by auto
  - with *inj-on-Abs-sum* have *inj-on* *Abs-sum* \(\{\text{Inl-Rep}\ a,\ \text{Inr-Rep}\ b\}\).  
  - with *Inl-Rep-not-Inr-Rep* [of *a* *b*] *inj-on-contraD* have *Abs-sum* (*Inl-Rep* *a*) \(\neq\) *Abs-sum* (*Inr-Rep* *b*)
    - by auto
  - then show \(\?\)thesis
    - by (simp add: *Inl-def* *Inr-def*)
**qed**

**lemma** *Inr-not-Inl*: *Inr* *b* \(\neq\) *Inl* *a*
  - using *Inl-Rep-not-Inr* by (rule *not-sym*)

**lemma** *sumE*:
  - assumes \(\forall x::\text{'}a.\ \ s = \text{Inl}\ x \Rightarrow\ P\)
  - and \(\forall y::\text{'}b.\ \ s = \text{Inr}\ y \Rightarrow\ P\)
THEORY "Sum-Type"

shows P
proof (rule Abs-sum-cases [of s])
  fix f
  assume s = Abs-sum f and f ∈ sum
  with assms show P
    by (auto simp add: sum-def Inl-def Inr-def)
qed

free-constructors case-sum for
isl: Inl projl
| Inr projr
by (erule sumE, assumption) (auto dest: Inl-inject Inr-inject simp add: Inl-not-Inr)

Avoid name clashes by prefixing the output of old-rep-datatype with old.

setup ⟨Sign.mandatory-path old⟩

old-rep-datatype Inl Inr
proof –
  fix P
  fix s :: 'a + 'b
  assume x: ∀x::'a. P (Inl x) and y: ∀y::'b. P (Inr y)
  then show P s by (auto intro: sumE [of s])
qed (auto dest: Inl-inject Inr-inject simp add: Inl-not-Inr)

setup ⟨Sign.parent-path⟩

But erase the prefix for properties that are not generated by free-constructors.

setup ⟨Sign.mandatory-path sum⟩

declare
  old.sum.inject[iff del]
  old.sum.distinct(1)[simp del, induct-simp del]

lemmas induct = old.sum.induct
lemmas inducts = old.sum.inducts
lemmas rec = old.sum.rec
lemmas simps = sum.inject sum.distinct sum.case sum.rec

setup ⟨Sign.parent-path⟩

primrec map-sum :: ('a ⇒ 'c) ⇒ ('b ⇒ 'd) ⇒ 'a + 'b ⇒ 'c + 'd
  where
    map-sum f1 f2 (Inl a) = Inl (f1 a)
| map-sum f1 f2 (Inr a) = Inr (f2 a)

functor map-sum: map-sum
proof –
  show map-sum f g ∘ map-sum h i = map-sum (f ∘ h) (g ∘ i) for f g h i
proof
show (map-sum f g ∘ map-sum h i) s = map-sum (f ∘ h) (g ∘ i) s for s
by (cases s) simp-all
qed
show map-sum id id = id
proof
  show map-sum id id s = id s for s
  by (cases s) simp-all
qed
qed

lemma split-sum-all: (∀x. P x) ↔ (∀x. P (Inl x)) ∧ (∀x. P (Inr x))
by (auto intro: sum.induct)

lemma split-sum-ex: (∃x. P x) ↔ (∃x. P (Inl x)) ∨ (∃x. P (Inr x))
using split-sum-all[of λx. ¬P x] by blast

13.2 Projections

lemma case-sum-KK [simp]: case-sum (λx. a) (λx. a) = (λx. a)
by (rule ext) (simp split: sum.split)

lemma surjective-sum: case-sum (λx::'a. f (Inl x)) (λy::'b. f (Inr y)) = f
proof
  fix s :: 'a + 'b
  show (case s of Inl x::'a ⇒ f (Inl x) | Inr y::'b ⇒ f (Inr y)) = f
  by (cases s) simp-all
qed

lemma case-sum-inject:
  assumes a: case-sum f1 f2 = case-sum g1 g2
  and r: f1 = g1 → f2 = g2 → P
shows P
proof (rule r)
  show f1 = g1
  proof
    fix x :: 'a
    from a have case-sum f1 f2 (Inl x) = case-sum g1 g2 (Inl x) by simp
    then show f1 x = g1 x by simp
  qed

  show f2 = g2
  proof
    fix y :: 'b
    from a have case-sum f1 f2 (Inr y) = case-sum g1 g2 (Inr y) by simp
    then show f2 y = g2 y by simp
  qed

primrec Suml :: ('a ⇒ 'c) ⇒ 'a + 'b ⇒ 'c
where Suml f (Inl x) = f x
primrec Sumr :: (′b ⇒ ′c) ⇒ ′a + ′b ⇒ ′c
  where Sumr f (Inr x) = f x

lemma Suml-inject:
  assumes Suml f = Suml g
  shows f = g
proof
  fix x :: ′a
  let ?s = Inl x :: ′a + ′b
  from assms have Suml f ?s = Suml g ?s by simp
  then show f x = g x by simp
qed

lemma Sumr-inject:
  assumes Sumr f = Sumr g
  shows f = g
proof
  fix x :: ′b
  let ?s = Inr x :: ′a + ′b
  from assms have Sumr f ?s = Sumr g ?s by simp
  then show f x = g x by simp
qed

13.3 The Disjoint Sum of Sets

definition Plus :: ′a set ⇒ ′b set ⇒ (′a + ′b) set (infixr <+>)
  where A <+> B = Inl 'A ∪ Inr 'B

hide-const (open) Plus — Valuable identifier

lemma InlI [intro!]: a ∈ A ⇒ Inl a ∈ A <+> B
by (simp add: Plus-def)

lemma InrI [intro!]: b ∈ B ⇒ Inr b ∈ A <+> B
by (simp add: Plus-def)

Exhaustion rule for sums, a degenerate form of induction

lemma PlusE [elim!]:
  u ∈ A <+> B ⇒ (∀x. x ∈ A ⇒ u = Inl x ⇒ P) ⇒ (∀y. y ∈ B ⇒ u = Inr y ⇒ P) ⇒ P
by (auto simp add: Plus-def)

lemma Plus-eq-empty-conv [simp]: A <+> B = {} ⇐⇒ A = {} ∧ B = {}
by auto

lemma UNIV-Plus-UNIV [simp]: UNIV <+> UNIV = UNIV
proof (rule set-eqI)
  fix u :: ′a + ′b
show \( u \in \text{UNIV} \leftrightarrow u \in \text{UNIV} \) by (cases \( u \)) auto
qed

lemma \( \text{UNIV-sum} \): \( \text{UNIV} = \text{Inl} \cdot \text{UNIV} \cup \text{Inr} \cdot \text{UNIV} \)
proof 
  have \( x \in \text{range Inl} \) if \( x \notin \text{range Inr} \) for \( x :: 'a + 'b \)
  using that by (cases \( x \)) simp-all
  then show ?thesis by auto
qed

hide-const (open) \( \text{Suml} \, \text{Sumr} \, \text{sum} \)
end

14 Rings

theory Rings
  imports Groups Set Fun
begin

14.1 Semirings and rings

class \( \text{semiring} = \) \( \text{ab-semigroup-add} + \text{semigroup-mult} + \)
  assumes distrib-right [\( \text{algebra-simps, algebra-split-simps} \): \( (a + b) \cdot c = a \cdot c + b \cdot c \)]
  assumes distrib-left [\( \text{algebra-simps, algebra-split-simps} \): \( a \cdot (b + c) = a \cdot b + a \cdot c \)]
begin

For the \( \text{combine-numerals} \) \( \text{simproc} \)

lemma \( \text{combine-common-factor} \): \( a \cdot e + (b \cdot e + c) = (a + b) \cdot e + c \)
  by (simp add: distrib-right \( \text{ac-simps} \))

end

class \( \text{mult-zero} = \text{times} + \text{zero} + \)
  assumes mult-zero-left [simp]: \( 0 \cdot a = 0 \)
  assumes mult-zero-right [simp]: \( a \cdot 0 = 0 \)
begin

lemma mult-not-zero: \( a \cdot b \neq 0 \implies a \neq 0 \land b \neq 0 \)
  by auto

end

class \( \text{semiring-0} = \text{semiring} + \text{comm-monoid-add} + \text{mult-zero} \)
class \( \text{semiring-0-cancel} = \text{semiring} + \text{cancel-comm-monoid-add} \)
begin
subclass semiring-0
proof
  fix a :: 'a
  have 0 * a + 0 * a = 0 * a + 0
    by (simp add: distrib-right [symmetric])
  then show 0 * a = 0
    by (simp only: add-left-cancel)
  have a * 0 + a * 0 = a * 0 + 0
    by (simp add: distrib-left [symmetric])
  then show a * 0 = 0
    by (simp only: add-left-cancel)
qed

end

class comm-semiring = ab-semigroup-add + ab-semigroup-mult +
  assumes distrib: (a + b) * c = a * c + b * c
begin
subclass semiring
proof
  fix a b c :: 'a
  show (a + b) * c = a * c + b * c
    by (simp add: distrib)
  have a * (b + c) = (b + c) * a
    by (simp add: ac-simps)
  also have ... = b * a + c * a
    by (simp only: distrib)
  also have ... = a * b + a * c
    by (simp add: ac-simps)
  finally show a * (b + c) = a * b + a * c
    by blast
qed

end

class comm-semiring-0 = comm-semiring + comm-monoid-add + mult-zero
begin
subclass semiring-0 ..

end

class comm-semiring-0-cancel = comm-semiring + cancel-comm-monoid-add
begin
subclass semiring-0-cancel ..
subclass comm-semiring-0 ..
end

class zero-neq-one = zero + one +
  assumes zero-neq-one [simp]: 0 ≠ 1
begin

lemma one-neq-zero [simp]: 1 ≠ 0
  by (rule not-sym) (rule zero-neq-one)

definition of-bool :: bool ⇒ ′a
  where of-bool p = (if p then 1 else 0)

lemma of-bool-eq [simp, code]:
  of-bool False = 0
  of-bool True = 1
  by (simp-all add: of-bool-def)

lemma of-bool-eq-iff: of-bool p = of-bool q ←→ p = q
  by (simp add: of-bool-def)

lemma split-of-bool [split]: P (of-bool p) ←→ (p → P 1) ∧ (¬ p → P 0)
  by (cases p) simp-all

lemma split-of-bool-asm: P (of-bool p) ←→ ( p ∧ P 1 ∨ ¬ p ∧ ¬ P 0)
  by (cases p) simp-all
end

class semiring-1 = zero-neq-one + semiring-0 + monoid-mult
begin

lemma of-bool-conj:
  of-bool (P ∧ Q) = of-bool P ∗ of-bool Q
  by auto
end

lemma lambda-zero: (λx::′a::mult-zero. 0) = (∗) 0
  by auto

lemma lambda-one: (λx::′a::monoid-mult. x) = (∗) 1
  by auto

14.2 Abstract divisibility

class dvd = times
begin
**THEORY “Rings”**

**definition dvd :: ’a ⇒ ’a ⇒ bool** (infix dvd 50)

where 

\[ b \text{ dvd } a ⇔ (\exists k. a = b * k) \]

**lemma dvdI [intro?]:** \( a = b * k \Rightarrow b \text{ dvd } a \)

unfolding dvd-def ..

**lemma dvdE [elim]:** \( b \text{ dvd } a \Rightarrow (\forall k. a = b * k \Rightarrow P) \Rightarrow P \)

unfolding dvd-def by blast

end

class dvd :

**lemma dvd-refl [simp]:** \( a \text{ dvd } a \)

proof

show \( a = a * 1 \) by simp

qed

**lemma dvd-trans [trans]:**

assumes \( a \text{ dvd } b \) and \( b \text{ dvd } c \)

shows \( a \text{ dvd } c \)

proof –

from assms obtain \( v \) where \( b = a * v \)

by auto

moreover from assms obtain \( w \) where \( c = b * w \)

by auto

ultimately have \( c = a * (v * w) \)

by (simp add: mult.assoc)

then show \( \text{thesis} \) ..

qed

**lemma subset-divisors-dvd: \{ c. c dvd a \} ⊆ \{ c. c dvd b \} \iff \text{a dvd b}**

by (auto simp add: subset-iff intro: dvd-trans)

**lemma strict-subset-divisors-dvd: \{ c. c dvd a \} ⊂ \{ c. c dvd b \} \iff a dvd b ∧ ¬ b dvd a**

by (auto simp add: subset-iff intro: dvd-trans)

**lemma one-dvd [simp]:** \( 1 \text{ dvd } a \)

by (auto intro: dvdI)

**lemma dvd-mult [simp]:** \( a \text{ dvd } (b * c) \) if \( a \text{ dvd } c \)

using that by rule (auto intro: mult.left-commute dvdI)

**lemma dvd-mult2 [simp]:** \( a \text{ dvd } (b * c) \) if \( a \text{ dvd } b \)
using that dvd-mult [of a b c] by (simp add: ac-simps)

lemma dvd-triv-right [simp]: a dvd b * a  
by (rule dvd-mult) (rule dvd-refl)

lemma dvd-triv-left [simp]: a dvd a * b  
by (rule dvd-mult2) (rule dvd-refl)

lemma mult-dvd-mono:  
assumes a dvd b  
and c dvd d  
shows a * c dvd b * d  
proof –  
from ⟨a dvd b⟩ obtain b’ where b = a * b’ ..  
moreover from ⟨c dvd d⟩ obtain d’ where d = c * d’ ..  
ultimately have b * d = (a * c) * (b’ * d’)  
by (simp add: ac-simps)  
then show ?thesis ..  
qed

lemma dvd-mult-left: a * b dvd c =⇒ a dvd c  
by (simp add: dvd-def mult.assoc) blast

lemma dvd-mult-right: a * b dvd c =⇒ b dvd c  
using dvd-mult-left [of b a c] by (simp add: ac-simps)

end

class comm-semiring-1 = zero-neq-one + comm-semiring-0 + comm-monoid-mult
begin

subclass semiring-1 ..

lemma dvd-0-left-iiff [simp]: 0 dvd a ←→ a = 0  
by auto

lemma dvd-0-right [iff]: a dvd 0  
proof  
show 0 = a * 0 by simp  
qed

lemma dvd-0-left: 0 dvd a =⇒ a = 0  
by simp

lemma dvd-add [simp]:  
assumes a dvd b and a dvd c  
shows a dvd (b + c)  
proof –  
from ⟨a dvd b⟩ obtain b’ where b = a * b’ ..
moreover from \( \langle a \text{ dvd } c \rangle \) obtain \( c' \) where \( c = a \times c' \).
ultimately have \( b + c = a \times (b' + c') \)
by \( (\text{simp add: distrib-left}) \)
then show \(?thesis\).
qed

\textbf{class} semiring-1-cancel = semiring + cancel-comm-monoid-add
+ zero-neq-one + monoid-mult
\begin{end}
\textbf{subclass} semiring-0-cancel ..
\textbf{subclass} semiring-1 ..
\end

\textbf{class} comm-semiring-1-cancel =
comm-semiring + cancel-comm-monoid-add + zero-neq-one + comm-monoid-mult
+
\textbf{assumes} right-diff-distrib' [algebra-simps, algebra-split-simps]:
\( a \times (b - c) = a \times b - a \times c \)
\begin{begin}
\textbf{subclass} semiring-1-cancel ..
\textbf{subclass} comm-semiring-0-cancel ..
\textbf{subclass} comm-semiring-1 ..
\end

\textbf{lemma} left-diff-distrib' [algebra-simps, algebra-split-simps]:
\( (b - c) \times a = b \times a - c \times a \)
by \( (\text{simp add: algebra-simps}) \)

\textbf{lemma} dvd-add-times-triv-left-iff [simp]: \( a \text{ dvd } c \times a + b \iff a \text{ dvd } b \)
\begin{proof}
\begin{assume}\( ?Q \)
\begin{then show} \( ?P \) by \( \text{simp} \)
\end
\end
\begin{next}
\begin{assume} \( ?P \)
\begin{then obtain} \( d \) where \( a \times c + b = a \times d \).
\begin{then have} \( a \times c + b = a \times c + a \times d - a \times c \) by \( \text{simp} \)
\begin{then have} \( b = a \times d - a \times c \) by \( \text{simp} \)
\begin{then have} \( b = a \times (d - c) \) by \( (\text{simp add: algebra-simps}) \)
\begin{then show} \( ?Q \).
\end
\end
\end
\end
\end
\end
\end
\end
\end
\end
\end
\end
\begin{qed}
\begin{then show} \( a \text{ dvd } c \times a + b \iff a \text{ dvd } b \) by \( (\text{simp add: ac-simps}) \)
\end
\end
\begin{qed}
lemma dvd-add-times-triv-right-iff [simp]: a dvd b + c * a ←→ a dvd b
  using dvd-add-times-triv-left-iff [of a c b] by (simp add: ac-simps)

lemma dvd-add-triv-left-iff [simp]: a dvd a + b ←→ a dvd b
  using dvd-add-triv-right-iff [of a b 1] by simp

lemma dvd-add-triv-right-iff [simp]: a dvd b + a ←→ a dvd b
  using dvd-add-times-triv-right-iff [of a b 1] by simp

lemma dvd-add-right-iff:
  assumes a dvd b
  shows a dvd b + c ←→ a dvd c (is ?P ←→ ?Q)
proof
  assume ?P
  then obtain d where b + c = a * d ..
  moreover from ⟨a dvd b⟩ obtain e where b = a * e ..
  ultimately have a * e + c - a * e = a * d - a * e by simp
  then have c = a * (d - e) by (simp add: algebra-simps)
  then show ?Q ..
next
  assume ?Q
  with assms show ?P by simp
qed

lemma dvd-add-left-iff: a dvd c =⇒ a dvd b + c ←→ a dvd b
  using dvd-add-right-iff [of a c b] by (simp add: ac-simps)

end

class ring = semiring + ab-group-add
begin

subclass semiring-0-cancel ..

Distribution rules

lemma minus-mult-left: − (a * b) = − a * b
  by (rule minus-unique) (simp add: distrib-right [symmetric])

lemma minus-mult-right: − (a * b) = a * − b
  by (rule minus-unique) (simp add: distrib-left [symmetric])

Extract signs from products

lemmas mult-minus-left [simp] = minus-mult-left [symmetric]
lemmas mult-minus-right [simp] = minus-mult-right [symmetric]

lemma minus-mult-minus [simp]: − a * − b = a * b
by simp

lemma minus-mult-commute: $- a * b = a * - b$
  by simp

lemma right-diff-distrib [algebra-simps, algebra-split-simps]:
  $a * (b - c) = a * b - a * c$
  using distrib-left [of $a b - c$] by simp

lemma left-diff-distrib [algebra-simps, algebra-split-simps]:
  $(a - b) * c = a * c - b * c$
  using distrib-right [of $a - b c$] by simp

lemmas ring-distribs = distrib-left distrib-right left-diff-distrib right-diff-distrib

lemma eq-add-iff1: $a * e + c = b * e + d \iff (a - b) * e + c = d$
  by (simp add: algebra-simps)

lemma eq-add-iff2: $a * e + c = b * e + d \iff c = (b - a) * e + d$
  by (simp add: algebra-simps)

end

lemmas ring-distribs = distrib-left distrib-right left-diff-distrib right-diff-distrib

class comm-ring = comm-semiring + ab-group-add
begin

subclass ring ..
subclass comm-semiring-0-cancel ..

lemma square-diff-square-factored: $x * x - y * y = (x + y) * (x - y)$
  by (simp add: algebra-simps)

end

class ring-1 = ring + zero-neq-one + monoid-mult
begin

subclass semiring-1-cancel ..

lemma square-diff-one-factored: $x * x - 1 = (x + 1) * (x - 1)$
  by (simp add: algebra-simps)

end

class comm-ring-1 = comm-ring + zero-neq-one + comm-monoid-mult
begin
subclass ring-1 ..
subclass comm-semiring-1-cancel
  by standard (simp add: algebra-simps)

lemma dvd-minus-iff [simp]:  
x dvd \( y \) \iff \ x dvd y
proof
  assume \( x \) dvd \( y \)
  then have \( x \) dvd \( -1 \) * \( y \) by (rule dvd-mult)
  then show \( x \) dvd \( y \) by simp
next
  assume \( x \) dvd \( y \)
  then have \( x \) dvd \( -1 \) * \( y \) by (rule dvd-mult)
  then show \( x \) dvd \( -y \) by simp
qed

lemma minus-dvd-iff [simp]:  
\( -x \) dvd \( y \) \iff \( x \) dvd \( y \)
proof
  assume \( -x \) dvd \( y \)
  then obtain \( k \) where \( y = -x \) * \( k \) ..
  then have \( y = x \) * \( -k \) by simp
  then show \( x \) dvd \( y \) ..
next
  assume \( x \) dvd \( y \)
  then obtain \( k \) where \( y = x \) * \( k \) ..
  then have \( y = -x \) * \( -k \) by simp
  then show \( -x \) dvd \( y \) ..
qed

lemma dvd-diff [simp]:  
x dvd \( y \) \implies \( x \) dvd \( z \) \implies \( x \) dvd \( y \) - \( z \)
  using \( \text{dvd-add \ [of \ \( x \) \( y \) \(- z \)]} \) by simp

end

14.3 Towards integral domains

class semiring-no-zero-divisors = semiring-0 +
  assumes no-zero-divisors: \( a \neq 0 \implies b \neq 0 \implies a \ast b \neq 0 \)
begin

lemma divisors-zero:
  assumes \( a \ast b = 0 \)
  shows \( a = 0 \lor b = 0 \)
proof (rule classical)
  assume \( \neg \)thesis
  then have \( a \neq 0 \) \emph{and} \( b \neq 0 \) by auto
  with no-zero-divisors have \( a \ast b \neq 0 \) by blast
  with \( \text{assms show \ ?thesis by simp} \)
qed
lemma mult-eq-0-iff [simp]: \( a \cdot b = 0 \iff a = 0 \lor b = 0 \)

proof (cases \( a = 0 \lor b = 0 \))
  case False
  then have \( a \neq 0 \) and \( b \neq 0 \) by auto
  then show ?thesis using no-zero-divisors by simp
next
  case True
  then show ?thesis by auto
qed

end

class semiring-1-no-zero-divisors = semiring-1 + semiring-no-zero-divisors

begin

subclass semiring-no-zero-divisors-cancel proof
  fix \( a \ b \ c \)
  have \( a \cdot c = b \cdot c \iff (a - b) \cdot c = 0 \)
    by (simp add: algebra-simps)
  also have \( \ldots \iff c = 0 \lor a = b \)
    by auto
  finally show \( a \cdot c = b \cdot c \iff c = 0 \lor a = b \).
  have \( c \cdot a = c \cdot b \iff c \cdot (a - b) = 0 \)
    by (simp add: algebra-simps)
  also have \( \ldots \iff c = 0 \lor a = b \)
    by auto
  finally show \( c \cdot a = c \cdot b \iff c = 0 \lor a = b \).
qed

end

class ring-no-zero-divisors = ring + semiring-no-zero-divisors

begin

subclass semiring-no-zero-divisors-cancel

proof
  fix \( a \ b \ c \)
  have \( a \cdot c = b \cdot c \iff (a - b) \cdot c = 0 \)
    by (simp add: algebra-simps)
  also have \( \ldots \iff c = 0 \lor a = b \)
    by auto
  finally show \( a \cdot c = b \cdot c \iff c = 0 \lor a = b \).
  have \( c \cdot a = c \cdot b \iff c \cdot (a - b) = 0 \)
    by (simp add: algebra-simps)
  also have \( \ldots \iff c = 0 \lor a = b \)
    by auto
  finally show \( c \cdot a = c \cdot b \iff c = 0 \lor a = b \).
qed

end

class ring-1-no-zero-divisors = ring-1 + ring-no-zero-divisors

begin
subclass **semiring-1-no-zero-divisors** ..

**lemma** **square-eq-1-iff**:  
\[ x \times x = 1 \iff x = 1 \lor x = -1 \]

**proof**

- have \((x - 1) \times (x + 1) = x \times x - 1\)  
  by (simp add: algebra-simps)

- then have \(x \times x = 1 \iff (x - 1) \times (x + 1) = 0\)  
  by simp

- then show \(?thesis\)
  by (simp add: eq-neg-iff-add-eq-0)

**qed**

**lemma** **mult-cancel-right1** [simp]:  
\[ c = b \times c \iff c = 0 \lor b = 1 \]

**using** mult-cancel-right [of 1 c b] by auto

**lemma** **mult-cancel-right2** [simp]:  
\[ a \times c = c \iff c = 0 \lor a = 1 \]

**using** mult-cancel-right [of a c 1] by simp

**lemma** **mult-cancel-left1** [simp]:  
\[ c = c \times b \iff c = 0 \lor b = 1 \]

**using** mult-cancel-left [of c 1 b] by force

**lemma** **mult-cancel-left2** [simp]:  
\[ c \times a = c \iff c = 0 \lor a = 1 \]

**using** mult-cancel-left [of c a 1] by simp

**end**

**class** **semidom** = **comm-semiring-1-cancel** + **semiring-no-zero-divisors**

**begin**

**subclass** **semiring-1-no-zero-divisors** ..

**end**

**class** **idom** = **comm-ring-1** + **semiring-no-zero-divisors**

**begin**

**subclass** **semidom** ..

**subclass** **ring-1-no-zero-divisors** ..

**lemma** **dvd-mult-cancel-right** [simp]:  
\[ a \times c \operatorname{dvd} b \times c \iff c = 0 \lor a \operatorname{dvd} b \]

**proof**

- have \(a \times c \operatorname{dvd} b \times c \iff (\exists k. b \times c = (a \times k) \times c)\)  
  by (auto simp add: ac-simps)

- also have \((\exists k. b \times c = (a \times k) \times c) \iff c = 0 \lor a \operatorname{dvd} b\)  
  by auto

- finally show \(?thesis\).
lemma dvd-mult-cancel-left [simp]:
\( c \cdot a \, \text{dvd} \, c \cdot b \iff c = 0 \lor a \, \text{dvd} \, b \)
using dvd-mult-cancel-right [of a c b] by (simp add: ac-simps)

lemma square-eq-iff:
\( a \cdot a = b \cdot b \iff a = b \lor a = -b \)
proof
assume \( a \cdot a = b \cdot b \)
then have \( (a - b) \cdot (a + b) = 0 \)
  by (simp add: algebra-simps)
then show \( a = b \lor a = -b \)
  by (simp add: eq-neg-iff-add-eq-0)
next
assume \( a = b \lor a = -b \)
then show \( a \cdot a = b \cdot b \)
  by auto
qed

end

class idom-abs-sgn = idom + abs + sgn +
assumes sgn-mult-abs: \( \text{sgn} \, a \cdot |a| = a \)
  and sgn-sgn [simp]: \( \text{sgn} \, \text{sgn} \, a = \text{sgn} \, a \)
  and abs-abs [simp]: \( ||a|| = |a| \)
  and abs-0 [simp]: \( |0| = 0 \)
  and sgn-0 [simp]: \( \text{sgn} \, 0 = 0 \)
  and sgn-1 [simp]: \( \text{sgn} \, 1 = 1 \)
  and sgn-minus-1: \( \text{sgn} \, (-1) = -1 \)
  and sgn-mult: \( \text{sgn} \, (a \cdot b) = \text{sgn} \, a \cdot \text{sgn} \, b \)
begin

lemma sgn-eq-0-iff:
\( \text{sgn} \, a = 0 \iff a = 0 \)
proof −
\{ assume \( \text{sgn} \, a = 0 \)
  then have \( \text{sgn} \, a \cdot |a| = 0 \)
    by simp
  then have \( a = 0 \)
    by (simp add: sgn-mult-abs)
\} then show \(?thesis
  by auto
qed

lemma abs-eq-0-iff:
\( |a| = 0 \iff a = 0 \)
proof −
\{ assume \( |a| = 0 \)
  then have \( \text{sgn} \, a \cdot |a| = 0 \)
    by simp
\}
then have $a = 0$
  
  by (simp add: sgn-mult-abs)

} then show ?thesis

by auto

qed

lemma abs-mult-sgn:

$|a| \times \text{sgn } a = a$

using sgn-mult-abs [of $a$] by (simp add: ac-simps)

lemma abs-1 [simp]:

$|1| = 1$

using sgn-mult-abs [of $1$] by simp

lemma sgn-abs [simp]:

$\text{sgn } a = \text{of-bool } (a \neq 0)$

using sgn-mult-abs [of $\text{sgn } a$] mult-cancel-left [of $\text{sgn } a \text{ \mid } \text{sgn } a \text{ \mid } 1$]

by (auto simp add: sgn-eq-0-iff)

lemma abs-sgn [simp]:

$\text{sgn } |a| = \text{of-bool } (a \neq 0)$

using sgn-mult-abs [of $|a|$] mult-cancel-right [of $\text{sgn } |a| \text{ \mid } |a| \text{ \mid } 1$]

by (auto simp add: sgn-eq-0-iff)

lemma abs-mult:

$|a \times b| = |a| \times |b|$

proof (cases $a = 0 \lor b = 0$)

  case True
  then show ?thesis
  by auto

next

  case False

  then have $*$: $\text{sgn } (a \times b) \neq 0$

  by (simp add: sgn-eq-0-iff)

  from abs-mult-sgn [of $a \times b$] abs-mult-sgn [of $a$] abs-mult-sgn [of $b$]

  have $|a \times b| \times \text{sgn } (a \times b) = |a| \times \text{sgn } a \times |b| \times \text{sgn } b$

  by (simp add: ac-simps)

  then have $|a \times b| \times \text{sgn } (a \times b) = |a| \times |b| \times \text{sgn } (a \times b)$

  by (simp add: sgn-mult ac-simps)

  with $*$ show ?thesis

  by simp

qed

lemma sgn-minus [simp]:

$\text{sgn } (-a) = -\text{sgn } a$

proof

  from sgn-minus-1 have $\text{sgn } (-1 \times a) = -1 \times \text{sgn } a$

  by (simp only: sgn-mult)

  then show ?thesis

qed
by simp

qed

lemma abs-minus [simp]:
|− a| = |a|

proof
have [simp]: |− 1| = 1
  using sgn-mul-abs [of − 1] by simp
then have |− 1 * a| = 1 * |a|
  by (simp only: abs-mul)
then show ?thesis
  by simp
qed

end

14.4 (Partial) Division

class divide =
  fixes divide :: 'a ⇒ 'a ⇒ 'a (infixl div 70)

setup ⟨Sign.add-const-constraint (const-name divide), SOME typ ('a ⇒ 'a)⟩

context semiring
begin

lemma [field-simps, field-split-simps]:
  shows distrib-left-NO-MATCH: NO-MATCH (x div y) a ⇒ a * (b + c) = a * b + a * c
    and distrib-right-NO-MATCH: NO-MATCH (x div y) c ⇒ (a + b) * c = a * c + b * c
  by (rule distrib-left distrib-right)+

end

context ring
begin

lemma [field-simps, field-split-simps]:
  shows left-diff-distrib-NO-MATCH: NO-MATCH (x div y) c ⇒ (a − b) * c = a * c − b * c
    and right-diff-distrib-NO-MATCH: NO-MATCH (x div y) a ⇒ a * (b − c) = a * b − a * c
  by (rule left-diff-distrib right-diff-distrib)+

end

setup ⟨Sign.add-const-constraint (const-name divide), SOME typ ('a::divide ⇒...
Algebraic classes with division

class semidom-divide = semidom + divide +
  assumes nonzero-mult-div-cancel-right [simp]: \( b \neq 0 \implies (a * b) \div b = a \)
  assumes div-by-0 [simp]: \( a \div 0 = 0 \)

begin

lemma nonzero-mult-div-cancel-left [simp]: \( a \neq 0 \implies (a * b) \div a = b \)
  using nonzero-mult-div-cancel-right [of a b] by (simp add: ac-simps)

subclass semiring-no-zero-divisors-cancel
proof
  show \(*\): \( a * c = b * c \iff c = 0 \lor a = b \) for \( a \ b \ c \)
  proof (cases \( c = 0 \))
    case True
    then show \(?\)thesis by simp
  next
    case False
    have \( a = b \) if \( a * c = b * c \)
    proof -
      from that have \( a * c \div c = b * c \div c \)
      by simp
      with False show \(?\)thesis
      by simp
    qed
    then show \(?\)thesis by auto
  qed
  show \( c * a = c * b \iff c = 0 \lor a = b \) for \( a \ b \ c \)
    using \(*\) [of \( a \ c \ b \)] by (simp add: ac-simps)
qed

lemma div-self [simp]: \( a \neq 0 \implies a \div a = 1 \)
  using nonzero-mult-div-cancel-left [of \( a \ 1 \)] by simp

lemma div-0 [simp]: \( 0 \div a = 0 \)
proof (cases \( a = 0 \))
  case True
  then show \(?\)thesis by simp
next
  case False
  then have \( a * 0 \div a = 0 \)
  by (rule nonzero-mult-div-cancel-left)
  then show \(?\)thesis by simp
qed

lemma div-by-1 [simp]: \( a \div 1 = a \)
  using nonzero-mult-div-cancel-left [of \( 1 \ a \)] by simp
lemma dvd-div-eq-0-iff:
  assumes b dvd a
  shows a div b = 0 ⇐⇒ a = 0
  using assms by (elim dvdE, cases b = 0) simp-all

lemma dvd-div-eq-cancel:
  a div c = b div c ⇒ c dvd a ⇒ c dvd b ⇒ a = b
  by (elim dvdE, cases c = 0) simp-all

lemma dvd-div-eq-iff:
  c dvd a ⇒ c dvd b ⇒ a div c = b div c ⇐⇒ a = b
  by (elim dvdE, cases c = 0) simp-all

lemma inj-on-mult:
  inj-on ((∗) a) A if a ≠ 0
proof (rule inj-onI)
  fix b c
  assume a ∗ b = a ∗ c
  then have a ∗ b div a = a ∗ c div a
    by (simp only:)
  with that show b = c
    by simp
qed

end

class idom-divide = idom + semidom-divide
begin

lemma dvd-neg-div:
  assumes b dvd a
  shows −a div b = −(a div b)
proof (cases b = 0)
  case True
  then show ?thesis by simp
next case False
  from assms obtain c where a = b ∗ c ..
  then have −a div b = (b ∗ −c) div b
    by simp
  from False also have .. = −c
    by (rule nonzero-mult-div-cancel-left)
  with False ⟨a = b ∗ c⟩ show ?thesis
    by simp
qed

lemma dvd-div-neg:
  assumes b dvd a
  shows a div −b = −(a div b)
THEORY "Rings"

proof (cases b = 0)
case True then show ?thesis by simp
next
case False then have \(- b \neq 0\) by simp
from assms obtain c where \(a = b \cdot c\) ..
then have \(a \div b = (- b \cdot - c) \div b\)
by simp
from \((- b \neq 0\), also have \(\ldots = - c\)
by (rule nonzero-mult-div-cancel-left)
with False \(\langle a = b \cdot c\rangle\) show ?thesis
by simp
qed

class algebraic-semidom = semidom-divide
begin
Class algebraic-semidom enriches a integral domain by notions from algebra,
like units in a ring. It is a separate class to avoid spoiling fields with notions
which are degenerated there.

lemma dvd-times-left-cancel-iff [simp]:
assumes \(a \neq 0\)
shows \(a \cdot b \ dvd a \cdot c \longleftrightarrow b \ dvd c\)
(is ?lhs \longleftrightarrow ?rhs)
proof
assume ?lhs
then obtain d where \(a \cdot c = a \cdot b \cdot d\) ..
with assms have \(c = b \cdot d\) by (simp add: ac-simps)
then show ?rhs ..
next
assume ?rhs
then obtain d where \(c = b \cdot d\) ..
then have \(a \cdot c = a \cdot b \cdot d\) by (simp add: ac-simps)
then show ?lhs ..
qed

lemma dvd-times-right-cancel-iff [simp]:
assumes \(a \neq 0\)
shows \(b \cdot a \ dvd c \cdot a \longleftrightarrow b \ dvd c\)
using dvd-times-left-cancel-iff [of a b c] assms by (simp add: ac-simps)

lemma div-dvd-iff-mult:
assumes \(b \neq 0\) and \(b \ dvd a\)
shows \(a \div b \ dvd c \longleftrightarrow a \ dvd c \cdot b\)
proof –
lemma dvd-div-iff-mult:
  assumes c ≠ 0 and c dvd b
  shows a dvd b div c ←→ a * c dvd b
proof –
  from ⟨c dvd b⟩ obtain d where b = c * d ..
  with ⟨c ≠ 0⟩ show ?thesis by (simp add: mult.commute[of a])
qed

lemma div-dvd-div [simp]:
  assumes a dvd b and a dvd c
  shows (a + b) div c = a div c + b div c
proof (cases a = 0)
  case True
  with assms show ?thesis by simp
next
  case False
  moreover from assms obtain k l where b = a * k and c = a * l
  by blast
  ultimately show ?thesis by simp
qed

lemma div-add [simp]:
  assumes c dvd a and c dvd b
  shows (a + b) div c = a div c + b div c
proof (cases c = 0)
  case True
  then show ?thesis by simp
next
  case False
  moreover from assms obtain k l where a = c * k and b = c * l
  by blast
  moreover have c * k + c * l = c * (k + l)
  by (simp add: algebra-simps)
  ultimately show ?thesis
  by simp
qed

lemma div-mult-div-if-dvd:
  assumes b dvd a and d dvd c
  shows (a div b) * (c div d) = (a * c) div (b * d)
proof (cases b = 0 ∨ c = 0)
  case True
  with assms show ?thesis by auto
next
  case False
moreover from \textit{assms} obtain \( k \) \( l \) \textit{where} \( a = b * k \) \textit{and} \( c = d * l \)
by \textit{blast}
moreover have \( b * k * (d * l) \div (b * d) = (b * d) * (k * l) \div (b * d) \)
by \textit{(simp add: ac-simps)}
ultimately show \( ?thesis \) \textit{by simp}

\textbf{lemma} \textit{dvd-div-eq-mult}:  
\textit{assumes} \( a \neq 0 \) \textit{and} \( a \ \textit{dvd} \ \ b \)  
\textit{shows} \( b \ \textit{div} \ \ a = c \longleftrightarrow b = c * a \)  
\textit{(is \( ?lhs \longleftrightarrow ?rhs \))}  
\textit{proof}  
\textit{assume} \( ?rhs \)  
\textit{then show} \( ?lhs \) \textit{by (simp add: \textit{assms})}  
\textit{next}  
\textit{assume} \( ?lhs \)  
\textit{then have} \( b \ \textit{div} \ a = c \leftrightarrow b = c * a \) \textit{by simp}  
\textit{moreover from \textit{assms} have} \( b \ \textit{div} \ a = c \) \textit{by (auto simp add: \textit{ac-simps})}  
\textit{ultimately show} \( ?rhs \) \textit{by simp}
\textbf{qed}

\textbf{lemma} \textit{dvd-div-mult-self} \textit{[simp]}: \( a \ \textit{dvd} \ \ b \implies b \ \textit{div} \ a * a = b \)  
\textit{by (cases \( a = 0 \)) (auto simp add: \textit{ac-simps})}

\textbf{lemma} \textit{dvd-mult-div-cancel} \textit{[simp]}: \( a \ \textit{dvd} \ \ b \implies a * (b \ \textit{div} \ a) = b \)  
\textit{using} \textit{dvd-div-mult-self} \textit{[of \ \( a \ \textit{b} \)] by (simp add: \textit{ac-simps})}

\textbf{lemma} \textit{div-mult-swap}:  
\textit{assumes} \( c \ \textit{dvd} \ \ b \)  
\textit{shows} \( a * (b \ \textit{div} \ c) = (a * b) \ \textit{div} \ c \)  
\textit{proof} \textit{(cases \( c = 0 \))}  
\textit{case} \( True \)  
\textit{then show} \( ?thesis \) \textit{by simp}  
\textit{next}  
\textit{case} \( False \)  
\textit{from \textit{assms} obtain} \( d \) \textit{where} \( b = c * d \) \ldots  
\textit{moreover from} \( False \) \textit{have} \( a * \textit{divide} \ (d * c) \ c = ((a * d) * c) \ \textit{div} \ c \) \textit{by simp}  
\textit{ultimately show} \( ?thesis \) \textit{by (simp add: \textit{ac-simps})}
\textbf{qed}

\textbf{lemma} \textit{dvd-div-mult}: \( c \ \textit{dvd} \ \ b \implies b \ \textit{div} \ c * a = (b * a) \ \textit{div} \ c \)  
\textit{using} \textit{div-mult-swap} \textit{[of \ \( c \ \textit{b} \ \textit{a} \)] by (simp add: \textit{ac-simps})}

\textbf{lemma} \textit{dvd-div-mult2-eq}:  
\textit{assumes} \( b * c \ \textit{dvd} \ \ a \)  
\textit{shows} \( a \ \textit{div} \ (b * c) = a \ \textit{div} \ b \ \textit{div} \ c \)  
\textit{proof} \ldots
from assms obtain k where \( a = b \cdot c \cdot k \).

then show \(?\text{thesis}\)
  by (cases \( b = 0 \lor c = 0 \)) (auto, simp add: ac-simps)
qed

lemma dvd-div-div-eq-mult:
  assumes \( a \neq 0 \) \( c \neq 0 \) and \( a \) dvd \( b \) \( c \) dvd \( d \)
  shows \( b \div a = d \div c \leftrightarrow b \cdot c = a \cdot d \)
  (is \(?\text{lhs} \leftrightarrow ?\text{rhs}\))
proof
  from assms have \( a \cdot c \neq 0 \)
    by simp
  then have \(?\text{lhs} \leftrightarrow b \div a \cdot (a \cdot c) = d \div c \cdot (a \cdot c)\)
    by simp
  also have \( \ldots \leftrightarrow (a \cdot (b \div a)) \cdot c = (c \cdot (d \div c)) \cdot a \)
    by (simp add: ac-simps)
  also have \( \ldots \leftrightarrow (a \cdot b \div a) \cdot c = (c \cdot d \div c) \cdot a \)
    using assms by (simp add: div-mult-swap)
  also have \( \ldots \leftrightarrow ?\text{rhs} \)
    using assms by (simp add: ac-simps)
  finally show \(?\text{thesis}\).
qed

lemma dvd-mult-imp-div:
  assumes \( a \cdot c \) dvd \( b \)
  shows \( a \) dvd \( b \div c \)
proof (cases \( c = 0 \))
  case True
  then show \(?\text{thesis}\)
    by auto
next
  case False
  from \( a \cdot c \) dvd \( b \)
    obtain \( d \) where \( b = a \cdot c \cdot d \).
  with False show \(?\text{thesis}\)
    by (simp add: mult.commute [of a] mult.assoc)
qed

lemma div-div-eq-right:
  assumes \( c \) dvd \( b \cdot b \) dvd \( a \)
  shows \( a \div (b \div c) = a \div b \cdot c \)
proof (cases \( c = 0 \lor b = 0 \))
  case True
  then show \(?\text{thesis}\)
    by auto
next
  case False
  from assms obtain \( r \) \( s \) where \( b = c \cdot r \) and \( a = c \cdot r \cdot s \)
    by blast
  moreover with False have \( r \neq 0 \)
    by auto
  ultimately show \(?\text{thesis}\) using False
    by simp (simp add: mult.commute [of - r] mult.assoc mult.commute [of c])
**lemma div-div-div-same:**
assumes $d \mid b \land b \mid a$
shows $(a \div d) \div (b \div d) = a \div b$
proof (cases $b = 0 \lor d = 0$)
case $True$
with assms show $?thesis$
  by auto
next
case $False$
from assms obtain $r \ s$
  where $a = d \ast r \ast s$ and $b = d \ast r$
  by blast
with $False$ show $?thesis$
  by simp (simp add: ac-simps)
qed

Units: invertible elements in a ring

**abbreviation is-unit :: \'a \Rightarrow bool**
where is-unit $a \equiv a \mid 1$

**lemma not-is-unit-0 [simp]:** $\neg$ is-unit 0
  by simp

**lemma unit-imp-dvd [dest]:** is-unit $b \Rightarrow b \mid a$
  by (rule dvd-trans [of - 1]) simp-all

**lemma unit-dvdE:**
assumes is-unit $a$
obtains $c$ where $a \neq 0$ and $b = a \ast c$
proof --
  from assms have $a \mid b$ by auto
  then obtain $c$ where $b = a \ast c$ ..
moreover from assms have $a \neq 0$ by auto
ultimately show thesis using that by blast
qed

**lemma dvd-unit-imp-unit: a dvd b \Rightarrow is-unit b \Rightarrow is-unit a**
  by (rule dvd-trans)

**lemma unit-div-1-unit [simp, intro]:**
assumes is-unit $a$
shows is-unit (1 div $a$)
proof --
  from assms have $1 = 1 \div a \ast a$ by simp
  then show is-unit (1 div $a$) by (rule dvdI)
qed
lemma is-unitE [elim?]:
assumes is-unit b
obtains a where a ≠ 0 and b ≠ 0
and is-unit b and 1 div b = a
and a * b = 1
and 1 div a = c * b
proof (rule that)
define b where b = 1 div a
then show 1 div a = b by simp
from assms b-def show is-unit b by simp
with assms show a ≠ 0 and b ≠ 0 by auto
from assms b-def show a * b = 1 by simp
then have 1 = a * b ..
with b-def ⟨b ≠ 0⟩ show 1 div b = a by simp
from assms have a dvd c ..
then obtain d where c = a * d ..
with ⟨a ≠ 0⟩ ⟨a * b = 1⟩ show c div a = c * b
by (simp add: mult-ac assoc left-commute [of a])
qed

lemma unit-prod [intro]: is-unit a ⇒ is-unit b ⇒ is-unit (a * b)
by (subst mult-1-left [of 1, symmetric]) (rule mult-dvd-mono)

lemma is-unit-mult-iff: is-unit (a * b) ←→ is-unit a ∧ is-unit b
by (auto dest: dvd-mult-left dvd-mult-right)

lemma unit-div [intro]: is-unit a ⇒ is-unit b ⇒ is-unit (a div b)
by (erule is-unitE [of b a]) (simp add: ac-simps unit-prod)

lemma mult-unit-dvd-iff:
assumes is-unit b
shows a * b dvd c ⟷ a dvd c
proof
assume a * b dvd c
with assms show a dvd c
by (simp add: dvd-mult-left)
next
assume a dvd c
then obtain k where c = a * k ..
with assms have c = (a * b) * (1 div b * k)
by (simp add: mult-ac)
then show a * b dvd c by (rule dvdI)
qed

lemma mult-unit-dvd-iff': is-unit a ⇒ (a * b) dvd c ⟷ b dvd c
using mult-unit-dvd-iff [of a b c] by (simp add: ac-simps)

lemma dvd-mult-unit-iff:
assumes is-unit b
shows a dvd c * b ⟷ a dvd c
proof
  assume a dvd c * b
with assms have c * b dvd c * (b * (1 div b))
    by (subst mult-assoc [symmetric]) simp
also from assms have b * (1 div b) = 1
    by (rule is-unitE) simp
finally have c * b dvd c by simp
with ⟨a dvd c * b⟩ show a dvd c by (rule dvd-trans)
next
  assume a dvd c
  then show a dvd c * b by simp
qed

lemma dvd-mult-unit-iff': is-unit b =⇒ a dvd b * c if a dvd c
  using dvd-mult-unit-iff[of b a c] by (simp add: ac-simps)

lemma div-unit-dvd-iff: is-unit b =⇒ a div b dvd c if a dvd c
  by (erule is-unitE[of - a]) (auto simp add: mult-unit-dvd-iff)

lemma dvd-div-unit-iff: is-unit b =⇒ a dvd c div b if a dvd c
  by (erule is-unitE[of - c]) (simp add: dvd-mult-unit-iff)

lemmas unit-dvd-iff = mult-unit-dvd-iff mult-unit-dvd-iff'
dvd-mult-unit-iff dvd-mult-unit-iff'
div-unit-dvd-iff dvd-div-unit-iff

lemma unit-mult-div-div [simp]: is-unit a =⇒ b * (1 div a) = b div a
  by (erule is-unitE[of - b]) simp

lemma unit-div-mult-self [simp]: is-unit a =⇒ b div a * a = b
  by (rule dvd-div-mult-self) auto

lemma unit-div-1-div-1 [simp]: is-unit a =⇒ 1 div (1 div a) = a
  by (erule is-unitE) simp

lemma unit-div-mult-swap: is-unit c =⇒ a * (b div c) = (a * b) div c
  by (erule unit-dvdE[of - b]) (simp add: mult.left-commute[of - c])

lemma unit-div-commute: is-unit b =⇒ (a div b) * c = (a * c) div b
  using unit-div-mult-swap[of b c a] by (simp add: ac-simps)

lemma unit-eq-div1: is-unit b =⇒ a div b = c if a = c * b
  by (auto elim: is-unitE)

lemma unit-eq-div2: is-unit b =⇒ a = c div b if a * b = c
  using unit-eq-div1[of b c a] by auto

lemma unit-mult-left-cancel: is-unit a =⇒ a * b = a * c if b = c
  using mult-cancel-left[of a b c] by auto
lemma unit-mult-right-cancel: is-unit a → b * a = c * a ↔ b = c
  using unit-mult-left-cancel [of a b c] by (auto simp add: ac-simps)

lemma unit-div-cancel:
  assumes is-unit a
  shows b div a = c div a ↔ b = c
proof –
  from assms have is-unit (1 div a) by simp
  then have b * (1 div a) = c * (1 div a) ↔ b = c
    by (rule unit-mult-right-cancel)
  with assms show ?thesis by simp
qed

lemma is-unit-div-mult2-eq:
  assumes is-unit b and is-unit c
  shows a div (b * c) = a div b div c
proof –
  from assms have is-unit (b * c)
    by (simp add: unit-prod)
  then have b * c dvd a
    by (rule unit-imp-dvd)
  then show ?thesis
    by (rule dvd-div-mult2-eq)
qed

lemma is-unit-div-mult-cancel-left:
  assumes a ≠ 0 and is-unit b
  shows a div (a * b) = 1 div b
proof –
  from assms have a div (a * b) = a div a div b
    by (simp add: mult-unit-dvd-iff dvd-div-mult2-eq)
  with assms show ?thesis by simp
qed

lemma is-unit-div-mult-cancel-right:
  assumes a ≠ 0 and is-unit b
  shows a div (b * a) = 1 div b
  using assms is-unit-div-mult-cancel-left [of a b] by (simp add: ac-simps)

lemma unit-div-eq-0-iff:
  assumes is-unit b
  shows a div b = 0 ↔ a = 0
  by (rule dvd-div-eq-0-iff) (insert assms, auto)

lemma div-mult-unit2:
  is-unit c → b dvd a → a div (b * c) = a div b div c
  by (rule dvd-div-mult2-eq) (simp-all add: mult-unit-dvd-iff)

Coprimality
definition coprime :: 'a ⇒ 'a ⇒ bool
where coprime a b ←→ (∀ c. c dvd a → c dvd b → is-unit c)

lemma coprimeI:
assumes (∀ c. c dvd a → c dvd b → is-unit c)
shows coprime a b
using assms by (auto simp: coprime-def)

lemma not-coprimeI:
assumes c dvd a and c dvd b and ¬ is-unit c
shows ¬ coprime a b
using assms by (auto simp: coprime-def)

lemma coprime-common-divisor:
is-unit c if coprime a b and c dvd a and c dvd b
using that by (auto simp: coprime-def)

lemma not-coprimeE:
assumes ¬ coprime a b
obtains c where c dvd a and c dvd b and ¬ is-unit c
using assms by (auto simp: coprime-def)

lemma coprime-imp-coprime:
coprime a b if coprime c d
and (∀ e. ¬ is-unit e → e dvd a → e dvd b → e dvd c
and (∀ e. ¬ is-unit e → e dvd a → e dvd b → e dvd d
proof (rule coprimeI)
  fix e
  assume e dvd a and e dvd b
  with that have e dvd c and e dvd d
  by (auto intro: dvd-trans)
  with ⟨coprime c d⟩ show is-unit e
  by (rule coprime-common-divisor)
qed

lemma coprime-divisors:
coprime a b if a dvd c b dvd d and coprime c d
using ⟨coprime c d⟩ proof (rule coprime-imp-coprime)
  fix e
  assume e dvd a then show e dvd c
  using ⟨a dvd c⟩ by (rule dvd-trans)
  assume e dvd b then show e dvd d
  using ⟨b dvd d⟩ by (rule dvd-trans)
  qed

lemma coprime-self [simp]:
coprime a a ←→ is-unit a (is ?P ←→ ?Q)
proof
  assume ?P
then show \(?Q\)
  by (rule coprime-common-divisor) simp-all

next
assume \(?Q\)
show \(?P\)
  by (rule coprimeI) (erule dvd-unit-imp-unit, rule \(?Q\))

qed

lemma coprime-commute [ac-simps]:
coprime b a \iff coprime a b

unfolding coprime-def by auto

lemma is-unit-left-imp-coprime:
coprime a b if is-unit a

proof (rule coprimeI)
fix c
assume c dvd a
with that show is-unit c
  by (auto intro: dvd-unit-imp-unit)

qed

lemma is-unit-right-imp-coprime:
coprime a b if is-unit b
using that is-unit-left-imp-coprime [of b a] by (simp add: ac-simps)

lemma coprime-1-left [simp]:
coprime 1 a

by (rule coprimeI)

lemma coprime-1-right [simp]:
coprime a 1

by (rule coprimeI)

lemma coprime-0-left-iff [simp]:
coprime 0 a \iff is-unit a

by (auto intro: coprimeI dvd-unit-imp-unit coprime-common-divisor [of 0 a a])

lemma coprime-0-right-iff [simp]:
coprime a 0 \iff is-unit a

using coprime-0-left-iff [of a] by (simp add: ac-simps)

lemma coprime-mult-self-left-iff [simp]:
coprime (c * a) (c * b) \iff is-unit c \land coprime a b

by (auto intro: coprime-common-divisor)
(rule coprimeI, auto intro: coprime-common-divisor simp add: dvd-mult-unit-iff)

lemma coprime-mult-self-right-iff [simp]:
coprime (a * c) (b * c) \iff is-unit c \land coprime a b

using coprime-mult-self-left-iff [of c a b] by (simp add: ac-simps)
lemma coprime-absorb-left:
  assumes x dvd y
  shows  coprime x y ⟷ is-unit x
  using assms coprime-common-divisor is-unit-left-imp-coprime by auto

lemma coprime-absorb-right:
  assumes y dvd x
  shows  coprime x y ⟷ is-unit y
  using assms coprime-common-divisor is-unit-right-imp-coprime by auto

end

class unit-factor =
  fixes unit-factor :: 'a ⇒ 'a

class semidom-divide-unit-factor = semidom-divide + unit-factor +
  assumes unit-factor-0 [simp]: unit-factor 0 = 0
  and is-unit-unit-factor: a dvd 1 ⟷ unit-factor a = a
  and unit-factor-is-unit: a ≠ 0 ⟷ unit-factor a dvd 1
  and unit-factor-mult-unit-left: a dvd 1 ⟷ unit-factor (a * b) = a * unit-factor b
— This fine-grained hierarchy will later on allow lean normalization of polynomials

begin

lemma unit-factor-mult-unit-right: a dvd 1 ⟷ unit-factor (b * a) = unit-factor b * a
  using unit-factor-mult-unit-left[of a b] by (simp add: mult_ac)

lemmas [simp] = unit-factor-mult-unit-left unit-factor-mult-unit-right

end

class normalization-semidom = algebraic-semidom + semidom-divide-unit-factor +
  fixes normalize :: 'a ⇒ 'a
  assumes unit-factor-mult-normalize [simp]: unit-factor a * normalize a = a
  and normalize-0 [simp]: normalize 0 = 0

begin

Class normalization-semidom cultivates the idea that each integral domain
can be split into equivalence classes whose representants are associated, i.e.
divide each other. normalize specifies a canonical representant for each
equivalence class. The rationale behind this is that it is easier to reason
about equality than equivalences, hence we prefer to think about equality
of normalized values rather than associated elements.

declare unit-factor-is-unit [iff]

lemma unit-factor-dvd [simp]: a ≠ 0 ⟷ unit-factor a dvd b
by (rule unit-imp-dvd) simp

lemma unit-factor-self [simp]: unit-factor a dvd a
  by (cases a = 0) simp-all

lemma normalize-mult-unit-factor [simp]: normalize a * unit-factor a = a
  using unit-factor-mult-normalize [of a] by (simp add: ac-simps)

lemma normalize-eq-0-iff [simp]: normalize a = 0 ←→ a = 0
  (is ?lhs ↔ ?rhs)
proof
  assume ?lhs
  moreover have unit-factor a * normalize a = a by simp
  ultimately show ?rhs by simp
next
  assume ?rhs
  then show ?lhs by simp
qed

lemma unit-factor-eq-0-iff [simp]: unit-factor a = 0 ←→ a = 0
  (is ?lhs ↔ ?rhs)
proof
  assume ?lhs
  moreover have unit-factor a * normalize a = a by simp
  ultimately show ?rhs by simp
next
  assume ?rhs
  then show ?lhs by simp
qed

lemma div-unit-factor [simp]: a div unit-factor a = normalize a
proof (cases a = 0)
  case True
  then show ?thesis by simp
next
  case False
  then have unit-factor a ≠ 0
    by simp
  with nonzero-mult-cancel-left
  have unit-factor a * normalize a div unit-factor a = normalize a
    by blast
  then show ?thesis by simp
qed

lemma normalize-div [simp]: normalize a div a = 1 div unit-factor a
proof (cases a = 0)
  case True
  then show ?thesis by simp
next
case False
have normalize a div a = normalize a div (unit-factor a * normalize a)
  by simp
also have \ldots = 1 div unit-factor a
  using False by (subst is-unit-div-mult-cancel-right) simp-all
finally show ?thesis .
qed

lemma is-unit-normalize:
assumes is-unit a
shows normalize a = 1
proof –
from assms have unit-factor a = a
  by (rule is-unit-unit-factor)
moreover from assms have a \neq 0
  by auto
moreover have normalize a = a div unit-factor a
  by simp
ultimately show ?thesis
  by simp
qed

lemma unit-factor-1 [simp]: unit-factor 1 = 1
by (rule is-unit-unit-factor) simp

lemma normalize-1 [simp]: normalize 1 = 1
by (rule is-unit-normalize) simp

lemma normalize-1-iff: normalize a = 1 \iff is-unit a
(is ?lhs \iff ?rhs)
proof
assume ?rhs
then show ?lhs by (rule is-unit-normalize)
next
assume ?lhs
then have unit-factor a * normalize a = unit-factor a * 1
  by simp
then have unit-factor a = a
  by simp
moreover
from (?lhs) have a \neq 0 by auto
then have is-unit (unit-factor a) by simp
ultimately show ?rhs by simp
qed

lemma div-normalize [simp]: a div normalize a = unit-factor a
proof (cases a = 0)
case True
then show ?thesis by simp
next
  case False
  then have normalize a ≠ 0 by simp
  with nonzero-mult-div-cancel-right
  have unit-factor a * normalize a div normalize a = unit-factor a by blast
  then show ?thesis by simp
qed

lemma mult-one-div-unit-factor [simp]: a * (1 div unit-factor b) = a div unit-factor b
  by (cases b = 0) simp-all

lemma inv-unit-factor-eq-0-iff [simp]:
  1 div unit-factor a = 0 ←→ a = 0
  (is ?lhs ←→ ?rhs)
proof
  assume ?lhs
  then have a * (1 div unit-factor a) = a * 0
    by simp
  then show ?rhs
    by simp
next
  assume ?rhs
  then show ?lhs by simp
qed

lemma unit-factor-idem [simp]: unit-factor (unit-factor a) = unit-factor a
  by (cases a = 0) (auto intro: is-unit-unit-factor)

lemma normalize-unit-factor [simp]: a ≠ 0 ⟹ normalize (unit-factor a) = 1
  by (rule is-unit-normalize) simp

lemma normalize-mult-unit-left [simp]:
  assumes a dvd 1
  shows normalize (a * b) = normalize b
proof (cases b = 0)
  case False
  have a * unit-factor b * normalize (a * b) = unit-factor (a * b) * normalize (a * b)
    using assms by (subst unit-factor-mult-unit-left) auto
  also have ... = a * b by simp
  also have b = unit-factor b * normalize b by simp
  hence a * b = a * unit-factor b * normalize b
    by (simp only: mult-ac)
  finally show ?thesis
    using assms False by auto
qed auto

lemma normalize-mult-unit-right [simp]:
assumes \( b \text{ dvd } 1 \)
shows \( \text{normalize } (a \ast b) = \text{normalize } a \)
using assms by (subst mult.commute) auto

lemma normalize-idem [simp]: \( \text{normalize } (\text{normalize } a) = \text{normalize } a \)
proof (cases \( a = 0 \))
  case False
  have \( \text{normalize } a = \text{normalize } (\text{unit-factor } a \ast \text{normalize } a) \)
    by simp
  also from False have \( \ldots = \text{normalize } (\text{normalize } a) \)
    by (subst normalize-mult-unit-left) auto
  finally show \( ?\text{thesis} .. \)
qed auto

lemma unit-factor-normalize [simp]:
assumes \( a \neq 0 \)
shows \( \text{unit-factor } (\text{normalize } a) = 1 \)
proof
  from assms have \( \ast: \text{normalize } a \neq 0 \)
    by simp
  have \( \text{unit-factor } (\text{normalize } a) \ast \text{normalize } (\text{normalize } a) = \text{normalize } a \)
    by (simp only: unit-factor-mult-normalize)
  then have \( \text{unit-factor } (\text{normalize } a) \ast \text{normalize } a = \text{normalize } a \)
    by simp
  with \( \ast \) have \( \text{unit-factor } (\text{normalize } a) \ast \text{normalize } a \text{ div } \text{normalize } a = \text{normalize } a \text{ div } \text{normalize } a \)
    by simp
  with \( \ast \) show \( ?\text{thesis} \)
    by simp
qed

lemma normalize-dvd-iff [simp]: \( \text{normalize } a \text{ dvd } b \iff a \text{ dvd } b \)
proof
  have \( \text{normalize } a \text{ dvd } b \iff \text{unit-factor } a \ast \text{normalize } a \text{ dvd } b \)
    using mult-unit-dvd-iff [of \( \text{unit-factor } a \text{ normalize } a \text{ b} \)]
    by (cases \( a = 0 \)) simp-all
  then show \( ?\text{thesis} \) by simp
qed

lemma dvd-normalize-iff [simp]: \( a \text{ dvd normalize } b \iff a \text{ dvd } b \)
proof
  have \( a \text{ dvd normalize } b \iff a \text{ dvd normalize } b \ast \text{unit-factor } b \)
    using dvd-mult-unit-iff [of \( \text{unit-factor } b \text{ a normalize } b \)]
    by (cases \( b = 0 \)) simp-all
  then show \( ?\text{thesis} \) by simp
qed

lemma normalize-idem-imp-unit-factor-eq:
assumes \( \text{normalize } a = a \)
shows unit-factor \( a = \text{of-bool} (a \neq 0) \)

proof (cases \( a = 0 \))
  case True
  then show \(?thesis\)
    by simp
next
  case False
  then show \(?thesis\)
    using assms unit-factor-normalize \([of a]\) by simp
qed

lemma normalize-idem-imp-is-unit-iff:
  assumes normalize \( a = a \)
  shows \( \text{is-unit} a \iff a = 1 \)
  using assms by (cases \( a = 0 \)) \((auto dest: is-unit-normalize)\)

lemma coprime-normalize-left-iff [simp]:
  coprime \( (\text{normalize} a) b \iff \text{coprime} a b \)
  by (rule; rule coprimeI) \((auto intro: coprime-common-divisor)\)

lemma coprime-normalize-right-iff [simp]:
  coprime a \( (\text{normalize} b) \iff \text{coprime} a b \)
  using coprime-normalize-left-iff \([of b a]\) by \((simp add: ac-simps)\)

We avoid an explicit definition of associated elements but prefer explicit normalisation instead. In theory we could define an abbreviation like \( \text{associated} a b = (\text{normalize} a = \text{normalize} b) \) but this is counterproductive without suggestive infix syntax, which we do not want to sacrifice for this purpose here.

lemma associatedI:
  assumes \( a \mid b \) and \( b \mid a \)
  shows \( \text{normalize} a = \text{normalize} b \)
  proof (cases \( a = 0 \lor b = 0 \))
    case True
    with assms show \(?thesis\) by auto
next
  case False
  from \( (a \mid b) \) obtain \( c \) where \( b = a \cdot c \) ..
  moreover from \( (b \mid d) \) obtain \( d \) where \( a = b \cdot d \) ..
  ultimately have \( b \cdot 1 = b \cdot (c \cdot d) \)
    by \((simp add: ac-simps)\)
  with False have \( 1 = c \cdot d \)
    unfolding mult-cancel-left by simp
  then have \( \text{is-unit} c \) and \( \text{is-unit} d \)
    by auto
  with \( a \ b \) show \(?thesis\)
    by \((simp add: is-unit-normalize)\)
qed
lemma associatedD1: \(\text{normalize } a = \text{normalize } b \implies a \text{ dvd } b\)
using dvd-normalize-iff \([of \ b, \text{symmetric}]\) normalize-dvd-iff \([of \ a, \text{symmetric}]\)
by simp

lemma associatedD2: \(\text{normalize } a = \text{normalize } b \implies b \text{ dvd } a\)
using dvd-normalize-iff \([of \ a, \text{symmetric}]\) normalize-dvd-iff \([of \ b, \text{symmetric}]\)
by simp

lemma associated-unit: \(\text{normalize } a = \text{normalize } b \implies \text{is-unit } a \implies \text{is-unit } b\)
using dvd-unit-imp-unit by (auto dest: associatedD1 associatedD2)

lemma associated-iff-dvd: \(\text{normalize } a = \text{normalize } b \iff a \text{ dvd } b \land b \text{ dvd } a\)
(is \(?lhs \longleftrightarrow \?rhs\))
proof
assume \(?rhs\)
then show \(?lhs\) by (auto intro!: associatedI)
next
assume \(?lhs\)
then have \(\text{unit-factor } a \ast \text{normalize } a = \text{unit-factor } a \ast \text{normalize } b\)
by simp
then have \(*: \text{normalize } b \ast \text{unit-factor } a = a\)
by (simp add: ac-simps)
show \(?rhs\)
proof (cases a = 0 \lor b = 0)
case True
with \(?lhs\) show \(?thesis\) by auto
next
case False
then have \(b \text{ dvd } \text{normalize } b \ast \text{unit-factor } a\) and \(\text{normalize } b \ast \text{unit-factor } a\) \(\text{dvd } b\)
by (simp-all add: mult-unit-dvd-iff dvd-mult-unit-iff)
with \(*\) show \(?thesis\) by simp
qed

lemma associated-eqI:
assumes \(a \text{ dvd } b\) and \(b \text{ dvd } a\)
assumes \(\text{normalize } a = a\) and \(\text{normalize } b = b\)
shows \(a = b\)
proof
from \(\text{assms}\) have \(\text{normalize } a = \text{normalize } b\)
unfolding associated-iff-dvd by simp
with \(\text{normalize } a = a\) have \(a = \text{normalize } b\)
by simp
with \(\text{normalize } b = b\) show \(a = b\)
by simp
qed

lemma normalize-unit-factor-eqI:
assumes normalize $a = normalize b$
and unit-factor $a = unit-factor b$
shows $a = b$
proof
  from assms have unit-factor $a * normalize a = unit-factor b * normalize b$
    by simp
  then show \textit{thesis}
    by simp
qed

lemma normalize-mult-normalize-left [simp]: normalize ($normalize a * b$) = normalize ($a * b$)
  by (rule associated-eqI) (auto intro!: mult-dvd-mono)

lemma normalize-mult-normalize-right [simp]: normalize ($a * normalize b$) = normalize ($a * b$)
  by (rule associated-eqI) (auto intro!: mult-dvd-mono)

end

class normalization-semidom-multiplicative = normalization-semidom +
  assumes unit-factor-mult: unit-factor ($a * b$) = unit-factor $a * unit-factor b$
begin

lemma normalize-mult: normalize ($a * b$) = normalize $a * normalize b$
proof (cases $a = 0 \lor b = 0$)
  case True
  then show \textit{thesis} by auto
next
  case False
  have unit-factor ($a * b$) * normalize ($a * b$) = $a * b$
    by (rule unit-factor-mult-normalize)
  then have normalize ($a * b$) = $a * b$ div unit-factor ($a * b$)
    by simp
  also have \ldots = $a * b$ div unit-factor ($b * a$)
    by (simp add: ac-simps)
  also have \ldots = $a * b$ div unit-factor $b$ div unit-factor $a$
    using False by (simp add: unit-factor-mult is-unit-div-mult2-eq [symmetric])
  also have \ldots = $a * (b$ div unit-factor $b)$ div unit-factor $a$
    using False by (subst unit-div-mult-swap) simp-all
  also have \ldots = normalize $a * normalize b$
    using False
      by (simp add: mult.commute [of $a$] mult.commute [of normalize $a$] unit-div-mult-swap [symmetric])
  finally show \textit{thesis} .
qed

lemma dvd-unit-factor-div:
assumes \( b \) dvd \( a \)
shows \( \text{unit-factor} \left( \frac{a}{b} \right) = \text{unit-factor} \ a \ \text{div} \ \text{unit-factor} \ b \)
proof
  from \( \text{assms} \) have \( a = a \ \text{div} \ b \star b \)
    by simp
  then have \( \text{unit-factor} \ a = \text{unit-factor} \ (a \ \text{div} \ b \star b) \)
    by simp
  then show \(?\text{thesis} \)
    by (cases \( b = 0 \)) \( \text{simp-all: unit-factor-mult} \)
qed

lemma dvd-normalize-div:
assumes \( b \) dvd \( a \)
shows \( \text{normalize} \ \left( \frac{a}{b} \right) = \text{normalize} \ a \ \text{div} \ \text{normalize} \ b \)
proof
  from \( \text{assms} \) have \( a = a \ \text{div} \ b \star b \)
    by simp
  moreover have \( a \ \text{div} \ b \star b = a \ \text{div} \ b \star b \)
  moreover have \( a \ \text{mod} \ b = a \ \text{mod} \ b \)
  note that ultimately show \(?\text{thesis} \)
    by blast
qed

end

Syntactic division remainder operator

class \( \text{modulo} = \text{dvd} + \text{divide} + \)
  fixes \( \text{modulo} :: 'a \Rightarrow 'a \Rightarrow 'a \) \( \text{(infixl mod 70)} \)

Arbitrary quotient and remainder partitions

class \( \text{semiring-modulo} = \text{comm-semiring-1-cancel} + \text{divide} + \text{modulo} + \)
  assumes \( \text{div-mult-mod-eq} : a \ \text{div} \ b \star b + a \ \text{mod} \ b = a \)
begin

lemma mod-div-decomp:
  fixes \( a \) \( b \)
  obtains \( q \) \( r \) where \( q = a \ \text{div} \ b \) and \( r = a \ \text{mod} \ b \)
  and \( a = q \star b + r \)
proof
  from \( \text{div-mult-mod-eq} \) have \( a = a \ \text{div} \ b \star b + a \ \text{mod} \ b \) \( \text{by simp} \)
  moreover have \( a \ \text{div} \ b = a \ \text{div} \ b \)
  moreover have \( a \ \text{mod} \ b = a \ \text{mod} \ b \)
  note that ultimately show \(?\text{thesis} \)
    by blast
qed

lemma mult-div-mod-eq: \( b \star (a \ \text{div} \ b) + a \ \text{mod} \ b = a \)
  using \( \text{div-mult-mod-eq} \) [of \( a \) \( b \)] \( \text{by (simp add: ac-simps)} \)

lemma mod-div-mult-eq: \( a \ \text{mod} \ b + a \ \text{div} \ b \star b = a \)
using `div-mult-mod-eq [of a b]` by (simp add: ac-simps)

lemma `mod-mult-div-eq`: `a mod b + b * (a div b) = a`
using `div-mult-mod-eq [of a b]` by (simp add: ac-simps)

lemma `minus-div-mult-eq-mod`: `a - a div b * b = a mod b`
by (rule add-implies-diff [symmetric]) (fact mod-div-mult-eq)

lemma `minus-mult-div-eq-mod`: `a - b * (a div b) = a mod b`
by (rule add-implies-diff [symmetric]) (fact mod-mult-div-eq)

lemma `minus-mod-eq-div-mult`: `a - a mod b = a div b`
by (rule add-implies-diff [symmetric]) (fact div-mult-mod-eq)

lemma `minus-mod-eq-mult-div`: `a - a mod b = b * (a div b)`
by (rule add-implies-diff [symmetric]) (fact mult-div-mod-eq)

lemma `mod-0-imp-dvd [dest!]`: `b dvd a` if `a mod b = 0`
proof –
have `b dvd (a div b) * b` by simp
also have `(a div b) * b = a`
using `div-mult-mod-eq [of a b]` by (simp add: that)
finally show ?thesis .
qed

lemma `[nitpick-unfold]`: `a mod b = a - a div b * b`
by (fact minus-div-mult-eq-mod [symmetric])

end

14.5 Quotient and remainder in integral domains

class `semidom-modulo = algebraic-semidom + semiring-modulo`
begin

lemma `mod-0 [simp]`: `0 mod a = 0`
using `div-mult-mod-eq [of 0 a]` by simp

lemma `mod-by-0 [simp]`: `a mod 0 = a`
using `div-mult-mod-eq [of a 0]` by simp

lemma `mod-by-1 [simp]`:
`a mod 1 = 0`
proof –
from `div-mult-mod-eq [of a one]` `div-by-1` have `a + a mod 1 = a` by simp
then have `a + a mod 1 = a + 0` by simp
then show ?thesis by (rule add-left-imp-eq)
qed

lemma mod-self [simp]:
  \( a \mod a = 0 \)
  using div-mult-mod-eq [of a a] by simp

lemma dvd-imp-mod-0 [simp]:
  \( b \mod a = 0 \) if \( a \dvd b \)
  using that minus-div-mult-eq-mod [of b a] by simp

lemma mod-eq-0-iff-dvd:
  \( a \mod b = 0 \) ←→ \( b \dvd a \)
  by (auto intro: mod-0-imp-dvd)

lemma dvd-eq-mod-eq-0 [nitpick-unfold, code]:
  \( a \dvd b \) ←→ \( b \mod a = 0 \)
  by (simp add: mod-eq-0-iff-dvd)

lemma dvd-mod-iff:
  assumes c dvd b
  shows c dvd a mod b ←→ c dvd a
  proof –
    from assms have \( c \dvd a \mod b \) ←→ \( c \dvd ((a \div b) \cdot b + a \mod b) \)
      by (simp add: dvd-add-right-iff)
    also have \( (a \div b) \cdot b + a \mod b = a \)
      using div-mult-mod-eq [of a b] by simp
    finally show \?thesis .
  qed

lemma dvd-mod-imp-dvd:
  assumes c dvd a mod b and c dvd b
  shows c dvd a
  using assms dvd-mod-iff [of c b a] by simp

lemma dvd-minus-mod [simp]:
  \( b \dvd a - a \mod b \)
  by (simp add: minus-mod-eq-div-mult)

lemma cancel-div-mod-rules:
  \[ ((a \div b) \cdot b + a \mod b) + c = a + c \]
  \[ (b \cdot (a \div b) + a \mod b) + c = a + c \]
  by (simp-all add: div-mult-mod-eq mult-div-mod-eq)

end

class idom-modulo = idom + semidom-modulo
begin

subclass idom-divide ..
lemma div-diff [simp]:
\[ c \text{ dvd } a \implies c \text{ dvd } b \implies (a - b) \text{ div } c = a \text{ div } c - b \text{ div } c \]
using div-add [of \(- -\)] by (simp add: dvd-neg-div)
end

14.6 Interlude: basic tool support for algebraic and arithmetic calculations

named-theorems arith arith facts -- only ground formulas
ML-file (Tools/arith-data.ML)

ML-file (~~/src/Provers/Arith/cancel-div-mod.ML)

ML
structure Cancel-Div-Mod-Ring = Cancel-Div-Mod
{
  val div-name = const-name \langle divide \rangle;
  val mod-name = const-name \langle modulo \rangle;
  val mk-binop = HOLogic.mk-binop;
  val mk-sum = Arith-Data.mk-sum;
  val dest-sum = Arith-Data.dest-sum;

  val div-mod-eqs = map mk-meta-eq @{
    thms cancel-div-mod-rules\}

  val prove-eq-sums = Arith-Data.prove-conv2 all-tac (Arith-Data.simp-all-tac
    @{
    thms diff-conv-add-uminus add-0-left add-0-right ac-simps\}
  )
}

simp proc-setup cancel-div-mod-int ((a::\langle semidom-modulo \rangle) + b) =
\langle K Cancel-Div-Mod-Ring.proc \rangle

14.7 Ordered semirings and rings

The theory of partially ordered rings is taken from the books:

• Lattice Theory by Garret Birkhoff, American Mathematical Society, 1979
• Partially Ordered Algebraic Systems, Pergamon Press, 1963

Most of the used notions can also be looked up in

• http://www.mathworld.com by Eric Weisstein et. al.
• Algebra I by van der Waerden, Springer
class ordered-semiring = semiring + ordered-comm-monoid-add +
assumes mult-left-mono: a ≤ b ⟹ 0 ≤ c ⟹ c * a ≤ c * b
assumes mult-right-mono: a ≤ b ⟹ 0 ≤ c ⟹ a * c ≤ b * c
begin

lemma mult-mono: a ≤ b ⟹ c ≤ d ⟹ 0 ≤ b ⟹ 0 ≤ c ⟹ a * c ≤ b * d
apply (erule (1) mult-right-mono [THEN order-trans])
apply (erule (1) mult-left-mono)
done

lemma mult-mono': a ≤ b ⟹ c ≤ d ⟹ 0 ≤ a ⟹ 0 ≤ c ⟹ a * c ≤ b * d
by (rule mult-mono) (fast intro: order-trans)+
end

class ordered-semiring-0 = semiring-0 + ordered-semiring
begin

lemma mult-nonneg-nonneg [simp]: 0 ≤ a ⟹ 0 ≤ b ⟹ 0 ≤ a * b
using mult-left-mono [of 0 b a] by simp

lemma mult-nonneg-nonpos: 0 ≤ a ⟹ b ≤ 0 ⟹ a * b ≤ 0
using mult-left-mono [of b 0 a] by simp

lemma mult-nonpos-nonneg: a ≤ 0 ⟹ 0 ≤ b ⟹ a * b ≤ 0
using mult-right-mono [of a 0 b] by simp

Legacy – use mult-nonpos-nonneg.

lemma mult-nonneg-nonpos2: 0 ≤ a ⟹ b ≤ 0 ⟹ b * a ≤ 0
by (drule mult-right-mono [of b 0]) auto

lemma split-mult-neg-le: (0 ≤ a ∧ b ≤ 0) ∨ (a ≤ 0 ∧ 0 ≤ b) ⟹ a * b ≤ 0
by (auto simp add: mult-nonneg-nonpos mult-nonneg-nonpos2)
end

class ordered-cancel-semiring = ordered-semiring + cancel-comm-monoid-add
begin

subclass semiring-0-cancel ..

subclass ordered-semiring-0 ..

end

class linordered-semiring = ordered-semiring + linordered-cancel-ab-semigroup-add
begin

subclass ordered-cancel-semiring ..
subclass ordered-cancel-comm-monoid-add ..

subclass ordered-ab-semigroup-monoid-add-imp-le ..

lemma mult-left-less-imp-less: \( c \cdot a < c \cdot b \Rightarrow 0 \leq c \Rightarrow a < b \)
by (force simp add: mult-left-mono not-le [symmetric])

lemma mult-right-less-imp-less: \( a \cdot c < b \cdot c \Rightarrow 0 \leq c \Rightarrow a < b \)
by (force simp add: mult-right-mono not-le [symmetric])

end

class zero-less-one = order + zero + one +
  assumes zero-less-one [simp]: \( 0 < 1 \)

class linordered-semiring-1 = linordered-semiring + semiring-1 + zero-less-one
begin

lemma convex-bound-le:
  assumes \( x \leq a \) \( y \leq a \) \( 0 \leq u \) \( 0 \leq v \) \( u + v = 1 \)
  shows \( u \cdot x + v \cdot y \leq a \)
proof
  from assms have \( u \cdot x + v \cdot y \leq u \cdot a + v \cdot a \)
    by (simp add: add-mono mult-left-mono)
  with assms show \( ?thesis \)
    unfolding distrib-right [symmetric] by simp
qed

end

class linordered-semiring-strict = semiring + comm-monoid-add + linordered-cancel-ab-semigroup-add +
  assumes mult-strict-left-mono: \( a < b \Rightarrow 0 < c \Rightarrow c \cdot a < c \cdot b \)
  assumes mult-strict-right-mono: \( a < b \Rightarrow 0 < c \Rightarrow a \cdot c < b \cdot c \)
begin

subclass semiring-0-cancel ..

subclass linordered-semiring
proof
  fix \( a \) \( b \) \( c \) :: 'a
  assume \( *: a \leq b \) \( 0 \leq c \)
  then show \( c \cdot a \leq c \cdot b \)
    unfolding le-less
    using mult-strict-left-mono by (cases \( c = 0 \)) auto
from \( * \) show \( a \cdot c \leq b \cdot c \)
  unfolding le-less
  using mult-strict-right-mono by (cases \( c = 0 \)) auto

end
qed

lemma mult-left-le-imp-le: c \cdot a \leq c \cdot b \implies 0 \cdot c \implies a \leq b
  by (auto simp add: mult-strict-left-mono not-less [symmetric])

lemma mult-right-le-imp-le: a \cdot c \leq b \cdot c \implies 0 \cdot c \implies a \leq b
  by (auto simp add: mult-strict-right-mono not-less [symmetric])

lemma mult-pos-pos[simp]: 0 < a \implies 0 < b \implies 0 < a \cdot b
  using mult-strict-left-mono[of 0 b a] by simp

lemma mult-pos-neg: 0 < a \implies b < 0 \implies a \cdot b < 0
  using mult-strict-left-mono[of b 0 a] by simp

lemma mult-neg-pos: a < 0 \implies 0 < b \implies a \cdot b < 0
  using mult-strict-right-mono[of a 0 b] by simp

Legacy – use mult-neg-pos.

lemma mult-pos-neg2: 0 < a \implies b < 0 \implies b \cdot a < 0
  by (drule mult-strict-right-mono[of b 0]) auto

lemma zero-less-mult-pos: 0 < a \cdot b \implies 0 < a \implies 0 < b
  apply (cases b \leq 0)
    apply (auto simp add: le-less not-less)
    apply (drule-tac mult-pos-neg[of a b])
    apply (auto dest: less-not-sym)
  done

lemma zero-less-mult-pos2: 0 < b \cdot a \implies 0 < a \implies 0 < b
  apply (cases b \leq 0)
    apply (auto simp add: le-less not-less)
  apply (drule-tac mult-pos-neg2[of a b])
    apply (auto dest: less-not-sym)
  done

Strict monotonicity in both arguments

lemma mult-strict-mono:
  assumes a < b \and c < d \and 0 < b \and 0 \leq c
  shows a \cdot c < b \cdot d
  using assms
  apply (cases c = 0)
    apply simp
  apply (erule mult-strict-right-mono [THEN less-trans])
    apply (auto simp add: le-less)
  apply (erule (1) mult-strict-left-mono)
  done

This weaker variant has more natural premises

lemma mult-strict-mono':
assumes $a < b$ and $c < d$ and $0 \leq a$ and $0 \leq c$
shows $a \times c < b \times d$
by (rule mult-strict-mono) (insert assms, auto)

lemma mult-less-le-imp-less:
assumes $a < b$ and $c \leq d$ and $0 \leq a$ and $0 < c$
shows $a \times c < b \times d$
using assms
apply (subgoal-tac $a \times c < b \times c$)
apply (erule less-le-trans)
apply (erule mult-left-mono)
apply simp
apply (erule (1) mult-strict-right-mono)
done

lemma mult-le-less-imp-less:
assumes $a \leq b$ and $c < d$ and $0 < a$ and $0 \leq c$
shows $a \times c < b \times d$
using assms
apply (subgoal-tac $a \times c \leq b \times c$)
apply (erule le-less-trans)
apply (erule mult-strict-left-mono)
apply simp
apply (erule (1) mult-right-mono)
done

end

class linordered-semiring-1-strict = linordered-semiring-strict + semiring-1 + zero-less-one
begin
subclass linordered-semiring-1 ..

lemma convex-bound-lt:
assumes $x < a \quad y < a \quad 0 \leq u \quad 0 \leq v \quad u + v = 1$
shows $u \times x + v \times y < a$
proof
from assms have $u \times x + v \times y < u \times a + v \times a$
by (cases $u = 0$) (auto intro!: add-less-le_mono mult-strict-left-mono mult-left-mono)
with assms show ?thesis
unfolding distrib-right[symmetric] by simp
qed

end

class ordered-comm-semiring = comm-semiring-0 + ordered-ab-semigroup-add +
assumes comm-mult-left-mono: $a \leq b \quad \Rightarrow \quad 0 \leq c \quad \Rightarrow \quad c \times a \leq c \times b$
begin
subclass ordered-semiring
proof
  fix a b c :: 'a
  assume a ≤ b 0 ≤ c
  then show c * a ≤ c * b by (rule comm-mult-left-mono)
  then show a * c ≤ b * c by (simp only: mult.commute)
qed

end subclass ordered-cancel-comm-semiring = ordered-comm-semiring + cancel-comm-monoid-add
begin
subclass comm-semiring-0-cancel ..
subclass ordered-comm-semiring ..
subclass ordered-cancel-semiring ..
end subclass ordered-cancel-comm-semiring = ordered-comm-semiring + cancel-comm-monoid-add
begin
fix a b c :: 'a
assume a < b 0 < c
then show c * a < c * b by (rule comm-mult-strict-left-mono)
then show a * c < b * c by (simp only: mult.commute)
qed

subclass linordered-cancel-comm-semiring
proof
fix a b c :: 'a
assume a ≤ b 0 ≤ c
then show c * a ≤ c * b
  unfolding le-less
  using mult-strict-left-mono by (cases c = 0) auto
qed

end subclass linordered-cancel-comm-semiring
proof
fix a b c :: 'a
assume a ≤ b 0 ≤ c
then show c * a ≤ c * b
  unfolding le-less
  using mult-strict-left-mono by (cases c = 0) auto
qed

end subclass linordered-cancel-comm-semiring
proof
fix a b c :: 'a
assume a ≤ b 0 ≤ c
then show c * a ≤ c * b
  unfolding le-less
  using mult-strict-left-mono by (cases c = 0) auto
qed

end subclass ordered-cancel-comm-semiring
proof
fix a b c :: 'a
assume a ≤ b 0 ≤ c
then show c * a ≤ c * b
  unfolding le-less
  using mult-strict-left-mono by (cases c = 0) auto
qed

end subclass linordered-cancel-comm-semiring
proof
fix a b c :: 'a
assume a ≤ b 0 ≤ c
then show c * a ≤ c * b
  unfolding le-less
  using mult-strict-left-mono by (cases c = 0) auto
qed

end subclass linear-cancel-comm-semiring
proof
fix a b c :: 'a
assume a ≤ b 0 ≤ c
then show c * a ≤ c * b
  unfolding le-less
  using mult-strict-left-mono by (cases c = 0) auto
qed

end subclass linear-cancel-comm-semiring
proof
fix a b c :: 'a
assume a ≤ b 0 ≤ c
then show c * a ≤ c * b
  unfolding le-less
  using mult-strict-left-mono by (cases c = 0) auto
qed

end subclass ordered-ring = ring + ordered-cancel-semiring
begin
subclass ordered-ab-group-add ..
lemma less-add-iff1: \(a \cdot e + c < b \cdot e + d \iff (a - b) \cdot e + c < d\)
by (simp add: algebra-simps)

lemma less-add-iff2: \(a \cdot e + c < b \cdot e + d \iff c < (b - a) \cdot e + d\)
by (simp add: algebra-simps)

lemma le-add-iff1: \(a \cdot e + c \leq b \cdot e + d \iff (a - b) \cdot e + c \leq d\)
by (simp add: algebra-simps)

lemma le-add-iff2: \(a \cdot e + c \leq b \cdot e + d \iff c \leq (b - a) \cdot e + d\)
by (simp add: algebra-simps)

lemma mult-left-mono-neg: \(b \leq a \implies c \leq 0 \implies c \cdot a \leq c \cdot b\)
apply (drule mult-left-mono [of - - c])
apply simp-all
done

lemma mult-right-mono-neg: \(b \leq a \implies c \leq 0 \implies a \cdot c \leq b \cdot c\)
apply (drule mult-right-mono [of - - c])
apply simp-all
done

lemma mult-nonpos-nonpos: \(a \leq 0 \implies b \leq 0 \implies 0 \leq a \cdot b\)
using mult-right-mono-neg [of a 0 b] by simp

lemma split-mult-pos-le: \((0 \leq a \land 0 \leq b) \lor (a \leq 0 \land b \leq 0) \implies 0 \leq a \cdot b\)
by (auto simp add: mult-nonpos-nonpos)
end

class abs-if = minus + uminus + ord + zero + abs +
assumes abs-if: \(|a| = (if a < 0 then - a else a)\)

class linordered-ring = ring + linordered-semiring + linordered-ab-group-add + abs-if
begin
subclass ordered-ring ..

subclass ordered-ab-group-add-abs
proof
fix a b
show \(|a + b| \leq |a| + |b|\)
  by (auto simp add: abs-if not-le not-less algebra-simps
           simp del: add.commute dest: add-neg-neg add-nonneg-nonneg)
qed (auto simp: abs-if)

lemma zero-le-square [simp]: \(0 \leq a \cdot a\)
using linear [of 0 a] by (auto simp add: mult-nonpos-nonpos)

lemma not-square-less-zero [simp]: ¬ (a * a < 0)
by (simp add: not-less)

proposition abs-eq-iff: |x| = |y| ⟷ x = y ∨ x = −y
by (auto simp add: abs-if split: if-split-asm)

lemma abs-eq-iff':
|a| = b ⟷ b ≥ 0 ∧ (a = b ∨ a = − b)
by (cases a ≥ 0) auto

lemma eq-abs-iff':
a = |b| ⟷ a ≥ 0 ∧ (b = a ∨ b = − a)
using abs-eq-iff' [of b a] by auto

lemma sum-squares-ge-zero: 0 ≤ x * x + y * y
by (intro add-nonneg-nonneg zero-le-square)

lemma not-sum-squares-lt-zero: ¬ x * x + y * y < 0
by (simp add: not-less sum-squares-ge-zero)

end

class linordered-ring-strict = ring + linordered-semiring-strict
  + ordered-ab-group-add + abs-if
begin

subclass linordered-ring ..

lemma mult-strict-left-mono-neg: b < a ⟹ c < 0 ⟹ c * a < c * b
using mult-strict-left-mono [of b a − c] by simp

lemma mult-strict-right-mono-neg: b < a ⟹ c < 0 ⟹ a * c < b * c
using mult-strict-right-mono [of b a − c] by simp

lemma mult-neg-neg: a < 0 ⟹ b < 0 ⟹ 0 < a * b
using mult-strict-right-mono-neg [of a 0 b] by simp

subclass ring-no-zero-divisors

proof
  fix a b

  assume a ≠ 0
  then have a: a < 0 ∨ 0 < a by (simp add: neg-iff)
  assume b ≠ 0
  then have b: b < 0 ∨ 0 < b by (simp add: neg-iff)
  have a * b < 0 ∨ 0 < a * b
    proof (cases a < 0)
      case True
    qed
  qed
show \( \text{thesis} \)
proof (cases \( b < 0 \))
case True
  with \( a < 0 \) show \( \text{thesis} \) by (auto dest: mult-neg-neg)
next
case False
  with \( b \) have \( 0 < b \) by auto
  with \( a < 0 \) show \( \text{thesis} \) by (auto dest: mult-strict-right-mono)
qed
next
case False
  with \( a \) have \( 0 < a \) by auto
  show \( \text{thesis} \) proof (cases \( b < 0 \))
case True
    with \( 0 < a \) show \( \text{thesis} \) by (auto dest: mult-strict-right-mono-neg)
  qed
next
case False
  with \( b \) have \( 0 < b \) by auto
  with \( 0 < a \) show \( \text{thesis} \) by auto
qed
then show \( a \cdot b \neq 0 \)
  by (simp add: neq_iff)
qed

lemma \text{zero-less-mult-iff} \ [algebra-split-simps, field-split-simps]:
\( 0 < a \cdot b \leftrightarrow 0 < a \land 0 < b \lor a < 0 \land b < 0 \)
by (cases \( a 0 b 0 \) rule: linorder-cases\[case-product linorder-cases]\)
(auto simp add: mult-neg-neg not-less le-less dest: zero-less-mult-pos zero-less-mult-pos2)

lemma \text{zero-le-mult-iff} \ [algebra-split-simps, field-split-simps]:
\( 0 \leq a \cdot b \leftrightarrow 0 \leq a \land 0 \leq b \lor a \leq 0 \land b \leq 0 \)
by (auto simp add: eq-commute \[of 0\] le-less not-less zero-less-mult-iff)

lemma \text{mult-less-0-iff} \ [algebra-split-simps, field-split-simps]:
\( a \cdot b < 0 \leftrightarrow 0 < a \land b < 0 \lor a < 0 \land 0 < b \)
using \text{zero-less-mult-iff} \ [of \( \cdot a b \)] by auto

lemma \text{mult-le-0-iff} \ [algebra-split-simps, field-split-simps]:
\( a \cdot b \leq 0 \leftrightarrow 0 \leq a \land b \leq 0 \lor a \leq 0 \land 0 \leq b \)
using \text{zero-le-mult-iff} \ [of \( \cdot a b \)] by auto

Cancellation laws for \( c \cdot a < c \cdot b \) and \( a \cdot c < b \cdot c \), also with the relations \( \leq \) and equality.

These “disjunction” versions produce two cases when the comparison is an assumption, but effectively four when the comparison is a goal.
lemma mult-less-cancel-right-disj: \( a \cdot c < b \cdot c \leftrightarrow 0 < c \land a < b \lor c < 0 \land b < a \)
apply (cases \( c = 0 \))
apply (auto simp add: neq-iff mult-strict-right-mono mult-strict-right-mono-neg)
apply (auto simp add: not-less not-le [symmetric, of a\(\cdot c\)] not-le [symmetric, of a])
apply (erule-tac [!] notE)
apply (auto simp add: less-imp-le mult-right-mono mult-right-mono-neg)
done

lemma mult-less-cancel-left-disj: \( c \cdot a < c \cdot b \leftrightarrow 0 < c \land a < b \lor c < 0 \land b < a \)
apply (cases \( c = 0 \))
apply (auto simp add: neq-iff mult-strict-left-mono mult-strict-left-mono-neg)
apply (auto simp add: not-less not-le [symmetric, of c\(\cdot a\)] not-le [symmetric, of a])
apply (erule-tac [!] notE)
apply (auto simp add: less-imp-le mult-left-mono mult-left-mono-neg)
done

The “conjunction of implication” lemmas produce two cases when the comparison is a goal, but give four when the comparison is an assumption.

lemma mult-less-cancel-right: \( a \cdot c < b \cdot c \leftrightarrow (0 \leq c \rightarrow a < b) \land (c \leq 0 \rightarrow b < a) \)
using mult-less-cancel-right-disj [of a c b] by auto

lemma mult-less-cancel-left: \( c \cdot a < c \cdot b \leftrightarrow (0 \leq c \rightarrow a < b) \land (c \leq 0 \rightarrow b < a) \)
using mult-less-cancel-left-disj [of c a b] by auto

lemma mult-le-cancel-right: \( a \cdot c \leq b \cdot c \leftrightarrow (0 < c \rightarrow a \leq b) \land (c < 0 \rightarrow b \leq a) \)
by (simp add: not-less [symmetric] mult-less-cancel-right-disj)

lemma mult-le-cancel-left: \( c \cdot a \leq c \cdot b \leftrightarrow (0 < c \rightarrow a \leq b) \land (c < 0 \rightarrow b \leq a) \)
by (simp add: not-less [symmetric] mult-less-cancel-left-disj)

lemma mult-le-cancel-left-pos: \( 0 < c \Longrightarrow c \cdot a \leq c \cdot b \leftrightarrow a \leq b \)
by (auto simp: mult-le-cancel-left)

lemma mult-le-cancel-left-neg: \( c < 0 \Longrightarrow c \cdot a \leq c \cdot b \leftrightarrow b \leq a \)
by (auto simp: mult-le-cancel-left)

lemma mult-le-cancel-left-pos: \( 0 < c \Longrightarrow c \cdot a < c \cdot b \leftrightarrow a < b \)
by (auto simp: mult-le-cancel-left)

lemma mult-le-cancel-left-neg: \( c < 0 \Longrightarrow c \cdot a < c \cdot b \leftrightarrow b < a \)
by (auto simp: mult-le-cancel-left)
THEORY "Rings"

end

lemmas mult-sign-intros =
  mult-nonneg-nonneg mult-nonneg-nonpos
  mult-nonneg-nonpos mult-nonneg-nonpos
  mult-pos-pos mult-pos-neg
  mult-neg-pos mult-neg-neg

class ordered-comm-ring = comm-ring + ordered-comm-semiring
begin
subclass ordered-ring ..
subclass ordered-cancel-comm-semiring ..
end

class linordered-nonzero-semiring = ordered-comm-semiring + monoid-mult +
  linorder + zero-less-one +
  assumes add-mono1: a < b ==> a + 1 < b + 1
begin
subclass zero-neq-one
  by standard (insert zero-less-one, blast)
subclass comm-semiring-1
  by standard (rule mult-1-left)
lemma zero-le-one [simp]: 0 <= 1
  by (rule zero-less-one [THEN less-imp-le])
lemma not-one-le-zero [simp]: ~1 <= 0
  by (simp add: not-le)
lemma not-one-less-zero [simp]: ~1 < 0
  by (simp add: not-less)
lemma mult-left-le: c <= 1 ==> 0 <= a ==> a * c <= a
  using mult-left-mono[of c 1 a] by simp
lemma mult-le-one: a <= 1 ==> 0 <= b ==> b <= 1 ==> a * b <= 1
  using mult-mono[of a 1 b 1] by simp
lemma zero-less-two: 0 < 1 + 1
  using add-pos-pos[of 0 zero-less-one zero-less-one] ..
end

class linordered-semidom = semidom + linordered-comm-semiring-strict + zero-less-one
assumes le-add-diff-inverse2 [simp]: \( b \leq a \implies a - b + b = a \)

begin

subclass linordered-nonzero-semiring
proof
  show \( a + 1 < b + 1 \) if \( a < b \) for \( a, b \)
  proof (rule ccontr, simp add: not-less)
    assume \( b \leq a \)
    with that show False
    by (simp add: )
  qed
qed

Addition is the inverse of subtraction.

lemma le-add-diff-inverse [simp]: \( b \leq a \implies b + (a - b) = a \)
  by (frule le-add-diff-inverse2) (simp add: add.commute)

lemma add-diff-inverse: \( \neg a < b \implies b + (a - b) = a \)
  by simp

lemma add-le-imp-le-diff: \( i + k \leq n \implies i \leq n - k \)
  apply (subst add-le-cancel-right [where \( c=k \), symmetric])
  apply (frule le-add-diff-inverse2)
  apply (simp only: add.assoc [symmetric])
  using add-implies-diff
  apply fastforce
  done

lemma add-le-add-imp-diff-le:
  assumes 1: \( i + k \leq n \)
  and 2: \( n \leq j + k \)
  shows \( i + k \leq n \implies n \leq j + k \implies n - k \leq j \)
proof -
  have \( n - (i + k) + (i + k) = n \)
    using 1 by simp
  moreover have \( n - k = n - k - i + i \)
    using 1 by (simp add: add-le-add-imp-diff)
  ultimately show \( \text{thesis} \)
  using 2
    apply (simp add: add.assoc [symmetric])
    apply (rule add-le-imp-le-diff [of \( k, j + k \), simplified add-diff-cancel-right])
    apply (simp add: add.commute diff-diff-add)
  done
qed

lemma less-1-mult: \( 1 < m \implies 1 < n \implies 1 < m \cdot n \)
  using mult-strict-mono [of \( 1 \) \( m \) \( n \)] by (simp add: less-trans [OF zero-less-one])
end

class linordered-idom = comm-ring-1 + linordered-comm-semiring-strict +
    ordered-ab-group-add + abs-if + sgn +
assumes sgn-if: sgn x = (if x = 0 then 0 else if 0 < x then 1 else −1)
begin

subclass linordered-ring-strict ..
subclass linordered-semiring-strict
proof
  have 0 ≤ 1 * 1
    by (fact zero-le-square)
  then show 0 < 1
    by (simp add: le-less)
qed

subclass ordered-comm-ring ..
subclass idom ..

subclass linordered-semidom
  by standard simp

subclass idom-abs-sgn
  by standard
    (auto simp add: sgn-if abs-if zero-less-mult-iff)

lemma linorder-neqE-linordered-idom:
  assumes x ≠ y
  obtains x < y | y < x
  using assms by (rule neqE)

These cancellation simp rules also produce two cases when the comparison
is a goal.

lemma mult-le-cancel-right1: c ≤ b * c ←→ (0 < c → 1 ≤ b) ∧ (c < 0 → b ≤ 1)
  using mult-le-cancel-right [of 1 c b] by simp

lemma mult-le-cancel-right2: a * c ≤ c ←→ (0 < c → a ≤ 1) ∧ (c < 0 → 1 ≤ a)
  using mult-le-cancel-right [of a c 1] by simp

lemma mult-le-cancel-left1: c ≤ c * b ←→ (0 < c → 1 ≤ b) ∧ (c < 0 → b ≤ 1)
  using mult-le-cancel-left [of c 1 b] by simp

lemma mult-le-cancel-left2: c * a ≤ c ←→ (0 < c → a ≤ 1) ∧ (c < 0 → 1 ≤ a)
  using mult-le-cancel-left [of c a 1] by simp
theory "Rings"

lemma mult-less-cancel-right1: \( c < b \cdot c \leftrightarrow (0 \leq c \rightarrow 1 < b) \land (c \leq 0 \rightarrow b < 1) \)
using mult-less-cancel-right [of 1 c b] by simp

lemma mult-less-cancel-right2: \( a \cdot c < c \leftrightarrow (0 \leq c \rightarrow a < 1) \land (c \leq 0 \rightarrow 1 < a) \)
using mult-less-cancel-right [of a c 1] by simp

lemma mult-less-cancel-left1: \( c < c \cdot b \leftrightarrow (0 \leq c \rightarrow 1 < b) \land (c \leq 0 \rightarrow b < 1) \)
using mult-less-cancel-left [of c 1 b] by simp

lemma mult-less-cancel-left2: \( c \cdot a < c \leftrightarrow (0 \leq c \rightarrow a < 1) \land (c \leq 0 \rightarrow 1 < a) \)
using mult-less-cancel-left [of c a 1] by simp

lemma sgn-0-0: \( \text{sgn } a = 0 \leftrightarrow a = 0 \)
by (fact sgn-eq-0-iff)

lemma sgn-1-pos: \( \text{sgn } a = 1 \leftrightarrow a > 0 \)
unfolding sgn-if by simp

lemma sgn-1-neg: \( \text{sgn } a = -1 \leftrightarrow a < 0 \)
unfolding sgn-if by auto

lemma sgn-pos [simp]: \( 0 < a \Longrightarrow \text{sgn } a = 1 \)
by (simp only: sgn-1-pos)

lemma sgn-neg [simp]: \( a < 0 \Longrightarrow \text{sgn } a = -1 \)
by (simp only: sgn-1-neg)

lemma abs-sgn: \( |k| = k * \text{sgn } k \)
unfolding sgn-if abs-if by auto

lemma sgn-greater [simp]: \( 0 < \text{sgn } a \leftrightarrow 0 < a \)
unfolding sgn-if by auto

lemma sgn-less [simp]: \( \text{sgn } a < 0 \leftrightarrow a < 0 \)
unfolding sgn-if by auto

lemma abs-sgn-eq-1 [simp]:
a \neq 0 \Longrightarrow |\text{sgn } a| = 1
by simp

lemma abs-sgn-eq: \( |\text{sgn } a| = (\text{if } a = 0 \text{ then } 0 \text{ else } 1) \)
by (simp add: sgn-if)

lemma sgn-mult-self-eq [simp]:
sgn a * sgn a = of-bool (a ≠ 0)
by (cases a > 0) simp-all

lemma abs-mult-self-eq [simp]:
|a| * |a| = a * a
by (cases a > 0) simp-all

lemma same-sgn-sgn-add:
sgn (a + b) = sgn a if sgn b = sgn a
proof (cases a 0 rule: linorder-cases)
case equal
with that show ?thesis
  by simp
next
case less
with that have b < 0
  by (simp add: sgn-1-neg)
with ⟨a < 0⟩ have a + b < 0
  by (rule add-neg-neg)
with ⟨a < 0⟩ show ?thesis
  by simp
next
case greater
with that have b > 0
  by (simp add: sgn-1-pos)
with ⟨a > 0⟩ have a + b > 0
  by (rule add-pos-pos)
with ⟨a > 0⟩ show ?thesis
  by simp
qed

lemma same-sgn-abs-add:
|a + b| = |a| + |b| if sgn b = sgn a
proof –
  have a + b = sgn a * |a| + sgn b * |b|
    by (simp add: sgn-mult-abs)
  also have ... = sgn a * (|a| + |b|)
    using that by (simp add: algebra-simps)
finally show ?thesis
  by (auto simp add: abs-mult)
qed

lemma sgn-not-eq-imp:
sgn a = −sgn b if sgn b ≠ sgn a and sgn a ≠ 0 and sgn b ≠ 0
using that by (cases a < 0) (auto simp add: sgn-0-0 sgn-1-pos sgn-1-neg)

lemma abs-dvd-iff [simp]: |m| dvd k ↔ m dvd k
by (simp add: abs-if)
lemma dvd-abs-iff [simp]: \( m \mid k \iff m \mid d \cdot k \)
by (simp add: abs-if)

lemma dvd-if-abs-eq [simp]: \( l \mid l \iff l \mid k \)
by (subst abs-dvd-iff [symmetric]) simp

The following lemmas can be proven in more general structures, but are dangerous as simp rules in absence of (\(- ?a = ?a = (\theta::'a)\), (\(- ?a < ?a) = ((\theta::'a) < ?a), (\(- ?a \leq ?a) = ((\theta::'a) \leq ?a).

lemma equation-minus-iff-1 [simp, no-atp]: \( 1 = - a \iff a = - 1 \)
by (fact equation-minus-iff)

lemma minus-equation-iff-1 [simp, no-atp]: \(- a = 1 \iff a = - 1 \)
by (subst minus-equation-iff, auto)

lemma le-minus-iff-1 [simp, no-atp]: \( 1 \leq - b \iff b \leq - 1 \)
by (fact le-minus-iff)

lemma minus-le-iff-1 [simp, no-atp]: \(- a \leq 1 \iff - 1 \leq a \)
by (fact minus-le-iff)

lemma less-minus-iff-1 [simp, no-atp]: \( 1 < - b \iff b < - 1 \)
by (fact less-minus-iff)

lemma minus-less-iff-1 [simp, no-atp]: \(- a < 1 \iff - 1 < a \)
by (fact minus-less-iff)

lemma add-less-zeroD:
shows \( x + y < 0 \Longrightarrow x < 0 \land y < 0 \)
by (auto simp: not-less intro: le-less-trans [of - x + y])

end

Reasoning about inequalities with division

context linordered-semidom
begin

lemma less-add-one: \( a < a + 1 \)
proof
  have \( a + 0 < a + 1 \)
  by (blast intro: zero-less-one add-strict-left-mono)
  then show \?thesis by simp
qed

end

context linordered-idom
begin
lemma mult-right-le-one-le: $0 \leq x \Rightarrow 0 \leq y \Rightarrow y \leq 1 \Rightarrow x \cdot y \leq x$
  by (rule mult-left-le)

lemma mult-left-le-one-le: $0 \leq x \Rightarrow 0 \leq y \Rightarrow y \leq 1 \Rightarrow y \cdot x \leq x$
  by (auto simp add: mult-cancel-right2)

end

Absolute Value

context linordered-idom
begin

lemma mult-sgn-abs: $\sgn x \cdot \vert x \vert = x$
  by (fact sgn-mult-abs)

lemma abs-one: $\vert 1 \vert = 1$
  by (fact abs-1)

end

class ordered-ring-abs = ordered-ring + ordered-ab-group-add-abs +
  assumes abs-eq-mult:
    $(0 \leq a \lor a \leq 0) \land (0 \leq b \lor b \leq 0) \Rightarrow \vert a \cdot b \vert = \vert a \vert \cdot \vert b \vert$

context linordered-idom
begin

subclass ordered-ring-abs
  by standard (auto simp: abs-if not-less mult-less-0-iff)

lemma abs-mult-self: $\vert a \vert \cdot \vert a \vert = a \cdot a$
  by (fact abs-mult-self-eq)

lemma abs-mult-less:
  assumes ac: $\vert a \vert < c$
    and bd: $\vert b \vert < d$
  shows $\vert a \vert \cdot \vert b \vert < c \cdot d$

proof -
  from ac have $0 < c$
    by (blast intro: le-less-trans abs-ge-zero)
  with bd show ?thesis by (simp add: ac mult-strict-mono)
  qed

lemma abs-less-iff: $\vert a \vert < b \iff a < b \land -a < b$
  by (simp add: less-le abs-le-iff) (auto simp add: abs-if)

lemma abs-mult-pos: $0 \leq x \Rightarrow \vert y \vert \cdot x = \vert y \cdot x \vert$
  by (simp add: abs-mult)
**THEORY “Nat”**

**lemma** abs-diff-less-iff: $|x - a| < r \iff a - r < x < a + r$
by (auto simp add: diff-less_eq ac-simps abs-less_iff)

**lemma** abs-diff-le-iff: $|x - a| \leq r \iff a - r \leq x \land x \leq a + r$
by (auto simp add: diff-le_eq ac-simps abs-le_iff)

**lemma** abs-add-one-gt-zero: $0 < 1 + |x|$
by (auto simp: abs-if not-less intro: zero-less_one add-strict-increasing less-trans)

end

14.8 Dioids

Dioids are the alternative extensions of semirings, a semiring can either be a ring or a dioid but never both.

**class** dioid = semiring_1 + canonically_ordered_monoid_add
begin

subclass ordered_semiring
by standard (auto simp: le_iff_add distrib_left distrib_right)

end

hide-fact (open) comm_mult_left_mono comm_mult_strict_left_mono distrib

code-identifier
code-module Rings -> (SML) Arith and (OCaml) Arith and (Haskell) Arith

end

15 Natural numbers

theory Nat
imports Inductive Typedef Fun Rings
begin

15.1 Type ind
typedecl ind

axiomatization Zero-Rep :: ind and Suc-Rep :: ind ⇒ ind
— The axiom of infinity in 2 parts:
where Suc-Rep-inject: Suc-Rep x = Suc-Rep y ⇔ x = y
15.2 Type nat

Type definition

```
inductive Nat :: ind ⇒ bool
where
  Zero-RepI : Nat Zero-Rep
  | Suc-RepI : Nat i ⇒ Nat (Suc-Rep i)

typedef nat = { n. Nat n }
  morphisms Rep-Nat Abs-Nat
  using Nat.Zero-RepI by auto

lemma Nat-Rep-Nat: Nat (Rep-Nat n)
  using Rep-Nat by simp

lemma Nat-Abs-Nat-inverse: Nat n ⇒ Rep-Nat (Abs-Nat n) = n
  using Abs-Nat-inverse by simp

lemma Nat-Abs-Nat-inject: Nat n ⇒ Nat m ⇒ Abs-Nat n = Abs-Nat m ↔ n = m
  using Abs-Nat-inject by simp

instantiation nat :: zero
begin

definition Zero-nat-def: 0 = Abs-Nat Zero-Rep

instance ..
end

definition Suc :: nat ⇒ nat
where Suc n = Abs-Nat (Suc-Rep (Rep-Nat n))

lemma Suc-not-Zero: Suc m ≠ 0
  by (simp add: Zero-nat-def Suc-def Suc-RepI Zero-RepI

lemma Zero-not-Suc: 0 ≠ Suc m
  by (rule not-sym) (rule Suc-not-Zero)

lemma Suc-Rep-inject': Suc-Rep x = Suc-Rep y ⟷ x = y
  by (rule iffI, rule Suc-Rep-inject) simp-all

lemma nat-induct0:
  assumes P 0
  and ∀n. P n ⇒ P (Suc n)
  shows P n
  using assms
apply (unfold Zero-nat-def Suc-def)
apply (rule Rep-Nat-inverse [THEN subst]) — types force good instantiation
apply (erule Nat-Rep-Nat [THEN Nat.induct])
apply (iprover elim: Nat-Abs-Nat-inverse [THEN subst])
done

free-constructors case-nat for 0 :: nat | Suc pred
where pred (0 :: nat) = (0 :: nat)
apply atomize-elim
  apply (rename-tac n, induct-tac n rule: nat-induct0, auto)
  Rep-Nat-inject)
apply (simp only: Suc-not-Zero)
done

— Avoid name clashes by prefixing the output of old-rep-datatype with old.
setup (Sign.mandatory-path old)

old-rep-datatype 0 :: nat Suc
  apply (erule nat-induct0)
  apply assumption
  apply (rule nat.inject)
  apply (rule nat.distinct(1))
done

setup (Sign.parent-path)

— But erase the prefix for properties that are not generated by free-constructors.
setup (Sign.mandatory-path nat)

declare old.nat.inject[iff del]
  and old.nat.distinct(1)[simp del, induct-simp del]

lemmas induct = old.nat.induct
lemmas inducts = old.nat.inducts
lemmas rec = old.nat.rec
lemmas simps = nat.inject nat.distinct nat.case nat.rec

setup (Sign.parent-path)

abbreviation rec-nat :: 'a ⇒ (nat ⇒ 'a ⇒ 'a) ⇒ nat ⇒ 'a
where rec-nat ≡ old.rec-nat

declare nat.sel[code del]

hide-const (open) Nat.pred — hide everything related to the selector
hide-fact
  nat.case-eq-if
  nat.collapse
THEORY “Nat”

nat.expand
nat.sel
nat.exhaust.sel
nat.split.sel
nat.split-sel-asn

lemma nat-exhaust [case-names 0 Suc, cases type: nat]:
(y = 0 ⇒ P) ⇒ (∀n. y = Suc n ⇒ P) ⇒ P
— for backward compatibility – names of variables differ
by (rule old.nat.exhaust)

lemma nat-induct [case-names 0 Suc, induct type: nat]:
fixes n
assumes P 0 and \( \land n. P n \Rightarrow P (Suc n) \)
shows P n
— for backward compatibility – names of variables differ
using assms by (rule nat.induct)

hide-fact
nat-exhaust
nat-induct0

ML

val nat-basic-lfp-sugar =
let
  val ctr-sugar = the (Ctr-Sugar.ctr-sugar-of-global theory type-name ⟨nat⟩);
  val recx = Logic.varify-types-global term ⟨rec-nat⟩;
  val C = body-type (fastype-of recx);
  in
  { T = HOLogic.natT, fp-res-index = 0, C = C, fun-arg-Tss = [[], [[HOLogic.natT, C]]],
    ctr-sugar = ctr-sugar, recx = recx, rec-thms = @{thms nat.rec}}
end;

setup

fun basic-lfp-sugars-of - [typ : nat] - - ctxt =
  ([], [0], [nat-basic-lfp-sugar], [], [], [], TrueI (*dummy*), [], false, ctxt)
  | basic-lfp-sugars-of bs arg-Ts callers calssss ctxt =
    BNF-LFP-Rec-Sugar.default-basic-lfp-sugars-of bs arg-Ts callers calssss ctxt;
  in
  BNF-LFP-Rec-Sugar.register-lfp-rec-extension
  { nested-simps = [], special-endgame-tac = K (K (K no-tac))), is-new-datatype
    = K (K true),
    basic-lfp-sugars-of = basic-lfp-sugars-of, rewrite-nested-rec-call = NONE}
end

Injectiveness and distinctness lemmas
lemma inj-Suc [simp]:
inj-on Suc N
by (simp add: inj-on-def)

lemma bij-betw-Suc [simp]:
bij-betw Suc M N ⇔ Suc ' M = N
by (simp add: bij-betw-def)

lemma Suc-neq-Zero: Suc m = 0 → R
by (rule notE) (rule Suc-not-Zero)

lemma Zero-neq-Suc: 0 = Suc m → R
by (rule Suc-neq-Zero) (erule sym)

lemma Suc-inject:
Suc x = Suc y → x = y
by (rule inj-Suc [THEN injD])

lemma n-not-Suc-n:
\(n \neq Suc n\)
by (induct n) simp-all

lemma Suc-n-not-n:
\(Suc n \neq n\)
by (rule not-sym) (rule n-not-Suc-n)

A special form of induction for reasoning about \(m < n\) and \(m - n\).

lemma diff-induct:
assumes \(⋀x. P x 0\)
and \(⋀y. P 0 (Suc y)\)
and \(⋀x y. P x y \Rightarrow P (Suc x) (Suc y)\)
shows \(P m n\)
proof (induct n arbitrary: m)
case 0
show ?case by (rule assms(1))
next
case (Suc n)
show ?case
proof (induct m)
case 0
show ?case by (rule assms(2))
next
case (Suc m)
from \(P m n\) show ?case by (rule assms(3))
qed

15.3 Arithmetic operators

instantiation nat :: comm-monoid-diff
begin
primrec plus-nat
where
  add-0: 0 + n = (n::nat)
  | add-Suc: Suc m + n = Suc (m + n)

lemma add-0-right [simp]: m + 0 = m
  for m :: nat
  by (induct m) simp-all

lemma add-Suc-right [simp]: m + Suc n = Suc (m + n)
  by (induct m) simp-all

declare add-0 [code]

lemma add-Suc-shift [code]: Suc m + n = m + Suc n
  by simp

primrec minus-nat
where
  diff-0 [code]: m - 0 = (m::nat)
  | diff-Suc: m - Suc n = (case m - n of 0 ⇒ 0 | Suc k ⇒ k)

declare diff-Suc [simp del]

lemma diff-0-eq-0 [simp, code]: 0 - n = 0
  for n :: nat
  by (induct n) (simp-all add: diff-Suc)

lemma diff-Suc-Suc [simp, code]: Suc m - Suc n = m - n
  by (induct n) (simp-all add: diff-Suc)

instance
proof
  fix n m q :: nat
  show (n + m) + q = n + (m + q) by (induct n) simp-all
  show n + m = m + n by (induct n) simp-all
  show m + n - m = n by (induct m) simp-all
  show n - m - q = n - (m + q) by (induct q) (simp-all add: diff-Suc)
  show 0 + n = n by simp
  show 0 - n = 0 by simp
qed

end

hide-fact (open) add-0 add-0-right diff-0

instantiation nat :: comm-semiring-1-cancel
begin
\textbf{definition} One-nat-def [simp]: 1 = Suc 0

\textbf{primrec} times-nat
\begin{itemize}
\item \textit{mult-0}: 0 \ast n = (0::nat)
\item \textit{mult-Suc}: Suc m \ast n = n + (m \ast n)
\end{itemize}

\textbf{lemma} mult-0-right [simp]: m \ast 0 = 0
\begin{itemize}
\item for m :: nat
\item by (induct m) simp-all
\end{itemize}

\textbf{lemma} mult-Suc-right [simp]: m \ast Suc n = m + (m \ast n)
\begin{itemize}
\item by (induct m) (simp-all add: add.left-commute)
\end{itemize}

\textbf{lemma} add-mult-distrib: (m + n) \ast k = (m \ast k) + (n \ast k)
\begin{itemize}
\item for m n k :: nat
\item by (induct m) (simp-all add: add.mult-distrib)
\end{itemize}

\textbf{instance}
\textbf{proof}
\begin{itemize}
\item fix k n m q :: nat
\item show 0 \neq (1::nat)
\item by simp
\item show 1 + n = n
\item by simp
\item show n + m = m + n
\item by (induct n) simp-all
\item show (n + m) + q = n + (m + q)
\item by (induct n) (simp-all add: add-mult-distrib)
\item show (n + m) + q = n + q + m + q
\item by (rule add-mult-distrib)
\item show k * (m - n) = (k * m) - (k * n)
\item by (induct m n rule: diff-induct) simp-all
\end{itemize}
\textbf{qed}

\textbf{end}

15.3.1 Addition

Reasoning about \( m + 0 = 0 \), etc.

\textbf{lemma} add-is-0 [iff]: m + n = 0 \iff m = 0 \land n = 0
\begin{itemize}
\item for m n :: nat
\item by (cases m) simp-all
\end{itemize}

\textbf{lemma} add-is-1: m + n = Suc 0 \iff m = Suc 0 \land n = 0 \lor m = 0 \land n = Suc 0
\begin{itemize}
\item by (cases m) simp-all
\end{itemize}

\textbf{lemma} one-is-add: Suc 0 = m + n \iff m = Suc 0 \land n = 0 \lor m = 0 \land n =
Suc 0
by (rule trans, rule eq-commute, rule add-is-1)

lemma add-eq-self-zero: m + n = m ⇒ n = 0
for m n :: nat
by (induct m) simp-all

lemma plus-1-eq-Suc:
  plus 1 = Suc
by (simp add: fan-eq-iff)

lemma Suc-eq-plus1: Suc n = n + 1
by simp

lemma Suc-eq-plus1-left: Suc n = 1 + n
by simp

15.3.2 Difference

lemma Suc-diff-diff [simp]: (Suc m − n) − Suc k = m − n − k
by (simp add: diff-diff-add)

lemma diff-Suc-1 [simp]: Suc n − 1 = n
by simp

15.3.3 Multiplication

lemma mult-is-0 [simp]: m * n = 0 ↔ m = 0 ∨ n = 0 for m n :: nat
by (induct m) auto

lemma mult-eq-1-iff [simp]: m * n = Suc 0 ↔ m = Suc 0 ∧ n = Suc 0
proof (induct m)
  case 0
  then show ?case by simp
next
  case (Suc m)
  then show ?case by (induct n) auto
qed

lemma one-eq-mult-iff [simp]: Suc 0 = m * n ↔ m = Suc 0 ∧ n = Suc 0
apply (rule trans)
apply (rule-tac [2] mult-eq-1-iff)
apply fastforce
done

lemma nat-mul-eq-1-iff [simp]: m * n = 1 ↔ m = 1 ∧ n = 1
for m n :: nat
unfolding One-nat-def by (rule mult-eq-1-iff)

lemma nat-1-eq-mult-iff [simp]: 1 = m * n ↔ m = 1 ∧ n = 1
for \( m \) \( n \) :: nat
unfolding One-nat-def by (rule one-eq-mult-iff)

lemma mult-cancel1 [simp]: \( k \cdot m = k \cdot n \iff m = n \lor k = 0 \)
for \( k \) \( m \) \( n \) :: nat
proof -
  have \( k \neq 0 \Longrightarrow k \cdot m = k \cdot n \Longrightarrow m = n \)
  proof (induct n arbitrary: \( m \))
    case 0
    then show \( m = 0 \) by simp
  next
    case (Suc \( n \))
    then show \( m = \text{Suc} n \)
    by (cases \( m \)) (simp-all add: eq-commute [of 0])
  qed
  then show ?thesis by auto
  qed

lemma mult-cancel2 [simp]: \( m \cdot k = n \cdot k \iff m = n \lor k = 0 \)
for \( k \) \( m \) \( n \) :: nat
by (simp add: mult.commute)

lemma Suc-mult-cancel1: \( \text{Suc} k \cdot m = \text{Suc} k \cdot n \iff m = n \)
by (subst mult-cancel1) simp

15.4 Orders on nat
15.4.1 Operation definition

instantiation nat :: linorder
begin

primrec less-eq-nat
where
  \( \text{(0::nat)} \leq n \iff \text{True} \)
| \( \text{Suc} m \leq n \iff (\text{case} n \text{ of } 0 \Rightarrow \text{False} \mid \text{Suc} n \Rightarrow m \leq n) \)

declare less-eq-nat.simps [simp del]

lemma le0 [iff]: \( 0 \leq n \) for \( n \) :: nat
by (simp add: less-eq-nat.simps)

lemma [code]: \( 0 \leq n \iff \text{True} \)
for \( n \) :: nat
by simp

definition less-nat
where less-eq-Suc-le: \( n < m \iff \text{Suc} n \leq m \)
lemma Suc-le-mono [iff]: Suc n ≤ Suc m ←→ n ≤ m
  by (simp add: less-eq-nat.simps(2))

lemma Suc-le-eq [code]: Suc m ≤ n ←→ m < n
  unfolding less-eq-Suc-le ..

lemma le-0-eq [iff]: n ≤ 0 ←→ n = 0
  for n :: nat
  by (induct n) (simp-all add: less-eq-nat.simps(2))

lemma not-less0 [iff]: ¬ n < 0
  for n :: nat
  by (simp add: less-eq-Suc-le)

lemma less-nat-zero-code [code]: n < 0 ←→ False
  for n :: nat
  by simp

lemma Suc-less-eq [iff]: Suc m < Suc n ←→ m < n
  by (simp add: less-eq-Suc-le)

lemma less-Suc-eq-le [code]: m < Suc n ←→ m ≤ n
  by (simp add: less-eq-Suc-le)

lemma Suc-less-eq2: Suc n < m ←→ (∃ m'. m = Suc m' ∧ n < m')
  by (cases m) auto

lemma le-SucI: m ≤ n ⇒ m ≤ Suc n
  by (induct m arbitrary: n) (simp-all add: less-eq-nat.simps(2) split: nat.splits)

lemma Suc-leD: Suc m ≤ n ⇒ m ≤ n
  by (cases n) (auto intro: le-SucI)

lemma less-SucI: m < n ⇒ m < Suc n
  by (simp add: less-eq-Suc-le) (erule Suc-leD)

lemma Suc-lessD: Suc m < n ⇒ m < n
  by (simp add: less-eq-Suc-le) (erule Suc-leD)

instance

proof
  fix n m q :: nat

  show n < m ←→ n ≤ m ∧ ¬ m ≤ n
  proof (induct n arbitrary: m)
    case 0
    then show ?case
      by (cases m) (simp-all add: less-eq-Suc-le)

    next
case (Suc n)
then show ?case
  by (cases m) (simp-all add: less-eq-Suc-le)
qed

show n ≤ n
  by (induct n) simp-all
then show n = m if n ≤ m and m ≤ n
  using that by (induct n arbitrary: m)
  (simp-all add: less-eq-nat.simps(2) split: nat.splits)

show n ≤ q if n ≤ m and m ≤ q
  using that

proof (induct n arbitrary: m q)
  case 0
  show ?case by simp
next
  case (Suc n)
  then show ?case
    by (simp-all (no-asn-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,
      simp-all (no-asn-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,
      simp-all (no-asn-use) add: less-eq-nat.simps(2) split: nat.splits)
qed

show n ≤ m ∨ m ≤ n
  by (induct n arbitrary: m)
  (simp-all add: less-eq-nat.simps(2) split: nat.splits)
qed

deefinition bot-nat :: nat
  where bot-nat = 0

instance
  by standard (simp add: bot-nat-def)

end

instance nat :: no-top
  by standard (auto intro: less-Suc-eq-le [THEN iffD2])

15.4.2 Introduction properties

lemma lessI [iff]: n < Suc n
  by (simp add: less-Suc-eq-le)

lemma zero-less-Suc [iff]: 0 < Suc n
  by (simp add: less-Suc-eq-le)
15.4.3 Elimination properties

**lemma** less-not-refl: \( \neg n < n \)
  for \( n :: \text{nat} \)
  by (rule order-less-irrefl)

**lemma** less-not-refl2: \( n < m \Rightarrow m \neq n \)
  for \( m n :: \text{nat} \)
  by (rule not-sym) (rule less-imp-neq)

**lemma** less-not-refl3: \( s < t \Rightarrow s \neq t \)
  for \( s t :: \text{nat} \)
  by (rule less-imp-neq)

**lemma** less-irrefl-nat: \( n < n \Rightarrow R \)
  for \( n :: \text{nat} \)
  by (rule notE, rule less-not-refl)

**lemma** less-zeroE: \( n < 0 \Rightarrow R \)
  for \( n :: \text{nat} \)
  by (rule notE) (rule not-less0)

**lemma** less-Suc-eq: \( m < \text{Suc} n \iff m < n \lor m = n \)
  unfolding less-Suc-eq-le le-less ..

**lemma** less-Suc0 [iff]: \( (n < \text{Suc} 0) = (n = 0) \)
  by (simp add: less-Suc-eq)

**lemma** less-one [iff]: \( n < 1 \iff n = 0 \)
  for \( n :: \text{nat} \)
  unfolding One-nat-def by (rule less-Suc0)

**lemma** Suc-mono: \( m < n \Rightarrow \text{Suc} m < \text{Suc} n \)
  by simp

"Less than" is antisymmetric, sort of.

**lemma** less-antisym: \( \neg n < m \Rightarrow n < \text{Suc} m \Rightarrow m = n \)
  unfolding not-less less-Suc-eq-le by (rule antisym)

**lemma** nat-neq-iff: \( m \neq n \iff m < n \lor n < m \)
  for \( m n :: \text{nat} \)
  by (rule linorder-neq-iff)

15.4.4 Inductive (?) properties

**lemma** Suc-less1: \( m < n \Rightarrow \text{Suc} m \neq n \Rightarrow \text{Suc} m < n \)
  unfolding less-eq-Suc-le [of m] le-less by simp

**lemma** lessE:
  assumes major: \( i < k \)
and 1: \( k = \text{Suc} \ i \implies P \)
and 2: \( \forall j. \ i < j \implies k = \text{Suc} \ j \implies P \)
shows \( P \)
proof –
from major have \( \exists j. \ i \leq j \land k = \text{Suc} \ j \)
  unfolding less-eq-Suc-le by (induct \( k \)) simp-all
then have \( (\exists j. \ i < j \land k = \text{Suc} \ j) \lor k = \text{Suc} \ i \)
  by (auto simp add: less-le)
with 1 2 show \( P \) by auto
qed

lemma less-SucE:
  assumes major: \( m < \text{Suc} \ n \)
  and less: \( m < n \implies P \)
  and eq: \( m = n \implies P \)
  shows \( P \)
  apply (rule major [THEN lessE])
  apply (rule eq)
  apply blast
  apply (rule less)
  apply blast
  done

lemma Suc-lessE:
  assumes major: \( \text{Suc} \ i < k \)
  and minor: \( \forall j. \ i < j \implies k = \text{Suc} \ j \implies P \)
  shows \( P \)
  apply (rule major [THEN lessE])
  apply (erule lessI [THEN minor])
  apply (erule Suc-lessD [THEN minor])
  apply assumption
  done

lemma Suc-less-SucD: \( \text{Suc} \ m < \text{Suc} \ n \implies m < n \)
by simp

lemma less-trans-Suc:
  assumes le: \( i < j \)
  shows \( j < k \implies \text{Suc} \ i < k \)
proof (induct \( k \))
  case 0
  then show \(?case\) by simp
next
  case (Suc \( k \))
  with le show \(?case\)
    by simp (auto simp add: less-Suc-eq dest: Suc-lessD)
qed

Can be used with less-Suc-eq to get \( n = m \lor n < m \).
lemma not-less-eq: ¬ m < n ←→ n < Suc m
  by (simp only: not-less less-Suc-eq-le)

lemma not-less-eq-eq: ¬ m ≤ n ←→ Suc n ≤ m
  by (simp only: not-le Suc-le-eq)

Properties of "less than or equal".

lemma le-imp-less-Suc: m ≤ n =⇒ m < Suc n
  by (simp only: less-Suc-eq-le)

lemma Suc-n-not-le-n: ¬ Suc n ≤ n
  by (simp add: not-le less-Suc-eq-le)

lemma le-Suc-eq: m ≤ Suc n ←→ m ≤ n ∨ m = Suc n
  by (simp add: less-Suc-eq-le [symmetric] less-Suc-eq)

lemma le-SucE: m ≤ Suc n =⇒ (m ≤ n =⇒ R) =⇒ (m = Suc n =⇒ R) =⇒ R
  by (drule le-Suc-eq [THEN iffD1], iprover+)

lemma Suc-leI: m < n =⇒ Suc m ≤ n
  by (simp only: Suc-le-eq)

Stronger version of Suc-leD.

lemma Suc-le-lessD: Suc m ≤ n =⇒ m < n
  by (simp only: Suc-le-eq)

lemma less-imp-le-nat: m < n =⇒ m ≤ n for m n :: nat
  unfolding less-eq-Suc-le by (rule Suc-leD)

For instance, (Suc m < Suc n) = (Suc m ≤ n) = (m < n)

lemmas le-simps = less-imp-le-nat less-Suc-eq-le Suc-le-eq

Equivalence of m ≤ n and m < n ∨ m = n

lemma less-or-eq-imp-le: m < n ∨ m = n =⇒ m ≤ n
  for m n :: nat
  unfolding le-less .

lemma le-eq-less-or-eq: m ≤ n =⇒ m < n ∨ m = n
  for m n :: nat
  by (rule le-less)

Useful with blast.

lemma eq-imp-le: m = n =⇒ m ≤ n
  for m n :: nat
  by auto

lemma le-refl: n ≤ n
  for n :: nat
THEORY "Nat"

by simp

lemma le-trans: i ≤ j ⟹ j ≤ k ⟹ i ≤ k
  for i j k :: nat
  by (rule order-trans)

lemma le-antisym: m ≤ n ⟹ n ≤ m ⟹ m = n
  for m n :: nat
  by (rule antisym)

lemma nat-less-le: m < n ⟷ m ≤ n ∧ m ≠ n
  for m n :: nat
  by (rule less-le)

lemma le-neq-implies-less: m ≤ n ⟹ m ≠ n ⟹ m < n
  for m n :: nat
  unfolding less-le ..

lemma nat-le-linear: m ≤ n ∨ n ≤ m
  for m n :: nat
  by (rule linear)

lemmas linorder-neqE-nat = linorder-neqE [where 'a = nat]

lemma le-less-Suc-eq: m ≤ n ⟹ n < Suc m ⟷ n = m
  unfolding less-Suc-eq-le by auto

lemma not-less-less-Suc-eq: ¬ n < m ⟹ n < Suc m ⟷ n = m
  unfolding not-less by (rule le-less-Suc-eq)

lemmas not-less-simps = not-less-less-Suc-eq le-less-Suc-eq

lemma not0-implies-Suc: n ≠ 0 ⟹ ∃ m. n = Suc m
  by (cases n) simp-all

lemma gr0-implies-Suc: n > 0 ⟹ ∃ m. n = Suc m
  by (cases n) simp-all

lemma gr-implies-not0: m < n ⟹ n ≠ 0
  for m n :: nat
  by (cases n) simp-all

lemma neq0-conv[iff]: n ≠ 0 ⟷ 0 < n
  for n :: nat
  by (cases n) simp-all

This theorem is useful with blast

lemma gr0I: (n = 0 ⟹ False) ⟹ 0 < n
  for n :: nat
by (rule neq0-conv[THEN iffD1]) iprover

lemma gr0-conv-Suc: 0 < n <-> (∃ m. n = Suc m)
  by (fast intro: not0-implies-Suc)

lemma not-gr0 [iff]: ~ 0 < n <-> n = 0
  for n :: nat
  using neq0-conv by blast

lemma Suc-le-D: Suc n ≤ m' =⇒ ∃ m. m' = Suc m
  by (induct m') simp-all

Useful in certain inductive arguments

lemma All-less-Suc2: (∀ i < Suc n. P i) = (P 0 ∨ (∃ i < n. P i))
  by (auto simp: less-Suc-eq-0-disj)

lemma Ex-less-Suc2: (∃ i < Suc n. P i) = (P 0 ∨ (∃ i < n. P i))
  by (auto simp: less-Suc-eq-0-disj)

mono (non-strict) doesn’t imply increasing, as the function could be constant

lemma strict-mono-imp-increasing:
  fixes n::nat
  assumes strict-mono f shows f n ≥ n
  proof (induction n)
    case 0
    then show ?case by auto
  next
    case (Suc n)
    then show ?case
      unfolding not-less-eq-eq [symmetric]
      using Suc-n-not-le-n assms order-trans strict-mono-less-eq by blast
  qed

15.4.5 Monotonicity of Addition

lemma Suc-pred [simp]: n > 0 =⇒ Suc (n - Suc 0) = n
  by (simp add: diff-Suc split: nat.split)

lemma Suc-diff-1 [simp]: 0 < n =⇒ Suc (n - 1) = n
unfolding \texttt{One-nat-def} by (rule Suc-pred)

\textbf{lemma} nat-add-left-cancel-le [simp]: $k + m \leq k + n \iff m \leq n$
\begin{itemize}
  \item for $k m n :: \text{nat}$
  \item by (induct $k$) simp-all
\end{itemize}

\textbf{lemma} nat-add-left-cancel-less [simp]: $k + m < k + n \iff m < n$
\begin{itemize}
  \item for $k m n :: \text{nat}$
  \item by (induct $k$) simp-all
\end{itemize}

\textbf{lemma} add-gr-0 [iff]: $m + n > 0 \iff m > 0 \lor n > 0$
\begin{itemize}
  \item for $m n :: \text{nat}$
  \item by (auto dest: gr0-implies-Suc)
\end{itemize}

\textbf{strict, in 1st argument}

\textbf{lemma} add-less-mono1: $i < j \Rightarrow i + k < j + k$
\begin{itemize}
  \item for $i j k :: \text{nat}$
  \item by (induct $k$) simp-all
\end{itemize}

\textbf{strict, in both arguments}

\textbf{lemma} add-less-mono: $i < j \Rightarrow k < l \Rightarrow i + k < j + l$
\begin{itemize}
  \item for $i j k l :: \text{nat}$
  \item apply (rule add-less-mono1 [THEN less-trans], assumption+)
  \item apply (induct $j$)
  \item apply simp-all
  \item done
\end{itemize}

\textbf{lemma} less-imp-Suc-add: $m < n \Rightarrow \exists k. n = Suc (m + k)$
\begin{itemize}
  \item proof (induct $n$)
    \begin{itemize}
      \item case 0
        \begin{itemize}
          \item then show ?case by simp
        \end{itemize}
    \end{itemize}
  \item next
    \begin{itemize}
      \item case Suc
        \begin{itemize}
          \item then show ?case
            \begin{itemize}
              \item by (simp add: order-le-less)
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}

\textbf{qd}

\textbf{lemma} le-Suc-ex: $k \leq l \Rightarrow (\exists n. l = k + n)$
\begin{itemize}
  \item for $k l :: \text{nat}$
  \item by (auto simp: less-Suc-eq-le[symmetric] dest: less-imp-Suc-add)
\end{itemize}

\textbf{lemma} less-natE:
\begin{itemize}
  \item assumes $m < n$
  \item obtains $q$ where $n = Suc (m + q)$
  \item using assms by (auto dest: less-imp-Suc-add intro: that)
\end{itemize}

\textbf{strict, in 1st argument; proof is by induction on $k > 0$}
lemma `mult-less-mono2`:
fixes `i j :: nat`
assumes `i < j` and `0 < k`
shows `k * i < k * j`
using `(0 < k)`
proof (induct `k`
  case 0
  then show ?case by simp
next
  case (Suc `k`)
  with `(i < j)` show ?case
  by (cases `k`) (simp-all add: add-less-mono)
qed

Addition is the inverse of subtraction: if `n ≤ m` then `n + (m − n) = m`.

lemma `add-diff-inverse-nat`:
for `m n :: nat`
by (induct `m n` rule: diff-induct) simp-all

lemma `nat-le-iff-add`:
for `m n :: nat`
using `nat-add-left-cancel-le[ of m 0]` by (auto dest: le-Suc-ex)

The naturals form an ordered semidom and a dioid.

instance `nat :: linordered-semidom`
proof
  fix `m n q :: nat`
  show `0 < (1::nat)`
    by simp
  show `m ≤ n ⇒ q + m ≤ q + n`
    by simp
  show `m < n ⇒ 0 < q ⇒ q * m < q * n`
    by (simp add: mult-less-mono2)
  show `m ≠ 0 ⇒ n ≠ 0 ⇒ m * n ≠ 0`
    by simp
  show `n ≤ m ⇒ (m − n) + n = m`
    by (simp add: add-diff-inverse-nat add.commute linorder-not-less)
qed

instance `nat :: dioid`
by standard (rule nat-le-iff-add)

declare `le0[simp del]` — This is now `(0::'a) ≤ ?x`
declare `le-0-eq[simp del]` — This is now `(?n ≤ (0::'a)) = (?n = (0::'a))`
declare `not-less0[simp del]` — This is now `¬ ?n < (0::'a)`
declare `not-gr0[simp del]` — This is now `¬ (0::'a) < ?n) = (?n = (0::'a))`

instance `nat :: ordered-cancel-comm-monoid-add` ..
instance `nat :: ordered-cancel-comm-monoid-diff` ..
15.4.6  \textit{min} and \textit{max}

\textbf{lemma} mono-Suc: mono Suc
  \begin{itemize}
  \item by (rule monoI) simp
  \end{itemize}

\textbf{lemma} min-0L [simp]: \( \text{min } 0 \ n = 0 \)
  \begin{itemize}
  \item for \( n :: \text{nat} \)
  \item by (rule min-absorb1) simp
  \end{itemize}

\textbf{lemma} min-0R [simp]: \( \text{min } n \ 0 = 0 \)
  \begin{itemize}
  \item for \( n :: \text{nat} \)
  \item by (rule min-absorb2) simp
  \end{itemize}

\textbf{lemma} min-Suc-Suc [simp]: \( \text{min } \text{Suc } m \) \( \text{Suc } n \) \( = \text{Suc } (\text{min } m \ n) \)
  \begin{itemize}
  \item by (simp add: mono-Suc min-of-mono)
  \end{itemize}

\textbf{lemma} min-Suc1: \( \text{min } \text{Suc } n \) \( m = (\text{case } m \text{ of } 0 \Rightarrow 0 \mid \text{Suc } m' \Rightarrow \text{Suc}(\text{min } n \ m')) \)
  \begin{itemize}
  \item by (simp split: nat.split)
  \end{itemize}

\textbf{lemma} min-Suc2: \( \text{min } m \) \( \text{Suc } n \) \( = (\text{case } m \text{ of } 0 \Rightarrow 0 \mid \text{Suc } m' \Rightarrow \text{Suc}(\text{min } m' \ n)) \)
  \begin{itemize}
  \item by (simp split: nat.split)
  \end{itemize}

\textbf{lemma} max-0L [simp]: \( \text{max } 0 \ n = n \)
  \begin{itemize}
  \item for \( n :: \text{nat} \)
  \item by (rule max-absorb2) simp
  \end{itemize}

\textbf{lemma} max-0R [simp]: \( \text{max } n \ 0 = n \)
  \begin{itemize}
  \item for \( n :: \text{nat} \)
  \item by (rule max-absorb1) simp
  \end{itemize}

\textbf{lemma} max-Suc-Suc [simp]: \( \text{max } \text{Suc } m \) \( \text{Suc } n \) \( = \text{Suc } (\text{max } m \ n) \)
  \begin{itemize}
  \item by (simp add: mono-Suc max-of-mono)
  \end{itemize}

\textbf{lemma} max-Suc1: \( \text{max } \text{Suc } n \) \( m = (\text{case } m \text{ of } 0 \Rightarrow \text{Suc } n \mid \text{Suc } m' \Rightarrow \text{Suc}(\text{max } m \ n')) \)
  \begin{itemize}
  \item by (simp split: nat.split)
  \end{itemize}

\textbf{lemma} max-Suc2: \( \text{max } m \) \( \text{Suc } n \) \( = (\text{case } m \text{ of } 0 \Rightarrow \text{Suc } n \mid \text{Suc } m' \Rightarrow \text{Suc}(\text{max } m' \ n)) \)
  \begin{itemize}
  \item by (simp split: nat.split)
  \end{itemize}

\textbf{lemma} nat-mult-min-left: \( \text{min } m \ n \ast q = \text{min } (m \ast q) \ast (n \ast q) \)
  \begin{itemize}
  \item for \( m \ n \ q :: \text{nat} \)
  \item by (simp add: min-def not-le)
    \begin{itemize}
    \item (auto dest: mult-right-le-imp-le mult-right-less-imp-less le-less-trans)
    \end{itemize}
  \end{itemize}

\textbf{lemma} nat-mult-min-right: \( m \ast \text{min } n \ q = \text{min } (m \ast n) \ast (m \ast q) \)
  \begin{itemize}
  \item for \( m \ n \ q :: \text{nat} \)
  \end{itemize}
by (simp add: min-def not-le)
(auto dest: mult-left-le-imp-le mult-left-less-imp-less le-less-trans)

lemma nat-add-max-left: max m n + q = max (m + q) (n + q)
for m n q :: nat
by (simp add: max-def)

lemma nat-add-max-right: m + max n q = max (m + n) (m + q)
for m n q :: nat
by (simp add: max-def)

lemma nat-mult-max-left: max m n * q = max (m * q) (n * q)
for m n q :: nat
by (simp add: max-def not-le)
(auto dest: mult-right-le-imp-le mult-right-less-imp-less le-less-trans)

lemma nat-mult-max-right: m * max n q = max (m * n) (m * q)
for m n q :: nat
by (simp add: max-def not-le)
(auto dest: mult-left-le-imp-le mult-left-less-imp-less le-less-trans)

15.4.7 Additional theorems about (≤)
Complete induction, aka course-of-values induction

instance nat :: wellorder
proof
  fix P and n :: nat
  assume step: (∀m. m < n ⇒ P m) ⇒ P n for n :: nat
  have (∀q. q ≤ n ⇒ P q)
  proof (induct n)
    case (0 n)
    have P 0 by (rule step) auto
    with 0 show ?case by auto
  next
    case (Suc m n)
    then have n ≤ m ∨ n = Suc m
    by (simp add: le-Suc-eq)
    then show ?case
    proof
      assume n ≤ m
      then show P n by (rule Suc(1))
    next
      assume n: n = Suc m
      show P n by (rule step) (rule Suc(1), simp add: n le-simps)
    qed
  qed
  then show P n by auto
  qed
lemma Least-eq-0 [simp]: \( P \ 0 \Rightarrow \text{Least} \ P = 0 \)
for \( P :: \text{nat} \Rightarrow \text{bool} \)
by (rule Least-equality [OF - le0])

lemma Least-Suc: \( P \ n \Rightarrow \neg \ P \ 0 \Rightarrow (\text{LEAST} \ n. \ P \ n) = \text{Suc} \ (\text{LEAST} \ m. \ P \ (\text{Suc} \ m)) \)
apply (cases \( n \))
apply auto
apply (rule LeastI)
apply (drule-tac \( \lambda x. \ P \ (\text{Suc} \ x) \) in LeastI)
apply (subgoal-tac \( (\text{LEAST} \ x. \ P \ x) \leq \text{Suc} \ (\text{LEAST} \ x. \ P \ (\text{Suc} \ x))) \)
apply (erule-tac [2] Least-le)
apply (cases \( \text{LEAST} \ x. \ P \ x \))
apply auto
apply (drule-tac \( \lambda x. \ P \ (\text{Suc} \ x) \) in Least-le)
apply (blast intro: order-antisym)
done

lemma Least-Suc2: \( P \ n \Rightarrow Q \ m \Rightarrow \neg \ P \ 0 \Rightarrow \forall k. \ P \ (\text{Suc} \ k) = Q \ k \Rightarrow \text{Least} \ P = \text{Suc} \ (\text{Least} \ Q) \)
by (erule (1) Least-Suc [THEN ssubst]) simp

lemma ex-least-nat-le: \( \neg \ P \ 0 \Rightarrow P \ n \Rightarrow \exists k \leq n. (\forall i < k. \neg \ P \ i) \land P \ k \)
for \( P :: \text{nat} \Rightarrow \text{bool} \)
apply (cases \( n \))
apply blast
apply (rule-tac \( x=\text{LEAST} \ k. \ P \ k \) in exI)
apply (blast intro: Least-le dest: not-less-Least intro: LeastI-ex)
done

lemma ex-least-nat-less: \( \neg \ P \ 0 \Rightarrow P \ n \Rightarrow \exists k < n. (\forall i < k. \neg \ P \ i) \land P \ (k + 1) \)
for \( P :: \text{nat} \Rightarrow \text{bool} \)
apply (cases \( n \))
apply blast
apply (frule (1) ex-least-nat-le)
apply (erule exE)
apply (case-tac \( k \))
apply simp
apply (rename-tac \( ki \))
apply (rule-tac \( x=ki \) in exI)
apply (auto simp add: less-eq-Suc-le)
done

lemma nat-less-induct:
fixes \( P :: \text{nat} \Rightarrow \text{bool} \)
assumes \( \forall n. \forall m. \ m < n \Rightarrow P \ m \Rightarrow P \ n \)
shows \( P \ n \)
using assms less-induct by blast
lemma measure-induct-rule [case-names less]:
  fixes f :: 'a ⇒ 'b::wellorder
  assumes step: ∀x. (∀y. f y < f x ⇒ P y) ⇒ P x
  shows P a
  by (induct m ≡ f a arbitrary: a rule: less-induct) (auto intro: step)

old style induction rules:

lemma measure-induct:
  fixes f :: 'a ⇒ 'b::wellorder
  shows (∀x. (∀y. f y < f x ⇒ P y) ⇒ P x) ⇒ P a
  by (rule measure-induct-rule [of f P a]) iprover

lemma full-nat-induct:
  assumes step: ∀n. (∀m. Suc m ≤ n ⇒ P m) ⇒ P n
  shows P n
  by (rule less-induct) (auto intro: step simp: le-simps)

An induction rule for establishing binary relations

lemma less-Suc-induct [consumes 1]:
  assumes less: i < j
  and step: ∀i. P i (Suc i)
  and trans: ∀i j k. i < j ⇒ j < k ⇒ P i j ⇒ P j k ⇒ P i k
  shows P i j
  proof
  from less obtain k where j: j = Suc (i + k)
  by (auto dest: less-imp-Suc-add)
  have P i (Suc (i + k))
  proof (induct k)
    case 0
    show ?case by (simp add: step)
  next
    case (Suc k)
    have 0 + i < Suc k + i by (rule add-less-mono1) simp
    then have i < Suc (i + k) by (simp add: add.commute)
    from trans[OF this lessI Suc step] show ?case by simp
  qed
  then show P i j by (simp add: j)
  qed

The method of infinite descent, frequently used in number theory. Provided by Roelof Oosterhuis. P n is true for all natural numbers if

  • case “0”: given n = 0 prove P n
  • case “smaller”: given n > 0 and ¬ P n prove there exists a smaller natural number m such that ¬ P m.
lemma infinite-descent: \((\forall n. \neg P n \implies \exists m < n. \neg P m) \implies P n\) for \(P :: \text{nat} \Rightarrow \text{bool}\)

— compact version without explicit base case

by (induct n rule: less-induct) auto

lemma infinite-descent0 [case-names 0 smaller]:

fixes \(P :: \text{nat} \Rightarrow \text{bool}\)

assumes \(P 0\)

and \(\forall n. n > 0 \implies \neg P n \implies \exists m. m < n \land \neg P m\)

shows \(P n\)

apply (rule infinite-descent)

using assms

apply (case-tac n > 0)

apply auto

done

Infinite descent using a mapping to \(\text{nat}\): \(P x\) is true for all \(x \in D\) if there exists a \(V \in D \Rightarrow \text{nat}\) and

- case “0”: given \(V x = 0\) prove \(P x\)
- “smaller”: given \(V x > 0\) and \(\neg P x\) prove there exists a \(y \in D\) such that \(V y < V x\) and \(\neg P y\).

corollary infinite-descent0-measure [case-names 0 smaller]:

fixes \(V :: 'a \Rightarrow \text{nat}\)

assumes 1: \(\forall x. V x = 0 \implies P x\)

and 2: \(\forall x. V x > 0 \implies \neg P x \implies \exists y. V y < V x \land \neg P y\)

shows \(P x\)

proof —

obtain \(n\) where \(n = V x\) by auto

moreover have \(\forall x. V x = n \implies P x\)

proof (induct n rule: infinite-descent0)

case 0

with 1 show \(P x\) by auto

next

case (smaller \(n\))

then obtain \(x\) where \(*: V x = n\) and \(V x > 0 \land \neg P x\) by auto

with 2 obtain \(y\) where \(V y < V x \land \neg P y\) by auto

with \(*: y\) obtain \(m\) where \(m = V y \land m < n \land \neg P y\) by auto

then show \(?case\) by auto

qed

ultimately show \(P x\) by auto

qed

Again, without explicit base case:

lemma infinite-descent-measure:

fixes \(V :: 'a \Rightarrow \text{nat}\)

assumes \(\forall x. \neg P x \implies \exists y. V y < V x \land \neg P y\)
shows $P \, x$

proof
from assms obtain $n$ where $n = V \, x$ by auto
moreover have $\forall x. \, V \, x = n \implies P \, x$
proof (induct $n$ rule: infinite-descent, auto)
show $\exists m < V \, x. \, \exists y. \, V \, y = m \land \neg P \, y$ if $\neg P \, x$ for $x$
using assms and that by auto
qed
ultimately show $P \, x$ by auto
qed

A (clumsy) way of lifting $<$ monotonicity to $\leq$ monotonicity

lemma less-mono-imp-le-mono:
fixes $f :: \text{nat} \Rightarrow \text{nat}$
and $i \, j :: \text{nat}$
assumes $\forall i \, j :: \text{nat}. \, i < j \implies f \, i < f \, j$
and $i \leq j$
show $f \, i \leq f \, j$
using assms by (auto simp add: order-le-less)

non-strict, in 1st argument

lemma add-le mono1: $i \leq j \implies i + k \leq j + k$
for $i \, j \, k :: \text{nat}$
by (rule add-right-mono)

non-strict, in both arguments

lemma add-le mono: $i \leq j \implies k \leq l \implies i + k \leq j + l$
for $i \, j \, k \, l :: \text{nat}$
by (rule add-mono)

lemma le-add2: $n \leq m + n$
for $m \, n :: \text{nat}$
by simp

lemma le-add1: $n \leq n + m$
for $m \, n :: \text{nat}$
by simp

lemma less-add-Suc1: $i < \text{Suc} \, (i + m)$
by (rule le-less-trans, rule le-add1, rule lessI)

lemma less-add-Suc2: $i < \text{Suc} \, (m + i)$
by (rule le-less-trans, rule le-add2, rule lessI)

lemma less-iff-Suc-add: $m < n \iff (\exists k. \, n = \text{Suc} \, (m + k))$
by (iprover intro!: less-add-Suc1 less-imp-Suc-add)

lemma trans-le-add1: $i \leq j \implies i \leq j + m$
for $i \, j \, m :: \text{nat}$
by (rule le-trans, assumption, rule le-add1)

lemma trans-le-add2: \(i \leq j \implies i \leq m + j\)  
for \(i \, j \, m::\text{nlat}\)  
by (rule le-trans, assumption, rule le-add2)

lemma trans-less-add1: \(i < j \implies i < j + m\)  
for \(i \, j \, m::\text{nlat}\)  
by (rule less-le-trans, assumption, rule le-add1)

lemma trans-less-add2: \(i < j \implies i < m + j\)  
for \(i \, j \, m::\text{nlat}\)  
by (rule less-le-trans, assumption, rule le-add2)

lemma add-lessD1: \(i + j < k \implies i < k\)  
for \(i \, j \, k::\text{nlat}\)  
by (rule le-less-trans \[of - i+j\]) (simp-all add: le-add1)

lemma not-add-less1 [iff]: \(\neg i + j < i\)  
for \(i \, j::\text{nlat}\)  
apply (rule notI)  
apply (drule add-lessD1)  
apply (erule less-irrefl \[THEN notE\])  
done

lemma not-add-less2 [iff]: \(\neg j + i < i\)  
for \(i \, j::\text{nlat}\)  
by (simp add: add.commute)

lemma add-leD1: \(m + k \leq n \implies m \leq n\)  
for \(k \, m \, n::\text{nlat}\)  
by (rule order-trans \[of - m+k\]) (simp-all add: le-add1)

lemma add-leD2: \(m + k \leq n \implies k \leq n\)  
for \(k \, m \, n::\text{nlat}\)  
apply (simp add: add.commute)  
apply (erule add-leD1)  
done

lemma add-leE: \(m + k \leq n \implies (m \leq n \implies k \leq n \implies R) \implies R\)  
for \(k \, m \, n::\text{nlat}\)  
by (blast dest: add-leD1 add-leD2)

needs \(\land k\) for ac-simps to work

lemma less-add-eq-less: \(\land k, \, k < l \implies m + l = k + n \implies m < n\)  
for \(l \, m \, n::\text{nlat}\)  
by (force simp del: add-Suc-right simp add: less-iff-Suc-add add-Suc-right \[symmetric\]  
ac-simps)
15.4.8 More results about difference

lemma Suc-diff-le: \( n \leq m \implies \text{Suc } m - n = \text{Suc } (m - n) \)
  by (induct m n rule: diff-induct) simp-all

lemma diff-less-Suc: \( m - n < \text{Suc } m \)
  apply (induct m n rule: diff-induct)
  apply (erule-tac [\] less-SucE)
  apply (simp-all add: less-Suc-eq)
done

lemma diff-less-Suc: \( m - n < \text{Suc } m \)
  apply (induct m n rule: diff-induct)
  apply (erule-tac [\] less-SucE)
  apply (simp-all add: less-Suc-eq)
done

lemma diff-le-self [simp]: \( m - n \leq m \)
  for m n :: nat
  by (induct m n rule: diff-induct) (simp-all add: le-SucI)

lemma less-imp-diff-less: \( j < k \implies j - n < k \)
  for j k n :: nat
  by (rule le-less-trans, rule diff-le-self)

lemma diff-Suc-less [simp]: \( \emptyset < n \implies n - \text{Suc } i < n \)
  for cases n (auto simp add: le-simps)

lemma diff-add-assoc: \( k \leq j \implies (i + j) - k = i + (j - k) \)
  for i j k :: nat
  by (fact ordered-cancel-comm-monoid-diff-class.diff-add-assoc)

lemma add-diff-assoc [simp]: \( k \leq j \implies i + (j - k) = i + j - k \)
  for i j k :: nat
  by (fact ordered-cancel-comm-monoid-diff-class.add-diff-assoc)

lemma diff-add-assoc2: \( k \leq j \implies (j + i) - k = (j - k) + i \)
  for i j k :: nat
  by (fact ordered-cancel-comm-monoid-diff-class.diff-add-assoc2)

lemma add-diff-assoc2 [simp]: \( k \leq j \implies j - k + i = j + i - k \)
  for i j k :: nat
  by (fact ordered-cancel-comm-monoid-diff-class.add-diff-assoc2)

lemma le-imp-diff-is-add: \( i \leq j \implies (j - i = k) = (j = k + i) \)
  for i j k :: nat
  by auto

lemma diff-is-0-eq [simp]: \( m - n = \emptyset \iff m \leq n \)
  for m n :: nat
  by (induct m n rule: diff-induct) simp-all

lemma diff-is-0-eq' [simp]: \( m \leq n \implies m - n = \emptyset \)
  for m n :: nat
  by (rule iffD2, rule diff-is-0-eq)
lemma zero-less-diff [simp]: \( 0 < n - m \iff m < n \)
for \( m n :: \text{nat} \)
by (induct \( m n \) rule: diff-induct) simp-all

lemma less-imp-add-positive:
assumes \( i < j \)
shows \( \exists k :: \text{nat}. \ 0 < k \land i + k = j \)
proof
from \( \text{assms} \) show \( 0 < j - i \land i + (j - i) = j \)
  by (simp add: order-less-imp-le)
qed

a nice rewrite for bounded subtraction

lemma nat-minus-add-max: \( n - m + m = \max n m \)
for \( m n :: \text{nat} \)
by (simp add: max-def not-le order-less-imp-le)

lemma nat-diff-split: \( P (a - b) \iff (a < b \longrightarrow P 0) \land (\forall d. a = b + d \longrightarrow P d) \)
for \( a b :: \text{nat} \)
— elimination of \( - \) on \( \text{nat} \)
by (cases \( a < b \)) (auto simp: not-less le-less dest: add-eq-self-zero [OF sym])

lemma nat-diff-split-asm: \( P (a - b) \iff \neg (a < b \land \neg P 0 \lor (\exists d. a = b + d \land \neg P d)) \)
for \( a b :: \text{nat} \)
— elimination of \( - \) on \( \text{nat} \) in assumptions
by (auto split: nat-diff-split)

lemma Suc-pred': \( 0 < n \implies n = \text{Suc}(n - 1) \)
by simp

lemma add-eq-if: \( m + n = (\text{if} \ m = 0 \ \text{then} \ n \ \text{else} \ \text{Suc} \ ((m - 1) + n)) \)
unfolding One-nat-def by (cases \( m \)) simp-all

lemma mult-eq-if: \( m * n = (\text{if} \ m = 0 \ \text{then} \ 0 \ \text{else} \ n + ((m - 1) * n)) \)
for \( m n :: \text{nat} \)
by (cases \( m \)) simp-all

lemma Suc-diff-eq-diff-pred: \( 0 < n \implies \text{Suc} m - n = m - (n - 1) \)
by (cases \( n \)) simp-all

lemma diff-Suc-eq-diff-pred: \( m - \text{Suc} n = (m - 1) - n \)
by (cases \( m \)) simp-all

lemma Let-Suc [simp]: \( \text{Let} \ (\text{Suc} n) \ f \equiv f \ (\text{Suc} n) \)
by (fact Let-def)
15.4.9 Monotonicity of multiplication

lemma mult-le-mono1: \( i \leq j \implies i \cdot k \leq j \cdot k \)
for \( i, j, k :: \text{nat} \)
by (simp add: mult-right-mono)

lemma mult-le-mono2: \( i \leq j \implies k \cdot i \leq k \cdot j \)
for \( i, j, k :: \text{nat} \)
by (simp add: mult-left-mono)

\( \leq \) monotonicity, BOTH arguments

lemma mult-le-mono: \( i \leq j \implies k \leq l \implies i \cdot k \leq j \cdot l \)
for \( i, j, k, l :: \text{nat} \)
by (simp add: mult-mono)

lemma mult-less-mono1: \( i < j \implies 0 < k \implies i \cdot k < j \cdot k \)
for \( i, j, k :: \text{nat} \)
by (simp add: mult-strict-right-mono)

Differs from the standard zero-less-mult-iff in that there are no negative numbers.

lemma nat-0-less-mult-iff [simp]: \( 0 < m \cdot n \iff 0 < m \land 0 < n \)
for \( m, n :: \text{nat} \)
proof (induct m)
case 0
then show \(?case\) by simp
next
case (Suc m)
then show \(?case\) by (cases n) simp-all
qed

lemma one-le-mult-iff [simp]: \( \text{Suc } 0 \leq m \cdot n \iff \text{Suc } 0 \leq m \land \text{Suc } 0 \leq n \)
proof (induct m)
case 0
then show \(?case\) by simp
next
case (Suc m)
then show \(?case\) by (cases n) simp-all
qed

lemma mult-less-cancel2 [simp]: \( m \cdot k < n \cdot k \iff 0 < k \land m < n \)
for \( k, m, n :: \text{nat} \)
apply (safe intro: mult-less-mono1)
apply (cases k)
apply auto
apply (simp add: linorder-not-le [symmetric])
apply (blast intro: mult-le-mono1)
done
lemma mult-less-cancel1 [simp]: \( k \cdot m < k \cdot n \leftrightarrow 0 < k \land m < n \)
for \( k m n :: nat \)
by (simp add: mult.commute [of k])

lemma mult-le-cancel1 [simp]: \( k \cdot m \leq k \cdot n \leftrightarrow (0 < k \longrightarrow m \leq n) \)
for \( k m n :: nat \)
by (simp add: linorder-not-less [symmetric], auto)

lemma mult-le-cancel2 [simp]: \( m \cdot k \leq n \cdot k \leftrightarrow (0 < k \longrightarrow m \leq n) \)
for \( k m n :: nat \)
by (simp add: linorder-not-less [symmetric], auto)

lemma Suc-mult-less-cancel1: \( \ Suc \ k \cdot m < \ Suc \ k \cdot n \leftrightarrow m < n \)
by (subst mult-less-cancel1) simp

lemma Suc-mult-le-cancel1: \( \ Suc \ k \cdot m \leq \ Suc \ k \cdot n \leftrightarrow m \leq n \)
by (subst mult-le-cancel1) simp

lemma le-square: \( m \leq m \cdot m \)
for \( m :: nat \)
by (cases m) (auto intro: le-add1)

lemma le-cube: \( m \leq m \cdot (m \cdot m) \)
for \( m :: nat \)
by (cases m) (auto intro: le-add1)

Lemma for gcd

lemma mult-eq-self-implies-10: \( m = m \cdot n \Longrightarrow n = 1 \lor m = 0 \)
for \( m n :: nat \)
apply (drule sym)
apply (rule disjCI)
apply (rule linorder-cases)
defer
apply assumption
apply (drule mult-less-mono2)
apply auto
done

lemma mono-times-nat:
fixes \( n :: nat \)
assumes \( n > 0 \)
shows mono \( (times n) \)
proof
fix \( m q :: nat \)
assume \( m \leq q \)
with assms show \( n \cdot m \leq n \cdot q \) by simp
qed

The lattice order on \( nat \).


**15.5 Natural operation of natural numbers on functions**

We use the same logical constant for the power operations on functions and relations, in order to share the same syntax.

**consts** compow :: nat ⇒ ′a ⇒ ′a

**abbreviation** compower :: ′a ⇒ nat ⇒ ′a (infixr ˆ 80)

**notation** (latex output)

\( f ^{n} = f \circ \ldots \circ f \), the \( n \)-fold composition of \( f \)

**overloading**

funpow ≡ compow :: nat ⇒ (′a ⇒ ′a) ⇒ (′a ⇒ ′a)

**begin**

**primrec** funpow :: nat ⇒ (′a ⇒ ′a) ⇒ ′a

**where**

funpow 0 f = id

| funpow (Suc n) f = f \circ funpow n f

**end**

**lemma** funpow-0 [simp]: (f ^ 0) x = x

**by** simp

**lemma** funpow-Suc-right: f ^ Suc n = f ^ n \circ f

**proof** (induct n)

**case 0**

**then show** by simp

**next**

**fix** n

**assume** f ^ Suc n = f ^ n \circ f
then show \( f \uparrow \uparrow \text{Suc} \ (\text{Suc} \ n) = f \uparrow \uparrow \text{Suc} \ n \circ f \)
  by (simp add: o-assoc)
qed

lemmas funpow-simps-right = funpow.simps(1) funpow-Suc-right

For code generation.

definition funpow :: \(\text{nat} \Rightarrow (\tau \Rightarrow \tau) \Rightarrow \tau \Rightarrow \tau\)
where funpow-code-def:
  \[\text{funpow} = \text{compow}\]

lemma [code]:
  funpow (Suc n) f = f \circ funpow n f
  funpow 0 f = id
  by (simp-all add: funpow-code-def)

hide-const (open) funpow

lemma funpow-add: \(f \uparrow \uparrow (m + n) = f \uparrow \uparrow m \circ f \uparrow \uparrow n\)
  by (induct m) simp-all

lemma funpow-mult: \((f \uparrow \uparrow m) \uparrow \uparrow n = f \uparrow \uparrow (m * n)\)
  for f :: 
  by (induct n) (simp-all add: funpow-add)

lemma funpow-swap1: \(f \ (f \uparrow \uparrow n) \ x = (f \uparrow \uparrow n) \ (f \ x)\)
proof -
  have \(f \ (f \uparrow \uparrow n) \ x = (f \uparrow \uparrow (n + 1)) \ x\) by simp
  also have \(\ldots = (f \uparrow \uparrow n \circ f \uparrow \uparrow 1) \ x\)
    by (simp only: funpow-add)
  also have \(\ldots = (f \uparrow \uparrow n) \ (f \ x)\)
    by simp
  finally show ?thesis .
qed

lemma comp-funpow: \(\text{comp} \ f \uparrow \uparrow n = \text{comp} \ (f \uparrow \uparrow n)\)
  for f :: 
  by (induct n) simp-all

lemma Suc-funpow[simp]: \(\text{Suc} \uparrow \uparrow n = (+) \ n\)
  by (induct n) simp-all

lemma id-funpow[simp]: \(\text{id} \uparrow \uparrow n = \text{id}\)
  by (induct n) simp-all

lemma funpow-mono: mono f \Rightarrow A \leq B \Rightarrow (f \uparrow \uparrow n) \ A \leq (f \uparrow \uparrow n) \ B
  for f :: 
  by (induct n arbitrary: A B)
    (auto simp del: funpow.simps(2) simp add: funpow-Suc-right mono-def)

lemma funpow-mono2:
  assumes mono f
and $i \leq j$
and $x \leq y$
and $x \leq f \cdot x$
shows $(f \cdot^* i) \cdot x \leq (f \cdot^* j) \cdot y$
using assms(2,3)
proof (induct $j$ arbitrary: $y$)
case 0
then show $\ ?case$ by simp
next
case $(Suc \ j)$
show $\ ?case$
proof (cases $i = Suc \ j$)
case True
with assms(1) Suc show $\ ?thesis$
by (simp del: funpow.simps add: funpow-simps-right monoD funpow-mono)
next
case False
with assms(1,4) Suc show $\ ?thesis$
by (simp del: funpow.simps add: funpow-simps-right le-eq-less-or-eq less-Suc-eq-le)
(simp add: Suc.hyps monoD order-subst1)
qed
qed

lemma inj-fn[simp]:
fixes $f:\ 'a \Rightarrow 'a$
assumes inj $f$
shows inj $(f \cdot^n)$
proof (induction $n$)
  case Suc thus $\ ?case$ using inj-compose[OF assms Suc.IH] by (simp del: comp-apply)
qed simp

lemma surj-fn[simp]:
fixes $f:\ 'a \Rightarrow 'a$
assumes surj $f$
shows surj $(f \cdot^n)$
proof (induction $n$)
  case Suc thus $\ ?case$ by (simp add: comp-surj[OF Suc.IH assms] del: comp-apply)
qed simp

lemma bij-fn[simp]:
fixes $f:\ 'a \Rightarrow 'a$
assumes bij $f$
shows bij $(f \cdot^n)$
by (rule bijI[OF inj-fn[OF bij-is-inj[OF assms]] surj-fn[OF bij-is-surj[OF assms]]])

15.6 Kleene iteration

lemma Kleene-iter-lfp:
fixes $f : \ 'a::order-bot \Rightarrow 'a$
assumes mono $f$
and $f \leq p$
shows $(f \sim k) \bot \leq p$
proof (induct $k$)
case $0$
show ?case by simp
next
case Suc
show ?case
using monoD[OF assms(1) Suc] assms(2) by simp
qed

lemma lfp-Kleene-iter:
assumes mono $f$
and $(f \sim Suc k) \bot = (f \sim k) \bot$
shows $lfp f = (f \sim k) \bot$
proof (rule antisym)
show $lfp f \leq (f \sim k) \bot$
proof (rule lfp-lowerbound)
show $(f \sim k) \bot \leq (f \sim k) \bot$
using assms(2) by simp
qed
show $(f \sim k) \bot \leq lfp f$
using Kleene-iter-lfpf[OF assms(1)] lfp-unfold[OF assms(1)] by simp
qed

lemma mono-pow: mono $f$ $\Rightarrow$ mono $(f \sim n)$ for $f : \alpha \Rightarrow \alpha ::$ complete-lattice by (induct $n$) (auto simp: mono-def)

lemma lfp-funpow:
assumes $f :$ mono $f$
shows $lfp (f \sim Suc n) = lfp f$
proof (rule antisym)
show $lfp f \leq lfp (f \sim Suc n)$
proof (rule lfp-lowerbound)
have $f (lfp (f \sim Suc n)) = lfp (\lambda x. f (f \sim n) x)$
unfolding funpow-Suc-right by (simp add: lfp-rolling f mono-pow comp-def)
then show $f (lfp (f \sim Suc n)) \leq lfp (f \sim Suc n)$
by (simp add: comp-def)
qed
have $(f \sim n) (lfp f) = lfp f$ for $n$
by (induct $n$) (auto intro: f lfp-fixpoint)
then show $lfp (f \sim Suc n) \leq lfp f$
by (intro lfp-lowerbound) (simp del: funpow.simps)
qed

lemma gfp-funpow:
assumes $f :$ mono $f$
shows \( gfp (f ^^ Suc n) = gfp f \)
proof (rule antisym)
  show \( gfp f \geq gfp (f ^^ Suc n) \)
  proof (rule gfp-upperbound)
    have \( f (gfp (f ^^ Suc n)) = gfp (\lambda x. f ((f ^^ n) x)) \)
    unfolding funpow-Suc-right by (simp add: gfp-rolling f mono-pow comp-def)
  then show \( f (gfp (f ^^ Suc n)) \geq gfp (f ^^ Suc n) \)
  by (simp add: comp-def)
qed

have \( (f ^^ n) (gfp f) = gfp f \) for \( n \)
by (induct \( n \)) (auto intro: f gfp-fixpoint)
then show \( gfp (f ^^ Suc n) \geq gfp f \)
by (intro gfp-upperbound) (simp del: funpow.simps)
qed

lemma Kleene-iter-gpfp:
  fixes \( f :: 'a::order-top \Rightarrow 'a \)
  assumes mono \( f \) and \( p \leq f p \)
  shows \( p \leq (f ^^ k) \) top
proof (induct \( k \))
  case 0
  show ?case by simp
next
  case Suc
  show ?case
  using monoD[OF assms(1) Suc] assms(2) by simp
qed

lemma gfp-Kleene-iter:
  assumes mono \( f \)
  and \( (f ^^ Suc k) \) top = \( (f ^^ k) \) top
  shows \( gfp f = (f ^^ k) \) top
  (is \?lhs = \?rhs)
proof (rule antisym)
  have \?rhs \leq f \?rhs
  using assms(2) by simp
  then show \?rhs \leq \?lhs
  by (rule gfp-upperbound)
  show \?lhs \leq \?rhs
  using Kleene-iter-gpfp[OF assms(1)] gfp-unfold[OF assms(1)] by simp
qed

15.7 Embedding of the naturals into any \( \text{semiring-1} \): of-nat

context \( \text{semiring-1} \)
begin

definition of-nat :: \( \text{nat} \Rightarrow 'a \)


where of-nat \( n = (\text{plus} \ 1 \ ^{\wedge} \ n) \ 0 \)

**lemma** of-nat-simps [simp]:
* shows of-nat-0: of-nat \( 0 = 0 \)
  and of-nat-Suc: of-nat (Suc \( m \)) = 1 + of-nat \( m \)
  by (simp-all add: of-nat-def)

**lemma** of-nat-1 [simp]: of-nat \( 1 = 1 \)
  by (simp add: of-nat-def)

**lemma** of-nat-add [simp]: of-nat \( (m + n) = of-nat \( m + of-nat \( n \) \)
  by (induct \( m \)) (simp-all add: ac-simps)

**lemma** of-nat-mult [simp]: of-nat \( (m * n) = of-nat \( m * of-nat \( n \) \)
  by (induct \( m \)) (simp-all add: ac-simps distrib-right)

**lemma** mult-of-nat-commute: of-nat \( x * y = y * of-nat \( x \) \)
  by (induct \( x \)) (simp-all add: algebra-simps)

**primrec** of-nat-aux :: \( 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \)
  where of-nat-aux inc 0 i = i
  | of-nat-aux inc (Suc \( n \)) \( i = of-nat-aux \( inc \( i \) \) \)

**lemma** of-nat-code: of-nat \( n = of-nat-aux \( (\lambda i. \ i + 1) \) \)
  proof (induct \( n \))
  case 0
    then show ?case by simp
  next
  case (Suc \( n \))
  have \( \forall i. of-nat-aux \( (\lambda i. \ i + 1) \) \) \( (i + 1) = of-nat-aux \( (\lambda i. \ i + 1) \) \)
    by (induct \( n \)) simp-all
  from this [of 0] have of-nat-aux \( (\lambda i. \ i + 1) \) \( n \) \( 1 = of-nat-aux \( (\lambda i. \ i + 1) \) \)
    + 1
    by simp
  with Suc show ?case
    by (simp add: add.commute)
  qed

**lemma** of-nat-of-bool [simp]:
  of-nat \( (\text{of-bool} \ P) = \text{of-bool} \ P \)
  by auto

end

**declare** of-nat-code [code]

**context** semiring-1-cancel
**begin**
lemma of-nat-diff:
  of-nat (m − n) = of-nat m − of-nat n if \( n \leq m \)

proof -
  from that obtain q where \( m = n + q \)
  by (blast dest: le-Suc-ex)
  then show \( \text{thesis} \)
  by simp
qed

end

Class for unital semirings with characteristic zero. Includes non-ordered rings like the complex numbers.

class semiring-char-0 = semiring-1 +
  assumes inj-of-nat: inj of-nat
begin

lemma of-nat-eq-iff [simp]: of-nat m = of-nat n \( \longleftrightarrow \) m = n
  by (auto intro: inj-of-nat injD)

Special cases where either operand is zero

lemma of-nat-0-eq-iff [simp]: 0 = of-nat n \( \longleftrightarrow \) 0 = n
  by (fact of-nat-eq-iff [of 0 n, unfolded of-nat-0])

lemma of-nat-eq-0-iff [simp]: of-nat m = 0 \( \longleftrightarrow \) m = 0
  by (fact of-nat-eq-0-iff [of m 0, unfolded of-nat-0])

lemma of-nat-1-eq-iff [simp]: 1 = of-nat n \( \longleftrightarrow \) n = 1
  using of-nat-eq-iff by fastforce

lemma of-nat-eq-1-iff [simp]: of-nat n = 1 \( \longleftrightarrow \) n = 1
  using of-nat-eq-iff by fastforce

lemma of-nat-neq-0 [simp]: of-nat (Suc n) \( \neq \) 0
  unfolding of-nat-eq-0-iff by simp

lemma of-nat-0-neq [simp]: 0 \( \neq \) of-nat (Suc n)
  unfolding of-nat-0-eq-iff by simp

end

class ring-char-0 = ring-1 + semiring-char-0

class linordered-nonzero-semiring

begin

lemma of-nat-0-le-iff [simp]: 0 \( \leq \) of-nat n
  by (induct n) simp-all
lemma of-nat-less-0-iff [simp]: \( \neg \text{of-nat } m < 0 \)
  by (simp add: not-less)

lemma of-nat-mono [simp]: \( i \leq j \implies \text{of-nat } i \leq \text{of-nat } j \)
  by (auto simp: le_iff_add intro!: add-increasing2)

lemma of-nat-less-iff [simp]: \( \text{of-nat } m < \text{of-nat } n \iff m < n \)
proof (induct m n rule: diff-induct)
  case (1 m) then show ?case
    by auto
next
case (2 n) then show ?case
  by (simp add: add-pos-nonneg)
next
case (3 m n)
  then show ?case
    by (auto simp: add-commute[of 1] add-mono1 not-less add-right-mono leD)
qed

lemma of-nat-le-iff [simp]: \( \text{of-nat } m \leq \text{of-nat } n \iff m \leq n \)
by (simp add: not-less [symmetric] linorder-not-less [symmetric])

lemma less-imp-of-nat-less: \( m < n \implies \text{of-nat } m < \text{of-nat } n \)
by simp

lemma of-nat-less-imp-less: \( \text{of-nat } m < \text{of-nat } n \implies m < n \)
by simp

Every linordered-nonzero-semiring has characteristic zero.

subclass semiring-char-0
  by standard (auto intro!: injI simp add: eq_iff)

Special cases where either operand is zero

lemma of-nat-le-0-iff [simp]: \( \text{of-nat } m \leq 0 \iff m = 0 \)
  by (rule of-nat-le-iff [of 0, simplified])

lemma of-nat-0-less-iff [simp]: \( 0 < \text{of-nat } n \iff 0 < n \)
  by (rule of-nat-less-iff [of 0, simplified])

end

context linordered-nonzero-semiring
begin

lemma of-nat-max: \( \text{of-nat } (\text{max } x \ y) = \text{max } (\text{of-nat } x) (\text{of-nat } y) \)
  by (auto simp: max_def ord_class.max_def)

lemma of-nat-min: \( \text{of-nat } (\text{min } x \ y) = \text{min } (\text{of-nat } x) (\text{of-nat } y) \)
THEORY "Nat"

by (auto simp: min-def ord-class.min-def)

end

context linordered-semidom
begin

subclass linordered-nonzero-semiring ..

subclass semiring-char-0 ..

end

context linordered-idom
begin

lemma abs-of-nat [simp]:
  \(|\text{of-nat } n\)| = \text{of-nat } n
  by (simp add: abs-if)

lemma sgn-of-nat [simp]:
  sgn (\text{of-nat } n) = \text{of-bool } (n > 0)
  by simp

end

lemma of-nat-id [simp]: \text{of-nat } n = n
  by (induct n) simp-all

lemma of-nat-eq-id [simp]: \text{of-nat} = id
  by (auto simp add: fun-eq-iff)

15.8 The set of natural numbers

context semiring-1
begin

definition Nats :: 'a set (\text{IN})
  where \text{IN} = range of-nat

lemma of-nat-in-Nats [simp]: \text{of-nat } n \in \text{IN}
  by (simp add: Nats-def)

lemma Nats-0 [simp]: 0 \in \text{IN}
  apply (simp add: Nats-def)
  apply (rule range-eql)
  apply (rule of-nat-0 [symmetric])
  done
lemma Nats-1 [simp]: \(1 \in \mathbb{N}\)
apply (simp add: Nats-def)
apply (rule range-eqI)
apply (rule of-nat-1 [symmetric])
done

lemma Nats-add [simp]: \(a \in \mathbb{N} \Longrightarrow b \in \mathbb{N} \Longrightarrow a + b \in \mathbb{N}\)
apply (auto simp add: Nats-def)
apply (rule range-eqI)
apply (rule of-nat-add [symmetric])
done

lemma Nats-mult [simp]: \(a \in \mathbb{N} \Longrightarrow b \in \mathbb{N} \Longrightarrow a \cdot b \in \mathbb{N}\)
apply (auto simp add: Nats-def)
apply (rule range-eqI)
apply (rule of-nat-mult [symmetric])
done

lemma Nats-cases [cases set: Nats]:
assumes \(x \in \mathbb{N}\)
obtains (of-nat) \(n\) where \(x = \text{of-nat } n\)
unfolding Nats-def
proof  
  from \(x \in \mathbb{N}\) have \(x \in \text{range of-nat}\) unfolding Nats-def .  
  then obtain \(n\) where \(x = \text{of-nat } n\) .  
  then show thesis ..
qed

lemma Nats-induct [case-names of-nat, induct set: Nats]: \(x \in \mathbb{N} \Longrightarrow (\forall n. P\ (\text{of-nat } n)) \Longrightarrow P\ x\)
  by (rule Nats-cases) auto

end

lemma Nats-diff [simp]:
fixes \(a::'a::linordered-idom\)
assumes \(a \in \mathbb{N} \ b \in \mathbb{N} \ b \leq a\) shows \(a - b \in \mathbb{N}\)
proof  
  obtain \(i\) where \(i: a = \text{of-nat } i\)
     using Nats-cases assms by blast  
  obtain \(j\) where \(j: b = \text{of-nat } j\)
     using Nats-cases assms by blast  
  have \(j \leq i\)
     using \(b \leq a\) \(i\ j\ \text{of-nat-le-iff}\) by blast  
  then have \(*: \text{of-nat } i - \text{of-nat } j = (\text{of-nat } (i-j) :: 'a)\)
     by (simp add: of-nat-diff)
  then show \(?thesis\)
     by (simp add: * i j)
qed
15.9 Further arithmetic facts concerning the natural numbers

**lemma** subst-equals:
assumes $t = s$ and $u = t$
shows $u = s$
using assms(2,1) by (rule trans)

**locale** nat-arith
begin

**lemma** add1: $(A::'a::comm-monoid-add) \equiv k + a \Rightarrow A + b \equiv k + (a + b)$
by (simp only: ac-simps)

**lemma** add2: $(B::'a::comm-monoid-add) \equiv k + b \Rightarrow a + B \equiv k + (a + b)$
by (simp only: ac-simps)

**lemma** suc1: $A = k + a \Rightarrow Suc A \equiv k + Suc a$
by (simp only: add-Suc-right)

**lemma** rule0: $(a::'a::comm-monoid-add) \equiv a + 0$
by (simp only: add-0-right)

end

**ML-file** (Tools/nat-arith.ML)

**simproc-setup** nateq-cancel-sums
$\langle (l::nat) + m = n \mid (l::nat) = m + n \mid Suc m = n \mid m = Suc n) =$
$\langle fn\ \phi\ =\ try\ o\ Nat-Arith.cancel-eq-conv $\rangle$

**simproc-setup** natless-cancel-sums
$\langle (l::nat) + m < n \mid (l::nat) < m + n \mid Suc m < n \mid m < Suc n) =$
$\langle fn\ \phi\ =\ try\ o\ Nat-Arith.cancel-less-conv $\rangle$

**simproc-setup** natle-cancel-sums
$\langle (l::nat) + m \leq n \mid (l::nat) \leq m + n \mid Suc m \leq n \mid m \leq Suc n) =$
$\langle fn\ \phi\ =\ try\ o\ Nat-Arith.cancel-le-conv $\rangle$

**simproc-setup** natdiff-cancel-sums
$\langle (l::nat) + m - n \mid (l::nat) - (m + n) \mid Suc m - n \mid m - Suc n) =$
$\langle fn\ \phi\ =\ try\ o\ Nat-Arith.cancel-diff-conv $\rangle$

**context** order
begin

**lemma** lift-Suc-mono-le:
assumes mono: $\forall n. f n \leq f (Suc n)$
and $n \leq n'$
shows $f n \leq f n'$
proof (cases \( n < n' \))
  case True
  then show \( \text{thesis} \)
  by (induct \( n \) \( n' \) rule: less-Suc-induct) (auto intro: mono)
next
  case False
  with \( n \leq n' \) show \( \text{thesis} \) by auto
qed

lemma lift-Suc-antimono-le:
  assumes mono: \( \forall n. \ f \ n \geq f \ (\text{Suc} \ n) \)
  and \( n \leq n' \)
  shows \( f \ n \geq f \ n' \)
proof (cases \( n < n' \))
  case True
  then show \( \text{thesis} \)
  by (induct \( n \) \( n' \) rule: less-Suc-induct) (auto intro: mono)
next
  case False
  with \( n \leq n' \) show \( \text{thesis} \) by auto
qed

lemma lift-Suc-mono-less:
  assumes mono: \( \forall n. \ f \ n < f \ (\text{Suc} \ n) \)
  and \( n < n' \)
  shows \( f \ n < f \ n' \)
  using \( n < n' \) by (induct \( n \) \( n' \) rule: less-Suc-induct) (auto intro: mono)

lemma lift-Suc-mono-less-iff: \( \forall n. \ f \ n < f \ (\text{Suc} \ n) \) \( \Rightarrow \) \( f \ n < f \ m \longleftrightarrow n < m \)
  by (blast intro: less-asym' lift-Suc-mono-less [of \( f \)])
  dest: linorder-not-less[THEN iffD1] le-eq-less-or-eq [THEN iffD1]

end

lemma mono-iff-le-Suc: mono \( f \) \( \leftrightarrow \) \( \forall n. \ f \ n \leq f \ (\text{Suc} \ n) \)
  unfolding mono-def by (auto intro: lift-Suc-mono-le [of \( f \)])

lemma antimono-iff-le-Suc: antimono \( f \) \( \leftrightarrow \) \( \forall n. \ f \ (\text{Suc} \ n) \leq f \ n \)
  unfolding antimono-def by (auto intro: lift-Suc-antimono-le [of \( f \)])

lemma mono-nat-linear-lb:
  fixes \( f :: \text{nat} \Rightarrow \text{nat} \)
  assumes \( \forall m.\ n.\ m < n \Rightarrow f \ m < f \ n \)
  shows \( f \ m + k \leq f \ (m + k) \)
proof (induct \( k \))
  case 0
  then show \( \text{case} \) by simp
next
  case (Suc \( k \))
then have Suc \( f \cdot m + k \) ≤ Suc \( f \cdot (m + k) \) by simp
also from assms[of \( m + k \) Suc \( m + k \)] have Suc \( f \cdot (m + k) \) ≤ \( f \cdot (Suc \cdot m + k) \)
  by (simp add: Suc-le-eq)
finally show \(?\)case by simp
qed

Subtraction laws, mostly by Clemens Ballarin

lemma diff-less-mono:
  fixes \( a \) \( b \) \( c \) :: nat
  assumes \( a < b \) and \( c ≤ a \)
  shows \( a − c < b − c \)
proof –
  from assms obtain \( d \) \( e \) where \( b = c + (d + e) \) and \( a = c + e \) and \( d > 0 \)
    by (auto dest!: le-Suc-ex less-imp-Suc-add simp add: ac-simps)
  then show \(?\)thesis by simp
qed

lemma less-diff-conv: \( i < j − k \) ⟷ \( i + k < j \)
  for \( i \) \( j \) \( k \) :: nat
  by (cases \( k ≤ j \)) (auto simp add: not-le dest: less-imp-Suc-add le-Suc-ex)

lemma less-diff-conv2: \( k ≤ j \) ⟹ \( j − k < i ↔ j < i + k \)
  for \( j \) \( k \) :: nat
  by (auto dest: le-Suc-ex)

lemma le-diff-conv: \( j − k ≤ i \) ⟷ \( j ≤ i + k \)
  for \( j \) \( k \) :: nat
  by (cases \( k ≤ j \)) (auto simp add: not-le dest: less-imp-Suc-add le-Suc-ex)

lemma diff-diff-cancel [simp]: \( i ≤ n \) ⟹ \( n − (n − i) = i \)
  for \( i \) \( n \) :: nat
  by (auto dest: le-Suc-ex)

lemma diff-less [simp]: \( 0 < n \) ⟹ \( 0 < m \) ⟹ \( m − n < m \)
  for \( i \) \( n \) :: nat
  by (auto dest: less-imp-Suc-add)

Simplification of relational expressions involving subtraction

lemma diff-diff-eq: \( k ≤ m \) ⟹ \( k ≤ n \) ⟹ \( m − k − (n − k) = m − n \)
  for \( m \) \( n \) \( k \) :: nat
  by (auto dest!: le-Suc-ex)

hide-fact (open) diff-diff-eq

lemma eq-diff-iff: \( k ≤ m \) ⟹ \( k ≤ n \) ⟹ \( m − k = n − k ↔ m = n \)
  for \( m \) \( n \) \( k \) :: nat
  by (auto dest: le-Suc-ex)
lemma less-diff-iff: \( k \leq m \implies k \leq n \implies m - k < n - k \iff m < n \)
for \( m, n, k :: \text{nat} \)
by (auto dest!: le-Suc-ex)

lemma le-diff-iff: \( k \leq m \implies k \leq n \implies m - k \leq n - k \iff m \leq n \)
for \( m, n, k :: \text{nat} \)
by (auto dest: le-Suc-ex)

lemma le-diff-iff': \( a \leq c \implies b \leq c \implies c - a \leq c - b \iff b \leq a \)
for \( a, b, c :: \text{nat} \)
by (force dest: le-Suc-ex)

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma diff-le-mono: \( m \leq n \implies m - l \leq n - l \)
for \( m, n, l :: \text{nat} \)
by (auto dest: less-imp-le less-imp-Suc-add split: nat-diff-split)

lemma diff-le-mono2: \( m \leq n \implies l - n \leq l - m \)
for \( m, n, l :: \text{nat} \)
by (auto dest: less-imp-le le-Suc-ex less-imp-Suc-add less-le-trans split: nat-diff-split)

lemma diff-less-mono2: \( m < n \implies m < l \implies l - n < l - m \)
for \( m, n, l :: \text{nat} \)
by (auto dest: less-imp-Suc-add split: nat-diff-split)

lemma diffs0-imp-equal: \( m - n = 0 \implies n - m = 0 \implies m = n \)
for \( m, n :: \text{nat} \)
by (simp split: nat-diff-split)

lemma min-diff: \( \text{min} \ (m - i) \ (n - i) = \text{min} \ m \ n - i \)
for \( m, n, i :: \text{nat} \)
by (cases m n rule: le-cases)
  (auto simp add: not-le min.absorb1 min.absorb2 min.absorb-iff1 [symmetric]
  diff-le-mono)

lemma inj-on-diff-nat:
  fixes \( k :: \text{nat} \)
  assumes \( \forall n. \ n \in N \implies k \leq n \)
  shows inj-on \( \lambda n. \ n - k \) \( N \)
proof (rule inj-onI)
  fix \( x, y \)
  assume \( a: x \in N \ y \in N \ x - k = y - k \)
  with \( \text{assms} \)
  have \( x - k + k = y - k + k \) \( \text{by auto} \)
  with \( a \ \text{assms} \)
  show \( x = y \) \( \text{by (auto simp add: eq-diff-iff)} \)
qed

Rewriting to pull differences out

lemma diff-diff-right [simp]: \( k \leq j \implies i - (j - k) = i + k - j \)
for \( i, j, k :: \text{nat} \)
by (fact diff-diff-right)

lemma diff-Suc-diff-eq1 [simp]:
assumes $k \leq j$
shows $i - \text{Suc} (j - k) = i + k - \text{Suc} j$
proof –
  from assms have $*: \text{Suc} (j - k) = \text{Suc} j - k$
    by (simp add: Suc-diff-le)
  from assms have $k \leq \text{Suc} j$
    by (rule order-trans) simp
  with diff-diff-right [of $k$ Suc $j$ $i$] * show ?thesis
    by simp
qed

lemma diff-Suc-diff-eq2 [simp]:
assumes $k \leq j$
shows $\text{Suc} (j - k) - i = \text{Suc} j - (k + i)$
proof –
  from assms obtain $n$ where $j = k + n$
    by (auto dest: le-Suc-ex)
  moreover have $\text{Suc} n - i = (k + \text{Suc} n) - (k + i)$
    using add-diff-cancel-left [of $k$ Suc $n$ $i$] by simp
  ultimately show ?thesis by simp
qed

lemma Suc-diff-Suc:
assumes $n < m$
shows $\text{Suc} (m - \text{Suc} n) = m - n$
proof –
  from assms obtain $q$ where $m = n + \text{Suc} q$
    by (auto dest: less-imp-Suc-add)
  moreover define $r$ where $r = \text{Suc} q$
  ultimately have $\text{Suc} (m - \text{Suc} n) = r$ and $m = n + r$
    by simp-all
  then show ?thesis by simp
qed

lemma one-less-mult: $\text{Suc} 0 < n \Rightarrow \text{Suc} 0 < m \Rightarrow \text{Suc} 0 < m * n$
using less-1-mult [of $n$ $m$] by (simp add: ac-simps)

lemma n-less-m-mult-n: $0 < n \Rightarrow \text{Suc} 0 < m \Rightarrow n < m * n$
using mult-strict-right-mono [of 1 $m$ $n$] by simp

lemma n-less-n-mult-m: $0 < n \Rightarrow \text{Suc} 0 < m \Rightarrow n < n * m$
using mult-strict-left-mono [of 1 $m$ $n$] by simp

Induction starting beyond zero

lemma nat-induct-at-least [consumes 1, case-names base Suc]:
P $n$ if $n \geq m$
P $m \land n \geq m \Rightarrow P n \Rightarrow P (\text{Suc} n)$
proof
  
  define \( q \) where \( q = n - m \)

  with \( n \geq m \) have \( n = m + q \)

  by simp

  moreover have \( P (m + q) \)

  by (induction \( q \)) (use that \textbf{in} simp-all)

  ultimately show \( P n \)

  by simp

qed

lemma nat-induct-non-zero [consumes 1, case-names 1 Suc]:
\[
P n \quad \text{if} \quad n > 0 \quad P 1 \land n > 0 \Rightarrow P n \Rightarrow P (Suc n)
\]
proof

  from \( n > 0 \) have \( n \geq 1 \)

  by (cases \( n \)) simp-all

  moreover note \( P 1 \)

  moreover have \( \land n \geq 1 \Rightarrow P n \Rightarrow P (Suc n) \)

  using \( \land n > 0 \Rightarrow P n \Rightarrow P (Suc n) \)

  by (simp add: Suc-le-eq)

  ultimately show \( P n \)

  by (rule nat-induct-at-least)

qed

Specialized induction principles that work "backwards":

lemma inc-induct [consumes 1, case-names base step]:

assumes less: \( i \leq j \)

and base: \( P j \)

and step: \( \land n. \ i \leq n \Rightarrow n < j \Rightarrow P (Suc n) \Rightarrow P n \)

shows \( P i \)

using less step

proof (induct \( j - i \) arbitrary: \( i \))

  case \( (0 \ i) \)

  then have \( i = j \) by simp

  with base show \( \text{?case} \) by simp

next

  case \( (Suc \ d \ n) \)

  from Suc.hyps have \( n \neq j \) by auto

  with Suc have \( n < j \) by (simp add: less-le)

  from Suc \( d = j - n \) have \( d + 1 = j - n \) by simp

  then have \( d + 1 = j = n + 1 \) by simp

  then have \( d = j = n - 1 \) by simp

  then have \( d = j = Suc n \) by simp

  moreover from \( n < j \) have \( Suc n \leq j \) by (simp add: Suc-le-eq)

  ultimately have \( P (Suc n) \)

  proof (rule Suc.hyps)

    fix \( q \)

    assume Suc \( n \leq q \)

    then have \( n \leq q \) by (simp add: Suc-le-eq less-imp-le)
moreover assume $q < j$
moreover assume $P(Suc\ q)$
ultimately show $P\ q$ by (rule Suc.prems)
qed
with order-refl ($n < j$) show $P\ n$ by (rule Suc.prems)
qed

lemma strict-inc-induct [consumes 1, case-names base step]:
assumes less: $i < j$
and base: $\forall i, j. i = Suc\ i \implies P\ i$
and step: $\forall i. i < j \implies P(\ Suc\ i) \implies P\ i$
shows $P\ i$
using less proof (induct $j - i - 1$ arbitrary: $i$)
case (0 $i$)
from $i < j$ obtain $n$ where $j = i + n$ and $n > 0$
by (auto dest!: less-imp-Suc-add)
with 0 have $j = Suc\ i$
by (auto intro: order-antisym simp add: Suc-le-eq)
with base show ?case by simp
next
case (Suc $d\ i$)
from $Suc\ d = j - i - 1$ have $*: Suc\ d = j - Suc\ i$
by (simp add: diff-diff-add)
then have $Suc\ d - 1 = j - Suc\ i - 1$ by simp
then have $d = j - Suc\ i - 1$ by simp
moreover from $*$ have $j - Suc\ i \neq 0$ by auto
then have $Suc\ i < j$ by (simp add: not-le)
ultimately have $P(\ Suc\ i)$ by (rule Suc.hyps)
with $i < j$ show $P\ i$ by (rule step)
qed

lemma zero-induct-lemma: $P\ k \implies (\forall n. P(\ Suc\ n) \implies P\ n) \implies P(\ k - i)$
using inc-induct[of $k - i\ k\ P$, simplified] by blast

lemma zero-induct: $P\ k \implies (\forall n. P(\ Suc\ n) \implies P\ n) \implies P\ 0$
using inc-induct[of 0 $k\ P$] by blast

Further induction rule similar to \[ \forall i \leq \forall j; ?P\ ?j; \forall n. [\forall i \leq n; n < \forall j; ?P(\ Suc\ n)] \implies ?P\ ?n \] \implies ?P\ ?i.

lemma dec-induct [consumes 1, case-names base step]:
i $\leq$ j $\implies$ P i $\implies (\forall n. i \leq n \implies n < j \implies P\ n \implies P(\ Suc\ n)) \implies P\ j$
proof (induct $j$ arbitrary: $i$)
case 0
then show ?case by simp
next
case (Suc $j$)
from Suc.prems consider $i \leq j | i = Suc\ j$
by (auto simp add: le-Suc-eq)
then show ?case
proof cases
  case 1
  moreover have $j < \text{Suc } j$ by simp
  moreover have $P \ j$ using $(i \leq j) \ i P$
  proof (rule Suc.prems)
    fix $q$
    assume $i \leq q$
    moreover assume $q < j$ then have $q < \text{Suc } j$
      by (simp add: less-Suc-le)
    moreover assume $P \ q$
    ultimately show $P \ (\text{Suc } q)$
      by (rule Suc.prems)
  qed
  qed
next
  case 2
  with $(P \ i)$ show $P \ (\text{Suc } j)$ by simp
  qed
  qed

lemma transitive-stepwise-le:
  assumes $m \leq n \ \& \ x. \ R \ x \ x \ \& \ y z. \ R \ x \ y \Longrightarrow R \ y \ z \Longrightarrow R \ x \ z$ \ and \ $\forall n. \ R \ n$
  $(\text{Suc } n)$
  shows $R \ m \ n$
  using $(m \leq n)$
  by (induction rule: dec-induct) (use assms in blast)+

15.9.1 Greatest operator

lemma ex-has-greatest-nat:
  $P \ (k::nat) \Longrightarrow \ \forall y. \ P \ y \Longrightarrow y \leq b \Longrightarrow \ \exists x. \ P \ x \ \& \ \forall y. \ P \ y \Longrightarrow y \leq x$
proof (induction $b \ k$ arbitrary: $b \ k$ rule: less-induct)
  case less
  show $\exists \ case$
  proof cases
    assume $\exists n > k. \ P \ n$
    then obtain $n$ where $n > k. \ P \ n$ by blast
    have $n \leq b$ using $(P \ n)$ less.prems(2) by auto
    hence $b - n < b - k$
      by (rule diff_less_mono2 [OF $(k < n)$ less_trans [OF $(k < n)$]])
    from less.prems [OF this $(P \ n)$ less.prems(2)]
    show $\exists n > k. \ P \ n$
  qed
next
  assume $\neg \ (\exists n > k. \ P \ n)$
  hence $\forall y. \ P \ y \Longrightarrow y \leq k$ by (auto simp: not_less)
  thus $\exists \ case$ using less.prems(1) by auto
  qed
  qed

lemma GreatestI-nat:
THEORY "Nat"

[ \[ P(k::nat) ; \forall y. P y \rightarrow y \leq b \] \Rightarrow P \text{ (Greatest P)} \]
apply (drule (1) ex-has-greatest-nat)
using GreatestI2-order by auto

lemma Greatest-le-nat:
[ \[ P(k::nat) ; \forall y. P y \rightarrow y \leq b \] \Rightarrow k \leq (\text{Greatest P}) \]
apply (erule (1) ex-has-greatest-nat)
using GreatestI2-order
where \( P = P \) and \( Q = \langle \lambda x. k \leq x \rangle \)
by auto

lemma GreatestI-ex-nat:
[ \[ \exists k::nat. P k ; \forall y. P y \rightarrow y \leq b \] \Rightarrow P \text{ (Greatest P)} \]
apply (erule exE)
apply (erule (1) GreatestI-nat)
done

15.10 Monotonicity of \( \text{funpow} \)

lemma funpow-increasing: \(...\) \Rightarrow \( \text{mono} f \Rightarrow (f ^^ n) \top \leq (f ^^ m) \top \)
for \( f :: \text{'}a::\{\text{lattice,order-top}\} \Rightarrow \text{'}a \)
by (induct rule: inc-induct)
(auto simp del: funpow.simps(2) simp add: funpow-Suc-right
 intro: order-trans[OF - funpow-mono])

lemma funpow-decreasing: \(...\) \Rightarrow \( \text{mono} f \Rightarrow (f ^^ m) \bot \leq (f ^^ n) \bot \)
for \( f :: \text{'}a::\{\text{lattice,order-bot}\} \Rightarrow \text{'}a \)
by (induct rule: dec-induct)
(auto simp del: funpow.simps(2) simp add: funpow-Suc-right
 intro: order-trans[OF - funpow-mono])

lemma mono-funpow: \( \text{mono} Q \Rightarrow \text{mono} \langle \lambda i. (Q ^^ i) \bot \rangle \)
for \( Q :: \text{'}a::\{\text{lattice,order-bot}\} \Rightarrow \text{'}a \)
by (auto intro!: funpow-decreasing simp: mono-def)

lemma antimono-funpow: \( \text{mono} Q \Rightarrow \text{antimono} \langle \lambda i. (Q ^^ i) \top \rangle \)
for \( Q :: \text{'}a::\{\text{lattice,order-top}\} \Rightarrow \text{'}a \)
by (auto intro!: funpow-increasing simp: antimono-def)

15.11 The divides relation on \( \text{nat} \)

lemma dvd-1-left [iff]: \( \text{Suc 0 dvd k} \)
by (simp add: dvd-def)

lemma dvd-1-iff-1 [simp]: \( m \text{ dvd Suc 0 } \iff m = \text{Suc 0} \)
by (simp add: dvd-def)

lemma nat-dvd-1-iff-1 [simp]: \( m \text{ dvd } 1 \iff m = 1 \)
for \( m :: \text{nat} \)
by (simp add: dvd-def)

lemma dvd-antisym: \( m \text{ dvd } n \Rightarrow n \text{ dvd } m \Rightarrow m = n \)
for \( m, n :: \text{nat} \)
unfolding dvd-def by (force dest: mult-eq-self-implies-10 simp add: mult.assoc)

lemma dvd-diff-nat \([\text{simp}]: k \text{ dvd } m \implies k \text{ dvd } n \implies k \text{ dvd } (m - n)\)
for \( k, m, n :: \text{nat} \)
unfolding dvd-def by (blast intro: right-diff-distrib \([\text{symmetric}]\))

lemma dvd-diffD: \( k \text{ dvd } m - n \implies k \text{ dvd } n \implies n \leq m \implies k \text{ dvd } m \)
for \( k, m, n :: \text{nat} \)
apply (erule linorder-not-less \([\text{THEN iffD2, THEN add-diff-inverse, THEN subst}]\))
apply (blast intro: dvd-add)
done

lemma dvd-diffD1: \( k \text{ dvd } m - n \implies k \text{ dvd } m \implies n \leq m \implies k \text{ dvd } n \)
for \( k, m, n :: \text{nat} \)
by (drule-tac m = m in dvd-diff-nat) auto

lemma dvd-mult-cancel:
fixes \( m, n, k :: \text{nat} \)
assumes \( k \cdot m \text{ dvd } k \cdot n \) and \( 0 < k \)
shows \( m \text{ dvd } n \)
proof –
  from assms(1) obtain \( q \) where \( k \cdot n = (k \cdot m) \cdot q \) ..
  then have \( k \cdot n = k \cdot (m \cdot q) \) by (simp add: ac-simps)
  with \( 0 < k \) have \( n = m \cdot q \) by (auto simp add: mult-left-cancel)
  then show \( \text{thesis} \) ..
qed

lemma dvd-mult-cancel1: \( 0 < m \implies m \cdot n \text{ dvd } m \iff n = 1 \)
for \( m :: \text{nat} \)
apply auto
apply (subgoal-tac m \( \cdot n \text{ dvd } m \cdot 1 \))
apply (drule dvd-mult-cancel)
apply auto
done

lemma dvd-mult-cancel2: \( 0 < m \implies n \cdot m \text{ dvd } m \iff n = 1 \)
for \( m :: \text{nat} \)
using dvd-mult-cancel1 [of \( m \)] by (simp add: ac-simps)

lemma dvd-imp-le: \( k \text{ dvd } n \implies 0 < n \implies k \leq n \)
for \( k :: \text{nat} \)
by (auto elim!: dvdE) (auto simp add: gr0-conv-Suc)

lemma nat-dvd-not-less: \( 0 < m \implies m < n \implies \neg n \text{ dvd } m \)
for \( m :: \text{nat} \)
by (auto elim!: dvdE) (auto simp add: gr0-conv-Suc)
THEORY "Nat"

lemma less-eq-dvd-minus:
fixes m n :: nat
assumes m ≤ n
shows m dvd n ←→ m dvd n – m
proof –
from assms have n = m + (n – m) by simp
then obtain q where n = m + q ..
then show ?thesis by (simp add: add.commute[of m])
qed

lemma dvd-minus-self: m dvd n – m ←→ n < m ∨ m dvd n
for m n :: nat
by (cases n < m) (auto elim!: dvdE simp add: not-less le-imp-diff-is-add dest: less-imp-le)

lemma dvd-minus-add:
fixes m n q r :: nat
assumes q ≤ n q ≤ r * m
shows m dvd n – q ←→ m dvd n + (r * m – q)
proof –
have m dvd n – q ←→ m dvd r * m + (n – q)
  using dvd-add-times-triv-left-iff[of m r] by simp
also from assms have ... ←→ m dvd r * m + n – q by simp
also from assms have ... ←→ m dvd (r * m – q) + n by simp
also have ... ←→ m dvd n + (r * m – q) by (simp add: add.commute)
finally show ?thesis .
qed

15.12 Aliasses

lemma nat-mult-1: 1 * n = n
for n :: nat
by (fact mult-1-left)

lemma nat-mult-1-right: n * 1 = n
for n :: nat
by (fact mult-1-right)

lemma nat-add-left-cancel: k + m = k + n ←→ m = n
for k m n :: nat
by (fact add-left-cancel)

lemma nat-add-right-cancel: m + k = n + k ←→ m = n
for k m n :: nat
by (fact add-right-cancel)

lemma diff-mult-distrib: (m – n) * k = (m * k) – (n * k)
for k m n :: nat
by (fact left-diff-distrib)
lemma diff-mult-distrib2: \( k \times (m - n) = (k \times m) - (k \times n) \)  
for \( k m n :: \text{naturals} \)  
by (fact right-diff-distrib')

lemma le-add-diff: \( k \leq n \implies m \leq n + m - k \)  
for \( k m n :: \text{naturals} \)  
by (fact le-add-diff)

lemma le-diff-conv2: \( k \leq j \implies (i \leq j - k) = (i + k \leq j) \)  
for \( i j k :: \text{naturals} \)  
by (fact le-diff-conv2)

lemma diff-self-eq-0 [simp]: \( m - m = 0 \)  
for \( m :: \text{naturals} \)  
by (fact diff-cancel)

lemma diff-diff-left [simp]: \( i - j - k = i - (j + k) \)  
for \( i j k :: \text{naturals} \)  
by (fact diff-diff-add)

lemma diff-commute: \( i - j - k = i - k - j \)  
for \( i j k :: \text{naturals} \)  
by (fact diff-right-commute)

lemma diff-add-inverse: \( (n + m) - n = m \)  
for \( m n :: \text{naturals} \)  
by (fact add-diff-cancel-left')

lemma diff-add-inverse2: \( (m + n) - n = m \)  
for \( m n :: \text{naturals} \)  
by (fact add-diff-cancel-right')

lemma diff-cancel: \( (k + m) - (k + n) = m - n \)  
for \( k m n :: \text{naturals} \)  
by (fact add-diff-cancel-left)

lemma diff-cancel2: \( (m + k) - (n + k) = m - n \)  
for \( k m n :: \text{naturals} \)  
by (fact add-diff-cancel-right)

lemma diff-add-0: \( n - (n + m) = 0 \)  
for \( m n :: \text{naturals} \)  
by (fact diff-add-zero)

lemma add-mult-distrib2: \( k \times (m + n) = (k \times m) + (k \times n) \)  
for \( k m n :: \text{naturals} \)  
by (fact distrib-left)
lemmas nat-distrib =
  add-mult-distrib distrib-left diff-mult-distrib diff-mult-distrib2

15.13 Size of a datatype value

class size =
  fixes size :: 'a ⇒ nat — see further theory Wellfounded

instantiation nat :: size
begin

definition size-nat where [simp, code]: size (n::nat) = n

instance ..
end

lemmas size-nat = size-nat-def

15.14 Code module namespace

code-identifier
  code-module Nat → (SML) Arith and (OCaml) Arith and (Haskell) Arith

hide-const (open) of-nat-aux

end

16 Fields

theory Fields
imports Nat
begin

context idom
begin

lemma inj-mult-left [simp]: inj ((*) a) ←→ a ≠ 0
(is (?P ←→ ?Q))
proof
  assume ?P
  show ?Q
  proof
    assume ⟨a = 0⟩
    with ⟨?P⟩ have inj ((*) 0)
      by simp
    moreover have 0 * 0 = 0 * 1
      by simp
    ultimately have 0 = 1
      by (rule injD)
then show False
  by simp
qed

next
  assume ?Q then show ?P
  by (auto intro: injI)
qed

end

16.1 Division rings

A division ring is like a field, but without the commutativity requirement.

class inverse = divide +
fixes inverse :: 'a ⇒ 'a
begin

abbreviation inverse-divide :: 'a ⇒ 'a ⇒ 'a (infixl '/ 70)
where
  inverse-divide ≡ divide

end

Setup for linear arithmetic prover

ML-file ⟨~/src/Provers/Arith/fast-lin-arith.ML⟩
ML-file ⟨Tools/lin-arith.ML⟩
setup ⟨Lin-Arith.global-setup⟩
declaration ⟨K (Lin-Arith.init-arith-data
  #> Lin-Arith.add-discrete-type type-name ⟨nat⟩
  #> Lin-Arith.add-lessD @{thm Suc-leI}
  #> Lin-Arith.add-simps @{thms simp-thms ring-distrib if-True if-False
    minus-diff-eq
    add-0-left add-0-right order-less-irrefl
    zero-neq-one zero-less-one zero-le-one
    zero-neq-one [THEN not-sym] not-one-le-zero not-one-less-zero
    add-Suc add-Suc-right nat.inject
    Suc-le-mono Suc-less-eq Zero-not-Suc
    Suc-not-Zero le-0-eq One-not-def}⟩
  #> Lin-Arith.add-simprocs [simproc ⟨group-cancel-add⟩, simproc ⟨group-cancel-diff⟩,
    simproc ⟨group-cancel-eq⟩, simproc ⟨group-cancel-le⟩,
    simproc ⟨group-cancel-less⟩,
    simproc ⟨nateq-cancel-sums⟩, simproc ⟨natless-cancel-sums⟩,
    simproc ⟨natle-cancel-sums⟩])

simproc-setup fast-arith-nat ((m::nat) < n | (m::nat) ≤ n | (m::nat) = n) =
  ⟨K Lin-Arith.simproc⟩ — Because of this simproc, the arithmetic solver is really only useful to detect inconsistencies among the premises for subgoals which are not themselves (in)equalities, because the latter activate fast-nat-arith-simproc anyway.
However, it seems cheaper to activate the solver all the time rather than add the additional check.

lemmas [arith-split] = nat-diff-split split-min split-max

Lemmas divide-simps move division to the outside and eliminates them on (in)equalities.

named-theorems divide-simps rewrite rules to eliminate divisions

class division-ring = ring-1 + inverse +
  assumes left-inverse [simp]: a ≠ 0 ⇒ inverse a * a = 1
  assumes right-inverse [simp]: a ≠ 0 ⇒ a * inverse a = 1
  assumes divide-inverse: a / b = a * inverse b
  assumes inverse-zero [simp]: inverse 0 = 0
begin

subclass ring-1-no-zero-divisors
proof
  fix a b :: 'a
  assume a: a ≠ 0 and b: b ≠ 0
  show a * b ≠ 0
  proof
    assume ab: a * b = 0
    hence 0 = inverse a * (a * b) * inverse b by simp
    also have ... = (inverse a * a) * (b * inverse b)
      by (simp only: mult.assoc)
    also have ... = 1 using a b by simp
    finally show False by simp
  qed
qed

lemma nonzero-imp-inverse-nonzero:
  a ≠ 0 ⇒ inverse a ≠ 0
proof
  assume ianz: inverse a = 0
  assume a ≠ 0
  hence 1 = a * inverse a by simp
  also have ... = 0 by (simp add: ianz)
  finally have 1 = 0 .
  thus False by (simp add: eq-commute)
qed

lemma inverse-zero-imp-zero:
  inverse a = 0 ⇒ a = 0
apply (rule classical)
apply (drule nonzero-imp-inverse-nonzero)
apply auto
done
lemma inverse-unique:
assumes ab: \( a \ast b = 1 \)
shows inverse \( a = b \)
proof
have \( a \neq 0 \) using \( ab \) by (cases \( a = 0 \)) simp-all
moreover have inverse \( a \ast (a \ast b) = \) inverse \( a \) by (simp add: ab)
ultimately show \( \)thesis by (simp add: mult.assoc [symmetric])
qed

lemma nonzero-inverse-minus-eq:
\( a \neq 0 \imp \) inverse \( (-a) = -\) inverse \( a \)
by (rule inverse-unique) simp

lemma nonzero-inverse-inverse-eq:
\( a \neq 0 \imp \) inverse \( (\)inverse \( a) = a \)
by (rule inverse-unique) simp

lemma nonzero-inverse-eq-imp-eq:
assumes inverse \( a = \) inverse \( b \)
and \( a \neq 0 \) and \( b \neq 0 \)
shows \( a = b \)
proof
from \( inverse a = inverse b \)
have inverse \( (inverse a) = inverse (inverse b) \) by (rule arg-cong)
with \( a \neq 0 \) and \( b \neq 0 \) show \( a = b \)
by (simp add: nonzero-inverse-inverse-eq)
qed

lemma inverse-1 [simp]: inverse \( 1 = 1 \)
by (rule inverse-unique) simp

lemma nonzero-inverse-mult-distrib:
assumes \( a \neq 0 \) and \( b \neq 0 \)
shows inverse \( (a \ast b) = \) inverse \( b \ast inverse a \)
proof
have \( a \ast (b \ast inverse b) \ast inverse a = 1 \) using assms by simp
hence \( a \ast b \ast (inverse b \ast inverse a) = 1 \) by (simp only: mult.assoc)
thus \( \)thesis by (rule inverse-unique)
qed

lemma division-ring-inverse-add:
\( a \neq 0 \imp b \neq 0 \imp inverse a \ast inverse b = inverse a \ast (a + b) \ast inverse b \)
by (simp add: algebra-simps)

lemma division-ring-inverse-diff:
\( a \neq 0 \imp b \neq 0 \imp inverse a - inverse b = inverse a \ast (b - a) \ast inverse b \)
by (simp add: algebra-simps)

lemma right-inverse-eq: \( b \neq 0 \imp a / b = 1 \longleftrightarrow a = b \)
proof
assume neq: \( b \neq 0 \)

{  
  hence \( a = (a / b) \cdot b \) by (simp add: divide-inverse mult.assoc)  
  also assume \( a / b = 1 \)  
  finally show \( a = b \) by simp  
next  
  assume \( a = b \)  
  with neq show \( a / b = 1 \) by (simp add: divide-inverse)  
}

qed

lemma nonzero-inverse-eq-divide: \( a \neq 0 \implies \text{inverse } a = 1 / a \)
by (simp add: divide-inverse)

lemma divide-self [simp]: \( a \neq 0 \implies a / a = 1 \)
by (simp add: divide-inverse)

lemma inverse-eq-divide [field-simps, field-split-simps, divide-simps]: \( \text{inverse } a = 1 / a \)
by (simp add: divide-inverse)

lemma add-divide-distrib: \( (a + b) / c = a / c + b / c \)
by (simp add: divide-inverse algebra-simps)

lemma times-divide-eq-right [simp]: \( a * (b / c) = (a * b) / c \)
by (simp add: divide-inverse mult.assoc)

lemma minus-divide-left: \( -(a / b) = (-a) / b \)
by (simp add: divide-inverse)

lemma nonzero-minus-divide-right: \( b \neq 0 \implies -(a / b) = a / (-b) \)
by (simp add: divide-inverse nonzero-inverse-minus-eq)

lemma nonzero-minus-divide-divide: \( b \neq 0 \implies (-a) / (-b) = a / b \)
by (simp add: divide-inverse nonzero-inverse-minus-eq)

lemma divide-minus-left [simp]: \( -(a / b) = - (a / b) \)
by (simp add: divide-inverse)

lemma diff-divide-distrib: \( (a - b) / c = a / c - b / c \)
using add-divide-distrib [of a - b c] by simp

lemma nonzero-eq-divide-eq [field-simps]: \( c \neq 0 \implies a = b / c \iff a * c = b \)
proof -  
  assume [simp]: \( c \neq 0 \)  
  have \( a = b / c \iff a * c = (b / c) * c \) by simp  
  also have \( ... \iff a * c = b \) by (simp add: divide-inverse mult.assoc)  
  finally show \( ?thesis \).
qed
lemma nonzero-divide-eq-eq [field-simps]: $c \neq 0 \implies b \div c = a \iff b = a \times c$
proof
  assume [simp]: $c \neq 0$
  have $b \div c = a \iff (b \div c) \times c = a \times c$ by simp
  also have $\ldots \iff b = a \times c$ by (simp add: divide-inverse mult.assoc)
  finally show $\vdots$ thesis .
qed

lemma nonzero-neg-divide-eq-eq [field-simps]: $b \neq 0 \implies -(a \div b) = c \iff -a = c \times b$
  using nonzero-divide-eq-eq[of $b - a c$] by simp

lemma nonzero-neg-divide-eq-eq2 [field-simps]: $b \neq 0 \implies c = -(a \div b) \iff c \times b = -a$
  using nonzero-neg-divide-eq-eq[of $b a c$] by auto

lemma divide-eq-imp: $c \neq 0 \implies b = a \times c \implies b \div c = a$
  by (simp add: divide-inverse mult.assoc)

lemma eq-divide-imp: $c \neq 0 \implies a \times c = b \implies a = b \div c$
  by (drule sym) (simp add: divide-inverse mult.assoc)

lemma add-divide-eq-iff [field-simps]:
  $z \neq 0 \implies x + y \div z = (x \times z + y) \div z$
  by (simp add: add-divide-distrib nonzero-eq-divide-eq)

lemma divide-add-eq-iff [field-simps]:
  $z \neq 0 \implies x \div z + y = (x + y \times z) \div z$
  by (simp add: add-divide-distrib nonzero-eq-divide-eq)

lemma diff-divide-eq-iff [field-simps]:
  $z \neq 0 \implies x - y \div z = (x \times z - y) \div z$
  by (simp add: diff-divide-distrib nonzero-eq-divide-eq)

lemma minus-divide-add-eq-iff [field-simps]:
  $z \neq 0 \implies -(x \div z) + y = (- x + y \times z) \div z$
  by (simp add: add-divide-distrib nonzero-eq-divide-eq)

lemma divide-diff-eq-iff [field-simps]:
  $z \neq 0 \implies x \div z - y = (x - y \times z) \div z$
  by (simp add: field-simps)

lemma minus-divide-diff-eq-iff [field-simps]:
  $z \neq 0 \implies -(x \div z) - y = (- x - y \times z) \div z$
  by (simp add: divide-diff-eq-iff[symmetric])

lemma division-ring-divide-zero [simp]:
  $a \div 0 = 0$
by (simp add: divide-inverse)

lemma divide-self-if [simp]:
  \( a / a = (\text{if } a = 0 \text{ then } 0 \text{ else } 1) \)
by simp

lemma inverse-nonzero-iff-nonzero [simp]:
  inverse a = 0 \iff a = 0
by rule (fact inverse-zero-imp-zero, simp)

lemma inverse-minus-eq [simp]:
  inverse (- a) = - inverse a
proof cases
  assume a=0 thus ?thesis by simp
next
  assume a\neq0
  thus ?thesis by (simp add: nonzero-inverse-minus-eq)
qed

lemma inverse-inverse-eq [simp]:
  inverse (inverse a) = a
proof cases
  assume a=0 thus ?thesis by simp
next
  assume a\neq0
  thus ?thesis by (simp add: nonzero-inverse-inverse-eq)
qed

lemma inverse-eq-imp-eq:
  inverse a = inverse b \implies a = b
by (drule arg-cong [where f=inverse], simp)

lemma inverse-eq-iff-eq [simp]:
  inverse a = inverse b \iff a = b
by (force dest!: inverse-eq-imp-eq)

lemma mult-commute-imp-mul-inverse-commute:
  assumes y * x = x * y
  shows  inverse y * x = x * inverse y
proof (cases y=0)
  case False
  hence x * inverse y = inverse y * y * x * inverse y
  by simp
also have \ldots = inverse y * (x * y * inverse y)
  by (simp add: mult.assoc assms)
finally show ?thesis by (simp add: mult.assoc False)
qed simp

lemmas mult-inverse-of-nat-commute =
lemma divide-divide-eq-left':
(a / b) / c = a / (c * b)
by (cases b = 0 ∨ c = 0)
  (auto simp: divide-inverse mult.assoc nonzero-inverse-mult-distrib)

lemma add-divide-eq-if-simps [field-split-simps, divide-simps]:
a + b / z = (if z = 0 then a else (a * z + b) / z)
a / z + b = (if z = 0 then b else (a + b * z) / z)
− (a / z) + b = (if z = 0 then b else (−a + b * z) / z)
a − b / z = (if z = 0 then a else (a * z − b) / z)
a / z − b = (if z = 0 then −b else (a − b * z) / z)

lemma [field-split-simps, divide-simps]:
shows divide-eq-eq: b / c = a → (if c ≠ 0 then b = a * c else a = 0)
  and eq-divide-eq: a = b / c ↔ (if c ≠ 0 then a * c = b else a = 0)
  and minus-divide-eq-eq: − (b / c) = a ↔ (if c ≠ 0 then −b = a * c else a = 0)
  and eq-minus-divide-eq: a = − (b / c) ↔ (if c ≠ 0 then a * c = −b else a = 0)
by (auto simp add: field-simps)

16.2 Fields

class field = comm-ring-1 + inverse +
  assumes field-inverse: a ≠ 0 → inverse a * a = 1
  assumes field-divide-inverse: a / b = a * inverse b
  assumes field-inverse-zero: inverse 0 = 0
begin

subclass division-ring
proof
  fix a :: ’a
  assume a ≠ 0
  thus inverse a * a = 1 by (rule field-inverse)
  thus a * inverse a = 1 by (simp only: mult.commute)
next
  fix a b :: ’a
  show a / b = a * inverse b by (rule field-divide-inverse)
next
  show inverse 0 = 0
    by (fact field-inverse-zero)
qed
subclass idom-divide
proof
  fix b a
  assume b ≠ 0
  then show a * b / b = a
      by (simp add: divide-inverse ac-simps)
next
  fix a
  show a / 0 = 0
      by (simp add: divide-inverse)
qed

There is no slick version using division by zero.

lemma inverse-add:
  a ≠ 0 ⇒ b ≠ 0 ⇒ inverse a + inverse b = (a + b) * inverse a * inverse b
  by (simp add: division-ring-inverse-add ac-simps)

lemma nonzero-mult-divide-mult-cancel-left [simp]:
  assumes [simp]: c ≠ 0
  shows (c * a) / (c * b) = a / b
proof (cases b = 0)
  case True then show ?thesis by simp
next
  case False
  then have (c*a)/(c*b) = c * a * (inverse b * inverse c)
      by (simp add: divide-inverse nonzero-inverse-mult-distrib)
  also have ... = a * inverse b * (inverse c * c)
      by (simp only: ac-simps)
  also have ... = a * inverse b by simp
  finally show ?thesis by (simp add: divide-inverse)
qed

lemma nonzero-mult-divide-mult-cancel-right [simp]:
  c ≠ 0 ⇒ (a * c) / (b * c) = a / b
using nonzero-mult-divide-mult-cancel-left [of c a b] by (simp add: ac-simps)

lemma times-divide-eq-left [simp]: (b / c) * a = (b * a) / c
by (simp add: divide-inverse ac-simps)

lemma divide-inverse-commute: a / b = inverse b * a
by (simp add: divide-inverse mult.commute)

lemma add-frac-eq:
  assumes y ≠ 0 and z ≠ 0
  shows x / y + w / z = (x * z + w * y) / (y * z)
proof
  have x / y + w / z = (x * z) / (y * z) + (y * w) / (y * z)
      using assms by simp
also have \[ (x * z + y * w) / (y * z) \]
  by (simp only: add-divide-distrib)
finally show \(?thesis\)
  by (simp only: mult.commute)
qed

Special Cancellation Simprules for Division

lemma nonzero-divide-mult-cancel-right [simp]:
  \[ b \neq 0 \implies b / (a * b) = 1 / a \]
using nonzero-mult-divide-mult-cancel-right [of b 1 a] by simp

lemma nonzero-divide-mult-cancel-left [simp]:
  \[ a \neq 0 \implies a / (a * b) = 1 / b \]
using nonzero-mult-divide-mult-cancel-left [of a 1 b] by simp

lemma nonzero-mult-divide-mult-cancel-left2 [simp]:
  \[ c \neq 0 \implies (c * a) / (b * c) = a / b \]
using nonzero-mult-divide-mult-cancel-left [of c a b] by (simp add: ac-simps)

lemma nonzero-mult-divide-mult-cancel-right2 [simp]:
  \[ c \neq 0 \implies (a * c) / (c * b) = a / b \]
using nonzero-mult-divide-mult-cancel-right [of b c a] by (simp add: ac-simps)

lemma diff-frac-eq:
  \[ y \neq 0 \implies z \neq 0 \implies x / y - w / z = (x * z - w * y) / (y * z) \]
by (simp add: field-simps)

lemma frac-eq-eq:
  \[ y \neq 0 \implies z \neq 0 \implies (x / y = w / z) = (x * z = w * y) \]
by (simp add: field-simps)

lemma divide-minus1 [simp]: \[ x / -1 = -x \]
using nonzero-minus-divide-right [of 1 x] by simp

This version builds in division by zero while also re-orienting the right-hand side.

lemma inverse-mult-distrib [simp]:
  inverse (a * b) = inverse a * inverse b
proof cases
  assume a \neq 0 \land b \neq 0
  thus \(?thesis\) by (simp add: nonzero-inverse-mult-distrib ac-simps)
next
  assume \( \neg (a \neq 0 \land b \neq 0) \)
  thus \(?thesis\) by force
qed

lemma inverse-divide [simp]:
  inverse (a / b) = b / a
by (simp add: divide-inverse mult.commute)
Calculations with fractions

There is a whole bunch of simp-rules just for class *field* but none for class *field* and *nonzero-divides* because the latter are covered by a simproc.

**lemmas** `mult-divide-mult-cancel-left = nonzero-mult-divide-mult-cancel-left`

**lemmas** `mult-divide-mult-cancel-right = nonzero-mult-divide-mult-cancel-right`

**lemma** `divide-divide-eq-right [simp]:`
\[ a / (b / c) = (a * c) / b \]
by (simp add: divide-inverse ac-simps)

**lemma** `divide-divide-eq-left [simp]:`
\[ (a / b) / c = a / (b * c) \]
by (simp add: divide-inverse mult.assoc)

**lemma** `divide-divide-times-eq:`
\[ (x / y) / (z / w) = (x * w) / (y * z) \]
by simp

**Special Cancellation Simprules for Division**

**lemma** `mult-divide-mult-cancel-left-if [simp]:`
\[ (c * a) / (c * b) = (if c = 0 then 0 else a / b) \]
by simp

**Division and Unary Minus**

**lemma** `minus-divide-right:`
\[- (a / b) = a / - b \]
by (simp add: divide-inverse)

**lemma** `divide-minus-right [simp]:`
\[ a / - b = - (a / b) \]
by (simp add: divide-inverse)

**lemma** `minus-divide-divide:`
\[- (a) / (- b) = a / b \]
by (cases b=0) (simp-all add: nonzero-minus-divide-divide)

**lemma** `inverse-eq-1-iff [simp]:`
inverse \( x = 1 \) \( \iff \) \( x = 1 \)
by (insert inverse-eq-1-iff-eq [of \( x = 1 \)], simp)

**lemma** `divide-eq-0-iff [simp]:`
\[ a / b = 0 \iff a = 0 \lor b = 0 \]
by (simp add: divide-inverse)

**lemma** `divide-cancel-right [simp]:`
\[ a / c = b / c \iff c = 0 \lor a = b \]
by (cases c=0) (simp-all add: divide-inverse)
lemma divide-cancel-left [simp]:
  \( c / a = c / b \iff c = 0 \lor a = b \)
by (cases c=0) (simp-all add: divide-inverse)

lemma divide-eq-1-iff [simp]:
  \( a / b = 1 \iff b \neq 0 \land a = b \)
by (cases b=0) (simp-all add: right-inverse-eq)

lemma one-eq-divide-iff [simp]:
  \( 1 = a / b \iff b \neq 0 \land a = b \)
by (simp add: eq-commute [of 1])

lemma divide-eq-minus-1-iff:
  \( (a / b) = -1 \iff b \neq 0 \land a = -b \)
using divide-eq-1-iff by fastforce

lemma times-divide-times-eq:
  \( x / y \cdot z / w = (x \cdot z) / (y \cdot w) \)
by simp

lemma add-frac-num:
  \( y \neq 0 \implies x / y + z = (x + z \cdot y) / y \)
by (simp add: add-divide-distrib)

lemma add-num-frac:
  \( y \neq 0 \implies z + x / y = (x + z \cdot y) / y \)
by (simp add: add-divide-distrib add.commute)

lemma dvd-field-iff:
  \( a \ dvd b \iff (a = 0 \implies b = 0) \)
proof (cases a = 0)
  case False
  then have \( b = a \cdot (b / a) \)
  by (simp add: field-simps)
  then have \( a \ dvd b \) ..
  with False show \( \?)thesis
  by simp
qed simp

lemma inj-divide-right [simp]:
  \( inj (\lambda b. b / a) \iff a \neq 0 \)
proof
  have \( (\lambda b. b / a) = (\ast) \cdot (inverse a) \)
  by (simp add: field-simps fun-eq-iff)
  then have \( inj (\lambda y. y / a) \iff inj ((\ast) \cdot (inverse a)) \)
  by simp
  also have \( \ldots \iff inverse a \neq 0 \)
  by simp
also have \( \leftrightarrow a \neq 0 \)
by simp
finally show ?thesis
by simp
qed
end

class field-char-0 = field + ring-char-0

16.3 Ordered fields

class field-abs-sgn = field + idom-abs-sgn
begin

lemma sgn-inverse [simp]:
  sgn (inverse a) = inverse (sgn a)
proof (cases a = 0)
case True then show ?thesis by simp
next
case False
then have a * inverse a = 1
by simp
then have sgn (a * inverse a) = sgn 1
by simp
then have sgn a * sgn (inverse a) = 1
by (simp add: sgn-mult)
then have inverse (sgn a) * (sgn a * sgn (inverse a)) = inverse (sgn a) * 1
by simp
then have (inverse (sgn a) * sgn a) * sgn (inverse a) = inverse (sgn a)
by (simp add: ac-simps)
with False show ?thesis
by (simp add: sgn-eq-0-iff)
qed

lemma abs-inverse [simp]:
  \(|inverse a| = inverse |a|
proof
  from sgn-mult-abs [of inverse a] sgn-mult-abs [of a]
  have inverse (sgn a) * \(|inverse a| = inverse (sgn a * |a|
  by simp
  then show ?thesis by (auto simp add: sgn-eq-0-iff)
qed

lemma sgn-divide [simp]:
  sgn (a / b) = sgn a / sgn b
unfolding divide-inverse sgn-mult by simp

lemma abs-divide [simp]:
\[ |a / b| = |a| / |b| \]

unfolding divide-inverse abs-mult by simp

end

class linordered-field = field + linordered-idom
begin

lemma positive-imp-inverse-positive:
  assumes a-gt-0: \( 0 < a \)
  shows \( 0 < inverse a \)
proof (rule classical)
  assume \( \neg b \leq a \)
  hence \( a < b \) by (simp add: linorder-not-le)
  hence bpos: \( 0 < b \) by (blast intro: apos less-trans)
  hence \( a * inverse a \leq a * inverse b \)
    by (simp add: apos invle less-imp-le mult-left-mono)
  hence \( (a * inverse a) * b \leq (a * inverse b) * b \)
    by (simp add: bpos less-imp-le mult-right-mono)
  thus \( b \leq a \) by (simp add: mult.assoc apos bpos less-imp-not-eq2)
qed

lemma inverse-le-imp-le:
  assumes invle: \( inverse a \leq inverse b \) and apos: \( 0 < a \)
  shows \( b \leq a \)
proof (rule classical)
  assume \( \neg b \leq a \)
  hence \( a < b \) by (simp add: linorder-not-le)
  hence bpos: \( 0 < b \) by (blast intro: apos less-trans)
  hence \( a * inverse a \leq a * inverse b \)
    by (simp add: apos invle less-imp-le mult-left-mono)
  hence \( (a * inverse a) * b \leq (a * inverse b) * b \)
    by (simp add: bpos less-imp-le mult-right-mono)
  thus \( b \leq a \) by (simp add: mult.assoc apos bpos less-imp-not-eq2)
qed

lemma inverse-positive-imp-positive:
  assumes inv-gt-0: \( 0 < inverse a \) and nz: \( a \neq 0 \)
  shows \( 0 < a \)
proof (rule classical)
  assume \( \neg 0 < inverse a \)
  using inv-gt-0 by (rule positive-imp-inverse-positive)
  thus \( 0 < a \)
    using nz by (simp add: nonzero-inverse-inverse-eq)
qed

lemma inverse-negative-imp-negative:
  assumes inv-less-0: \( inverse a < 0 \) and nz: \( a \neq 0 \)
shows $a < 0$

proof –
  have $\text{inverse (inverse a)} < 0$
    using $\text{inv-less-0}$ by (rule $\text{negative-imp-inverse-negative}$)
  thus $a < 0$ using $\text{nz}$ by (simp add: $\text{nonzero-inverse-inverse-eq}$)
  qed

lemma $\text{linordered-field-no-lb}$:
\[
\forall x. \exists y. y < x
\]
proof
  fix $x :: a$
  have $m1: - (1 :: a) < 0$ by simp
  from $\text{add-strict-right-mono}[\text{OF m1, where c=x}]$
  have $(- 1) + x < x$ by simp
  thus $\exists y. y < x$ by blast
  qed

lemma $\text{linordered-field-no-ub}$:
\[
\forall x. \exists y. y > x
\]
proof
  fix $x :: a$
  have $m1: (1 :: a) > 0$ by simp
  from $\text{add-strict-right-mono}[\text{OF m1, where c=x}]$
  have $1 + x > x$ by simp
  thus $\exists y. y > x$ by blast
  qed

lemma $\text{less-imp-inverse-less}$:
assumes less: $a < b$ and apos: $0 < a$
shows $\text{inverse b} < \text{inverse a}$
proof (rule ccontr)
  assume $\neg \text{inverse b} < \text{inverse a}$
  hence $\text{inverse a} \leq \text{inverse b}$ by simp
  hence $\neg (a < b)$
    by (simp add: $\text{not-less inverse-le-imp-le}$ [OF $\text{apos}$])
  thus False by (rule notE [OF $\text{less}$])
  qed

lemma $\text{inverse-less-imp-less}$:
$\text{inverse a} < \text{inverse b} \Longrightarrow 0 < a \Longrightarrow b < a$
apply (simp add: $\text{less-le}$ [of $\text{inverse a}$] $\text{less-le}$ [of $\text{b}$])
apply (force dest!: $\text{inverse-le-imp-le}$ $\text{nonzero-inverse-eq-imp-eq}$)
done

Both premises are essential. Consider -1 and 1.

lemma $\text{inverse-less-iff-less}$ [simp]:
\[
0 < a \Longrightarrow 0 < b \Longrightarrow \text{inverse a} < \text{inverse b} \iff b < a
\]
by (blast intro: $\text{less-imp-inverse-less}$ dest: $\text{inverse-less-imp-less}$)
lemma \text{le-imp-inverse-le}:

\quad a \leq b \implies 0 < a \implies \text{inverse } b \leq \text{inverse } a

by (force simp add: \text{le-less less-imp-inverse-less})

lemma \text{inverse-le-iff-le} [simp]:

\quad 0 < a \implies 0 < b \implies \text{inverse } a \leq \text{inverse } b \iff b \leq a

by (blast intro: \text{le-imp-inverse-le dest: inverse-le-imp-le})

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

lemma \text{inverse-le-imp-le-neg}:

\quad \text{inverse } a \leq \text{inverse } b \implies b < 0 \implies b \leq a

apply (rule classical)
apply (subgoal-tac a < 0)
pref 2 apply force
apply (insert inverse-le-imp-le [of \(-b\) \(-a\)])
apply (simp add: nonzero-inverse-minus-eq)
done

lemma \text{less-imp-inverse-less-neg}:

\quad a < b \implies b < 0 \implies \text{inverse } b < \text{inverse } a

apply (subgoal-tac a < 0)
pref 2 apply (blast intro: less-trans)
apply (insert less-imp-inverse-less [of \(-b\) \(-a\)])
apply (simp add: nonzero-inverse-minus-eq)
done

lemma \text{inverse-less-imp-less-neg}:

\quad \text{inverse } a < \text{inverse } b \implies b < 0 \implies b < a

apply (rule classical)
apply (subgoal-tac a < 0)
pref 2
apply force
apply (insert inverse-less-imp-less [of \(-b\) \(-a\)])
apply (simp add: nonzero-inverse-minus-eq)
done

lemma \text{inverse-less-iff-less-neg} [simp]:

\quad a < 0 \implies b < 0 \implies \text{inverse } a < \text{inverse } b \iff b < a

apply (insert inverse-less-iff-less [of \(-b\) \(-a\)])
apply (simp del: inverse-less-iff-less
add: nonzero-inverse-minus-eq)
done

lemma \text{le-imp-inverse-le-neg}:

\quad a \leq b \implies b < 0 \implies \text{inverse } b \leq \text{inverse } a

by (force simp add: \text{le-less less-imp-inverse-less-neg})

lemma \text{inverse-le-iff-le-neg} [simp]:
a < 0 \quad \Rightarrow \quad b < 0 \quad \Rightarrow \quad \text{inverse} \ a \leq \text{inverse} \ b \quad \Leftrightarrow \quad b \leq a

by (\text{blast intro: le-imp-inverse-le-neg dest: inverse-le-imp-le-neg})

\text{lemma one-less-inverse:}
\quad \theta < a \quad \Rightarrow \quad a < 1 \quad \Rightarrow \quad 1 < \text{inverse} \ a

\text{using le-imp-inverse-less [of a 1, unfolded inverse-1].}

\text{lemma one-le-inverse:}
\quad \theta < a \quad \Rightarrow \quad a \leq 1 \quad \Rightarrow \quad 1 \leq \text{inverse} \ a

\text{using le-imp-inverse-le [of a 1, unfolded inverse-1].}

\text{lemma pos-le-divide-eq [field-simps]:}
\quad \text{assumes} \quad 0 < c
\quad \text{shows} \quad a \leq \frac{b}{c} \quad \Leftrightarrow \quad a \cdot c \leq b

\text{proof –}
\quad \text{from} \ \text{assms} \ \text{have} \quad a \leq \frac{b}{c} \quad \Leftrightarrow \quad a \cdot c \leq \frac{b}{c} \cdot c
\quad \text{using mult-le-cancel-right [of a c b * inverse c] by (auto simp add: field-simps)}
\quad \text{also have} \quad \ldots \quad \Leftrightarrow \quad a \cdot c \leq b
\quad \text{by (simp add: less-imp-not-eq2 [OF assms] divide-inverse mult.assoc)}
\quad \text{finally show} \quad \text{thesis}.

\text{qed}

\text{lemma pos-less-divide-eq [field-simps]:}
\quad \text{assumes} \quad 0 < c
\quad \text{shows} \quad a < \frac{b}{c} \quad \Leftrightarrow \quad a \cdot c < b

\text{proof –}
\quad \text{from} \ \text{assms} \ \text{have} \quad a < \frac{b}{c} \quad \Leftrightarrow \quad a \cdot c < \frac{b}{c} \cdot c
\quad \text{using mult-less-cancel-right [of a c b] by auto}
\quad \text{also have} \quad \ldots \quad = \quad (a \cdot c < b)
\quad \text{by (simp add: less-imp-not-eq2 [OF assms] divide-inverse mult.assoc)}
\quad \text{finally show} \quad \text{thesis}.

\text{qed}

\text{lemma neg-less-divide-eq [field-simps]:}
\quad \text{assumes} \quad c < 0
\quad \text{shows} \quad a < \frac{b}{c} \quad \Leftrightarrow \quad b < a \cdot c

\text{proof –}
\quad \text{from} \ \text{assms} \ \text{have} \quad a < \frac{b}{c} \quad \Leftrightarrow \quad (b / c) \cdot c < a \cdot c
\quad \text{using mult-less-cancel-right [of b / c c a] by auto}
\quad \text{also have} \quad \ldots \quad \Leftrightarrow \quad b < a \cdot c
\quad \text{by (simp add: less-imp-not-eq [OF assms] divide-inverse mult.assoc)}
\quad \text{finally show} \quad \text{thesis}.

\text{qed}

\text{lemma neg-le-divide-eq [field-simps]:}
\quad \text{assumes} \quad c < 0
\quad \text{shows} \quad a \leq \frac{b}{c} \quad \Leftrightarrow \quad (b / c) \cdot c \leq a \cdot c

\text{proof –}
\quad \text{from} \ \text{assms} \ \text{have} \quad a \leq \frac{b}{c} \quad \Leftrightarrow \quad (b / c) \cdot c \leq a \cdot c
using mult-le-cancel-right [of b * inverse c c a] by (auto simp add: field-simps)
also have ... \( \rightarrow \) \( b \leq a \ast c \)
  by (simp add: less-imp-not-eq [OF assms] divide-inverse mult.assoc)
finally show \(?thesis\).
qed

lemma pos-divide-le-eq [field-simps]:
  assumes \( 0 < c \)
  shows \( b / c \leq a \leftrightarrow b \leq a \ast c \)
proof –
  from assms have \( b / c \leq a \leftrightarrow (b / c) \ast c \leq a \ast c \)
  using mult-le-cancel-right [of b / c c a] by auto
also have ... \( \leftrightarrow \) \( b \leq a \ast c \)
  by (simp add: less-imp-not-eq2 [OF assms] divide-inverse mult.assoc)
finally show \(?thesis\).
qed

lemma pos-divide-less-eq [field-simps]:
  assumes \( 0 < c \)
  shows \( b / c < a \leftrightarrow b < a \ast c \)
proof –
  from assms have \( b / c < a \leftrightarrow (b / c) \ast c < a \ast c \)
  using mult-less-cancel-right [of b / c c a] by auto
also have ... \( \leftrightarrow \) \( b < a \ast c \)
  by (simp add: less-imp-not-eq2 [OF assms] divide-inverse mult.assoc)
finally show \(?thesis\).
qed

lemma neg-divide-le-eq [field-simps]:
  assumes \( c < 0 \)
  shows \( b / c \leq a \leftrightarrow a \ast c \leq b \)
proof –
  from assms have \( b / c \leq a \leftrightarrow a \ast c \leq (b / c) \ast c \)
  using mult-le-cancel-right [of a c b / c] by auto
also have ... \( \leftrightarrow \) \( a \ast c \leq b \)
  by (simp add: less-imp-not-eq [OF assms] divide-inverse mult.assoc)
finally show \(?thesis\).
qed

lemma neg-divide-less-eq [field-simps]:
  assumes \( c < 0 \)
  shows \( b / c < a \leftrightarrow a \ast c < b \)
proof –
  from assms have \( b / c < a \leftrightarrow a \ast c < b / c \ast c \)
  using mult-less-cancel-right [of a c b / c] by auto
also have ... \( \leftrightarrow \) \( a \ast c < b \)
  by (simp add: less-imp-not-eq [OF assms] divide-inverse mult.assoc)
finally show \(?thesis\).
qed
The following field-simps rules are necessary, as minus is always moved atop of division but we want to get rid of division.

lemma pos-le-minus-divide-eq [field-simps]: \(0 < c \implies a \leq - (b / c) \iff a * c \leq - b\)
unfolding minus-divide-left by (rule pos-le-divide-eq)

lemma neg-le-minus-divide-eq [field-simps]: \(c < 0 \implies a \leq - (b / c) \iff - b \leq a * c\)
unfolding minus-divide-left by (rule neg-le-divide-eq)

lemma pos-less-minus-divide-eq [field-simps]: \(0 < c \implies a < - (b / c) \iff a * c < - b\)
unfolding minus-divide-left by (rule pos-less-divide-eq)

lemma neg-less-minus-divide-eq [field-simps]: \(c < 0 \implies a < - (b / c) \iff - b < a * c\)
unfolding minus-divide-left by (rule neg-less-divide-eq)

lemma pos-minus-divide-less-eq [field-simps]: \(0 < c \implies - (b / c) < a \iff - b < a * c\)
unfolding minus-divide-left by (rule pos-divide-less-eq)

lemma neg-minus-divide-less-eq [field-simps]: \(c < 0 \implies - (b / c) < a \iff a * c < - b\)
unfolding minus-divide-left by (rule neg-divide-less-eq)

lemma pos-minus-divide-le-eq [field-simps]: \(0 < c \implies - (b / c) \leq a \iff - b \leq a * c\)
unfolding minus-divide-left by (rule pos-divide-le-eq)

lemma neg-minus-divide-le-eq [field-simps]: \(c < 0 \implies - (b / c) \leq a \iff a * c \leq - b\)
unfolding minus-divide-left by (rule neg-divide-le-eq)

lemma frac-less-eq:
\[ y \neq 0 \implies z \neq 0 \implies x / y < w / z \iff (x * z - w * y) / (y * z) < 0 \]
by (subst less-iff-diff-less-0) (simp add: diff-fracl-eq)

lemma frac-le-eq:
\[ y \neq 0 \implies z \neq 0 \implies x / y \leq w / z \iff (x * z - w * y) / (y * z) \leq 0 \]
by (subst le-iff-diff-le-0) (simp add: diff-fracl-eq)

lemma divide-pos-pos[simp]:
\[ 0 < x \implies 0 < y \implies 0 < x / y \]
by (simp add: field-simps)

lemma divide-nonneg-pos:
\[ 0 \leq x \implies 0 < y \implies 0 \leq x / y \]
by (simp add: field-simps)
lemma divide-neg-pos:
\[ x < 0 \implies 0 < y \implies x / y < 0 \]
by(simp add:field-simps)

lemma divide-nonpos-pos:
\[ x \leq 0 \implies 0 < y \implies x / y \leq 0 \]
by(simp add:field-simps)

lemma divide-pos-neg:
\[ 0 < x \implies y < 0 \implies x / y < 0 \]
by(simp add:field-simps)

lemma divide-nonneg-neg:
\[ 0 \leq x \implies y < 0 \implies x / y \leq 0 \]
by(simp add:field-simps)

lemma divide-neg-neg:
\[ x < 0 \implies y < 0 \implies 0 < x / y \]
by(simp add:field-simps)

lemma divide-nonpos-neg:
\[ x \leq 0 \implies y < 0 \implies 0 \leq x / y \]
by(simp add:field-simps)

lemma divide-strict-right-mono:
\[[| a < b; 0 < c |] \implies a / c < b / c\]
by (simp add: less-imp-not-eq2 divide-inverse mult-strict-right-mono
positive-imp-inverse-positive)

lemma divide-strict-right-mono-neg:
\[[| b < a; c < 0 |] \implies a / c < b / c\]
apply (drule divide-strict-right-mono [of - - c], simp)
apply (simp add: less-imp-not-eq nonzero-minus-divide-right [symmetric])
done

The last premise ensures that a and b have the same sign

lemma divide-strict-left-mono:
\[[| b < a; 0 < c; 0 < a*b |] \implies c / a < c / b\]
by (auto simp: field-simps zero-less-mult-iff mult-strict-right-mono)

lemma divide-left-mono:
\[[| b \leq a; 0 \leq c; 0 < a*b |] \implies c / a \leq c / b\]
by (auto simp: field-simps zero-less-mult-iff mult-right-mono)

lemma divide-strict-left-mono-neg:
\[[| a < b; c < 0; 0 < a*b |] \implies c / a < c / b\]
by (auto simp: field-simps zero-less-mult-iff mult-strict-right-mono-neg)
lemma mult-imp-div-pos-le: \( 0 < y \Rightarrow x \leq z \times y \Rightarrow x / y \leq z \)
by (subst pos-divide-le-eq, assumption+)

lemma mult-imp-le-div-pos: \( 0 < y \Rightarrow z \times y \leq x \Rightarrow z \leq x / y \)
by (simp add: field-simps)

lemma mult-imp-div-pos-less: \( 0 < y \Rightarrow x \times y \leq z \Rightarrow x / y < z \)
by (simp add: field-simps)

lemma frac-le: \( 0 \leq x \Rightarrow x \leq y \Rightarrow 0 < w \Rightarrow w \leq z \Rightarrow x / z \leq y / w \)
apply (rule mult-imp-div-pos-le)
apply simp
apply (subst times-divide-eq-left)
apply (rule mult-imp-le-div-pos, assumption)
apply (rule mult-mono)
apply simp-all
done

lemma frac-less: \( 0 \leq x \Rightarrow x < y \Rightarrow 0 < w \Rightarrow w < z \Rightarrow x / z < y / w \)
apply (rule mult-imp-div-pos-less)
apply simp
apply (subst times-divide-eq-left)
apply (rule mult-imp-less-div-pos, assumption)
apply (erule mult-less-le-imp-less)
apply simp-all
done

lemma frac-less2: \( 0 < x \Rightarrow x < y \Rightarrow 0 < w \Rightarrow w < z \Rightarrow x / z < y / w \)
apply (rule mult-imp-div-pos-less)
apply simp-all
apply (rule mult-imp-less-div-pos, assumption)
apply (erule mult-le-less-imp-less)
apply simp-all
done

lemma less-half-sum: \( a < b \Rightarrow a < (a + b) / (1 + 1) \)
by (simp add: field-simps zero-less-two)
lemma gtl-half-sum: \( a < b \implies \frac{(a+b)(1+1)}{1} < b \)
by \((simp add: field-simps zero-less-two)\)

subclass unbounded-dense-linorder
proof
fix \( x, y :: 'a \)
from less-add-one show \( \exists y. x < y \) ..
from less-add-one have \( x + (-1) < (x + 1) + (-1) \) by \((rule add-strict-right-mono)\)
then have \( x - 1 < x + 1 - 1 \) by simp
then have \( x - 1 < x \) by \((simp add: algebra-simps)\)
then show \( \exists y. y < x \) ..
show \( x < y \implies \exists z > x. z < y \) by \((blast intro: gtl-half-sum gtl-half-sum)\)
qued

subclass field-abs-sgn ..

lemma inverse-sgn [simp]:
inverse \((sgn a)\) = sgn a
by \((cases a 0 rule: linorder-cases) simp-all\)

lemma divide-sgn [simp]:
\( a / sgn b \) = \( a \times sgn b \)
by \((cases b 0 rule: linorder-cases) simp-all\)

lemma nonzero-abs-inverse:
\( a \neq 0 \implies |inverse a| = inverse |a| \)
by \((rule abs-inverse)\)

lemma nonzero-abs-divide:
\( b \neq 0 \implies |a / b| = |a| / |b| \)
by \((rule abs-divide)\)

lemma field-le-epsilon:
assumes e: \( \forall e. 0 < e \implies x \leq y + e \)
shows \( x \leq y \)
proof \((rule dense-le)\)
fix \( t \) assume \( t < x \)
hence \( 0 < x - t \) by \((simp add: less-diff-eq)\)
from e \((OF this)\) have \( x + \theta \leq x + (y - t) \) by \((simp add: algebra-simps)\)
then have \( 0 \leq y - t \) by \((simp only: add-le-cancel-left)\)
then show \( t \leq y \) by \((simp add: algebra-simps)\)
qued

lemma inverse-positive-iff-positive [simp]:
\( (0 < inverse a) = (0 < a) \)
apply \((cases a = 0, simp)\)
apply \((blast intro: inverse-positive-imp-positive positive-imp-inverse-positive)\)
done
lemma inverse-negative-iff-negative [simp]:
  \( (\text{inverse } a < 0) \equiv (a < 0) \)
apply (cases a = 0, simp)
apply (blast intro: inverse-negative-imp-negative negative-imp-inverse-negative)
done

lemma inverse-nonnegative-iff-nonnegative [simp]:
  \( 0 \leq \text{inverse } a \leftrightarrow 0 \leq a \)
by (simp add: not-less [symmetric])

lemma inverse-nonpositive-iff-nonpositive [simp]:
  \( \text{inverse } a \leq 0 \leftrightarrow a \leq 0 \)
by (simp add: not-less [symmetric])

lemma one-less-inverse-iff: \( 1 < \text{inverse } x \leftrightarrow 0 < x \land x < 1 \)
using less-trans[of 1 x 0 for x]
by (cases x 0 rule: linorder-cases) (auto simp add: field-simps)

lemma one-le-inverse-iff: \( 1 \leq \text{inverse } x \leftrightarrow 0 < x \land x \leq 1 \)
proof (cases x 0)
case True then show ?thesis by simp
next
case False then have \( \text{inverse } x \neq 1 \) by simp
then have \( 1 \leq \text{inverse } x \leftrightarrow 1 < \text{inverse } x \) by (simp add: le-less)
with False show ?thesis by (auto simp add: one-less-inverse-iff)
qed

lemma inverse-less-1-iff: \( \text{inverse } x < 1 \leftrightarrow x \leq 0 \lor 1 < x \)
by (simp add: not-le [symmetric] one-le-inverse-iff)

lemma inverse-le-1-iff: \( \text{inverse } x \leq 1 \leftrightarrow x \leq 0 \lor 1 \leq x \)
by (simp add: not-less [symmetric] one-less-inverse-iff)

lemma [field-split-simps, divide-simps]:
  shows le-divide-eq: \( a \leq b / c \leftrightarrow (if 0 < c then a * c \leq b else if c < 0 then b \leq a * c else a \leq 0) \)
  and divide-le-eq: \( b / c \leq a \leftrightarrow (if 0 < c then b \leq a * c else if c < 0 then a * c \leq b else 0 \leq a) \)
  and less-divide-eq: \( a < b / c \leftrightarrow (if 0 < c then a * c < b else if c < 0 then b < a * c else a < 0) \)
  and divide-less-eq: \( b / c < a \leftrightarrow (if 0 < c then b < a * c else if c < 0 then a * c < b else 0 < a) \)
  and le-minus-divide-eq: \( a \leq - (b / c) \leftrightarrow (if 0 < c then a * c \leq - b else if c < 0 then - b \leq a * c else a \leq 0) \)
  and minus-divide-le-eq: \( - (b / c) \leq a \leftrightarrow (if 0 < c then - b \leq a * c else if c < 0 then a * c \leq - b else 0 \leq a) \)
  and less-minus-divide-eq: \( a < - (b / c) \leftrightarrow (if 0 < c then a * c < - b else if c < 0 then - b < a * c else a < 0) \)
and minus-divide-less-eq: \(-\frac{b}{c} < a \iff (\text{if } 0 < c \text{ then } -b < a \ast c \text{ else if } c < 0 \text{ then } a \ast c < -b \text{ else } 0 < a)\)
by (auto simp: field-simps not-less dest: antisym)

Division and Signs

lemma shows zero-less-divide-iff: \(0 < a / b \iff 0 < a \land 0 < b \lor a < 0 \land b < 0\)
and divide-less-0-iff: \(a / b < 0 \iff 0 < a \land b < 0 \lor a < 0 \land 0 < b\)
and zero-le-divide-iff: \(0 \leq a / b \iff 0 \leq a \land 0 \leq b \lor a < 0 \land b < 0\)
and divide-le-0-iff: \(a / b \leq 0 \iff 0 \leq a \land b \leq 0 \lor a < 0 \land 0 < b\)
by (auto simp add: field-split-simps)

Division and the Number One

Simplify expressions equated with 1

lemma zero-eq-1-divide-iff [simp]: \(0 = 1 / a \iff a = 0\)
by (cases a = 0) (auto simp: field-simps)

lemma one-divide-eq-0-iff [simp]: \(1 / a = 0 \iff a = 0\)
using zero-eq-1-divide-iff [of a] by simp

Simplify expressions such as \(0 < 1/x\) to \(0 < x\)

lemma zero-le-divide-1-iff [simp]:
\(0 \leq 1 / a \iff 0 \leq a\)
by (simp add: zero-le-divide-iff)

lemma zero-less-divide-1-iff [simp]:
\(0 < 1 / a \iff 0 < a\)
by (simp add: zero-less-divide-iff)

lemma divide-le-0-1-iff [simp]:
\(1 / a \leq 0 \iff a \leq 0\)
by (simp add: divide-le-0-iff)

lemma divide-less-0-1-iff [simp]:
\(1 / a < 0 \iff a < 0\)
by (simp add: divide-less-0-iff)

lemma divide-right-mono:
\([|a \leq b; 0 \leq c|] \Longrightarrow a/c \leq b/c\)
by (force simp add: divide-strict-right-mono le-less)

lemma divide-right-mono-neg: \(a <= b\)
\(\Longrightarrow c <= 0 \Longrightarrow b / c <= a / c\)
apply (drule divide-right-mono [of - - - c])
apply auto
apply
apply auto done

lemma divide-left-mono-neg: \(a <= b\)
apply (drule divide-left-mono [of - - c])
apply (auto simp add: mult.commute)
done

lemma inverse-le-iff: inverse a ≤ inverse b ⟷ (0 < a * b ⟷ b ≤ a) ∧ (a * b ≤ 0 ⟷ a ≤ b)
  by (cases a 0 b 0 rule: linorder-cases [case-product linorder-cases])
    (auto simp add: field_simps zero_less_mult_iff mult_le_0_iff)

lemma inverse-less-iff: inverse a < inverse b ⟷ (0 < a * b ⟷ b < a) ∧ (a * b ≤ 0 ⟷ a < b)
  by (subst less-le) (auto simp add: inverse-le-iff)

lemma divide-le-cancel: a / c ≤ b / c ⟷ (0 < c ⟷ a ≤ b) ∧ (c < 0 ⟷ b ≤ a)
  by (simp add: divide-inverse mult-le-cancel-right)

lemma divide-less-cancel: a / c < b / c ⟷ (0 < c ⟷ a < b) ∧ (c < 0 ⟷ b ≤ a) ∧ c ≠ 0
  by (auto simp add: divide-inverse mult-less-cancel-right)

Simplify quotients that are compared with the value 1.

lemma le-divide-eq-1:
  (1 ≤ b / a) = ((0 < a ∧ a ≤ b) ∨ (a < 0 ∧ b ≤ a))
  by (auto simp add: le-divide-eq)

lemma divide-le-eq-1:
  (b / a ≤ 1) = ((0 < a ∧ b ≤ a) ∨ (a < 0 ∧ a ≤ b) ∨ a=0)
  by (auto simp add: divide-le-eq)

lemma less-divide-eq-1:
  (1 < b / a) = ((0 < a ∧ a < b) ∨ (a < 0 ∧ b < a))
  by (auto simp add: less-divide-eq)

lemma divide-less-eq-1:
  (b / a < 1) = ((0 < a ∧ b < a) ∨ (a < 0 ∧ a < b) ∨ a=0)
  by (auto simp add: divide-less-eq)

lemma divide-nonneg-nonneg [simp]:
  0 ≤ x ⟹ 0 ≤ y ⟹ 0 ≤ x / y
  by (auto simp add: field-split-simps)

lemma divide-nonpos-nonpos:
  x ≤ 0 ⟹ y ≤ 0 ⟹ 0 ≤ x / y
  by (auto simp add: field-split-simps)

lemma divide-nonneg-nonpos:
  0 ≤ x ⟹ y ≤ 0 ⟹ x / y ≤ 0
by (auto simp add: field-split-simps)

**lemma** divide-nonpos-nonneg:
\[ x \leq 0 \implies 0 \leq y \implies x \div y \leq 0 \]
by (auto simp add: field-split-simps)

Conditional Simplification Rules: No Case Splits

**lemma** le-divide-eq-1-pos [simp]:
\[ 0 < a \implies (1 \leq b/a) = (a \leq b) \]
by (auto simp add: le-divide-eq)

**lemma** le-divide-eq-1-neg [simp]:
\[ a < 0 \implies (1 \leq b/a) = (b \leq a) \]
by (auto simp add: le-divide-eq)

**lemma** divide-le-eq-1-pos [simp]:
\[ 0 < a \implies (b/a \leq 1) = (b \leq a) \]
by (auto simp add: divide-le-eq)

**lemma** divide-le-eq-1-neg [simp]:
\[ a < 0 \implies (b/a \leq 1) = (a \leq b) \]
by (auto simp add: divide-le-eq)

**lemma** less-divide-eq-1-pos [simp]:
\[ 0 < a \implies (1 < b/a) = (a < b) \]
by (auto simp add: less-divide-eq)

**lemma** less-divide-eq-1-neg [simp]:
\[ a < 0 \implies (1 < b/a) = (b < a) \]
by (auto simp add: less-divide-eq)

**lemma** divide-less-eq-1-pos [simp]:
\[ 0 < a \implies (b/a < 1) = (b < a) \]
by (auto simp add: divide-less-eq)

**lemma** divide-less-eq-1-neg [simp]:
\[ a < 0 \implies b/a < 1 \iff a < b \]
by (auto simp add: divide-less-eq)

**lemma** eq-divide-eq-1 [simp]:
\[ (1 = b/a) = ((a \neq 0 \land a = b)) \]
by (auto simp add: eq-divide-eq)

**lemma** divide-eq-eq-1 [simp]:
\[ (b/a = 1) = ((a \neq 0 \land a = b)) \]
by (auto simp add: divide-eq-eq)

**lemma** abs-div-pos: 0 < y ==>
\[ |x| \div y = |x / y| \]
apply (subst abs-divide)
apply (simp add: order-less-imp-le)
done

lemma zero-le-divide-abs-iff [simp]: (0 ≤ a / |b|) = (0 ≤ a ∨ b = 0)
by (auto simp: zero-le-divide-iff)

lemma divide-le-0-abs-iff [simp]: (a / |b| ≤ 0) = (a ≤ 0 ∨ b = 0)
by (auto simp: divide-le-0-iff)

lemma field-le-mult-one-interval:
assumes ∗: \( \forall z. (0 < z; z < 1) \Rightarrow z \cdot x \leq y \)
supports \( x \leq y \)
proof (cases 0 < x)
  assume 0 < x
  thus \(?thesis\)
    using dense-le-bounded[of 0 1 y / x] \* unfolding le-divide-eq if-P \[ OF \langle 0 < x \rangle \] by simp
next
  assume \( \neg 0 < x \) hence \( x \leq 0 \)
    by simp
obtain s::'a where s: \( 0 < s s < 1 \) \* unfolding le-divide-eq if-P[of 0 < x] by fastforce
  then have \( (r/s) \cdot (v - u) \leq 1 \cdot (v - u) \)
    apply (rule mult-right-mono)
    using \( \neg 0 < x \)
    by simp
  then show \(?thesis \)
    by (simp add: field-simps)
qed

For creating values between \( u \) and \( v \).

lemma scaling-mono:
assumes u ≤ v 0 ≤ r r ≤ s
  shows u + r \cdot (v - u) / s ≤ v
proof –
  have r/s ≤ 1 using \( \neg 0 < x \)
    using \( \neg 0 < x \)
    by fastforce
  then have \( (r/s) \cdot (v - u) \leq 1 \cdot (v - u) \)
    apply (rule mult-right-mono)
    using \( \neg 0 < x \)
    by simp
  then show \(?thesis \)
    by (simp add: field-simps)
qed

Min/max Simplification Rules

lemma min-mult-distrib-left:
fixes x::'a::linordered-idom
  shows \( p \cdot \min x y = (if 0 \leq p then \min (p\cdot x) (p\cdot y) \else \max (p\cdot x) (p\cdot y)) \)
by (auto simp add: min-def max-def mult-le-cancel-left)

lemma min-mult-distrib-right:
fixes \( x :: 'a :: \text{linordered-idom} \)
shows \( \min x y * p = (\text{if } 0 \leq p \text{ then } \min (x * p) (y * p) \text{ else } \max (x * p) (y * p)) \)
by (auto simp add: min_def max_def mult_le_cancel_right)

lemma min-divide-distrib-right:
fixes \( x :: 'a :: \text{linordered-field} \)
shows \( \min x y / p = (\text{if } 0 \leq p \text{ then } \min (x / p) (y / p) \text{ else } \max (x / p) (y / p)) \)
by (simp add: min_mult_distrib_right divide_inverse)

lemma max-mult-distrib-left:
fixes \( x :: 'a :: \text{linordered-idom} \)
shows \( p * \max x y = (\text{if } 0 \leq p \text{ then } \max (p * x) (p * y) \text{ else } \min (p * x) (p * y)) \)
by (auto simp add: min_def max_def mult_le_cancel_left)

lemma max-mult-distrib-right:
fixes \( x :: 'a :: \text{linordered-idom} \)
shows \( \max x y * p = (\text{if } 0 \leq p \text{ then } \max (x * p) (y * p) \text{ else } \min (x * p) (y * p)) \)
by (auto simp add: min_def max_def mult_le_cancel_right)

lemma max-divide-distrib-right:
fixes \( x :: 'a :: \text{linordered-field} \)
shows \( \max x y / p = (\text{if } 0 \leq p \text{ then } \max (x / p) (y / p) \text{ else } \min (x / p) (y / p)) \)
by (simp add: max_mult_distrib_right divide_inverse)

hide-fact (open) field-inverse field-divide-inverse field-inverse-zero

code-identifier
  code-module Fields → (SML) Arith and (OCaml) Arith and (Haskell) Arith

end

17 Finite sets

theory Finite-Set
  imports Product-Type Sum-Type Fields
begin

17.1 Predicate for finite sets

text notes [[inductive-internals]]
begin

inductive finite :: 'a set ⇒ bool
where
  emptyI [simp, intro!]: finite {}
| insertI [simp, intro!]: finite A → finite (insert a A)

end
simproc-setup finite-Collect (finite (Collect P)) = (K Set-Comprehension-Pointfree.simproc)

declare [[simproc del: finite-Collect]]

lemma finite-induct [case-names empty insert, induct set: finite]:
— Discharging $x \notin F$ entails extra work.
assumes finite $F$
assumes $P \{\}$
and insert: $\forall x. \text{finite } F \implies x \notin F \implies P F \implies P (\text{insert } x F)$
shows $P F$
using (finite $F$)
proof induct
show $P \{\}$ by fact
next
fix $x F$
assume $F$: finite $F$ and $P$: $P F$
show $P (\text{insert } x F)$
proof cases
  assume $x \in F$
  then have insert $x F = F$ by (rule insert-absorb)
  with $P$ show $\text{thesis}$ by (simp only:)
next
  assume $x \notin F$
  from $F$ this $P$ show $\text{thesis}$ by (rule insert)
qed

lemma infinite-finite-induct [case-names infinite empty insert]:
assumes infinite: $\forall A. \neg \text{finite } A \implies P A$
and empty: $P \{\}$
and insert: $\forall x. \text{finite } F \implies x \notin F \implies P F \implies P (\text{insert } x F)$
shows $P A$
proof (cases finite $A$)
  case False
  with infinite show $\text{thesis}$.
next
  case True
  then show $\text{thesis}$ by (induct $A$) (fact empty insert)+
qed

17.1.1 Choice principles

lemma ex-new-if-finite: — does not depend on def of finite at all
assumes $\neg \text{finite } (\text{UNIV :: } 'a \text{ set})$ and finite $A$
shows $\exists a::'a. a \notin A$
proof —
  from assms have $A \neq \text{UNIV}$ by blast
  then show $\text{thesis}$ by blast
qed
A finite choice principle. Does not need the SOME choice operator.

**Lemma** `finite-set-choice`: finite $A \implies \forall x \in A. \exists y. \forall x \in A. P x (f x)$

**Proof** (induct rule: `finite-induct`)

- **Case** `empty`
  - then show `?case` by `simp`

- **Case** `(insert a A)`
  - then obtain $f b$ where $f: \forall x \in A. P x (f x)$ and $ab: P a b$
    - by `auto`
  - show `?case` (is $\exists f. ?P f$)
    - proof
      - show $?P (\lambda x. a x = a \text{ then } b \text{ else } f x)$
        - using $f ab$ by `auto`
    - qed
  - qed

17.1.2 Finite sets are the images of initial segments of natural numbers

**Lemma** `finite-imp-nat-seg-image-inj-on`:

- **Assumes** `finite A`
- **Shows** $\exists (n :: nat) f. A = f ^{\{ i :: nat. i < n \}} \land \text{inj-on} f \{ i. i < n \}$
- **Proof** `induct`
  - **Case** `empty`
    - show `?case` by `simp`
  - **Case** `(insert a A)`
    - have `notinA`: $a \notin A$ by `fact`
    - from `insert.hyps` obtain $n f$ where $A = f ^{\{ i :: nat. i < n \}} \land \text{inj-on} f \{ i. i < n \}$
      - by `blast`
    - then have `insert a A = f (n := a) ^{\{ i. i < Suc n \}} \land \text{inj-on} (f (n := a)) \{ i. i < Suc n \}`
      - using `notinA` by `(auto simp add: image-def Ball-def inj-on-def less-Suc-eq)`
    - then show `?case` by `blast`
  - qed

**Lemma** `nat-seg-image-imp-finite`: $A = f ^{\{ i :: nat. i < n \}} \implies \text{finite } A$

**Proof** (induct $n$ arbitrary: $A$)

- **Case** `0`
  - then show `?case by simp`

- **Case** `(Suc n)`
  - let `$B = f ^{\{ i. i < n \}}$`
  - have `finB: finite ?B` by `(rule Suc.hyps[OF refl])`
show ?case
proof (cases \( \exists k < n. \ f \ n = f \ k \))
  case True
  then have \( A = ?B \)
    using Suc.prems by (auto simp:less-Suc-eq)
  then show ?thesis
    using finB by simp
next
  case False
  then have \( A = \text{insert} \ (f \ n) \ ?B \)
    using Suc.prems by (auto simp:less-Suc-eq)
  then show ?thesis using finB by simp
qed

lemma finite-conv-nat-seg-image: finite \( A \) \( \iff \) \( \exists n f. \ A = f \cdot \{i::\text{nat}. \ i < n\} \)
by (blast intro: nat-seg-image-imp-finite dest: finite-imp-nat-seg-image-inj-on)

lemma finite-imp-inj-to-nat-seg:
assumes finite \( A \)
shows \( \exists n \ f. \ f \cdot A = \{i::\text{nat}. \ i < n\} \land \text{inj-on} \ f A \)
proof -
  from finite-imp-nat-seg-image-inj-on [OF \( \langle \text{finite} \ A \rangle \)]
  obtain \( f \) and \( n :: \text{nat} \) where bij: \( \text{bij-betw} \ f \ \{i. \ i < n\} \ A \)
    by (auto simp: bij-betw-def)
  let \( \exists f = \text{the-inv-into} \ \{i. \ i < n\} \ f \)
  have inj-on \( \exists f \land \exists f \cdot A = \{i. \ i < n\} \)
    by (fold bij-betw-def) (rule bij-betw-the-inv-into[OF bij])
  then show \( ?\text{thesis} \) by blast
qed

lemma finite-Collect-less-nat [iff]: finite \( \{n::nat. \ n < k\} \)
by (fastforce simp: finite-conv-nat-seg-image)

lemma finite-Collect-le-nat [iff]: finite \( \{n::nat. \ n \leq k\} \)
by (simp add: le-eq-less-or-eq Collect-disj-eq)

17.2 Finiteness and common set operations

lemma rev-finite-subset: finite \( B \) \( \implies \) \( A \subseteq B \implies \text{finite} \ A \)
proof (induct arbitrary: \( A \) rule: finite-induct)
  case empty
  then show \( ?\text{case} \) by simp
next
  case (insert \( x \) \( F \) \( A \))
  have \( A \subseteq \text{insert} \ x \ F \land \text{r:} \ A - \{x\} \subseteq F \implies \text{finite} \ (A - \{x\}) \)
    by fact
  show \( \text{finite} \ A \)
  proof cases
assume $x: x \in A$

with $\tau$ have $A - \{x\} \subseteq F$ by (simp add: subset-insert-iff)

with $\tau$ have finite $(A - \{x\})$.

then have finite $(\text{insert } x (A - \{x\}))$.

also have $\text{insert } x (A - \{x\}) = A$

using $x$ by (rule insert-Diff)

finally show $\theta$thesis.

next

show $\theta$thesis when $A \subseteq F$

using that by fact

assume $x \notin A$

with $A$ show $A \subseteq F$

by (simp add: subset-insert-iff)

qed

lemma finite-subset: $A \subseteq B \Rightarrow \text{finite } B \Rightarrow \text{finite } A$

by (rule rev-finite-subset)

lemma finite-UnI:

assumes finite $F$ and finite $G$

shows finite $(F \cup G)$

using assms by induct simp-all

lemma finite-Un [iff]: finite $(F \cup G) \iff \text{finite } F \land \text{finite } G$

by (blast intro: finite-UnI finite-subset [of $- F \cup G$])

lemma finite-insert [simp]: finite $(\text{insert } a A) \iff \text{finite } A$

proof

have $\text{finite } \{a\} \land \text{finite } A \iff \text{finite } \{a\} \cup A$ by simp

then have $\text{finite } \{a\} \cup A \iff \text{finite } A$ by (simp only: finite-Un)

then show $\theta$thesis by simp

qed

lemma finite-Int [simp, intro]: finite $F \lor \text{finite } G \Rightarrow \text{finite } (F \cap G)$

by (blast intro: finite-subset)

lemma finite-Collect-conjI [simp, intro]:

$\text{finite } \{x. P x \land Q x\} \Rightarrow \text{finite } \{x. P x \land Q x\}$

by (simp add: Collect-conj-eq)

lemma finite-Collect-disjI [simp]:

$\text{finite } \{x. P x \lor Q x\} \iff \text{finite } \{x. P x\} \land \text{finite } \{x. Q x\}$

by (simp add: Collect-disj-eq)

lemma finite-Diff [simp, intro]: finite $A \Rightarrow \text{finite } (A - B)$

by (rule finite-subset, rule Diff-subset)

lemma finite-Diff2 [simp]:
assumes finite B
shows finite (A - B) → finite A
proof
  have finite A → finite ((A - B) ∪ (A ∩ B))
    by (simp add: Un-Diff-Int)
  also have . . . → finite (A - B)
    using finite B by simp
  finally show ?thesis ..
qed

lemma finite-Diff-insert [iff]: finite (A - insert a B) → finite (A - B)
proof
  have finite (A - B) → finite (A - B - {a}) by simp
  moreover have A - insert a B = A - B - {a} by auto
  ultimately show ?thesis by simp
qed

lemma finite-compl [simp]: finite (A :: 'a set) =⇒ finite (- A) ←→ finite (UNIV :: 'a set)
  by (simp add: Compl-eq-Diff-UNIV)

lemma finite-Collect-not [simp]: finite {x :: 'a. P x} =⇒ finite {x. ¬ P x} ←→ finite (UNIV :: 'a set)
  by (simp add: Collect-neg-eq)

lemma finite-Union [simp, intro]: finite A =⇒ (∀M. M ∈ A =⇒ finite M) =⇒ finite (∪ A)
  by (induct rule: finite-induct) simp-all

lemma finite-UN-I [intro]: finite A =⇒ (∀a ∈ A =⇒ finite (B a)) =⇒ finite (∪ a∈A. B a)
  by (induct rule: finite-induct) simp-all

lemma finite-UN [simp]: finite A =⇒ finite (∪ (B ' A)) ←→ (∀x∈A. finite (B x))
  by (blast intro: finite-subset)

lemma finite-Inter [intro]: ∃A∈M. finite A =⇒ finite (∩ M)
  by (blast intro: Inter-lower finite-subset)

lemma finite-INT [intro]: ∃x∈I. finite (A x) =⇒ finite (∩ x∈I. A x)
  by (blast intro: INT-lower finite-subset)

lemma finite-imageI [simp, intro]: finite F =⇒ finite (h ' F)
  by (induct rule: finite-induct) simp-all

lemma finite-image-set [simp]: finite {x. P x} =⇒ finite {f x | x. P x}
  by (simp add: image-Collect [symmetric])

lemma finite-image-set2:
finite \{ x. \ P x \} \implies finite \{ y. \ Q y \} \implies finite \{ f x y \mid x, y. \ P x \land Q y \} 
by (rule finite-subset [where \(B = \bigcup x \in \{ x. \ P x \}, \bigcup y \in \{ y. \ Q y \}. \{ f x y \}\)])
auto

lemma finite-imageD:
  assumes finite (f ' A) and inj-on f A
  shows finite A
  using assms
proof (induct f ' A arbitrary: A)
case empty
  then show ?case by simp
next
case (insert x B)
  then have B-A: insert x B = f ' A
    by simp
  then obtain y where x = f y and y \in A
    by blast
  from B-A (x \notin B: have B = f ' A - \{x\}
    by blast
  with B-A (x \notin B: \(x = f y \land \inj-on f A \land y \in A \land have B = f ' (A - \{y\})
    by (simp add: inj-on-image-set-diff)
moreover from \inj-on f A have \inj-on f (A - \{y\})
  by (rule inj-on-diff)
ultimately have finite (A - \{y\})
  by (rule insert.hyps)
then show finite A
  by simp
qed

lemma finite-image-iff: \inj-on f A \implies finite (f ' A) \iff finite A
  using finite-imageD by blast

lemma finite-surj: finite A \implies B \subseteq f ' A \implies finite B
  by (erule finite-subset) (rule finite-imageI)

lemma finite-range-imageI: finite (range g) \implies finite (range (\lambda x. f (g x)))
  by (drule finite-imageI) (simp add: range-composition)

lemma finite-subset-image:
  assumes finite B
  shows B \subseteq f ' A \implies \exists C \subseteq A. finite C \land B = f ' C
  using assms
proof induct
case empty
  then show ?case by simp
next
case insert
  then show ?case
    by (clarsimp simp del: image-insert simp add: image-insert [symmetric]) blast
qed

lemma all-subset-image: \((\forall B. B \subseteq f \cdot A \rightarrow P \cdot B) \iff (\forall B. B \subseteq A \rightarrow P(f \cdot B))\)
  by (safe elim!: subset-imageE) (use image-mono in ⟨blast+⟩)

lemma all-finite-subset-image:
  \((\forall B. \text{finite } B \land B \subseteq f \cdot A \rightarrow P \cdot B) \iff (\forall B. \text{finite } B \land B \subseteq A \rightarrow P(f \cdot B))\)
proof safe
  fix B :: 'a set
  assume B: finite B B \subseteq f \cdot A and P: \(\forall B. \text{finite } B \land B \subseteq A \rightarrow P(f \cdot B)\)
  show P B
    using finite-subset-image [OF B] P by blast
qed blast

lemma ex-finite-subset-image:
  \((\exists B. \text{finite } B \land B \subseteq f \cdot A \land P \cdot B) \iff (\exists B. \text{finite } B \land B \subseteq A \land P(f \cdot B))\)
proof safe
  fix B :: 'a set
  assume B: finite B B \subseteq f \cdot A and P B
  show \(\exists B. \text{finite } B \land B \subseteq A \land P(f \cdot B)\)
    using finite-subset-image [OF B] (P B) by blast
qed blast

lemma finite-vimage-IntI: finite F \rightarrow inj-on h A \rightarrow finite (h \cdot F \cap A)
proof (induct rule: finite-induct)
  case (insert x F)
  then show \(?thesis\)
    by (simp only: * assms finite-UN-I)
qed simp

lemma finite-finite-vimage-IntI:
  assumes finite F
  and \(\forall y. y \in F \rightarrow finite ((h \cdot \{ y \}) \cap A)\)
  shows finite (h \cdot F \cap A)
proof
  have \(*: h \cdot F \cap A = (\bigsqcup y \in F. (h \cdot \{ y \}) \cap A)\)
    by blast
  show \(?thesis\)
    by (simp only: * assms finite-UN-I)
qed

lemma finite-vimage1: finite F \rightarrow inj h \rightarrow finite (h \cdot F)
using finite-vimage-IntI[of F h UNIV] by auto

lemma finite-vimageD': finite (f \cdot A) \rightarrow A \subseteq range f \rightarrow finite A
by (auto simp add: subset-image-iff intro: finite-subset[rotated])
lemma finite-vimageD: finite \((h -' F)\) \(\implies\) surj \(h\) \(\implies\) finite \(F\)
by (auto dest: finite-vimageD')

lemma finite-vimage-iff: bij \(h\) \(\implies\) finite \((h -' F)\) \(\iff\) finite \(F\)
unfolding bij-def by (auto elim: finite-vimageD finite-vimageI)

lemma finite-Collect-bex [simp]:
assumes finite \(A\)
shows finite \(\{x. \exists y \in A. \ Q x y\}\) \(\iff\) \((\forall y \in A. \ \{x. \ Q x y\}\) by auto
with assms show ?thesis by simp
qed

lemma finite-Collect-bounded-ex [simp]:
assumes finite \(\{y. \ P y\}\)
shows finite \(\{x. \exists y. \ P y \land Q x y\}\) \(\iff\) \((\forall y. \ P y \implies finite \{x. \ Q x y\}\) by auto
with assms show ?thesis by simp
qed

lemma finite-Plus: finite \(A\) \(\implies\) finite \(B\) \(\implies\) finite \((A <+> B)\)
by (simp add: Plus-def)

lemma finite-PlusD:
fixes \(A::'a\) set and \(B::'b\) set
assumes fin: finite \((A <+> B)\)
shows finite \(A\) finite \(B\)
proof –
have \(\text{Inl} ' A \subseteq A <+> B\)
  by auto
then have finite \(\text{Inl} ' A::('a + 'b)\) set)
  using fin by (rule finite-subset)
then show finite \(A\)
  by (rule finite-imageD) (auto intro: inj-onl)
next
have \(\text{Inr} ' B \subseteq A <+> B\)
  by auto
then have finite \(\text{Inr} ' B::('a + 'b)\) set)
  using fin by (rule finite-subset)
then show finite \(B\)
  by (rule finite-imageD) (auto intro: inj-onl)
qed

lemma finite-Plus-iff [simp]: finite \((A <+> B)\) \(\iff\) finite \(A \land finite B\)
by (auto intro: finite-PlusD finite-Plus)
lemma finite-Plus-UNIV-iff [simp]:
  finite (UNIV :: ('a + 'b) set) ↔ finite (UNIV :: 'a set) ∧ finite (UNIV :: 'b set)
  by (subst UNIV-Plus-UNIV [symmetric]) (rule finite-Plus-iff)

lemma finite-SigmaI [simp, intro]:
  finite A ⇒ (⋀a. a ∈ A ⇒ finite (B a)) ⇒ finite (SIGMA a:A. B a)
  unfolding Sigma-def by blast

lemma finite-SigmaI2:
  assumes finite {x∈A. B x ≠ {}} and ⋀a. a ∈ A ⇒ finite (B a)
  shows finite (SIGMA A B)
  proof -
    from assms have finite (Sigma {x∈A. B x ≠ {}} B)
      by auto
    also have Sigma {x:A. B x ≠ {}} B = Sigma A B
      by auto
    finally show ?thesis .
  qed

lemma finite-cartesian-product: finite A ⇒ finite B ⇒ finite (A × B)
  by (rule finite-SigmaI)

lemma finite-Prod-UNIV:
  finite (UNIV :: 'a set) ⇒ finite (UNIV :: 'b set) ⇒ finite (UNIV :: ('a × 'b) set)
  by (simp only: UNIV-Times-UNIV [symmetric] finite-cartesian-product)

lemma finite-cartesian-productD1:
  assumes finite (A × B) and B ≠ {}
  shows finite A
  proof -
    from assms obtain n f where A × B = f ' {i::nat. i < n}
      by (auto simp add: finite-conv-nat-seg-image)
    then have fst ' (A × B) = fst ' f ' {i::nat. i < n}
      by simp
    with B ≠ {} have A = (fst o f) ' {i::nat. i < n}
      by (simp add: image-comp)
    then have ∃n f. A = f ' {i::nat. i < n}
      by blast
    then show ?thesis
      by (auto simp add: finite-conv-nat-seg-image)
  qed

lemma finite-cartesian-productD2:
  assumes finite (A × B) and A ≠ {}
  shows finite B
proof

from assms obtain n f where $A \times B = f \cdot \{i::nat. i < n\}$
by (auto simp add: finite-cone-nat-seg-image)
then have $\text{snd} \cdot (A \times B) = \text{snd} \cdot f \cdot \{i::nat. i < n\}$
by simp
with $A \neq \{\}$ have $B = (\text{snd} \circ f) \cdot \{i::nat. i < n\}$
by (simp add: image-comp)
then have $\exists n f. B = f \cdot \{i::nat. i < n\}$
by blast
then show $\text{thesis}$
by (auto simp add: finite-cone-nat-seg-image)
qed

lemma finite-cartesian-product-iff:
$\begin{align*}
\text{finite} (A \times B) & \iff (A = \{\} \lor B = \{\} \lor (\text{finite} A \land \text{finite} B)) \\
\end{align*}$
by (auto dest: finite-cartesian-productD1 finite-cartesian-productD2 finite-cartesian-product)

lemma finite-prod:
$\begin{align*}
\text{finite} (\text{UNIV} :: (\alpha \times \beta) \text{ set}) & \iff \text{finite} (\text{UNIV} :: \alpha \text{ set}) \land \text{finite} (\text{UNIV} :: \beta \text{ set}) \\
\end{align*}$
using finite-cartesian-product-iff[of UNIV UNIV] by simp

lemma finite-Pow-iff [iff]: $\text{finite} (\text{Pow} A) \iff \text{finite} A$
proof
assume finite (Pow A)
then have finite $(\lambda x. \{x\}) \cdot A$
by (blast intro: finite-subset)
then show finite A
by (rule finite-imageD [unfolded inj-on-def]) simp
next
assume finite A
then show finite (Pow A)
by induct (simp-all add: Pow-insert)
qed

corollary finite-Collect-subsets [simp, intro]: $\text{finite} A \imp \text{finite} \{B. B \subseteq A\}$
by (simp add: Pow-def [symmetric])

lemma finite-set: finite (UNIV :: 'a set set) $\iff$ finite (UNIV :: 'a set)
by (simp only: finite-Pow-iff Pow-UNIV[ symmetric])

lemma finite-UnionD: finite $(\bigcup A) \imp \text{finite} A$
by (blast intro: finite-subset [OF subset-Pow-Union])

lemma finite-bind:
assumes finite S
assumes $\forall x \in S. \text{finite} (f x)$
shows finite $(\text{Set}.\text{bind} S f)$
using assms by (simp add: bind-UNION)
lemma finite-filter [simp]: finite $S$ $\Rightarrow$ finite $(\text{Set.filter} \ P \ S)$
unfolding Set.filter-def by simp

lemma finite-set-of-finite-funs:
  assumes finite $A$ finite $B$
  shows finite $\{ f . \ \forall x. (x \in A \rightarrow f \in B) \land (x \notin A \rightarrow f = d) \}$ (is finite $?S$)
proof (let $?F = \lambda f. \{ (a,b). a \in A \land b = f \in \}.$)
  have $?F \subseteq \text{Pow}(A \times B)$
    by auto
  from finite-subset[OF this] have 1: finite $(?F \subseteq ?S)$
    by simp
  have 2: inj-on $?F$ $?S$
    by (fastforce simp add: inj-on-def set-eq-iff fun-eq-iff)
  show ?thesis
    by (rule finite-imageD[OF 1 2])
qed

lemma not-finite-existsD:
  assumes $\neg$ finite $\{ a. P a \}$
  shows $\exists a. P a$
proof (rule classical)
  assume $\neg$ ?thesis
  with assms show ?thesis
    by auto
qed

17.3 Further induction rules on finite sets

lemma finite-ne-induct [case-names singleton insert, consumes 2]:
  assumes finite $F$ and $F \neq \{ \}$
  shows $\forall x. P \{ x \}$
  and $\forall \ F. \text{finite} \ F \Longrightarrow F \neq \{ \} \Longrightarrow x \notin F \Longrightarrow P \ F \Longrightarrow P \ (\text{insert} x \ F)$
proof induct
  case empty
  then show ?case by simp
next
  case (insert $x \ F$)
  then show ?case by cases auto
qed

lemma finite-subset-induct [consumes 2, case-names empty insert]:
  assumes finite $F$ and $F \subseteq A$
  and $\emptyset: P \{ \}$
  and $\text{insert}: \forall a \ F. \text{finite} \ F \Longrightarrow a \in A \Longrightarrow a \notin F \Longrightarrow P \ F \Longrightarrow P \ (\text{insert} a \ F)$
  shows $P \ F$
  using (finite $F$), ($F \subseteq A$)
proof induct
  show $P \{\}$ by fact
next
  fix $x \in F$
assume finite $F$ and $x \notin F$ and $P$: $F \subseteq A \Longrightarrow PF$ and $i$: insert $x F \subseteq A$
show $P$ (insert $x F$)
proof (rule insert)
  from $i$ show $x \in A$ by blast
  from $i$ have $F \subseteq A$ by blast
  with $P$ show $PF$.
  show finite $F$ by fact
  show $x \notin F$ by fact
qed
qed

lemma finite-empty-induct:
assumes finite $A$
  and $P A$
  and remove: $\forall a A. \text{finite } A \Longrightarrow a \in A \Longrightarrow P A \Longrightarrow P (A - \{a\})$
shows $P \{\}$
proof -
  have $P (A - B)$ if $B \subseteq A$ for $B :: 'a set$
proof -
  from (finite $A$) that have finite $B$
    by (rule rev-finite-subset)
  from this ($B \subseteq A$) show $P (A - B)$
proof induct
  case empty
    from ($P A$) show ?case by simp
next
  case (insert $b B$)
    have $P (A - B - \{b\})$
    proof (rule remove)
      from (finite $A$) show finite $(A - B)$
        by induct auto
      from insert show $b \in A - B$
        by simp
      from insert show $P (A - B)$
        by simp
    qed
    also have $A - B - \{b\} = A - \text{insert } b B$
      by (rule Diff-insert [symmetric])
    finally show ?case .
    qed
    qed
  then have $P (A - A)$ by blast
  then show ?thesis by simp
qed
lemma finite-update-induct [consumes 1, case-names const update]:

assumes finite: finite \{a. f a \neq c\}
and const: P (\lambda a. c)
and update: \forall a b f. finite \{a. f a \neq c\} \Rightarrow f a = c \Rightarrow b \neq c \Rightarrow P f \Rightarrow P
(f(a := b))

shows P f
using finite

proof (induct \{a. f a \neq c\} arbitrary: f)
case empty
with const show ?case by simp
next
case (insert a A)
then have A = \{a'. (f(a := c)) a' \neq c\} and f a \neq c
by auto
with finite A have finite \{a'. (f(a := c)) a' \neq c\}
by simp
have (f(a := c)) a = c
by simp
from insert \{a'. (f(a := c)) a' \neq c\} have P (f(a := c))
by simp
with finite \{a'. (f(a := c)) a' \neq c\} have P ((f(a := c))(a := f a))
by (rule update)
then show ?case by simp
qed

lemma finite-subset-induct' [consumes 2, case-names empty insert]:

assumes finite F and F \subseteq A
and empty: P \{
and insert: \forall a F. [finite F; a \in A; F \subseteq A; a \notin F; P F \} \Rightarrow P (insert a F)

shows P F
using assms(1,2)

proof induct
show P \{
by fact
next
fix x F
assume finite F and x \notin F and
P: F \subseteq A \Rightarrow P F and i: insert x F \subseteq A
show P (insert x F)

proof (rule insert)
from i show x \in A by blast
from i have F \subseteq A by blast
with P show P F .
show finite F by fact
show x \notin F by fact
show F \subseteq A by fact
qed
qed
17.4 Class finite

class finite =
  assumes finite-UNIV: finite (UNIV :: 'a set)
begin

lemma finite [simp]: finite (A :: 'a set)
  by (rule subset-UNIV finite-UNIV finite-subset)+

lemma finite-code [code]: finite (A :: 'a set) ⟷ True
  by simp

end

instance prod :: (finite, finite) finite
  by standard (simp only: UNIV-Times-UNIV [symmetric] finite-cartesian-product finite)

lemma inj-graph: inj (λf. {(x, y). y = f x})
  by (rule inj-onI) (auto simp add: set-eq-iff fun-eq-iff)

instance fun :: (finite, finite) finite
proof
  show finite (UNIV :: ('a ⇒ 'b) set)
  proof (rule finite-imageD)
    let ?graph = λf::'a ⇒ 'b. {(x, y). y = f x}
    have range ?graph ⊆ Pow UNIV
      by simp
    moreover have finite (Pow (UNIV :: ('a * 'b) set))
      by (simp only: finite-Pow-iff finite)
    ultimately show finite (range ?graph)
      by (rule finite-subset)
    show inj ?graph
      by (rule inj-graph)
  qed

instance bool :: finite
  by standard (simp add: UNIV-bool)

instance set :: (finite) finite
  by standard (simp only: Pow-UNIV [symmetric] finite-Pow-iff finite)

instance unit :: finite
  by standard (simp add: UNIV-unit)

instance sum :: (finite, finite) finite
  by standard (simp only: UNIV-Plus-UNIV [symmetric] finite-Plus finite)
17.5 A basic fold functional for finite sets

The intended behaviour is \( \text{fold } f \ z \ \{ x_1, \ldots, x_n \} = f \ x_1 (\ldots (f \ x_n \ z)\ldots) \) if \( f \) is “left-commutative”:

locale comp-fun-commute =
  fixes \( f : 'a \Rightarrow 'b \Rightarrow 'b \)
  assumes comp-fun-commute: \( f \ y \circ f \ x = f \ x \circ f \ y \)
begin

lemma fun-left-comm: \( f \ y \ (f \ x \ z) = f \ x \ (f \ y \ z) \)
using comp-fun-commute by (simp add: fun-eq-iff)

lemma commute-left-comp: \( f \ y \circ (f \ x \circ g) = f \ x \circ (f \ y \circ g) \)
by (simp add: o-assoc comp-fun-commute)
end

inductive fold-graph :: \( ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a \set \Rightarrow 'b \Rightarrow \text{bool} \)
for \( f : 'a \Rightarrow 'b \Rightarrow 'b \) and \( z : 'b \)
where
  emptyI [intro]: fold-graph \( f \ z \ \{ \} \ z \)
  insertI [intro]: \( x \notin A \Longrightarrow \text{fold-graph } f \ z \ A \ y \Longrightarrow \text{fold-graph } f \ z \ (\text{insert } x \ A) \ (f \ x \ y) \)

inductive-cases empty-fold-graphE [elim!]: fold-graph \( f \ z \ \{ \} \ x \)

lemma fold-graph-closed-lemma: fold-graph \( f \ z \ A \ x \land x \in B \)
if fold-graph \( g \ z \ A \ x \)
  \( \forall a. b. a \in A \Longrightarrow b \in B \Longrightarrow f \ a \ b = g \ a \ b \)
  \( \forall a. b. a \in A \Longrightarrow b \in B \Longrightarrow g \ a \ b \in B \)
  \( z \in B \)
using that(1–3)
proof (induction rule: fold-graph.induct)
case (insertI \( x \ A \ y \))
have fold-graph \( f \ z \ A \ y \ y \in B \)
  unfolding atomize-conj
  by (rule insertI.IH) (auto intro: insertI.prems)
then have \( g \ x \ y \in B \) and \( f \ x \ y = g \ x \ y \)
  by (auto simp: insertI.prems)
moreover have fold-graph \( f \ z \ (\text{insert } x \ A) \ (f \ x \ y) \)
  by (rule fold-graph.insertI; fact)
ultimately
show \( ?\text{case} \)
  by (simp add: f-eq)
qed (auto intro!: that)

lemma fold-graph-closed-eq: fold-graph \( f \ z \ A = \text{fold-graph } g \ z \ A \)
if $\forall a, b. a \in A \Rightarrow b \in B \Rightarrow f a b = g a b$
$\forall a, b. a \in A \Rightarrow b \in B \Rightarrow g a b \in B$
$z \in B$

using fold-graph-closed-lemma[of $f z A - B g$]

$z \in B$

A tempting alternative for the definiens is if finite $A$ then THE $y$. fold-graph $f z A y$ else $e$. It allows the removal of finiteness assumptions from the theorems fold-comm, fold-reindex and fold-distrib. The proofs become ugly. It is not worth the effort. (???)

lemma finite-imp-fold-graph: finite $A$ $\Rightarrow \exists x$. fold-graph $f z A x$

proof (induct rule: finite-induct) auto

17.5.1 From fold-graph to fold

class context comp-fun-commute

begin

lemma fold-graph-finite:
  assumes fold-graph $f z A y$
  shows finite $A$
  using assms by (induct simp-all)

lemma fold-graph-insertE-aux:
  fold-graph $f z A y \Rightarrow a \in A \Rightarrow \exists y'. y = f a y' \land$ fold-graph $f z (A - \{a\}) y'$

proof (induct set: fold-graph)
  case emptyI
  then show ?thesis by simp
next
  case (insertI $x A y$)
  show ?thesis
  proof (cases $x = a$)
    case True
    with insertI show ?thesis by auto
next
  case False
  then obtain $y'$ where $y = f a y'$ and $y'$: fold-graph $f z (A - \{a\}) y'$
  using insertI by auto
have \( f x y = f a (f x y') \)

unfolding \( y \) by (rule fun-left-comm)

moreover have \( \text{fold-graph } f z (\text{insert } x A \setminus \{a\}) (f x y') \)

using \( y' \) and \( x \neq a \) and \( x \notin A \)

by (simp add: insert-Diff-if fold-graph.insertI)

ultimately show \( \text{thesis} \)

by fast

qed

lemma fold-graph-insertE:

assumes \( \text{fold-graph } f z (\text{insert } x A) v \) \( \text{and} \ x \notin A \)

obtains \( y \) where \( v = f x y \) \( \text{and} \) \( \text{fold-graph } f z A y \)

using \( \text{assms} \) by (auto dest: fold-graph-insertE-aux [OF insertI1])

lemma fold-graph-determ: \( \text{fold-graph } f z A x \Rightarrow \text{fold-graph } f z A y \Rightarrow y = x \)

proof (induct arbitrary: \( y \) set: fold-graph)

case emptyI

then show \( \text{?case} \) by fast

next

case (insertI x A y v)

from \( \text{fold-graph } f z (\text{insert } x A) v \) \( \text{and} \ x \notin A \)

obtain \( y' \) where \( v = f x y' \) \( \text{and} \) \( \text{fold-graph } f z A y' \)

by (rule fold-graph-insertE)

from \( \text{fold-graph } f z A y' \) have \( y' = y \)

by (rule insertI)

with \( v = f x y' \) show \( v = f x y \)

by simp

qed

lemma fold-equality: \( \text{fold-graph } f z A y \Rightarrow \text{fold } f z A = y \)

by (cases finite \( A \)) (auto simp add: fold-def intro: fold-graph-determ dest: fold-graph-finite)

lemma fold-graph-fold:

assumes finite \( A \)

shows \( \text{fold-graph } f z A (\text{fold } f z A) \)

proof –

from \( \text{assms} \) have \( \exists x. \text{fold-graph } f z A x \)

by (rule finite-imp-fold-graph)

moreover note fold-graph-determ

ultimately have \( \exists! x. \text{fold-graph } f z A x \)

by (rule ex-ex1I)

then have \( \text{fold-graph } f z A (\text{The } (\text{fold-graph } f z A)) \)

by (rule theI')

with \( \text{assms} \) show \( \text{thesis} \)

by (simp add: fold-def)

qed

The base case for \( \text{fold} \):
lemma (in −) fold-infinite [simp]: \(\neg \text{finite } A \implies \text{fold } f \ z \ A = z\)
by (auto simp: fold-def)

lemma (in −) fold-empty [simp]: \(\text{fold } f \ z \ \{\} = z\)
by (auto simp: fold-def)

The various recursion equations for fold:

lemma fold-insert [simp]:
assumes finite A and \(x \notin A\)
shows \(\text{fold } f \ z \ (\text{insert } x \ A) = f x \ (\text{fold } f \ z \ A)\)
proof (rule fold-equality)
fix \(z\)
from ⟨finite A⟩ have fold-graph f z A (fold f z A)
by (rule fold-graph-fold)
with ⟨x /∈ A⟩ have fold-graph f z (insert x A) (f x (fold f z A))
by (rule fold-graph-insertI)
then show fold-graph f z (insert x A) (f x (fold f z A))
by simp
qed
declare (in −) empty-fold-graphE [rule del] fold-graph.intros [rule del]
— No more proofs involve these.

lemma fold-fun-left-comm: finite A \(\implies f x \ (\text{fold } f \ z \ A) = \text{fold } f \ (f x z) \ A\)
proof (induct rule: finite-induct)
case empty
then show ?case by simp
next
case insert
then show ?case by (simp add: fun-left-comm [of x])
qed

lemma fold-insert2: finite A \(\implies x \notin A \implies \text{fold } f \ z \ (\text{insert } x \ A) = \text{fold } f \ (f x z) \ A\)
by (simp add: fold-fun-left-comm)

lemma fold-rec:
assumes finite A and \(x \in A\)
shows \(\text{fold } f \ z \ A = f x \ (\text{fold } f \ z \ (A - \{x\}))\)
proof
have A: \(A = \text{insert } x \ (A - \{x\})\)
using ⟨x \in A⟩ by blast
then have fold f z A = fold f z (insert x (A - \{x\}))
by simp
also have ... = f x (fold f z (A - \{x\}))
by (rule fold-insert) (simp add: (finite A)+)
finally show ?thesis .
qed
THEORY “Finite-Set"

444

lemma fold-insert-remove:
  assumes finite A
  shows fold f z (insert x A) = f x (fold f z (A - {x}))
proof -
  from ⟨finite A⟩ have finite (insert x A)
    by auto
  moreover have x ∈ insert x A
    by auto
  ultimately have fold f z (insert x A) = f x (fold f z (insert x A - {x}))
    by (rule fold-rec)
  then show ?thesis
    by simp
 qed

lemma fold-set-union-disj:
  assumes finite A finite B A ∩ B = {}
  shows Finite-Set.fold f z (A ∪ B) = Finite-Set.fold f (Finite-Set.fold f z A) B
using assms(2,1,3) by induct simp-all

end

Other properties of fold:

lemma fold-image:
  assumes inj-on g A
  shows fold f z (g ' A) = fold (f ◦ g) z A
proof (cases finite A)
  case False
  with assms show ?thesis
    by (auto dest: finite-imageD simp add: fold-def)
next
  case True
  have fold-graph f z (g ' A) = fold-graph (f ◦ g) z A
  proof
    fix w
    show fold-graph f z (g ' A) w ←→ fold-graph (f ◦ g) z A w (is ?P ←→ ?Q)
    proof
      assume ?P
      then show ?Q
        using assms
        proof (induct g ' A w arbitrary: A)
          case emptyI
          then show ?case by (auto intro: fold-graph.emptyI)
        next
          case (insertI x A r B)
          from ⟨inj-on g B; x ∉ A; insert x A = image g B⟩ obtain x' A'
            where x' ∉ A' and [simp]: B = insert x' A' x = g x' A = g ' A'
            by (rule inj-img-insertE)
          from insertI.prems have fold-graph (f ◦ g) z A' r
by (auto intro: insertI.hyps)
with \( x' \notin A' \) have \( \text{fold-graph} (f \circ g) z (\text{insert} x' A') ((f \circ g) x' r) \)
by (rule fold-graph.insertI)
then show \( ?\text{case} \)
by simp
qed

next
assume \( ?Q \)
then show \( ?P \)
using assms
proof induct
case emptyI
then show \( ?\text{case} \)
by (auto intro: fold-graph.emptyI)

next
case (insertI \( x A r \))
from \( x \notin A \) insertI.prems have \( g x \notin g' A \)
by auto
moreover from insertI have \( \text{fold-graph} f z (g' A) r \)
by simp
ultimately have \( \text{fold-graph} f z (\text{insert} (g x) (g' A)) (f (g x) r) \)
by (rule fold-graph.insertI)
then show \( ?\text{case} \)
by simp
qed

qed

with True assms show \( ?\text{thesis} \)
by (auto simp add: fold-def)
qed

lemma fold-cong:
assumes comp-fun-commute \( f \) comp-fun-commute \( g \)
and finite \( A \)
and cong: \( \forall x. x \in A \rightarrow f x = g x \)
and \( s = t \) and \( A = B \)
shows \( \text{fold} f s A = \text{fold} g t B \)
proof -
have \( \text{fold} f s A = \text{fold} g s A \)
using (finite \( A \) cong)
proof (induct \( A \))
case empty
then show \( ?\text{case} \) by simp
next
case insert
interpret \( f: \text{comp-fun-commute} f \) by (fact (comp-fun-commute \( f \)))
interpret \( g: \text{comp-fun-commute} g \) by (fact (comp-fun-commute \( g \)))
from insert show \( ?\text{case} \) by simp
qed
with assms show ?thesis by simp

qed

A simplified version for idempotent functions:

locale comp-fun-idem = comp-fun-commute +
  assumes comp-fun-idem: \( f \circ f x = f x \)
begin

lemma fun-left-idem: \( f x \circ (f x z) = f x z \)
  using comp-fun-idem by (simp add: fun-eq-iff)

lemma fold-insert-idem,
  assumes fin: finite A
  shows fold f z (insert x A) = f x (fold f z A)
proof cases
  assume x \in A
  then obtain B where A = insert x B and x /\in B
    by (rule set-insert)
  then show ?thesis
    using assms by (simp add: comp-fun-idem fun-left-idem)
next
  assume x /\in A
  then show ?thesis
    using assms by simp
qed

declare fold-insert [simp del] fold-insert-idem [simp]

lemma fold-insert-idem2: finite A \implies fold f z (insert x A) = fold f (f x z) A
  by (simp add: fold-fun-left-comm)

end

17.5.2 Liftings to comp-fun-commute etc.

lemma (in comp-fun-commute) comp-fun-commute: comp-fun-commute (f \circ g)
  by standard (simp-all add: comp-fun-commute)

lemma (in comp-fun-idem) comp-fun-idem: comp-fun-idem (f \circ g)
  by (rule comp-fun-idem.intro, rule comp-fun-comp-commute, unfold-locales)
  (simp-all add: comp-fun-idem)

lemma (in comp-fun-commute) comp-fun-commute-funpow: comp-fun-commute (\( \lambda x. \ f x \ promptly g x \))
proof
  show \( \prompt^\prime\ \ g y \circ f x \ promptly g x = f x \ promptly g x \circ f y \ promptly g y \) for x y
  proof (cases x = y)
    case True
  qed
then show ?thesis by simp
next
case False
show ?thesis
proof (induct g x arbitrary: g)
case 0
then show ?case by simp
next
case (Suc n g)
have hyp1: f y ∘ f x = f x ∘ f y ∘ g y
proof (induct g y arbitrary: g)
case 0
then show ?case by simp
next
case (Suc n g)
define h where h z = g z + 1 for z
with Suc have n = h y
by simp
with Suc have hyp: f y ∘ h y ∘ f x = f x ∘ f y ∘ h y
by auto
from Suc h-def have g y = Suc (h y)
by simp
then show ?case
by (simp add: comp-assoc hyp) (simp add: o-assoc comp-fun-commute)
qed
define h where h z = (if z = x then g x + 1 else g z) for z
with Suc have n = h x
by simp
with Suc have f y ∘ h y ∘ f x ∘ h x = f x ∘ h x ∘ f y ∘ h y
by auto
with False h-def have hyp2: f y ∘ h y ∘ f x ∘ h x = f x ∘ h x ∘ f y ∘ h y
by simp
from Suc h-def have g x = Suc (h x)
by simp
then show ?case
by (simp del: funpow.simps add: funpow-Suc-right o-assoc hyp2) (simp add: comp-assoc hyp1)
qed
qed
qed

17.5.3 Expressing set operations via fold
lemma comp-fun-commute-const: comp-fun-commute (λ-. f)
by standard rule

lemma comp-fun-idem-insert: comp-fun-idem insert
by standard auto
lemma comp-fun-idem-remove: comp-fun-idem Set.remove
   by standard auto

lemma (in semilattice-inf) comp-fun-idem-inf: comp-fun-idem inf
   by standard (auto simp add: inf-left-commute)

lemma (in semilattice-sup) comp-fun-idem-sup: comp-fun-idem sup
   by standard (auto simp add: sup-left-commute)

lemma union-fold-insert:
   assumes finite A
   shows A ∪ B = fold insert B A
proof –
   interpret comp-fun-idem insert
   by (fact comp-fun-idem-insert)
   from ⟨finite A⟩ show ?thesis
   by (induct A arbitrary: B) simp-all
qed

lemma minus-fold-remove:
   assumes finite A
   shows B − A = fold Set.remove B A
proof –
   interpret comp-fun-idem Set.remove
   by (fact comp-fun-idem-remove)
   from ⟨finite A⟩ have fold Set.remove B A = B − A
   by (induct A arbitrary: B) auto
   then show ?thesis ..
qed

lemma comp-fun-commute-filter-fold:
   comp-fun-commute (λx A'. if P x then Set.insert x A' else A')
proof –
   interpret comp-fun-idem Set.insert by (fact comp-fun-idem-insert)
   show ?thesis by standard (auto simp: fun-eq-iff)
qed

lemma Set-filter-fold:
   assumes finite A
   shows Set.filter P A = fold (λx A'. if P x then Set.insert x A' else A') {} A
   using assms
   by induct
   (auto simp add: Set.filter_def comp-fun-commute.fold-insert[OF comp-fun-commute-filter-fold])

lemma inter-Set-filter:
   assumes finite B
   shows A ∩ B = Set.filter (λx. x ∈ A) B
   using assms
   by induct (auto simp: Set.filter_def)
lemma image-fold-insert:
  assumes finite A
  shows image f A = fold (λk A. Set.insert (f k) A) {} A
proof –
  interpret comp-fun-commute λk A. Set.insert (f k) A
  by standard auto
  show ?thesis
    using assms by (induct A) auto
qed

lemma Ball-fold:
  assumes finite A
  shows Ball A P = fold (λk s. s ∧ P k) True A
proof –
  interpret comp-fun-commute λk s. s ∧ P k
  by standard auto
  show ?thesis
    using assms by (induct A) auto
qed

lemma Bex-fold:
  assumes finite A
  shows Bex A P = fold (λk s. s ∨ P k) False A
proof –
  interpret comp-fun-commute λk s. s ∨ P k
  by standard auto
  show ?thesis
    using assms by (induct A) auto
qed

lemma comp-fun-commute-Pow-fold: comp-fun-commute (λx A. A ∪ Set.insert x ' A)
  by (clarsimp simp: fun-eq-iff comp-fun-commute-def) blast

lemma Pow-fold:
  assumes finite A
  shows Pow A = fold (λx A. A ∪ Set.insert x ' A) {{}} A
proof –
  interpret comp-fun-commute λx A. A ∪ Set.insert x ' A
  by (rule comp-fun-commute-Pow-fold)
  show ?thesis
    using assms by (induct A) (auto simp: Pow-insert)
qed

lemma fold-union-pair:
  assumes finite B
  shows (∪y∈B. {(x, y)}) ∪ A = fold (λy. Set.insert (x, y)) A B
proof –
interpret \textit{comp-fun-commute} \(\lambda y. \text{Set.insert} \; (x, \; y)\)
by \textit{standard auto}
show \(\text{thesis}\)
using \textit{assms} by (induct arbitrary: \(A\)) simp-all
qed

\textbf{lemma} \textit{comp-fun-commute-product-fold}:
finite \(B\) \(\implies\) \textit{comp-fun-commute} \(\lambda x \; z. \text{fold} \; (\lambda y. \text{Set.insert} \; (x, \; y)) \; z \; B\)
by \textit{standard} (auto simp: fold-union-pair [symmetric])

\textbf{lemma} \textit{product-fold}:
\begin{align*}
\text{assumes} & \quad \text{finite} \; A \; \text{finite} \; B \\
\text{shows} & \quad A \times B = \text{fold} \; (\lambda x \; z. \text{fold} \; (\lambda y. \text{Set.insert} \; (x, \; y)) \; z \; B) \; \{\} \; A \\
\text{using} & \quad \textit{assms unfolding} \; \text{Sigma-def} \\
\text{by} & \quad \text{(induct} \; A) \\
& \quad \text{(simp-all add: comp-fun-commute_product-fold \[OF comp-fun-commute-product-fold\] fold-union-pair)}
\end{align*}

\textbf{context} \textit{complete-lattice}
\begin{align*}
\text{begin} \\
\textbf{lemma} \; \textit{inf-Inf-fold-inf}:
\begin{align*}
\text{assumes} & \quad \text{finite} \; A \\
\text{shows} & \quad \text{inf} \; (\text{Inf} \; A) \; B = \text{fold} \; \text{inf} \; B \; A \\
\text{proof} & \quad - \\
& \quad \text{interpret} \; \textit{comp-fun-idem inf} \\
& \quad \text{by} \quad \text{(fact comp-fun-idem-inf)} \\
& \quad \text{from} \; \langle \text{finite} \; A \rangle \; \text{fold-fun-left-comm} \; \text{show} \; \text{thesis} \\
& \quad \text{by} \quad \text{(induct} \; A \; \text{arbitrary:} \; B) \; \text{(simp-all add: inf-commute fun-eq-iff)}
\end{align*}
\text{qed} \\
\textbf{lemma} \; \textit{sup-Sup-fold-sup}:
\begin{align*}
\text{assumes} & \quad \text{finite} \; A \\
\text{shows} & \quad \text{sup} \; (\text{Sup} \; A) \; B = \text{fold} \; \text{sup} \; B \; A \\
\text{proof} & \quad - \\
& \quad \text{interpret} \; \textit{comp-fun-idem sup} \\
& \quad \text{by} \quad \text{(fact comp-fun-idem-sup)} \\
& \quad \text{from} \; \langle \text{finite} \; A \rangle \; \text{fold-fun-left-comm} \; \text{show} \; \text{thesis} \\
& \quad \text{by} \quad \text{(induct} \; A \; \text{arbitrary:} \; B) \; \text{(simp-all add: sup-commute fun-eq-iff)}
\end{align*}
\text{qed} \\
\textbf{lemma} \; \textit{Inf-fold-inf}:
finite \(A\) \(\implies\) \textit{Inf} \; A = \text{fold} \; \text{inf} \; \text{top} \; A \\
\text{using} \; \textit{inf-Inf-fold-inf} \; \text{[of} \; A \; \text{top]} \; \text{by} \; \text{(simp add: inf-absorb2)}

\textbf{lemma} \; \textit{Sup-fold-sup}:
finite \(A\) \(\implies\) \textit{Sup} \; A = \text{fold} \; \text{sup} \; \text{bot} \; A \\
\text{using} \; \textit{sup-Sup-fold-sup} \; \text{[of} \; A \; \text{bot]} \; \text{by} \; \text{(simp add: sup-absorb2)}

\textbf{lemma} \; \textit{inf-INF-fold-inf}:
\begin{align*}
\text{assumes} & \quad \text{finite} \; A
\end{align*}
shows \( \inf B (\bigcap (f \ A)) = \text{fold} (\inf \circ f) B A \) (is \( ?\inf = ?\text{fold} \))

proof –
interpret comp-fun-idem inf by (fact comp-fun-idem-inf)
interpret comp-fun-idem inf \( \circ f \) by (fact comp-comp-fun-idem)
from \( \text{finite } A \) have \( ?\text{fold} = ?\inf \)
by (induct \( A \) arbitrary: \( B \)) (simp-all add: inf-left-commute)
then show \( ?\text{thesis} \) ..
qed

lemma sup-SUP-fold-sup:
assumes finite \( A \)
shows \( \sup B (\bigcup (f \ A)) = \text{fold} (\sup \circ f) B A \) (is \( ?\sup = ?\text{fold} \))
proof –
interpret comp-fun-idem sup by (fact comp-fun-idem-sup)
interpret comp-fun-idem sup \( \circ f \) by (fact comp-comp-fun-idem)
from \( \text{finite } A \) have \( ?\text{fold} = ?\sup \)
by (induct \( A \) arbitrary: \( B \)) (simp-all add: sup-left-commute)
then show \( ?\text{thesis} \) ..
qed

lemma INF-fold-inf: finite \( A \) \( \Rightarrow \) \( \bigcap (f \ A) = \text{fold} (\inf \circ f) \text{ top } A \)
using inf-INF-fold-inf \([\text{of } A \text{ top}]\) by simp

lemma SUP-fold-sup: finite \( A \) \( \Rightarrow \) \( \bigcup (f \ A) = \text{fold} (\sup \circ f) \text{ bot } A \)
using sup-SUP-fold-sup \([\text{of } A \text{ bot}]\) by simp

end

17.6 Locales as mini-packages for fold operations

17.6.1 The natural case

locale folding =
fixes \( f :: 'a \Rightarrow 'b \Rightarrow 'b \) and \( z :: 'b \)
assumes comp-fun-commute: \( f y \circ f x = f x \circ f y \)
begin

interpretation fold?: comp-fun-commute \( f \)
by standard (use comp-fun-commute in (simp add: fun-eq-iff))

definition \( F :: 'a \text{ set } \Rightarrow 'b \)
where eq-fold: \( F A = \text{fold } f \ z \ A \)

lemma empty \([\text{simp}]: F \{\} = z \)
by (simp add: eq-fold)

lemma infinite \([\text{simp}]: \neg \text{finite } A \Rightarrow F A = z \)
by (simp add: eq-fold)

lemma insert \([\text{simp}]: \)
assumes finite A and x /∈ A
shows F (insert x A) = f x (F A)
proof –
  from fold-insert assms
  have fold f z (insert x A) = f x (fold f z A) by simp
with ‹finite A› show ‹thesis› by (simp add: eq-fold fun-eq-iff)
qed

lemma remove:
  assumes finite A and x ∈ A
  shows F A = f x (F (A − {x}))
proof –
  from ‹x ∈ A› obtain B where A = insert x B and x /∈ B
    by (auto dest: mk-disjoint-insert)
  moreover from ‹finite A› have ‹finite B› by simp
  ultimately show ‹thesis› by simp
qed

lemma insert-remove: finite A → F (insert x A) = f x (F (A − {x}))
  by (cases x ∈ A) (simp-all add: remove insert-absorb)
end

17.6.2 With idempotency
locale folding-idem = folding +
  assumes comp-fun-idem: f x ◦ f x = f x
begin
declare insert [simp del]

interpretation fold?: comp-fun-idem f
  by standard (insert comp-fun-commute comp-fun-idem, simp add: fun-eq-iff)

lemma insert-idem [simp]:
  assumes finite A
  shows F (insert x A) = f x (F A)
proof –
  from fold-insert-idem assms
  have fold f z (insert x A) = f x (fold f z A) by simp
  with ‹finite A› show ‹thesis› by (simp add: eq-fold fun-eq-iff)
qed

end

17.7 Finite cardinality

The traditional definition card A ≡ LEAST n. ∃ f. A = {f i | i. i < n} is ugly to work with. But now that we have fold things are easy:
global-interpretation card: folding \( \lambda \). Suc 0
  
defines card = folding.F (\( \lambda \). Suc) 0
  
  by standard rule

lemma card-infinite: \( \neg \) finite \( A \implies \) card \( A = 0 \)
  
  by (fact card.infinite)

lemma card-empty: card \( \{\} = 0 \)
  
  by (fact card.empty)

lemma card-insert-disjoint: finite \( A \implies x \notin A \implies \) card (insert \( x \) \( A \)) = Suc (card \( A \))
  
  by (fact card.insert)

lemma card-insert-if: finite \( A \implies \) card (insert \( x \) \( A \)) = (if \( x \in A \) then card \( A \) else Suc (card \( A \)))
  
  by auto (simp add: card.insert-remove card.remove)

lemma card-ge-0-finite: card \( A > 0 \implies \) finite \( A \)
  
  by (rule ccontr) simp

lemma card-0-eq [simp]: finite \( A \implies \) card \( A = 0 \) \iff \( A = \{\} \)
  
  by (auto dest: mk-disjoint-insert)

lemma finite-UNIV-card-ge-0: finite (UNIV :: 'a set) \implies \) card (UNIV :: 'a set) \> 0
  
  by (rule ccontr) simp

lemma card-eq-0-iff: card \( A = 0 \) \iff \( A = \{\} \lor \neg \) finite \( A \)
  
  by auto

lemma card-range-greater-zero: finite (range \( f \)) \implies \) card (range \( f \)) \> 0
  
  by (rule ccontr) (simp add: card-eq-0-iff)

lemma card-gt-0-iff: 0 < card \( A \) \iff \( A = \{\} \land \) finite \( A \)
  
  by (simp add: neq0-conv [symmetric] card-eq-0-iff)

lemma card-Suc-Diff1: finite \( A \implies x \in A \implies \) Suc (card \( A - \{x\} \)) = card \( A \)
  
  apply (rule insert-Diff [THEN subst, where \( t = A \)])
  
  apply assumption
  
  apply (simp del: insert-Diff-single)
  
  done

lemma card-insert-le-m1: \( n > 0 \implies \) card \( y \leq n - 1 \implies \) card (insert \( x \) \( y \)) \leq n
  
  apply (cases finite \( y \))
  
  apply (cases \( x \in y \))
  
  apply (auto simp: insert-absorb)
  
  done
THEORY “Finite-Set”

lemma card-Diff-singleton: finite A ⟹ x ∈ A ⟹ card (A − {x}) = card A − 1
  by (simp add: card-Suc-Diff1 [symmetric])

lemma card-Diff-singleton-if:
  finite A ⟹ card (A − {x}) = (if x ∈ A then card A − 1 else card A)
  by (simp add: card-Diff-singleton)

lemma card-Diff-insert[simp]:
  assumes finite A and a ∈ A and a /∈ B
  shows card (A − insert a B) = card (A − B) − 1
proof –
  have A − insert a B = (A − B) − {a}
    using assms by blast
  then show ?thesis
    using assms by (simp add: card-Diff-singleton)
qed

lemma card-insert: finite A ⟹ card (insert x A) = Suc (card (A − {x}))
  by (fact card-insert-remove)

lemma card-insert-le: finite A ⟹ card A ≤ card (insert x A)
  by (simp add: card-insert-if)

lemma card-Collect-less-nat[simp]: card {i::nat. i < n} = n
  by (induct n) (simp-all add: less-Suc-eq Collect-disj-eq)

lemma card-Collect-le-nat[simp]: card {i::nat. i ≤ n} = Suc n
  using card-Collect-less-nat[of Suc n] by (simp add: less-Suc-eq-le)

lemma card-mono:
  assumes finite B and A ⊆ B
  shows card A ≤ card B
proof –
  from assms have finite A
    by (auto intro: finite-subset)
  then show ?thesis
    using assms
  proof (induct A arbitrary: B)
    case empty
    then show ?case by simp
  next
    case (insert x A)
    then have x ∈ B
      by simp
    from insert have A ⊆ B − {x} and finite (B − {x})
      by auto
    with insert.hyps have card A ≤ card (B − {x})
      by auto
    with ⟨finite A⟩ ⟨x /∈ A⟩ ⟨finite B⟩ ⟨x ∈ B⟩ show ?case
by simp (simp only: card.remove)
qed

lemma card-seteq: finite B \implies (\forall A. A \subseteq B \implies card B \leq card A \implies A = B)
apply (induct rule: finite-induct)
apply simp
apply clarify
apply (subgoal-tac finite A \land A - \{x\} \subseteq F)
prefer 2 apply (blast intro: finite-subset, atomize)
apply (drule-tac x = A - \{x\} in spec)
apply (simp add: card-Diff-singleton-if split: if-split-asmp)
apply (case-tac card A, auto)
done

lemma psubset-card-mono: finite B \implies A < B \implies card A < card B
apply (simp add: psubset-eq linorder-not-le [symmetric])
apply (blast dest: card-seteq)
done

lemma card-Un-Int:
assumes finite A finite B
shows card A + card B = card (A \cup B) + card (A \cap B)
using assms
proof (induct A)
case empty
then show ?case by simp
next
case insert
then show ?case
by (auto simp add: insert-absorb Int-insert-left)
qed

lemma card-Un-disjoint: finite A \implies finite B \implies A \cap B = {} \implies card (A \cup B) = card A + card B
using card-Un-Int [of A B] by simp

lemma card-Un-le: card (A \cup B) \leq card A + card B
proof (cases finite A \land finite B)
case True
then show ?thesis
using le_iff_add card-Un-Int [of A B] by auto
qed auto

lemma card-Diff-subset:
assumes finite B
and B \subseteq A
shows card (A - B) = card A - card B
using assms
proof (cases finite A)
  case False
  with assms show ?thesis
    by simp
next
  case True
  with assms show ?thesis
    by (induct B arbitrary: A) simp-all
qed

lemma card-Diff-subset-Int:
  assumes finite (A ∩ B)
  shows card (A − B) = card A − card (A ∩ B)
proof –
  have A − B = A − A ∩ B by auto
  with assms show ?thesis
    by (simp add: card-Diff-subset)
qed

lemma diff-card-le-card-Diff:
  assumes finite B
  shows card A − card B ≤ card (A − B)
proof –
  have card A − card B ≤ card A − card (A ∩ B)
    using card-mono [OF assms Int-lower2, of A] by arith
  also have ... = card (A − B)
    using assms by (simp add: card-Diff-subset-Int)
  finally show ?thesis
qed

lemma card-le-sym-Diff:
  assumes finite A finite B card A ≤ card B
  shows card (A − B) ≤ card (B − A)
proof –
  have card(A − B) = card A − card (A ∩ B) using assms(1,2) by (simp add: card-Diff-subset-Int)
  also have ... ≤ card B − card (A ∩ B) using assms(3) by linarith
  also have ... = card(B − A) using assms(1,2) by (simp add: card-Diff-subset-Int Int-commute)
  finally show ?thesis
qed

lemma card-less-sym-Diff:
  assumes finite A finite B card A < card B
  shows card(A − B) < card(B − A)
proof –
  have card(A − B) = card A − card (A ∩ B) using assms(1,2) by (simp add: card-Diff-subset-Int)
  also have ... < card B − card (A ∩ B) using assms(1,3) by (simp add:
card-mono diff-less-mono
also have ... = card(B - A) using assms(1,2) by(simp add: card-Diff-subset-Int Int-commute)
finally show ?thesis .
qed

lemma card-Diff1-less: finite A ⇒ x ∈ A ⇒ card (A - {x}) < card A
by (rule Suc-less-SucD) (simp add: card-Suc-Diff1 del: card-Diff-insert)

lemma card-Diff2-less: finite A ⇒ x ∈ A ⇒ y ∈ A ⇒ card (A - {x} - {y}) < card A
apply (cases x = y)
  apply (simp add: card-Diff1-less del: card-Diff-insert)
apply (rule less-trans)
prefer 2 apply (auto intro!: card-Diff1-less simp del: card-Diff-insert)
done

lemma card-Diff1-le: finite A ⇒ card (A - {x}) ≤ card A
by (cases x ∈ A) (simp add: card-Diff1-less less-imp-le)

lemma card-psubset: finite B ⇒ A ⊆ B ⇒ card A < card B ⇒ A < B
by (erule psubsetI) blast

lemma card-le-inj:
  assumes fA: finite A
    and fB: finite B
    and c: card A ≤ card B
  shows ∃ f. f ' A ⊆ B ∧ inj-on f A
using fA fB c
proof (induct arbitrary: B rule: finite-induct)
  case empty
  then show ?case by simp
next
case (insert x s t)
  then show ?case
proof (induct rule: finite-induct [OF insert.prems(1)])
    case 1
    then show ?case by simp
next
case (2 y t)
  from 2.prems(1,2,5) 2.hyps(1,2) have cst: card s ≤ card t
  by simp
  from 2.prems(3) [OF 2.hyps(1) cst]
  obtain f where f ' s ⊆ t inj-on f s
  by blast
  with 2.prems(2) 2.hyps(2) show ?case
  apply -
  apply (rule exI[where x = λz. if z = x then y else f z])
  apply (auto simp add: inj-on-def)
done
qed
qed

lemma card-subset-eq:
assumes fB: finite B
and AB: A ⊆ B
and c: card A = card B
shows A = B
proof -
from fB AB have fA: finite A
  by (auto intro: finite-subset)
from fA fB have fBA: finite (B − A)
  by auto
have e: A ∩ (B − A) = {}
  by blast
have eq: A ∪ (B − A) = B
  using AB by blast
from card-Un-disjoint[OF fA fBA e, unfolded eq c] have card (B − A) = 0
  by arith
then have B − A = {}
  unfolding card-eq-0-iff using fA fB by simp
with AB show A = B
  by blast
qed

lemma insert-partition:
x /∈ F ⇒ ∀ c1 ∈ insert x F. ∀ c2 ∈ insert x F. c1 ≠ c2 −→ c1 ∩ c2 = {}
⇒
x ∩ ⋃ F = {}
by auto

lemma infinite-psubset-induct [consumes 1, case_names psubset]:
assumes finite: finite A
  and major: A. finite A ⇒ (A. B ⊂ A ⇒ P B) ⇒ P A
shows P A
using finite
proof (induct A taking: card rule: measure-induct-rule)
case (less A)
have fin: finite A by fact
have ih: card B < card A ⇒ finite B ⇒ P B for B by fact
have P B if B ⊂ A for B
proof -
from that have card B < card A
  using psubset-card-mono fin by blast
moreover
from that have B ⊆ A
  by auto
then have finite B
  using fin finite-subset by blast
ultimately show \( \text{thesis using } \text{ih by } \text{simp} \)
\text{qed}
with \( \text{fin show } P A \text{ using } \text{major by } \text{blast} \)
\text{qed}

\text{lemma } \text{finite-induct-select } \text{[consumes 1, case-names empty select]}:
\text{assumes } \text{finite } S 
\text{and } P \{\}
\text{and } \text{select: } \bigvee T. T \subseteq S \implies P T \implies \exists S - T. P (\text{insert } s T)
\text{shows } P S
\text{proof --}
\text{have } 0 \leq \text{card } S \text{ by } \text{simp}
\text{then have } \exists T \subseteq S. \text{card } T = \text{card } S \land P T
\text{proof (induct rule: dec-induct)}
\text{case base with } (P \{\})
\text{show ?case}
\text{by (intro exI[of - \{\}]) auto}
\text{next}
\text{case (step } n)
\text{then obtain } T \text{ where } T \subseteq S \text{ card } T = n P T
\text{by auto}
\text{with } (n < \text{card } S) \text{ have } T \subseteq S P T
\text{by auto}
\text{with } \text{select[of } T\text{] obtain } s \text{ where } s \in S s \notin T P (\text{insert } s T)
\text{by auto}
\text{with } \text{step(2) } T \text{ (finite } S\text{) show ?case}
\text{by (intro exI[of - insert } s T\text{]) (auto dest: finite-subset)}
\text{qed}
\text{with } (\text{finite } S) \text{ show } P S
\text{by (auto dest: card-subset-eq)}
\text{qed}

\text{lemma } \text{remove-induct } \text{[case-names empty infinite remove]}:
\text{assumes } \text{empty } P (\{\} :: \text{a set})
\text{and } \text{infinite } (\neg \text{finite } B \implies P B)
\text{and } \text{remove: } \bigwedge A. \text{finite } A \implies A \neq \{\} \implies A \subseteq B \implies (\bigwedge x. x \in A \implies P (A - \{x\})) \implies P A
\text{shows } P B
\text{proof (cases finite } B\text{)}
\text{case False}
\text{then show ?thesis by (rule infinite)}
\text{next}
\text{case True}
\text{define } A \text{ where } A = B
\text{with } \text{True have } \text{finite } A A \subseteq B
\text{by simp-all}
\text{then show } P A
\text{proof (induct } A\text{ arbitrary: } A)
\text{case 0}
then have $A = \{\}$ by auto

with empty show ?case by simp

next
  case (Suc n A)
  from $A \subseteq B$ and (finite B) have finite $A$
    by (rule finite-subset)
  moreover from Suc.hyps have $A \neq \{\}$ by auto
  moreover note $A \subseteq B$
  moreover have $P (A - \{x\})$ if $x \in A$ for $x$
    using $x$. Suc.prems (Suc n = card $A$) by (intro Suc) auto
  ultimately show ?case by (rule remove)
  qed

qed

lemma finite-remove-induct [consumes 1, case-names empty remove]:
fixes $P :: 'a set \Rightarrow bool$
assumes finite $B$
and $P \{\}$
and $\forall A. finite A \Rightarrow A \neq \{\} \Rightarrow A \subseteq B \Rightarrow (\forall x. x \in A \Rightarrow P (A - \{x\}))$

\implies $P A$
defines $B' \equiv B$
shows $P B'$
by (induct $B'$ rule: remove-induct) (simp-all add: assms)

Main cardinality theorem.

lemma card-partition [rule-format]:
finite $C \implies finite (\bigcup C) \implies (\forall c \in C. \ card c = k) \implies$
$\forall c1 \in C. \forall c2 \in C. \ c1 \neq c2 \implies c1 \cap c2 = \{\} \implies$
k * card $C = \ card (\bigcup C)$

proof (induct rule: finite-induct)
  case empty
  then show ?case by simp

next
  case (insert $x$ $F$)
  then show ?case
    by (simp add: card-Un-disjoint insert-partition finite-subset [of - $\bigcup (insert -$)])
  qed

lemma card-eq-UNIV-imp-eq-UNIV:
assumes fin: finite (UNIV :: 'a set)
and card: card $A = \ card (UNIV :: 'a set)$
shows $A = (UNIV :: 'a set)$

proof
  show $A \subseteq UNIV$ by simp
  show $UNIV \subseteq A$
  proof
    show $x \in A$ for $x$
    proof (rule ccontr)
      assume $x \notin A$
      qed
  qed

then have \( A \subseteq \text{UNIV} \) by auto
with \( \text{fin} \) have \( \text{card} \ A < \text{card}(\text{UNIV} :: 'a \text{ set}) \)
by (fact psubset-card-mono)
with \( \text{card} \) show \( \text{False} \) by simp
qed
qed
qed

The form of a finite set of given cardinality

**lemma** card-eq-SucD:
assumes \( \text{card} \ A = \text{Suc} \ k \)
shows \( \exists \ b \ B. \ A = \text{insert} \ b \ B \land b \notin B \land \text{card} \ B = k \land (k = 0 \rightarrow B = \{\}) \)
proof –
\begin{itemize}
\item have \( \text{fin: finite} \ A \)
  using \( \text{assms} \) by (auto intro: ccontr)
\item moreover have \( \text{card} \ A \neq 0 \)
  using \( \text{assms} \) by auto
\item essentially obtain \( b \) where \( b \in A \)
  by auto
\item show \( \text{thesis} \)
  proof (intro exI conjI)
  \begin{itemize}
  \item show \( A = \text{insert} \ b (A - \{b\}) \)
    using \( b \) by blast
  \item show \( b \notin A - \{b\} \)
    by blast
  \item show \( \text{card} (A - \{b\}) = k \) \( \land \)
    \( k = 0 \rightarrow A - \{b\} = \{\} \)
    using \( \text{assms} \) \( b \) \( \text{fin} \) by (fastforce dest: mk-disjoint-insert)+
  \end{itemize}
  qed
\end{itemize}
qed

**lemma** card-Suc-eq:
\( \text{card} A = \text{Suc} \ k \)
apply (auto elim!: card-eq-SucD)
apply (subst card.insert)
apply (auto simp add: intro:ccontr)
done

**lemma** card-1-singletonE:
assumes \( \text{card} \ A = 1 \)
obtains \( x \) where \( A = \{x\} \)
using \( \text{assms} \) by (auto simp: card-Suc-eq)

**lemma** is-singleton-altdef: is-singleton \( A \)
\( \leftrightarrow \text{card} \ A = 1 \)
\text{unfolding} is-singleton-def
by (auto elim!: card-1-singletonE is-singletonE simp del: One-nat-def)

**lemma** card-1-singleton-iff: \( \text{card} A = \text{Suc} \ 0 \)
\( \leftrightarrow (\exists \ x. \ A = \{x\}) \)
by (simp add: card-Suc-eq)
lemma card-le-Suc0-iff-eq:
  assumes finite A
  shows card A ≤ Suc 0 ⟷ (∀ a1 ∈ A. ∀ a2 ∈ A. a1 = a2) (is ?C = ?A)
proof
  assume ?C thus ?A using assms by (auto simp: le-Suc-eq dest: card-eq-SucD)
next
  assume ?A show ?C proof cases
    assume A = {} thus ?C using ⟨?A⟩ by simp
  next
    assume A ≠ {} then obtain a where A = {a} using ⟨?A⟩ by blast
    thus ?C by simp
  qed
qed

lemma card-le-Suc-iff:
  Suc n ≤ card A = (∃ a B. A = insert a B ∧ a /∈ B ∧ n ≤ card B ∧ finite B)
apply (cases finite A)
apply (fastforce simp: card-Suc-eq less-eq-nat, insert-eq-iff
  dest: subset-singletonD split: if-splits)
by auto

lemma finite-fun-UNIVD2:
  assumes fin: finite (UNIV :: ('a ⇒ 'b) set)
  shows finite (UNIV :: 'b set)
proof
  from fin have finite (range (λf :: 'a ⇒ 'b. f arbitrary)) for arbitrary
    by (rule finite-imageI)
  moreover have UNIV = range (λf :: 'a ⇒ 'b. f arbitrary) for arbitrary
    by (rule UNIV-eq-I) auto
  ultimately show finite (UNIV :: 'b set)
    by simp
qed

lemma card-UNIV-unit [simp]: card (UNIV :: unit set) = 1
unfolding UNIV-unit by simp

lemma infinite-arbitrarily-large:
  assumes ¬ finite A
  shows ∃ B. finite B ∧ card B = n ∧ B ⊆ A
proof (induction n)
  case 0
  show ?case by (intro exI[of - {}]) auto
next
  case (Suc n)
  then obtain B where B: finite B ∧ card B = n ∧ B ⊆ A ..
with \( \neg \text{finite } A \) have \( A \neq B \) by auto
with \( B \) have \( B \subseteq A \) by auto
then have \( \exists x. x \in A - B \)
  by (elim psubset_imp_ex_mem)
then obtain \( x \) where \( x \in A - B \) ..
with \( B \) have finite \((\text{insert } x \ B) \) \( \land \) card \((\text{insert } x \ B)\) = \( \text{Suc } n \) \( \land \) \( \text{insert } x \ B \subseteq A \)
  by auto
then show \( \exists B. \text{finite } B \land \text{card } B = \text{Suc } n \land B \subseteq A \) ..
qed

Sometimes, to prove that a set is finite, it is convenient to work with finite subsets and to show that their cardinalities are uniformly bounded. This possibility is formalized in the next criterion.

**Lemma** finite-if-finite-subsets-card-bdd:
assumes \( \forall G. G \subseteq F \Rightarrow \text{finite } G \Rightarrow \text{card } G \leq C \)
shows \( \text{finite } F \land \text{card } F \leq C \)
proof (cases \( \text{finite } F \))
case False
obtain \( n :: \text{nat} \) where \( n > \max C \ 0 \) by auto
obtain \( G \) where \( G \subseteq F \) \( \text{card } G = n \) using infinite-arbitrarily-large[OF False] by auto
hence \( \text{finite } G \) using \((n > \max C \ 0)\) using card-infinite gr-implies-not0 by blast
hence False using assms \( G \) \( n \) not-less by auto
thus \(?thesis \) ..
next
case True thus \(?thesis \) using assms[of \( F \)] by auto
qed

17.7.1 Cardinality of image

**Lemma** card-image-le: finite \( A \Rightarrow \text{card } (f \ A) \leq \text{card } A \)
by (induct rule: finite-induct) (simp-all add: le-SucI card-insert-if)

**Lemma** card-image: inj-on \( f \ A \Rightarrow \text{card } (f \ A) = \text{card } A \)
proof (induct \( A \) rule: infinite-finite-induct)
case \( \text{infinite } A \)
then have \( \neg \text{finite } (f \ A) \) by (auto dest: finite-imageD)
with \( \text{infinite } \) show \(?case \) by simp
qed simp-all

**Lemma** bij-betw-same-card: bij-betw \( f \ A \ B \Rightarrow \text{card } A = \text{card } B \)
by (auto simp: card-image bij-betw-def)

**Lemma** endo-inj-surj: finite \( A \Rightarrow f \ A \subseteq A \Rightarrow \text{inj-on } f \ A \Rightarrow f \ A = A \)
by (simp add: card-seteq card-image)

**Lemma** eq-card-imp-inj-on:
assumes finite \( A \) \( \text{card}(f \ A) = \text{card } A \)
shows \( \text{inj-on } f \ A \)
using assms

proof (induct rule:finite-induct)
  case empty
  show ?case by simp
next
  case (insert x A)
  then show ?case
    using card-image-le[of A f]
    by (simp add: card-insert-if split: if_splits)
qed

lemma inj-on-iff-eq-card: finite A \implies inj-on f A \iff (f -' A) = card A
  by (blast intro: card-image eq-card-imp-inj-on)

lemma card-inj-on-le:
  assumes inj-on f A f -' A \subseteq B finite B
  shows card A \leq card B
proof
  have finite A
    using assms
    by (blast intro: finite-imageD dest: finite-subset)
  then show ?thesis
    using assms
    by (force intro: card-mono simp: card-image [symmetric])
qed

lemma inj-on-iff-card-le:
  assumes finite A finite B
  \implies (\exists f. inj-on f A \land f -' A \subseteq B)
  \iff (card A \leq card B)
using card-inj-on-le[of - A B] card-le-inj[of A B]
by blast

lemma surj-card-le: finite A \implies B \subseteq f -' A \iff card B \leq card A
  by (blast intro: card-image-le card-mono le-trans)

lemma card-bij-eq:
  inj-on f A \implies f -' A \subseteq B \implies inj-on g B \implies g -' B \subseteq A \implies finite A \implies finite B
  \implies card A = card B
by (auto intro: le-antisym card-inj-on-le)

lemma bij-betw-finite: bij-betw f A B \implies finite A \iff finite B
  unfolding bij-betw_def using finite-imageD[of f A] by auto

lemma inj-on-finite: inj-on f A \implies f -' A \subseteq B \implies finite B \implies finite A
  using finite-imageD finite-subset by blast

lemma card-vimage-inj: inj f \implies A \subseteq range f \implies card (f -' A) = card A
  by (auto 4 3 simp: subset-image-iff inj-vimage-image-eq
    intro: card-image[symmetric, OF subset-inj-on])

17.7.2 Pigeonhole Principles

lemma pigeonhole: card A > card (f -' A) \implies \neg inj-on f A
  by (auto dest: card-image less-irrefl-nat)
lemma pigeonhole-infinite:
  assumes ¬ finite A and finite (f' A)
  shows ∃ a0 ∈ A. ¬ finite { a ∈ A. f a = f a0 }
  using assms(2,1)
proof (induct f' A arbitrary; A rule: finite-induct)
case empty
  then show ?case by simp
next
case (insert b F)
  show ?case
proof (cases finite { a ∈ A. f a = b })
  case True
  with ⟨ ¬ finite A ⟩ have ¬ finite (A − { a ∈ A. f a = b })
    by simp
  also have A − { a ∈ A. f a = b } = { a ∈ A. f a ≠ b }
    by blast
  finally have ¬ finite { a ∈ A. f a ≠ b }.
  from insert(3)[OF - this] insert(2,4) show ?thesis
    by simp (blast intro: rev-finite-subset)
next
case False
  then have { a ∈ A. f a = b } ≠ {} by force
  with False show ?thesis by blast
qed

lemma pigeonhole-infinite-rel:
  assumes ¬ finite A
  and finite B
  and ∀ a ∈ A. ∃ b ∈ B. R a b
  shows ∃ b ∈ B. ¬ finite { a ∈ A. R a b }
proof
  let ?F = λ a. { b ∈ B. R a b }
  from finite-Pow iff[OF ∴ iffD2, OF (finite B)] have finite (? F ' A)
    by (blast intro: rev-finite-subset)
  from pigeonhole-infinite [where f = ?F, OF assms(1) this]
  obtain a0 where a0 ∈ A and infinite: ¬ finite { a ∈ A. ? F a = ? F a0 } ..
  obtain b0 where b0 ∈ B and R a0 b0
  using a0 ∈ A; assms(3) by blast
  have finite { a ∈ A. ? F a = ? F a0 } if finite { a ∈ A. R a b0 }
    using b0 ∈ B; ⟨ R a0 b0 ⟩; that by (blast intro: rev-finite-subset)
  with infinite { b0 ∈ B } show ?thesis
    by blast
qed

17.7.3 Cardinality of sums

lemma card-Plus:
assumes finite A finite B 
shows card (A <+> B) = card A + card B
proof –
  have Inl'A ∩ Inr'B = {} by fast
  with assms show ?thesis
    by (simp add: Plus-def card-Un-disjoint card-image)
qed

lemma card-Plus-conv-if:
card (A <+> B) = (if finite A ∧ finite B then card A + card B else 0)
by (auto simp add: card-Plus)

Relates to equivalence classes. Based on a theorem of F. Kammüller.

lemma dvd-partition:
assumes f: finite (∪ C)
  and ∀ c∈C. k dvd card c ∀ c1∈C. ∀ c2∈C. c1 ≠ c2 −→ c1 ∩ c2 = {}
shows k dvd card (∪ C)
proof –
  have finite C
    by (rule finite-UnionD [OF f])
  then show ?thesis
    using assms
      proof (induct rule: finite-induct)
        case empty
        show ?case by simp
      next
        case insert
        then show ?case
          apply simp
          apply (subst card-Un-disjoint)
          apply (auto simp add: disjoint-eq-subset-Compl)
          done
      qed
qed

17.7.4 Relating injectivity and surjectivity

lemma finite-surj-inj:
assumes finite A A ⊆ f ′ A
shows inj-on f A
proof –
  have f ′ A = A
    by (rule card-seteq [THEN sgm]) (auto simp add: assms card-image-le)
  then show ?thesis using assms
    by (simp add: eq-card-imp-inj-on)
qed

lemma finite-UNIV-surj-inj: finite(UNIV:: 'a set) ⇒ surj f ⇒ inj f
for f :: 'a ⇒ 'a
by (blast intro: finite-surj-injsubset-UNIV)

**lemma** finite-UNIV-inj-surj: finite(UNIV::'a set) ==> inj f ==> surj f
for f :: 'a => 'a
by (fastforce simp:surj-def dest!: endo-inj-surj)

**lemma** surjective-iff-injective-gen:
assumes fS: finite S
and fT: finite T
and c: card S = card T
and ST: f : S <= T
shows (\forall y \in T. \exists x \in S. f x = y) <-> inj-on f S
(is \?lhs <-\> \?rhs)
proof
assume h: \?lhs
{ 
fix x y
assume x: x \in S
assume y: y \in S
assume f: f x = f y
from x fS have S0: card S \neq 0
  by auto
have x = y
proof (rule ccontr)
  assume xy: \neg \?thesis
  have th: card S <= card (f' (S - \{y\}))
    unfolding c
    proof (rule card-mono)
    show finite (f' (S - \{y\}))
      by (simp add: fS)
    have [x \neq y; x \in S; z \in S; f x = f y]
      ==> \exists x \in S. x \neq y \land f z = f x for z
      by (case-tac z = y --> z = x) auto
    then show T \subseteq f' (S - \{y\})
      using h xy x y f by fastforce
    qed
    also have ... \leq card (S - \{y\})
      by (simp add: card-image-le fS)
    also have ... \leq card S - 1 using y fS by simp
    finally show False using S0 by arith
    qed
  }
then show \?rhs
  unfolding inj-on-def by blast
next
assume h: \?rhs
have f' S = T
  by (simp add: ST c card-image card-subset-eq fT h)
then show \?lhs by blast

Infinite Sets

Some elementary facts about infinite sets, mostly by Stephan Merz. Beware! Because "infinite" merely abbreviates a negation, these lemmas may not work well with blast.

abbreviation infinite :: 'a set ⇒ bool
  where infinite S ≡ ¬ finite S

Infinite sets are non-empty, and if we remove some elements from an infinite set, the result is still infinite.

lemma infinite-UNIV-nat [iff]: infinite (UNIV :: nat set)
proof
  assume finite (UNIV :: nat set)
  with finite-UNIV-inj-surj [of Suc] show False
    by simp (blast dest: Suc-neq-Zero surjD)
qed

lemma infinite-UNIV-char-0:
  infinite (UNIV :: 'a::semiring-char-0 set)
proof
  assume finite (UNIV :: 'a set)
  with subset-UNIV have finite (range of-nat :: 'a set)
    by (rule finite-subset)
  moreover have inj (of-nat :: nat ⇒ 'a)
    by (simp add: inj-on-def)
  ultimately have finite (UNIV :: nat set)
    by (rule finite-imageD)
  then show False
    by simp
qed

lemma infinite-imp-nonempty: infinite S ⇒ S ≠ {}
  by auto

lemma infinite-remove:
  infinite S ⇒ infinite (S − {a})
  by simp

lemma Diff-infinite-finite:
  assumes finite T infinite S
  shows infinite (S − T)
  using finite T
proof induct
  from (infinite S) show infinite (S − {})
    by auto
next
THEORY “Finite-Set”

\(\text{fix } T x\)
\(\text{assume } \text{ih}: \text{infinite } (S - T)\)
\(\text{have } S - (\text{insert } x T) = (S - T) - \{x\}\)
\(\text{by (rule Diff-insert)}\)
\(\text{with } \text{ih show } \text{infinite } (S - (\text{insert } x T))\)
\(\text{by (simp add: infinite-remove)}\)
\(\text{qed}\)

\text{lemma} Un-infinite: \(\text{infinite } S \implies \text{infinite } (S \cup T)\)
\(\text{by simp}\)

\text{lemma} infinite-Un: \(\text{infinite } (S \cup T) \iff \text{infinite } S \lor \text{infinite } T\)
\(\text{by simp}\)

\text{lemma} infinite-super:
\(\text{assumes } S \subseteq T\)
\(\text{and } \text{infinite } S\)
\(\text{shows } \text{infinite } T\)
\(\text{proof}\)
\(\text{assume } \text{finite } T\)
\(\text{with } (S \subseteq T) \text{ have } \text{finite } S \text{ by (simp add: finite-subset)}\)
\(\text{with } (\text{infinite } S) \text{ show False by simp}\)
\(\text{qed}\)

\text{proposition} infinite-coinduct [consumes 1, case-names infinite]:
\(\text{assumes } X A\)
\(\text{and step: } \forall A. \ X A \implies \exists x \in A. \ X (A - \{x\}) \lor \text{infinite } (A - \{x\})\)
\(\text{shows } \text{infinite } A\)
\(\text{proof}\)
\(\text{assume } \text{finite } A\)
\(\text{then show False}\)
\(\text{using } (X A)\)
\(\text{proof (induction rule: finite-psubset-induct)}\)
\(\text{case } \text{psubset } A\)
\(\text{then obtain } x \text{ where } x \in A \ X (A - \{x\}) \lor \text{infinite } (A - \{x\})\)
\(\text{using local.step psubset.prems by blast}\)
\(\text{then have } X (A - \{x\})\)
\(\text{using psubset.hyps by blast}\)
\(\text{show False}\)
\(\text{apply (rule psubset.IH [where } B = A - \{x\}]\))
\(\text{apply (use } x \in A \text{ in blast)}\)
\(\text{apply (simp add: } X (A - \{x\}))\)
\(\text{done}\)
\(\text{qed}\)
\(\text{qed}\)

For any function with infinite domain and finite range there is some element that is the image of infinitely many domain elements. In particular, any infinite sequence of elements from a finite set contains some element that
occurs infinitely often.

**lemma inf-img-fin-dom':**

assumes \( img: \text{finite } (f ' A) \)
and \( dom: \text{infinite } A \)
shows \( \exists y \in f ' A. \text{infinite } (f -' \{y\} \cap A) \)

**proof (rule ccontr)**

have \( A \subseteq (\bigcup y\in f ' A. f -' \{y\} \cap A) \) by auto
moreover assume \( \neg \text{thesis} \)
with \( img \) have \( \text{finite } (\bigcup y\in f ' A. f -' \{y\} \cap A) \) by blast
ultimately have \( \text{finite } A \) by (rule finite-subset)
with \( dom \) show False by contradiction
qed

**lemma inf-img-fin-domE':**

assumes \( \text{finite } (f ' A) \) and \( \text{infinite } A \)
obtains \( y \) where \( y \in f ' A \) and \( \text{infinite } (f -' \{y\} \cap A) \)
using \( \text{assms} \) by (blast dest: inf-img-fin-dom')

**lemma inf-img-fin-dom:**

assumes \( \text{finite } (f ' A) \) and \( \text{infinite } A \)
shows \( \exists y \in f ' A. \text{infinite } (f -' \{y\}) \)
using inf-img-fin-dom'[OF \( \text{assms} \)] by auto

**lemma inf-img-fin-domE:**

assumes \( \text{finite } (f ' A) \) and \( \text{infinite } A \)
obtains \( y \) where \( y \in f ' A \) and \( \text{infinite } (f -' \{y\}) \)
using \( \text{assms} \) by (blast dest: inf-img-fin-dom)

**proposition finite-image-absD:** \( \text{finite } (\text{abs } S) \implies \text{finite } S \)
for \( S :: 'a::linordered-ring set \)
by (rule ccontr) (auto simp: abs-eq-iff vimage-def dest: inf-img-fin-dom)

### 17.9 The finite powerset operator

**definition** \( \text{Fpow } :: 'a \text{ set } \Rightarrow 'a \text{ set set} \)
where \( \text{Fpow } A \equiv \{ X. X \subseteq A \wedge \text{finite } X \} \)

**lemma** \( \text{Fpow-mono}: A \subseteq B \implies \text{Fpow } A \subseteq \text{Fpow } B \)
unfolding \( \text{Fpow-def} \) by auto

**lemma** \( \text{empty-in-Fpow}: \{\} \in \text{Fpow } A \)
unfolding \( \text{Fpow-def} \) by auto

**lemma** \( \text{Fpow-not-empty}: \text{Fpow } A \neq \{\} \)
using \( \text{empty-in-Fpow} \) by blast

**lemma** \( \text{Fpow-subset-Pow}: \text{Fpow } A \subseteq \text{Pow } A \)
unfolding \( \text{Fpow-def} \) by auto
lemma \texttt{Fpow-Pow-finite}: \texttt{Fpow A} = \texttt{Pow A \{A, finite \ A\}}

\textbf{unfolding} \texttt{Fpow-def Pow-def by blast}

lemma \texttt{inj-on-image-Fpow}:
\textbf{assumes} \texttt{inj-on f A}
\textbf{shows} \texttt{inj-on (image f) (Fpow A)}
\textbf{using} \texttt{assms Fpow-subset-Pow[of A] subset-inj-on[of image f Pow A] inj-on-image-Pow by blast}

lemma \texttt{image-Fpow-mono}:
\textbf{assumes} \texttt{f ' A \subseteq B}
\textbf{shows} \texttt{(image f) ' (Fpow A) \subseteq Fpow B}
\textbf{using} \texttt{assms by (unfold Fpow-def, auto)}

end

18 \hspace{1em} \textbf{Relations – as sets of pairs, and binary predicates} \\

\textbf{theory} \texttt{Relation} \\
\textbf{imports} \texttt{Finite-Set} \\
\textbf{begin}

A preliminary: classical rules for reasoning on predicates

\textbf{declare} \texttt{predicate1I [Pure.intro!, intro!]} \\
\textbf{declare} \texttt{predicate1D [Pure.dest, dest]} \\
\textbf{declare} \texttt{predicate2I [Pure.intro!, intro!]} \\
\textbf{declare} \texttt{predicate2D [Pure.dest, dest]} \\
\textbf{declare} \texttt{bot1E [elim!]} \\
\textbf{declare} \texttt{bot2E [elim!]} \\
\textbf{declare} \texttt{top1I [intro!]} \\
\textbf{declare} \texttt{top2I [intro!]} \\
\textbf{declare} \texttt{inf1I [intro!]} \\
\textbf{declare} \texttt{inf2I [intro!]} \\
\textbf{declare} \texttt{inf1E [elim!]} \\
\textbf{declare} \texttt{inf2E [elim!]} \\
\textbf{declare} \texttt{sup1I1 [intro?]}
\textbf{declare} \texttt{sup2I1 [intro?]}
\textbf{declare} \texttt{sup1I2 [intro?]}
\textbf{declare} \texttt{sup2I2 [intro?]}
\textbf{declare} \texttt{sup1E [elim!]} \\
\textbf{declare} \texttt{sup2E [elim!]} \\
\textbf{declare} \texttt{sup1CI [intro!]} \\
\textbf{declare} \texttt{sup2CI [intro!]} \\
\textbf{declare} \texttt{Inf1-I [intro!]} \\
\textbf{declare} \texttt{INF1-I [intro!]} \\
\textbf{declare} \texttt{Inf2-I [intro!]} \\
\textbf{declare} \texttt{INF2-I [intro!]}
18.1 Fundamental

18.1.1 Relations as sets of pairs

type-synonym 'a rel = ('a × 'a) set

lemma subrelI: (∀x y. (x, y) ∈ r → (x, y) ∈ s) → r ⊆ s
— Version of subsetI for binary relations
by auto

lemma lfp-induct2:
  (a, b) ∈ lfp f → mono f →
  (∀a b. (a, b) ∈ f → (lfp f ∩ {(x, y). P x y}) → P a b) → P a b
— Version of lfp-induct for binary relations
using lfp-induct-set [of (a, b) f case-prod P] by auto

18.1.2 Conversions between set and predicate relations

lemma pred-equals-eq [pred-set-conv]: (λx. x ∈ R) = (λx. x ∈ S) ↔ R = S
  by (simp add: set-eq-iff fun-eq-iff)

lemma pred-equals-eq2 [pred-set-conv]: (λx y. (x, y) ∈ R) = (λx y. (x, y) ∈ S)
  ↔ R = S
  by (simp add: set-eq-iff fun-eq-iff)

lemma pred-subset-eq [pred-set-conv]: (λx. x ∈ R) ≤ (λx. x ∈ S) ↔ R ⊆ S
  by (simp add: subset-iff le-fun-def)

lemma pred-subset-eq2 [pred-set-conv]: (λx y. (x, y) ∈ R) ≤ (λx y. (x, y) ∈ S)
  ↔ R ⊆ S
  by (simp add: subset-iff le-fun-def)

lemma bot-empty-eq [pred-set-conv]: ⊥ = (λx. x ∈ {})
by (auto simp add: fun-eq-iff)

lemma bot-empty-eq2 [pred-set-conv]: \( \bot = (\lambda x. \ (x, \ y) \in \{\}) \)
  by (auto simp add: fun-eq-iff)

lemma top-empty-eq [pred-set-conv]: \( \top = (\lambda x. \ x \in \text{UNIV}) \)
  by (auto simp add: fun-eq-iff)

lemma top-empty-eq2 [pred-set-conv]: \( \top = (\lambda x. \ (x, \ y) \in \text{UNIV}) \)
  by (auto simp add: fun-eq-iff)

lemma inf-Int-eq [pred-set-conv]: \( (\lambda x. \ x \in R) \cap (\lambda x. \ x \in S) = (\lambda x. \ x \in R \cap S) \)
  by (simp add: inf-fun-def)

lemma inf-Int-eq2 [pred-set-conv]: \( (\lambda x y. \ (x, \ y) \in R) \cap (\lambda x y. \ (x, \ y) \in S) = (\lambda x y. \ (x, \ y) \in R \cap S) \)
  by (simp add: inf-fun-def)

lemma sup-Un-eq [pred-set-conv]: \( (\lambda x. \ x \in R) \cup (\lambda x. \ x \in S) = (\lambda x. \ x \in R \cup S) \)
  by (simp add: sup-fun-def)

lemma sup-Un-eq2 [pred-set-conv]: \( (\lambda x y. \ (x, \ y) \in R) \cup (\lambda x y. \ (x, \ y) \in S) = (\lambda x y. \ (x, \ y) \in R \cup S) \)
  by (simp add: sup-fun-def)

lemma INF-INT-eq [pred-set-conv]: \( (\prod i \in S. \ (\lambda x. \ x \in r \ i)) = (\lambda x. \ x \in (\prod i \in S. r \ i)) \)
  by (simp add: fun-eq-iff)

lemma INF-INT-eq2 [pred-set-conv]: \( (\prod i \in S. \ (\lambda x y. \ (x, \ y) \in r \ i)) = (\lambda x y. \ (x, \ y) \in (\prod i \in S. r \ i)) \)
  by (simp add: fun-eq-iff)

lemma SUP-UN-eq [pred-set-conv]: \( (\bigcup i \in S. \ (\lambda x. \ x \in r \ i)) = (\lambda x. \ x \in (\bigcup i \in S. r \ i)) \)
  by (simp add: fun-eq-iff)

lemma SUP-UN-eq2 [pred-set-conv]: \( (\bigcup i \in S. \ (\lambda x y. \ (x, \ y) \in r \ i)) = (\lambda x y. \ (x, \ y) \in (\bigcup i \in S. r \ i)) \)
  by (simp add: fun-eq-iff)

lemma Inf-INT-eq [pred-set-conv]: \( \prod S = (\lambda x. \ x \in (\bigcap (\text{Collect} \cdot S))) \)
  by (simp add: fun-eq-iff)

lemma INF-Int-eq [pred-set-conv]: \( (\prod i \in S. \ (\lambda x. \ x \in i)) = (\lambda x. \ x \in \bigcap S) \)
  by (simp add: fun-eq-iff)

lemma Inf-INT-eq2 [pred-set-conv]: \( \prod S = (\lambda x y. \ (x, \ y) \in (\prod (\text{Collect} \cdot \text{case-prod} \cdot S))) \)
theory "Relation"

by (simp add: fun-eq-iff)

lemma INF-Int-eq2 [pred-set-conv]: \( \bigcap i \in S. (\lambda x y. (x, y) \in i) \) = (\( \lambda x y. (x, y) \in \bigcap S \))
  by (simp add: fun-eq-iff)

lemma Sup-SUP-eq [pred-set-conv]: \( \bigcup S = (\lambda x. x \in \bigcup (\text{Collect } \cdot S)) \)
  by (simp add: fun-eq-iff)

lemma SUP-Sup-eq [pred-set-conv]: (\( \bigcup i \in S. (\lambda x. x \in i) \)) = (\( \lambda x y. (x, y) \in \bigcup S \))
  by (simp add: fun-eq-iff)

lemma Sup-SUP-eq2 [pred-set-conv]: \( \bigcup S = (\lambda x y. (x, y) \in (\bigcup (\text{Collect } \cdot \text{case-prod } \cdot S))) \)
  by (simp add: fun-eq-iff)

lemma SUP-Sup-eq2 [pred-set-conv]: (\( \bigcup i \in S. (\lambda x y. (x, y) \in i) \)) = (\( \lambda x y. (x, y) \in \bigcup S \))
  by (simp add: fun-eq-iff)

18.2 Properties of relations

18.2.1 Reflexivity

definition refl-on :: \( 'a \) set \( \Rightarrow \) 'a rel \( \Rightarrow \) bool
  where refl-on A r \( \iff \) r \( \subseteq \) A \times A \( \land \) (\( \forall x \in A. (x, x) \in r \))

abbreviation refl :: \( 'a \) rel \( \Rightarrow \) bool — reflexivity over a type
  where refl \( \equiv \) refl-on UNIV

definition reflp :: \( 'a \Rightarrow \) \( 'a \Rightarrow \) bool \( \Rightarrow \) bool
  where reflp r \( \iff \) (\( \forall x. r \ x \ x \))

lemma reflp-refl-eq [pred-set-conv]: reflp (\( \lambda x y. (x, y) \in r \)) \( \iff \) refl r
  by (simp add: refl-on-def reflp-def)

lemma refl-onI [intro?]: r \( \subseteq \) A \times A \( \Longrightarrow \) (\( \forall x \in A \Rightarrow (x, x) \in r \)) \( \Longrightarrow \) refl-on A r
  unfolding refl-on-def by (iprover intro: ballI)

lemma refl-onD: refl-on A r \( \Longrightarrow \) a \( \in \) A \( \Longrightarrow \) (a, a) \( \in \) r
  unfolding refl-on-def by blast

lemma refl-onD1: refl-on A r \( \Longrightarrow \) (x, y) \( \in \) r \( \Longrightarrow \) x \( \in \) A
  unfolding refl-on-def by blast

lemma refl-onD2: refl-on A r \( \Longrightarrow \) (x, y) \( \in \) r \( \Longrightarrow \) y \( \in \) A
  unfolding refl-on-def by blast

lemma reflpI [intro?]: (\( \forall x. r \ x \ x \)) \( \Longrightarrow \) reflp r
by (auto intro: refl-onI simp add: reflp-def)

lemma reflpE:
  assumes reflp r
  obtains r x x
  using assms by (auto dest: refl-onD simp add: reflp-def)

lemma reflpD [dest?!]:
  assumes reflp r
  shows r x x
  using assms by (auto elim: reflpE)

lemma refl-on-Int: refl-on A r \Rightarrow refl-on B s \Rightarrow refl-on (A \cap B) (r \cap s)
  unfolding refl-on-def by blast

lemma reflp-inf: reflp r \Rightarrow reflp s \Rightarrow reflp (r \cap s)
  by (auto intro: reflpI elim: reflpE)

lemma refl-on-Un: refl-on A r \Rightarrow refl-on B s \Rightarrow refl-on (A \cup B) (r \cup s)
  unfolding refl-on-def by blast

lemma reflp-sup: reflp r \Rightarrow reflp s \Rightarrow reflp (r \cup s)
  by (auto intro: reflpI elim: reflpE)

lemma refl-on-INTER: \forall x \in S. refl-on (A x) (r x) \Rightarrow refl-on (\bigcap ( A ' S)) (\bigcap (r ' S))
  unfolding refl-on-def by fast

lemma refl-on-UNION: \forall x \in S. refl-on (A x) (r x) \Rightarrow refl-on (\bigcup ( A ' S)) (\bigcup (r ' S))
  unfolding refl-on-def by blast

lemma refl-on-empty [simp]: refl-on {} {}
  by (simp add: refl-on-def)

lemma refl-on-singleton [simp]: refl-on {x} {{x, x}}
  by (blast intro: refl-onI)

lemma refl-on-def': [nitpick-unfold, code]:
  refl-on A r \iff (\forall (x, y) \in r. x \in A \land y \in A) \land (\forall x \in A. (x, x) \in r)
  by (auto intro: refl-onI dest: refl-onD refl-onD1 refl-onD2)

lemma reflp-equality [simp]: reflp (=)
  by (simp add: reflp-def)

lemma reflp-mono: reflp R \Rightarrow (\land x y. R x y \Rightarrow Q x y) \Rightarrow reflp Q
  by (auto intro: reflpI dest: reflpD)
18.2.2 Irreflexivity

definition irrefl :: 'a rel ⇒ bool
  where irrefl r ←→ (∀ a. (a, a) /∈ r)

definition irreflp :: ('a ⇒ 'a ⇒ bool) ⇒ bool
  where irreflp R ←→ (∀ a. ¬ R a a)

lemma irreflp-irrefl-eq [pred-set-conv]: irreflp (λ a b. (a, b) ∈ R) ←→ irrefl R
  by (simp add: irrefl-def irreflp-def)

lemma irreflI [intro?]: (∀ a b. (a, b) /∈ R) =⇒ irrefl R
  by (simp add: irrefl-def)

lemma irreflpI [intro?]: (∀ a. ¬ R a a) =⇒ irreflp R
  by (fact irreflI [to-pred])

lemma irrefl-distinct [code]: irrefl r ←→ (∀ (a, b). a ≠ b)
  by (auto simp add: irrefl-def)

18.2.3 Asymmetry

inductive asym :: 'a rel ⇒ bool
  where asymI: irrefl R =⇒ (∀ a b. (a, b) ∈ R =⇒ (b, a) /∈ R) =⇒ asym R

inductive asymp :: ('a ⇒ 'a ⇒ bool) ⇒ bool
  where asympI: irreflp R =⇒ (∀ a b. R a b =⇒ ¬ R b a) =⇒ asymp R

lemma asymp-asym-eq [pred-set-conv]: asymp (λ a b. (a, b) ∈ R) ←→ asym R
  by (auto intro!: asympI elim: asymI asympI cases simp add: irreflp-irrefl-eq)

18.2.4 Symmetry

definition sym :: 'a rel ⇒ bool
  where sym r ←→ (∀ x y. (x, y) ∈ r =⇒ (y, x) ∈ r)

definition symp :: ('a ⇒ 'a ⇒ bool) ⇒ bool
  where symp r ←→ (∀ x y. x y y =⇒ r y x)

lemma symp-sym-eq [pred-set-conv]: symp (λ x y. (x, y) ∈ r) ←→ sym r
  by (simp add: sym-def symp-def)

lemma symI [intro?]: (∀ a b. (a, b) ∈ r =⇒ (b, a) ∈ r) =⇒ sym r
  by (unfold sym-def) iprover

lemma sympI [intro?]: (∀ a b. r a b =⇒ r b a) =⇒ symp r
  by (fact symI [to-pred])

lemma symE:
  assumes sym r and (b, a) ∈ r
obtains \((a, b) \in r\) using assms by (simp add: sym-def)

lemma sympE:
  assumes symp \(r\) and \(r b a\)
  obtains \(r a b\) using assms by (rule symE [to-pred])

lemma sympD [dest?):
  assumes symp \(r\) and \((b, a) \in r\)
  shows \((a, b) \in r\) using assms by (rule symE)

lemma symp-Int:
  symp \(r\) = symp \(s\) =⇒ symp \((r \cap s)\)
by (fast intro: sympI elim: symE)

lemma symp-inf:
  symp \(r\) = symp \(s\) =⇒ symp \((r \cap s)\)
by (fact symp-Int [to-pred])

lemma symp-Un:
  symp \(r\) = symp \(s\) =⇒ symp \((r \cup s)\)
by (fast intro: sympI elim: symE)

lemma symp-sup:
  symp \(r\) = symp \(s\) =⇒ symp \((r \cup s)\)
by (fact symp-Un [to-pred])

lemma symp-INTER:
  \(\forall x \in S.\) symp \((r x)\) =⇒ symp \((\bigcap (r \cdot S))\)
by (fast intro: symI elim: symE)

lemma symp-INF:
  \(\forall x \in S.\) symp \((r x)\) =⇒ symp \((\bigcap (r \cdot S))\)
by (fact symp-INTER [to-pred])

lemma symp-UNION:
  \(\forall x \in S.\) symp \((r x)\) =⇒ symp \((\bigcup (r \cdot S))\)
by (fast intro: symI elim: symE)

lemma symp-SUP:
  \(\forall x \in S.\) symp \((r x)\) =⇒ symp \((\bigcup (r \cdot S))\)
by (fact sym-UNION [to-pred])

18.2.5 Antisymmetry

definition antisym :: \('a rel \Rightarrow bool\)
where antisym \(r\) =\(\langle \forall x y. (x, y) \in r \longrightarrow (y, x) \in r \longrightarrow x = y\rangle\)

definition antisymp :: \('(a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool\)
where antisymp \(r\) =\(\langle \forall x y. r \cdot x \cdot y \longrightarrow r \cdot y x \longrightarrow x = y\rangle\)
lemma antisymp-antisym-eq [pred-set-cone]: antisymp $(\lambda x\ y.\ (x,\ y) \in r) \iff \text{antisym } r$
   by (simp add: antisym-def antisymp-def)

lemma antisymI [intro?]:
$(\forall x\ y.\ (x,\ y) \in r \Longrightarrow (y,\ x) \in r \Longrightarrow x = y) \Longrightarrow \text{antisym } r$
unfolding antisym-def by iprover

lemma antisymD [dest?]:
antisym r \Longrightarrow (a, b) \in r \Longrightarrow (b, a) \in r \Longrightarrow a = b
unfolding antisym-def by iprover

lemma antisymD [dest?]:
antisym r \Longrightarrow r a b \Longrightarrow r b a \Longrightarrow a = b
by (fact antisymD [to-pred])

lemma antisym-subset:
$r \subseteq s \Longrightarrow \text{antisym } s \Longrightarrow \text{antisym } r$
unfolding antisym-def by blast

lemma antisym-less-eq:
$r \Longrightarrow \text{antisym } s \Longrightarrow \text{antisym } r$
by (fact antisym-subset [to-pred])

lemma antisym-empty [simp]:
antisym {}
unfolding antisym-def by blast

lemma antisym-bot [simp]:
antisym $\bot$
by (fact antisym-empty [to-pred])

lemma antisym-equivalence [simp]:
antisym $\text{HOL.eq}$
by (auto intro: antisymI)

lemma antisym-singleton [simp]:
antisym $\{x\}$
by (blast intro: antisymI)

18.2.6 Transitivity

definition trans :: `'a rel \Rightarrow bool
where trans r \iff (\forall x\ y\ z.\ (x,\ y) \in r \longrightarrow (y,\ z) \in r \longrightarrow (x,\ z) \in r)
definition transp :: (‘a ⇒ ‘a ⇒ bool) ⇒ bool
  where transp r ←→ (∀ x y z. r x y → r y z → r x z)

lemma transp-trans-eq [pred-set-conv]: transp (λx y. (x, y) ∈ r) ←→ trans r
  by (simp add: trans-def transp-def)

lemma transI [intro?]: (∀ x y z. (x, y) ∈ r → (y, z) ∈ r → (x, z) ∈ r) ⇒ trans r
  by (unfold trans-def) iprover

lemma transpI [intro?]: (∀ x y z. r x y =⇒ r y z =⇒ r x z) =⇒ transp r
  by (fact transI [to-pred])

lemma transE:
  assumes trans r and (x, y) ∈ r and (y, z) ∈ r
  obtains (x, z) ∈ r
  using assms by (unfold trans-def) iprover

lemma transpE:
  assumes transp r and r x y and r y z
  obtains r x z
  using assms by (rule transE [to-pred])

lemma transD [dest?):
  assumes trans r and (x, y) ∈ r and (y, z) ∈ r
  shows (x, z) ∈ r
  using assms by (rule transE)

lemma transpD [dest?):
  assumes transp r and r x y and r y z
  shows r x z
  using assms by (rule transD [to-pred])

lemma trans-Int: trans r =⇒ trans s =⇒ trans (r ∩ s)
  by (fast intro: transI elim: transE)

lemma trans-inf: transp r =⇒ transp s =⇒ transp (r ∩ s)
  by (fact trans-Int [to-pred])

lemma trans-INTER: ∀ x ∈ S. trans (r x) =⇒ trans (⋂ (r ∩ S))
  by (fast intro: transI elim: transD)

lemma trans-INF: ∀ x ∈ S. transp (r x) =⇒ transp (⋂ (r ∩ S))
  by (fact trans-INTER [to-pred])

lemma trans-join [code]: trans r ←→ (∀ (x, y1) ∈ r. ∀ (y2, z) ∈ r. y1 = y2 → (x, z) ∈ r)
  by (auto simp add: trans-def)
lemma transp-trans: \( \text{transp } r \leftrightarrow \text{trans } \{(x, y). r x y\} \)
by (simp add: trans-def transp-def)

lemma transp-equality [simp]: \( \text{transp } (=) \)
by (auto intro: transpI)

lemma trans-empty [simp]: \( \text{trans } \{\} \)
by (blast intro: transI)

lemma transp-empty [simp]: \( \text{transp } (\lambda x y. \text{False}) \)
using trans-empty [to-pred] by (simp add: bot-fun-def)

lemma trans-singleton [simp]: \( \text{trans } \{ (a, a) \} \)
by (blast intro: transI)

lemma transp-singleton [simp]: \( \text{transp } (\lambda x y. x = a \land y = a) \)
by (simp add: transp-def)

context preorder
begin

lemma transp-le [simp]: \( \text{transp } (\leq) \)
by (auto simp add: transp-def intro: order-trans)

lemma transp-less [simp]: \( \text{transp } (<) \)
by (auto simp add: transp-def intro: less-trans)

lemma transp-ge [simp]: \( \text{transp } (\geq) \)
by (auto simp add: transp-def intro: order-trans)

lemma transp-gr [simp]: \( \text{transp } (>) \)
by (auto simp add: transp-def intro: less-trans)

end

18.2.7 Totality

definition total-on :: 'a set ⇒ 'a rel ⇒ bool
where total-on \( A \) \( r \) := (\( \forall x \in A. \forall y \in A. x \neq y \rightarrow (x, y) \in r \lor (y, x) \in r \))

lemma total-onI [intro?]:
(\( \forall x y. [x \in A; y \in A; x \neq y] \rightarrow (x, y) \in r \lor (y, x) \in r \)) \Rightarrow total-on \( A \) \( r \)
unfolding total-on-def by blast

abbreviation total ≡ total-on UNIV

lemma total-on-empty [simp]: total-on \( \{\} \) \( r \)
by (simp add: total-on-def)
lemma total-on-singleton [simp]: total-on {x} {(x, x)}
unfolding total-on-def by blast

18.2.8 Single valued relations
definition single-valued :: ('a × 'b) set ⇒ bool
where single-valued r ←→ (∀ x y. (x, y) ∈ r → (∀ z. (x, z) ∈ r → y = z))
definition single-valuedp :: ('a ⇒ 'b ⇒ bool) ⇒ bool
where single-valuedp r ←→ (∀ x y. r x y → (∀ z. r x z → y = z))
lemma single-valuedp-single-valued-eq [pred-set-cone]:
single-valuedp (λx y. (x, y) ∈ r) ←→ single-valued r
by (simp add: single-valued-def single-valuedp-def)
lemma single-valuedI:
(∀ x y. (x, y) ∈ r =⇒ (∀ z. (x, z) ∈ r =⇒ y = z)) =⇒ single-valued r
unfolding single-valued-def by blast
lemma single-valuedpI:
(∀ x y. r x y =⇒ (∀ z. r x z =⇒ y = z)) =⇒ single-valuedp r
by (fact single-valuedI [to-pred])
lemma single-valuedD:
single-valued r =⇒ (x, y) ∈ r =⇒ (x, z) ∈ r =⇒ y = z
by (simp add: single-valued-def)
lemma single-valuedpD:
single-valuedp r =⇒ r x y =⇒ r x z =⇒ y = z
by (fact single-valuedD [to-pred])
lemma single-valued-empty [simp]:
single-valued {}
by (simp add: single-valued-def)
lemma single-valuedp-bot [simp]:
single-valuedp ⊥
by (fact single-valued-empty [to-pred])
lemma single-valued-subset:
r ⊆ s =⇒ single-valued s =⇒ single-valued r
unfolding single-valued-def by blast
lemma single-valuedp-less-eq:
r ≤ s =⇒ single-valuedp s =⇒ single-valuedp r
by (fact single-valued-subset [to-pred])
18.3 Relation operations

18.3.1 The identity relation

definition Id :: 'a rel
  where Id = \{p. \exists x. p = (x, x)\}

lemma IdI [intro]: (a, a) \in Id
  by (simp add: Id-def)

lemma IdE [elim!]: p \in Id \Longrightarrow (\\forall x. p = (x, x) \Longrightarrow P) \Longrightarrow P
  unfolding Id-def by (iprover elim: CollectE)

lemma pair-in-Id-conv [iff]: (a, b) \in Id \longleftrightarrow a = b
  unfolding Id-def by blast

lemma refl-Id: refl Id
  by (simp add: refl-on-def)

lemma antisym-Id: antisym Id
  -- A strange result, since Id is also symmetric.
  by (simp add: antisym-def)

lemma sym-Id: sym Id
  by (simp add: sym-def)

lemma trans-Id: trans Id
  by (simp add: trans-def)

lemma single-valued-Id [simp]: single-valued Id
  by (unfold single-valued-Id) blast

lemma irrefl-diff-Id [simp]: irrefl (r - Id)
  by (simp add: irrefl-def)

lemma trans-diff-Id: trans r \Longrightarrow antisym r \Longrightarrow trans (r - Id)
  unfolding antisym-def trans-def by blast

lemma total-on-diff-Id [simp]: total-on A (r - Id) = total-on A r
  by (simp add: total-on-def)

lemma Id-fstsnd-eq: Id = \{x. fst x = snd x\}
  by force

18.3.2 Diagonal: identity over a set

definition Id-on :: 'a set \Rightarrow 'a rel
  where Id-on A = (\{x\in A. \{(x, x)\}\})

lemma Id-on-empty [simp]: Id-on {} = {}
lemma \textit{Id-on-eqI}: \(a = b \implies a \in A \implies (a, b) \in \text{Id-on } A\)
by (\textit{simp add: Id-on-def})

lemma \textit{Id-onI} [intro!]: \(a \in A \implies (a, a) \in \text{Id-on } A\)
by (rule \textit{Id-on-eqI}) (rule \textit{refl})

lemma \textit{Id-onE} [elim!]: \(c \in \text{Id-on } A \implies (\forall x \in A \implies c = (x, x) \implies P) \implies P\)
— The general elimination rule.
unfolding \textit{Id-on-def} by (\textit{iprover elim: UN-E singletonE})

lemma \textit{Id-on-iff}: \((x, y) \in \text{Id-on } A \iff x = y \land x \in A\)
by blast

lemma \textit{Id-on-def'} [nitpick-unfold]: \(\text{Id-on } \{x. A x\} = \text{Collect } (\lambda (x, y). x = y \land A x)\)
by auto

lemma \textit{Id-on-subset-Times}: \(\text{Id-on } A \subseteq A \times A\)
by blast

lemma \textit{refl-on-Id-on}: \(\text{refl-on } A (\text{Id-on } A)\)
by (rule \textit{refl-onI} [OF \textit{Id-on-subset-Times} \textit{Id-onI}])

lemma \textit{antisym-Id-on} [simp]: \(\text{antisym } (\text{Id-on } A)\)
unfolding \textit{antisym-def} by blast

lemma \textit{sym-Id-on} [simp]: \(\text{sym } (\text{Id-on } A)\)
by (rule \textit{symI}) clarify

lemma \textit{trans-Id-on} [simp]: \(\text{trans } (\text{Id-on } A)\)
by (fast intro: \textit{transI} elim: \textit{transD})

lemma \textit{single-valued-Id-on} [simp]: \(\text{single-valued } (\text{Id-on } A)\)
unfolding \textit{single-valued-def} by blast

18.3.3 Composition

\textbf{inductive-set} \textit{relcomp} :: \(('a \times 'b) \text{ set } \Rightarrow ('b \times 'c) \text{ set } \Rightarrow ('a \times 'c) \text{ set}\) \textbf{(infixr} \textit{O 75})
\textbf{for} \textit{r} :: \(('a \times 'b) \text{ set } \text{ and } s :: ('b \times 'c) \text{ set}\)
\textbf{where} \textit{relcompI} [intro]: \((a, b) \in r \implies (b, c) \in s \implies (a, c) \in r \textit{ O } s\)

\textbf{notation} \textit{relcompp} \textbf{(infixr} \textit{OO 75})

\textbf{lemmas} \textit{relcomppI = relcompp.intro}\n
For historic reasons, the elimination rules are not wholly corresponding. Feel
free to consolidate this.

**inductive-cases** `relcompEpair`: $(a, c) \in r \circ s$

**inductive-cases** `relcomppE [elim!]`: $(r \circ\circ s) a c$

**lemma** `relcomp [elim!]:` $xz \in r \circ s \Rightarrow \left( \forall x y z. \, (x, z) \Rightarrow (x, y) \in r \Rightarrow (y, z) \in s \Rightarrow P \right) \Rightarrow P$

apply (cases $xz$)
apply simp
apply (erule `relcompEpair`)
done

**lemma** `R-O-Id [simp]:` $R \circ Id = R$
by fast

**lemma** `Id-O-R [simp]:` $Id \circ R = R$
by fast

**lemma** `relcomp-empty1 [simp]:` `{}` $\circ R = {}$
by blast

**lemma** `relcompp-bot1 [simp]:` $\bot \circ\circ R = \bot$
by (fact `relcomp-empty1 [to-pred]`)

**lemma** `relcomp-empty2 [simp]:` $R \circ {} = {}$
by blast

**lemma** `relcompp-bot2 [simp]:` $R \circ\circ \bot = \bot$
by (fact `relcomp-empty2 [to-pred]`)

**lemma** `O-assoc: (R \circ S) \circ T = R \circ (S \circ T)`
by blast

**lemma** `relcompp-assoc: (r \circ\circ s) \circ\circ t = r \circ\circ (s \circ\circ t)`
by (fact `O-assoc [to-pred]`)

**lemma** `trans-O-subset: trans r \Rightarrow r \circ s \subseteq r`
by (unfold trans-def) blast

**lemma** `transp-relcompp-less-eq: transp r \Rightarrow r \circ\circ s \leq r`
by (fact `trans-O-subset [to-pred]`)

**lemma** `relcomp-mono: r' \subseteq r \Rightarrow s' \subseteq s \Rightarrow r' \circ\circ s' \subseteq r \circ\circ s$
by blast

**lemma** `relcompp-mono: r' \leq r \Rightarrow s' \leq s \Rightarrow r' \circ\circ s' \leq r \circ\circ s$
by (fact `relcomp-mono [to-pred]`)

**lemma** `relcomp-subset-Sigma: r \subseteq A \times B \Rightarrow s \subseteq B \times C \Rightarrow r \circ s \subseteq A \times C`
THEORY "Relation" by blast

lemma relcomp-distrib [simp]: \( R O (S \cup T) = (R O S) \cup (R O T) \)
  by auto

lemma relcomp-distrib [simp]: \( R OO (S \cup T) = R OO S \cup R OO T \)
  by (fact relcomp-distrib [to-pred])

lemma relcomp-distrib2 [simp]: \((S \cup T) O R = (S O R) \cup (T O R)\)
  by auto

lemma relcompp-distrib2 [simp]: \((S \sqcup T) OO R = S OO R \sqcup T OO R\)
  by (fact relcomp-distrib2 [to-pred])

lemma relcomp-UNION-distrib: \( s O \bigcup (r ^ I) = \bigcup i \in I. s O r i \)
  by auto

lemma relcompp-SUP-distrib: \( s OO \bigvee (r ^ I) = \bigvee i \in I. s OO r i \)
  by (fact relcomp-UNION-distrib [to-pred])

lemma relcomp-UNION-distrib2: \( \bigcup (r ^ I) O s = \bigcup i \in I. r i O s \)
  by auto

lemma relcompp-SUP-distrib2: \( \bigvee (r ^ I) OO s = \bigvee i \in I. r i OO s \)
  by (fact relcomp-UNION-distrib2 [to-pred])

lemma single-valued-relcomp: \( single-valued r \Rightarrow single-valued s \Rightarrow single-valued (r O s) \)
  unfolding single-valued-def by blast

lemma relcomp-unfold: \( r O s = \{ (x, z). \exists y. (x, y) \in r \land (y, z) \in s \} \)
  by (auto simp add: set-eq-iff)

lemma relcompp-apply: \( (R OO S) a c \longleftrightarrow (\exists b. R a b \land S b c) \)
  unfolding relcomp-unfold [to-pred] ..

lemma eq-OO: \( (=) OO R = R \)
  by blast

lemma OO-eq: \( R OO (=) = R \)
  by blast

18.3.4 Converse

inductive-set converse :: \( ('a \times 'b) set \Rightarrow ('b \times 'a) set \)
  ((\sim ^{-1}) [1000] 999)
for \( r :: ('a \times 'b) set \)
  where \( (a, b) \in r \Rightarrow (b, a) \in r ^{-1} \)
notation conversep ((\sim ^{-1}) [1000] 1000)
notation (ASCII)
  converse  ((^−1) [1000] 999) and
  conversep (((−−1) [1000] 1000)

lemma converseI [sym]: (a, b) ∈ r ⇒ (b, a) ∈ r⁻¹
  by (fact converse.intros)

lemma conversepI: r a b ⇒ r⁻¹⁻¹ b a
  by (fact conversep.intros)

lemma converseD [sym]: (a, b) ∈ r⁻¹ ⇒ (b, a) ∈ r
  by (erule converse.cases) iprover

lemma conversepD : r⁻¹⁻¹ b a ⇒ r a b
  by (fact converseD [to-pred])

lemma converseE [elim!]: yx ∈ r⁻¹ ⇒ (∀x y. yx = (y, x) ⇒ (x, y) ∈ r ⇒ P) ⇒ P
  — More general than converseD, as it “splits” the member of the relation.
  apply (cases yx)
  apply simp
  apply (erule converse.cases)
  apply iprover
  done

lemmas conversepE [elim!] = conversep.cases

lemma converse-iff [iff]: (a, b) ∈ r⁻¹ ⇐⇒ (b, a) ∈ r
  by (auto intro: converseI)

lemma conversep-iff [iff]: r⁻¹⁻¹ a b = r b a
  by (fact converse-iff [to-pred])

lemma converse-converse [simp]: (r⁻¹)⁻¹ = r
  by (simp add: set-eq-iff)

lemma conversep-conversep [simp]: (r⁻¹⁻¹)⁻¹⁻¹ = r
  by (fact converse-converse [to-pred])

lemma converse-empty [simp]: {}⁻¹ = {}
  by auto

lemma converse-UNIV [simp]: UNIV⁻¹ = UNIV
  by auto

lemma converse-relcomp: (r O s)⁻¹ = s⁻¹ O r⁻¹
  by blast
lemma converse-relcompp: \((r \ O O s)^{-1} = s^{-1} \ O O r^{-1}\)
  by (iprover intro: order-antisym conversepI relcomppI elim: relcomppE dest: conversepD)

lemma converse-Int: \((r \cap s)^{-1} = r^{-1} \cap s^{-1}\)
  by blast

lemma converse-meet: \((r \cap s)^{-1}^{-1} = r^{-1} \cap s^{-1}^{-1}\)
  by (simp add: inf-fun-def) (iprover intro: conversepI ext dest: conversepD)

lemma converse-Un: \((r \cup s)^{-1} = r^{-1} \cup s^{-1}\)
  by blast

lemma converse-join: \((r \cup s)^{-1}^{-1} = r^{-1} \cup s^{-1}^{-1}\)
  by (simp add: sup-fun-def) (iprover intro: conversepI ext dest: conversepD)

lemma converse-INTER: \((\bigcap (r \cdot S))^{-1} = (\bigcap x \in S. (r x)^{-1})\)
  by fast

lemma converse-UNION: \((\bigcup (r \cdot S))^{-1} = (\bigcup x \in S. (r x)^{-1})\)
  by blast

lemma converse-mono[simp]: \(r^{-1} \subseteq s^{-1} \iff r \subseteq s\)
  by auto

lemma conversep-mono[simp]: \(r^{-1}^{-1} \leq s^{-1}^{-1} \iff r \leq s\)
  by (fact converse-mono[to-pred])

lemma converse-inject[simp]: \(r^{-1} = s^{-1} \iff r = s\)
  by auto

lemma conversep-inject[simp]: \(r^{-1}^{-1} = s^{-1}^{-1} \iff r = s\)
  by (fact converse-inject[to-pred])

lemma converse-subset-swap: \(r \subseteq s^{-1} \iff r^{-1} \subseteq s\)
  by auto

lemma conversep-le-swap: \(r \leq s^{-1} \iff r^{-1} \leq s\)
  by (fact converse-subset-swap[to-pred])

lemma converse-Id [simp]: \(Id^{-1} = Id\)
  by blast

lemma converse-Id-on [simp]: \((Id-on A)^{-1} = Id-on A\)
  by blast

lemma refl-on-converse [simp]: \(refl-on A (converse r) = refl-on A r\)
  by (auto simp: refl-on-def)
lemma sym-converse [simp]: sym (converse r) = sym r
unfolding sym-def by blast

lemma antisym-converse [simp]: antisym (converse r) = antisym r
unfolding antisym-def by blast

lemma trans-converse [simp]: trans (converse r) = trans r
unfolding trans-def by blast

lemma sym-conv-converse-eq: sym r ←→ r⁻¹ = r
unfolding sym-def by fast

lemma sym-Un-converse: sym (r ∪ r⁻¹)
unfolding sym-def by blast

lemma sym-Int-converse: sym (r ∩ r⁻¹)
unfolding sym-def by blast

lemma total-on-converse [simp]: total-on A (r⁻¹) = total-on A r
by (auto simp: total-on-def)

lemma finite-converse [iff]: finite (r⁻¹) = finite r
unfolding converse-def conversep-iff using [[simproc add: finite-Collect]]
by (auto elim: finite-imageD simp: inj-on-def)

lemma card-inverse [simp]: card (R⁻¹) = card R
proof -
  have ∗: ∀R. prod.swap ' R = R⁻¹ by auto
  { assume ¬finite R
    hence ?thesis
    by auto
  }
  moreover {
    assume finite R
    with card-image-le[of R prod.swap] card-image-le[of R⁻¹ prod.swap]
    have ?thesis by (auto simp: ∗)
  }
  ultimately show ?thesis by blast
qed

lemma conversep-noteq [simp]: (≠)⁻¹⁻¹ = (≠)
by (auto simp: fun-eq-iff)

lemma conversep-eq [simp]: (=)⁻¹⁻¹ = (=)
by (auto simp: fun-eq-iff)

lemma converse-unfold [code]: r⁻¹ = {(y, x). (x, y) ∈ r}
by (simp add: set-eq-iff)
18.3.5 Domain, range and field

**inductive-set** `Domain :: ('a × 'b) set ⇒ 'a set for r :: ('a × 'b) set`  
where `DomainI [intro]: (a, b) ∈ r ⇒ a ∈ Domain r`

**lemmas** `DomainPI = Domainp.DominI`

**inductive-cases** `DomainE [elim!]: a ∈ Domain r`

**inductive-cases** `DomainpE [elim!]: Domainp r a`

**inductive-set** `Range :: ('a × 'b) set ⇒ 'b set for r :: ('a × 'b) set`  
where `RangeI [intro]: (a, b) ∈ r ⇒ b ∈ Range r`

**lemmas** `RangePI = Rangep.RangeI`

**inductive-cases** `RangeE [elim!]: b ∈ Range r`

**inductive-cases** `RangepE [elim!]: Rangep r b`

**definition** `Field :: 'a rel ⇒ 'a set`  
where `Field r = Domain r ∪ Range r`

**lemma** `FieldI1: (i, j) ∈ R ⇒ i ∈ Field R`  
**unfolding** `Field-def by blast`

**lemma** `FieldI2: (i, j) ∈ R ⇒ j ∈ Field R`  
**unfolding** `Field-def by auto`

**lemma** `Domain-fst [code]: Domain r = fst ' r`  
**by force**

**lemma** `Range-snd [code]: Range r = snd ' r`  
**by force**

**lemma** `fst-eq-Domain: fst ' R = Domain R`  
**by force**

**lemma** `snd-eq-Range: snd ' R = Range R`  
**by force**

**lemma** `range-fst [simp]: range fst = UNIV`  
**by (auto simp: fst-eq-Domain)**

**lemma** `range-snd [simp]: range snd = UNIV`  
**by (auto simp: snd-eq-Range)**

**lemma** `Domain-empty [simp]: Domain {} = {}`  
**by auto**

**lemma** `Range-empty [simp]: Range {} = {}`  
**by auto**
lemma Field-empty [simp]: Field {} = {}
  by (simp add: Field-def)

lemma Domain-empty-iff: Domain r = {} ⟷ r = {}
  by auto

lemma Range-empty-iff: Range r = {} ⟷ r = {}
  by auto

lemma Domain-insert [simp]: Domain (insert (a, b) r) = insert a (Domain r)
  by blast

lemma Range-insert [simp]: Range (insert (a, b) r) = insert b (Range r)
  by blast

lemma Field-insert [simp]: Field (insert (a, b) r) = {a, b} ∪ Field r
  by (auto simp add: Field-def)

lemma Domain-Id [simp]: Domain Id = UNIV
  by blast

lemma Range-Id [simp]: Range Id = UNIV
  by blast

lemma Domain-Id-on [simp]: Domain (Id-on A) = A
  by blast

lemma Range-Id-on [simp]: Range (Id-on A) = A
  by blast

lemma Domain-Un-eq: Domain (A ∪ B) = Domain A ∪ Domain B
  by blast

lemma Range-Un-eq: Range (A ∪ B) = Range A ∪ Range B
  by blast

lemma Field-Un [simp]: Field (r ∪ s) = Field r ∪ Field s
  by (auto simp: Field-def)

lemma Domain-Int-subset: Domain (A ∩ B) ⊆ Domain A ∩ Domain B
  by blast
lemma Range-Int-subset: Range \((A \cap B)\) \(\subseteq\) Range \(A \cap B\)
by blast

lemma Domain-Diff-subset: Domain \(A - Domain B\) \(\subseteq\) Domain \((A - B)\)
by blast

lemma Range-Diff-subset: Range \(A - Range B\) \(\subseteq\) Range \((A - B)\)
by blast

lemma Domain-Union: Domain \((\bigcup S)\) = \((\bigcup A\in S. Domain A)\)
by blast

lemma Range-Union: Range \((\bigcup S)\) = \((\bigcup A\in S. Range A)\)
by blast

lemma Field-Union [simp]: Field \((\bigcup R)\) = \(\bigcup (Field ' R)\)
by (auto simp: Field-def)

lemma Domain-converse [simp]: Domain \((r^{-1})\) = Range \(r\)
by auto

lemma Range-converse [simp]: Range \((r^{-1})\) = Domain \(r\)
by blast

lemma Field-converse [simp]: Field \((r^{-1})\) = Field \(r\)
by (auto simp: Field-def)

lemma Domain-Collect-case-prod [simp]: Domain \{\((x, y)\). P x y\} = \{x. \exists y. P x y\}
by auto

lemma Range-Collect-case-prod [simp]: Range \{\((x, y)\). P x y\} = \{y. \exists x. P x y\}
by auto

lemma finite-Domain: finite \(r\) \(\implies\) finite \((Domain r)\)
by (induct set: finite) auto

lemma finite-Range: finite \(r\) \(\implies\) finite \((Range r)\)
by (induct set: finite) auto

lemma finite-Field: finite \(r\) \(\implies\) finite \((Field r)\)
by (simp add: Field-def finite-Domain finite-Range)

lemma Domain-mono: \(r \subseteq s\) \(\implies\) Domain \(r\) \(\subseteq\) Domain \(s\)
by blast

lemma Range-mono: \(r \subseteq s\) \(\implies\) Range \(r\) \(\subseteq\) Range \(s\)
by blast
**THEORY “Relation”**

**lemma** mono-Field: \( r \subseteq s \implies \text{Field } r \subseteq \text{Field } s \)

by (auto simp: Field-def Domain-def Range-def)

**lemma** Domain-unfold: Domain \( r = \{ x. \exists y. (x, y) \in r \} \)

by blast

**lemma** Field-square [simp]: Field \((x \times x) = x\)

unfolding Field-def by blast

### 18.3.6 Image of a set under a relation

**definition** Image :: \(('a \times 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a set \Rightarrow 'b set)\)

where \( r \circ s = \{ y. \exists x \in s. (x, y) \in r \} \)

**lemma** Image-iff: \( b \in r \circ A \iff (\exists x \in A. (x, b) \in r) \)

by (simp add: Image-def)

**lemma** Image-singleton: \( r \circ \{ a \} = \{ b. (a, b) \in r \} \)

by (simp add: Image-def)

**lemma** Image-singleton-iff [iff]: \( b \in r \circ \{ a \} \iff (a, b) \in r \)

by (rule Image-iff [THEN trans]) simp

**lemma** ImageI [intro]: \( (a, b) \in r \Rightarrow a \in A \Rightarrow b \in r \circ A \)

unfolding Image-def by blast

**lemma** ImageE [elim!]: \( b \in r \circ A \Rightarrow (\forall x. (x, b) \in r \Rightarrow x \in A \Rightarrow P) \Rightarrow P \)

unfolding Image-def by (iprover elim: CollectE bexE)

**lemma** rev-ImageI: \( a \in A \Rightarrow (a, b) \in r \Rightarrow b \in r \circ A \)

— This version’s more effective when we already have the required \( a \)

by blast

**lemma** Image-empty1 [simp]: \{\} \circ X = {} by auto

**lemma** Image-empty2 [simp]: \( R \circ \{\} = {} \)

by blast

**lemma** Image-Id [simp]: Id \( \circ A = A \)

by blast

**lemma** Image-Id-on [simp]: Id-on A \( \circ B = A \cap B \)

by blast

**lemma** Image-Int-subset: \( r \circ (A \cap B) \subseteq r \circ A \cap r \circ B \)

by blast

**lemma** Image-Int-eq: single-valued (converse R) \( \implies R \circ (A \cap B) = R \circ A \cap R \)
THEORY "Relation" 493

" \{ B \\
by (auto simp: single-valued-def)

lemma Image-Un: R " (A ∪ B) = R " A ∪ R " B 
by blast

lemma Un-Image: (R ∪ S) " A = R " A ∪ S " A 
by blast

lemma Image-subset: r ⊆ A × B ⟹ r"C ⊆ B 
by (iprover intro!: subsetI elim!: ImageE dest!: subsetD SigmaD2)

lemma Image-eq-UN: r"B = (⋃ y ∈ B. r"{y}) 
— NOT suitable for rewriting 
by blast

lemma Image-mono: r' ⊆ r ⟹ A' ⊆ A ⟹ (r"A') ⊆ (r"A) 
by blast

lemma Image-UN: r"(⋃(B ' A)) = (⋃x ∈ A. r"(B x)) 
by blast

lemma UN-Image: (∪ i ∈ I. X i) " S = (∪ i ∈ I. X i " S) 
by auto

lemma Image-INT-subset: (r"(⋂(B ' A))) ⊆ (⋂x ∈ A. r"(B x)) 
by blast

Converse inclusion requires some assumptions

lemma Image-INT-eq: single-valued (r⁻¹) ⟹ A ≠ {} ⟹ r"(⋂(B ' A)) = (∩x ∈ A. r"(B x)) 
apply (rule equalityI)
apply (rule Image-INT-subset)
apply (auto simp add: single-valued-def)
apply blast
done

lemma Image-subset-eq: r"A ⊆ B ⟷ A ⊆ (r⁻¹) " (¬ B) 
by blast

lemma Image-Collect-case-prod [simp]: {{x, y}. P x y} " A = {y. ∃x ∈ A. P x y} 
by auto

lemma Sigma-Image: (SIGMA x:A. B x) " X = (∪ x ∈ X ∩ A. B x) 
by auto

lemma relcomp-Image: (X O Y) " Z = Y " (X " Z) 
by auto
lemma finite-Image[simp]: assumes finite R shows finite (R "\ Vie A
by (rule finite-subset[OF finite-Range[OF assms]]) auto

18.3.7 Inverse image
definition inv-image :: 'b rel ⇒ ('a ⇒ 'b) ⇒ ('a rel
  where inv-image r f = {((x, y), (f x, f y)) ∈ r}
definition inv-imagep :: ('b ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ bool
  where inv-imagep r f = (λx y. r (f x) (f y))
lemma [pred-set-conv]: inv-imagep (λx y. (x, y) ∈ r) f = (λx y. (x, y) ∈ inv-image r f)
  by (simp add: inv-image-def inv-imagep-def)
lemma sym-inv-image: sym r ⇒ sym (inv-image r f)
  unfolding sym-def inv-image-def by blast
lemma trans-inv-image: trans r ⇒ trans (inv-image r f)
  unfolding trans-def inv-image-def by (simp (no-asm)) blast
lemma total-inv-image: [inj f; total r] ⇒ total (inv-image r f)
  unfolding inv-image-def total-on-def by (auto simp: inj-eq)
lemma in-inv-image[simp]: (x, y) ∈ inv-image r f ⟷ (f x, f y) ∈ r
  by (auto simp: inv-image-def)
lemma converse-inv-image[simp]: (inv-image R f)⁻¹ = inv-image (R⁻¹) f
  unfolding inv-image-def converse-unfold by auto
lemma in-inv-imagep [simp]: inv-imagep r f x y = r (f x) (f y)
  by (simp add: inv-imagep-def)

18.3.8 Powerset
definition Powp :: ('a ⇒ bool) ⇒ 'a set ⇒ bool
  where Powp A = (λB. ∀x ∈ B. A x)
lemma Powp-Pow-eq [pred-set-conv]: Powp (λx. x ∈ A) = (λx. x ∈ Pow A)
  by (auto simp add: Powp-def fun-eq-iff)
lemmas Powp-mono [mono] = Pow-mono [to-pred]

18.3.9 Expressing relation operations via Finite-Set.fold
lemma Id-on-fold:
  assumes finite A
  shows Id-on A = Finite-Set.fold (λx. Set.insert (Pair x x)) {} A
proof –
interpret \textit{comp-fun-commute} \( \lambda x. \:\textit{Set.insert} \ (\text{Pair} \ x \ x) \) 
by standard auto
from \textit{assms} show \( ?\text{thesis} \)
  unfolding \( \text{Id-on-def} \) by (induct \( A \)) simp-all
qed

lemma \textit{comp-fun-commute-Image-fold}:
  \textit{comp-fun-commute} \( (\lambda (x,y) \ A. \text{if} \ x \in S \text{ then } \text{Set.insert} \ y \ A \text{ else } A) \)
proof –
  interpret \textit{comp-fun-idem} \textit{Set.insert}
  by \textit{(fact \textit{comp-fun-idem-insert})}
  show \( ?\text{thesis} \)
    by standard \textit{(auto simp: \textit{fun-eq-iff} \textit{comp-fun-commute} split: \textit{prod.split})}
qed

lemma \textit{Image-fold}:
  assumes \textit{finite} \( R \)
  shows \( R 
\text{``} S = \text{Finite-Set.fold} \ (\lambda (x,y) \ A. \text{if} \ x \in S \text{ then } \text{Set.insert} \ y \ A \text{ else } A) \)
\( \{\} \) \( R \)
proof –
  interpret \textit{comp-fun-commute} \( (\lambda (x,y) \ A. \text{if} \ x \in S \text{ then } \text{Set.insert} \ y \ A \text{ else } A) \)
  by \textit{(rule \textit{comp-fun-commute-Image-fold})}
  have \( *: \textstyle \forall \ F. \: \text{Set.insert} \ x \ F 
\text{``} S = (\text{if} \ \text{fst} \ x \in S \text{ then } \text{Set.insert} \ (\text{snd} \ x) \ (F 
\text{``} S) \text{ else } (F 
\text{``} S)) \)
    by \textit{(force intro: \textit{rev-ImageI})}
  show \( ?\text{thesis} \)
    using \textit{assms} by \textit{(induct \( R \))} \textit{(auto simp: \( \ast \))}
qed

lemma \textit{insert-relcomp-union-fold}:
  assumes \textit{finite} \( S \)
  shows \( \{x\} \text{\ O \ S} \cup X = \text{Finite-Set.fold} \ (\lambda (w,z) \ A'. \text{if} \ \text{snd} \ x = w \text{ then } \text{Set.insert} \ (\text{fst} \ x,z) \ A' \text{ else } A') \ X S \)
proof –
  interpret \textit{comp-fun-commute} \( \lambda (w,z) \ A'. \text{if} \ \text{snd} \ x = w \text{ then } \text{Set.insert} \ (\text{fst} \ x,z) \ A' \text{ else } A' \)
  proof –
    interpret \textit{comp-fun-idem} \textit{Set.insert}
    by \textit{(fact \textit{comp-fun-idem-insert})}
    show \textit{comp-fun-commute} \( \lambda (w,z) \ A'. \text{if} \ \text{snd} \ x = w \text{ then } \text{Set.insert} \ (\text{fst} \ x,z) \ A' \text{ else } A' \)
      by standard \textit{(auto simp add: \textit{fun-eq-iff} \textit{split: \textit{prod.split})}
  qed
  have \( *: \{x\} \text{\ O \ S} = \{(x', z). \ x' = \text{fst} \ x \wedge (\text{snd} \ x, z) \in S\} \)
    by \textit{(auto simp: \textit{relcomp-unfold} intro!: \textit{exI})}
  show \( ?\text{thesis} \)
    unfolding \( \ast \) using \textit{(finite \( S \))} by \textit{(induct \( S \))} \textit{(auto split: \textit{prod.split})}
qed
lemma insert-relcomp-fold:
  assumes finite S
  shows Set.insert x R O S = Finite-Set.fold (λ(w,z) A'. if snd x = w then Set.insert (fst x,z) A' else A') (R O S) S
proof
  have Set.insert x R O S = ({x} O S) ∪ (R O S)
    by auto
  then show ?thesis
    by (auto simp: insert-relcomp-union-fold [OF assms])
qed

lemma comp-fun-commute-relcomp-fold:
  assumes finite S
  shows comp-fun-commute (λ(x,y) A. Finite-Set.fold (λ(w,z) A'. if y = w then Set.insert (x,z) A' else A') A S)
proof
  have ∗: ∀a b A.
    Finite-Set.fold (λ(w,z) A'. if b = w then Set.insert (a,z) A' else A') A S = {{(a,b)} O S ∪ A
    by (auto simp: insert-relcomp-union-fold[OF assms] cong: if-cong)
  show ?thesis
    by standard (auto simp: ∗)
qed

lemma relcomp-fold:
  assumes finite R finite S
  shows R O S = Finite-Set.fold
    (λ(x,y) A. Finite-Set.fold (λ(w,z) A'. if y = w then Set.insert (x,z) A' else A') A S) {{}} R
using assms
by (induct R)
(auto simp: comp-fun-commute-fold-insert comp-fun-commute-relcomp-fold insert-relcomp-fold
  cong: if-cong)

end

19 Reflexive and Transitive closure of a relation

theory Transitive-Closure
  imports Relation
  abbrevs "* = * **
  and "+ = + ++
  and "+ = +++
begin

ML-file (~~/src/Provers/trancl.ML)

rtrancl is reflexive/transitive closure, trancl is transitive closure, reflcl is
reflexive closure.

These postfix operators have *maximum priority*, forcing their operands to be atomic.

**context notes** [[inductive-internals]]

**begin**

**inductive-set** \( \text{rtrancl} :: (\'a \times \'a) \text{ set} \Rightarrow (\'a \times \'a) \text{ set} \ ((\sim)^{*} [1000] 999) \)

**for** \( r :: (\'a \times \'a) \text{ set} \)

**where**

- \( \text{rtrancl-refl} \ [\text{intro}, \text{Pure.intro}, \text{simp}] :: (a, a) \in r^{*} \)
- \( \text{rtrancl-into-rtrancl} \ [\text{Pure.intro}] :: (a, b) \in r^{*} \Longrightarrow (b, c) \in r \Longrightarrow (a, c) \in r^{*} \)

**inductive-set** \( \text{trancl} :: (\'a \times \'a) \text{ set} \Rightarrow (\'a \times \'a) \text{ set} \ ((\sim)^{+} [1000] 999) \)

**for** \( r :: (\'a \times \'a) \text{ set} \)

**where**

- \( \text{r-into-trancl} \ [\text{intro, Pure.intro}] :: (a, b) \in r^{=} \Longrightarrow (a, b) \in r^{+} \)
- \( \text{trancl-into-trancl} \ [\text{Pure.intro}] :: (a, b) \in r^{+} \Longrightarrow (b, c) \in r \Longrightarrow (a, c) \in r^{+} \)

**notation**

- \( \text{rtranclp} ((\sim)^{**} [1000] 1000) \text{ and} \)
- \( \text{tranclp} ((\sim)^{++} [1000] 1000) \text{ and} \)

**declare**

- \( \text{rtrancl-def} \ [\text{nitpick-unfold del}] \)
- \( \text{rtranclp-def} \ [\text{nitpick-unfold del}] \)
- \( \text{trancl-def} \ [\text{nitpick-unfold del}] \)
- \( \text{tranclp-def} \ [\text{nitpick-unfold del}] \)

**end**

**abbreviation** \( \text{reflcl} :: (\'a \times \'a) \text{ set} \Rightarrow (\'a \times \'a) \text{ set} \ ((\sim)^{=} [1000] 999) \)

**where** \( r^{=} \equiv r \cup \text{Id} \)

**abbreviation** \( \text{reflclp} :: (\'a \Rightarrow \'a \Rightarrow \text{bool}) \Rightarrow (\'a \Rightarrow \'a \Rightarrow \text{bool}) \ ((\sim)^{=} [1000] 1000) \)

**where** \( r^{=} \equiv \sup r \ (=) \)

**notation** (ASCII)

- \( \text{rtrancl} ((\sim)^{*} [1000] 999) \text{ and} \)
- \( \text{trancl} ((\sim)^{+} [1000] 999) \text{ and} \)
- \( \text{reflcl} ((\sim)^{=} [1000] 999) \text{ and} \)
- \( \text{rtranclp} ((\sim)^{**} [1000] 1000) \text{ and} \)
- \( \text{tranclp} ((\sim)^{++} [1000] 1000) \text{ and} \)
- \( \text{reflclp} ((\sim)^{=} [1000] 1000) \text{ and} \)

**19.1 Reflexive closure**

**lemma** \( \text{refl-reflcl} \ [\text{simp}] :: \text{refl} \ (r^{=}) \)

**by** \( \text{(simp add: refl-on-def)} \)
THEORY “Transitive-Closure”

lemma antisym-refcl[simp]: antisym \( r^\omega \) = antisym \( r \)
by (simp add: antisym-def)

lemma trans-refclI[simp]: trans \( r \) \(\implies\) trans \( r^\omega \)
unfolding trans-def by blast

lemma reflclp-idemp [simp]: \((P^=)^= = P^=\)
by blast

19.2 Reflexive-transitive closure

lemma refl-set-eq [pred-set-conv]: \((\sup x y. (x, y) \in r) (=)\) = \((\lambda x y. (x, y)) \in r \cup Id)\)
by (auto simp: fun-eq-iff)

lemma r-into-rtrancl [intro]: \(\forall p. p \in r \implies p \in r^*\)
— rtrancl of \(r\) contains \(r\)
apply (simp only: split-tupled-all)
apply (erule rtrancl-refl [THEN rtrancl-into-rtrancl])
done

lemma r-into-rtranclp [intro]: \(r x y \implies r^{**} x y\)
— rtrancl of \(r\) contains \(r\)
by (erule rtranclp.rtrancl-refl [THEN rtranclp.rtrancl-into-rtrancl])

lemma rtranclp-mono: \(r \leq s \implies r^{**} \leq s^{**}\)
— monotonicity of rtrancl
apply (rule predicate2I)
apply (erule rtranclp.induct)
apply (rule-tac [2] rtranclp.rtrancl-into-rtrancl, blast+)
done

lemma mono-rtranclp[mono]: \((\forall a b x a b \longrightarrow y a b) \implies x^{**} a b \longrightarrow y^{**} a b\)
using rtranclp-mono[of \(x y\)] by auto

lemmas rtrancl-mono = rtranclp-mono [to-set]

theorem rtranclp-induct [consumes 1, case-names base step, induct set: rtranclp]:
assumes \(a: r^{**} a b\)
and cases: \(P a \wedge y z. r^{**} a y \implies r y z \implies P y \implies P z\)
shows \(P b\)
using \(a\) by (induct \(x \equiv a b\) (rule cases)+

lemmas rtrancl-induct [induct set: rtrancl] = rtranclp-induct [to-set]

lemmas rtranclp-induct2 =
rtranclp-induct[of \(- (ax, ay) \ wfr (bx, by)\), split-rule, consumes 1, case-names refl step]

lemmas rtrancl-induct2 =
lemma refl-rtrancl: refl (r*) unfolding refl-on-def by fast

Transitivity of transitive closure.

lemma trans-rtrancl: trans (r*)
proof (rule transI)
  fix x y z
  assume (x, y) ∈ r*
  assume (y, z) ∈ r*
  then show (x, z) ∈ r*
proof induct
  case base
  show (x, y) ∈ r* by fact
next
  case (step u v)
  from ⟨(x, u) ∈ r*⟩ and ⟨(u, v) ∈ r⟩
  show (x, v) ∈ r* ..
qed

lemmas rtrancl-trans = trans-rtrancl [THEN transD]

lemma rtranclp-trans:
  assumes r** x y
  and r** y z
  shows r** x z
using assms(2,1) by induct iprover+

lemma rtranclE [cases set: rtrancl]:
  fixes a b :: 'a
  assumes major: (a, b) ∈ r*
  obtains
    (base) a = b
  | (step) y where (a, y) ∈ r* and (y, b) ∈ r
  — elimination of rtrancl — by induction on a special formula
proof —
  have a = b ∨ (∃ y. (a, y) ∈ r* ∧ (y, b) ∈ r)
  by (rule major [THEN rtrancl-induct]) blast+
then show ?thesis
  by (auto intro: base step)
qed

lemma rtrancl-Int-subset: Id ⊆ s ≡ (r* ∩ s) O r ⊆ s ≡ r* ⊆ s
apply clarify
apply (erule rtrancl-induct, auto)
done
THEORY “Transitive-Closure”

**Lemma** converse-rtranclp-into-rtranclp: \( r \ a \ b \rightarrow r^{**} \ b \ c \rightarrow r^{**} \ a \ c \)
by (rule rtranclp-trans) 

**Lemmas** converse-rtrancl-into-rtrancl = converse-rtranclp-into-rtranclp [to-set]

More \( r^{*} \) equations and inclusions.

**Lemma** rtrancl-idemp [simp]: \((r^{**})^{**} = r^{**}\)
apply (auto intro: order-antisym)
apply (erule rtranclp-induct)
apply (rule rtranclp.rtrancl-refl)
apply (blast intro: rtranclp-trans)
done

**Lemmas** rtrancl-idemp [simp] = rtranclp-idemp [to-set]

**Lemma** rtrancl-idemp-self-comp [simp]: \( R^{*} \ O \ R^{*} = R^{*} \)
apply (rule set-eqI)
apply (simp only: split-tupled-all)
apply (blast intro: rtrancl-trans)
done

**Lemma** rtrancl-subset-rtrancl: \( r \subseteq s^{*} \Rightarrow r^{*} \subseteq s^{*} \)
by (drule rtrancl-mono, simp)

**Lemma** rtrancl-subset: \( R \leq S \Rightarrow S \leq R^{**} \Rightarrow S^{**} = R^{**} \)
apply (drule rtranclp-mono)
apply (drule rtranclp-mono, simp)
done

**Lemmas** rtrancl-subset = rtranclp-subset [to-set]

**Lemma** rtranclp-sup-rtranclp: \((\sup (R^{**}) (S^{**}))^{**} = (\sup R S)^{**}\)
by (blast intro!: rtranclp-subset intro: rtranclp-mono [THEN predicate2D])

**Lemmas** rtrancl-Un-rtrancl = rtranclp-sup-rtranclp [to-set]

**Lemma** rtranclp-reflclp [simp]: \((R^{=})^{**} = R^{**}\)
by (blast intro!: rtranclp-subset)

**Lemmas** rtrancl-refl [simp] = rtranclp-reflclp [to-set]

**Lemma** rtrancl-r-diff-Id: \((r - Id)^{*} = r^{*}\)
by (rule rtrancl-subset [symmetric]) auto

**Lemma** rtranclp-r-diff-Id: \((\inf r (\#))^{**} = r^{**}\)
by (rule rtranclp-subset [symmetric]) auto

**Theorem** rtranclp-converseD:
assumes \((r^{-1})^{**} x y\)
shows \(r^{**} y x\)
using assms by induct (iprover intro: rtranclp-trans dest!: conversepD)+

lemmas rtrancl-converseD = rtranclp-converseD [to-set]

theorem rtranclp-converseI:
assumes \(r^{**} y x\)
shows \((r^{-1})^{**} x y\)
using assms by induct (iprover intro: rtranclp-trans conversepI)+

lemmas rtrancl-converseI = rtranclp-converseI [to-set]

lemma rtrancl-converse: \((r^{-1})^* = (r^*)^{-1}\)
by (fast dest!: rtrancl-converseD intro!: rtrancl-converseI)

lemma sym-rtrancl: sym \(r\) \(\implies\) sym \((r^*)\)
by (simp only: sym-conv-converse-eq rtrancl-converse [symmetric])

theorem converse-rtranclp-induct [consumes 1, case-names base step]:
assumes major: \(r^{**} a b\)
and cases: \(P b \land y z \implies r^{**} z b \implies P z \implies P y\)
shows \(P a\)
using rtranclp-converseI [OF major]
by induct (iprover intro: cases dest!: conversepD rtranclp-converseD)+

lemmas converse-rtranclp-induct = converse-rtranclp-induct [to-set]

lemmas converse-rtranclp-induct2 =
converse-rtranclp-induct [of \((ax, ay)\) \((bx, by)\), split-rule, consumes 1, case-names refl step]

lemmas converse-rtranclp-induct2 =
converse-rtrancl-induct [of \((ax, ay)\) \((bx, by)\), split-format (complete),
consumes 1, case-names refl step]

lemma converse-rtranclpE [consumes 1, case-names base step]:
assumes major: \(r^{**} x z\)
and cases: \(x = z \implies P \land y. r x y \implies r^{**} y z \implies P\)
shows \(P\)
proof –
have \(x = z \lor (\exists y. r x y \land r^{**} y z)\)
  by (rule_tac major [THEN converse-rtranclp-induct]) iprover+
then show ?thesis
  by (auto intro: cases)
qed

lemmas converse-rtranclE = converse-rtranclpE [to-set]
lemmas converse-rtranclE2 = converse-rtranclE [of (xa,xb) (za,zb), split-rule]

lemmas converse-rtranclE2 = converse-rtranclE [of (xa,xb) (za,zb), split-rule]

lemma r-comp-rtrancl-eq: r O r* = r* O r
  by (blast elim: rtranclE converse-rtranclE
       intro: rtrancl-into-rtrancl converse-rtrancl-into-rtrancl)

lemma rtrancl-unfold: r* = Id ∪ r* O r
  by (auto intro: rtrancl-into-rtrancl elim: rtranclE)

lemma rtrancl-Un-separatorE:
  (a, b) ∈ (P ∪ Q)* =⇒ ∀ x y. (a, x) ∈ P* → (x, y) ∈ Q → x = y =⇒ (a, b)
proof (induct rule: rtrancl.induct)
  case rtrancl-refl
  then show ?case by blast
next
  case rtrancl-into-rtrancl
  then show ?case by (blast intro: rtrancl-trans)
qed

lemma rtrancl-Un-separator-converseE:
  (a, b) ∈ (P ∪ Q)* =⇒ ∀ x y. (x, b) ∈ P* → (y, x) ∈ Q → y = x =⇒ (a, b)
proof (induct rule: converse-rtrancl-induct)
  case base
  then show ?case by blast
next
  case step
  then show ?case by (blast intro: rtrancl-trans)
qed

lemma Image-closed-trancl:
  assumes r " X ⊆ X
  shows r* " X = X
proof
  from asms have **: {y. ∃ x∈X. (x, y) ∈ r} ⊆ X
    by auto
  have x ∈ X if 1: (y, x) ∈ r* and 2: y ∈ X for x y
proof
  from 1 show x ∈ X
  proof induct
    case base
    show ?case by (fact 2)
  next
    case step
    with ** show ?case by auto
  qed
qed
  then showthesis by auto
qed

19.3 Transitive closure

lemma trancl-mono: \( \forall p. p \in r^+ \implies r \subseteq s \implies p \in s^+ \)
  apply (simp add: split-tupled-all)
  apply (erule trancl.induct)
  apply (iprover dest: subsetD)+
  done

lemma r-into-trancl': \( \forall p. p \in r \implies p \in r^+ \)
  by (simp only: split-tupled-all) (erule r-into-trancl)

Conversions between trancl and rtrancl.

lemma tranclp-into-rtranclp: \( r^+ a b \implies r^{**} a b \)
  by (erule tranclp.induct) iprover+

lemmas trancl-into-rtrancl = tranclp-into-rtranclp [to-set]

lemma rtranclp-into-tranclp1:
  assumes \( r^{**} a b \)
  shows \( r b c \implies r^+ a c \)
  using assms by (induct arbitrary: c) iprover+

lemmas rtranclp-into-tranclp2 = rtranclp-into-tranclp1 [to-set]

lemma rtranclp-into-tranclp2: \( r a b \implies r^{**} b c \implies r^+ a c \)
  — intro rule from \( r \) and rtrancl
  apply (erule rtranclp.cases, iprover)
  apply (rule rtranclp-trans [THEN rtranclp-into-tranclp1])
  apply (simp | rule r-into-rtranclp)+
  done

lemmas rtranclp-into-tranclp2 = rtranclp-into-tranclp2 [to-set]

Nice induction rule for trancl

lemma tranclp-induct [consumes 1, case-names base step, induct pred: tranclp]:
  assumes \( a: r^+ a b \)
  and cases: \( \forall y. r a y \implies P y \) \( \forall y z. r^{++} a y \implies r y z \implies P y \implies P z \)
  shows \( P b \)
  using a by (induct x\equiv a b) (iprover intro: cases)+

lemmas tranclp-induct [induct set: trancl] = tranclp-induct [to-set]

lemmas tranclp-induct2 =
  tranclp-induct [of - (ax, ay) (bx, by), split-rule, consumes 1, case-names base step]
lemmas trancl-induct2 =
trancl-induct [of \((ax, ay) (bx, by)\), split-format (complete),
consumes 1, case-names base step]

lemma tranclp-trans-induct:
assumes major: \(r^+ \ x \ y\)
and cases: \(\forall x, y. r \ x \ y \Rightarrow P \ x \ y \ \forall x, y, z. r^+ \ x \ y \Rightarrow P \ x \ y \Rightarrow r^+ \ y \ z \Rightarrow\)
P y z \Rightarrow P x z
shows P x y
— Another induction rule for trancl, incorporating transitivity
by (iprover intro: major [THEN tranclp-induct] cases)

lemmas trancl-trans-induct = tranclp-trans-induct [to-set]

lemma tranclE [cases set: trancl]:
assumes \((a, b) \in r^+\)
obtains
(base) \((a, b) \in r\)
| (step) \(c\) where \((a, c) \in r^+\) and \((c, b) \in r\)
using assms by cases simp-all

lemma trancl-Int-subset: \(r \subseteq s \Rightarrow (r^+ \cap s) O r \subseteq s \Rightarrow r^+ \subseteq s\)
apply clarify
apply (erule trancl-induct, auto)
done

lemma trancl-unfold: \(r^+ = r \cup r^+\) O r
by (auto intro: trancl-into-trancl elim: tranclE)

Transitivity of \(r^+\)

lemma trans-trancl [simp]: trans \((r^+)\)
proof (rule transI)
fix x y z
assume \((x, y) \in r^+\)
assume \((y, z) \in r^+\)
then show \((x, z) \in r^+\)
proof induct
  case (base u)
  from \((x, y) \in r^+\) and \((y, u) \in r\)
  show \((x, u) \in r^+\) ..
next
  case (step u v)
  from \((x, u) \in r^+\) and \((u, v) \in r\)
  show \((x, v) \in r^+\) ..
qed
qed

lemmas trancl-trans = trans-trancl [THEN transD]
lemma tranclp-trans:
assumes $r^+ x y$
and $r^+ y z$
shows $r^+ x z$
using assms(2,1) by induct iprover+

lemma trancl-id [simp]: $\text{trans } r \implies r^+ = r$
apply auto
apply (erule trancl-induct, assumption)
apply (unfold trans-def, blast)
done

lemma rtranclp-tranclp-tranclp:
assumes $r^{**} x y$
shows $\forall z. r^+ y z \implies r^+ x z$
using assms by induct (iprover intro: tranclp-trans)+

lemmas rtrancl-trancl-trancl = rtranclp-tranclp-tranclp [to-set]

lemma tranclp-into-tranclp2: $r a b \implies r^+ b c \implies r^+ a c$
by (erule tranclp-trans [OF tranclp.r-into-trancl])

lemmas tranclp-into-trancl2 = tranclp-into-tranclp2 [to-set]

lemma tranclp-converseI: $(r^+)^{-1-1} x y \implies (r^{-1-1})^{++} x y$
apply (erule conversepD)
apply (erule tranclp-induct)
apply (iprover intro: conversepI tranclp-trans)+
done

lemmas tranclp-converseI = tranclp-converseI [to-set]

lemma tranclp-converseD: $(r^{-1-1})^{++} x y \implies (r^{++})^{-1-1} x y$
apply (rule conversepI)
apply (erule tranclp-induct)
apply (iprover dest: conversepD intro: tranclp-trans)+
done

lemmas tranclp-converseD = tranclp-converseD [to-set]

lemma tranclp-converse: $(r^{-1-1})^{++} = (r^{++})^{-1-1}$
by (fastforce simp add: fun-eq-iff intro!: tranclp-converseI dest!: tranclp-converseD)

lemmas trancl-converse = tranclp-converse [to-set]

lemma sym-trancl: $\text{sym } r \implies \text{sym } (r^+)$
by (simp only: sym-conv-converse-eq trancl-converse [symmetric])
lemma converse-tranclp-induct [consumes 1, case-names base step]:
assumes major: \( r^{++} a b \)
and cases: \( \forall y. r y b \rightarrow P y \land \forall z. r y z \rightarrow r^{++} z b \rightarrow P z \rightarrow P y \)
shows \( P a \)
apply (rule tranclp-induct [OF tranclp-converseI, OF conversepI, OF major])
apply (blast intro: cases)
apply (blast intro: assms dest: tranclp-converseD)
done

lemmas converse-trancl-induct = converse-tranclp-induct [to-set]

lemma tranclpD: \( R^{++} x y \rightarrow \exists z. R x z \land R^{++} z y \)
apply (erule converse-tranclp-induct, auto)
apply (blast intro: rtranclp-trans)
done

lemmas tranclD = tranclpD [to-set]

lemma converse-tranclpE:
assumes major: tranclp \( r x z \)
and base: \( r x z \rightarrow P \)
and step: \( \forall y. r x y \rightarrow \text{tranclp} r y z \rightarrow P \)
shows \( P \)
proof —
from tranclpD [OF major] obtain \( y \) where \( r x y \) and \( \text{rtranclp} r y z \)
by iprover
from this(2) show \( P \)
proof (cases rule: rtranclp.cases)
case rtrancl-refl
with \( r x y \) base show \( P \)
by iprover
next
case rtrancl-into-rtrancl
from this have tranclp \( r y z \)
by (iprover intro: rtranclp-into-tranclpI)
with \( r x y \) step show \( P \)
by iprover
qed
qed

lemmas converse-tranclE = converse-tranclpE [to-set]

lemma tranclD2: \( (x, y) \in R^{+} \rightarrow \exists z. (x, z) \in R^{*} \land (z, y) \in R \)
by (blast elim: tranclE intro: trancl-into-rtrancl)

lemma irrefl-tranclI: \( r^{-1} \cap r^{*} = \{\} \rightarrow (x, x) \notin r^{+} \)
by (blast elim: tranclE dest: trancl-into-rtrancl)

lemma irrefl-trancl-rD: \( \forall x. (x, x) \notin r^{+} \rightarrow (x, y) \in r \rightarrow x \neq y \)
by (blast dest: r-into-trancl)

lemma trancl-subset-Sigma-aux: \((a, b) \in r^* \Longrightarrow r \subseteq A \times A \Longrightarrow a = b \lor a \in A\)

by (induct rule: rtrancl-induct) auto

lemma trancl-subset-Sigma: \(r \subseteq A \times A \Longrightarrow r^+ \subseteq A \times A\)

apply (clarsimp simp:)
apply (erule tranclE)
apply (blast dest: trancl-into-rtrancl trancl-subset-Sigma-aux+)
done

lemma reflclp-tranclp [simp]: \((r^++)^* = r^{**}\)
apply (safe intro: order-antisym)
apply (erule tranclp-into-rtranclp)
apply (blast elim: rtranclp_cases dest: rtranclp-into-tranclp1)
done

lemmas reflcl-trancl [simp] = reflclp-tranclp [to-set]

lemma trancl-reflcl [simp]: \((r^=)^+ = r^*\)

proof
  have \((a, b) \in (r^=)^+ \Longrightarrow (a, b) \in r^*\) for \(a, b\)
    by (force dest: trancl-into-rtrancl)
  moreover have \((a, b) \in (r^=)^+\) if \((a, b) \in r^*\) for \(a, b\)
    using that
  proof (cases a b rule: rtranclE)
    case step
    show \(?thesis\)
      by (rule rtrancl-into-trancl1) (use step in auto)
  qed auto
ultimately show \(?thesis\)
  by auto
qed

lemma rtrancl-trancl-reflcl [code]: \(r^* = (r^+)^=\)
by simp

lemma trancl-empty [simp]: \(\{\}\^+ = \{\}\)
by (auto elim: trancl-induct)

lemma trancl-empty [simp]: \(\{\}\^* = Id\)
by (rule subst [OF reflcl-trancl]) simp

lemma rtranclpD: \(R^{**} a b \Longrightarrow a = b \lor a \neq b \land R^{++} a b\)
by (force simp: reflclp-tranclp [symmetric] simp del: reflclp-tranclp)

lemmas rtranclD = rtranclpD [to-set]

lemma rtrancl-eq-or-trancl: \((x, y) \in R^* \iff x = y \lor x \neq y \land (x, y) \in R^+\)
by (fast elim: trancl-into-rtrancl dest: rtranclD)

lemma trancl-unfold-right: \( r^+ = r^* \circ r \)
by (auto dest: tranclD2 intro: rtrancl-into-trancl1)

lemma trancl-unfold-left: \( r^+ = r \circ r^* \)
by (auto dest: tranclD intro: rtrancl-into-trancl2)

lemma trancl-insert: \( (\text{insert} \ (y, x) \ r^*)_+ = r^* \cup \{(a, b), (a, y) \in r^* \land (x, b) \in r^*\} \)
— primitive recursion for \( trancl \) over finite relations

proof –
have \( \forall a \ b. (a, b) \in (\text{insert} \ (y, x) \ r^*)_+ \Rightarrow (a, b) \in r^* \cup \{(a, b), (a, y) \in r^* \land (x, b) \in r^*\} \)
by (erule trancl-induct) (blast intro: rtrancl-into-trancl1 trancl-into-rtrancl trancl-trans) +
moreover have \( r^* \cup \{(a, b), (a, y) \in r^* \land (x, b) \in r^*\} \subseteq (\text{insert} \ (y, x) \ r^*)_+ \)
by (blast intro: trancl-mono rtrancl-mono [THEN \[2\] rev-subsetD]
   rtrancl-trancl-trancl rtrancl-into-trancl2)
ultimately show \(?thesis\)
by auto
qed

lemma trancl-insert2:
\( (\text{insert} \ (a, b) \ r^*)_+ = r^* \cup \{(x, y), ((x, a) \in r^* \lor x = a) \land ((b, y) \in r^* \lor y = b)\} \)
by (auto simp: trancl-insert rtrancl-eq-or-trancl)

lemma rtrancl-insert: \( (\text{insert} \ (a,b) \ r)^* = r^* \cup \{(x, y), (x, a) \in r^* \land (b, y) \in r^*\} \)
using trancl-insert[of \ a \ b \ r]
by (simp add: rtrancl-trancl-refcl del: reflcl-trancl) blast

Simplifying nested closures

lemma rtrancl-trancl-absorb[simp]: \( (R^*)_+ = R^* \)
by (simp add: trans-rtrancl)

lemma trancl-rtrancl-absorb[simp]: \( (R^+)^* = R^* \)
by (subst reflcl-trancl[symmetric]) simp

lemma rtrancl-refcl-absorb[simp]: \( (R^*)^o = R^* \)
by auto

Domain and Range

lemma Domain-rtrancl [simp]: Domain \( (R^*) = \text{UNIV} \)
by blast

lemma Range-rtrancl [simp]: Range \( (R^*) = \text{UNIV} \)
by blast
theory "Transitive-Closure"

lemma rtrancl-Un-subset: \((R^* \cup S^*) \subseteq (R \cup S)^*\)
  by (rule rtrancl-Un-rtrancl [THEN subst]) fast

lemma in-rtrancl-UnI: \(x \in R^* \lor x \in S^* \Rightarrow x \in (R \cup S)^*\)
  by (blast intro: subsetD [OF rtrancl-Un-subset])

lemma trancl-domain [simp]: Domain \((r^+) = \text{Domain } r\)
  by (unfold Domain-unfold) (blast dest: tranclD)

lemma trancl-range [simp]: Range \((r^+) = \text{Range } r\)
  unfolding Domain-converse [symmetric] by (simp add: trancl-converse [symmetric])

lemma Not-Domain-rtrancl:
  assumes \(x \notin \text{Domain } R\)
  shows \((x, y) \in R^* \longleftrightarrow x = y\)
proof
  have \((x, y) \in R^* \Rightarrow x = y\)
    by (erule rtrancl-induct) (use assms in auto)
  then show ?thesis
    by auto
qed

lemma trancl-subset-Field2: \(r^+ \subseteq \text{Field } r \times \text{Field } r\)
  apply clarify
  apply (erule trancl-induct)
  apply (auto simp: Field-def)
  done

lemma finite-trancl[simp]: finite \((r^+) = \text{finite } r\)
proof
  show \((r^+) \Rightarrow \text{finite } r\)
    by (blast intro: r-into-trancl' finite-subset)
  show \(\text{finite } r \Rightarrow \text{finite } (r^+)\)
    apply (rule trancl-subset-Field2 [THEN finite-subset])
    apply (auto simp: finite-Field)
  done
qed

lemma finite-rtrancl-Image[simp]: assumes \(\text{finite } R \text{ finite } A\)
  shows \(\text{finite } (R^* "\ " A)\)
proof (rule contr)
  assume infinite \((R^* "\ " A)\)
  with assms show False
    by (simp add: rtrancl-trancl-refl Un-Image del: reflcl-trancl)
qed

More about converse rtrancl and trancl, should be merged with main body.

lemma single-valued-confluent:
  assumes \(\text{single-valued } r \text{ and } xy: (x, y) \in r^* \text{ and } xz: (x, z) \in r^*\)
  shows \((y, z) \in r^* \lor (z, y) \in r^*\)
using xy

proof (induction rule: rtrancl-induct)
  case base
  show ?case
    by (simp add: assms)
next
  case (step y z)
  with xz ⟨single-valued r⟩
  show ?case
    apply (auto simp: elim: converse-rtranclE dest: single-valuedD)
    apply (blast intro: rtrancl-trans)
    done
qed

lemma r-r-into-trancl: 
  (a, b) ∈ R \implies (b, c) ∈ R \implies (a, c) ∈ R^+
  by (fast intro: trancl-trans)

lemma trancl-into-trancl: 
  (a, b) ∈ r^+ \implies (b, c) ∈ r \implies (a, c) ∈ r^+
  by (induct rule: trancl-induct) (fast intro: r-r-into-trancl trancl-trans)+

lemma tranclp-rtranclp-tranclp: 
  r++ a b \implies r** b c \implies r++ a c
  apply (drule tranclpD)
  apply (elim exE conjE)
  apply (drule rtranclp-trans, assumption)
  apply (drule (2) rtranclp-into-tranclp2)
  done

lemma rtranclp-conversep: r^−1−1** = r**−1−1
  by (auto simp add: fun-eq-iff intro: rtranclp-converseI rtranclp-converseD)

lemmas symp-rtranclp = sym-rtrancl[to-pred]
lemmas symp-conv-conversep-eq = sym-conv-converse-eq[to-pred]
lemmas rtranclp-rtranclp-absorb[simp] = rtrancl-absorb[to-pred]
lemmas tranclp-rtranclp-absorb[simp] = trancl-absorb[to-pred]
lemmas rtranclp-reflclp-absorb[simp] = rtrancl-refl-absorb[to-pred]

lemmas tranclp-tranclp-tranclp = tranclp-rtranclp-tranclp [to-set]
lemmas transitive-closure-trans'[trans] =
  r-r-into-trancl trancl-trans rtrancl-trans
  trancl.trancl-into-trancl trancl-trans
  rtrancl.rtrancl-into-rtrancl converse-rtrancl-into-rtrancl
  trancl-trancl-trancl trancl-rtrancl-trancl

lemmas transitive-closurep-trans'[trans] =
  tranclp-trans rtranclp-trans
  tranclp.trancl-into-trancl tranclp-trans
  rtranclp.rtranclp-into-rtrancl transitive-closurep-trans
rtranclp-tranclp-tranclp tranclp-rtranclp-tranclp

declare trancl-into-rtrancl [elim]

19.4 Symmetric closure

definition symclp :: (′a ⇒ ′a ⇒ bool) ⇒ ′a ⇒ ′a ⇒ bool
where symclp r x y ←→ r x y ∨ r y x

lemma symclpI [simp, intro?]:
  shows symclpI1: r x y ⇒ symclp r x y
  and symclpI2: r y x ⇒ symclp r x y
by (simp-all add: symclp-def)

lemma symclpE [consumes 1, cases pred]:
  assumes symclp r x y
  obtains (base) r x y | (sym) r y x
  using assms by (auto simp add: symclp-def)

lemma symclp-pointfree: symclp r = sup r r−1−1
by (auto simp add: symclp-pointfree sup-commute)

lemma symclp-greater: r ≤ symclp r
by (simp add: symclp-pointfree)

lemma symclp-conversep [simp]: symclp r−1−1 = symclp r
by (simp add: symclp-pointfree sup-commute)

lemma symp-symclp [simp]: symp (symclp r)
by (auto simp add: symp-def elim: symclpE intro: symclpI)

lemma symp-symclp-eq: symp r ⇒ symclp r = r
by (simp add: symp-pointfree symp-conv-conversep-eq)

lemma symp-rtranclp-symclp [simp]: symp (symclp r)**
by (simp add: symp-rtranclp)

lemma rtranclp-symclp-sym [sym]: (symclp r)** x y ⇒ (symclp r)** y x
by (rule sympD[OF symp-rtranclp-symclp])

lemma symclp-idem [simp]: symclp (symclp r) = symclp r
by (simp add: symclp-pointfree sup-commute converse-join)

lemma reflp-rtranclp [simp]: reflp R**
using reflp-rtrancl[to-pred, of R] reflp-refl-eq[of {(x, y). R** x y}] by simp

19.5 The power operation on relations

R ^^ n = R O ... O R, the n-fold composition of R
overloading
relpow ≡ compow :: nat ⇒ ('a × 'a) set ⇒ ('a × 'a) set
relpowp ≡ compow :: nat ⇒ ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ 'a ⇒ bool)

begin
primrec relpow :: nat ⇒ ('a × 'a) set ⇒ ('a × 'a) set
where
relpow 0 R = Id
| relpow (Suc n) R = (R ° n) O R

primrec relpowp :: nat ⇒ ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ 'a ⇒ bool)
where
relpowp 0 R = HOL.eq
| relpowp (Suc n) R = (R ° n) OO R

end

lemma relpowp-relpow-eq [pred-set-conv]:
(λx y. (x, y) ∈ R) ° n = (λx y. (x, y) ∈ R ° n) for R :: 'a rel
by (induct n) (simp-all add: relcompp-relcomp-eq)

For code generation:
definition relpow :: nat ⇒ ('a × 'a) set ⇒ ('a × 'a) set
where relpow-code-def [code-abbrev]: relpow = compow
definition relpowp :: nat ⇒ ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ 'a ⇒ bool)
where relpowp-code-def [code-abbrev]: relpowp = compow

lemma [code]:
relpow (Suc n) R = (relpow n R) O R
relpow 0 R = Id
by (simp-all add: relpow-code-def)

lemma [code]:
relpowp (Suc n) R = (R ° n) OO R
relpowp 0 R = HOL.eq
by (simp-all add: relpowp-code-def)

hide-const (open) relpow
hide-const (open) relpowp

lemma relpow-1 [simp]: R ° 1 = R
for R :: ('a × 'a) set
by simp

lemma relpowp-1 [simp]: P ° 1 = P
for P :: 'a ⇒ 'a ⇒ bool
by (fact relpow-1 [to-pred])
THEORY “Transitive-Closure”

lemma relpow-0-I: \((x, x) \in R \^\^ 0\)
  by simp

lemma relpowp-0-I: \((P \^\^ 0) \ x \ x\)
  by (fact relpow-0-I [to-pred])

lemma relpow-Suc-I: \((x, y) \in R \^\^ n \implies (y, z) \in R \implies (x, z) \in R \^\^ Suc n\)
  by auto

lemma relpowp-Suc-I: \((P \^\^ n) \ x \ y \implies P y \ z \implies (P \^\^ Suc n) \ x \ z\)
  by (fact relpow-Suc-I [to-pred])

lemma relpow-Suc-I2: \((x, y) \in R \implies (y, z) \in R \^\^ n \implies (x, z) \in R \^\^ Suc n\)
  by (induct n arbitrary: z) (simp, fastforce)

lemma relpowp-Suc-I2: \((P \^\^ n) \ x \ y \implies P y \ z \implies (P \^\^ Suc n) \ x \ z\)
  by (fact relpow-Suc-I2 [to-pred])

lemma relpow-0-E: \((x, y) \in R \^\^ 0 \implies (x = y \implies P) \implies P\)
  by simp

lemma relpowp-0-E: \((P \^\^ 0) \ x \ y \implies (x = y \implies Q) \implies Q\)
  by (fact relpow-0-E [to-pred])

lemma relpow-Suc-E: \((x, z) \in R \^\^ Suc n \implies (\\(\forall\)y. (x, y) \in R \^\^ n \implies (y, z) \in R \implies P) \implies P\)
  by auto

lemma relpowp-Suc-E: \((P \^\^ Suc n) \ x \ z \implies (\\(\forall\)y. (P \^\^ n) \ x \ y \implies P y \ z \implies Q) \implies Q\)
  by (fact relpow-Suc-E [to-pred])

lemma relpow-E:
\((x, z) \in R \^\^ n \implies (n = 0 \implies x = z \implies P) \implies (\\(\forall\)m. n = Suc m \implies (x, y) \in R \^\^ m \implies (y, z) \in R \implies P) \implies P\)
  by (cases n) auto

lemma relpowp-E:
\((P \^\^ n) \ x \ z \implies (n = 0 \implies x = z \implies Q) \implies (\\(\forall\)m. n = Suc m \implies (P \^\^ m) \ x \ y \implies P y \ z \implies Q) \implies Q\)
  by (fact relpow-E [to-pred])

lemma relpow-Suc-D2: \((x, z) \in R \^\^ Suc n \implies (\exists y. (x, y) \in R \land (y, z) \in R \^\^ n)\)
  by (induct n arbitrary: x y) (blast intro: relpow-0-I relpow-Suc-I elim: relpow-0-E relpow-Suc-E)
lemma relpowp-Suc-D2: \((P \circ n \circ Suc \ n) \ x \ z \implies \exists y. \ P \ x \ y \land (P \circ n) \ y \ z\)
by (fact relpow-Suc-D2 [to-pred])

lemma relpow-Suc-E2: \((x, z) \in R \circ Suc \ n \implies (\forall y. \ (x, y) \in R \implies (y, z) \in R \circ n \implies P) \implies P\)
by (blast dest: relpow-Suc-D2)

lemma relpow-Suc-E2 ': \forall x y z. \ ((P \circ n) \ x \ y \land P y z \implies (\exists w. \ ((P \circ n) \ w \ z) \implies Q) \implies Q\)
by (fact relpow-Suc-E2' [to-pred])

lemma relpow-Suc-D2': \forall x y z. \ ((P \circ n) \ x \ y \land P y z \implies (\exists w. \ ((P \circ n) \ w \ z) \implies P) \implies P\)
by (induct n) (simp-all, blast)

lemma relpowp-Suc-D2': \forall x y z. \ ((P \circ n) \ x \ y \land (P \circ n) \ y \ z \implies Q) \implies Q\)
by (fact relpow-Suc-D2' [to-pred])

lemma relpow-E2:
assumes \((x, z) \in R \circ n \ n = 0 \implies x = z \implies P\)
\(\forall y m. \ n = Suc m \implies (x, y) \in R \implies (y, z) \in R \circ m \implies P\)
shows \(P\)
proof (cases n)
case \(0\)
with assms show ?thesis
by simp
next
case \(Suc m\)
with assms relpow-Suc-D2' [of m R] show ?thesis
by force
qed

lemma relpowp-E2:
\((P \circ n) \ x \ z \implies (n = 0 \implies x = z \implies Q) \implies (\forall y m. \ n = Suc m \implies P \ x \ y \implies (P \circ m) \ y \ z \implies Q) \implies Q\)
by (fact relpow-E2 [to-pred])

lemma relpow-add: \(R \circ (m + n) = R \circ m \ O \ R \circ n\)
by (induct n) auto

lemma relpowp-add: \((P \circ (m + n)) = P \circ m \ OO \ P \circ n\)
by (fact relpow-add [to-pred])

lemma relpow-commute: \(R \ O \ R \circ n = R \circ n \ O \ R\)
by (induct n) (simp-all add: O-assoc [symmetric])

lemma relpowp-commute: \(P \ O \ P \circ n = P \circ n \ OO \ P\)
by (fact relpow-commute [to-pred])

lemma relpow-empty: \(0 < n \implies (\{\} :: (\{a\} \times \{a\}) ^ ^ n = \{\})\)
by (cases n) auto

lemma relpowp-bot: \(0 < n \implies (\bot :: \{a\} \Rightarrow \{a\} \Rightarrow \{bool\}) ^ ^ n = \bot\)
by (fact relpow-empty [to-pred])

lemma rtrancl-imp-UN-relpow:
assumes \(p \in R^*\)
shows \(p \in (\bigcup n. R ^ ^ n)\)
proof (cases p)
  case (Pair x y)
  with assms have \((x, y) \in R^*\) by simp
  then have \((x, y) \in (\bigcup n. R ^ ^ n)\)
  proof induct
    case base
    show ?case by (blast intro: relpow-0-I)
  next
    case step
    then show ?case by (blast intro: relpow-Suc-I)
  qed
  with Pair show ?thesis by simp
qed

lemma rtranclp-imp-Sup-relpowp:
assumes \((P^{**}) x y\)
shows \((\bigcup n. P ^ ^ n) x y\)
using assms and rtrancl-imp-UN-relpow [of \((x, y)\), to-pred] by simp

lemma relpow-imp-rtrancl:
assumes \(p \in R ^ ^ n\)
shows \(p \in R^*\)
proof (cases p)
  case (Pair x y)
  with assms have \((x, y) \in R ^ ^ n\) by simp
  then have \((x, y) \in R^*\)
  proof (induct n arbitrary: \(x y\))
    case 0
    then show ?case by simp
  next
    case Suc
    then show ?case
      by (blast elim: relpow-Suc-E intro: rtrancl-into-rtrancl)
  qed
  with Pair show ?thesis by simp
qed

lemma relpowp-imp-rtranclp: \((P ^ ^ n) x y \implies (P^{**}) x y\)
using relpow-imp-rtrancl [of \((x, y)\), to-pred] by simp

lemma rtrancl-is-UN-relpow: \(R^* = \bigcup n. R \quad n\)
  by (blast intro: rtrancl-imp-UN-relpow relpow-imp-rtrancl)

lemma rtranclp-is-Sup-relpowp: \((P^*) = \bigcup n. P \quad n\)
  using rtrancl-is-UN-relpow [to-pred, of P] by auto

lemma rtrancl-power: \(p \in R^* \iff (\exists n. p \in R \quad n)\)
  by (simp add: rtrancl-is-UN-relpow)

lemma rtranclp-power: \((P^*) x y \iff (\exists n. (P \quad n) x y)\)
  by (simp add: rtranclp-is-Sup-relpowp)

lemma trancl-power: \(p \in R^+ \iff (\exists n > 0. p \in R \quad n)\)
  proof
    have \((a, b) \in R^+ = (\exists n>0. (a, b) \in R \quad n)\) for \(a b\)
      proof safe
        show \((a, b) \in R^+ \implies (\exists n>0. (a, b) \in R \quad n)\)
          apply (drule tranclD2)
          apply (fastforce simp: rtrancl-is-UN-relpow relcomp-unfold)
          done
        show \((a, b) \in R^+ \iff (\exists n > 0. (a, b) \in R \quad n)\) for \(n\)
          proof (cases n)
            case (Suc m)
            with that show \(?thesis\)
              by (auto simp: dest: relpow-imp-rtrancl rtrancl-into-trancl1)
          qed (use that in auto)
        qed
      qed
    then show \(?thesis\)
      by (cases p) auto
    qed

lemma tranclp-power: \((P^{++}) x y \iff (\exists n > 0. (P \quad n) x y)\)
  using trancl-power [to-pred, of P (x, y)] by simp

lemma rtrancl-imp-relpow: \(p \in R^* \implies (\exists n. p \in R \quad n)\)
  by (auto dest: rtrancl-imp-UN-relpow)

lemma rtranclp-imp-relpowp: \((P^{++}) x y \implies (\exists n. (P \quad n) x y)\)
  by (auto dest: rtranclp-imp-Sup-relpowp)

By Sternagel/Thiemann:

lemma relpow-fun-conv: \((a, b) \in R \quad n \iff (\exists f. f 0 = a \land f n = b \land (\forall i<n. (f i, f (Suc i)) \in R))\)
  proof (induct n arbitrary: b)
    case 0
    show \(?case\) by auto
  next
THEORY "Transitive-Closure"

case (Suc n)
show ?case
proof (simp add: relcomp-unfold Suc)
  show \( (\exists y. (\exists f. f 0 = a \land f n = y \land (\forall i<n. (f i, f (Suc i)) \in R)) \land (y, b) \in R) \iff (\exists f. f 0 = a \land f (Suc n) = b \land (\forall i<Suc n. (f i, f (Suc i)) \in R)) \) (is ?l = ?r)
  proof
    assume ?l
    then obtain \( c \) \( f \).
    where \( 1: f 0 = a \land f n = c \land \forall i\leq n. (f i, f (Suc i)) \in R \land (c, b) \in R \) by auto
    let ?g = \( \lambda m. \) if \( m = Suc \) then \( b \) else \( f m \)
    show ?r by (rule exI[of - ?g]) (simp add: 1)
  next
    assume ?r
    then obtain \( f \).
    where \( 1: f 0 = a \land f (Suc n) = b \land \forall i\leq Suc n. (f i, f (Suc i)) \in R \) by auto
    show ?l by (rule exI[of - f]) (rule conjI, rule exI[of - f], insert 1, auto)
  qed
qed

lemma relpowp-fun-conv: \( (P ^{\leq n}) x y \iff (\exists f. f 0 = x \land f n = y \land (\forall i<n. P (f i, f (Suc i)))) \) by (fact relpow-fun-conv[to-pred])

lemma relpow-finite-bounded1:
  fixes \( R \colon (\'a \times \'a) \) set
  assumes \( \text{finite } R \) and \( k \geq 0 \)
  shows \( R^{\leq k} \subseteq (\bigcup n \in \{ n. 0 < n \land n \leq \card R \}. R^{\leq n}) \) (is - \subseteq ?r)
  proof
    have \( (a, b) \in R^{\leq} (Suc \) \( k \) \) \( \implies \) \( \exists n. 0 < n \land n \leq \card R \land (a, b) \in R^{\leq n} \) for \( a \) \( b \) \( k \)
    proof (induct \( k \) arbitrary: \( b \))
      case \( 0 \)
      then have \( R \neq \{ \} \) by auto
      with \( \text{card-0-eq[OF \text{finite } R]} \) have \( \text{card } R \geq Suc 0 \) by auto
      then show ?case using \( 0 \) by force
    next
      case (Suc \( k \))
      then obtain \( a' \) where \( (a, a') \in R^{\leq} (Suc \) \( k \) \) and \( (a', b) \in R \) by auto
      from \( \text{Suc}(1)[OF \ (a, a') \in R^{\leq}(Suc \) \( k \)] \) obtain \( n \) where \( n \leq \card R \) and \( (a, a') \in R^{\leq n} \) by auto
      have \( (a, b) \in R^{\leq} (Suc \) \( n \) \)
    qed
  qed
using \((a, a') \in R^{-n}\) and \((a', b) \in R\) by auto
from \(n \leq \text{card } R\) consider \(n < \text{card } R\) \(\mid n = \text{card } R\) by force
then show \(\text{thesis}\)
proof cases
  case 1
  then show \(\text{thesis}\) using \((a, b) \in R^{\text{Suc } n}\) by blast
next
case 2
from \((a, b) \in R^{-\text{Suc } n}\) obtain \(f\) where \(f 0 = a\) and \(f (\text{Suc } n) = b\)
  and steps: \(\forall i.\ i \leq n \implies (f i, f (\text{Suc } i)) \in R\) by auto
let \(\mathcal{N} = \{i.\ i \leq n\}\)
  have \(\mathcal{N} \subseteq R\) by blast
next
  case 2
  from \((a, b) \in R^{\text{Suc } n}\) obtain \(i, j\) where \(i \leq n\) and \(j \leq n\) and \(i \neq j\) and \(\mathcal{N} \subseteq R\)
  by (rule pigeonhole)
then obtain \(i, j\) where \(i \leq n\) and \(j \leq n\) and \(i \neq j\) and \(\mathcal{N} \subseteq R\)
by (auto simp: inj-on_def)
let \(\mathcal{N} = \{i.\ i \leq n\}\)
  have \(\mathcal{N} \subseteq \text{card } R\) by auto
finally have \(\neg \text{inj-on } \mathcal{N}\) by blast
let \(\mathcal{N} = \{i.\ i \leq n\}\)
  have \(\mathcal{N} \subseteq \text{card } R\) by auto
next
fix \(k\)
assume \(k < \mathcal{N}\)
show \((a, b) \in R\) by auto
proof cases \(k < i\)
  case True
  with \(i\) have \(k \leq n\) by auto
  from \((a, b) \in R\) show \(\text{thesis}\) using True by blast
next
case False
then have \( i \leq k \) by auto
show \(?thesis\)
proof (cases \( k = i \))
case True
then show \(?thesis\)
using \( ij \) pij steps[\( OF \ i \)] by simp
next
case False
with \( (i \leq k) \) have \( i < k \) by auto
then have \( \text{small} : k + (j - i) \leq n \)
using \( \langle k < ?n \rangle \) by arith
show \(?thesis\)
using steps[\( OF \ \text{small} \) \( i < k \)] by auto
qed
qed
qed (simp add: \( \langle f 0 = a \rangle \))
moreover have \(?n \leq n\)
using \( i j \) ij steps\[ OF \ ?n \] by arith
ultimately show \(?thesis\)
using \( \langle n = \text{card} R \rangle \) by blast
qed
qed
then show \(?thesis\)
using gr0-implies-Suc[\( OF \ i > 0 \)] by auto
qed

lemma relpow-finite-bounded:
fixes \( R :: ('a \\times 'a) \) set
assumes finite \( R \)
shows \( R^* \subseteq (\bigcup n \in \{ n. \ n \leq \text{card} R \}. R^n) \)
apply (cases \( k, \) force)
apply (use relpow-finite-bounded1[\( OF \ \text{assms, of} \ k \] in auto)
done

lemma rtrancl-finite-eq-relpow: finite \( R \) \( \Rightarrow \) \( R^* = (\bigcup n \in \{ n. \ 0 < n \wedge n \leq \text{card} R \}. R^n) \)
by (fastforce simp: rtrancl-power dest: relpow-finite-bounded)

lemma trancl-finite-eq-relpow: finite \( R \) \( \Rightarrow \) \( R^+ = (\bigcup n \in \{ n. \ 0 < n \wedge n \leq \text{card} R \}. R^n) \)
apply (auto simp: trancl-power)
apply (auto dest: relpow-finite-bounded1)
done

lemma finite-relcomp[simp,intro]:
assumes finite \( R \) and finite \( S \)
shows finite \( (R \ O \ S) \)
proof-
have \( R \ O \ S = (\bigcup (x, y) \in R. \bigcup (u, v) \in S. \text{ if } u = y \text{ then } \{(x, v)\} \text{ else } \{\} ) \)
by (force simp: split-def image-constant-conv split: if-splits)
then show ?thesis
  using assms by clarsimp
qed

lemma finite-relpow [simp, intro]:
  fixes R :: ('a × 'a) set
  assumes finite R
  shows n > 0 ⇒ finite (R ^^ n)
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then show ?case by (cases n) (use assms in simp-all)
qed

lemma single-valued-relpow:
  fixes R :: ('a × 'a) set
  shows single-valued R ⇒ single-valued (R ^^ n)
proof (induct n arbitrary: R)
  case 0
  then show ?case by simp
next
  case (Suc n)
  show ?case
    by (rule single-valuedI)
      (use Suc in :fast dest: single-valuedD elim: relpow-Suc-E)
qed

19.6 Bounded transitive closure

definition ntrancl :: nat ⇒ ('a × 'a) set ⇒ ('a × 'a) set
where ntrancl n R = (⋃i∈{i. 0 < i ∧ i ≤ Suc n}. R ^^ i)

lemma ntrancl-Zero [simp, code]: ntrancl 0 R = R
proof
  show R ⊆ ntrancl 0 R
    unfolding ntrancl-def by fastforce
  have 0 < i ∧ i ≤ Suc 0 ⇐⇒ i = 1 for i
    by auto
  then show ntrancl 0 R ≤ R
    unfolding ntrancl-def by auto
qed

lemma ntrancl-Suc [simp]: ntrancl (Suc n) R = ntrancl n R O (Id ∪ R)
proof
  have (a, b) ∈ ntrancl n R O (Id ∪ R) if (a, b) ∈ ntrancl (Suc n) R for a b
  proof --
from that obtain \( i \) where \( 0 < i \leq \text{Suc} (\text{Suc} n) \) \( (a, b) \in R \^\^ i \)

unfolding ntrancl-def by auto

show \(?thesis\)

proof (cases \( i = 1 \))
  case True
  from this \( (a, b) \in R \^\^ i \) show \(?thesis\)
    by (auto simp: ntrancl-def)

next
  case False
  with \( 0 < i \) obtain \( j \) where \( j : i = \text{Suc} j \) \( 0 < j \)
    by (cases \( i \)) auto

with \( (a, b) \in R \^\^ i \) obtain \( c \) where \( c1 : (a, c) \in R \^\^ j \)
  and \( c2 : (c, b) \in R \)
  by (fastforce simp: ntrancl-def)

from \( c1 j \) \( i \leq \text{Suc} (\text{Suc} n) \) have \( (a, c) \in \text{ntrancl} n R \)
  by (fastforce simp: Let-def)

with \( c2 \) show \(?thesis\) by fastforce

qed

qed

then show \( \text{ntrancl} (\text{Suc} n) R \subseteq \text{ntrancl} n R \cup (\text{Id} \cup R) \)
  by auto

show \( \text{ntrancl} n R \cup (\text{Id} \cup R) \subseteq \text{ntrancl} (\text{Suc} n) R \)
  by (fastforce simp: ntrancl-def)

qed

lemma [code]: \( \text{ntrancl} (\text{Suc} n) r = (\text{let } r' = \text{ntrancl} n r \text{ in } r' \cup r' \cup \text{O } r) \)
  by (auto simp: Let-def)

lemma finite-trancl-ntranl: finite \( R \) \( \Rightarrow \) \( \text{trancl} R = \text{ntrancl} (\text{card} R - 1) R \)
  by (cases \( \text{card} R \)) (auto simp: trancl-finite-eq-relpow relpow-empty ntrancl-def)

19.7 Acyclic relations

definition acyclic :: \(('a \times 'a) \text{ set } \Rightarrow \text{bool}\)
  where acyclic \( r \) \( \iff \forall x. (x, x) \notin r^+ \)

abbreviation acyclicP :: \( ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}\)
  where acyclicP \( r \) \( \equiv \text{acyclic} \{ (x, y). r x y \} \)

lemma acyclic-irrefl [code]: \( \text{acyclic} r \iff \text{irrefl} (r^+) \)
  by (simp add: acyclic-def irrefl-def)

lemma acyclicI: \( \forall x. (x, x) \notin r^+ \Rightarrow \text{acyclic} r \)
  by (simp add: acyclic-def)

lemma (in preorder) acyclicI-order:
  assumes \( \ast: \forall a b. (a, b) \in r \Rightarrow f b < f a \)
  shows \( \text{acyclic} r \)
proof –
have \( f b < f a \) if \( (a, b) \in r^+ \) for \( a b \)
using that by induct (auto intro: * less-trans)
then show \( \text{thesis} \)
  by (auto intro!: acyclicI)
qed

lemma acyclic-insert [iff]: acyclic (insert \((y, x)\) \(r\)) \(\iff\) acyclic \(r\) \& \((x, y) \not\in r^*\)
by (simp add: acyclic-def trancl-insert) (blast intro: rtrancl-trans)

lemma acyclic-converse [iff]: acyclic \((r^{-1})\) \(\iff\) acyclic \(r\)
by (simp add: acyclic-def trancl-converse)

lemmas acyclicP-converse [iff] = acyclic-converse [to-pred]

lemma acyclic-impl-antisym-rtrancl: acyclic \(r\) \(\implies\) antisym \((r^+\))
by (simp add: acyclic-def antisym-def) (blast elim: rtranclE intro: rtrancl-into-trancl1 rtrancl-trancl-trancl)

lemma acyclic-subset: acyclic \(s\) \(\implies\) \(r \subseteq s\) \(\implies\) acyclic \(r\)
unfolding acyclic-def by (blast intro: trancl-mono)

19.8 Setup of transitivity reasoner

ML
structure Trancl-Tac = Trancl-Tac
{
  val r-into-trancl = @{thm trancl.r-into-trancl};
  val trancl-trans = @{thm trancl-trans};
  val rtrancl-refl = @{thm rtrancl.rtrancl-refl};
  val r-into-rtrancl = @{thm r-into-rtrancl};
  val trancl-into-rtrancl = @{thm trancl-into-rtrancl};
  val rtrancl-trancl-trancl = @{thm rtrancl-trancl-trancl};
  val trancl-rtrancl-trancl = @{thm trancl-rtrancl-trancl};
  val rtrancl-trans = @{thm rtrancl-trans};

  fun decomp (const:Trueprop) $ t =
      let
        fun dec (Const (const-name (Set.member), -) $ (Const (const-name (Pair),
            -) $ a $ b) $ rel) =
            let
              fun decr (Const (const-name (rtrancl), -) $ r) = (r,r*)
              | decr (Const (const-name (trancl), -) $ r) = (r,r+)
              | decr r = (r,r);
              val (rel,r) = decr (Envir.beta-eta-contract rel);
              in SOME (a,b,rel,r) end
            in dec t end
      in dec t end
structure Tranclp-Tac = Trancl-Tac
{
  val r-into-trancl = @\{thm tranclp.r-into-trancl\};
  val trancl-trans = @\{thm tranclp.trancl-trans\};
  val rtrancl-refl = @\{thm rtranclp.rtrancl-refl\};
  val r-into-rtrancl = @\{thm r-into-rtranclp\};
  val trancl-into-rtrancl = @\{thm tranclp-into-rtranclp\};
  val rtrancl-trancl-trancl = @\{thm rtranclp-tranclp-tranclp\};
  val trancl-rtrancl-trancl = @\{thm tranclp-rtranclp-tranclp\};
  val rtrancl-trans = @\{thm rtranclp-trans\};
}

fun decomp (const : Trueprop) $ t) =
  let
    fun dec (rel $ a $ b) =
      let
        fun decr (Const (const-name : tranclp, - ) $ r) = (r,r*)
        | decr (Const (const-name : tranclp, - ) $ r) = (r,r+)
        | decr r = (r,r);
        val (rel,r) = decr rel;
        in SOME (a, b, rel, r) end
      in dec t end
  in SOME (t) end

Optional methods.

method-setup trancl =
  (Scan.succeed (SIMPLE-METHOD' o Trancl-Tac.trancl-tac))
  (simple transitivity reasoner)

method-setup rtrancl =
  (Scan.succeed (SIMPLE-METHOD' o Trancl-Tac.rtrancl-tac))
  (simple transitivity reasoner)

method-setup tranclp =
  (Scan.succeed (SIMPLE-METHOD' o Tranclp-Tac.trancl-tac))
theory Wellfounded
  imports Transitive-Closure
begin

20 Well-founded Recursion

20.1 Basic Definitions

definition wf :: ('a × 'a) set ⇒ bool
  where wf r ←→ (∀P. (∀x. (∀y. (y, x) ∈ r → P y) → P x) → (∀x. P x))

definition wfP :: ('a ⇒ 'a ⇒ bool) ⇒ bool
  where wfP r ←→ wf {((x, y). r x y)}

lemma wfP-wf-eq [pred-set-conv]: wfP (λx y. (x, y) ∈ r) = wf r
  by (simp add: wfP-def)

lemma wfUNIVI: (∀x P. (∀y. (y, x) ∈ r → P y) → P x) → P x)
  unfolding wf-def by blast

lemmas wfPUNIVI = wfUNIVI [to-pred]

Restriction to domain A and range B. If r is well-founded over their inter-
section, then wf r.

lemma wfI:
  assumes r ⊆ A × B
  and (∀x P. (∀y. (y, x) ∈ r → P y) → P x; x ∈ A; x ∈ B) ⇒ P x
  shows wf r
  using assms unfolding wf-def by blast

lemma wf-induct:
  assumes wf r
  and (∀x. ∀y. (y, x) ∈ r → P y) → P x
  shows P a
  using assms unfolding wf-def by blast

lemmas wfP-induct = wf-induct [to-pred]

lemmas wf-induct-rule = wf-induct [rule-format, consumes 1, case-names less,
  induct set: wf]
lemmas wfP-induct-rule = wf-induct-rule [to-pred, induct set: wfP]

lemma wf-not-sym: wf r \implies (a, x) \in r \implies (x, a) /\in r
by (induct a arbitrary: x set: wf) blast

lemma wf-asym:
assumes wf r (a, x) \in r
obtains (x, a) /\in r
by (drule wf-not-sym[OF assms])

lemma wf-not-refl [simp]: wf r \implies (a, a) /\in r
by (blast elim: wf-asym)

lemma wf-irrefl:
assumes wf r
obtains (a, a) /\in r
by (drule wf-not-refl[OF assms])

lemma wf-wellorderI:
assumes wf: wf \{x::a::ord, y. x < y\}
and lin: OFCLASS(a::ord, linorder-class)
shows OFCLASS(a::ord, wellorder-class)
apply (rule wellorder-class.intro [OF lin])
apply (simp add: wellorder-class.intro class.wellorder-axioms.intro wf-induct-rule [OF wf])
done

lemma (in wellorder) wf: wf \{(x, y). x < y\}
unfolding wf-def by (blast intro: less-induct)

20.2 Basic Results

Point-free characterization of well-foundedness

lemma wfE-pf:
assumes wf: wf R
and a: A \subseteq R ``` A
shows A = {}
proof -
from wf have x /\in A for x
proof induct
fix x assume \(\forall y. (y, x) \in R \implies y /\in A\)
then have x /\in R ``` A by blast
with a show x /\in A by blast
qed
then show ?thesis by auto
qed

lemma wfI-pf:
assumes a: \(\forall A. A \subseteq R ``` A \implies A = {}\)
shows \( \text{wf } R \)

proof (rule \text{wfUNIVI})

fix \( P :: 'a \Rightarrow \text{bool} \) and \( x \)

let \( ?A = \{ x, \neg P x \} \)

assume \( \forall x. (\forall y. (y, x) \in R \Rightarrow P y) \Rightarrow P x \)

then have \( ?A \subseteq R \Rightarrow ?A \) by blast

with a show \( P x \) by blast

qed

20.2.1 Minimal-element characterization of well-foundedness

lemma \text{wfE-min}:

assumes \( \text{wf } R \)

obtains \( z \) where \( z \in Q \wedge (\forall y. (y, z) \in R \Rightarrow y \notin Q) \)

using \( Q \text{ wfE-pf} \) by blast

lemma \text{wfE-min}':

\( \text{wf } R \Rightarrow Q \neq \{ \} \Rightarrow (\forall z. z \in Q \Rightarrow (\forall y. (y, z) \in R \Rightarrow y \notin Q) \Rightarrow \text{thesis}) \)

using \( \text{wfE-min} \) by blast

lemma \text{wfI-min}:

assumes \( a: (\forall x Q. x \in Q \Rightarrow \exists z \in Q. (\forall y. (y, z) \in R \Rightarrow y \notin Q)) \)

shows \( \text{wf } R \)

proof (rule \text{wfI-pf})

fix \( A \)

assume \( b: A \subseteq R \Rightarrow A \)

have \( \text{False} \) if \( x \in A \) for \( x \)

using \( a \) by blast

then show \( A = \{ \} \) by blast

qed

lemma \text{wf-eq-minimal}:

\( \text{wf } r \iff (\forall Q x. x \in Q \Rightarrow (\exists z \in Q. (\forall y. (y, z) \in r \Rightarrow y \notin Q))) \)

apply (rule iffI)

apply (blast intro: elim!: \text{wfE-min})

by (rule \text{wfI-min}) auto

lemmas \text{wfP-\text{eq-minimal} = wfP-\text{eq-minimal} [to-pred]}

20.2.2 Well-foundedness of transitive closure

lemma \text{wf-trancl}:

assumes \( \text{wf } r \)

shows \( \text{wf } (r^+) \)

proof

have \( P x \) if \( \text{induct-step: } (\forall x. (\forall y. (y, x) \in r^+ \Rightarrow P y) \Rightarrow P x) \rightarrow P x \) for \( P x \)

proof (rule \text{induct-step})

show \( P y \) if \( (y, x) \in r^+ \) for \( y \)

using \( \text{wf } r \) and that
proof (induct x arbitrary: y)
case (less x)
  note hyp = \(\forall x' y'. (x', x) \in r \implies (y', x') \in r^+ \implies P y'\)
  from \(\langle y, x \rangle \in r^+\) show P y
proof cases
  case base
  show P y
  proof (rule induct-step)
    fix y'
    assume \((y', y) \in r\)
    with \(\langle y, x \rangle \in r\) show P y'
    by (rule hyp [of y y'])
  qed
next
  case step
  then obtain x' where \((x', x) \in r\) and \((y, x') \in r^+\)
  by simp
  then show P y by (rule hyp [of x' y])
  qed
  qed
  qed
  then show ?thesis unfolding wf-def by blast
qed

lemmas wfP-trancl = wf-trancl [to-pred]

lemma wf-converse-trancl: \(wf (r^{-1}) \implies wf ((r^+)^{-1})\)
  apply (subst trancl-converse [symmetric])
  apply (erule wf-trancl)
  done

Well-foundedness of subsets
lemma wf-subset: \(wf r \implies p \subseteq r \implies wf p\)
  by (simp add: wf-eq-minimal) fast

lemmas wfP-subset = wf-subset [to-pred]

Well-foundedness of the empty relation
lemma wf-empty [iff]: \(wf \{\}\)
  by (simp add: wf-def)

lemma wfP-empty [iff]: \(wfP (\lambda x y. False)\)
  proof
    have \(wfP bot\)
      by (fact wf-empty[to-pred bot-empty-eq2])
    then show ?thesis
      by (simp add: bot-fun-def)
  qed
lemma wf-Int1: wf r \implies wf (r \cap r')
  by (erule wf-subset) (rule Int-lower1)

lemma wf-Int2: wf r \implies wf (r' \cap r)
  by (erule wf-subset) (rule Int-lower2)

Exponentiation.

lemma wf-exp:
  assumes wf (R ^^ n)
  shows wf R
proof (rule wfI-pf)
  fix A
  assume A \subseteq R ^^ A
  then have A \subseteq (R ^^ n) ^^ A
    by (induct n) force+
  with (wf (R ^^ n)) show A = {} by (rule wfE-pf)
qed

Well-foundedness of insert.

lemma wf-insert [iff]: wf (insert (y,x) r) \iff wf r \land (x,y) \notin r^* 
(is ?lhs = ?rhs)
proof
  assume ?lhs then show ?rhs
    by (blast elim: wf-trancl [THEN wf-irrefl]
      intro: rtrancl-into-trancl1 wf-subset rtrancl-mono [THEN subsetD])

next
  assume R: ?rhs
  then have R': Q \neq {} \implies (\exists z \in Q. \forall y. (y, z) \in r \implies y \notin Q) for Q
  by (auto simp: wf-eq-minimal)
  show ?lhs
    unfolding wf-eq-minimal
proof clarify
  fix Q :: 'a set and q
  assume q \in Q
  then obtain a where a \in Q and a: \forall y. (y, a) \in r \implies y \notin Q
    using R by (auto simp: wf-eq-minimal)
  show \exists z \in Q. \forall y'. (y', z) \in insert (y, x) r \implies y' \notin Q
  proof (cases a=x)
    case True
    show ?thesis
    proof (cases y \in Q)
      case True
      then obtain z where z \in Q (z, y) \in r^*
        \land z', (z', z) \in r \implies z' \in Q \implies (z', y) \notin r^*
        using R' [of {z \in Q. (z,y) \in r^*}] by auto
      with R show ?thesis
        by (rule-tac x=z in bexI) (blast intro: rtrancl-trans)
    next
      case False
      then show ?thesis
  qed

using $a \in Q$ by blast
qed
next
case False
with $a \in Q$ show $\text{thesis}$
  by blast
qed
qed
qed

20.2.3 Well-foundedness of image

lemma wf-map-prod-image-Dom-Ran:
  fixes $r :: ('a \times 'a)$ set
  and $f :: 'a \Rightarrow 'b$
  assumes $\text{wf-r}$: $\text{wf } r$
  and inj: $\forall a a'. a \in \text{Domain } r \Rightarrow a' \in \text{Range } r \Rightarrow f a = f a' \Rightarrow a = a'$
  shows $\text{wf} (\text{map-prod } f f' r)$
proof (unfold $\text{wf-eq-minimal}$, clarify)
  fix $B :: 'b$ set and $b :: 'b$
  assume $b \in B$
  define $A$ where $A = f - ' B \cap \text{Domain } r$
  show $\exists z \in B. \forall y. (y, z) \in \text{map-prod } f f' r \rightarrow y \notin B$
  proof (cases $A = \{\}$)
    case False
    then obtain $a0$ where $a0 \in A$ and $\forall a. (a, a0) \in r \rightarrow a \notin A$
      using $\text{wfE-min}$[OF $\text{wf-r}$] by auto
    thus $\text{thesis}$
      using inj unfolding $A$-def
      by (intro $\text{bexI}$[of -$f a0$]) auto
    qed
  qed
proof (rule $\text{wf-map-prod-image-Dom-Ran}$) (auto dest: inj-onD)
lemma wf-map-prod-image: $\text{wf } r \Rightarrow \text{inj } f \Rightarrow \text{wf } (\text{map-prod } f f' r)$

20.3 Well-Foundedness Results for Unions

lemma wf-union-compatible:
  assumes $\text{wf } R \text{ wf } S$
  assumes $R \circ S \subseteq R$
  shows $\text{wf } (R \cup S)$
proof (rule $\text{wfI-min}$)
  fix $x :: 'a$ and $Q$
  let $?Q' = \{ x \in Q. \forall y. (y, x) \in R \rightarrow y \notin Q \}$
  assume $x \in Q$
  obtain $a$ where $a \in ?Q'$
    by (rule $\text{wfE-min}$ [OF $\text{wf } R; x \in Q$]) blast
  with $\text{wf } S$ obtain $z$ where $z \in ?Q'$ and zmin: $\forall y. (y, z) \in S \Rightarrow y \notin ?Q'$
    by (erule $\text{wfE-min}$)
have $y \notin Q$ if $(y, z) \in S$ for $y$

proof
  from that have $y \notin ?Q'$ by (rule zmin)
  assume $y \in Q$
  with $(y, z) \notin ?Q'$ obtain $w$ where $(w, y) \in R$ and $w \in Q$ by auto
  from $(w, y) \in R \land (y, z) \in S$ have $(w, z) \in R \cup S$ by (rule relcompI)
  with $R \cup S \subseteq R$ have $(w, z) \in R$. 
  with $(w \in Q)$ show False by contradiction
  qed
with $(z \in ?Q')$ show $\exists z \in Q. \forall y. (y, z) \in R \cup S \rightarrow y \notin Q$ by blast
qed

Well-foundedness of indexed union with disjoint domains and ranges.

lemma \texttt{wf-UN}:
  assumes $r$: $\forall i. i \in I \implies wf (r \ i)$
  and disj: $\forall i j. [i \in I; j \in I; r \ i \neq r \ j] \implies \text{Domain} (r \ i) \cap \text{Range} (r \ j) = \{\}$
  shows $wf \ (\bigcup i \in I. r \ i)$
  unfolding \texttt{wf-eq-minimal}
proof clarify
  fix $A$ and $a :: \{b$
  assume $a \in A$
  show $\exists z \in A. \forall y. (y, z) \in \bigcup (r \ i) \rightarrow y \notin A$
    proof (cases $\exists i \in I. \exists a \in A. \exists b \in A. (b, a) \in r \ i$
    case True
    then obtain $i \ b \ c$ where ibc: $i \in I \ b \in A \ c \in A \ (c,b) \in r \ i$
    by blast
    have ri: $\forall Q. Q \neq \{\} \implies \exists z \in Q. \forall y. (y, z) \in r \ i \rightarrow y \notin Q$
    using $r \ [OF \ (i \in I)]$ unfolding \texttt{wf-eq-minimal} by auto
    show ?thesis
    using ri [of $\{a. a \in A \land (\exists b \in A. (b, a) \in r \ i)\}$] ibc disj
    by blast
  next
    case False
    with $(a \in A)$ show ?thesis
    by blast
  qed
qed

lemma \texttt{wfP-SUP}:
  $\forall i. \text{wfP} \ (r \ i) \implies \forall i j. r \ i \neq r \ j \implies \inf (\text{Domainp} \ (r \ i)) \ (\text{Rangep} \ (r \ j)) = \bot$
  $\implies$ \texttt{wfP} ($\bigcup (\text{range} \ r)$)
  by (rule \texttt{wf-UN[to-pred]}) simp-all

lemma \texttt{wf-Union}:
  assumes $\forall r \in R. \text{wf} \ r$
  and $\forall r \in R. \forall s \in R. r \neq s \implies \text{Domain} \ r \cap \text{Range} \ s = \{\}$
  shows $\text{wf} \ (\bigcup R)$
Intuition: We find an \( R \cup S \)-min element of a nonempty subset \( A \) by case distinction.

1. There is a step \( a \rightarrow R b \) with \( a, b \in A \). Pick an \( R \)-min element \( z \) of the (nonempty) set \( \{ a \in A \mid \exists b \in A. \ a \rightarrow R b \} \). By definition, there is \( z' \in A \) s.t. \( z \rightarrow R z' \). Because \( z \) is \( R \)-min in the subset, \( z' \) must be \( R \)-min in \( A \). Because \( z' \) has an \( S \)-predecessor, it cannot have an \( S \)-successor and is thus \( S \)-min in \( A \) as well.

2. There is no such step. Pick an \( S \)-min element of \( A \). In this case it must be an \( R \)-min element of \( A \) as well.

**lemma** \( \text{wf-Un:} \) \( \text{wf } r \implies \text{wf } s \implies \text{Domain } r \cap \text{Range } s = \{\} \implies \text{wf } (r \cup s) \)

**using** \( \text{wf-union-compatible[of } s \text{ } r \) by simp

**lemma** \( \text{wf-union-merge:} \) \( \text{wf } (R \cup S) = \text{wf } (R \circ R \cup S \circ R \cup S) \)

**using** \( \text{wf ?A = wf ?B} \)

**proof**

assume \( \text{wf } ?A \)

with \( \text{wf-trancl} \) have \( \text{wfT:} \) \( \text{wf } (?A^+) \).

moreover have \( ?B \subseteq ?A^+ \)

by (subst trancl-unfold, subst trancl-unfold) blast

ultimately show \( \text{wf } ?B \) by (rule \( \text{wf-subset} \))

next

assume \( \text{wf } ?B \)

show \( \text{wf } ?A \)

**proof** (rule \( \text{wfI-min} \))

fix \( Q :: \) a set and \( x \)

assume \( x \in Q \)

with \( \text{wf } ?B \) obtain \( z \) where \( \forall y. \ (y, z) \in ?B \implies y \notin Q \)

by (erule \( \text{wfE-min} \))

then have \( 1: \forall y. \ (y, z) \in R \circ R \implies y \notin Q \)

and \( 2: \forall y. \ (y, z) \in S \circ R \implies y \notin Q \)

and \( 3: \forall y. \ (y, z) \in S \implies y \notin Q \)

by auto

show \( \exists z \in Q. \forall y. \ (y, z) \in ?A \implies y \notin Q \)

**proof** (cases \( \forall y. \ (y, z) \in ?A \implies y \notin Q \))

**case** True

with (\( z \in Q \)) 3 show \( ?\text{thesis} \) by blast

next

**case** False

then obtain \( z' \) where \( z' \in Q \) (\( z', z \) \( \in R \) by blast

have \( \forall y. \ (y, z') \in ?A \implies y \notin Q \)

**proof** (intro allI impI)

fix \( y \) assume (\( y, z' \) \( \in ?A \))
then show \( y \notin Q \)

**proof**

assume \( (y, z') \in R \)
then have \( (y, z) \in R \circ R \) using \( (z', z) \in R \)
with 1 show \( y \notin Q \).

next
assume \( (y, z') \in S \)
then have \( (y, z) \in S \circ R \) using \( (z', z) \in R \)
with 2 show \( y \notin Q \).

**qed**

\[ \langle z' \in Q \rangle \] show ?thesis ..

**qed**

**lemma** \( \text{wf-comp-self} \):
\( \text{wf } R \iff \text{wf } (R \circ R) \) — special case
by (rule \text{wf-union-merge} [where \( S = \{\}, \text{simplified}\])

### 20.4 Well-Foundedness of Composition

Bachmair and Dershowitz 1986, Lemma 2. [Provided by Tjark Weber]

**lemma** \( \text{qc-wf-relto-iff} \):

assumes \( R \circ S \subseteq (R \cup S)^* \circ R \) — \( R \) quasi-commutes over \( S \)
shows \( \text{wf } (S^* \circ R \circ O \circ S^*) \iff \text{wf } R \)
(is \( \text{wf } ?S \iff - \))

**proof**

show \( \text{wf } R \) if \( \text{wf } ?S \)

**proof**

have \( R \subseteq ?S \) by auto
with \text{wf-subset} [of \( ?S \)] that show \( \text{wf } R \)
by auto

**qed**

next
show \( \text{wf } ?S \) if \( \text{wf } R \)

**proof** (rule \text{wfI-pf})

fix \( A \)
assume \( A: A \subseteq ?S \iff A \)
let \( ?X = (R \cup S)^* \iff A \)
have \( \forall x: R \circ (R \cup S)^* \subseteq (R \cup S)^* \circ R \)

**proof**

have \( (x, z) \in (R \cup S)^* \circ R \) if \( (y, z) \in (R \cup S)^* \) and \( (x, y) \in R \) for \( x, y, z \)
using that

**proof** (induct \( y, z \))

**case** \text{rtrancl-refl}
then show \( ?\text{case} \) by auto

next
**case** \( \text{rtrancl-into-rtrancl a b c} \)
then have \( (x, c) \in ((R \cup S)^* \circ (R \cup S)^*) \circ R \)
using assms by blast
then show \( ?\text{thesis} \) by simp
qed

then have \( R \circ S \subseteq (R \cup S)^* \circ R \)
  using rtrancl-\( \cup \)-subset by blast
then have \( ?S \subseteq (R \cup S)^* \circ (R \cup S)^* \circ R \)
  by (simp add: relcomp-\( \circ \)-mono rtrancl-\( \circ \)-mono)
also have \( \ldots = (R \cup S)^* \circ R \)
  by (simp add: O-assoc[\( \circ \)-symmetric])
finally have \( ?S \subseteq (R \cup S)^* \circ O \circ (R \cup S)^* \circ R \)
  by (simp add: relcomp-\( \circ \)-mono)
moreover have \( ?X \subseteq (\circ \text{Image}) \)
  by (auto simp: relcomp-\( \circ \)-Image)
ultimately have \( ?X = \{\} \)
  by (auto elim: wf-\( \text{Image} \)-pf)
moreover have \( A \subseteq ?X \) by auto
ultimately show \( A = \{\} \) by simp
qed

**corollary** \( \text{wf-relcomp-compatible} \):
assumes \( \text{wf } R \) and \( R \circ O \circ S \subseteq S \circ O \circ R \)
shows \( \text{wf } (S \circ O \circ R) \)
proof
  have \( R \circ O \circ S \subseteq (R \cup S)^* \circ O \circ R \)
    using assms by blast
  then have \( \text{wf } (S^* \circ O \circ R \circ S^*) \)
    by (simp add: assms qc-wf-relto-iff)
  then show \( ?\text{thesis} \)
    by (rule Wellfounded.wf-subset) blast
qed

**20.5 Acyclic relations**

**lemma** \( \text{wf-acyclic} : \text{wf } r \Rightarrow \text{acyclic } r \)
  by (simp add: acyclic-def) (blast elim: trancl[THEN \( \text{wf-irrefl} \)])

**lemmas** \( \text{wfP-acyclicP} = \text{wf-acyclic } \{\to-pred\} \)
20.5.1 Wellfoundedness of finite acyclic relations

lemma finite-acyclic-wf:
  assumes finite r acyclic r shows wf r
  using assms
proof (induction r rule: finite-induct)
case (insert x r)
  then show ?case
    by (cases x) simp
qed simp

lemma finite-acyclic-wf-converse: finite r =⇒ acyclic r =⇒ wf (r⁻¹)
apply (erule finite-converse [THEN iffD2, THEN finite-acyclic-wf])
apply (erule acyclic-converse [THEN iffD2])
done

Observe that the converse of an irreflexive, transitive, and finite relation is again well-founded. Thus, we may employ it for well-founded induction.

lemma wf-converse:
  assumes irrefl r and trans r and finite r
  shows wf (r⁻¹)
proof
  have acyclic r
    using ⟨irrefl r⟩ and ⟨trans r⟩
    by (simp add: irrefl-def acyclic-irrefl)
  with ⟨finite r⟩ show ?thesis
    by (rule finite-acyclic-wf-converse)
qed

lemma wf-iff-acyclic-if-finite: finite r =⇒ wf r = acyclic r
by (blast intro: finite-acyclic-wf wf-acyclic)

20.6 nat is well-founded

lemma less-nat-rel: (≤) = (λm n. n = Suc m)++
proof (rule ext, rule ext, rule iffI)
  fix n m :: nat
  show (λm n. n = Suc m)++ m n if m < n
    using that
  proof (induct n)
    case 0
    then show ?case by auto
  next
    case (Suc n)
    then show ?case
      by (auto simp add: less-Suc-eq-le le-less intro: tranclp.trancl-into-trancl)
  qed
  show m < n if (λm n. n = Suc m)++ m n
    using that by (induct n) (simp-all add: less-Suc-eq-le reflexive le-less)
qed
**THEORY “Wellfounded”**

**definition** pred-nat :: (nat × nat) set

where pred-nat = {(m, n). n = Suc m}

**definition** less-than :: (nat × nat) set

where less-than = pred-nat^+

**lemma** less-eq: (m, n) ∈ pred-nat^+ ⟷ m < n

**unfolding** less-nat-rel pred-nat-def trancl-def by simp

**lemma** pred-nat-trancl-eq-le: (m, n) ∈ pred-nat^+ ⟷ m ≤ n

**unfolding** less-eq rtrancl-eq-or-trancl by auto

**lemma** wf-pred-nat: wf pred-nat

apply (unfold wf-def pred-nat-def)

apply clarify

apply (induct-tac x)

apply blast+

done

**lemma** wf-less-than [iff]: wf less-than

by (simp add: less-than-def wf-pred-nat [THEN wf-trancl])

**lemma** trans-less-than [iff]: trans less-than

by (simp add: less-than-def)

**lemma** less-than-iff [iff]: ((x, y) ∈ less-than) = (x < y)

by (simp add: less-than-def less-eq)

**lemma** total-less-than: total less-than

using total-on-def by force

**lemma** wf-less: wf { (x, y::nat). x < y }

by (rule Wellfounded.wellorder-class.wf)

### 20.7 Accessible Part

Inductive definition of the accessible part acc r of a relation; see also [5].

**inductive-set** acc :: (′a × ′a) set ⇒ ′a set for r :: (′a × ′a) set

where accI: (∀y. (y, x) ∈ r ⟹ y ∈ acc r) ⟹ x ∈ acc r

**abbreviation** termip :: (′a ⇒ ′a ⇒ bool) ⇒ ′a ⇒ bool

where termip r ≡ accp (r\(^{-1}-1\))

**abbreviation** termi :: (′a × ′a) set ⇒ ′a set

where termi r ≡ acc (r\(^{-1}\))

**lemmas** accpI = accp.accI
**THEORY “Wellfounded”**

**lemma** accp-eq-acc [code]:

\[
\text{accp } r = (\lambda x. \; x \in \text{Wellfounded.}
acc \; \{ (x, y). \; r \; x \; y \})
\]

by (simp add: acc-def)

**Induction rules**

**theorem** accp-induct:

assumes major: \( \text{accp } r \; a \)

assumes hyp: \( \forall x. \; \text{accp } r \; x \implies \forall y. \; r \; y \; x \implies P \; y \implies P \; x \)

shows \( P \; a \)

apply (rule major [THEN accp.induct])

apply (rule hyp)

apply (rule accp.acci)

apply auto

done

**lemmas** accp-induct-rule = accp-induct [rule-format, induct set: accp]

**theorem** accp-downward: \( \text{accp } r \; b \implies r \; a \; b \implies \text{accp } r \; a \)

by (cases rule: accp.cases)

**lemma** not-accp-down:

assumes \( \text{na} \): \( \neg \; \text{accp } R \; x \)

obtains \( z \) where \( R \; z \; x \) and \( \neg \; \text{accp } R \; z \)

proof –

assume \( a \): \( \forall z. \; R \; z \; x \implies \neg \; \text{accp } R \; z \implies \text{thesis} \)

show thesis

proof (cases \( \forall z. \; R \; z \; x \implies \text{accp } R \; z \))

case True

then have \( \forall z. \; R \; z \; x \implies \text{accp } R \; z \) by auto

then have \( \text{accp } R \; x \) by (rule accp.acci)

with \( \text{na} \) show thesis ..

next

case False then obtain \( z \) where \( R \; z \; x \) and \( \neg \; \text{accp } R \; z \)

by auto

with \( a \) show thesis .

qed

qed

**lemma** accp-downwards-aux: \( r^{**} \; b \; a \implies \text{accp } r \; a \implies \text{accp } r \; b \)

by (erule rtranclp-induct) (blast dest: accp-downward)+

**theorem** accp-downwards: \( \text{accp } r \; a \implies r^{**} \; b \; a \implies \text{accp } r \; b \)

by (blast dest: accp-downwards-aux)

**theorem** accp-wfPI: \( \forall x. \; \text{accp } r \; x \implies \text{wfP } r \)

apply (rule wfPUNIVI)

apply (rule_tac \( P = P \) in accp-induct)

apply blast+

done
theorem accp-wfPD: \( \text{wfP } r \implies \text{accp } r \ x \)
apply (erule wfP-induct-rule)
apply (rule accp.accI)
apply blast
done

theorem wfP-accp-iff: \( \text{wfP } r = (\forall x. \text{accp } r \ x) \)
by (blast intro: accp-wfPI dest: accp-wfPD)

Smaller relations have bigger accessible parts:

lemma accp-subset:
assumes \( R1 \leq R2 \)
shows \( \text{accp } R2 \leq \text{accp } R1 \)
proof (rule predicate1I)
fix \( x \)
assume \( \text{accp } R2 \ x \)
then show \( \text{accp } R1 \ x \)
proof (induct \( x \))
fix \( x \)
assume \( \bigwedge y. \ R2 \ y \ x \Longrightarrow \text{accp } R1 \ y \)
with assms show \( \text{accp } R1 \ x \)
by (blast intro: accp.accI)
qed

qed

This is a generalized induction theorem that works on subsets of the accessible part.

lemma accp-subset-induct:
assumes subset: \( D \leq \text{accp } R \)
and dcl: \( \bigwedge x \ z. D \ x \Longrightarrow R \ z \ x \Longrightarrow D \ z \)
and \( D \ x \)
and istep: \( \bigwedge x. D \ x \Longrightarrow (\bigwedge z. R \ z \ x \Longrightarrow P \ z) \Longrightarrow P \ x \)
sows \( P \ x \)
proof
  from subset and \( (D \ x) \)
have \( \text{accp } R \ x \ .. \)
then show \( P \ x \) using \( (D \ x) \)
proof (induct \( x \))
fix \( x \)
assume \( D \ x \) and \( \bigwedge y. \ R \ y \ x \Longrightarrow D \ y \Longrightarrow P \ y \)
with dcl and istep show \( P \ x \) by blast
qed

qed

Set versions of the above theorems

lemmas acc-induct = accp-induct [to-set]
lemmas acc-induct-rule = acc-induct [rule-format, induct set: acc]
lemmas acc-downward = accp-downward [to-set]
lemmas not-acc-down = not-accp-down [to-set]
lemmas acc-downwards-aux = accp-downwards-aux [to-set]
lemmas acc-downwards = accp-downwards [to-set]
lemmas acc-wfI = accp-wfPI [to-set]
lemmas acc-wfD = accp-wfPD [to-set]
lemmas wf-acc-iff = wfP-accp-iff [to-set]
lemmas acc-subset = accp-subset [to-set]
lemmas acc-subset-induct = accp-subset-induct [to-set]

20.8 Tools for building wellfounded relations

Inverse Image

lemma wf-inv-image [simp,intro]:
  fixes f :: 'a ⇒ 'b
  assumes wf r
  shows wf (inv-image r f)
proof (clarsimp simp: inv-image_def wf-eq-minimal)
  fix P and x::'a
  assume x ∈ P
  then obtain w where w: w ∈ {w. ∃ x::'a. x ∈ P ∧ f x = w} by auto
  have ∗: ∃ Q u. u ∈ Q ⇒ ∃ z ∈ Q. ∀ y. (y, z) ∈ r −→ y /∈ Q
    using assms by (auto simp add: wf-eq-minimal)
  show ∃ z ∈ P. ∀ y. (f y, f z) ∈ r −→ y /∈ P
    using ∗ [OF w] by auto
qed

Measure functions into nat

definition measure :: ('a ⇒ nat) ⇒ ('a × 'a) set
  where measure = inv-image less-than

lemma in-measure[simp, code-unfold]: (x, y) ∈ measure f ⇔ f x < f y
  by (simp add:measure-def)

lemma wf-measure [iff]: wf (measure f)
unfolding measure-def by (rule wf-less-than [THEN wf-inv-image])

lemma wf-if-measure: (∀ x. P x ⇒ f(g x) < f x) ⇒ wf {(y,x). P x ∧ y = g x}
  for f :: 'a ⇒ nat
using wf-measure[of f] unfolding measure-def inv-image-def less-than-def less-eq
  by (rule wf-subset) auto

20.8.1 Lexicographic combinations

definition lex-prod :: ('a × 'a) set ⇒ ('b × 'b) set ⇒ (('a × 'b) × ('a × 'b)) set
  (infixr <*lex*> 80)
  where ra <*lex*> rb = {((a, b), (a', b')). (a, a') ∈ ra ∨ a = a' ∧ (b, b') ∈ rb}
lemma \textit{in-lex-prod}[simp]: 
\begin{align*}
((a, b), (a', b')) \in r \leftrightarrow (a, a') \in r \land a = a' \land (b, b') \in s \\
\text{by (auto simp: lex-prod-def)}
\end{align*}

lemma \textit{wf-lex-prod}[intro!]:
\begin{align*}
\text{assumes \ } \text{wf \ ra \ wf \ rb} \\
\text{shows \ } \text{wf \ (ra \ <\cdot\lex\cdot s)} \\
\text{proof (rule \ wfI)}
\end{align*}
\begin{align*}
\text{fix \ } z :: \ '('a \times 'b) \\
\text{and \ } P \text{ \ assume \ } \forall u. \forall v. (v, u) \in ra \ <\cdot\lex\cdot rb \ \rightarrow \ P v \ \rightarrow \ P u \\
\text{obtain \ } x y \text{ \ where \ } zeq: z = (x, y) \\
\text{by fastforce} \\
\text{have \ } P(x, y) \text{ \ using \ (wf \ ra)} \\
\text{proof (induction \ } x \ \text{arbitrary: \ } y \ \text{rule: \ wf-induct-rule)} \\
\text{\quad case \ (less \ } x) \\
\text{\quad note \ } lessx = less \\
\text{\quad show ?case \ using \ (wf \ rb) \ less} \\
\text{\quad proof (induction \ } y \ \text{rule: \ wf-induct-rule)} \\
\text{\quad \quad case \ (less \ } y) \\
\text{\quad \quad show ?case} \\
\text{\quad \quad \quad by (force \ intro: \ * \ \text{less.IH} \ \text{lessx})} \\
\text{qed} \\
\text{qed} \\
\text{then show \ } P z \\
\text{\quad by (simp \ add: \ zeq)} \\
\text{qed auto}
\end{align*}

\textit{<\cdot\lex\cdot>} \text{ preserves transitivity}

lemma \textit{trans-lex-prod}[simp,intro!]: 
\begin{align*}
\text{trans \ R1 \ \Rightarrow \ \text{trans \ R2 \ \Rightarrow \ trans \ (R1 \ <\cdot\lex\cdot s)} & \\
\text{unfolding \ trans-def \ lex-prod-def \ by \ blast}
\end{align*}

lemma \textit{total-on-lex-prod}[simp]: 
\begin{align*}
\text{total-on \ A \ r \ \Rightarrow \ total-on \ B \ s \ \Rightarrow \ total-on \ (A \times B) \ (R1 <\cdot\lex\cdot s)} & \\
\text{by (auto simp: total-on-def)}
\end{align*}

lemma \textit{total-lex-prod}[simp]: 
\begin{align*}
\text{total \ r \ \Rightarrow \ total \ s \ \Rightarrow \ total \ (r <\cdot\lex\cdot s)} & \\
\text{by (auto simp: total-on-def)}
\end{align*}

lexicographic combinations with measure functions

definition \textit{mlex-prod :: \ ('a \Rightarrow \text{nat}) \Rightarrow \ ('a \times 'a) \ set \Rightarrow \ ('a \times 'a) \ set (infixr <\cdot\mlex\cdot> 80)}:
\begin{align*}
\text{where \ } f <\cdot\mlex\cdot> R = \text{inv-image} \ (\text{less-than} <\cdot\lex\cdot> R) \ (\lambda x. (f x, x))
\end{align*}

lemma \textit{wf-mlex}: 
\begin{align*}
\text{wf \ \ } \text{R1 \ \Rightarrow \ } \text{wf \ } (f <\cdot\mlex\cdot> R) \text{ and} \\
\text{mlex-less: \ } f x < f y \ \Rightarrow \ (x, y) \in f <\cdot\mlex\cdot> R \text{ and} \\
\text{mlex-leq: \ } f x \leq f y \ \Rightarrow \ (x, y) \in R \ \Rightarrow \ (x, y) \in f <\cdot\mlex\cdot> R \text{ and} \\
\text{mlex-iff: \ } (x, y) \in f <\cdot\mlex\cdot> R \ \leftrightarrow \ f x < f y \ \lor \ f x = f y \ \land \ (x, y) \in R
\end{align*}
by (auto simp: mlex-prod-def)

Proper subset relation on finite sets.

definition finite-psubset :: ('a set × 'a set) set
where finite-psubset = {{(A, B). A ⊂ B ∧ finite B}}

lemma wf-finite-psubset[simp]: wf finite-psubset
apply (unfold finite-psubset-def)
apply (rule wf-measure [THEN wf-subset])
apply (simp add: measure-def inv-image-def less-than-def less-eq)
apply (fast elim!: psubset-card-mono)
done

lemma trans-finite-psubset: trans finite-psubset
by (auto simp: finite-psubset-def less-le trans-def)

lemma in-finite-psubset[simp]: (A, B) ∈ finite-psubset ←→ A ⊂ B ∧ finite B
    unfolding finite-psubset-def by auto

max- and min-extension of order to finite sets

inductive-set max-ext :: ('a × 'a) set ⇒ ('a set × 'a set) set
for R :: ('a × 'a) set
where max-extI [intro]:
  finite X =⇒ finite Y =⇒ Y ≠ {} =⇒ (∀x. x ∈ X =⇒ ∃y∈Y. (x, y) ∈ R)
  =⇒ (X, Y) ∈ max-ext R

lemma max-ext-wf:
assumes wf: wf r
shows wf (max-ext r)
proof (rule acc-wfI, intro allI)
  show M ∈ acc (max-ext r) (is - ∈ ?W) for M
  proof (induct M rule: infinite-finite-induct)
    case empty
    show ?case
    proof
      rule accI (auto elim: max-ext.cases)
    next
    case insert a M
    from wf ⟨M ∈ ?W⟩ (finite M) show insert a M ∈ ?W
    proof (induct arbitrary: M)
      fix M a
      assume M ∈ ?W
      assume [intro]: finite M
      assume hyp: ∀b M. (b, a) ∈ r ⇒ M ∈ ?W ⇒ finite M → insert b M ∈ ?W
      have add-less: M ∈ ?W ⇒ (∀y ∈ N ⇒ (y, a) ∈ r) ⇒ N ∪ M ∈ ?W
        if finite N finite M for N M :: 'a set
        using that by (induct N arbitrary: M) (auto simp: hyp)
      show insert a M ∈ ?W
      proof
        rule accI
      end
    end
  end
end
THEORY “Wellfounded”

fix \( N \)
assume \( N \lesscolon (N, \text{insert } a M) \in \text{max-ext } r \)
then have \( * \colon \forall x. x \in N \Rightarrow (x, a) \in r \lor (\exists y \in M. (x, y) \in r) \)
  by (auto elim!: max-ext.cases)

let \(?N1 = \{n \in N. (n, a) \in r\} \)
let \(?N2 = \{n \in N. (n, a) \not\in r\} \)
have \( N \colon ?N1 \cup ?N2 = N \) by (rule set-eqI) auto

from \( N \lesscolon \text{finite } N \) by (auto elim: max-ext.cases)
then have finites: \( \text{finite } ?N1 \text{ finite } ?N2 \) by auto

have \( ?N2 \in ?W \)
proof (cases \( M = \{\} \))
  case \[ simp \]: True
  have \( Mw \colon \{\} \in ?W \) by (rule accI) (auto elim: max-ext.cases)
  from \( * \) have \( ?N2 = \{\} \) by auto
  with \( Mw \) show \( ?N2 \in ?W \) by (simp only:)
next
  case False
  from \( * \) finites have \( N2 \colon (\exists ?N2 \in ?W) \) by (rule acc-downward)
  with \( Mw \) show \( ?N2 \in ?W \) by (simp only: N)
qed

with finites have \( ?N1 \cup ?N2 \in ?W \)
  by (rule add-less simp)
then show \( N \in ?W \) by (simp only: N)
qed

qed

next
  case \[ infinite \]
show \[ \cases \]
  by (rule accI) (auto elim: max-ext.cases simp: infinite)
qed

qed

lemma \( \text{max-ext-additive} \colon (A, B) \in \text{max-ext } R \Rightarrow (C, D) \in \text{max-ext } R \Rightarrow (A \cup C, B \cup D) \in \text{max-ext } R \)
  by (force elim!: max-ext.cases)

definition \( \text{min-ext} \colon \text{('a } \times \text{'a set) set } \Rightarrow (\text{'a set } \times \text{'a set) set} \)
where \( \text{min-ext } r = \{ (X, Y) \mid X \neq \{\} \land \forall y \in Y. (\exists x \in X. (x, y) \in r)\} \)

lemma \( \text{min-ext-wf} \)
assumes \( \text{wf } r \)
shows \( \text{wf } (\text{min-ext } r) \)
proof (rule wfI-min)
show \( \exists m \in Q. (\forall n. (n, m) \in \text{min-ext } r \Rightarrow n \not\in Q) \) if \( \text{nonempty } x \in Q \)
  for \( Q \colon \text{'a set set and } x \)
proof (cases \( Q = \{\}\) )
case True
then show ?thesis by (simp add: min-ext-def)
next
case False
with nonempty obtain e x where x ∈ Q e ∈ x by force
then have eU: e ∈ ∪ Q by auto
with (wf r)
obtain z where z ∈ ∪ Q ∧ y. (y, z) ∈ r =⇒ y ∉ ∪ Q
by (erule wfE-min)
from z obtain m where m ∈ Q z ∈ m by auto
from ⟨m ∈ Q⟩ obtain z where z ∈ ∪ Q ∧ y. (y, z) ∈ r =⇒ y ∉ ∪ Q
by (erule wfE-min)
from z obtain m where m ∈ Q z ∈ m by auto
from ⟨m ∈ Q⟩ show ?thesis
proof (intro rev-bexI allI impI)
  fix n
  assume smaller: (n, m) ∈ min-ext r
  with ⟨z ∈ m⟩ obtain y where y ∈ n (y, z) ∈ r
  by (auto simp: min-ext-def)
  with z(2) show n ∉ Q by auto
qed
qed

20.8.2 Bounded increase must terminate

lemma wf-bounded-measure:
  fixes ub :: 'a ⇒ nat
  and f :: 'a ⇒ nat
  assumes ∨ a. b. (b, a) ∈ r =⇒ ub b ≤ ub a ∧ ub a ≥ f b ∧ f b > f a
  shows wf r
by (rule wf-subset[OF wf-measure[of λ a. ub a − f a]]) (auto dest: assms)

lemma wf-bounded-set:
  fixes ub :: 'a ⇒ 'b set
  and f :: 'a ⇒ 'b set
  assumes ∨ a. b. (b,a) ∈ r =⇒ finite (ub a) ∧ ub b ⊆ ub a ∧ ub a ⊇ f b ∧ f b ⊇ f a
  shows wf r
apply (rule wf-bounded-measure[of r λa. card (ub a) λa. card (f a)])
apply (drule assms)
apply (blast intro: card-mono finite-subset psubset-card-mono dest: psubset-eq[THEN iffD2])
done

lemma finite-subset-wf:
  assumes finite A
  shows wf {(X, Y). X ⊆ Y ∧ Y ⊆ A}
by (rule wf-subset[OF wf-finite-psubset[unfolded finite-psubset-def]])
  (auto intro: finite-subset[OF - assms])

hide-const (open) acc accp
21 Well-Founded Recursion Combinator

theory Wfrec
  imports Wellfounded
begin

inductive wfrec-rel :: ('a × 'a) set ⇒ (('a ⇒ 'b) ⇒ ('a ⇒ 'b)) ⇒ 'a ⇒ 'b ⇒ bool
  for R F
  where wfrecI: (⋀ z. (z, x) ∈ R ⇒ wfrec-rel R F z (g z)) ⇒ wfrec-rel R F x (F g x)

definition cut :: ('a ⇒ 'b) ⇒ ('a × 'a) set ⇒ 'a ⇒ 'a ⇒ 'b
  where cut f R x = (λ y. if (y, x) ∈ R then f y else undefined)

definition adm-wf :: ('a × 'a) set ⇒ (('a ⇒ 'b) ⇒ ('a ⇒ 'b)) ⇒ bool
  where adm-wf R F ←→ (∀ f g x. (∀ z. (z, x) ∈ R −→ f z = g z) −→ F f x = F g x)

definition wfrec :: ('a × 'a) set ⇒ (('a ⇒ 'b) ⇒ ('a ⇒ 'b)) ⇒ ('a ⇒ 'b)
  where wfrec R F = (λ x. THE y. wfrec-rel R (λ f x. F (cut f R x)) x y)

lemma cuts-eq: (cut f R x = cut g R x) ←→ (∀ y. (y, x) ∈ R −→ f y = g y)
  by (simp add: fun-eq-iff cut-def)

lemma cut-apply: (x, a) ∈ R −→ cut f R a x = f x
  by (simp add: cut-def)

Inductive characterization of wfrec combinator; for details see: John Harrison, "Inductive definitions: automation and application".

lemma theI-unique: ∃! x. P x −→ P x −→ x = The P
  by (auto intro: the-equality[symmetric] theI)

lemma wfrec-unique:
  assumes adm-wf R F wf R
  shows ∃! y. wfrec-rel R F x y
  using wf R;
  proof induct
    define f where f y = (THE z. wfrec-rel R F y z) for y
    case (less x)
    then have ℧ y z. (y, x) ∈ R −→ wfrec-rel R F y z −→ z = f y
      unfolding f-def by (rule theI-unique)
    with ⟨adm-wf R F⟩ show ?case
      by (subst wfrec-rel.simps) (auto simp: adm-wf-def)
  qed

lemma adm-lemma: adm-wf R (λ f x. F (cut f R x)) x
by (auto simp: adm-wf-def intro: arg-cong [where f=λx. F x y for y] cuts-eq [THEN iffD2])

**Lemma** \( \text{wfrec} \): \( \text{wf } R \implies \text{wfrec } R \ F \ a = F \ (\text{cut } (\text{wfrec } R \ F) \ R \ a) \ a \)
apply (simp add: wfrec-def)
apply (rule adm-lemma [THEN wfrec-unique, THEN the1-equality])
apply assumption
apply (rule wfrec-rel wfrecI)
apply (erule adm-lemma [THEN wfrec-unique, THEN theI'])
done

This form avoids giant explosions in proofs. NOTE USE OF \( \equiv \).

**Lemma** \( \text{def-wfrec} \): \( f \equiv \text{wfrec } R \ F \ a \implies \text{wf } R \implies f \ a = F \ (\text{cut } f \ R \ a) \ a \)
by (auto intro: wfrec)

21.0.1 Well-founded recursion via genuine fixpoints

**Lemma** \( \text{wfrec-fixpoint} \):
assumes \( \text{wf } R \)
and \( \text{adm } \equiv \text{adm-wf } R \ F \)
shows \( \text{wfrec } R \ F = F \ (\text{wfrec } R \ F) \)
proof (rule ext)
fix \( x \)
have \( \text{wfrec } R \ F \ x = F \ (\text{cut } (\text{wfrec } R \ F) \ R \ x) \ x \)
using \( \text{wfrec} [\text{of } R \ F] \) \( \text{wf } \) by simp
also
have \( \forall y. (y, x) \in R \implies \text{cut } (\text{wfrec } R \ F) \ R \ x \ y = \text{wfrec } R \ F \ y \)
by (auto simp add: cut-apply)
then have \( F \ (\text{cut } (\text{wfrec } R \ F) \ R \ x) \ x = F \ (\text{wfrec } R \ F) \ x \)
using \( \text{adm } \) \( \text{adm-wf-def } [\text{of } R \ F] \) by auto
finally show \( \text{wfrec } R \ F \ x = F \ (\text{wfrec } R \ F) \ x \).
qed

21.1 Wellfoundedness of same-fst

**Definition** \( \text{same-fst} \) :: \( \text{('a ⇒ bool) ⇒ ('a ⇒ ('b × 'b) set) ⇒ (('a × 'b) × ('a × 'b)) set} \)
where \( \text{same-fst } P \ R = \{(x', y'), (x, y) \}. x' = x \land P x \land (y', y) \in R x \} \)
— For \( \text{wfrec} \) declarations where the first \( n \) parameters stay unchanged in the recursive call.

**Lemma** \( \text{same-fstI} \) [intro!]: \( P \ x \implies (y', y) \in R x \implies ((x', y'), (x, y)) \in \text{same-fst } P \ R \)
by (simp add: same-fst-def)

**Lemma** \( \text{wf-same-fst} \):
assumes \( \forall x. P \ x \implies \text{wf } (R x) \)
shows \( \text{wf } (\text{same-fst } P \ R) \)
proof (clarsimp simp add: wf-def same-fst-def)
fix \( Q \ a \ b \)
assume \( \forall x. (\forall a \in R \ a \rightarrow Q \ (a,x)) \rightarrow Q \ (a,b) \)
show \( Q(a,b) \)
proof (cases \( \text{wf} \ (R \ a) \))
case \( \text{True} \)
  then show \( \text{thesis} \)
  by (induction \( b \) rule: \( \text{wf-induct-rule} \)) (use \( \ast \) in \text{blast})
qed (use \( \ast \) in \text{blast})
qed

declaration of order-Relation

22 Orders as Relations

theory Order-Relation
imports Wfrec
begin

22.1 Orders on a set

definition preorder-on A r \( \equiv \) refl-on A r \( \land \) trans r

definition partial-order-on A r \( \equiv \) preorder-on A r \( \land \) antisym r

definition linear-order-on A r \( \equiv \) partial-order-on A r \( \land \) total-on A r

definition strict-linear-order-on A r \( \equiv \) trans r \( \land \) irrefl r \( \land \) total-on A r

definition well-order-on A r \( \equiv \) linear-order-on A r \( \land \) \( \text{wf} \ (r - \text{Id}) \)

lemmas order-on-defs =
  preorder-on-def partial-order-on-def linear-order-on-def
  strict-linear-order-on-def well-order-on-def

lemma partial-order-onD:
  assumes partial-order-on A r shows refl-on A r and trans r and antisym r
  using assms unfolding partial-order-on-def preorder-on-def by auto

lemma preorder-on-empty[simp]: preorder-on \{\} \{\}
  by (simp add: preorder-on-def trans-def)

lemma partial-order-on-empty[simp]: partial-order-on \{\} \{\}
  by (simp add: partial-order-on-def)

lemma linear-order-on-empty[simp]: linear-order-on \{\} \{\}
  by (simp add: linear-order-on-def)

lemma well-order-on-empty[simp]: well-order-on \{\} \{\}
  by (simp add: well-order-on-def)
lemma preorder-on-converse[simp]: preorder-on A \((r^{-1})\) = preorder-on A r
by (simp add: preorder-on-def)

lemma partial-order-on-converse[simp]: partial-order-on A \((r^{-1})\) = partial-order-on A r
by (simp add: partial-order-on-def)

lemma linear-order-on-converse[simp]: linear-order-on A \((r^{-1})\) = linear-order-on A r
by (simp add: linear-order-on-def)

lemma partial-order-on-acyclic:
  partial-order-on A r \(\Rightarrow\) acyclic \((r - \text{Id})\)
by (simp add: acyclic-irrefl partial-order-on-def preorder-on-def trans-diff-Id)

lemma partial-order-on-well-order-on:
  finite r \(\Rightarrow\) partial-order-on A r \(\Rightarrow\) wf \((r - \text{Id})\)
by (simp add: finite-acyclic-wf partial-order-on-acyclic)

lemma strict-linear-order-on-diff-Id: linear-order-on A r \(\Rightarrow\) strict-linear-order-on A \((r - \text{Id})\)
by (simp add: order-on-defs trans-diff-Id)

lemma linear-order-on-singleton [simp]: linear-order-on \(\{x\}\) \(\{\{x, x\}\}\)
by (simp add: order-on-defs)

lemma linear-order-on-acyclic:
  assumes linear-order-on A r
  shows acyclic \((r - \text{Id})\)
  using strict-linear-order-on-diff-Id[OF assms]
by (auto simp add: acyclic-irrefl strict-linear-order-on-def)

lemma linear-order-on-well-order-on:
  assumes finite r
  shows linear-order-on A r \(\iff\) well-order-on A r
  unfolding well-order-on-def
  using assms finite-acyclic-wf[OF - linear-order-on-acyclic, of r] by blast

22.2 Orders on the field

abbreviation \(\text{Refl} r \equiv \text{refl-on} (\text{Field} r) r\)

abbreviation \(\text{Preorder} r \equiv \text{preorder-on} (\text{Field} r) r\)

abbreviation \(\text{Partial-order} r \equiv \text{partial-order-on} (\text{Field} r) r\)
**THEORY “Order-Relation”**

abbreviation **Total** \( r \equiv \text{total-on} \ (\text{Field} \ r) \ r \)

abbreviation **Linear-order** \( r \equiv \text{linear-order-on} \ (\text{Field} \ r) \ r \)

abbreviation **Well-order** \( r \equiv \text{well-order-on} \ (\text{Field} \ r) \ r \)

**lemma** **subset-Image-Image-iff**:
\[
\begin{align*}
\text{Preorder } r & \implies A \subseteq \text{Field} \ r \implies B \subseteq \text{Field} \ r \implies \\
r \ `` A \subseteq r " " B & \iff (\forall a \in A. \exists b \in B. \ (b, a) \in r)
\end{align*}
\]
apply (simp add: preorder-on-def refl-on-def Image-def subset-eq)
apply (simp only: trans-def)
apply fast
done

**lemma** **subset-Image1-Image1-iff**:
\[
\begin{align*}
\text{Preorder } r & \implies a \in \text{Field} \ r \implies b \in \text{Field} \ r \implies r \ `` \{a\} \subseteq r \ " \{b\} \longleftrightarrow (b, a) \\
& \in r
\end{align*}
\]
by (simp add: subset-Image-Image-iff)

**lemma** **Refl-antisym-eq-Image1-Image1-iff**:
assumes **Refl** \( r \)
and as: **antisym** \( r \)
and abf: \( a \in \text{Field} \ r \) \( b \in \text{Field} \ r \)
shows \( r \ " \{a\} = r \ " \{b\} \longleftrightarrow a = b \)
(is \ ?lhs \longleftrightarrow \ ?rhs)
proof
assume \ ?lhs
then have \(?\): \( \forall x. \ (a, x) \in r \longleftrightarrow (b, x) \in r \)
by (simp add: set-eq-iff)
have \((a, a) \in r (b, b) \in r \) using \(\text{Refl } r \) abf by (simp-all add: refl-on-def)
then have \((a, b) \in r (b, a) \in r \) using \(\text{of } a \) \(\text{of } b \) by simp-all
then show \ ?rhs
  using \(\text{of } \text{antisym } r \)\text{[unfolded antisym-def]} by blast
next
assume \ ?rhs
then show \ ?lhs by fast
qed

**lemma** **Partial-order-eq-Image1-Image1-iff**:
\[
\begin{align*}
\text{Partial-order } r & \implies a \in \text{Field} \ r \implies b \in \text{Field} \ r \implies r \ " \{a\} = r \ " \{b\} \longleftrightarrow a = b \\
& = b
\end{align*}
\]
by (auto simp: order-on-defs Refl-antisym-eq-Image1-Image1-iff)

**lemma** **Total-Id-Field**:
assumes **Total** \( r \)
and not-Id: \( \neg r \subseteq Id \)
shows \( \text{Field} \ r = \text{Field} \ (r - Id) \)
using mono-Field[of \( r - Id \ r \) \ Diff-subset[of \ r \ Id]}

proof auto
  fix a assume "a ∈ Field r"
  from not-Id have "r ≠ {}" by fast
  with not-Id obtain b and c where "b ≠ c ∧ (b, c) ∈ r" by auto
  then have "b ≠ c ∧ (b, c) ⊆ Field r" by (auto simp: Field-def)
  with * obtain d where "d ∈ Field r ∧ d ≠ a" by auto
  with * obtain d where "d ∈ Field r ∧ d ≠ a" by (simp add: total-on-def)
  with ⟨d ≠ a⟩ show "a ∈ Field (r - Id)" unfolding Field-def by blast
qed

22.3 Relations given by a predicate and the field

definition relation-of :: "('a ⇒ 'a ⇒ bool) ⇒ 'a set ⇒ ('a × 'a) set"
where relation-of P A ≡ \{(a, b) ∈ A × A. P a b\}

lemma Field-relation-of:
  assumes refl-on A (relation-of P A) shows Field (relation-of P A) = A
  using assms unfolding refl-on-def relation-of-def by auto

lemma partial-order-on-relation-off:
  assumes refl: \(\forall a. a ∈ A \Rightarrow P a a\)
  and trans: \(\forall a b c. (a ∈ A; b ∈ A; c ∈ A) \Rightarrow P a b \Rightarrow P b c \Rightarrow P a c\)
  and antisym: \(\forall a b. (a ∈ A; b ∈ A) \Rightarrow P a b \Rightarrow P b a \Rightarrow a = b\)
  shows partial-order-on A (relation-of P A)
proof –
  from refl have refl-on A (relation-of P A)
    unfolding refl-on-def relation-of-def by auto
  moreover have trans (relation-of P A) and antisym (relation-of P A)
  ultimately show ?thesis
    unfolding partial-order-on-def preorder-on-def by simp
qed

lemma Partial-order-relation-off:
  assumes partial-order-on A (relation-of P A) shows Partial-order (relation-of P A)
  using Field-relation-of assms partial-order-on-def preorder-on-def by fastforce

22.4 Orders on a type

abbreviation strict-linear-order ≡ strict-linear-order-on UNIV

abbreviation linear-order ≡ linear-order-on UNIV

abbreviation well-order ≡ well-order-on UNIV
22.5 Order-like relations

In this subsection, we develop basic concepts and results pertaining to order-like relations, i.e., to reflexive and/or transitive and/or symmetric and/or total relations. We also further define upper and lower bounds operators.

22.5.1 Auxiliaries

**Lemma** refl-on-domain: refl-on \( A \times r \implies (a, b) \in r \implies a \in A \land b \in A \)

by (auto simp add: refl-on-def)

**Corollary** well-order-on-domain: well-order-on \( A \times r \implies (a, b) \in r \implies a \in A \land b \in A \)

by (auto simp add: refl-on-domain order-on-defs)

**Lemma** well-order-on-Field: well-order-on \( A \times r \implies A = \text{Field } r \)

by (auto simp add: refl-on-def Field-def order-on-defs)

**Lemma** well-order-on-Well-order: well-order-on \( A \times r \implies A = \text{Field } r \land \text{Well-order } r \)

using well-order-on-Field \([of A]\) by auto

**Lemma** Total-subset-Id:

assumes Total r

and \( r \subseteq Id \)

shows \( r = \{\} \lor (\exists a. r = \{(a, a)\}) \)

proof –

have \( \exists a. r = \{(a, a)\} \) if \( r \neq \{\} \)

proof –

from that obtain \( a \ b \) where \( (a, b) \in r \) by fast

with \( r \subseteq Id \) have \( a = b \) by blast

with \( ab \) have \( aa: (a, a) \in r \) by simp

have \( a = c \land a = d \) if \( (c, d) \in r \) for \( c \ d \)

proof –

from that have \( \{a, c, d\} \subseteq \text{Field } r \)

using \( ab \) unfolding Field-def by blast

then have \( ((a, c) \in r \lor (c, a) \in r \lor a = c) \land ((a, d) \in r \lor (d, a) \in r \lor a = d) \)

using \( \text{Total } r \) unfolding total-on-def by blast

with \( (r \subseteq Id) \) show ?thesis by blast

qed

then have \( r \subseteq \{\{a, a\}\} \) by auto

with \( aa \) show ?thesis by blast

qed

then show ?thesis by blast

qed

**Lemma** Linear-order-in-diff-Id:

assumes Linear-order \( r \)
and $a \in \text{Field } r$
and $b \in \text{Field } r$
shows $(a, b) \in r \leftrightarrow (b, a) \notin r - \text{Id}$
using assms unfolding order-on-defs total-on-def antisym-def Id-def refl-on-def
by force

22.5.2 The upper and lower bounds operators

Here we define upper ("above") and lower ("below") bounds operators. We think of $r$ as a non-strict relation. The suffix $S$ at the names of some operators indicates that the bounds are strict – e.g., under$S$ a is the set of all strict lower bounds of $a$ (w.r.t. $r$). Capitalization of the first letter in the name reminds that the operator acts on sets, rather than on individual elements.

definition under :: `'a rel ⇒ 'a ⇒ 'a set
  where under r a ≡ \{b. (b, a) ∈ r\}

definition underS :: `'a rel ⇒ 'a ⇒ 'a set
  where underS r a ≡ \{b. b ≠ a ∧ (b, a) ∈ r\}

definition Under :: `'a rel ⇒ 'a set ⇒ 'a set
  where Under r A ≡ \{b ∈ Field r. ∀ a ∈ A. (b, a) ∈ r\}

definition UnderS :: `'a rel ⇒ 'a set ⇒ 'a set
  where UnderS r A ≡ \{b ∈ Field r. ∀ a ∈ A. b ≠ a ∧ (b, a) ∈ r\}

definition above :: `'a rel ⇒ 'a ⇒ 'a set
  where above r a ≡ \{b. (a, b) ∈ r\}

definition aboveS :: `'a rel ⇒ 'a ⇒ 'a set
  where aboveS r a ≡ \{b. b ≠ a ∧ (a, b) ∈ r\}

definition Above :: `'a rel ⇒ 'a set ⇒ 'a set
  where Above r A ≡ \{b ∈ Field r. ∀ a ∈ A. (a, b) ∈ r\}

definition AboveS :: `'a rel ⇒ 'a set ⇒ 'a set
  where AboveS r A ≡ \{b ∈ Field r. ∀ a ∈ A. b ≠ a ∧ (a, b) ∈ r\}

definition ofilter :: `'a rel ⇒ 'a set ⇒ bool
  where ofilter r A ≡ A ⊆ Field r ∧ (∀ a ∈ A. under r a ⊆ A)

Note: In the definitions of Above$S$ and Under$S$, we bounded comprehension by Field $r$ in order to properly cover the case of $A$ being empty.

lemma underS-subset-under: underS r a ⊆ under r a
  by (auto simp add: underS-def under-def)

lemma underS-notIn: $a \notin$ under$S$ r a
  by (simp add: underS-def)
lemma Refl-under-in: Refl r ⟷ a ∈ Field r ⟷ a ∈ under r a
  by (simp add: refl-on-def under-def)

lemma AboveS-disjoint: A ∩ (AboveS r A) = {}
  by (auto simp add: AboveS-def)

lemma in-AboveS-underS: a ∈ Field r ⟷ a ∈ AboveS r (underS r a)
  by (auto simp add: AboveS-def underS-def)

lemma Refl-under-underS: Refl r ⟷ a ∈ Field r ⟷ under r a = underS r a ∪ {a}
  unfolding under-def underS-def
  using refl-on-def[of - r] by fastforce

lemma underS-empty: a /∈ Field r ⟷ underS r a = {}
  by (auto simp: Field-def underS-def)

lemma under-Field: under r a ⊆ Field r
  by (auto simp: under-def Field-def)

lemma underS-Field: underS r a ⊆ Field r
  by (auto simp: underS-def Field-def)

lemma underS-Field2: a ∈ Field r ⟷ underS r a ⊆ Field r
  using underS-notIn underS-Field by fast

lemma underS-Field3: Field r ≠ {} ⟷ underS r a ⊆ Field r
  by (cases a ∈ Field r) (auto simp: underS-Field2 underS-empty)

lemma AboveS-Field: AboveS r A ⊆ Field r
  by (auto simp: AboveS-def Field-def)

lemma under-incr:
  assumes trans r
  and (a, b) ∈ r
  shows under r a ⊆ under r b
  unfolding under-def
  proof auto
    fix x assume (x, a) ∈ r
    with assms trans-def[of r] show (x, b) ∈ r by blast
  qed

lemma underS-incr:
  assumes trans r
  and antisym r
  and ab: (a, b) ∈ r
  shows underS r a ⊆ underS r b
  unfolding underS-def
proof auto
  assume *: b ≠ a and **: (b, a) ∈ r
  with ⟨antisym r⟩ antisym-def[of r] ab show False
    by blast
next
  fix x assume x ≠ a (x, a) ∈ r
  with ab ⟨trans r⟩ trans-def[of r] show (x, b) ∈ r
    by blast
qed

lemma underS-incl-iff:
  assumes LO: Linear-order r
    and INa: a ∈ Field r
    and INb: b ∈ Field r
  shows underS r a ⊆ underS r b ⟷ (a, b) ∈ r
    (is ?lhs ⟷ ?rhs)
proof
  assume ?rhs
  with ⟨Linear-order r⟩ show ?lhs
    by (simp add: order-on-defs underS-incr)
next
  assume *: ?lhs
  have (a, b) ∈ r if a = b
    using assms that by (simp add: order-on-defs refl-on-def)
  moreover have False if a ≠ b (b, a) ∈ r
    proof
      from that have b ∈ underS r a unfolding underS-def by blast
      with * have b ∈ underS r b by blast
      then show ?thesis by (simp add: underS-notIn)
    qed
  ultimately show (a, b) ∈ r
    using assms order-on-defs[of Field r r] total-on-def[of Field r r] by blast
qed

lemma finite-Partial-order-induct[consumes 3, case-names step]:
  assumes Partial-order r
    and x ∈ Field r
    and finite r
    and step: ∀x. x ∈ Field r ⟹ (∀y. y ∈ aboveS r x ⟹ P y) ⟹ P x
  shows P x
  using assms(2)
proof (induct rule: wf-induct[of r⁻¹ − Id])
  case 1
  from assms(1,3) show wf (r⁻¹ − Id)
    using partial-order-on-well-order-on partial-order-on-converse by blast
next
  case prems: (2 x)
  show ?case
    by (rule step) (use prems in (auto simp: aboveS-def intro: FieldI2))
lemma finite-Linear-order-induct[consumes 3, case-names step]:
  assumes Linear-order r
  and x ∈ Field r
  and finite r
  and step: (∀x. x ∈ Field r → (∀y. y ∈ aboveS r x → P y) → P x)
  shows P x
  using assms(2)
proof (induct rule: wf-induct[of r −1 − Id])
  case 1
  from assms(1,3) show wf (r −1 − Id)
    using linear-order-on-well-order-on linear-order-on-converse
    unfolding well-order-on-def by blast
next
  case prems: (2 x)
  show ?case by (rule step) (use prems in ⟨auto simp: aboveS-def intro: FieldI2⟩)
qed

22.6 Variations on Well-Founded Relations

This subsection contains some variations of the results from HOL.Wellfounded:

- means for slightly more direct definitions by well-founded recursion;
- variations of well-founded induction;
- means for proving a linear order to be a well-order.

22.6.1 Characterizations of well-foundedness

A transitive relation is well-founded iff it is “locally” well-founded, i.e., iff
its restriction to the lower bounds of of any element is well-founded.

lemma trans-wf-iff:
  assumes trans r
  shows (wf r ←→ (∀a. wf (r ∩ (r −1 × {a} × r −1 × {a}))))
proof –
  define R where R a = r ∩ (r −1 × {a} × r −1 × {a}) for a
  have wf (R a) if wf r for a
    using that R-def wf-subset[of r R a] by auto
  moreover
  have wf r if *: ∀a. wf(R a)
    unfolding wf-def
  proof clarify
    fix phi a
    assume **: ∀a. (∀b. (b, a) ∈ r → phi b) → phi a
define \( \text{chi} \) where \( \text{chi} b \iff (b, a) \in r \rightarrow \text{phi} b \) for \( b \)

with \( * \) have \( \text{wf} (R a) \) by \( \text{auto} \)

then have \((\forall b. (\forall c. (c, b) \in R a \rightarrow \text{chi} c) \rightarrow \text{chi} b) \rightarrow (\forall b. \text{chi} b)\)

unfolding \( \text{wf-def} \) by \( \text{blast} \)

also have \((\forall b. (\forall c. (c, b) \in R a \rightarrow \text{chi} c) \rightarrow \text{chi} b)\)

proof \((\text{auto simp add: chi-def R-def})\)

fix \( b \)

assume \((b, a) \in r \) and \((\forall c. (c, b) \in r \land (c, a) \in r \rightarrow \text{phi} c)\)

then have \((\forall c. (c, b) \in r \rightarrow \text{phi} c)\)

using assms trans-def \([\text{of } r]\) by \( \text{blast} \)

with \( * * \) show \( \text{phi} b \) by \( \text{blast} \)

qed

finally have \((\forall b. \text{chi} b)\)

with \( * * \) \( \text{chi-def} \) show \( \text{phi} a \) by \( \text{blast} \)

qed

ultimately show \( ?\text{thesis unfolding } R\text{-def} \) by \( \text{blast} \)

qed

A transitive relation is well-founded if all initial segments are finite.

corollary \( \text{wf-finite-segments}\):

assumes \( \text{irrefl } r \) and \( \text{trans } r \)

\( \land \forall x. \text{finite } \{ y. (y, x) \in r \} \)

shows \( \text{wf } (r) \)

proof \((\text{clarsimp simp: trans-wf-iff wf-iff-acyclic-if-finite converse-def assms})\)

fix \( a \)

have \( \text{trans } (r \cap (\{ x. (x, a) \in r \} \times \{ x. (x, a) \in r \}))\)

using assms unfolding \( \text{trans-def Field-def} \) by \( \text{blast} \)

then show \( \text{acyclic } (r \cap \{ x. (x, a) \in r \} \times \{ x. (x, a) \in r \})\)

using assms acyclic-def assms \( \text{irrefl-def} \) by \( \text{fastforce} \)

qed

The next lemma is a variation of \( \text{wf-eq-minimal}\) from \( \text{Wellfounded} \), allowing one to assume the set included in the field.

lemma \( \text{wf-eq-minimal2}\): \( \text{wf } r \iff (\forall A. A \subseteq \text{Field } r \land A \neq \{\} \rightarrow (\exists a \in A. \forall a' \in A. (a', a) \notin r))\)

proof

let \( ?\text{phi} = \lambda A. A \neq \{\} \rightarrow (\exists a \in A. \forall a' \in A. (a', a) \notin r)\)

have \( \text{wf } r \iff (\forall A. ?\text{phi } A)\)

apply \( (\text{auto simp: ex-in-conv [THEN symp])}\)

apply \( (\text{erule wfE-min})\)

apply assumption

apply \( \text{blast} \)

apply \( (\text{rule wfI-min})\)

apply \( \text{fast} \)

done

also have \((\forall A. ?\text{phi } A) \iff (\forall B \subseteq \text{Field } r. ?\text{phi } B)\)

proof

assume \( \forall A. ?\text{phi } A\)

then show \( \forall B \subseteq \text{Field } r. ?\text{phi } B \) by \( \text{simp} \)

next
assume $\forall B \subseteq \text{Field } r$. $\phi B$

show $\forall A. \phi A$

proof clarify

fix $A :: 'a set$

assume $\ast \ast : A \neq \emptyset$

define $B$ where $B = A \cap \text{Field } r$

show $\exists a \in A. \forall a' \in A. (a', a) \notin r$

proof (cases $B = \emptyset$)

  case True
  
  with $\ast \ast$ obtain $a$ where $a \in A$ $a \notin \text{Field } r$
  
  unfolding $B$-def by blast
  
  with $a$ have $\forall a' \in A. (a', a) \notin r$
  
  unfolding $\text{Field}$-def by blast
  
  with $a$ show $\phi$thesis by blast

next

  case False

  have $B \subseteq \text{Field } r$ unfolding $B$-def by blast
  
  with False $\ast$ obtain $a$ where $a \in B \forall a' \in B$. $(a', a) \notin r$
  
  by blast
  
  have $(a', a) \notin r$ if $a' \in A$ for $a'$

proof

  assume $a'a$: $(a', a) \in r$
  
  with that have $a' \in B$ unfolding $\text{B-def Field}$-def by blast
  
  with $a'a$ show False by blast

qed

with $a$ show $\phi$thesis unfolding $B$-def by blast

qed

qed

finally show $\phi$thesis by blast

qed

22.6.2 Characterizations of well-foundedness

The next lemma and its corollary enable one to prove that a linear order is a well-order in a way which is more standard than via well-foundedness of the strict version of the relation.

lemma Linear-order-wf-diff-Id:

assumes Linear-order $r$

shows $\text{wf } (r - \text{Id}) \iff (\forall A \subseteq \text{Field } r. A \neq \emptyset \longrightarrow (\exists a \in A. \forall a' \in A. (a, a') \in r))$

proof (cases $r \subseteq \text{Id}$)

  case True
  
  then have $\ast : r - \text{Id} = \emptyset$ by blast
  
  have $\text{wf } (r - \text{Id})$ by (simp add: $\ast$)
  
  moreover have $\exists a \in A. \forall a' \in A. (a, a') \in r$
  
  if $\ast: A \subseteq \text{Field } r$ and $\ast \ast : A \neq \emptyset$ for $A$

proof

  from $\text{Linear-order } r$ True
obtain a where a: r = {} ∨ r = {{a, a}}
unfolding order-on-defs using Total-subset-Id [of r] by blast
with * ** have A = {a} ∧ r = {(a, a)}
unfolding Field-def by blast
with a show ?thesis by blast
qed
ultimately show ?thesis by blast
next
case False
with ⟨Linear-order r⟩ have Field: Field r = Field (r − Id)
unfolding order-on-defs using Total-Id-Field [of r] by blast
show ?thesis
proof
assume *: wf (r − Id)
show ∀A ⊆ Field r. A ≠ {} → (∃a ∈ A. ∀a′ ∈ A. (a, a′) ∈ r)
proof clarify
fix A
assume **: A ⊆ Field r and ***: A ≠ {} then have ∃a ∈ A. ∀a′ ∈ A. (a′, a) ∉ r − Id
using Field * unfolding wf-eq-minimal2 by simp
moreover have ∀a ∈ A. ∀a′ ∈ A. (a, a′) ∈ r ↔ (a′, a) ∉ r − Id
using Linear-order-in-diff-Id [OF ⟨Linear-order r⟩] ** by blast
ultimately show ∃a ∈ A. ∀a′ ∈ A. (a, a′) ∈ r by blast
qed
next
assume *: ∀A ⊆ Field r. A ≠ {} → (∃a ∈ A. ∀a′ ∈ A. (a, a′) ∈ r)
show wf (r − Id)
unfolding wf-eq-minimal2
proof clarify
fix A
assume **: A ⊆ Field(r − Id) and ***: A ≠ {} then have ∃a ∈ A. ∀a′ ∈ A. (a′, a) ∈ r
using Field * by simp
moreover have ∀a ∈ A. ∀a′ ∈ A. (a, a′) ∈ r ↔ (a′, a) ∉ r − Id
using Linear-order-in-diff-Id [OF ⟨Linear-order r⟩] ** mono-Field[of r − Id r] by blast
ultimately show ∃a ∈ A. ∀a′ ∈ A. (a′, a) ∉ r − Id by blast
qed
qed

corollary Linear-order-Well-order-iff:
Linear-order r ⇒
Well-order r ←→ (∀A ⊆ Field r. A ≠ {} → (∃a ∈ A. ∀a′ ∈ A. (a, a′) ∈ r))
unfolding well-order-on-def using Linear-order-wf-diff-Id[of r] by blast
end
23 Hilbert’s Epsilon-Operator and the Axiom of Choice

theory Hilbert-Choice
  imports Wellfounded
  keywords specification :: thy-goal-defn
begin

23.1 Hilbert’s epsilon

axiomatization Eps :: ('a ⇒ bool) ⇒ 'a
  where someI: P x ⟷ P (Eps P)

syntax (epsilon)
  -Eps :: pttrn ⇒ bool ⇒ 'a (∃ε./ -) [0, 10] 10
syntax (input)
  -Eps :: pttrn ⇒ bool ⇒ 'a (∃@./ -) [0, 10] 10
syntax
  -Eps :: pttrn ⇒ bool ⇒ 'a (∃SOME -./ -) [0, 10] 10
translations
  SOME x. P ⇌ CONST Eps (λx. P)
print-translation
⟨[const-syntax ⟨Eps⟩], fn - => fn [Abs abs] =>
  let val (x, t) = Syntax-Trans.atomic-abs-tr′ abs
  in Syntax.const syntax-const (-Eps) $ x $ t end⟩
  — to avoid eta-contraction of body

definition inv-into :: 'a set ⇒ ('a ⇒ 'b) ⇒ ('b ⇒ 'a) where
  inv-into A f = (λx. SOME y. y ∈ A ∧ f y = x)
lemma inv-into-def2: inv-into A f x = (SOME y. y ∈ A ∧ f y = x)
  by (simp add: inv-into-def)
abbreviation inv :: ('a ⇒ 'b) ⇒ ('b ⇒ 'a) where
  inv ≡ inv-into UNIV

23.2 Hilbert’s Epsilon-operator

lemma Eps-cong:
  assumes f·x. P x = Q x
  shows Eps P = Eps Q
  using ext[of P Q, OF assms] by simp

Easier to apply than someI if the witness comes from an existential formula.

lemma someI-ex [elim?]: ∃ x. P x ⟷ P (SOME x. P x)
  apply (erule exE)
  apply (erule someI)
done
lemma some-eq-imp:
  assumes Eps P = a P b shows P a
  using assms someI-ex by force

Easier to apply than someI because the conclusion has only one occurrence of P.

lemma someI2: P a \rightarrow (\forall x. P x \Rightarrow Q x) \Rightarrow Q (SOME x. P x)
  by (blast intro: someI)

Easier to apply than someI2 if the witness comes from an existential formula.

lemma someI2-ex: \exists a. P a \rightarrow (\forall x. P x \Rightarrow Q x) \Rightarrow Q (SOME x. P x)
  by (blast intro: someI2)

lemma someI2-bex: \exists a \in A. P a \rightarrow (\forall x. x \in A \land P x \Rightarrow Q x) \Rightarrow Q (SOME x. P x)
  by (blast intro: someI2)

lemma some-equality [intro]: P a \rightarrow (\forall x. P x \Rightarrow x = a) \Rightarrow (SOME x. P x) = a
  by (blast intro: someI)

lemma some1-equality: \exists! x. P x \Rightarrow P a \Rightarrow (SOME x. P x) = a
  by blast

lemma some-eq-ex: P (SOME x. P x) \iff (\exists x. P x)
  by (blast intro: someI)

lemma some-in-eq: (SOME x. x \in A) \in A \iff A \neq \{} unfolding ex-in-conv[symmetric] by (rule some-eq-ex)

lemma some-eq-trivial [simp]: (SOME y. y = x) = x
  by (rule some-equality) (rule refl)

lemma some-sym-eq-trivial [simp]: (SOME y. x = y) = x
  apply (rule some-equality)
  apply (rule refl)
  apply (erule sym)
  done

23.3 Axiom of Choice, Proved Using the Description Operator

lemma choice: \forall x. \exists y. Q x y \Rightarrow \exists f. \forall x. Q x (f x)
  by (fast elim: someI)

lemma bchoice: \forall x \in S. \exists y. Q x y \Rightarrow \exists f. \forall x \in S. Q x (f x)
  by (fast elim: someI)
lemma choice-iff: \((\forall x. \exists y. Q x y) \iff (\exists f. \forall x. Q x (f x))\)
by (fast elim: someI)

lemma choice-iff': \((\forall x. P x \rightarrow (\exists y. Q x y)) \iff (\exists f. \forall x. P x \rightarrow Q x (f x))\)
by (fast elim: someI)

lemma bchoice-iff: \((\forall x \in S. \exists y. Q x y) \iff (\exists f. \forall x \in S. Q x (f x))\)
by (fast elim: someI)

lemma bchoice-iff': \((\forall x \in S. P x \rightarrow (\exists y. Q x y)) \iff (\exists f. \forall x \in S. P x \rightarrow Q x (f x))\)
by (fast elim: someI)

lemma dependent-nat-choice:
assumes 1: \(\exists x. P 0 x\)
and 2: \(\\forall n. P n x \Rightarrow \exists y. P (Suc n) y \land Q n x y\)
shows \(\exists f. \forall n. P n (f n) \land Q n (f n) (f (Suc n))\)
proof (intro exI allI conjI)
fix \(n\)
define \(f\) where \(f = \text{rec-nat} (\text{SOME } x. P 0 x) (\lambda n x. \text{SOME } y. P (Suc n) y \land Q n x y)\)
then have \(P 0 (f 0) \land n. P n (f n) \Rightarrow P (Suc n) (f (Suc n)) \land Q n (f n) (f (Suc n))\)
using someI-ex[OF 1] someI-ex[OF 2] by simp-all
then show \(\exists f. \forall n. P n (f n) (f (Suc n))\)
by (induct \(n\)) auto
qed

lemma finite-subset-Union:
assumes finite \(A\) \(A \subseteq \bigcup B\)
obtains \(\exists F\) where finite \(F\) \(F \subseteq B\) \(A \subseteq \bigcup F\)
proof
have \(\forall x \in A. \exists B \in B. x \in B\)
using assms by blast
then obtain \(f\) where \(f: \\forall x. x \in A \Rightarrow f x \in B \land x \in f x\)
by (auto simp add: bchoice-iff Bex-def)
show thesis
proof
show finite \((f : A)\)
using assms by auto
qed (use \(f\) in auto)
qed

23.4 Function Inverse

lemma inv-def: \(inv f = (\lambda y. \text{SOME } x. f x = y)\)
by (simp add: inv-into-def)

lemma inv-into-into: \(x \in f ^t A \Rightarrow inv-into A f x \in A\)
by (simp add: inv-into-def) (fast intro: someI2)

lemma inv-identity [simp]: inv (λa. a) = (λa. a)
  by (simp add: inv-def)

lemma inv-id [simp]: inv id = id
  by (simp add: id-def)

lemma inv-into-f-f [simp]: inj-on f A → x ∈ A → inv-into A f (f x) = x
  by (simp add: inv-into-def inj-on-def) (blast intro: someI2)

lemma inv-f-f: inj f =⇒ inv f (f x) = x
  by simp

lemma f-inv-into-f: y ∈ f:A → f (inv-into A f y) = y
  by (simp add: inv-into-def) (fast intro: someI2)

lemma inv-into-f-eq: inj-on f A → x ∈ A → f x = y → inv-into A f y = x
  by (erule subst) (fast intro: inv-into-f-f)

lemma inv-f-eq: inj f =⇒ f x = y → inv f y = x
  by (simp add: inv-into-f-eq)

lemma inj-imp-inv-eq: inj f =⇒ ∀x. f (g x) = x =⇒ inv f = g
  by (blast intro: inv-into-f-eq)

But is it useful?

lemma inj-transfer:
  assumes inj: inj f
  and minor: ∀y. y ∈ range f → P (inv f y)
  shows P x
proof –
  have f x ∈ range f by auto
  then have P(inv f (f x)) by (rule minor)
  then show P x by (simp add: inv-into-f-f [OF inj])
qed

lemma inj-iff: inj f ⇔ inv f ◦ f = id
  by (simp add: o-def lan-eq-iff) (blast intro: inj-on-inverseI inv-into-f-f)

lemma inv-o-cancel[simp]: inj f → inv f ◦ f = id
  by (simp add: inj-iff)

lemma o-inv-o-cancel[simp]: inj f → g ◦ inv f ◦ f = g
  by (simp add: comp-assoc)

lemma inv-into-image-cancel[simp]: inj-on f A → S ⊆ A → inv-into A f ◦ f ◦ S = S
  by (fastforce simp: image-def)
lemma inj-imp-surj-imp-inv: inj f \implies surj (inv f)
by (blast intro!: surjI inv-into-f-f)

lemma surj-f-imp-inv: surj f \implies f (inv f y) = y
by (simp add: f-imp-inv)

lemma bij-inv-imp-inv: bij p \implies x = inv p y \iff p x = y
using surj-f-imp-inv[of p] by (auto simp add: bij-def)

lemma inv-into-injective:
assumes eq: inv-into A f x = inv-into A f y
and x: x \in f'A
and y: y \in f'A
shows x = y
proof -
from eq have f (inv-into A f x) = f (inv-into A f y)
  by simp
with x y show ?thesis
  by (simp add: f-imp-inv)
qed

lemma inj-on-inv-into: B \subseteq f'A \implies inj-on (inv-into A f) B
by (blast intro: inj-onI dest: inv-into-injective injD)

lemma bij-betw-inv-into: bij-betw f A B \implies bij-betw (inv-into A f) B A
by (auto simp add: bij-betw-def inj-on-inv-into)

lemma surj-imp-inv-inv: surj f \implies inj (inv f)
by (simp add: inj-imp-inv-into)

lemma surj-iff: surj f \iff f \circ inv f = id
by (auto intro!: surjI simp: surj-f-imp-inv fun-imp-inv[where 'b='a])

lemma surj-iff-all: surj f \iff (\forall x. f (inv f x) = x)
by (simp add: o-def surj-iff fun-imp-inv)

lemma surj-imp-inv-imp: surj f \implies \forall x. g (f x) = x \implies inv f = g
apply (rule ext)
apply (drule_tac x = inv f x in spec)
apply (simp add: surj-f-imp-inv)
done

lemma bij-imp-bij-inv: bij f \implies bij (inv f)
by (simp add: bij-imp-inv)

lemma inv-imp-inv: (\forall x. g (f x) = x) \implies (\forall y. f (g y) = y) \implies inv f = g
by (rule ext) (auto simp add: inv-into-def)
lemma inv-inv-eq: \(\text{bij } f \implies \text{inv } (\text{inv } f) = f\)
by (rule inv-equality) (auto simp add: bij-def surj-f-inv-f)

\(\text{bij } (\text{inv } f)\) implies little about \(f\). Consider \(f : \text{bool} \Rightarrow \text{bool}\) such that \(f \text{ True} = f \text{ False} = \text{True}\). Then it is consistent with axiom \textit{someI} that \(\text{inv } f\) could be any function at all, including the identity function. If \(\text{inv } f = \text{id}\) then \(\text{inv } f\) is a bijection, but \(\text{inj } f, \text{surj } f\) and \(\text{inv } (\text{inv } f) = f\) all fail.

lemma inv-into-comp:
\[\text{inj-on } f \ (g \ A) \implies \text{inj-on } g \ A \implies x \in f \ g \ A \implies \text{inv-into } A \ (f \circ g) \ x = (\text{inv-into } A \ g \circ \text{inv-into } (g \ A) \ f) \ x\]
apply (rule inv-into-f-eq)
apply (fast intro: comp-inj-on)
apply (simp add: inv-into-into)
done

lemma o-inv-distrib: \(\text{bij } f \implies \text{bij } g \implies \text{inv } (f \circ g) = \text{inv } g \circ \text{inv } f\)
by (rule inv-equality) (auto simp add: bij-def surj-f-inv-f)

lemma image-f-inv-f: \(\text{surj } f \implies f \ A \ (\text{inv } f \ A) = A\)
by simp

lemma image-inv-f-f: \(\text{inj } f \implies \text{inv } f \ A \ (f \ A) = A\)
by simp

lemma bij-image-Collect-eq: \(\text{bij } f \implies f \ \text{Collect } P = \{y. \ P \ (\text{inv } f \ y)\}\)
apply auto
apply (force simp add: bij-is-inj)
apply (blast intro: bij-is-surj [THEN surj-f-inv-f, symmetric])
done

lemma bij-vimage-eq-inv-image:
\[\text{bij } f \implies f \ominus A = \text{inv } f \ A\]
apply (auto simp add: bij-is-surj [THEN inv-into-f-f, symmetric])
done

lemma inv-fn-o-fn-is-id:
fixes f::'a \Rightarrow 'a
assumes bij f
sows ((\text{inv } f) "n) o (f"n) = (\lambda x. x)
proof
have ((\text{inv } f) "n)((f"n) x) = x for x
proof (induction n)
case (Suc n)
have *: \((\text{inv } f) "n)(f"n) y = y\ for y
by (simp add: assms bij-is-inj)
have (\text{inv } f "Suc n) ((f "Suc n) x) = (\text{inv } f "n) (\text{inv } f (f ((f "n) x)))
by (simp add: funpow-swapf)
also have ... = (\text{inv } f "n) ((f "n) x)

using \* by auto
also have \ldots = x using Suc.IH by auto
finally show \?case by simp
qed (auto)
then show \?thesis unfolding o-def by blast
qed

lemma fn-o-inv-fn-is-id:
fixes f::'a ⇒'a
assumes bij f
shows \((f^n)\ o \ ((inv f)^n)\) = (λx. x)
proof –
have \((f^n)\ o \ ((inv f)^n)\) = x for x n
proof (induction n)
case Suc n
have \*: f\(\ inv f\ y\) = y for y
using bij-inv-eq-iff[OF assms] by auto
have \((f^n)\ o \ ((inv f)^n)\) = (λx. x)
by (simp add: funpow_swap1)
also have \ldots = (f^n)\ (inv f^n)\ x
using \* by auto
also have \ldots = y using Suc.IH by auto
finally show \?case by simp
qed (auto)
then show \?thesis unfolding o-def by blast
qed

lemma inv-fn:
fixes f::'a ⇒'a
assumes bij f
shows inv \((f^n)\) = \((inv f)^n\)
proof –
have inv \((f^n)\) = x for x
apply (rule inv-into-f-eq, auto simp add: inj-fn[OF bij-is-inj[OF assms]])
using fn-o-inv-fn-is-id[OF assms, of n, THEN fun-cong] by (simp)
then show \?thesis by auto
qed

lemma mono-inv:
fixes f::'a:linorder ⇒'b:linorder
assumes mono f bij f
shows mono (inv f)
proof
fix x y::'b assume x ≤ y
from \(bij f\) obtain a b where x: \(x = f a\) and y: \(y = f b\) by(fastforce simp: bij-def sarj-def)
show inv f x ≤ inv f y
proof (rule le-cases)
assume a ≤ b
thus ?thesis using ⟨bij f⟩ x y by (simp add: bij-def inv-f-f)

next
  assume b ≤ a
  hence f b ≤ f a by (rule monoD[OF ⟨mono f⟩])
  hence y ≤ x using x y by simp
  hence x = y using ⟨x ≤ y⟩ by auto
thus ?thesis by simp

qed

lemma mono-bij-Inf:
  fixes f :: 'a::complete-linorder ⇒ 'b::complete-linorder
  assumes mono f bij f
  shows f (Inf A) = Inf (f' A)
proof –
  have surj f using ⟨bij f⟩ by (auto simp: bij-betw-def)
  have *: (inv f) (Inf (f' A)) ≤ Inf (f (inv f) (f' A))
    using mono-Inf[OF mono-inv[OF assms], of f'A] by simp
  have Inf (f' A) ≤ f (Inf (f (inv f) (f' A)))
    using monoD[OF ⟨mono f⟩ *] by (simp add: surj-f-inv-f[OF ⟨surj f⟩])
  also have ... = f (Inf A)
    using assms by (simp add: bij-is-inj)
  finally show ?thesis using mono-Inf[OF assms (1), of A] by auto

qed

lemma finite-fun-UNIVD1:
  assumes fin: finite (UNIV :: ('a ⇒ 'b) set)
            and card: card (UNIV :: 'b set) ≠ Suc 0
  shows finite (UNIV :: 'a set)
proof –
  let ?UNIV-b = UNIV :: 'b set
  from fin have finite ?UNIV-b
    by (rule finite-fun-UNIVD2)
  with card have card ?UNIV-b ≥ Suc (Suc 0)
    by (cases card ?UNIV-b) (auto simp: card-eq-0-iff)
  then have card ?UNIV-b = Suc (Suc (Suc 0))
    by simp
  then obtain b1 b2 :: 'b where b1 b2: b1 ≠ b2
    by (auto simp: card-Suc-eq)
  from fin have fin': finite (range (λf :: 'a ⇒ 'b. inv f b1))
    by (rule finite-image1)
  have UNIV = range (λf :: 'a ⇒ 'b. inv f b1)
    proof (rule UNIV-eq-I)
      fix x :: 'a
      from b1 b2 have x = inv (λy. if y = x then b1 else b2) b1
        by (simp add: inv-into-def)
      then show x ∈ range (λf::'a ⇒ 'b. inv f b1)
        by blast
    qed

qed
with fin' show thesis
  by simp
qed

Every infinite set contains a countable subset. More precisely we show that a set $S$ is infinite if and only if there exists an injective function from the naturals into $S$.

The “only if” direction is harder because it requires the construction of a sequence of pairwise different elements of an infinite set $S$. The idea is to construct a sequence of non-empty and infinite subsets of $S$ obtained by successively removing elements of $S$.

**lemma infinite-countable-subset:**

assumes inf: ¬ finite S

shows $\exists f :: \text{nat} \Rightarrow 'a. \text{inj} \ f \land \text{range} \ f \subseteq S$

— Courtesy of Stephan Merz

**proof**

- define $Sseq$ where $Sseq = \text{rec-nat} S \ (\lambda n. T. T - \{\text{SOME} \ e. \ e \in T\})$
- define pick where $\text{pick} \ n = (\text{SOME} \ e. \ e \in Sseq \ n)$ for $n$
- have $*: Sseq \ n \subseteq S \sim \text{finite} (Sseq \ n)$ for $n$
- by (induct $n$) (auto simp: Sseq-def inf)
- then have $**: \forall n. \text{pick} \ n \in Sseq \ n$
- unfolding pick-def by (subst (asm) finite.simps) (auto simp add: ex-in-conv intro: someI-ex)
- with * have range-def by auto
- moreover have $\text{pick} \ n \neq \text{pick} \ (n + \text{Suc} \ m)$ for $m \ n$
- proof
  - have $\text{pick} \ n \notin Sseq \ (n + \text{Suc} \ m)$
  - by (induct $m$) (auto simp add: Sseq-def pick-def)
- with ** show thesis by auto
- qed
- then have in?pick
  - by (intro linorder-injI) (auto simp add: less-iff-Suc-add)
- ultimately show thesis by blast
- qed

**lemma infinite-iff-countable-subset:** ¬ finite S $\iff$ $(\exists f :: \text{nat} \Rightarrow 'a. \text{inj} \ f \land \text{range} \ f \subseteq S)$

— Courtesy of Stephan Merz

using finite-imageD finite-subset infinite-UNIV-char-0 infinite-countable-subset
by auto

**lemma image-inv-into-cancel:**

assumes surj: $fA = A'$

and sub: $B' \subseteq A'$

shows $f (\text{inv-into} \ A \ f) B' = B'$

using assms

**proof** (auto simp: f-inv-into-f)

let $?f' = \text{inv-into} \ A \ f$
THEORY "Hilbert-Choice"

fix $a'$
assume $*: a' \in B'$
with sub have $a' \in A'$ by auto
with surj have $a' = f \ (\ ?f' \ a')$
  by (auto simp: f-inv-into-f)
with $*$ show $a' \in f' \ (\ ?f' \ B')$ by blast
qed

lemma inv-into-inv-into-eq:
assumes bij-betw f A A'
and $a: a \in A$
shows inv-into A' (inv-into A f) $a = f a$
proof --
let $?f' = \text{inv-into A f}$
let $?f'' = \text{inv-into A' ?f'}$
from assms have $*: \text{bij-betw ?f' A' A}$
  by (auto simp: bij-betw-inv-into)
with $a$ obtain $a'$ where $a': a' \in A' \ ?f' \ a' = a$
unfolding bij-betw-def by force
with $a$ have $?f'' \ a = a'$
  by (auto simp: f-inv-into-f bij-betw-def)
moreover from assms $a'$ have $f a = a'$
  by (auto simp: bij-betw-def)
ultimately show $?f'' \ a = f a$ by simp
qed

lemma inj-on-iff-surj:
assumes $A \neq \{}$
shows $(\exists f. \ \text{inj-on} f A \land f ' A \subseteq A') \leftrightarrow (\exists g. g ' A' = A)$
proof safe
fix $f$
assume inj: inj-on f A and incl: $f ' A \subseteq A'$
let $?phi = \lambda a'. a \in A \land f a = a'$
let $?csi = \lambda a. a \in A$
let $?g = \lambda a'. \ if \ a' \in f ' A \ then \ (\text{SOME} a. ?phi a' \ a) \ else \ (\text{SOME} a. ?csi a)$
have $?g ' A' = A$
proof
show $?g ' A' \subseteq A$
proof clarify
fix $a'$
assume $*: a' \in A'$
show $?g a' \in A$
proof (cases $a' \in f ' A$)
case True
then obtain a where $?phi a' \ a$ by blast
then have $?phi a' \ (\text{SOME} a. ?phi a' \ a)$
  using someI[of $?phi a' \ a$] by blast
with True show $?thesis$ by auto
next
case False
  with assms have \( ?csi \ (\text{SOME} \ a. \ ?csi \ a) \)
  using someI-ex[\( \text{of} \ ?csi \)] by blast
  with False show \( \theta \)thesis by auto
qed
qed
next
show \( A \subseteq \ ?g \cdot A' \)
proof -
  have \( ?g \ (f \ a) = a \land f \ a \in A' \) if \( a : a \in A \) for \( a \)
  proof -
    let \( ?b = \text{SOME} \ aa. \ ?phi \ (f \ a) \ aa \)
    from \( a \) have \( ?phi \ (f \ a) \ a \) by auto
    then have \( \ast : ?phi \ (f \ a) \ ?b \)
      using someI[\( \text{of} \ ?phi \ (f \ a) \ a \)] by blast
    then have \( ?g \ (f \ a) = \ ?b \)
      by (auto simp add: inj-on-def)
    ultimately have \( ?g(f(a)) = a \)
      by simp
    with incl a show \( \theta \)thesis by auto
  qed
  then show \( \theta \)thesis by force
qed
qed
then show \( \exists \ g. \ g' \cdot A' = A \)
  by blast
next
  fix \( g \)
  let \( ?f = \text{inv-into} \ A' \ g \)
  have inj-on \( ?f \ (g' \cdot A') \)
    by (auto simp: inj-on-inv-into)
  moreover have \( ?f \ (g \ a') \in A' \) if \( a' : a' \in A' \) for \( a' \)
  proof -
    let \( ?phi = \lambda \ b'. \ b' \in A' \land g \ b' = g \ a' \)
    from \( a' \) have \( ?phi \ a' \) by auto
    then have \( ?phi \ (\text{SOME} \ b'. \ ?phi \ b') \)
      using someI[\( \text{of} \ ?phi \)] by blast
    then show \( \theta \)thesis by (auto simp: inv-into-def)
  qed
  ultimately show \( \exists \ f. \ \text{inj-on} \ f \ (g' \cdot A') \land f' \cdot g' \cdot A' \subseteq A' \)
    by auto
qed

lemma Ex-inj-on-UNION-Sigma:
\( \exists \ f. \ (\text{inj-on} \ f \ (\bigcup i \in I. \ A \ i) \land f' \cdot (\bigcup i \in I. \ A \ i) \subseteq (\text{SIGMA} \ i : I. \ A \ i)) \)
proof
  let \( ?phi = \lambda a. \ i \in I \land a \in A \ i \)
  let \( ?sm = \lambda a. \ \text{SOME} \ i. \ ?phi \ a \ i \)
  let \( ?f = \lambda a. \ (\text{SOME} \ a. \ ?sm \ a \ i) \)
  have inj-on \( ?f \ (\bigcup i \in I. \ A \ i) \)
by (auto simp: inj-on-def)
moreover
have ?sm a ∈ I ∧ a ∈ A(?sm a) if i ∈ I and a ∈ A i for i a
  using that someI[of ?phi a i] by auto
then have ?f · (⋃ i ∈ I. A i) ⊆ (SIGMA i : I. A i)
  by auto
ultimately show inj-on ?f · (⋃ i ∈ I. A i) ∧ ?f · (⋃ i ∈ I. A i) ⊆ (SIGMA i : I. A i)
  by auto
qed

lemma inv-unique-comp:
  assumes fg: f ∘ g = id
  and gf: g ∘ f = id
  shows inv f = g
  using fg gf inv-equality[of g f]
  by (auto simp add: fun-eq-iff)

lemma subset-image-inj:
  S ⊆ f · T ←→ (∃ U. U ⊆ T ∧ inj-on f U ∧ S = f · U)
proof safe
  show ∃ U ⊆ T. inj-on f U ∧ S = f · U
    if S ⊆ f · T
  proof —
    from that [unfolded subset-image-iff subset-iff]
    obtain g where g: ∀ x. x ∈ S → g x ∈ T ∧ x = f (g x)
      by (auto simp add: image-iff Bex-def choice-iff)
    show ?thesis
    proof (intro cxI conjI)
      show g · S ⊆ T
        by (simp add: g image-subsetI)
      show inj-on f (g · S)
        using g by (auto simp: inj-on-def)
      show S = f · (g · S)
        using g image-subset-iff by auto
    qed
  qed
qed blast

23.5 Other Consequences of Hilbert’s Epsilon

Hilbert’s Epsilon and the split Operator

Looping simprule!

lemma split-paired-Eps: (SOME x. P x) = (SOME (a, b). P (a, b))
  by simp

lemma Eps-case-prod: Eps (case-prod P) = (SOME xy. P (fst xy) (snd xy))
  by (simp add: split-def)
A relation is wellfounded iff it has no infinite descending chain.

**Lemma** \(\text{wf-iff-no-infinite-down-chain}: wf r \iff (\exists f. \forall i. (f (\text{Suc} i), f i) \in r)\)

**Proof**

Assume \(wf r\)

Show \(\neg \exists x\)

Proof

Assume \(\exists x\)

Then obtain \(f\) where \(f (\text{Suc} i), f i) \in r\) for \(i\)

By \(\text{blast}\)

From \(wf r\) have minimal: \(x \in Q \Longrightarrow \exists z \in Q, \forall y. (y, z) \in r \rightarrow y \notin Q\) for \(x \in Q\)

By \((\text{auto simp: wf-obj-minimal})\)

Let \(Q = \{ w. \exists i. w = f i \}\)

Fix \(n\)

Have \(f n \in Q\) by \(\text{blast}\)

From minimal \([\text{OF this}]\) obtain \(j\) where \((y, f j) \in r \Longrightarrow y \notin Q\) for \(y\)

By \(\text{blast}\)

With \(\text{this}\) \([\text{OF} (f (\text{Suc} j), f j) \in r]\) have \(f (\text{Suc} j) \notin Q\) by \(\text{simp}\)

Then show \(\text{False}\) by \(\text{blast}\)

Qed

Next

Assume \(\neg \exists x\)

Then show \(\neg wf r\)

Proof \((\text{rule contrapos-np})\)

Assume \(\neg wf r\)

Then obtain \(Q x\) where \(x \in Q\) and \(\text{rec}: z \in Q \Longrightarrow \exists y. (y, z) \in r \land y \in Q\) for \(z\)

By \((\text{auto simp add: wf-obj-minimal})\)

Obtain \(\text{descend} : \text{nat} \Rightarrow 'a\)

Where \(\text{descend-0}: \text{descend} 0 = x\)

And \(\text{descend-Suc}: \text{descend} (\text{Suc} n) = (\text{SOME} y. y \in Q \land (y, \text{descend} n) \in r)\) for \(n\)

By \((\text{rule that [of rec-nat x (\lambda- rec. (\text{SOME} y. y \in Q \land (y, \text{rec}) \in r))]) simp-all}\)

Have \(\text{descend-Q}: \text{descend} n \in Q\) for \(n\)

Proof \((\text{induct n})\)

Case 0

With \(x\) show \(\text{?case}\) by \((\text{simp only: descend-0})\)

Next

Case \(\text{Suc}\)

Then show \(\text{?case}\) by \((\text{simp only: descend-Suc})\) \((\text{rule some12-ex; use rec in blast})\)

Qed

Have \((\text{descend} (\text{Suc} i), \text{descend} i) \in r\) for \(i\)

By \((\text{simp only: descend-Suc})\) \((\text{rule some12-ex; use descend-Q rec in blast})\)

Then show \(\exists f. \forall i. (f (\text{Suc} i), f i) \in r\) by \(\text{blast}\).
qed
lemma wf-no-infinite-down-chainE:
assumes \( \text{wf } r \)
obtains \( k \) where \( (f \text{ (Suc } k), f k) \notin r \)
using assms wf-iff-no-infinite-down-chain[of \( r \)] by blast

A dynamically-scoped fact for TFL
lemma tfl-some:
\( \forall P \ x. \ P x \longrightarrow P (\text{Eps } P) \)
by (blast intro: someI)

23.6 An aside: bounded accessible part
Finite monotone eventually stable sequences
lemma finite-mono-remains-stable-implies-strict-prefix:
fixes \( f :: \text{nat } \Rightarrow 'a :: \text{order} \)
assumes \( S: \text{finite } (\text{range } f) \) \( \text{mono } f \)
and eq: \( \forall n. \ f n = f \text{ (Suc } n) \longrightarrow f \text{ (Suc } n) = f \text{ (Suc } (\text{Suc } n)) \)
shows \( \exists N. (\forall n \leq N. \forall m \leq N. \ m < n \longrightarrow f m < f n) \land (\forall n \geq N. \ f N = f n) \)
using assms
proof
have \( \exists n. \ f n = f \text{ (Suc } n) \)
proof (rule ccontr)
assume \( \neg \text{thesis} \)
then have \( \forall n. \ f n \neq f \text{ (Suc } n) \) by auto
with \( \text{mono } f \) have \( \forall n. \ f n < f \text{ (Suc } n) \)
by (auto simp: le_less mono_iff_le_Suc)
with lift-Suc-mono-less-iff[of \( f \)] have \( *: \forall n m. \ n < m \Longrightarrow f n < f m \)
by auto
have inj \( f \)
proof (intro injI)
fix \( x \ y \)
assume \( f x = f y \)
then show \( x = y \)
by (cases \( x \ y \) rule: linorder-cases) (auto dest: *)
qed
with \( \text{finite } (\text{range } f) \) have \( \text{finite } (\text{UNIV::nat set}) \)
by (rule finite_imageD)
then show \( \text{False} \) by simp
qed
then obtain \( n \) where \( n: \ f n = f \text{ (Suc } n) .. \)
define \( N \) where \( N = (\text{LEAST } n. \ f n = f \text{ (Suc } n)) \)
have \( \text{N-def } \) using \( n \) by (rule LeastI)
show \( ?\text{thesis} \)
proof (intro exI[of - \( N \)] conjI allI impl)
fix \( n \)
assume \( N \leq n \)
then have $\forall m. N \leq m \implies m \leq n \implies f m = f N$

proof (induct rule: dec-induct)
  case base
  then show ?case by simp
next
case (step n)
  then show ?case
    using eq [rule-format, of $n - 1$] $N$
    by (cases $n$) (auto simp add: le-Suc-eq)
qed

from this [of $n$] \langle $N \leq n$ \rangle show $f N = f n$ by auto

next
fix $n m :: nat$
assume $m < n \quad n \leq N$
then show $f m < f n$
proof (induct rule: less-Suc-induct)
  case (1 i)
  then have $i < N$ by simp
  then have $f i \neq f (Suc i)$ unfolding $N$-def
    by (rule not-less-Least)
    with \langle mono $f$ \rangle show ?case by (simp add: mono-iff-le-Suc less-le)
  next
  case 2
  then show ?case by simp
qed

qed

lemma finite-mono-strict-prefix-implies-finite-fixpoint:
  fixes $f :: nat \Rightarrow 'a set$
  assumes $S :: \forall i. f i \subseteq S \quad \text{finite} \quad S$
    and \exists $N. (\forall n \leq N. \forall m \leq N. m < n \implies f m \subseteq f n) \quad \land \quad (\forall n \geq N. f N = f n)$
  shows $f (\text{card} \; S) = (\bigcup n. f n)$
proof
  from \exists obtain $N$ where inj: $\forall m. n \leq N \implies m \leq N \implies m < n \implies f m \subseteq f n$
    and eq: $\forall n \geq N. f N = f n$
    by atomize auto
  have $i \leq N \implies i \leq \text{card} \; (f i)$ for $i$
  proof (induct $i$)
    case 0
    then show ?case by simp
  next
    case (Suc $i$)
    with inj [of Suc $i$] have $(f i) \subseteq (f \; (Suc \; i))$ by auto
    moreover have finite $(f \; (Suc \; i))$ using $S$ by (rule finite-subset)
    ultimately have $\text{card} \; (f \; i) < \text{card} \; (f \; (Suc \; i))$ by (intro psubset-card-mono)
    with Suc inj show ?case by auto
  qed
then have $N \leq \text{card} \ (f \ N)$ by simp
also have $\ldots \leq \text{card} \ S$ using $S$ by (intro card-mono)
finally have $f \ (\text{card} \ S) = f \ N$ using eq by auto
then show ?thesis
  using eq inj \[of N]\ 
  apply auto
  apply (case-tac $n < N$)
  apply (auto simp: not-less)
done
qed

23.7 More on injections, bijections, and inverses

locale bijection =
  fixes $f :: 'a \Rightarrow 'a$
  assumes bij: bij $f$
begin

lemma bij-inv: bij ($inv f$)
  using bij by (rule bij-imp-bij-inv)

lemma surj \[simp\]: surj $f$
  using bij by (rule bij-is-surj)

lemma inj: inj $f$
  using bij by (rule bij-is-inj)

lemma surj-inv \[simp\]: surj ($inv f$)
  using inj by (rule inj-imp-surj-inv)

lemma inj-inv: inj ($inv f$)
  using surj by (rule surj-imp-inj-inv)

lemma eqI: $f \ a = f \ b \Longrightarrow a = b$
  using inj by (rule injD)

lemma eq-iff \[simp\]: $f \ a = f \ b \Longleftrightarrow a = b$
  by (auto intro: eqI)

lemma eq-invI: inv $f \ a = inv f \ b \Longrightarrow a = b$
  using inj-inv by (rule injD)

lemma eq-inv-iff \[simp\]: inv $f \ a = inv f \ b \Longleftrightarrow a = b$
  by (auto intro: eq-invI)

lemma inv-left \[simp\]: inv $f \ (f \ a) = a$
  using inj by (simp add: inv-f-eq)

lemma inv-comp-left \[simp\]: inv $f \circ f = id$
by (simp add: fun-eq-iff)

lemma inv-right [simp]: \( f(\text{inv } a) = a \)
using surj by (simp add: surj-f-inv-f)

lemma inv-comp-right [simp]: \( f \circ \text{inv } f = id \)
by (simp add: fun-eq-iff)

lemma inv-left-eq-iff [simp]: \( \text{inv } f a = b \leftrightarrow f b = a \)
by auto

lemma inv-right-eq-iff [simp]: \( b = \text{inv } f a \leftrightarrow f b = a \)
by auto

end

lemma infinite-imp-bij-betw:
assumes infinite: \( \neg \text{finite } A \)
shows \( \exists h. \text{bij-betw } h A (A - \{a\}) \)
proof (cases \( a \in A \))
case False
then have \( A - \{a\} = A \) by blast
then show ?thesis using bij-betw-id[of A] by auto
next
case True
with infinite have \( \neg \text{finite } (A - \{a\}) \) by auto
with infinite iff-countable-subset[of A - \{a\}]
obtain f :: nat \Rightarrow 'a where 1: inj f and 2: \( f \cdot \text{UNIV} \subseteq A - \{a\} \) by blast
define g where g n = (if n = 0 then a else f (Suc n)) for n
define A' where A' = g · UNIV
have \( \forall y. f y \neq a \) using 2 by blast
have 3: inj-on g UNIV \( \land g \cdot \text{UNIV} \subseteq A \land a \in g \cdot \text{UNIV} \)
apply (auto simp add: True g-def [abs-def])
apply (unfold inj-on-def)
apply (intro ballI impI)
apply (case_tac x = 0)
apply (auto simp add: 2)
proof
fix y
assume a = (if y = 0 then a else f (Suc y))
then show y = 0 by (cases y = 0) (use * in auto)
next
fix x y
assume f (Suc x) = (if y = 0 then a else f (Suc y))
with f * show x = y by (cases y = 0) (auto simp: inj-on-def)
next
fix n
from 2 show f (Suc n) \( \in A \) by blast
qed
then have 4: bij-betw g UNIV A′ ∧ a ∈ A′ ∧ A′ ⊆ A
  using inj-on-imp-bij-betw[of g] by (auto simp: A′-def)
then have 5: bij-betw (inv g) A′ UNIV
  by (auto simp add: bij-betw-inv-into)
from 3 obtain n where n: g n = a by auto
have 6: bij-betw (inv g) A′ UNIV
  by (rule bij-betw-subset) (use 3 4 n in ⟨auto simp add: bij-betw-inv-into⟩)
define v where v m = (if m < n then m else Suc m) for m
have 7: bij-betw v UNIV (UNIV − {n}) proof
  unfold bij-betw-def inj-on-def, intro conjI, clarify
  fix m1 m2
  assume v m1 = v m2
  then show m1 = m2
    apply (cases m1 < n)
    apply (cases m2 < n)
    apply (auto simp: inj-on-def v-def [abs-def])
  apply (cases m2 < n)
  apply auto done
next
  show v ' UNIV = UNIV − {n}
  proof (auto simp: v-def [abs-def])
    fix m
    assume m ≠ n
    assume *: m ∉ Suc {m'. ¬ m' < n}
    have False if n ≤ m
      proof
        from ⟨m ≠ n⟩ that have **: Suc n ≤ m by auto
        from Suc-le-D [OF this] obtain m' where m' = Suc m' ..
        with ** have n ≤ m' by auto
        with m' ∗ show ?thesis by auto
      qed
      then show m < n by force
  qed
  qed
define h' where h' = g ∘ v ∘ (inv g)
with 5 6 7 have 8: bij-betw h' A' (A' − {a})
  by (auto simp add: bij-betw-trans)
define h where h b = (if b ∈ A' then h' b else b) for b
then have ∀ b ∈ A', h b = h' b by simp
with 8 have bij-betw h A' (A' − {a})
  using bij-betw-cong[of A' h] by auto
moreover
have ∀ b ∈ A − A', h b = b by (auto simp: h-def)
then have bij-betw h (A − A') (A − A')
  using bij-betw-cong[of A − A' h id] bij-betw-id[of A − A'] by auto
moreover
from 4 have (A' ∩ (A − A')) = {} ∧ A' ∪ (A − A') = A ∧
THEORY “Hilbert-Choice”

\[(A' - \{a\}) \cap (A - A') = \{\} \land (A' - \{a\}) \cup (A - A') = A - \{a\}\]

by blast

ultimately have bij-betw h A (A - {a})
using bij-betw-combine[of h A' A' - {a}] A A' A A' by simp
then show ?thesis by blast

qed

lemma infinite-imp-bij-betw2:
assumes ¬ finite A
shows ∃ h. bij-betw h A (A ∪ {a})
proof (cases a ∈ A)
  case True
  then have A ∪ {a} = A by blast
  then show ?thesis using bij-betw-id[of A] by auto
next
  case False
  let ?A' = A ∪ {a}
  from False have A = ?A' - {a} by blast
  moreover from assms have ¬ finite ?A' by auto
  ultimately obtain f where bij-betw f ?A' A
    using infinite-imp-bij-betw[of ?A' a] by auto
  then have bij-betw (inv-into ?A' f) A ?A' by (rule bij-betw-inv-into)
  then show ?thesis by auto
qed

lemma bij-betw-inv-into-left: bij-betw f A A' a ∈ A A' f (f a) = a
unfolding bij-betw-def by clarify (rule inv-into-f-f)

lemma bij-betw-inv-into-right: bij-betw f A A' a' ∈ A' A' f (inv-into A f a') = a'
unfolding bij-betw-def using f-inv-into-f by force

lemma bij-betw-inv-into-subset:
bij-betw f A A' B ⊆ A f ' B = B' ⇒ bij-betw (inv-into A f) B' B
by (auto simp: bij-betw-def intro: inj-on-inv-into)

23.8 Specification package – Hilbertized version

lemma exE-some: Ex P ⇒ c ≡ Eps P ⇒ P c
by (simp only: someI-ex)

ML-file ⟨Tools/choice-specification.ML⟩

23.9 Complete Distributive Lattices – Properties depending on Hilbert Choice

context complete-distrib-lattice
begin
lemma Sup-Inf: \[
\bigcup (\text{Inf} \, \ A) = \bigcap (\text{Sup} \, \ A | f \, \forall \, B \in A. \ f \, \in \, B)
\]

proof (rule antisym)

\begin{itemize}
\item show \[
\bigcup (\text{Inf} \, \ A) \leq \bigcap (\text{Sup} \, \ A | f \, \forall \, B \in A. \ f \, \in \, B)
\] (rule Sup-least, rule INF-greatest)
\item using Inf-lower2 Sup-upper by auto
\end{itemize}

next

\begin{itemize}
\item show \[
\bigcap (\text{Sup} \, \ A | f \, \forall \, B \in A. \ f \, \in \, B)) \leq \bigcup (\text{Inf} \, \ A)
\] (simp add: Inf-Sup, rule SUP-least, simp, safe)
\item fix \( f \)
\item assume \( \forall \, Y. \, (\exists f. \, Y = f \, \forall \, A \land (\forall \, Y \in A. \, f \, \in \, Y)) \quad \longrightarrow \quad \exists \, Y \in Y
\) from this have \( B: \quad \forall \, Y \in A. \ f \, \in \, Y \)
\item have \( F \, Z \in Y \)
\item have \( B: \quad \forall \, Y \in A. \ f \, \in \, Y \)
\item have \( F \, Z \in Y \)
\item define \( F \) where \( F = (\lambda \, Z \cdot \text{SOME} \, x. \, x \in Z \land \neg \, \bigcap (f \, \forall \, Y \in A. \, f \, \in \, Y)) \leq x
\)
\item have \( C: \quad \forall \, Y \cdot \, Y \in A \quad \Longrightarrow \quad F \, Y \leq Y
\) using INF-greatest by blast
\item have \( E: \quad \forall \, Y \cdot \, Y \in A \quad \Longrightarrow \quad \neg \, \bigcap (f \, \forall \, Y \in A. \, f \, \in \, Y) \leq F \, Y
\) using X by (simp add: F-def, rule some12-ex, auto)
\item from \( C \) and \( B \) obtain \( Z \) where \( D: \quad Z \in A \quad \text{and} \quad Y: \quad f \, (F \, A) = F \, Z
\)
\item from \( E \) and \( D \) have \( W: \quad \neg \, \bigcap (f \, \forall \, Y \in A. \, f \, \in \, Y) \leq F \, Z
\)
\item have \( \bigcap (f \, \forall \, Y \in A. \, f \, \in \, Y) \leq F \, (F \, A)
\) (simp add: INF-greatest)
\item using \( C \) by blast
\item from this and \( W \) and \( Y \) show \( \exists \)
\item by simp
\item qed
\end{itemize}

lemma dual-complete-distrib-lattice:
THEORY “Hilbert-Choice”

class complete-distrib-lattice Sup Inf sup (\geq) (>) inf ⊥
apply (rule class.complete-distrib-lattice.intro)
apply (fact dual-complete-lattice)
by (simp add: class.complete-distrib-lattice-axioms-def Sup-Inf)

lemma sup-Inf: a ⊔ ∩ B = \bigcap((∪∩ a ' B)
proof (rule antisym)
show a ⊔ ∩ B ≤ \bigcap((∪∩ a ' B)
  apply (rule INF-greatest)
using Inf-lower sup_mono by fastforce
next
have \bigcap((∪∩ a ' B) ≤ a ⊔ ∩ B
  by (simp)
qed

lemma inf-Sup: a ⊓ ∪ B = ∪((\cap a ' B)
using dual-complete-distrib-lattice
by (rule complete-distrib-lattice sup-Inf)

lemma INF-SUP: (\bigcap y. \bigcup x. P x y) = (\bigcup f. \bigcap x. P (f x) x)
proof (rule antisym)
  show (SUP x. INF y. P (x y) y) ≤ (INF y. SUP x. P x y)
  by (rule SUP-least, rule INF-greatest, rule SUP-upper2, simp-all, rule INF-lower2, simp, blast)
next
  have (INF y. SUP x. ((P x y))) ≤ Inf (SUP i. {P x y | x . True} | y . True })
  (is ?A ≤ ?B)
  proof (rule INF-greatest, clarsimp)
    fix y
    have ?A ≤ (SUP x. P x y)
      by (rule INF-lower, simp)
    also have ... ≤ Sup {uu. ∃x. uu = P x y}
      by (simp add: full-SetCompr-eq)
    finally show ?A ≤ Sup {uu. ∃x. uu = P x y}
      by simp
  qed
also have ... ≤ (SUP x. INF y. P (x y) y)
proof (subst Inf-Sup, rule SUP-least, clarsimp)
  fix f
  assume A: ∀ Y. (∃ y. Y = {uu. ∃x. uu = P x y}) → f Y ∈ Y
  have \bigcap(f ' {(uu. ∃ y. uu = {uu. ∃x. uu = P x y})}) ≤ 
    (\bigcap) P (SOME x. f {P x y | x . True} = P x y y)
  proof (rule INF-greatest, clarsimp)
    fix y
have \((\inf x \in \{uu. \exists y. uu = \{uu. \exists x. uu = P x y\}\}). f x) \leq f \{uu. \exists x. uu = P x y\}\)

by \(\text{rule INF-lower, blast}\)

also have \(\ldots \leq P (\text{SOME } x. f \{uu. \exists x. uu = P x y\} = P x y)\) y

apply \(\text{rule someI2-ex}\)

using \(A\) by \(auto\)

finally show \(\bigcap \{ f \cdot \{ uu. \exists y. uu = \{ uu. \exists x. uu = P x y \} \} \} \leq P (\text{SOME } x. f \{ uu. \exists x. uu = P x y \} = P x y)\) y

by \(\text{simp}\)

qed

also have \(\ldots \leq (\sup x. \inf y. P (x y) y)\)

by \(\text{rule SUP-upper, simp}\)

finally show \(\bigcap \{ f \cdot \{ uu. \exists y. uu = \{ uu. \exists x. uu = P x y \} \} \} \leq (\bigcup x. \bigcap y. P (x y) y)\)

by \(\text{simp}\)

qed

lemma \(\inf-sup-set:\ (\bigcap B \in A. \bigcup (g \cdot B)) = (\bigcup B \in \{ f \cdot A \mid f. \forall C \in A. f C \in C \}. \bigcap (g \cdot B))\)

proof \(\text{rule antisym}\)

have \(\bigcap \{ (g \circ f) \cdot \{ A \} \} \leq \bigcup (g \cdot B)\) if \(\forall B. B \in A \implies f B \in B \) and \(B \in A\)

for \(f\) and \(B\)

using that by \(\text{rule INF-lower2 SUP-upper INF-lower2}\)

then show \(\bigcup x \in \{ f \cdot \{ A \} \mid f. \forall Y \in A. f Y \in Y \}. \bigcap (g \cdot A. \bigcup \{ a \in x. g a \})\)

by \(\text{auto intro!: SUP-least INF-greatest simp add: image-comp}\)

next

show \(\exists x \in \{ f \cdot \{ A \} \mid f. \forall Y \in A. f Y \in Y \}. \bigcup \{ a \in x. g a \}\)

proof \(\text{cases } {}\in A\)

\(\text{case True}\)

then show \(\text{thesis}\)

by \(\text{rule INF-lower2 simp-all}\)

next

\(\text{case False}\)

have \(*: \exists f B. B \in A \implies f B \in B \implies (\bigcap B. \text{if } B \in A \text{ then } f B \in B \text{ then } g (f B) \text{ else } \bot \text{ else } \top) \leq g (f B)\)

by \(\text{rule INF-lower2, auto}\)

have \(**: \forall f B. B \in A \implies f B \notin B \implies (\bigcap B. \text{if } B \in A \text{ then } f B \in B \text{ then } g (f B) \text{ else } \bot \text{ else } \top) \leq g (\text{SOME } x. x \in B)\)

by \(\text{rule INF-lower2, auto}\)

have \(**:*: \exists f B. B \in A \implies (\bigcap B. \text{if } B \in A \text{ then } f B \in B \text{ then } g (f B) \text{ else } \bot \text{ else } \top) \leq \bot (f B \in B \text{ then } g (f B) \text{ else } g (\text{SOME } x. x \in B)\))

by \(\text{rule INF-lower2 auto}\)

have \(*:*: \exists f. (\bigcap B. \text{if } B \in A \text{ then } f B \in B \text{ then } g (x B) \text{ else } \bot \text{ else } \top)\)
\[ \leq \bigcup x \in \{ f \cdot A | f \cdot \forall Y \in A \cdot f Y \in Y \}, \bigcap x \in x \cdot g x \]

**proof**

- **fix** \( x \)
  
  define \( F \) where \( F = (\lambda y::\text{b set}. \text{if } x \cdot y \in x \cdot y \text{ then } x \cdot y \text{ else } (\text{SOME } x \cdot x y)) \)

have \( B: (\forall Y \in A \cdot F Y \in Y) \)

using \( \text{False some-in-eq F-def by auto} \)

have \( A: F \cdot A \in \{ f \cdot A | f \cdot \forall Y \in A \cdot f Y \in Y \} \)

using \( B \) by blast

show \((\bigcap xa. \text{ if } xa \in A \text{ then if } xa \cdot xa \in xa \text{ then } g (xa) \text{ else } \bot \text{ else } \top) \leq \bigcup x \in \{ f \cdot A | f \cdot \forall Y \in A \cdot f Y \in Y \}, \bigcap x \in x \cdot g x \)

using \( A \) apply (rule SUP-upper2)

apply (rule INF-greatest)

using **

apply (auto simp add: F-def)

done

**qed**

\{**fix** \( x \)

have \((\bigcap x \in A. \bigcup x \in x \cdot g x) \leq (\bigcup xa. \text{ if } x \in A \text{ then if } xa \in x \text{ then } g xa \text{ else } \bot \text{ else } \top) \)

**proof** (cases \( x \in A \))

- **case** True

  then show \( \text{?thesis} \)

  apply (rule INF-lower2)

  apply (rule SUP-least)

  apply (rule SUP-upper2)

  apply auto

  done

next

- **case** False

  then show \( \text{?thesis by simp} \)

**qed**

\}

from this have \((\bigcap x \in A. \bigcup a \in x \cdot g a) \leq (\bigcap x. \bigcup xa. \text{ if } x \in A \text{ then if } xa \in x \text{ then } g xa \text{ else } \bot \text{ else } \top) \)

by (rule INF-greatest)

also have \( \ldots = (\bigcup x. \bigcap xa. \text{ if } xa \in A \text{ then if } xa \in xa \text{ then } g (xa) \text{ else } \bot \text{ else } \top) \)

by (simp only: INF-SUP)

also have \( \ldots \leq (\bigcup x \in \{ f \cdot A | f \cdot \forall Y \in A \cdot f Y \in Y \}, \bigcap a \in x \cdot g a) \)

apply (rule SUP-least)

using **apply simp

done

finally show \( \text{?thesis by simp} \)

**qed**

**qed**

**lemma** \( \text{SUP-INF: } (\bigcup y. \bigcap x. P x y) = (\bigcap x. \bigcup y. P (x y) y) \)
using dual-complete-distrib-lattice
by (rule complete-distrib-lattice.INF-SUP)

lemma SUP-INF-set: (⋃x∈A. ⋂ (g · x)) = (⋂x∈{f · A | f. ∀Y∈A. f Y ∈ Y}).
by (rule complete-distrib-lattice)

context complete-distrib-lattice
begin

lemma sup-INF: a ⊔ (⋂ b∈B. f b) = (⋂ b∈B. a ⊔ f b)
by (simp add: sup-Inf image-comp)

lemma inf-SUP: a ∩ (⋃ b∈B. f b) = (⋃ b∈B. a ∩ f b)
by (simp add: inf-Sup image-comp)

lemma Inf-sup: ⋂ B ⊔ a = (⋂ b∈B. b ⊔ a)
by (simp add: sup-Inf sup-commute)

lemma Sup-inf: ⋃ B ∩ a = (⋃ b∈B. b ∩ a)
by (simp add: inf-Sup inf-commute)

lemma INF-sup: (⋂ b∈B. f b) ⊔ a = (⋂ b∈B. f b ⊔ a)
by (simp add: sup-INF sup-commute)

lemma SUP-inf: (⋃ b∈B. f b) ∩ a = (⋃ b∈B. f b ∩ a)
by (simp add: inf-SUP inf-commute)

lemma Inf-sup-eq-top-iff: (⋂ b∈B. b ⊔ a = ⊤) ←→ (∀ b∈B. b ⊔ a = ⊤)
by (simp only: Inf-sup INF-top-conv)

lemma Sup-inf-eq-bot-iff: (⋃ B ∩ a = ⊥) ←→ (∀ b∈B. b ∩ a = ⊥)
by (simp only: Sup-inf SUP-bot-conv)

lemma INF-sup-distrib2: (⋂ b∈B. f b) ⊔ (⋂ b∈B. g b) = (⋂ a∈A. f a ⊔ g b)
by (subst INF-commute) (simp add: sup-INF INF-sup)

lemma SUP-inf-distrib2: (⋃ b∈B. f b) ∩ (⋃ b∈B. g b) = (⋃ a∈A. f a ∩ g b)
by (subst SUP-commute) (simp add: inf-SUP SUP-inf)

end

context complete-boolean-algebra
begin

lemma dual-complete-boolean-algebra:
  class.complete-boolean-algebra Sup Inf sup (≥) (> ) inf ⊥ ⊤ (λ x y. x ⊔ − y) uminus
  by (rule class.complete-boolean-algebra.intro,
       rule dual-complete-distrib-lattice,
       rule dual-boolean-algebra)
end

instantiation set :: (type) complete-distrib-lattice
begin
instance proof (standard, clarsimp)
  fix A :: (('a set) set) set
  fix x::'a
  define F where F = (λ Y. (SOME X . (Y ∈ A ∧ X ∈ Y ∧ x ∈ X)))
  assume A: ∀ xa∈A. ∃ X∈xa. x ∈ X

  from this have B: (∀ xa ∈ F ' A. x ∈ xa)
    apply (safe, simp add: F-def)
    by (rule someI2-ex, auto)
  have C: (∀ Y∈A. F Y ∈ Y)
    apply (simp add: F-def, safe)
    apply (rule someI2-ex)
    using A by auto
  have (∃f. F ' A = f ' A ∧ (∀ Y∈A. f Y ∈ Y))
    using C by blast

  from B and this show ∃ X. (∃ f. X = f ' A ∧ (∀ Y∈A. f Y ∈ Y)) ∧ (∀ xa∈X. x ∈ xa)
    by auto
qed
end

instance set :: (type) complete-boolean-algebra ..

instantiation fun :: (type, complete-distrib-lattice) complete-distrib-lattice
begin
instance by standard (simp add: le-fun-def INF-SUP-set image-comp)
end

instance fun :: (type, complete-boolean-algebra) complete-boolean-algebra ..

context complete-linorder
begin
subclass complete-distrib-lattice
proof (standard, rule ccontr)
  fix A
  assume ~ (∇ (Sup ' A)) ≤ ∪ (Inf ' {f' A | f. ∀ Y∈A. f Y ∈ Y})
  then have C: (∇ (Sup ' A)) > ∪ (Inf ' {f' A | f. ∀ Y∈A. f Y ∈ Y})
    by (simp add: not-le)
  show False
    proof (cases ∃ z. (∇ (Sup ' A)) > z ∧ z > ∪ (Inf ' {f' A | f. ∀ Y∈A. f Y ∈ Y})
    case True
    from this obtain z where A: z < (∇ (Sup ' A)) and X: z > ∪ (Inf ' {f' A | f. ∀ Y∈A. f Y ∈ Y})
    by blast
    from A have ∨ Y. Y ∈ A ⇒ z < Sup Y
      by (simp add: less-INF-D)
    from this have B: ∨ Y. Y ∈ A ⇒ ∃ k ∈ Y. z < k
      using local.less-Sup-iff by blast
    define F where F = (∀ Y. SOME k. k ∈ Y ∧ z < k)
    have D: ∨ Y. Y ∈ A ⇒ z < F Y
      using B apply (simp add: F-def)
      by (rule someI2-ex, auto)
    have E: ∨ Y. Y ∈ A ⇒ F Y ∈ Y
      using B apply (simp add: F-def)
      by (rule someI2-ex, auto)
    have z ≤ Inf (F ' A)
      by (simp add: D local.INF-greatest local.order.strict-implies-order)
    also have ... ≤ ∪ (Inf ' {f' A | f. ∀ Y∈A. f Y ∈ Y})
      apply (rule SUP-upper, safe)
      using E by blast
    finally have z ≤ ∪ (Inf ' {f' A | f. ∀ Y∈A. f Y ∈ Y})
      by simp
    from X and this show ?thesis
      using local.not-less by blast
    next
      case False
      from this have A: ∨ z. (∇ (Sup ' A)) ≤ z ∨ z ≤ ∪ (Inf ' {f' A | f. ∀ Y∈A. f Y ∈ Y})
      using local.le-less-linear by blast
THEORY "Zorn"

from C have \( \forall Y . Y \in A \implies \bigcup (\text{Inf} \ f \ A \mid f. \forall Y \in A. f \ Y \in Y) < \Sup Y \)
by (simp add: less-INF-D)

from this have B: \( \forall Y . Y \in A \implies \exists \ k \in Y . \bigcup (\text{Inf} \ f \ A \mid f. \forall Y \in A. f \ Y \in Y) < k \)
using local.less-Sup-iff by blast

define F where \( F = (\lambda Y . \text{SOME} \ k \in Y . \bigcup (\text{Inf} \ f \ A \mid f. \forall Y \in A. f \ Y \in Y)) < k \)

have D: \( \forall Y . Y \in A \implies \bigcup (\text{Inf} \ f \ A \mid f. \forall Y \in A. f \ Y \in Y) < F \ Y \)
using B apply (simp add: F-def)
by (rule someI2-ex, auto)

have E: \( \forall Y . Y \in A \implies F \ Y \in Y \)
using B apply (simp add: F-def)
by (rule someI2-ex, auto)

have \( \forall Y . Y \in A \implies \bigcap (\Sup A) \leq F \ Y \)
using D False local.leI by blast

from this have \( \bigcap (\Sup A) \leq \text{Inf} \ (F \ A) \)
by (simp add: local.INF-greatest)

also have \( \text{Inf} \ (F \ A) \leq \bigcup (\text{Inf} \ f \ A \mid f. \forall Y \in A. f \ Y \in Y) \)
apply (rule SUP-upper, safe)
using E by blast

finally have \( \bigcap (\Sup A) \leq \bigcup (\text{Inf} \ f \ A \mid f. \forall Y \in A. f \ Y \in Y) \)
by simp

from C and this show \( \text{thesis} \)
using not-less by blast

done

done

24 Zorn’s Lemma

theory Zorn
  imports Order-Relation Hilbert-Choice
begin
24.1 Zorn’s Lemma for the Subset Relation

24.1.1 Results that do not require an order

Let $P$ be a binary predicate on the set $A$.

locale pred-on =
  fixes $A$ :: ’a set
  and $P$ :: ’a ⇒ ’a ⇒ bool (infix ⊏)

begin

abbreviation $Peq$ :: ’a ⇒ ’a ⇒ bool (infix ⊑)
  where $x ⊑ y ≡ P==x y$

A chain is a totally ordered subset of $A$.

definition chain :: ’a set ⇒ bool
  where $\text{chain } C ≡ C ⊆ A \land (\forall x \in C. \forall y \in C. x ⊏ y \lor y ⊏ x)$

We call a chain that is a proper superset of some set $X$, but not necessarily
a chain itself, a superchain of $X$.

abbreviation superchain :: ’a set ⇒ ’a set ⇒ bool (infix <c)
  where $X <c C ≡ \text{chain } C \land X \subset C$

A maximal chain is a chain that does not have a superchain.

definition maxchain :: ’a set ⇒ bool
  where $\text{maxchain } C ≡ \text{chain } C \land (\not\exists S. C <c S)$

We define the successor of a set to be an arbitrary superchain, if such exists,
or the set itself, otherwise.

definition suc :: ’a set ⇒ ’a set
  where $\text{suc } C = (\text{if } \not\text{chain } C \lor \text{maxchain } C \text{ then } C \text{ else } (\exists S. C <c S))$

lemma chainI [Pure.intro?]: $C ⊆ A ⇒ (\forall x y. x \in C \Rightarrow y \in C \Rightarrow x \subseteq y \lor y \subseteq x) \Rightarrow \text{chain } C$
  unfolding chain-def by blast

lemma chain-total: $\text{chain } C ⇒ x \in C ⇒ y \in C ⇒ x \subseteq y \lor y \subseteq x$
  by (simp add: chain-def)

lemma not-chain-suc [simp]: $\neg \text{chain } X ⇒ \text{suc } X = X$
  by (simp add: suc-def)

lemma maxchain-suc [simp]: $\text{maxchain } X ⇒ \text{suc } X = X$
  by (simp add: suc-def)

lemma suc-subset: $X \subseteq \text{suc } X$
  by (auto simp: suc-def maxchain-def intro: someI2)

lemma chain-empty [simp]: $\text{chain } \{\}$
THEORY “Zorn” 585

by (auto simp: chain-def)

lemma not-maxchain-Some: chain C \implies \neg maxchain C \implies C <c (SOME D. C <c D)
  by (rule someI-ex) (auto simp: maxchain-def)

lemma suc-not-equals: chain C \implies \neg maxchain C \implies suc C \neq C
  using not-maxchain-Some by (auto simp: suc-def)

lemma subset-suc:
  assumes X \subseteq Y
  shows X \subseteq suc Y
  using assms by (rule subset-trans) (rule suc-subset)

We build a set \( C \) that is closed under applications of \( suc \) and contains the
union of all its subsets.

inductive-set suc-Union-closed (\( C \))
  where
  suc: X \in C \implies suc X \in C
  | Union [unfolded Pow-iff]: X \in Pow C \implies \bigcup X \in C

Since the empty set as well as the set itself is a subset of every set, \( C \) contains
at least \( \emptyset \in C \) and \( \bigcup C \in C \).

lemma suc-Union-closed-empty: \( \emptyset \in C \)
  and suc-Union-closed-Union: \( \bigcup C \in C \)
  using Union [of \( \emptyset \)] and Union [of \( C \)] by simp-all

Thus closure under \( suc \) will hit a maximal chain eventually, as is shown
below.

lemma suc-Union-closed-induct [consumes 1, case-names suc Union, induct pred: suc-Union-closed]:
  assumes X \in C
  and \( \forall X. X \in C \implies Q X \implies Q (suc X) \)
  and \( \forall X. X \subseteq C \implies \forall x \in X. Q x \implies Q (\bigcup X) \)
  shows Q X
  using assms by induct blast+

lemma suc-Union-closed-cases [consumes 1, case-names suc Union, cases pred: suc-Union-closed]:
  assumes X \in C
  and \( \forall Y. X = suc Y \implies Y \in C \implies Q \)
  and \( \forall Y. X = \bigcup Y \implies Y \subseteq C \implies Q \)
  shows Q
  using assms by cases simp-all

On chains, \( suc \) yields a chain.

lemma chain-suc:
  assumes chain X
shows \( \text{chain}(suc\ X) \)

using \( \text{assms} \)
by (cases \( \neg\ \text{chain}\ X \lor\ \text{maxchain}\ X \)) (force simp: \text{not-maxchain-Some})+

lemma \( \text{chain-sucD} \):
assumes \( \text{chain}\ X \)
shows \( suc\ X \subseteq A \land\ \text{chain}(suc\ X) \)

proof –
from \( \langle\ \text{chain}\ X\ \rangle \) have \( *:\ \text{chain}(suc\ X) \)
by (rule \text{chain-suc})
then have \( suc\ X \subseteq A \)
unfolding \text{chain-def} by blast
with \( * \) show \( \text{thesis} \) by blast
qed

lemma \( suc\text{-Union-closed-total}! \):
assumes \( X \in\mathcal{C}\ \text{and } Y \in\mathcal{C} \)
and \( *:\ \forall Z.\ Z \in\mathcal{C} \implies Z \subseteq Y \implies Z = Y \lor suc\ Z \subseteq Y \)
shows \( X \subseteq Y \lor suc\ Y \subseteq X \)
using \( \langle X \in\mathcal{C} \rangle \)

proof (induct)
case \( suc\ X \)
with \( * \) show \( \text{case} \) by (blast del: \text{subsetI intro: \text{subset-suc}})
next
case \text{Union}
then show \( \text{case} \) by blast
qed

lemma \( suc\text{-Union-closed-subsetD} \):
assumes \( Y \subseteq X\ \text{and } X \in\mathcal{C}\ \text{and } Y \in\mathcal{C} \)
shows \( X = Y \lor suc\ Y \subseteq X \)
using \( \text{assms}(2,3,1) \)

proof (induct arbitrary: \( Y \))
case \( suc\ X \)
note \( * = (\bigwedge Y.\ Y \in\mathcal{C} \implies Y \subseteq X \implies X = Y \lor suc\ Y \subseteq X) \)
with \( suc\text{-Union-closed-total}!\ [OF \langle Y \in\mathcal{C} \rangle \langle X \in\mathcal{C} \rangle] \)
have \( Y \subseteq X \lor suc\ X \subseteq Y \) by blast
then show \( \text{case} \)
proof
assume \( Y \subseteq X \)
with \( * \) and \( \langle Y \in\mathcal{C} \rangle \) have \( X = Y \lor suc\ Y \subseteq X \) by blast
then show \( \text{thesis} \)
proof
assume \( X = Y \)
then show \( \text{thesis} \) by simp
next
assume \( suc\ Y \subseteq X \)
then have \( suc\ Y \subseteq suc\ X \) by (rule \text{subset-suc})
then show \( \text{thesis} \) by simp
qed

next
assume suc X ⊆ Y
with ⟨Y ⊆ suc X⟩ show thesis by blast
qed

next
case (Union X)
show ?case
proof (rule ccontr)
assume ¬ thesis
with ⟨Y ⊆ ∪X⟩ obtain x z
where ¬ suc Y ⊆ ∪X
and x ∈ X and y ∈ x and y /∈ Y
and z ∈ suc Y and ∀ x ∈ X. z /∈ x by blast
with ⟨X ⊆ C⟩ have x ∈ C by blast
from Union and ⟨x ∈ X⟩ have *: ∃ y. y ∈ C ⊢ y ⊆ x ⊢ x = y ∨ suc y ⊆ x
by blast
with suc-Union-closed-total' [OF ⟨Y ∈ C⟩⟨x ∈ C⟩] have Y ⊆ x ∨ suc x ⊆ Y
by blast
then show False
proof
assume Y ⊆ x
with * [OF ⟨Y ∈ C⟩] have x = Y ∨ suc Y ⊆ x by blast
then show False
proof
assume x = Y
with ⟨y ∈ x⟩ and ⟨y /∈ Y⟩ show False by blast
next
assume suc Y ⊆ x
with ⟨x ∈ X⟩ have suc Y ⊆ ∪X by blast
with ⟨¬ suc Y ⊆ ∪X⟩ show False by contradiction
qed

next
assume suc x ⊆ Y
moreover from suc-subset and ⟨y ∈ x⟩ have y ∈ suc x by blast
ultimately show False using ⟨y /∈ Y⟩ by blast
qed

qed

The elements of C are totally ordered by the subset relation.

lemma suc-Union-closed-total:
assumes X ∈ C and Y ∈ C
shows X ⊆ Y ∨ Y ⊆ X
proof (cases ∀ Z ∈ C. Z ⊆ Y → Z = Y ∨ suc Z ⊆ Y)
case True
with suc-Union-closed-total' [OF assms]
have X ⊆ Y ∨ suc Y ⊆ X by blast
with \textit{suc-subset} [of \textit{Y}] \textit{show} \ ?thesis \textit{by} blast 

next 
  \textit{case} False 
  \textit{then obtain} \textit{Z where} \textit{Z} \in \textit{C} and \textit{Z} \subseteq \textit{Y} and \textit{Z} \neq \textit{Y} and \n \textit{suc} \textit{Z} \subseteq \textit{Y} 
  \textit{by} blast 
  \textit{with} \textit{suc-Union-closed-subsetD} and \{ \textit{Y} \in \textit{C} \} \textit{show} \ ?thesis 
  \textit{by} blast 

\textit{qed} 

Once we hit a fixed point \textit{w.r.t. suc}, all other elements of \textit{C} are subsets of this fixed point. 

\textbf{lemma} \textit{suc-Union-closed-suc}: 
\textit{assumes} \textit{X} \in \textit{C} and \textit{Y} \in \textit{C} and \textit{suc} \textit{Y} = \textit{Y} 
\textit{shows} \textit{X} \subseteq \textit{Y} 
\textit{using} \langle \textit{X} \in \textit{C} \rangle 
\textit{proof} \textit{induct} 
  \textit{case} \langle \textit{suc} \textit{X} \rangle 
  \textit{with} \langle \textit{Y} \in \textit{C} \rangle \textit{and} \textit{suc} \textit{Union-closed-subsetD} \textit{have} \textit{X} = \textit{Y} \lor \textit{suc} \textit{X} \subseteq \textit{Y} 
  \textit{by} blast 
  \textit{then show} \ ?case 
  \textit{by} \langle \textit{auto simp:} \langle \textit{suc} \textit{Y} = \textit{Y} \rangle \rangle 
next 
  \textit{case} \textit{Union} 
  \textit{then show} \ ?case \textit{by} blast 
\textit{qed} 

\textbf{lemma} \textit{eq-suc-Union}: 
\textit{assumes} \textit{X} \in \textit{C} 
\textit{shows} \textit{suc} \textit{X} = \textit{X} \longleftrightarrow \textit{X} = \bigcup \textit{C} 
\textit{(is} \ ?lhs \longleftrightarrow \ ?rhs) 
\textit{proof} 
\textit{assume} \ ?lhs 
\textit{then have} \bigcup \textit{C} \subseteq \textit{X} 
\textit{by} \langle \textit{rule suc-Union-closed-suc [OF suc-Union-closed-Union} \langle \textit{X} \in \textit{C} \rangle \rangle \rangle 
\textit{with} \langle \textit{X} \in \textit{C} \rangle \textit{show} \ ?rhs 
\textit{by} blast 
\textit{next} 
\textit{from} \langle \textit{X} \in \textit{C} \rangle \textit{have} \textit{suc} \textit{X} \in \textit{C} \textit{by} \langle \textit{rule suc} \rangle 
\textit{then have} \textit{suc} \textit{X} \subseteq \bigcup \textit{C} \textit{by} blast 
\textit{moreover assume} \ ?rhs 
\textit{ultimately have} \textit{suc} \textit{X} \subseteq \textit{X} \textit{by} simp 
\textit{moreover have} \textit{X} \subseteq \textit{suc} \textit{X} \textit{by} \langle \textit{rule suc-subset} \rangle 
\textit{ultimately show} \ ?lhs .. 
\textit{qed} 

\textbf{lemma} \textit{suc-in-carrier}: 
\textit{assumes} \textit{X} \subseteq \textit{A} 
\textit{shows} \textit{suc} \textit{X} \subseteq \textit{A} 
\textit{using} \textit{assms}
by (cases ¬ chain X ∨ maxchain X) (auto dest: chain-sucD)

**lemma** suc-Union-closed-in-carrier:
- **assumes** \( X \in \mathcal{C} \)
- **shows** \( X \subseteq A \)
- **using** assms
- **by** induct (auto dest: suc-in-carrier)

All elements of \( \mathcal{C} \) are chains.

**lemma** suc-Union-closed-chain:
- **assumes** \( X \in \mathcal{C} \)
- **shows** chain \( X \)
- **using** assms
- **proof** induct
  - **case** (suc \( X \))
    - then show=?case
      - **using** not-maxchain-Some by (simp add: suc-def)
  - **next**
    - **case** (Union \( X \))
      - then have \( \bigcup X \subseteq A \)
        - **by** (auto dest: suc-Union-closed-in-carrier)
      - moreover have \( \forall x \in \bigcup X. \forall y \in \bigcup X. x \subseteq y \lor y \subseteq x \)
        - **proof** (intro ballI)
          - fix \( x \) \( y \)
            - assume \( x \in \bigcup X \) and \( y \in \bigcup X \)
            - then obtain \( u \) \( v \) where \( x \in u \) and \( u \in X \) and \( y \in v \) and \( v \in X \)
              - **by** blast
            - with Union have \( u \in \mathcal{C} \) and \( v \in \mathcal{C} \) and chain \( u \) and chain \( v \)
              - **by** blast+
            - with suc-Union-closed-total have \( u \subseteq v \lor v \subseteq u \)
              - **by** blast
            - then show \( x \subseteq y \lor y \subseteq x \)
              - **proof**
                - assume \( u \subseteq v \)
                  - from (chain v) show=?thesis
                  - **proof** (rule chain-total)
                    - show \( y \in v \) by fact
                    - show \( x \in v \) using (\( u \subseteq v \)) and (\( x \in u \)) by blast
                    - qed
                - **next**
                  - assume \( v \subseteq u \)
                    - from (chain u) show=?thesis
                    - **proof** (rule chain-total)
                      - show \( x \in u \) by fact
                      - show \( y \in u \) using (\( v \subseteq u \)) and (\( y \in v \)) by blast
                      - qed
                  - qed
                - qed
            - ultimately show=?case unfolding chain-def .
          - qed
24.1.2 Hausdorff’s Maximum Principle

There exists a maximal totally ordered subset of $A$. (Note that we do not require $A$ to be partially ordered.)

**Theorem** Hausdorff: $\exists C. \text{maxchain } C$

**Proof**

- Let $\mathcal{M} = \bigcup C$
- Have $\text{maxchain } \mathcal{M}$

**Proof (rule ccontr)**

- Assume $\neg \text{thesis}$
- Then have $\text{suc } \mathcal{M} \neq \mathcal{M}$
- Using $\text{suc-not-equals and suc-Union-closed-chain}$

**By simp**

- Moreover have $\text{suc } \mathcal{M} = \mathcal{M}$
- Using $\text{eq-suc-Union}$

**Ultimately show** $\text{False by contradiction}$

**QED**

Make notation $C$ available again.

**No-notation** $\text{suc-Union-closed } (C)$

**Lemma** chain-extend: $\text{chain } C \implies z \in A \implies \forall x \in C. x \subseteq z \implies \text{chain } (\{z\} \cup C)$

**Unfolding** chain-def **by blast**

**Lemma** maxchain-imp-chain: $\text{maxchain } C \implies \text{chain } C$

**By** (simp add: maxchain-def)

**End**

Hide constant $\text{pred-on.suc-Union-closed}$, which was just needed for the proof of Hausdorff’s maximum principle.

**Hide-const** $\text{pred-on.suc-Union-closed}$

**Lemma** chain-mono:

- Assumes $\forall x \ y \in A \implies y \in A \implies P \ x \ y \implies Q \ x \ y$
- And $\text{pred-on.chain } A \ P \ C$
- Shows $\text{pred-on.chain } A \ Q \ C$

**Using** assms **unfolding** pred-on.chain-def **by blast**

24.1.3 Results for the proper subset relation

**Interpretation** subset: $\text{pred-on } A (\subset) \text{ for } A$.

**Lemma** subset-maxchain-max:
assumes \( \text{subset.maxchain } A \ C \)
and \( X \in A \)
and \( \bigcup C \subseteq X \)
shows \( \bigcup C = X \)

proof (rule ccontr)
let \( \forall C = \{X\} \cup C \)
from (subset.maxchain \( A \ C \)) have subset.chain \( A \ C \)
and \( *: \forall S. \text{subset.chain } A \ S \implies \neg C \subseteq S \)
by (auto simp: subset.maxchain-def)
moreover have \( \forall x \in C. \ x \subseteq X \) using \( \bigcup C \subseteq X \) by auto
ultimately have \( \text{subset.chain } A \ ?C \)
moreover assume \( **: \bigcup C \neq X \)
moreover from \( ** \) have \( C \subset \ ?C \)
ultimately show \( \text{False} \) using \( * \) by blast
qed

lemma subset-chain-def: \( \forall A. \text{subset.chain } A \ C = (C \subseteq A \land (\forall X \in C. \forall Y \in C. \ X \subseteq Y \lor Y \subseteq X)) \)
by (auto simp: subset.chain-def)

lemma subset-chain-insert:
\( \text{subset.chain } A \ (\text{insert } B \ B) \longleftrightarrow B \in A \land (\forall X \in B. \ X \subseteq B \lor B \subseteq X) \land \text{subset.chain } A \ B \)
by (fastforce simp add: subset-chain-def)

\[ \text{24.1.4 Zorn’s lemma} \]

If every chain has an upper bound, then there is a maximal set.

theorem subset-Zorn:
assumes \( \forall C. \text{subset.chain } A \ C \implies \exists U \in A. \forall X \in C. X \subseteq U \)
shows \( \exists M \in A. \forall X \in A. \ M \subseteq X \implies X = M \)

proof
from subset.Hausdorff [of A] obtain \( M \) where subset.maxchain \( A \ M \)
then have subset.chain \( A \ M \)
by (rule subset.maxchain-imp-chain)
with \( \text{assms} \) obtain \( Y \) where \( Y \in A \land \forall X \in M. \ X \subseteq Y \)
by blast
moreover have \( \forall X \in A. \ Y \subseteq X \implies X = Y \)

proof (intro ballI impI)
fix \( X \)
assume \( X \in A \) and \( Y \subseteq X \)
show \( Y = X \)

proof (rule ccontr)
assume \( \neg \text{thesis} \)
with \( Y \subseteq X \) have \( \neg X \subseteq Y \) by blast
from subset.chain-extend [OF (subset.chain \( A \ M \)) \( \forall X \in A \) \( \forall X \in M. \ X \subseteq Y \)] have subset.chain \( A \ (\{X\} \cup M) \)
using \( Y \subseteq X \) by auto
moreover have \( M \subseteq \{X\} \cup M \)
using \( \forall X \in M . \ X \subseteq Y \) and \( \neg X \subseteq Y \) by auto
ultimately show False
using \( \text{subset} . \maxchain \ A \ M \) by (auto simp: subset.maxchain-def)
qed
qed
ultimately show \(?thesis\) by blast
qed

Alternative version of Zorn’s lemma for the subset relation.

**lemma** subset-Zorn’:
assumes \( \bigwedge C . \subsetset . \chain A C \Rightarrow \bigcup C \in A \)
shows \( \exists M \in A . \ \forall X \in A . \ M \subseteq X \rightarrow X = M \)
proof –
from subset.Hausdorff [of A] obtain M where subset.maxchain A M ..
then have subset.chain A M
by (rule subset.maxchain-imp-chain)
with assms have \( \bigcup M \in A \).
moreover have \( \forall Z \in A . \ \bigcup M \subseteq Z \rightarrow \bigcup M = Z \)
proof (intro ballI impI)
fix Z
assume \( Z \in A \) and \( \bigcup M \subseteq Z \)
with subset-maxchain-max \[OF \ (\subsetset . \maxchain \ A \ M)\]
show \( \bigcup M = Z \).
qed
ultimately show \(?thesis\) by blast
qed

24.2 Zorn’s Lemma for Partial Orders

Relate old to new definitions.

**definition** chain-subset :: \( 'a \ \text{set} \Rightarrow \text{bool} \ (\subsetset C) \)
where \( \subsetset C \leftrightarrow (\forall A \in C . \ \forall B \in C . \ A \subseteq B \lor B \subseteq A) \)

**definition** chains :: \( 'a \ \text{set} \Rightarrow 'a \ \text{set} \ \text{set} \)
where \( \text{chains} \ A = \{C . \ C \subseteq A \land \subsetset C\} \)

**definition** Chains :: \( ('a \times 'a) \ \text{set} \Rightarrow 'a \ \text{set} \ \text{set} \)
where \( \text{Chains} \ r = \{C . \ \forall a \in C . \ \forall b \in C . \ (a, b) \in r \lor (b, a) \in r\} \)

**lemma** chains-extend: \( c \in \text{chains} \ S \Rightarrow z \in S \Rightarrow \forall x \in c . \ x \subseteq z \Rightarrow \{z\} \cup c \in \text{chains} \ S \)
for \( z :: 'a \ \text{set} \)
unfolding chains-def chain-subset-def by blast

**lemma** mono-Chains: \( r \subseteq s \Rightarrow \text{Chains} \ r \subseteq \text{Chains} \ s \)
unfolding Chains-def by blast
lemma chain-subset-alt-def: chain\,\subseteq\,C = subset.chain\,\UNIV\,C

unfolding chain-subset-def subset.chain-def by fast

lemma chains-alt-def: chains\,A = \{C.\,subset.chain\,A\,C\}
by (simp add: chains-def chain-subset-alt-def subset.chain-def)

lemma Chains-subset: Chains\,r \subseteq \{C.\,pred-on.chain\,\UNIV\,(\lambda\,x\,y.\,(x,\,y)\in\,r)\,C\}
by (force simp add: Chains-def pred-on.chain-def)

lemma Chains-subset':
assumes refl\,r
shows \{C.\,pred-on.chain\,\UNIV\,(\lambda\,x\,y.\,(x,\,y)\in\,r)\,C\} \subseteq Chains\,r
using assms Chains-subset Chains-subset'
by (blast)

lemma Chains-alt-def:
assumes refl\,r
shows Chains\,r = \{C.\,pred-on.chain\,\UNIV\,(\lambda\,x\,y.\,(x,\,y)\in\,r)\,C\}
using assms Chains-subset Chains-subset' by blast

lemma Chains-relation-of:
assumes C \in Chains\,(relation-of\,P\,A)
shows C \subseteq A
using assms unfolding Chains-def relation-of-def by auto

lemma pairwise-chain-Union:
assumes P: \每一个人\,S \in C \implies pairwise\,R\,S\,and\,chain\,\subseteq\,C
shows pairwise\,R\,(\bigcup\,C)
using ⟨\,chain\,\subseteq\,C\,⟩ unfolding pairwise-def chain-subset-def
by (blast intro: P [unfolded pairwise-def, rule-format])

lemma Zorn-Lemma: \forall\,C\in\text{chains}\,A.\,\bigcup\,C \in A \implies \exists\,M\in A.\,\forall\,X\in A.\,M \subseteq X \implies X = M
using subset-Zorn' [of A] by (force simp: chains-alt-def)

lemma Zorn-Lemma2: \forall\,C\in\text{chains}\,A.\,\exists\,U\in A.\,\forall\,X\in C.\,X \subseteq U \implies \exists\,M\in A.\,\forall\,X\in A.\,M \subseteq X \implies X = M
using subset-Zorn [of A] by (auto simp: chains-alt-def)

24.3 Other variants of Zorn’s Lemma

lemma chainsD: c \in chains\,S \implies x \in c \implies y \in c \implies x \subseteq y \lor y \subseteq x
unfolding chains-def chain-subset-def by blast

lemma chainsD2: c \in chains\,S \implies c \subseteq S
unfolding chains-def by blast

lemma Zorns-po-lemma:
assumes po: Partial-order\,r
THEORY "Zorn"

and u: \( \bigwedge C.\ C \in \text{Chains} r \implies \exists u \in \text{Field} r.\ \forall a \in C.\ (a, u) \in r \)
shows \( \exists m \in \text{Field} r.\ \forall a \in \text{Field} r.\ (m, a) \in r \implies a = m \)
proof -
  have \( \text{Preorder} r \)
    using po by (simp add: partial-order-on-def)
Mirror \( r \) in the set of subsets below (wrt \( r \)) elements of \( A \).
let \( ?B = \lambda x.\ \text{r}^{-1} \{ x \} \)
let \( ?S = \lambda B \cdot \text{Field} r \)
have \( \exists u \in \text{Field} r.\ \forall A \in C.\ A \subseteq \text{r}^{-1} \{ u \} \) (is \( \exists u \in \text{Field} r.\ \exists P u \))
if 1: \( C \subseteq ?S \) and 2: \( \forall A \in C.\ \forall B \in C.\ A \subseteq B \lor B \subseteq A \) for \( C \)
proof -
  let \( ?A = \{ x \in \text{Field} r.\ \exists M \in C.\ M = ?B x \} \)
from 1 have \( C = ?B \cdot ?A \) by (auto simp: image-def)
have \( ?A \in \text{Chains} r \)
proof (simp add: Chains-def, intro allI implI, elim conjE)
  fix \( a \ b \)
  assume \( a \in \text{Field} r \) and \( ?B \ a \in \text{C} \) and \( b \in \text{Field} r \) and \( ?B \ b \in \text{C} \)
with 2 have \( ?B \ a \subseteq ?B \ b \lor ?B \ b \subseteq ?B \ a \) by auto
  then show \( (a, b) \in r \lor (b, a) \in r \)
    using (\text{Preorder} r) and (\( a \in \text{Field} r \)) and (\( b \in \text{Field} r \))
    by (simp add:subset-Image1-Image1-iff)
qed
then obtain \( u \) where \( uA:\ u \in \text{Field} r \forall a \in ?A.\ (a, u) \in r \)
by (auto simp: dest: \( u \))
have \( \exists P u \)
proof auto
  fix \( a \ B \) assume \( aB: B \in C \) a \( B \)
with 1 obtain \( x \) where \( x \in \text{Field} r \) and \( B = \text{r}^{-1} \{ x \} \) by auto
  then show \( (a, u) \in r \)
    using \( uA \) and \( aB \) and (\text{Preorder} r)
    unfolding preorder-on-def refl-on-def by simp (fast dest: transD)
qed
then show \( \exists \text{thesis} \)
  using \( u \in \text{Field} r \) by blast
qed
then have \( \forall C \in \text{chains} \ ?S.\ \exists U \in \?S.\ \forall A \in C.\ A \subseteq U \)
by (auto simp: chains-def chain-subset-def)
from Zorn-Lemma2 [OF this] obtain \( m \) \( B \)
  where \( m \in \text{Field} r \)
and \( B = \text{r}^{-1} \{ m \} \)
and \( \forall x \in \text{Field} r.\ B \subseteq \text{r}^{-1} \{ x \} \implies \text{r}^{-1} \{ x \} = B \)
by auto
then have \( \forall a \in \text{Field} r.\ (m, a) \in r \implies a = m \)
  using po and (\text{Preorder} r) and \( \exists m \in \text{Field} r \)
  by (auto simp: subset-Image1-Image1-iff Partial-order-eq-Image1-Image1-iff)
then show \( \exists \text{thesis} \)
  using \( m \in \text{Field} r \) by blast
qed
lemma predicate-Zorn:
assumes po: partial-order-on A (relation-of P A)
  and ch: \( \bigwedge C. C \in \text{Chains} \) (relation-of P A) \( \Rightarrow \exists u \in A. \forall a \in C. P a \leftarrow u \)
shows \( \exists m \in A. \forall a \in A. P m a \rightarrow a = m \)
proof −
have \( a \in A \) if \( C \in \text{Chains} \) (relation-of P A) and \( a \in C \) for \( C \)
  using that unfolding Chains-def relation-of-def by auto
moreover have \( (a, u) \in \text{relation-of P A} \) if \( a \in A \) and \( u \in A \) and \( P a u \) for \( a \)
  unfolding relation-of-def using that by auto
ultimately have \( \exists m \in A. \forall a \in A. P m a = a \) using that unfolding Chains-def relation-of-def by auto
by (auto simp: relation-of-def)
qed

lemma Union-in-chain: \([\text{finite } B; B \neq \{\}] \); subset.chain \( A B \) \( \Rightarrow \) \( \bigcup B \in B \)
proof (induction \( B \) rule: finite-induct)
case (insert \( B \) \( B \))
show ?case
proof (cases \( B \) = \( \{\} \))
case False
then show ?thesis
  using insert sup.absorb2 by (auto simp: subset-chain-insert dest: bspec [where \( x = (\bigcup B) \)])
qed auto
qed simp

lemma Inter-in-chain: \([\text{finite } B; B \neq \{\}] \); subset.chain \( A B \) \( \Rightarrow \) \( \bigcap B \in B \)
proof (induction \( B \) rule: finite-induct)
case (insert \( B \) \( B \))
show ?case
proof (cases \( B \) = \( \{\} \))
case False
then show ?thesis
  using insert inf.absorb2 by (auto simp: subset-chain-insert dest!: bspec [where \( x = (\bigcap B) \)])
qed auto
qed simp

lemma finite-subset-Union-chain:
assumes finite \( A A \subseteq \bigcup B \neq \{\} \) and sub: subset.chain \( A B \)
obtains \( B \) where \( B \in B A \subseteq B \)
proof −
obtain \( F \) where \( F \): finite \( F F \subseteq B A \subseteq \bigcup F \)
  using assms by (auto intro: finite-subset-Union)
show thesis
proof (cases $F = \{\})$

  case True

  then show ?thesis
  using ($A \subseteq \bigcup F \setminus B \neq \{\}$) that by fastforce

next

  case False

  show ?thesis
  proof
  show $\bigcup F \in B$
  using sub ($F \subseteq B \setminus \{\}$)
  by (simp add: Union-in-chain False subset.chain_def subset_iff)

  show $A \subseteq \bigcup F$
  using ($A \subseteq \bigcup F \setminus B \neq \{\}$) by blast

  qed

  qed

lemma subset-Zorn-nonempty:

  assumes $A \neq \{\}$ and ch: $\forall C \in A \setminus \{\} \subseteq \bigcup C \in A$

  shows $\exists M \in A \setminus \{\} \forall X \in A \setminus \{\}. M \subseteq X \rightarrow X = M$

proof (rule subset-Zorn)

  show $\exists U \in A \setminus \{\} \forall X \in C \setminus \{\}. X \subseteq U$ if subset.chain $A \setminus \{\}$ for $C$

  proof (cases $C = \{\}$)

  case True

  then show ?thesis
  using ($A \neq \{\}$) by blast

next

  case False

  show ?thesis
  by (blast intro!: ch False that Union-upper)

  qed

  qed

24.4 The Well Ordering Theorem

definition init-seg-of :: $\langle \langle a \times a \rangle \times \langle a \times a \rangle \rangle \times a \times a \times (\forall a \times b \times c. (a, b) \in s \land (b, c) \in r \rightarrow (a, b) \in r)$

where init-seg-of = $\{ (r, s). r \subseteq s \land (\forall a b c. (a, b) \in s \land (b, c) \in r \rightarrow (a, b) \in r) \}$

abbreviation initial-segment-of-syntax :: $\langle \langle a \times a \rangle \times \langle a \times a \rangle \rangle \times bool$

  (infix initial-segment-of-syntax 55)

where $r$ initial-segment-of $s \equiv (r, s) \in \text{init-seg-of}$

lemma refl-on-init-seg-of [simp]: $r$ initial-segment-of $r$

  by (simp add: init-seg-of_def)

lemma trans-init-seg-of:

  $r$ initial-segment-of $s \rightarrow s$ initial-segment-of $t \rightarrow r$ initial-segment-of $t$

  by (simp (no_asm_use) add: init-seg-of_def) blast
lemma antisym-init-seg-of: \( r \text{ initial-segment-of } s \implies s \text{ initial-segment-of } r \implies r = s \)

unfolding init-seg-of-def by safe

lemma Chains-init-seg-of-Union: \( R \in \text{Chains init-seg-of} \implies r \in R \implies r \text{ initial-segment-of } \bigcup R \)

by (auto simp: init-seg-of-def Ball-def Chains-def) blast

lemma chain-subset-trans-Union:
assumes chain \( \subseteq R \land r \in R. \text{ trans r} \)
shows trans (\( \bigcup R \))

proof (intro transI, elim UnionE)
fix \( S1, S2 :: 'a \text{ rel} \) and \( x y z :: 'a \)
assume \( S1 \in R \land S2 \in R \)
with assms(1) have \( S1 \subseteq S2 \lor S2 \subseteq S1 \)
  unfolding chain-subset-def by blast
moreover assume \( (x, y) \in S1 \land (y, z) \in S2 \)
ultimately have \( ((x, y) \in S1 \land (y, z) \in S1) \lor ((x, y) \in S2 \land (y, z) \in S2) \)
  by blast
with \( S1 \in R \land S2 \in R \) assms(2) show \( (x, z) \in \bigcup R \)
  by (auto elim: transE)
qed

lemma chain-subset-antisym-Union:
assumes chain \( \subseteq R \land r \in R. \text{ antisym r} \)
shows antisym (\( \bigcup R \))

proof (intro antisymI, elim UnionE)
fix \( S1, S2 :: 'a \text{ rel} \) and \( x y :: 'a \)
assume \( S1 \in R \land S2 \in R \)
with assms(1) have \( S1 \subseteq S2 \lor S2 \subseteq S1 \)
  unfolding chain-subset-def by blast
moreover assume \( (x, y) \in S1 \land (y, x) \in S2 \)
ultimately have \( ((x, y) \in S1 \land (y, x) \in S1) \lor ((x, y) \in S2 \land (y, x) \in S2) \)
  by blast
with \( S1 \in R \land S2 \in R \) assms(2) show \( x = y \)
  unfolding antisym-def by auto
qed

lemma chain-subset-Total-Union:
assumes chain \( \subseteq R \land \forall r \in R. \text{ Total r} \)
shows Total (\( \bigcup R \))

proof (simp add: total-on-def Ball-def, auto del: disjCI)
fix \( r s a b \)
assume A: \( r \in R \land s \in R \land a \in \text{Field r b} \in \text{Field s} \land a \neq b \)
from \( \text{chain } \subseteq R \) and \( v \in R \land (s \in R. \text{ have } r \subseteq s \lor s \subseteq r) \)
  by (auto simp add: chain-subset-def)
then show \( (\exists r \in R. (a, b) \in r) \lor (\exists r \in R. (b, a) \in r) \)
proof
THEORY “Zorn”

assume \( r \subseteq s \)
then have \((a, b) \in s \lor (b, a) \in s\)
using \( \text{assms}(2) \) \( A \) mono-Field[of \( r s \)]
by (auto simp add: total-on-def)
then show ?thesis
using \( s \in R \) by blast

next
assume \( s \subseteq r \)
then have \((a, b) \in r \lor (b, a) \in r\)
using \( \text{assms}(2) \) \( A \) mono-Field[of \( s r \)]
by (fastforce simp add: total-on-def)
then show ?thesis
using \( r \in R \) by blast
qed

lemma \( \text{wf-Union-wf-init-segs} \): \(\)
assumes \( R \in \text{Chains init-seg-of} \)
and \( \forall r \in R. \text{wf} r \)
shows \( \text{wf} (\bigcup R) \)
proof (simp add: \( \text{wf-iff-no-infinite-down-chain} \), rule ccontr, auto)
fix \( f \)
assume \( 1: \forall i. \exists r \in R. (f (\text{Suc} i), f i) \in r \)
then obtain \( r \) where \( r \in R \) and \( (f (\text{Suc} \emptyset), f \emptyset) \in r \) by auto
have \( (f (\text{Suc} i), f i) \in r \) for \( i \)
proof (induct \( i \))
case \( \emptyset \)
show ?case by fact
next
case \( \text{Suc} i \)
then obtain \( s \) where \( s: s \in R \) \( (f (\text{Suc} (\text{Suc} i)), f(\text{Suc} i)) \in s \)
using \( f \) by auto
then have \( s \) initial-segment-of \( r \) \( \lor \) \( r \) initial-segment-of \( s \)
using \( \text{assms}(1) \) \( \in R \) by (simp add: Chains-def)
with \( \text{Suc} s \) show ?case by (simp add: init-seg-of-def) blast
qed
then show \( \text{False} \)
using \( \text{assms}(2) \) and \( \in R \)
by (simp add: \( \text{wf-iff-no-infinite-down-chain} \) blast)
qed

lemma \( \text{initial-segment-of-Diff} \): \( p \) initial-segment-of \( q \implies p - s \) initial-segment-of \( q - s \)
unfolding init-seg-of-def by blast

lemma \( \text{Chains-init-seg-DiffI} \): \( R \in \text{Chains init-seg-of} \implies \{r - s \mid r, r \in R\} \in \text{Chains init-seg-of} \)
unfolding Chains-def by (blast intro: initial-segment-of-Diff)
theorem well-ordering: \( \exists r . \text{a rel. Well-order } r \land \text{Field } r = \text{UNIV} \)

proof –
  — The initial segment relation on well-orders:
  let \( ?WO = \{ r . \text{a rel. Well-order } r \} \)
  define \( I \) where \( I = \text{init-seg-of } \) \( ?WO \times ?WO \)
  then have \( \text{I-init: } I \subseteq \text{init-seg-of } \) simp
  then have \( \text{subch: } \bigwedge R . R \in \text{Chains } I \implies \text{chain } R \)
  unfolding \text{init-seg-of-def chain-subset-def Chains-def by blast}
  have \( \text{Chains-w0: } \bigwedge r . R \in \text{Chains } I \implies r \in R \implies \text{Well-order } r \)
  by (simp add: Chains-def I-def) blast
  have \( \text{FI: } \text{Field } I = ?WO \)
  by (auto simp add: I-def init-seg-of-def Field-def)
  then have \( 0 : \text{Partial-order } I \)
  by (auto simp: partial-order-on-def preorder-on-def antisym-def antisym-init-seg-of refl-on-def
       trans-def I-def elim!: trans-init-seg-of)
  — \( I \)-chains have upper bounds in \( ?WO \) wrt \( I \): their Union
  have \( \bigcup R \in ?WO \land (\forall r \in R . (r, \bigcup R) \in I ) \) if \( R \in \text{Chains } I \) for \( R \)
  proof –
  from that have \( \text{Ris: } R \in \text{Chains init-seg-of} \)
  using mono-Chains \( [OF \text{ I-init}] \) by blast
  have \( \text{subch: } \text{chain } \subseteq R \)
  using \( \{ R \in \text{Chains } I : \text{I-init by (auto simp: init-seg-of-def chain-subset-def Chains-def) } \}
  have \( \forall r \in R . \text{Refl } r \land \forall r \in R . \text{trans } r \land \forall r \in R . \text{antisym } r \)
  and \( \forall r \in R . \text{Total } r \land \forall r \in R . \text{wf } (r - \text{Id}) \)
  using \( \text{Chains-w0 } [OF : R \in \text{Chains } I ] \) by (simp-all add: order-on-defs)
  have \( \text{Refl } (\bigcup R) \)
  using \( \forall r \in R . \text{Refl } r \)
  unfolding refl-on-def by fastforce
  moreover have \( \text{trans } (\bigcup R) \)
  by (rule chain-subset-trans-Union \( [OF \text{ subch } \forall r \in R . \text{trans } r ] \))
  moreover have \( \text{antisym } (\bigcup R) \)
  by (rule chain-subset-antisym-Union \( [OF \text{ subch } \forall r \in R . \text{antisym } r ] \))
  moreover have \( \text{Total } (\bigcup R) \)
  by (rule chain-subset-Total-Union \( [OF \text{ subch } \forall r \in R . \text{Total } r ] \))
  moreover have \( \text{wf } ((\bigcup R) - \text{Id}) \)
  proof –
  have \( (\bigcup R) - \text{Id} = \bigcup \{ r - \text{Id} \mid r \in R \} \) by blast
  with \( \forall r \in R . \text{wf } (r - \text{Id}) \) and \( \text{wf-Union-wf-init-segs } [OF \text{ Chains-init-seg-I } [OF \text{ Ris}] \)
  show \( ?\text{thesis by fastforce} \)
  qed
  ultimately have \( \text{Well-order } (\bigcup R) \)
  by (simp add: order-on-defs)
  moreover have \( \forall r \in R . \text{r initial-segment-of } \bigcup R \)
  using \( \text{Ris by (simp add: Chains-init-seg-of-Union) } \)
  ultimately show \( ?\text{thesis} \)
  using mono-Chains \( [OF \text{ I-init}] \) \( \text{Chains-w0[of } R \text{ ] and } R \in \text{Chains } I \)
  unfolding \( \text{I-def by blast} \)
/Zorn

then have 1: \exists u \in \text{Field} I. \forall r \in R. (r, u) \in I \text{ if } R \in \text{Chains } I \text{ for } R
  \begin{itemize}
  \item using that by (subst FI) blast
  \end{itemize}
  Zorn's Lemma yields a maximal well-order m:
then obtain m :: 'a rel
  \begin{itemize}
  \item where Well-order m
  \item and max: \forall r. \text{Well-order } r \land (m, r) \in I \rightarrow r = m
  \item using Zorns-po-lemma[OF 0 1] unfolding FI by fastforce
  \end{itemize}
  Now show by contradiction that m covers the whole type:
  have False if x \notin Field m for x :: 'a
  \begin{itemize}
  \item proof
  \end{itemize}
  Assuming that x is not covered and extend m at the top with x
  have m \neq \{\}
  \begin{itemize}
  \item proof
  \end{itemize}
  
  \item assume m = \{}
  
  \item moreover have Well-order \{x, x\}
  \begin{itemize}
  \item by (simp add: order-on-defs refl-on-def trans-def antisym-def total-on-def Field-def)
  \end{itemize}
  ultimately show False using max
  \begin{itemize}
  \item by (auto simp: I-def init-seg-of-def simp del: Field-insert)
  \end{itemize}
  qed
  then have Field m \neq \{\}
  \begin{itemize}
  \item by (auto simp: Field-def)
  \end{itemize}
  moreover have wf (m - Id)
  \begin{itemize}
  \item using (Well-order m) by (simp add: well-order-on-def)
  \end{itemize}
  The extension of m by x:
  let \?s = \{(a, x) | a. a \in Field m\}
  let \?m = insert (x, x) m \cup \?s
  have Fm: Field \?m = insert x (Field m)
  \begin{itemize}
  \item by (auto simp: Field-def)
  \end{itemize}
  have Refl m and trans m and antisym m and Total m and wf (m - Id)
  \begin{itemize}
  \item using (Well-order m) by (simp-all add: order-on-defs)
  \end{itemize}
  We show that the extension is a well-order
  have Refl \?m
  \begin{itemize}
  \item using (Refl m) Fm unfolding refl-on-def by blast
  \end{itemize}
  moreover have trans \?m using (trans m) and (x \notin Field m)
  unfolding trans-def Field-def by blast
  moreover have antisym \?m
  \begin{itemize}
  \item using (antisym m) and (x \notin Field m) unfolding antisym-def Field-def by blast
  \end{itemize}
  moreover have Total \?m
  \begin{itemize}
  \item using (Total m) and Fm by (auto simp: total-on-def)
  \end{itemize}
  moreover have wf (\?m - Id)
  \begin{itemize}
  \item proof
  \end{itemize}
  have wf \?s
  \end{itemize}
  using \?x \notin Field m by (auto simp: wf-eq-minimal Field-def Bex-def)
  then show \?thesis
  \begin{itemize}
  \item using (wf (m - Id)) and \?x \notin Field m) wf-subset [OF (wf \?s) Diff-subset]
  \item by (auto simp: Un-Diff Field-def intro: wf-Un)
  \end{itemize}
  qed
ultimately have Well-order \( ?m \)
  by (simp add: order-on-defs)
— We show that the extension is above \( m \)
  moreover have \( (m, \ ?m) \in I \)
    using (Well-order ?m) and (Well-order m) and \( x \notin \text{Field} \ m \)
    by (fastforce simp: I-def init-seg-of-def Field-def)
ultimately
— This contradicts maximality of \( m \):
  show False
    using max and \( (x \notin \text{Field} \ m) \) unfolding Field-def by blast
qed
then have Field \( m = \text{UNIV} \) by auto
with (Well-order m) show ?thesis by blast
qed

**corollary** well-order-on: \( \exists \ r :: \ a \ rel. \ \text{well-order-on} \ A \ r \)

**proof**
  obtain \( r :: \ a \ rel \) where wo: Well-order \( r \) and univ: Field \( r = \text{UNIV} \)
    using well-ordering [where \( \ 'a = \ 'a \)] by blast
  let \( \{x, y\} : x \in A \land y \in A \land (x, y) \in r \}
  have 1: Field \( \ ?r = A \)
    using wo univ by (fastforce simp: Field-def order-on-defs refl-on-def)
  from (Well-order \( r \)) have Refl \( r \) trans \( r \) antisym \( r \) Total \( r \) wf \( (r = \text{Id}) \)
    by (simp-all add: order-on-defs)
  from (Refl \( r \)) have Refl \( \ ?r \)
    by (auto simp refl-on-def 1 univ)
  moreover from (trans \( r \)) have trans \( \ ?r \)
    unfolding trans-def by blast
  moreover from (antisym \( r \)) have antisym \( \ ?r \)
    unfolding antisym-def by blast
  moreover from (Total \( r \)) have Total \( \ ?r \)
    by (simp add: total-on-def 1 univ)
  moreover have \( \text{wf} \ (\ ?r = \text{Id}) \)
    by (rule wf-subset [OF \( \text{wf} \ (\text{Id}) \)]) blast
ultimately have Well-order \( \ ?r \)
    by (simp add: order-on-defs)
  with 1 show ?thesis by auto
qed

**lemma** wfrec-def-adm: \( \ f \equiv \text{wfrec} \ R \ F \longrightarrow \text{wf} \ R \longrightarrow \text{adm-wf} \ R \ F \longrightarrow \ f \ = \ F \ f \)
using wfrec-fixpoint by simp

**lemma** dependent-wf-choice:
  fixes \( P :: (\ 'a \Rightarrow \ 'b) \Rightarrow \ 'a \Rightarrow \ 'b \Rightarrow \text{bool} \)
  assumes \( \text{wf} \ R \)
  and \( \text{adm}: \{x \in R : \forall z. \ (\{z, x\} \in R \longrightarrow f \ z = g \ z\) \longrightarrow P f x = P g x\} \)
  and \( P: \{x f \in R : \forall y. \ (\forall y. \ (y, x) \in R \longrightarrow P f y (f y)) \longrightarrow \exists \ r. \ P f x r\}

THEORY “Zorn”
shows $\exists f. \forall x. P f x (f x)$
proof (intro exI allI)
  fix x
define $f$ where $f \equiv \text{wfree } R (\lambda f x. \text{SOME } r. P f x r)$
from (wf $R$) show $P f x (f x)$
proof (induct x)
  case (less x)
  show $P f x (f x)$
  proof (subst (2) wfree-def-adm[OF $f$-def $\langle \text{wf } R \rangle$])
    show $\text{adm-wf } R (\lambda f x. \text{SOME } r. P f x r)$
      by (auto simp add: adm-wf-def intro: arg-cong[where $f$=Eps] ext adm)
    show $P f x (\text{Eps } (P f x))$
      using $P$ by (rule someI-ex) fact
  qed
qed
qed
qed

definitions under
where under $\equiv$ Order-Relation.
under $r$
definitions underS
where underS $\equiv$ Order-Relation.
underS $r$
definitions Under
where Under $\equiv$ Order-Relation.
Under $r$
definitions UnderS
where UnderS $\equiv$ Order-Relation.
UnderS $r$
abbreviation aboveS where aboveS ≡ Order-Relation.aboveS r
abbreviation Above where Above ≡ Order-Relation.Above r
abbreviation AboveS where AboveS ≡ Order-Relation.AboveS r
abbreviation ofilter where ofilter ≡ Order-Relation.ofilter r
lemmas ofilter-def = Order-Relation.ofilter-def[of r]

25.1 Auxiliaries

lemma REFL: Refl r
using WELL order-on-defs[of - r] by auto

lemma TRANS: trans r
using WELL order-on-defs[of - r] by auto

lemma ANTISYM: antisym r
using WELL order-on-defs[of - r] by auto

lemma TOTAL: Total r
using WELL order-on-defs[of - r] by auto

lemma TOTALS: ∀ a ∈ Field r. ∀ b ∈ Field r. (a,b) ∈ r ∨ (b,a) ∈ r
using REFL TOTAL refl-on-def[of - r] total-on-def[of - r] by force

lemma LIN: Linear-order r
using WELL well-order-on-def[of - r] by auto

lemma WF: wf (r − Id)
using WELL well-order-on-def[of - r] by auto

lemma cases-Total:
∧ phi a b. [(a,b) <= Field r; ((a,b) ∈ r ⇒ phi a b); ((b,a) ∈ r ⇒ phi a b)]
⇒ phi a b
using TOTALS by auto

lemma cases-Total3:
∧ phi a b. [(a,b) ≤ Field r; ((a,b) ∈ r − Id ∨ (b,a) ∈ r − Id ⇒ phi a b);
(a = b ⇒ phi a b)] ⇒ phi a b
using TOTALS by auto

25.2 Well-founded induction and recursion adapted to non-strict well-order relations

Here we provide induction and recursion principles specific to non-strict well-order relations. Although minor variations of those for well-founded relations, they will be useful for doing away with the tediousness of having to take out the diagonal each time in order to switch to a well-founded relation.

lemma well-order-induct:
assumes IND: \( \forall x. \forall y. y \neq x \land (y, x) \in r \rightarrow P y \rightarrow P x \)
shows \( P a \)
proof –
  have \( \forall x. \forall y. (y, x) \in r - Id \rightarrow P y \rightarrow P x \)
  using IND by blast
  thus \( P a \) using WF wf-induct[of \( r - Id P a \)] by blast
qed

definition
\( \text{worec} :: (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \rightarrow 'b \)
where
\( \text{worec} F \equiv \text{wfrec} (r - Id) F \)

definition
\( \text{adm-wo} :: (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow \text{bool} \)
where
\( \text{adm-wo} H \equiv \forall f g x. (\forall y \in \text{underS} x. f y = g y) \rightarrow H f x = H g x \)

lemma worec-fixpoint:
assumes \( \text{ADM}: \text{adm-wo} H \)
shows \( \text{worec} H = H (\text{worec} H) \)
proof –
  let \( ?rS = r - Id \)
  have \( \text{adm-wf} (r - Id) H \)
  unfolding \( \text{adm-wf-def} \)
  using \( \text{ADM} \text{adm-wo-def[of } H \} \text{underS-def[of } r \} \) by auto
  hence \( \text{wfrec } ?rS H = H (\text{wfrec } ?rS H) \)
  using \( WF \text{wfrec-fixpoint[of } ?rS H \} \) by simp
  thus \( ?\text{thesis} \) unfolding \( \text{worec-def} \).
qed

25.3 The notions of maximum, minimum, supremum, successor and order filter

We define the successor of a set, and not of an element (the latter is of course a particular case). Also, we define the maximum of two elements, \( \text{max2} \), and the minimum of a set, \( \text{minim} \) – we chose these variants since we consider them the most useful for well-orders. The minimum is defined in terms of the auxiliary relational operator \( \text{isMinim} \). Then, supremum and successor are defined in terms of minimum as expected. The minimum is only meaningful for non-empty sets, and the successor is only meaningful for sets for which strict upper bounds exist. Order filters for well-orders are also known as “initial segments”.

definition \( \text{max2} :: 'a \Rightarrow 'a \Rightarrow 'a \)
where \( \text{max2} a b \equiv \text{if } (a, b) \in r \text{ then } b \text{ else } a \)

definition \( \text{isMinim} :: 'a \text{ set } \Rightarrow 'a \Rightarrow \text{bool} \)
where \( \text{isMinim} A b \equiv b \in A \land (\forall a \in A. (b, a) \in r) \)
definition minim :: 'a set ⇒ 'a
where minim A ≡ THE b. isMinim A b

definition supr :: 'a set ⇒ 'a
where supr A ≡ minim (Above A)

definition suc :: 'a set ⇒ 'a
where suc A ≡ minim (AboveS A)

25.3.1 Properties of max2

lemma max2-greater-among:
assumes a ∈ Field r and b ∈ Field r
shows (a, max2 a b) ∈ r ∧ (b, max2 a b) ∈ r ∧ max2 a b ∈ {a,b}
proof–
{assume (a,b) ∈ r
  hence ?thesis using max2-def assms REFL refl-on-def
  by (auto simp add: refl-on-def)
}
moreover
{assume a = b
  hence (a,b) ∈ r using REFL assms
  by (auto simp add: refl-on-def)
}
moreover
{assume ∗: a ≠ b ∧ (b,a) ∈ r
  hence (a,b) /∈ r using ANTSYM
  by (auto simp add: antisym-def)
  hence ?thesis using ∗ max2-def assms REFL refl-on-def
  by (auto simp add: refl-on-def)
}
ultimately show ?thesis using assms TOTAL
  total-on-def[of Field r r] by blast
qed

lemma max2-greater:
assumes a ∈ Field r and b ∈ Field r
shows (a, max2 a b) ∈ r ∧ (b, max2 a b) ∈ r
using assms by (auto simp add: max2-greater-among)

lemma max2-among:
assumes a ∈ Field r and b ∈ Field r
shows max2 a b ∈ {a, b}
using assms max2-greater-among[of a b] by simp

lemma max2-equals1:
assumes a ∈ Field r and b ∈ Field r
shows (max2 a b = a) = ((b,a) ∈ r)
using assms ANTISYM unfolding antisym-def using TOTALS
by (auto simp add: max2-def max2-among)

lemma max2-equals2:
assumes a ∈ Field r and b ∈ Field r
shows (max2 a b = b) = ((a, b) ∈ r)
using assms ANTISYM unfolding antisym-def using TOTALS
unfolding max2-def by auto

25.3.2 Existence and uniqueness for isMinim and well-definedness of minim

lemma isMinim-unique:
assumes MINIM: isMinim B a and MINIM’: isMinim B a’
shows a = a’
proof −
{ have a ∈ B
  using MINIM isMinim-def by simp
  hence (a’, a) ∈ r
  using MINIM’ isMinim-def by simp
}
moreover
{ have a’ ∈ B
  using MINIM’ isMinim-def by simp
  hence (a, a’) ∈ r
  using MINIM isMinim-def by simp
}
ultimately
show ?thesis using ANTISYM antisym-def[of r] by blast
qed

lemma Well-order-isMinim-exists:
assumes SUB: B ≤ Field r and NE: B ≠ {}
shows ∃ b. isMinim B b
proof −
from spec[OF WF] unfolded wf-eq-minimal[of r - Id], of B] NE obtain b where
*: b ∈ B ∧ (∀ b’. b’ ≠ b ∧ (b’, b) ∈ r → b’ ∉ B) by auto
show ?thesis
proof (simp add: isMinim-def, rule exI[of - b], auto)
  show b ∈ B using * by simp
next
  fix b’ assume As: b’ ∈ B
  hence **: b ∈ Field r ∧ b’ ∈ Field r using As SUB * by auto
from As * have b’ = b ∨ (b’, b) ∉ r by auto
moreover
{ assume b’ = b
  hence (b, b’) ∈ r
  using ** REFL by (auto simp add: refl-on-def)
moreover
{assume \(b' \neq b \land (b',b) \notin r\)
  hence \((b,b') \in r\)
  using ** TOTAL by (auto simp add: total-on-def)
}
ultimately show \((b,b') \in r\) by blast
qed

lemma minim-isMinim:
assumes \(SUB: B \leq \text{Field } r \text{ and } NE: B \neq \{\}\)
shows isMinim B (minim B)
proof−
  let \(?phi = (\lambda b. \text{isMinim } B b)\)
  from assms Well-order-isMinim-exists
  obtain b where *: ?phi b by blast
moreover
  have \(\bigwedge b'. \ ?phi b' \implies b' = b\)
  using isMinim-unique * by auto
ultimately show ?thesis
  unfolding minim-def using theI[of ?phi b] by blast
qed

25.3.3 Properties of minim

lemma minim-in:
assumes \(B \leq \text{Field } r \text{ and } B \neq \{\}\)
shows minim B \(\in B\)
proof−
  from minim-isMinim[of B] assms
  have isMinim B (minim B) by simp
  thus ?thesis by (simp add: isMinim-def)
qed

lemma minim-inField:
assumes \(B \leq \text{Field } r \text{ and } B \neq \{\}\)
shows minim B \(\in \text{Field } r\)
proof−
  have minim B \(\in B\) using assms by (simp add: minim-in)
  thus ?thesis using assms by blast
qed

lemma minim-least:
assumes \(SUB: B \leq \text{Field } r \text{ and } IN: b \in B\)
shows \((\text{minim } B, b) \in r\)
proof−
  from minim-isMinim[of B] assms
  have isMinim B (minim B) by auto
thus \( \text{thesis} \) by (auto simp add: isMinim-def IN)

qed

lemma equals-minim:
assumes SUB: \( B \subseteq \text{Field } r \) and IN: \( a \in B \) and
\[ \text{LEAST: } \bigwedge b. b \in B \implies (a,b) \in r \]
shows \( a = \text{minim } B \)
proof –
  from minim-isMinim[of B] assms
  have isMinim B (minim B) by auto
  moreover have isMinim B a using IN LEAST isMinim-def by auto
  ultimately show \( \text{thesis} \)
    using isMinim-unique by auto
qed

25.3.4 Properties of successor

lemma suc-AboveS:
assumes SUB: \( B \subseteq \text{Field } r \) and ABOVES: \( \text{AboveS } B \neq \{\} \)
shows suc B \( \in \text{AboveS } B \)
proof (unfold suc-def)
  have AboveS B \( \subseteq \text{Field } r \)
    using AboveS-Field[of r] by auto
  thus minim (AboveS B) \( \in \text{AboveS } B \)
    using assms by (simp add: minim-in)
qed

lemma suc-greater:
assumes SUB: \( B \subseteq \text{Field } r \) and ABOVES: \( \text{AboveS } B \neq \{\} \) and 
IN: \( b \in B \)
shows suc B \( \neq b \wedge (b, \text{suc } B) \in r \)
proof –
  from assms suc-AboveS
  have suc B \( \in \text{AboveS } B \) by simp
  with IN AboveS-def[of r] show \( \text{thesis} \) by simp
qed

lemma suc-least-AboveS:
assumes ABOVES: \( a \in \text{AboveS } B \)
shows (suc B,a) \( \in r \)
proof (unfold suc-def)
  have AboveS B \( \subseteq \text{Field } r \)
    using AboveS-Field[of r] by auto
  thus (minim (AboveS B),a) \( \in r \)
    using assms minim-least by simp
qed

lemma suc-inField:
assumes B \( \subseteq \text{Field } r \) and AboveS B \( \neq \{\} \)
shows \( \text{suc } B \in \text{Field } r \)
proof –
  have \( \text{suc } B \in \text{AboveS } B \) using \text{suc-AboveS} assms by simp
  thus \(?\text{thesis}\) using assms AboveS-Field[of \( r \)] by auto
qed

lemma equals-suc-AboveS:
assumes \( \text{SUB}: B \leq \text{Field } r \) and \( \text{ABV}: a \in \text{AboveS } B \) and
  \( \text{MINIM}: \bigwedge a', a' \in \text{AboveS } B \Longrightarrow (a, a') \in r \)
shows \( a = \text{suc } B \)
proof (unfold suc-def)
  have \( \text{AboveS } B \leq \text{Field } r \)
  using AboveS-Field[of \( r \) \( B \)] by auto
  thus \( a = \text{minim } (\text{AboveS } B) \)
  using assms equals-minim by simp
qed

lemma suc-underS:
assumes \( \text{IN}: a \in \text{Field } r \)
shows \( a = \text{suc } (\text{underS } a) \)
proof –
  have \( \text{underS } a \leq \text{Field } r \)
  using underS-Field[of \( r \)] by auto
  moreover
  have \( a \in \text{AboveS } (\text{underS } a) \)
  using in-AboveS-underS IN by fast
  moreover
  have \( \forall a' \in \text{AboveS } (\text{underS } a). (a, a') \in r \)
  proof (clarify)
    fix \( a' \)
    assume \(*: a' \in \text{AboveS } (\text{underS } a)\)
    hence \(**: a' \in \text{Field } r\)
    using AboveS-Field by fast
    \{ assume \( (a, a') \notin r \)
    hence \( a' = a \lor (a', a) \in r \)
    using TOTAL IN ** by (auto simp add: total-on-def)
    moreover
    \{ assume \( a' = a \)
    hence \( (a, a') \in r \)
    using REFL IN ** by (auto simp add: refl-on-def)
    \}
    moreover
    \{ assume \( a' \neq a \land (a', a) \in r \)
    hence \( a' \in \text{underS } a \)
    unfolding underS-def by simp
    hence \( a' \notin \text{AboveS } (\text{underS } a) \)
    using AboveS-disjoint by fast
  \}
with * have False by simp

ultimately have \((a, a') \in r\) by blast

thus \((a, a') \in r\) by blast

qed

ultimately show \(?thesis\)
using equals-suc-AboveS by auto

qed

25.3.5 Properties of order filters

lemma under-ofilter:
ofilter (under a)
proof (unfold ofilter-def under-def, auto simp add: Field-def)
fix aa x
assume \((aa, a) \in r\) \((x, aa) \in r\)
thus \((x, a) \in r\)
using TRANS trans-def[of r] by blast

qed

lemma underS-ofilter:
ofilter (underS a)
proof (unfold ofilter-def underS-def under-def, auto simp add: Field-def)
fix aa x
assume \((a, aa) \in r\) \((aa, a) \in r\) and DIFF: \(aa \neq a\)
thus False
using ANTISYM antisym-def[of r] by blast

next
fix aa x
assume \((aa, a) \in r\) \(aa \neq a\) \((x, aa) \in r\)
thus \((x, a) \in r\)
using TRANS trans-def[of r] by blast

qed

lemma Field-ofilter:
ofilter (Field r)
by (unfold ofilter-def under-def, auto simp add: Field-def)

lemma ofilter-underS-Field:
ofilter A = \(((\exists a \in Field r. A = \text{underS } a) \lor (A = \text{Field } r))\)
proof
assume \((\exists a \in Field r. A = \text{underS } a) \lor A = \text{Field } r\)
thus ofilter A
by (auto simp: underS-ofilter Field-ofilter)

next
assume *: ofilter A
let ?One = \((\exists a \in Field r. A = \text{underS } a)\)
let ?Two = \((A = \text{Field } r)\)
show ?One \lor ?Two
proof (cases ?Two, simp)
  let ?B = (Field r) - A
  let ?a = minim ?B

assume A ≠ Field r
moreover have A ≤ Field r using * ofilter-def by simp
ultimately have 1: ?B ≠ {} by blast

hence 2: ?a ∈ Field r using minim-inField[of ?B] by blast
hence 4: ?a ∉ A by blast

have 5: A ≤ Field r using * ofilter-def by auto

moreover
have A = underS ?a
proof
  show A ≤ underS ?a
  proof (unfold underS-def, auto simp add: 4)
    fix x assume **: x ∈ A
    hence 11: x ∈ Field r using 5 by auto
    have 12: x ≠ ?a using 4 ** by auto
    have 13: under x ≤ A using * ofilter-def ** by auto
      {assume (x, ?a) /∈ r
        hence (?a, x) ∈ r
          using TOTAL total-on-def[of Field r r]
          2 4 11 12 by auto
        hence ?a ∈ under x using under-def[of r] by auto
        hence ?a ∈ A using ** 13 by blast
        with 4 have False by simp
      }
    thus (x, ?a) ∈ r by blast
  qed
next
  show underS ?a ≤ A
  proof (unfold underS-def, auto)
    fix x
    assume **: x ≠ ?a and ***: (x, ?a) ∈ r
    hence 11: x ∈ Field r using Field-def by fastforce
      {assume x ≠ A
        hence x ∈ ?B using 11 by auto
        hence (?a, x) ∈ r using 3 minim-least[of ?B x] by blast
        hence False
          using ANTISYM antisym-def[of r] ** *** by auto
      }
    thus x ∈ A by blast
  qed

ultimately have ?One using 2 by blast
thus ?thesis by simp
qed
lemma ofilter-UNION:
(∀ i. i ∈ I ⇒ ofilter(A i)) ⇒ ofilter (⋃ i ∈ I. A i)
unfolding ofilter-def by blast

lemma ofilter-under-UNION:
assumes ofilter A
shows A = (⋃ a ∈ A. under a)
proof
  have ∀ a ∈ A. under a ≤ A
  using assms ofilter-def by auto
  thus (⋃ a ∈ A. under a) ≤ A by blast
next
  have ∀ a ∈ A. a ∈ under a
  using REFL Refl-under-in[of r] assms ofilter-def[of A] by blast
  thus A ≤ (⋃ a ∈ A. under a) by blast
qed

25.3.6 Other properties

lemma ofilter-linord:
assumes OF1: ofilter A and OF2: ofilter B
shows A ≤ B ∨ B ≤ A
proof(cases A = Field r)
  assume Case1: A = Field r
  hence B ≤ A using OF2 ofilter-def by auto
  thus ?thesis by simp
next
  assume Case2: A ≠ Field r
  with ofilter-underS-Field OF1 obtain a where
  1: a ∈ Field r ∧ A = underS a by auto
  show ?thesis
  proof(cases B = Field r)
    assume Case21: B = Field r
    hence A ≤ B using OF1 ofilter-def by auto
    thus ?thesis by simp
next
  assume Case22: B ≠ Field r
  with ofilter-underS-Field OF2 obtain b where
  2: b ∈ Field r ∧ B = underS b by auto
  have a = b ∨ (a,b) ∈ r ∨ (b,a) ∈ r
  using 1 2 TOTAL total-on-def[of r - r] by auto
  moreover
  {assume a = b with 1 2 have ?thesis by auto
   }
  moreover
  {assume (a,b) ∈ r
   with underS-incr[of r] TRANS ANTISYM 1 2
   have A ≤ B by auto
  }
hence thesis by auto

moreover

assume \((b,a) \in r\)

with \(\text{underS-incr[of } r\) TRANS ANTISYM I 2

have \(B \leq A\) by auto

hence thesis by auto

ultimately show thesis by blast

qed

lemma ofilter-AboveS-Field:

assumes ofilter \(A\)

shows \(A \cup (\text{AboveS } A) = \text{Field } r\)

proof

show \(A \cup (\text{AboveS } A) \leq \text{Field } r\)

using assms ofilter-def AboveS-Field[of \(r\)] by auto

next

{ fix \(x\) assume \(*\): \(x \in \text{Field } r\) and \(**\): \(x \notin A\)

fix \(y\) assume \(***\): \(y \in A\)

with ** have 1: \(y \neq x\) by auto

{ assume \((y,x) \notin r\)

moreover

have \(y \in \text{Field } r\) using assms ofilter-def \(***\) by auto

ultimately have \((x,y) \in r\)

using 1 * TOTAL total-on-def[of - \(r\)] by auto

with *** assms ofilter-def under-def[of \(r\)] have \(x \in A\) by auto

with ** have False by contradiction

} hence \((y,x) \in r\) by blast

with 1 have \(y \neq x \land (y,x) \in r\) by auto

} with * have \(x \in \text{AboveS } A\) unfolding AboveS-def by auto

} thus \(\text{Field } r \leq A \cup (\text{AboveS } A)\) by blast

qed

lemma suc-ofilter-in:

assumes \(OF\): ofilter \(A\) and \(ABOVE-NE\): \(\text{AboveS } A \neq \{\}\) and

\(REL\): \((b,\text{suc } A) \in r\) and \(DIFF\): \(b \neq \text{suc } A\)

shows \(b \in A\)

proof

have *: \(\text{suc } A \in \text{Field } r \land b \in \text{Field } r\)

using WELL REL well-order-on-domain[of Field \(r\)] by auto

{ assume **: \(b \notin A\)

hence \(b \in \text{AboveS } A\)

using \(OF\) * ofilter-AboveS-Field by auto

hence \((\text{suc } A, b) \in r\)
using suc-least-AboveS by auto
hence False using REL DIFF ANTISYM *
by (auto simp add: antisym-def)
}
thus ?thesis by blast
qed

26  Well-Order Embeddings as Needed by Bounded Natural Functors

theory BNF-Wellorder-Embedding
imports Hilbert-Choice BNF-Wellorder-Relation
begin

In this section, we introduce well-order embeddings and isomorphisms and prove their basic properties. The notion of embedding is considered from the point of view of the theory of ordinals, and therefore requires the source to be injected as an initial segment (i.e., order filter) of the target. A main result of this section is the existence of embeddings (in one direction or another) between any two well-orders, having as a consequence the fact that, given any two sets on any two types, one is smaller than (i.e., can be injected into) the other.

26.1  Auxiliaries

lemma UNION-inj-on-ofilter:
assumes WELL: Well-order r and
OF: \( \bigwedge i. i \in I \Rightarrow \text{wo-rel.ofilter} \ r \ (A i) \) and
INJ: \( \bigwedge i. i \in I \Rightarrow \text{inj-on} \ f \ (A i) \)
shows inj-on f (\( \bigcup i \in I. A i \))
proof
  have wo-rel r using WELL by (simp add: wo-rel-def)
  hence \( \bigwedge i j. [i \in I; j \in I] \Rightarrow A i \leq A j \lor A j \leq A i \)
  using wo-rel.ofilter-linord[of r] OF by blast
  with WELL INV show ?thesis
  by (auto simp add: inj-on-UNION-chain)
qed

lemma under-underS-bij-betw:
assumes WELL: Well-order r and WELL': Well-order r' and
IN: a ∈ Field r and IN': f a ∈ Field r' and
BIJ: bij-betw f (underS r a) (underS r'(f a))
shows bij-betw f (under r a) (under r'(f a))
proof
  have \( a \notin \text{under}_S r a \land f a \notin \text{under}_S r' (f a) \)
  unfolding \text{under}_S \text{-def} by auto
  moreover
  \{ have \text{Refl} r \land \text{Refl} r' \text{ using WELL WELL'}
  by (auto simp add: \text{order-on-defs})
  hence \( \text{under}_r a = \text{under}_S r a \cup \{ a \} \land \)
  \( \text{under}_r' (f a) = \text{under}_S r' (f a) \cup \{ f a \} \)
  using \text{IN IN'} by(auto simp add: \text{Refl-under-under}_S)
  \}
  ultimately show \( \text{thesis} \)
  using \text{BIJ notIn-Un-bij-betw[of a under}_S r a f \text{ under}_S r' (f a)] \text{ by auto}
qed

26.2 (Well-order) embeddings, strict embeddings, isomorphisms and order-compatible functions

Standardly, a function is an embedding of a well-order in another if it injectively and order-compatibly maps the former into an order filter of the latter. Here we opt for a more succinct definition (operator \text{embed}), asking that, for any element in the source, the function should be a bijection between the set of strict lower bounds of that element and the set of strict lower bounds of its image. (Later we prove equivalence with the standard definition – lemma \text{embed-iff-compat-inj-on-ofilter}.) A strict embedding (operator \text{embedS}) is a non-bijective embedding and an isomorphism (operator \text{iso}) is a bijective embedding.

definition \text{embed :: 'a rel \Rightarrow 'a' rel \Rightarrow ('a \Rightarrow 'a')} \Rightarrow bool
where
\text{embed r r' f \equiv \forall a \in Field r. bij-betw f (under r a) (under r' (f a))}

lemmas \text{embed-defs = embed-def embed-def[abs-def]}

Strict embeddings:
definition \text{embedS :: 'a rel \Rightarrow 'a' rel \Rightarrow ('a \Rightarrow 'a')} \Rightarrow bool
where
\text{embedS r r' f \equiv embed r r' f \land \sim bij-betw f (Field r) (Field r')}

lemmas \text{embedS-defs = embedS-def embedS-def[abs-def]}

definition \text{iso :: 'a rel \Rightarrow 'a' rel \Rightarrow ('a \Rightarrow 'a')} \Rightarrow bool
where
\text{iso r r' f \equiv embed r r' f \land bij-betw f (Field r) (Field r')}

lemmas \text{iso-defs = iso-def iso-def[abs-def]}

definition \text{compat :: 'a rel \Rightarrow 'a' rel \Rightarrow ('a \Rightarrow 'a')} \Rightarrow bool
where
compat \ r \ r' \ f \equiv \forall \ a \ b. \ (a,b) \in \ r \rightarrow (f \ a, f \ b) \in r'

**lemma** compat-wf:
**assumes** CMP: compat \ r \ r' \ f \ and WF: wf \ r'
**shows** wf \ r

**proof**
- have \ r \leq \ inv-image \ r' \ f
  unfolding inv-image-def using CMP
  by (auto simp add: compat-def)
  with WF show \ ?thesis
  using wf-inv-image[of \ r' \ f] wf-subset[of inv-image \ r' \ f] by auto
qed

**lemma** id-embed: embed \ r \ r id
by(auto simp add: id-def embed-def bij-betw-def)

**lemma** id-iso: iso \ r \ r id
by(auto simp add: id-def embed-def iso-def bij-betw-def)

**lemma** embed-in-Field:
**assumes** WELL: Well-order \ r \ and
  EMB: embed \ r \ r' \ f \ and IN: a \in Field \ r
**shows** f \ a \in Field \ r'

**proof**
- have \ Well: wo-rel \ r
  using WELL by (auto simp add: wo-rel-def)
  hence \ f: Reft \ r
  by (auto simp add: wo-rel.REFL)
  hence \ a \in under \ r \ a \ using \ IN \ Reft-under-in \ by \ fastforce
  hence \ f \ a \in under \ r' \ (f \ a)
  using EMB \ IN \ by \ (auto simp add: embed-def bij-betw-def)
  thus \ ?thesis unfolding Field-def
  by (auto simp: under-def)
qed

**lemma** comp-embed:
**assumes** WELL: Well-order \ r \ and
  EMB: embed \ r \ r' \ f \ and EMB': embed \ r' \ r'' \ f'
**shows** embed \ r \ r'' \ (f' \circ \ f)

**proof**
- unfold embed-def, auto
  fix \ a \ assume \ *: \ a \in Field \ r
  hence bij-btw \ f \ (under \ r \ a) \ (under \ r' \ (f \ a))
  using embed-def[of \ r'] \ EMB \ by \ auto
  moreover
  { have \ f \ a \in Field \ r'
    using EMB \ WELL \ * \ by \ (auto simp add: embed-in-Field)
    hence bij-btw \ f' \ (under \ r' \ (f \ a)) \ (under \ r'' \ (f' \ (f \ a)))
    using embed-def[of \ r''] \ EMB' \ by \ auto
  }
ultimately
\[
\text{show } \text{bij-betw } (f' \circ f) \text{ (under } r \text{ a) (under } r'' \text{ (f'(f a)))}
\]
\[\text{by (auto simp add: bij-betw-trans)}\]
\text{qed}

\text{lemma comp-iso:}
\text{assumes WELL: Well-order } r \text{ and EMB: iso } r r' f \text{ and EMB': iso } r' r'' f'
\text{shows iso } r r'' (f' \circ f)
\text{by (auto simp add: comp-embed bij-betw-trans)}

That embedS is also preserved by function composition shall be proved only later.

\text{lemma embed-Field:}
\text{assumes WELL: Well-order } r \text{ and WELL': Well-order } r' \text{ and EMB: embed } r r' f \text{ and OF: wo-rel.ofilter } r A
\text{shows wo-rel.ofilter } r' (f'A)
\text{proof–}

\text{from WELL have Well: wo-rel } r \text{ unfolding wo-rel-def .}
\text{from WELL' have Well': wo-rel } r' \text{ unfolding wo-rel-def .}
\text{from OF have : } A \subseteq \text{ Field } r \text{ by (auto simp add: Well wo-rel.ofilter-def)}

\text{show ?thesis using Well' WELL EMB 0 embed-Field[of } r \text{ r' f]}
\text{proof (unfold wo-rel.ofilter-def, auto simp add: image-def)}
\text{fix } a \text{ b'}
\text{assume *: } a \in A \text{ and **: } b' \in \text{ under } r' (f a)
\text{hence } a \in \text{ Field } r \text{ using } \theta \text{ by auto}
\text{hence bij-betw } f \text{ (under } r a) \text{ (under } r' \text{ (f a))}
\text{using * EMB by (auto simp add: embed-def)}
\text{hence } f'(\text{under } r a) = \text{ under } r' (f a)
\text{by (simp add: bij-betw-def)}
\text{with ** image-def[of } f \text{ under } r a] \text{ obtain } b \text{ where}
1: } b \in \text{ under } r a \land b' = f b \text{ by blast}
\text{hence } b \in A \text{ using Well * OF}
\text{by (auto simp add: wo-rel.ofilter-def)}
\text{with 1 show } \exists b \in A. b' = f b \text{ by blast}
\text{qed}
\text{qed}

\text{lemma embed-Field-ofilter:}
\text{assumes WELL: Well-order } r \text{ and WELL': Well-order } r' \text{ and EMB: embed } r r' f
\text{shows wo-rel.ofilter } r' (f'(\text{Field } r))
proof
  have wo-rel_ofilter r (Field r)
  using WELL by (auto simp add: wo-rel_def wo-rel.Field-ofilter)
  with WELL WELL' EMB
  show thesis by (auto simp add: embed-preserves-ofilter)
qed

lemma embed-compat:
assumes EMB: embed r r' f
shows compat r r' f
proof (unfold compat-def, clarify)
  fix a b
  assume *(a,b) ∈ r
  hence 1: b ∈ Field r using Field-def[of r] by blast
  have a ∈ under r b
    using * under-def[of r] by simp
    hence f a ∈ under r' (f b)
    using EMB embed-def[of r r' f]
    bij-betw-def[of f under r b under r' (f b)]
    image-def[of f under r b] 1 by auto
  thus (f a, f b) ∈ r'
    by (auto simp add: under-def)
qed

lemma embed-inj-on:
assumes WELL: Well-order r and EMB: embed r r' f
shows inj-on f (Field r)
proof (unfold inj-on-def, clarify)
  from WELL have Well: wo-rel r unfolding wo-rel-def
    with wo-rel.TOTAL[of r]
  have Total: Total r by simp
  from Well wo-rel.REFL[of r]
  have Refl: Refl r by simp
  fix a b
  assume *: a ∈ Field r and **: b ∈ Field r and
    ***: f a = f b
  hence 1: a ∈ Field r ∧ b ∈ Field r
    unfolding Field-def by auto
    {assume (a,b) ∈ r
     hence a ∈ under r b ∧ b ∈ under r b
     using Refl by(auto simp add: under-def refl-on-def)
     hence a = b
     using EMB 1 ***
     by (auto simp add: embed-def bij-betw-def inj-on-def)
    }
  moreover
    {assume (b,a) ∈ r
hence $a \in \text{under } r \ a \land b \in \text{under } r \ a$
using $\text{Refl}$
by (auto simp add: under-def refl-on-def)
hence $a = b$
using $\text{EMB} \ 1$ ***
by (auto simp add: embed-def bij-betw-def inj-on-def)
}\}
ultimately
show $a = b$
using $\text{Total} \ 1$
by (auto simp add: total-on-def)
qed

lemma embed-underS:
assumes WELL: Well-order $r$ and WELL': Well-order $r'$ and
$\text{EMB}$: embed $r \ r' \ f$ and $\text{IN}$: $a \in \text{Field } r$
shows bij-betw $f$ $(\text{underS } r \ a) \ (\text{underS } r' \ (f \ a))$
proof –
have bij-betw $f$ $(\text{under } r \ a) \ (\text{under } r' \ (f \ a))$
using assms by (auto simp add: embed-def)
moreover
{have $f \ a \in \text{Field } r'$
using assms embed-Field[of $r \ r' \ f$] by auto
hence under $r \ a = \text{underS } r \ a \cup \{a\} \land$
der $r' \ (f \ a) = \text{underS } r' \ (f \ a) \cup \{f \ a\}$
using assms by (auto simp add: order-on-defs Refl-under-underS)
} moreover
{have $a \notin \text{underS } r \ a \land f \ a \notin \text{underS } r' \ (f \ a)$
unfolding underS-def by blast
}
ultimately show ?thesis
by (auto simp add: notIn-Un-bij-betw3)
qed

lemma embed-iff-compat-inj-on-ofilter:
assumes WELL: Well-order $r$ and WELL': Well-order $r'$
shows embed $r \ r' \ f = (\text{compat } r \ r' \ f \land \text{inj-on } f \ (\text{Field } r) \land \text{wo-rel.ofilter } r' \ (f' \ (\text{Field } r)))$
using assms
proof(auto simp add: embed-compat embed-inj-on embed-Field-ofilter, unfold embed-def, auto)
fix $a$
assume *: inj-on $f \ (\text{Field } r)$ and
**: compat $r \ r' \ f$ and
***: wo-rel.ofilter $r' \ (f' \ (\text{Field } r))$ and
****: $a \in \text{Field } r$

have $\text{Well} \ r$ using WELL wo-rel-def[of $r$] by simp
hence $\text{Refl} \ r$
using wo-rel.REFL[of $r$] by simp
have Total: Total r
using Well wo-rel.TOTAL[of r] by simp
have Well': wo-rel r'
using WELL' wo-rel-def[of r'] by simp
hence Antisym': antisym r'
using wo-rel.ANTISYM[of r'] by simp
have (a,a) ∈ r
using **** Well wo-rel.REFL[of r]
refl-on-def[of - r] by auto
hence (f a, f a) ∈ r'
using ** by(auto simp add: compat-def)
hence 0: f a ∈ Field r'
unfolding Field-def by auto
have f a ∈ f(Field r)
using **** by auto
hence 2: under r' (f a) ≤ f'(Field r)
using Well' *** wo-rel.ofilter-def[of r' f'(Field r)] by fastforce

show bij-betw f (under r a) (under r' (f a))
proof(unfold bij-betw-def, auto)
  show inj-on f (under r a) by (rule subset-inj-on[OF * under-Field])
next
  fix b assume b ∈ under r a
  thus f b ∈ under r' (f a)
  unfolding under-def using **
  by (auto simp add: compat-def)
next
  fix b' assume *****: b' ∈ under r' (f a)
  hence b' ∈ f'(Field r)
  using 2 by auto
  with Field-def[of r] obtain b where
  3: b ∈ Field r and 4: b' = f b by auto
  have (b,a) ∈ r
  proof–
  {assume (a,b) ∈ r
   with ** 4 have (f a, b') ∈ r'
   by (auto simp add: compat-def)
   with ***** Antisym' have f a = b'
   by(auto simp add: under-def antisym-def)
   with 3 ***** 4 * have a = b
   by(auto simp add: inj-on-def)
  }
  moreover
  {assume a = b
   hence (b,a) ∈ r using Refl ***** 3
   by (auto simp add: refl-on-def)
  }
  ultimately
  show ?thesis using Total **** 3 by (fastforce simp add: total-on-def)
qed
with \(j\) show \(b' \in f'(\text{under } r \ a)\)
unfolding under-def by auto
qed
qed
lemma \text{inv-into-ofilter-embed}:
assumes \(\text{WELL}: \text{Well-order } r \ \text{and } \text{OF}: \text{wo-rel.ofilter } r \ A \ \text{and} \ \text{BIJ}: \forall b \in A. \ \text{bij-betw } f \ (\text{under } r \ b) \ (\text{under } r' \ (f \ b)) \ \text{and} \ \text{IMAGE}: f' \ A = \text{Field } r'\)
shows embed \(r' \ (\text{inv-into } A \ f)\)
proof–

have \(\text{Well}: \text{wo-rel } r\)
using \(\text{WELL} \ \text{wo-rel-def}[\text{of } r] \ \text{by simp}\)
have \(\text{Refl}: \text{Refl } r\)
using \(\text{Well } \text{wo-rel}. \text{REFL}[\text{of } r] \ \text{by simp}\)
have \(\text{Total}: \text{Total } r\)
using \(\text{Well } \text{wo-rel}. \text{TOTAL}[\text{of } r] \ \text{by simp}\)

have 1: \(\text{bij-betw } f \ A \ (\text{Field } r')\)
proof\((\text{unfold } \text{bij-betw-def inj-on-def}, \ \text{auto } \text{simp add}: \text{IMAGE})\)

fix \(b1 \ b2\)
assume \(*: b1 \in A \ \text{and } **: b2 \in A \ \text{and} \ ***: f \ b1 = f \ b2\)

have \(11: b1 \in \text{Field } r \ \land \ b2 \in \text{Field } r\)
using \(* \ \text{Well } \text{OF } \ \text{by } (\text{auto } \text{simp add}: \text{wo-rel.ofilter-def})\)
moreover

\{\text{assume } (b1,b2) \in r\}
\text{hence } b1 \in \text{under } r \ b2 \ \land \ b2 \in \text{under } r \ b2
unfolding under-def using \(11 \ \text{Refl}\)
by \((\text{auto } \text{simp add}: \text{refl-on-def})\)
\text{hence } b1 = b2 using \text{BIJ} \ ** ***
by \((\text{simp add}: \text{bij-betw-def inj-on-def})\)
\}
moreover

\{\text{assume } (b2,b1) \in r\}
\text{hence } b1 \in \text{under } r \ b1 \ \land \ b2 \in \text{under } r \ b1
unfolding under-def using \(11 \ \text{Refl}\)
by \((\text{auto } \text{simp add}: \text{refl-on-def})\)
\text{hence } b1 = b2 using \text{BIJ} \ ** ***
by \((\text{simp add}: \text{bij-betw-def inj-on-def})\)
\}
ultimately

show \(b1 = b2\)
using Total by \((\text{auto } \text{simp add}: \text{total-on-def})\)
qed

let \(\mathcal{f'} = (\text{inv-into } A \ f)\)
have 2: \( \forall b \in A. \bijbetw ?f \ (\under r \ (f b)) \ (\under r b) \)

proof (clarify)

fix \( b \) assume \( *: b \in A \)

hence \( \under r b \leq A \)

using Well OF by (auto simp add: wo.ofilter-def)

moreover

have \( f' \ (\under r b) = \under r' \ (f b) \)

using \( \ast \ BIJ \) by (auto simp add: bij-def)

ultimately

show \( \bijbetw ?f \ (\under r \ (f b)) \ (\under r b) \)

using 1 by (auto simp add: bij-inv-into-subset)

qed

have 3: \( \forall b' \in \Field r'. \bijbetw ?f' \ (\under r' \ (b')) \ (\under r (\?f' \ b')) \)

proof (clarify)

fix \( b' \) assume \( *: b' \in \Field r' \)

have \( b' = f \ (?f' \ b') \) using \( * 1 \)

by (auto simp add: bij-inv-into-right)

moreover

{ obtain \( b \) where \( 31: b \in A \ and \ f b = b' \) using IMAGE \* by force

hence \( \?f' \ b' = b \) using 1 by (auto simp add: bij-inv-into-left)

with \( 31 \) have \( \?f' \ b' \in A \) by auto
}

ultimately

show \( \bijbetw ?f' \ (\under r' \ b') \ (\under r (\?f' \ b')) \)

using 2 by auto

qed

thus \( \ast \)thesis unfolding embed-def.

qed

lemma inv-into-underS-embed:

assumes WELL: Well-order \( r \) and

BIJ: \( \forall b \in \underS r a. \bijbetw f \ (\under r b) \ (\under r' \ (f b)) \) and

IN: \( a \in \Field r \) and

IMAGE: \( f' \ (\underS r a) = \Field r' \)

shows \( \embed r' r \ (\inv-into \ (\underS r a) f) \)

using assms

by (auto simp add: wo-rel-def wo-rel.underS-ofilter inv-into-ofilter-embed)

lemma inv-into-Field-embed:

assumes WELL: Well-order \( r \) and EMB: \( \embed r r' f \) and

IMAGE: \( \Field r' \leq f' \ (\Field r) \)

shows \( \embed r' r \ (\inv-into \ (\Field r) f) \)

proof

have \( (\forall b \in \Field r. \bijbetw f \ (\under r b) \ (\under r' \ (f b))) \)

using EMB by (auto simp add: embed-def)

moreover
have \( f \cdot (\text{Field } r) \leq \text{Field } r' \)
using EMB WELL by (auto simp add: embed-Field) 
ultimately
show \(?\text{thesis using asms}\)
by (auto simp add: wo-rel-def wo-rel.Field-ofilter inv-into-ofilter-embed)
qed

lemma inv-into-Field-embed-bij-betw:
assumes WELL: \(\text{Well-order } r \) and
EMB: \(\text{embed } r \rightarrow r' f \) and
BIJ: \(\text{bij-betw } f \) (\(\text{Field } r\) ) (\(\text{Field } r'\) )
shows \( \text{embed } r' r \) (inv-into (\(\text{Field } r\) ) \(f\))
proof
− have \(\text{Field } r' \leq f \cdot (\text{Field } r)\)
using BIJ by (auto simp add: bij-betw-def)
thus \(?\text{thesis using asms}\)
by (auto simp add: inv-into-Field-embed)
qed

26.3 Given any two well-orders, one can be embedded in the other

Here is an overview of the proof of this fact, stated in theorem wellorders-totally-ordered: Fix the well-orders \(r::'a\rel\) and \(r'::'a'\rel\). Attempt to define an embedding \(f::'a \Rightarrow 'a'\rel\) from \(r\) to \(r'\) in the natural way by well-order recursion ("hoping" that \(\text{Field } r\) turns out to be smaller than \(\text{Field } r'\)), but also record, at the recursive step, in a function \(g::'a \Rightarrow \text{bool}\), the extra information of whether \(\text{Field } r'\) gets exhausted or not.
If \(\text{Field } r'\) does not get exhausted, then \(\text{Field } r\) is indeed smaller and \(f\) is the desired embedding from \(r\) to \(r'\) (lemma wellorders-totally-ordered-aux). Otherwise, it means that \(\text{Field } r'\) is the smaller one, and the inverse of (the "good" segment of) \(f\) is the desired embedding from \(r'\) to \(r\) (lemma wellorders-totally-ordered-aux2).

lemma wellorders-totally-ordered-aux:
fixed \(r::'a\rel\) and \(r'::'a'\rel\) and
\(f::'a \Rightarrow 'a'\rel\) and \(a::'a\)
assumes WELL: \(\text{Well-order } r \) and WELL': \(\text{Well-order } r' \) and IN: \(a \in \text{Field } r\) and
\(\text{IH: } \forall b \in \text{underS } r a. \text{bij-betw } f \) (\(\text{under } r b\) ) (\(\text{under } r' (f b)\) ) and
\(\text{NOT: } f \cdot (\text{underS } r a) \neq \text{Field } r' \) and SUC: \(f a = \text{wo-rel.suc } r' (f(\text{underS } r a))\)
shows \(\text{bij-betw } f \) (\(\text{under } r a\) ) (\(\text{under } r' (f a)\))
proof
− have Well: \(\text{wo-rel } r\) using WELL unfolding \(\text{wo-rel-def}\) .
  hence \(\text{Refl: } \text{Refl } r\) using \(\text{wo-rel.REFL [of } r]\) by auto
  have Trans: \(\text{trans } r\) using \(\text{Well } \text{wo-rel.TRANS [of } r]\) by auto
  have Well': \(\text{wo-rel } r'\) using WELL' unfolding \(\text{wo-rel-def}\) .
have OF: wo-rel.ofilter r (underS r a)
by (auto simp add: Well wo-rel.underS-ofilter)
hence UN: underS r a = (\( \bigcup b \in \text{under}S r a \). under r b)
using Well wo-rel.ofilter-under-UNION[of r underS r a] by blast

\{ fix b assume \*: b \in underS r a \\
  hence t0: (b,a) \in r \land b \neq a unfolding underS-def by auto \\
  have t1: b \in Field r \\
  using \* underS-Field[of r a] by auto \\
  have t2: \text{f}'(under r b) = under r' (f b) \\
  using IH \* by (auto simp add: bij-betw-def) \\
  hence t3: wo-rel.ofilter r' (\text{f}'(under r b)) \\
  using Well' by (auto simp add: wo-rel.under-ofilter) \\
  have f(under r b) \leq Field r' \\
  using t2 by (auto simp add: under-Field) \\
  moreover \\
  have b \in under r b \\
  using t1 by(auto simp add: Refl Refl-under-in) \\
  ultimately \\
  have t4: f b \in Field r' by auto \\
  have f(under r b) = under r' (f b) \\
  wo-rel.ofilter r' (\text{f}'(under r b)) \land \\
  f b \in Field r' \\
  using t2 t3 t4 by auto \\
\} \\
  hence bFact: \\
  \forall b \in underS r a. \text{f}'(under r b) = under r' (f b) \\
  wo-rel.ofilter r' (\text{f}'(under r b)) \land \\
  f b \in Field r' by blast \\

have subField: f(underS r a) \leq Field r' \\
using bFact by blast \\

have OF': wo-rel.ofilter r' (f(underS r a)) \\
proof \\
  have f(underS r a) = f(\bigcup b \in underS r a. under r b) \\
  using UN by auto \\
  also have \ldots = (\bigcup b \in underS r a. f(under r b)) by blast \\
  also have \ldots = (\bigcup b \in underS r a. (under r' (f b))) \\
  using bFact by auto \\
finally \\
  have f(underS r a) = (\bigcup b \in underS r a. (under r' (f b))) . \\
thus thesis \\
using Well' bFact \\
wo-rel.ofilter-UNION[of r' underS r a \land b. under r' (f b)] by fastforce \\
qed \\

have f(underS r a) \cup AboveS r' (f'(underS r a)) = Field r' \\
using Well' OF' by (auto simp add: wo-rel.ofilter-AboveS-Field)
hence \( NE: Above \under{s} \ r' (f(\under{s} \ r \ a)) \neq \{\} \)
using \( subField \ NOT \ by \ blast \)

have \( INCL1: \ f'(\under{s} \ r \ a) \leq \under{s} \ r' (f \ a) \)
proof(auto)
  fix b assume \( \star: b \in \under{s} \ r \ a \)
  have \( f b \neq f a \land (f b, f a) \in r' \)
  using \( subField \ Well' SUC NE \star \)
    wo-rel.suc-greater[of r' f(\under{s} \ r \ a) f b] by force
  thus \( f b \in \under{s} \ r' (f \ a) \)
  unfolding \( underS-def \ by \ simp \)
qed

have \( INCL2: \under{s} \ r' (f \ a) \leq f(\under{s} \ r \ a) \)
proof
  fix b' assume \( b' \in \under{s} \ r' (f \ a) \)
  hence \( b' \neq f a \land (b', f a) \in r' \)
  unfolding \( underS-def \ by \ simp \)
  thus \( b' \in f(\under{s} \ r \ a) \)
  using \( Well' SUC NE OF' \)
    wo-rel.suc-ofilter-in[of r' f \cdot \under{s} \ r \ a \ b'] by auto
qed

have \( INJ: \ inj-on f (\under{s} \ r \ a) \)
proof
  have \( \forall b \in \under{s} \ r \ a. \ inj-on f (\under r b) \)
    using \( IH \) by (auto simp add: bij-betw-def)
  moreover
  have \( \forall b. \ wo-rel.ofilter r (\under r b) \)
    using \( Well \) by (auto simp add: wo-rel.under-ofilter)
  ultimately show \( \ ?thesis \)
    using \( WELL bFact UN \)
      UNION-inj-on-ofilter[of r underS r a \lambda b. \under r b f]
    by auto
qed

have \( BIJ: \ bij-betw f (\under{s} \ r \ a) (\under{s} \ r' (f \ a)) \)
unfolding \( bij-betw-def \)
using \( INJ \ INCL1 \ INCL2 \ by \ auto \)

have \( f a \in \Field \ r' \)
using \( Well' subField NE SUC \)
by (auto simp add: wo-rel.suc-inField)
thus \( \ ?thesis \)
using \( WELL \ WELL' IN \ BIJ \under{-underS-bij-betw[of r r' a f]} \ by \ auto \)
qed

lemma \( wellorders-totally-ordered-aux2: \)
fixes \( r ::'a \ rel \ \text{and} \ r'::'a' \ rel \ and \)
f :: 'a ⇒ 'a' and g :: 'a ⇒ bool and a::'a

assumes WELL: Well-order r and WELL': Well-order r' and

MAIN1: \( \forall a. (\text{False} \notin g'(\text{underS } r a) \land f'(\text{underS } r a) \neq \text{Field } r') \rightarrow f a = \text{wo-rel.suc } r'(f'(\text{underS } r a)) \land g a = \text{True} \)

\( \land (\neg (\text{False} \notin (g'(\text{underS } r a)) \land f'(\text{underS } r a) \neq \text{Field } r') \rightarrow g a = \text{False}) \) and

MAIN2: \( \forall a. a \in \text{Field } r \land \neg \text{False} /\in g'(\text{underS } r a) \rightarrow \text{bij-betw } f (\text{under } r a) (\text{under } r'(f a)) \) and

Case: \( a \in \text{Field } r \land \neg \text{False} /\in g'(\text{under } r a) \)

shows \( \exists f'. \text{ embed } r' \rightarrow f' \)

proof–

have Well: wo-rel r using WELL unfolding wo-rel-def .

hence Refl: Refl r using wo-rel.REFL[of r] by auto

have Trans: trans r using Well wo-rel.TRANS[of r] by auto

have Antisym: antisym r using Well wo-rel.ANTISYM[of r] by auto

have Well': wo-rel r' using WELL' unfolding wo-rel-def .

have 0: \( \text{under } r a = \text{underS } r a \cup \{a\} \)

using Refl Case by(auto simp add: Refl-under-underS)

have 1: \( g a = \text{False} \)

proof–

\{assume g a ≠ False

with 0 Case have \( \text{False} /\in g'(\text{underS } r a) \) by blast

with MAIN1 have g a = False by blast\}

thus \( \exists \text{thesis} \) by blast

qed

let \( ?A = \{a \in \text{Field } r. g a = \text{False}\} \)

let \( ?a = (\text{wo-rel.minim } r ?A) \)

have 2: \( \exists A \neq \{\} \land \exists A \subseteq \text{Field } r \) using Case 1 by blast

have 3: \( \text{False} \notin g'(\text{underS } r \ ?a) \)

proof

assume \( \text{False} \notin g'(\text{underS } r \ ?a) \)

then obtain b where \( b \in \text{underS } r \ ?a \) and 31: \( g b = \text{False} \) by auto

hence 32: \( (b, ?a) \in r \land b \neq \?a \)

by (auto simp add: underS-def)

hence \( b \in \text{Field } r \) unfolding Field-def by auto

with 31 have \( b \in ?A \) by auto

hence \( (?a, b) \in r \) using wo-rel.minim-least 2 Well by fastforce

with 32 Antisym show False

by (auto simp add: antisym-def)

qed

have temp: \( ?a \in \?A \)

using \( \exists ?a \text{ wo-rel.minim-in}[of } r \ ?A \) by auto
hence 4: ?a ∈ Field r by auto

have 5: g ?a = False using temp by blast

have 6: f'(underS r ?a) = Field r' using MAIN1[of ?a] 3 5 by blast

have 7: ∀ b ∈ underS r ?a. bij-betw f (under r b) (under r' (f b))
proof
  fix b assume as: b ∈ underS r ?a
  moreover
  have wo-rel.ofilter r (underS r ?a)
  using Well by (auto simp add: wo-rel.underS-ofilter)
  ultimately
  have False ∉ g'(under r b) using 3 Well by (subst (asm) wo-rel.ofilter-def)
  fast+
  moreover have b ∈ Field r unfolding Field-def using as by (auto simp add: underS-def)
  ultimately
  show bij-betw f (under r b) (under r' (f b))
  using MAIN2 by auto
qed

have embed r' r (inv-into (underS r ?a) f)
using WELL WELL' 7 4 6 inv-into-underS-embed[of r ?a f r'] by auto
thus ?thesis
  unfolding embed-def by blast
qed

theorem wellorders-totally-ordered:
fixes r ::'a rel and r'::'a' rel
assumes WELL: Well-order r and WELL': Well-order r'
shows (∃ f. embed r r' f) ∨ (∃ f'. embed r' r f')
proof –

  have Well: wo-rel r using WELL unfolding wo-rel-def.
  have Refl: Refl r using wo-rel.REFL[of r] by auto
  have Trans: trans r using Well wo-rel.TRANS[of r] by auto
  have Well': wo-rel r' using WELL' unfolding wo-rel-def.

obtain H where H-def: H =
(λh a. if False ∉ (snd o h) ' (underS r a) ∧ (fst o h) ' (underS r a) ≠ Field r'
  then (wo-rel.suc r' ((fst o h) ' (underS r a)), True)
  else (undefined, False)) by blast

have Adm: wo-rel.adm-wo r H
using Well
proof (unfold wo-rel.adm-wo-def, clarify)
  fix h1::'a ⇒ 'a' * bool and h2::'a ⇒ 'a' * bool and x
  assume ∀ y ∈ underS r x. h1 y = h2 y
hence \( \forall y \in \text{underS} r x. (\text{fst} \circ h1) y = (\text{fst} \circ h2) y \land (\text{snd} \circ h1) y = (\text{snd} \circ h2) y \) \text{ by auto}

hence \( (\text{fst} \circ h1)'(\text{underS} r x) = (\text{fst} \circ h2)'(\text{underS} r x) \land (\text{snd} \circ h1)'(\text{underS} r x) = (\text{snd} \circ h2)'(\text{underS} r x) \)

\text{by (auto simp add: image-def)}

thus \( H h1 x = H h2 x \) \text{ by (simp add: H-def del: not-False-in-image-Ball)}

\text{qed}

obtain \( h::'a \Rightarrow 'a' \) \text{ bool and f::'a} \Rightarrow 'a' \text{ and g::'}a \Rightarrow \) \text{ bool}

\text{where h-def:} \ h = \text{wo-rel.worec} r H \text{ and}

\( f\)-def: \( f = \text{fst} \circ h \) \text{ and g-def:} \( g = \text{snd} \circ h \) \text{ by blast}

obtain \( \text{test where test-def:} \)

\( \text{test} = (\lambda a. \text{False} \notin (g'(\text{underS} r a)) \land f(\text{underS} r a) \neq \text{Field r'}) \) \text{ by blast}

have \( \star\cdot \) \( \text{a, h a} = H h a \)

\text{using Adm Well wo-rel.worec-fixpoint[of r H] by (simp add: h-def)}

have \( \text{Main1:} \)

\( \forall a. (\text{test a} \longrightarrow f a = \text{wo-rel.suc} r' (f'(\text{underS} r a)) \land g a = \text{True}) \land (\neg(\text{test a}) \longrightarrow g a = \text{False}) \)

\text{proof--}

\text{fix a show (test a} \longrightarrow f a = \text{wo-rel.suc} r' (f'(\text{underS} r a)) \land g a = \text{True}) \land (\neg(\text{test a}) \longrightarrow g a = \text{False})

\text{using \( \star\cdot \) test-def f-def g-def H-def by auto}

\text{qed}

let \( \lambda\) \( phi \cdot \lambda a. a \in \text{Field r} \land \text{False} \notin g'(\text{under r a}) \longrightarrow \) \text{bij-betw} f (\text{under r a}) (\text{under r'} (f a))

have \( \text{Main2:} \forall a. \lambda\) \( phi\) \( a\)

\text{proof--}

\text{fix a show \( \lambda\) \( phi\) \( a\) }

\text{proof(rule wo-rel.well-order-induct[of r \lambda\) \( phi\}], simp only: Well, clarify)}

\text{fix a}

\text{assume \( \text{IH:} \forall b. b \neq a \land (b,a) \in r \longrightarrow \lambda\) \( phi\) \( b\) and}

\( \star\cdot \) \( a \in \text{Field r and} \)

\( **\cdot \text{False} \notin g'(\text{under r a}) \)

\text{have \( \text{I:} \forall b \in \text{underS} r a. \text{bij-betw} f (\text{under r b}) (\text{under r'} (f b)) \)

\text{proof(clarify)}

\text{fix b assume \( **\cdot \) b \( \in \text{underS} r a \)

hence \( \theta: (b,a) \in r \land b \neq a \) \text{ unfolding underS-def by auto}

moreover have \( b \in \text{Field r} \)

using \( **\cdot \text{underS-Field[of r a]} \) \text{ by auto}

moreover have \( \text{False} \notin g'(\text{under r b}) \)

using \( \theta ** \cdot \text{Trans under-incr[of r b a]} \) \text{ by auto}

ultimately show \( \text{bij-betw} f (\text{under r b}) (\text{under r'} (f b)) \)

using \( \text{IH by auto} \)

\text{qed}

\text{have 21: False} \notin g'(\text{underS} r a)
using ** underS-subset-under[of r a] by auto
have 22: g'(under r a) ≤ {True} using ** by auto
moreover have 23: a ∈ under r a
using Reflex* by (auto simp add: Reflex-under-in)
ultimately have 24: g a = True by blast
have 2: f'(underS r a) ≠ Field r'
proof
  assume f'(underS r a) = Field r'
  hence g a = False using Main1 test-def by blast
with 24 show False using ** by blast
qed

have 3: f a = wo-rel.suc r' (f'(underS r a))
using 21 2 Main1 test-def by blast

show bij-betw f (under r a) (under r'(f a))
using WELL WELL' 1 2 3 *
wellorders-totally-ordered-aux[of r r' a f] by auto
qed

let ?chi = (λ a. a ∈ Field r ∧ False ∈ g'(under r a))
show ⊤thesis
proof(cases ∃a. ?chi a)
  assume ¬ (∃a. ?chi a)
  hence ∀ a ∈ Field r. bij-betw f (under r a) (under r'(f a))
  using Main2 by blast
  thus ⊤thesis unfolding embed-def by blast
next
  assume ∃a. ?chi a
  then obtain a where ?chi a by blast
  hence ∃f'. embed r' r f'
  using wellorders-totally-ordered-aux2[of r r' f a]
  WELL WELL' Main1 Main2 test-def by fast
  thus ⊤thesis by blast
qed
qed

26.4 Uniqueness of embeddings

Here we show a fact complementary to the one from the previous subsection – namely, that between any two well-orders there is at most one embedding, and is the one definable by the expected well-order recursive equation. As a consequence, any two embeddings of opposite directions are mutually inverse.

lemma embed-determined:
assumes WELL: Well-order r and WELL': Well-order r' and
EMB: embed r r' f and IN: a ∈ Field r
shows \( f a = \text{wo-rel.suc} \ r' (f'(\text{underS} \ r \ a)) \)

proof –
- have \( \text{bij-betw} \ f (\text{underS} \ r \ a) (\text{underS} \ r' (f \ a)) \)
  using assms by (auto simp add: embed-underS)
- hence \( f'(\text{underS} \ r \ a) = \text{underS} \ r' (f \ a) \)
  by (auto simp add: bij-betw-def)
moreover
\{ have \( f \ a \in \text{Field} \ r' \) using IN
  using EMB WELL embed-Field[of \ r' \ f] by auto
  hence \( f \ a = \text{wo-rel.suc} \ r' (\text{underS} \ r' (f \ a)) \)
  using WELL by (auto simp add: wo-rel-def wo-rel.suc-underS)
\}
ultimately show \( \text{thesis} \) by simp
qed

lemma embed-unique:
assumes WELL: \( \text{Well-order} \ r \) and WELL': \( \text{Well-order} \ r' \) and
EMBf: \( \text{embed} \ r \ r' \ f \) and EMBg: \( \text{embed} \ r \ r' \ g \)
shows \( \forall a \in \text{Field} \ r \rightarrow f \ a = g \ a \)
proof (rule wo-rel.well-order-induct[of \ r], auto simp add: WELL wo-rel-def)
  fix \( a \)
  assume IH: \( \forall b. \ b \neq a \land (b,a) \in r \rightarrow b \in \text{Field} \ r \rightarrow f \ b = g \ b \) and
  \( \ast: \ a \in \text{Field} \ r \)
  hence \( \forall b \in \text{underS} r \ a. \ f \ b = g \ b \)
  unfolding underS-def by (auto simp add: Field-def)
  hence \( f'(\text{underS} \ r \ a) = g'(\text{underS} \ r \ a) \) by force
  thus \( f \ a = g \ a \)
  using assms \( \ast \) embed-determined[of \ r \ r' \ f \ a] embed-determined[of \ r \ r' \ g \ a] by auto
qed

lemma embed-bothWays-inverse:
assumes WELL: \( \text{Well-order} \ r \) and WELL': \( \text{Well-order} \ r' \) and
EMB: \( \text{embed} \ r \ r' \ f \) and EMB': \( \text{embed} \ r' \ r \ f' \)
shows \( \forall a \in \text{Field} \ r. \ f'(f \ a) = a \land (\forall a' \in \text{Field} \ r'. \ f(f' \ a') = a') \)
proof –
- have \( \text{embed} \ r \ r (f' \circ f) \) using assms
  by (auto simp add: comp-embed)
- moreover have \( \text{embed} \ r \ \text{id} \) using assms
  by (auto simp add: id-embed)
- ultimately have \( \forall a \in \text{Field} \ r. \ f'(f \ a) = a \)
  using assms embed-unique[of \ r \ r' \ f \ a \ f \ id] id-def by auto
moreover
\{ have \( \text{embed} \ r' \ r' (f \circ f') \) using assms
  by (auto simp add: comp-embed)
- moreover have \( \text{embed} \ r' \ \text{id} \) using assms
  by (auto simp add: id-embed)
- ultimately have \( \forall a' \in \text{Field} \ r'. \ f(f' \ a') = a' \)
  using assms embed-unique[of \ r' \ r' \ f \ f' \ id] id-def by auto
ultimately show \( ? \text{thesis} \) by blast

\( \text{lemma embed-bothWays-bij-betw:} \)

\( \text{assumes WELL: Well-order } r \text{ and WELL': Well-order } r' \text{ and} \)

\( \text{EMB: embed } r r' f \text{ and EMB': embed } r' r g \)

\( \text{shows bij-betw } f (\text{Field } r) (\text{Field } r') \)

\( \text{proof} - \)

\( \text{let } ?A = \text{Field } r \text{ let } ?A' = \text{Field } r' \)

\( \text{have embed } r r (g \circ f) \land \text{embed } r' r' (f \circ g) \)

\( \text{using assms by (auto simp add: comp-embed)} \)

\( \text{hence } I: (\forall a \in ?A. g(f a) = a) \land (\forall a' \in ?A'. f(g a') = a') \)

\( \text{using WELL id-embed[of } r \text{] embed-unique[of } r \text{ } r \text{ } g \text{ id]} \)

\( \text{WELL' id-embed[of } r' \text{ ] embed-unique[of } r' \text{ } r' \text{ } f \text{ } \circ \text{ } g \text{ id]} \)

\( \text{id-def by auto} \)

\( \text{have 2: } (\forall a \in ?A. f a \in ?A') \land (\forall a' \in ?A'. g a' \in ?A) \)

\( \text{using assms embed-Field[of } r \text{ } r' \text{ ] embed-Field[of } r' \text{ } r \text{ } g \text{ by blast} \)

\( \text{show } ? \text{thesis} \)

\( \text{proof(unfold bij-betw-def inj-on-def, auto simp add: 2)} \)

\( \text{fix a b assume *: } a \in ?A \text{ } b \in ?A \text{ and **: } f a = f b \)

\( \text{have a = g(f a) \land b = g(f b) using * 1 by auto} \)

\( \text{with ** show } a = b \text{ by auto} \)

\( \text{next} \)

\( \text{fix a' assume *: } a' \in ?A' \)

\( \text{hence g a' \in ?A \land f(g a') = a' using 1 2 by auto} \)

\( \text{thus } a' \in f' ?A \text{ by force} \)

\( \text{qed} \)

\( \text{qed} \)

\( \text{lemma embed-bothWays-iso:} \)

\( \text{assumes WELL: Well-order } r \text{ and WELL': Well-order } r' \text{ and} \)

\( \text{EMB: embed } r r' f \text{ and EMB': embed } r' r f' \)

\( \text{shows iso } r r' f \)

\( \text{unfolding iso-def using assms by (auto simp add: embed-bothWays-bij-betw)} \)

\( \text{26.5 More properties of embeddings, strict embeddings and isomorphisms} \)

\( \text{lemma embed-bothWays-Field-bij-betw:} \)

\( \text{assumes WELL: Well-order } r \text{ and WELL': Well-order } r' \text{ and} \)

\( \text{EMB: embed } r r' f \text{ and EMB': embed } r' r f' \)

\( \text{shows bij-betw } f (\text{Field } r) (\text{Field } r') \)

\( \text{proof} - \)

\( \text{have } (\forall a \in \text{Field } r. f'(f a) = a) \land (\forall a' \in \text{Field } r'. f(f' a') = a') \)

\( \text{using assms by (auto simp add: embed-bothWays-inverse)} \)

\( \text{moreover} \)

\( \text{have } f'(\text{Field } r) \leq \text{Field } r' \land f' : (\text{Field } r') \leq \text{Field } r \)
using assms by (auto simp add: embed-Field)
ultimately
show \(?\text{thesis}\) using bij-betw-byWitness[of Field \( f \) \( f' \) Field \( r' \)] by auto
qed

lemma embedS-comp-embed:
assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \) and WELL'': Well-order \( r'' \)
and EMB: embedS \( r \) \( r' \) \( f \) and EMB': embed \( r' \) \( r'' \) \( f' \)
shows embedS \( r \) \( r'' \) \(( f' \circ f)\)
proof
let \(?g = (f' \circ f)\) let \(?h = \text{inv-into} (\text{Field} \( r \)) \( ?g\)
have 1: embed \( r \) \( r' \) \( f \) \( \land \neg (\text{bij-betw} \( f \) \( (\text{Field} \( r \)) \( (\text{Field} \( r' \))\))
using EMB by (auto simp add: embedS-def)

hence 2: embed \( r \) \( r'' \) \(?g\)
using WELL EMB' comp-embed[of \( r \) \( r' \) \( f \) \( r'' \) \( f' \)] by auto
moreover
{assume bij-betw \(?g (\text{Field} \( r \)) (\text{Field} \( r'' \))

hence embed \( r'' \) \( r \) \(?h\) using 2 WELL
by (auto simp add: inv-into-Field-embed-bij-betw)

hence embed \( r' \) \( r'' \) \((?h \circ f')\) using WELL' EMB'
by (auto simp add: comp-embed)

hence bij-betw \( f \) \( (\text{Field} \( r \)) \( (\text{Field} \( r' \)) \) using WELL WELL' 1
by (auto simp add: embed-bothWays-Field-bij-betw)
with 1 have False by blast
}
ultimately show \(?\text{thesis}\) unfolding embedS-def by auto
qed

lemma embed-comp-embedS:
assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \) and WELL'': Well-order \( r'' \)
and EMB: embed \( r \) \( r' \) \( f \) and EMB': embedS \( r' \) \( r'' \) \( f' \)
shows embedS \( r \) \( r'' \) \(( f' \circ f)\)
proof
let \(?g = (f' \circ f)\) let \(?h = \text{inv-into} (\text{Field} \( r \)) \( ?g\)
have 1: embed \( r' \) \( r'' \) \( f' \) \( \land \neg (\text{bij-betw} \( f' \) \( (\text{Field} \( r' \)) \( (\text{Field} \( r'' \))\))
using EMB' by (auto simp add: embedS-def)

hence 2: embed \( r \) \( r'' \) \(?g\)
using WELL EMB comp-embed[of \( r \) \( r' \) \( f \) \( r'' \) \( f' \)] by auto
moreover
{assume bij-betw \(?g (\text{Field} \( r \)) (\text{Field} \( r'' \))

hence embed \( r'' \) \( r \) \(?h\) using 2 WELL
by (auto simp add: inv-into-Field-embed-bij-betw)

hence embed \( r'' \) \( r' \) \((f \circ ?h)\) using WELL'' EMB
by (auto simp add: comp-embed)

hence bij-betw \( f' \) \( (\text{Field} \( r' \)) \( (\text{Field} \( r'' \)) \) using WELL' WELL'' 1
by (auto simp add: embed-bothWays-Field-bij-betw)
with 1 have False by blast
}
ultimately show \textit{thesis} unfolding embedS-def by auto

\textbf{lemma} \textit{embed-comp-iso}:
\textbf{assumes} \textit{WELL}: Well-order \textit{r} and \textit{WELL'}: Well-order \textit{r'} and \textit{WELL''}: Well-order \textit{r''}
\quad and \textit{EMB}: embed \textit{r} \textit{r'} \textit{f} and \textit{EMB'}: iso \textit{r'} \textit{r''} \textit{f}'
\textit{shows} embed \textit{r} \textit{r''} (\textit{f}' \circ \textit{f})
\textit{using} assms unfolding iso-def
\textit{by} (auto simp add: comp-embed)

\textbf{lemma} \textit{iso-comp-embed}:
\textbf{assumes} \textit{WELL}: Well-order \textit{r} and \textit{WELL'}: Well-order \textit{r'} and \textit{WELL''}: Well-order \textit{r''}
\quad and \textit{EMB}: iso \textit{r} \textit{r'} \textit{f} and \textit{EMB'}: embed \textit{r'} \textit{r''} \textit{f}'
\textit{shows} embed \textit{r} \textit{r''} (\textit{f}' \circ \textit{f})
\textit{using} assms unfolding iso-def
\textit{by} (auto simp add: comp-embed)

\textbf{lemma} \textit{embedS-comp-iso}:
\textbf{assumes} \textit{WELL}: Well-order \textit{r} and \textit{WELL'}: Well-order \textit{r'} and \textit{WELL''}: Well-order \textit{r''}
\quad and \textit{EMB}: embedS \textit{r} \textit{r'} \textit{f} and \textit{EMB'}: embedS \textit{r'} \textit{r''} \textit{f}'
\textit{shows} embedS \textit{r} \textit{r''} (\textit{f}' \circ \textit{f})
\textit{using} assms unfolding iso-def
\textit{by} (auto simp add: embedS-comp-embed)

\textbf{lemma} \textit{iso-comp-embedS}:
\textbf{assumes} \textit{WELL}: Well-order \textit{r} and \textit{WELL'}: Well-order \textit{r'} and \textit{WELL''}: Well-order \textit{r''}
\quad and \textit{EMB}: iso \textit{r} \textit{r'} \textit{f} and \textit{EMB'}: embedS \textit{r'} \textit{r''} \textit{f}'
\textit{shows} embedS \textit{r} \textit{r''} (\textit{f}' \circ \textit{f})
\textit{using} assms unfolding iso-def
\textit{by} (auto simp add: embedS-comp-embedS)

\textbf{lemma} \textit{embedS-Field}:
\textbf{assumes} \textit{WELL}: Well-order \textit{r} and \textit{EMB}: embedS \textit{r} \textit{r'} \textit{f}
\textit{shows} \textit{f}' (\textit{Field} \textit{r}) < \textit{Field} \textit{r'}
\textit{proof} –
\quad have \textit{f}' (\textit{Field} \textit{r}) \leq \textit{Field} \textit{r'} using assms
\quad by (auto simp add: embed-Field embedS-def)
\quad moreover
\quad \{ have \textit{inj-on} \textit{f} (\textit{Field} \textit{r}) using assms
\quad \quad by (auto simp add: embedS-def embed-inj-on)
\quad \quad hence \textit{f}' (\textit{Field} \textit{r}) \neq \textit{Field} \textit{r'} using \textit{EMB}
\quad \quad by (auto simp add: embedS-def bij-beta-def)
\quad \}
\quad ultimately show \textit{thesis} by blast
lemma embedS-iff:
assumes WELL: Well-order r and ISO: embed r r' f
shows embedS r r' f = (f' (Field r) < Field r')
proof
  assume embedS r r' f
  thus f' Field r ⊂ Field r'
  using WELL by (auto simp add: embedS-Field)
next
  assume f' Field r ⊂ Field r'
  hence ¬ bij-betw f (Field r) (Field r')
  unfolding bij-betw-def by blast
  thus embedS r r' f unfolding embedS-def
  using ISO by auto
qed

lemma iso-Field:
  iso r r' f ⇒ f' (Field r) = Field r'
by (auto simp add: iso-def bij-betw-def)

lemma iso-iff:
assumes Well-order r
shows iso r r' f = (embed r r' f ∧ f' (Field r) = Field r')
proof
  assume iso r r' f
  thus embed r r' f ∧ f' (Field r) = Field r'
  by (auto simp add: iso-Field iso-def)
next
  assume *: embed r r' f ∧ f' Field r = Field r'
  hence inj-on f (Field r) using assms by (auto simp add: embed-inj-on)
  with * have bij-betw f (Field r) (Field r')
  unfolding bij-betw-def by simp
  with * show iso r r' f unfolding iso-def by auto
qed

lemma iso-iff2:
assumes Well-order r
shows iso r r' f = (bij-betw f (Field r) (Field r') ∧
  (∀ a ∈ Field r. ∀ b ∈ Field r.
  (((a, b) ∈ r) = (((f a, f b) ∈ r'))))
using assms
proof(auto simp add: iso-def)
  fix a b
  assume embed r r' f
  hence compat r r' f using embed-compat[of r] by auto
  moreover assume (a,b) ∈ r
  ultimately show (f a, f b) ∈ r' using compat-def[of r] by auto
next
let \( f' = \text{inv-into} (\text{Field } r') f \)
assume embed \( r' \) \( f \) and 1: bij-betw \( f \) (Field \( r \)) (Field \( r' \))
hence embed \( r' \) \( r \) \( f' \) using \( \text{assms} \)
by (auto simp add: inv-into-Field-embed-bij-betw)

next
fix \( a, b \)
assume *: \( a \in \text{Field } r \) \( b \in \text{Field } r \)
and **: \( (f \ a, f \ b) \in r' \)
hence \( \text{compat } r \) \( r' \) \( f \) \( f' \) using 1
by (auto simp add: bij-betw-inv-into-left)
thus \( (a, b) \in r \) using ** 2 compat-def[of \( r' \) \( r' \) \( f \) ] by fastforce

next
fix \( b \)
assume \( b \in \text{under } r \)
hence \( a \in \text{Field } r \) \( b \in \text{Field } r \) \( (b, a) \in r \)
by (auto simp add: under-Field)

next
fix \( b' \)
assume \( b' \in \text{under } r' \) \( f \ a \)
hence \( 3: (b', f \ a) \in r' \)
unfolding under-def by (auto simp add: Field-def Range-def Domain-def)
with 1 ** show \( f \ b \in \text{under } r' (f \ a) \)

next

lemma iso-iff3:
assumes WELL: Well-order \( r \) and WELL': Well-order \( r' \)
shows iso \( r \) \( r' \) \( f = (\text{bij-betw } f \) (Field \( r )) (Field \( r' )) \) \( \text{compat } r \) \( r' \) \( f \)
proof
assume iso \( r \) \( r' \) \( f \)
thus \( \text{bij-betw } f \) (Field \( r )) (Field \( r' )) \( \text{compat } r \) \( r' \) \( f \)
unfolding compat-def using WELL by (auto simp add: iso-iff2 Field-def)

next
have Well: wo-rel \( r \) \( \text{wo-rel } r' \) using WELL WELL'
by (auto simp add: wo-rel-def)
assume *: bij-betw \( f \) (Field \( r )) (Field \( r' )) \( \text{compat } r \) \( r' \) \( f \)
thus iso r r' f
unfolding compat-def using assms
proof (auto simp add: iso-iff2)
  fix a b assume **: a ∈ Field r b ∈ Field r and
  ***: (f a, f b) ∈ r'
  {assume (b,a) ∈ r ⟦ b = a
  hence (b,a) ∈ r using Well ** wo-rel.REFL[of r] refl-on-def[of - r] by blast
  hence (f b, f a) ∈ r' using * unfolding compat-def by auto
  hence a = b using Well *** wo-rel.ANTISYM[of r'] antisym-def[of r'] by blast
  hence (a,b) ∈ r using Well ** wo-rel.REFL[of r] refl-on-def[of - r] by blast
  }thus (a,b) ∈ r
  using Well ** wo-rel.TOTAL[of r] total-on-def[of - r] by blast
qed
qed

27 Constructions on Wellorders as Needed by Bounded Natural Functors

theory BNF-Wellorder-Constructions
imports BNF-Wellorder-Embedding
begin

In this section, we study basic constructions on well-orders, such as restriction to a set/order filter, copy via direct images, ordinal-like sum of disjoint well-orders, and bounded square. We also define between well-orders the relations ordLeq, of being embedded (abbreviated ≤), ordLess, of being strictly embedded (abbreviated <o), and ordIso, of being isomorphic (abbreviated =o). We study the connections between these relations, order filters, and the aforementioned constructions. A main result of this section is that <o is well-founded.

27.1 Restriction to a set

abbreviation Restr :: 'a rel ⇒ 'a set ⇒ 'a rel
where Restr r A ≡ r Int (A × A)

lemma Restr-subset:
  A ≤ B ⇒ Restr (Restr r B) A = Restr r A
by blast

lemma Restr-Field: Restr r (Field r) = r
unfolding Field-def by auto
lemma Refl-Restr: $\text{Refl } r \implies \text{Refl}(\text{Restr } r A)$
unfolding refl-on-def Field-def by auto

lemma linear-order-on-Restr:
  linear-order-on $A \implies \text{linear-order-on } (A \cap \text{above } r x) \ (\text{Restr } r (\text{above } r x))$
by(simp add: order-on-defs refl-on-def trans-def antisym-def total-on-def)(safe; blast)

lemma antisym-Restr:
  antisym $r \implies \text{antisym}(\text{Restr } r A)$
unfolding antisym-def Field-def by auto

lemma Total-Restr:
  Total $r \implies \text{Total}(\text{Restr } r A)$
unfolding total-on-def Field-def by auto

lemma trans-Restr:
  trans $r \implies \text{trans}(\text{Restr } r A)$
unfolding trans-def Field-def by blast

lemma Preorder-Restr:
  Preorder $r \implies \text{Preorder}(\text{Restr } r A)$
unfolding preorder-on-def by (simp add: Refl-Restr trans-Restr)

lemma Partial-order-Restr:
  Partial-order $r \implies \text{Partial-order}(\text{Restr } r A)$
unfolding partial-order-on-def by (simp add: Preorder-Restr antisym-Restr)

lemma Linear-order-Restr:
  Linear-order $r \implies \text{Linear-order}(\text{Restr } r A)$
unfolding linear-order-on-def by (simp add: Partial-order-Restr Total-Restr)

lemma Well-order-Restr:
assumes Well-order $r$
shows Well-order(\text{Restr } r A)
proof -
  have $\text{Restr } r A - \text{Id } \leq r - \text{Id}$ using Restr-subset by blast
  hence $\text{wf}(\text{Restr } r A - \text{Id})$ using assms
  using well-order-on-def wf-subset by blast
  thus ?thesis using assms unfolding well-order-on-def
  by (simp add: Linear-order-Restr)
qed

lemma Field-Restr-subset: Field(\text{Restr } r A) \leq A
by (auto simp add: Field-def)

lemma Refl-Field-Restr:
  Refl $r \implies \text{Field}(\text{Restr } r A) = \text{(Field } r) \ \text{Int } A$
unfolding refl-on-def Field-def by blast

lemma Refl-Field-Restr2:
[\text{Refl } r; A \leq \text{Field } r] \implies \text{Field}(\text{Restr } r A) = A
by (auto simp add: Refl-Field-Restr)

lemma well-order-on-Restr:
assumes WELL: Well-order r and SUB: A \leq \text{Field } r
shows well-order-on A (Restr r A)
using assms
using Well-order-Restr[of A] Refl-Field-Restr2[of A]
order-on-defs[of Field r r] by auto

27.2 Order filters versus restrictions and embeddings

lemma Field-Restr-ofilter:
[\text{Well-order } r; \text{wo-rel}.ofilter r A] \implies \text{Field}(\text{Restr } r A) = A
by (auto simp add: wo-rel-def wo-rel.ofilter-def wo-rel.REFL Refl-Field-Restr2)

lemma ofilter-Restr-under:
assumes WELL: Well-order r and OF: \text{wo-rel}.ofilter r A and IN: a \in A
shows under (Restr r A) a = under r a
using assms wo-rel-def
proof(auto simp add: wo-rel.ofilter-def under-def)
  fix b assume *: a \in A and (b,a) \in r
  hence b \in under r a \land a \in \text{Field } r
  unfolding under-def using Field-def by fastforce
  thus b \in A using * assms by (auto simp add: wo-rel-def wo-rel.ofilter-def)
qed

lemma ofilter-embed:
assumes Well-order r
shows wo-rel.ofilter r A = (A \leq \text{Field } r \land \text{embed } (\text{Restr } r A) r id)
proof
  assume *: \text{wo-rel}.ofilter r A
  show A \leq \text{Field } r \land \text{embed } (\text{Restr } r A) r id
  proof(unfold embed-def, auto)
    fix a assume a \in A thus a \in \text{Field } r using assms *
    by (auto simp add: wo-rel-def wo-rel.ofilter-def)
  next
    fix a assume a \in \text{Field } (\text{Restr } r A)
    thus bij-betw id (under (Restr r A) a) (under r a) using assms *
    by (simp add: ofilter-Restr-under Field-Restr-ofilter)
  qed
next
  assume *: A \leq \text{Field } r \land \text{embed } (\text{Restr } r A) r id
  hence Field(\text{Restr } r A) \leq \text{Field } r
  using assms embed-Field[of Restr r A r id] id-def
  Well-order-Restr[of r] by auto
{fix a assume a ∈ A
  hence a ∈ Field(Restr r A) using * assms
  by (simp add: order-on-defs Refl-Field-Restr2)
  hence bij-btw id (under (Restr r A) a) (under r a)
  using * unfolding embed-def by auto
  hence under r a ≤ under (Restr r A) a
  unfolding bij-btw-def by auto
  also have ... ≤ Field(Restr r A) by (simp add: under-Field)
  also have ... ≤ A by (simp add: Field-Restr-subset)
  finally have under r a ≤ A .
}
thus wo-rel.ofilter r A using assms * by (simp add: wo-rel-def wo-rel.ofilter-def)
qed

lemma ofilter-Restr-Int:
assumes WELL: Well-order r and OFA: wo-rel.ofilter r A
shows wo-rel.ofilter (Restr r B) (A Int B)
proof−
  let ?rB = Restr r B
  have Well: wo-rel r unfolding wo-rel-def using WELL .
  hence Refl: Refl r by (simp add: wo-rel.REFL)
  hence Field: Field ?rB = Field r Int B
  using Refl-Field-Restr by blast
  have WellB: wo-rel ?rB ∧ Well-order ?rB using WELL
    by (simp add: Well-order-Restr wo-rel-def)
  show ?thesis using WellB assms
proof(auto simp add: wo-rel.ofilter-def under-def)
  fix a assume a ∈ A and *: a ∈ B
  hence a ∈ Field r using OFA Well by (auto simp add: wo-rel.ofilter-def)
  with * show a ∈ Field ?rB using Field by auto
next
  fix a b assume a ∈ A and (b,a) ∈ r
  thus b ∈ A using Well OFA by (auto simp add: wo-rel.ofilter-def under-def)
qed
qed

lemma ofilter-Restr-subset:
assumes WELL: Well-order r and OFA: wo-rel.ofilter r A and SUB: A ≤ B
shows wo-rel.ofilter (Restr r B) A
proof−
  have A Int B = A using SUB by blast
  thus ?thesis using assms offilter-Restr-Int[of r A B] by auto
qed

lemma ofilter-subset-embed:
assumes WELL: Well-order r and
  OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
shows (A ≤ B) = (embed (Restr r A) (Restr r B) id)
proof
  let ?rA = Restr r A  let ?rB = Restr r B
  have Well: wo-rel r unfolding wo-rel-def using WELL .
  hence Refl: Refl r by (simp add: wo-rel.REFL)
  hence FieldA: Field ?rA = Field r Int A
      using Refl-Field-Restr by blast
  have FieldB: Field ?rB = Field r Int B
      using Refl Refl-Field-Restr by blast
  have WellA: wo-rel ?rA ∧ Well-order ?rA using WELL
      by (simp add: Well-order-Restr wo-rel-def)
  have WellB: wo-rel ?rB ∧ Well-order ?rB using WELL
      by (simp add: Well-order-Restr wo-rel-def)
  hence ?thesis
proof
  assume ∗: A ≤ B
  hence wo-rel.ofilter (Restr r B) A using assms
      by (simp add: ofilter-Restr-subset)
  hence embed (Restr ?rB A) (Restr r B) id
      using WellB ofilter-embed[of ?rB A] by auto
  thus embed (Restr r A) (Restr r B) id
      using ∗ by (simp add: Restr-subset)
next
  assume ∗: embed (Restr r A) (Restr r B) id
  { fix a assume ∗∗: a ∈ A
    hence a ∈ Field r using Well OFA by (auto simp add: wo-rel.ofilter-def)
    with ∗ FieldA have a ∈ Field ?rA by auto
    hence a ∈ Field ?rB using ∗ WellA embed-Field[of ?rA ?rB id] by auto
    hence a ∈ B using FieldB by auto
  }
  thus A ≤ B by blast
qed

lemma ofilter-subset-embedS-iso:
assumes WELL: Well-order r and
  OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
shows
  ((A < B) = (embedS (Restr r A) (Restr r B) id)) ∧
  ((A = B) = (iso (Restr r A) (Restr r B) id))
proof
  let ?rA = Restr r A  let ?rB = Restr r B
  have Well: wo-rel r unfolding wo-rel-def using WELL .
  hence Refl: Refl r by (simp add: wo-rel.REFL)
  hence Field ?rA = Field r Int A
    using Refl-Field-Restr by blast
  hence FieldA: Field ?rA = A using OFA Well
      by (auto simp add: wo-rel.ofilter-def)
  have Field ?rB = Field r Int B
    using Refl Refl-Field-Restr by blast
hence FieldB: Field rB = B using OFB Well
by (auto simp add: wo-rel.ofilter-def)

show ?thesis unfolding embedS-def iso-def
using assms ofilter-subset-embed[of r A B]
FieldA FieldB bij-betw-id-iff[of A B] by auto
qed

lemma ofilter-subset-embedS:
assumes WELL: Well-order r and
OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
shows (A < B) = embedS (Restr r A) (Restr r B) id
using assms by (simp add: ofilter-subset-embedS-iso)

lemma embed-implies-iso-Restr:
assumes WELL: Well-order r and WELL′: Well-order r′ and
EMB: embed r′ r f
shows iso r′ (Restr r (f ′ (Field r′))) f
proof –
let ?A′ = Field r′
let ?r′′ = Restr r (f ′ ( ?A′)
have 0: Well-order ?r′′ using WELL Well-order-Restr by blast
have 1: wo-rel.ofilter r (f ′ (?A′) using assms embed-Field-ofilter by blast
hence Field ?r′′ = f ′ (Field r′) using WELL Field-Restr-ofilter by blast
hence bij-betw f ?A′ (Field ?r′′) using EMB embed-inj-on WELL′ unfolding bij-beta-def by blast
moreover
{have ∀ a b. (?a , b) ∈ r′ =→ a ∈ Field r′ ∧ b ∈ Field r′
  unfolding Field-def by auto
  hence compat r′ ?r′′ f
  using assms embed-iff-compat-inj-on-ofilter
  unfolding compat-def by blast
}
ultimately show ?thesis using WELL′ 0 iso-iff3 by blast
qed

27.3 The strict inclusion on proper ofilters is well-founded

definition ofilterIncl :: 'a rel ⇒ 'a set rel
where
ofilterIncl r ≡ {(A,B). wo-rel.ofilter r A ∧ A ≠ Field r ∧
                 wo-rel.ofilter r B ∧ B ≠ Field r ∧ A < B}

lemma wf-ofilterIncl:
assumes WELL: Well-order r
shows wf(ofilterIncl r)
proof –
  have Well: wo-rel r using WELL by (simp add: wo-rel-def)
  hence Lo: Linear-order r by (simp add: wo-rel.LIN)
let \( ?h = (\lambda A. \text{wo-rel}\ r\ A) \)
let \( ?rS = r - \text{Id} \)
have \( \text{wf}\ ?rS\ \text{using WELL}\ \text{by (simp add: order-on-defs)} \)
moreover
have \( \text{compat (ofilterIncl r)} ?rS ?h \)

proof (unfold compat-def ofilterIncl-def, intro allI impI, simp, elim conjE)

fix \( A\ B \)
assume \( *: \text{wo-rel.ofilter}\ r\ A\ A \neq \text{Field}\ r\ \text{and} \)
\( **: \text{wo-rel.ofilter}\ r\ B\ B \neq \text{Field}\ r\ \text{and} \)
\( ***: A < B \)
then obtain \( a\ \text{and}\ b\ \text{where} \)
\( 0: a \in \text{Field}\ r \land b \in \text{Field}\ r\ \text{and} \)
\( 1: A = \text{underS } r\ a \land B = \text{underS } r\ b \)
using Well by (auto simp add: wo-rel.ofilter-underS-Field)
hence \( a \neq b\ \text{using} \)
moreover
have \( (a,b) \in r\ \text{using}\ 0 1\ \text{Lo} \)
by (auto simp add: underS-incl-iff)
moreover
have \( a = \text{wo-rel.suc}\ r\ A \land b = \text{wo-rel.suc}\ r\ B \)
using Well 0 1 by (simp add: wo-rel.suc-underS)
ultimately
show \( (\text{wo-rel.suc}\ r\ A, \text{wo-rel.suc}\ r\ B) \in r \land \text{wo-rel.suc}\ r\ A \neq \text{wo-rel.suc}\ r\ B \)
by simp
qed

ultimately show \( \text{wf (ofilterIncl r)}\ \text{by (simp add: compat-wf)} \)
qed

27.4 Ordering the well-orders by existence of embeddings

We define three relations between well-orders:

- \( \text{ordLeq}, \) of being embedded (abbreviated \( \leq_o \));
- \( \text{ordLess}, \) of being strictly embedded (abbreviated \( <_o \));
- \( \text{ordIso}, \) of being isomorphic (abbreviated \( =_o \)).

The prefix "ord" and the index "o" in these names stand for "ordinal-like". These relations shall be proved to be inter-connected in a similar fashion as the trio \( \leq, <, = \) associated to a total order on a set.

definition \( \text{ordLeq} :: (\text{'}a\ \text{rel} \times \text{'}a\ \text{rel})\ \text{set} \)
where
\( \text{ordLeq} = \{(r,r'). \text{ Well-order } r \land \text{ Well-order } r' \land (\exists f. \text{ embed } r r' f)\} \)

abbreviation \( \text{ordLeq2} :: \text{'}a\ \text{rel} \Rightarrow \text{'}a\ \text{rel} \Rightarrow \text{bool} \ (\text{infix} \ \leq_o 50) \)
where \( r \leq_o r' \equiv (r,r') \in \text{ordLeq} \)

abbreviation \( \text{ordLeq3} :: \text{'}a\ \text{rel} \Rightarrow \text{'}a\ \text{rel} \Rightarrow \text{bool} \ (\text{infix} \ \leq_o 50) \)
where \( r \leq_o r' \equiv r \leq_o r' \)
definition $\text{ordLess} :: (\text{'}a \text{ rel} \ast \text{'}a \text{ rel}) \text{ set}$
where
$\text{ordLess} = \{(r,r') \mid \text{Well-order } r \land \text{Well-order } r' \land (\exists f \ . \ \text{embedS } r \ r' \ f)\}$

abbreviation $\text{ordLess2 :: 'a rel \Rightarrow 'a rel \Rightarrow bool}$ (infix < o 50)
where $r < o r' \equiv (r,r') \in \text{ordLess}$

definition $\text{ordIso} :: (\text{'}a \text{ rel} \ast \text{'}a \text{ rel}) \text{ set}$
where
$\text{ordIso} = \{(r,r') \mid \text{Well-order } r \land \text{Well-order } r' \land (\exists f \ . \ \text{iso } r \ r' \ f)\}$

abbreviation $\text{ordIso2 :: 'a rel \Rightarrow 'a rel \Rightarrow bool}$ (infix = o 50)
where $r = o r' \equiv (r,r') \in \text{ordIso}$

lemmas $\text{ordRels-def} = \text{ordLeq-def ordLess-def ordIso-def}$

lemma $\text{ordLeq-Well-order-simp}$:
assumes $r \leq o r'$
shows $\text{Well-order } r \land \text{Well-order } r'$
using assms unfolding $\text{ordLeq-def}$ by simp

Notice that the relations $\leq o$, $< o$, $= o$ connect well-orders on potentially distinct types. However, some of the lemmas below, including the next one, restrict implicitly the type of these relations to $(\langle a \text{ rel} \rangle \ast \langle a \text{ rel} \rangle)$ set, i.e., to $\langle a \text{ rel} \rangle$.

lemma $\text{ordLeq-reflexive}$:
$\text{Well-order } r \Rightarrow r \leq o r$
unfolding $\text{ordLeq-def}$ using $\text{id-embed [of } r\text{]}$ by blast

lemma $\text{ordLeq-transitive}$:
assumes $*: r \leq o r'$ and $**: r' \leq o r''$
shows $r \leq o r''$
proof
obtain $f$ and $f'$
where $I: \text{Well-order } r \land \text{Well-order } r' \land \text{Well-order } r''$ and
embed $r \ r' \ f$ and embed $r' \ r'' \ f'$
using $**$ unfolding $\text{ordLeq-def}$ by blast
hence embed $r \ r'' \ (f' \circ f)$
using $\text{comp-embed [of } r \ r' \ f \ r'' \ f'\text{]}$ by auto
thus $r \leq o r''$ unfolding $\text{ordLeq-def}$ using $I$ by auto
qed

lemma $\text{ordLeq-total}$:
$[\text{Well-order } r; \text{Well-order } r'] \Rightarrow r \leq o r' \lor r' \leq o r$
unfolding $\text{ordLeq-def}$ using $\text{Wellorders-totally-ordered}$ by blast

lemma $\text{ordIso-reflexive}$:
$\text{Well-order } r \Rightarrow r = o r$
unfolding \textit{ordIso-def} using \textit{id-isoo[of r]} by blast

\textbf{lemma} \textit{ordIso-transitive}[trans]:
\textit{assumes} \textbf{\ast}: \textit{r} = o \textit{r}' \textbf{and} \textbf{\ast\ast}: \textit{r}' = o \textit{r}''
\textit{shows} \textit{r} = o \textit{r}''
\textit{proof} –
\textbf{obtain} \( f \) \textbf{and} \( f' \)
where \textbf{1}: \textit{Well-order r} \land \textit{Well-order r}' \land \textit{Well-order r}'' \textbf{and}
\textbf{2}: \textit{iso r r} f \textbf{and} \textbf{3}: \textit{iso r'} r'' f'
\textit{using} \textbf{\ast\ast} unfolding \textit{ordIso-def} by auto
\textbf{hence} \textit{iso r} r'' (f' \circ f) \textbf{by auto}
\textbf{thus} \textit{r} = o \textit{r}'' unfolding \textit{ordIso-def} using \( f \) by auto
\textbf{qed}

\textbf{lemma} \textit{ordIso-symmetric}:
\textit{assumes} \textbf{\ast}: \textit{r} = o \textit{r}'
\textit{shows} \textit{r}' = o \textit{r}
\textit{proof} –
\textbf{obtain} \( f \) where \textbf{1}: \textit{Well-order r} \land \textit{Well-order r}'
\textbf{2}: \textit{embed r r}' f \land \textit{bij-betw} f (\textit{Field r}) (\textit{Field r}')
\textit{using} \textbf{\ast} by (auto simp add: \textit{ordIso-def iso-def})
\textbf{let} \( ?f' = \textit{inv-into} (\textit{Field r}) f \)
\textbf{have} \textit{embed r'} r ?f' \land \textit{bij-betw} ?f' (\textit{Field r'}) (\textit{Field r})
\textit{using} \( 1 \ 2 \) \textbf{by} (simp add: bij-betw-inv-into \textit{inv-into-Field-embed-bij-betw})
\textbf{thus} \textit{r}' = o \textit{r} unfolding \textit{ordIso-def} using \( f \) by (auto simp add: iso-def)
\textbf{qed}

\textbf{lemma} \textit{ordLeq-ordLess-trans}[trans]:
\textit{assumes} \textit{r \leq o r}' \textbf{and} \textit{r}' \textbf{<} o \textit{r}''
\textit{shows} \textit{r} \textbf{<} o \textit{r}''
\textit{proof} –
\textbf{have} \textit{Well-order r} \land \textit{Well-order r}''
\textit{using} \textbf{assms} unfolding \textit{ordLeq-def ordLess-def} by auto
\textbf{thus} \textbf{\?thesis} using \textbf{assms} unfolding \textit{ordLeq-def ordLess-def}
\textit{using embed-comp-embedS} by blast
\textbf{qed}

\textbf{lemma} \textit{ordLess-ordLeq-trans}[trans]:
\textit{assumes} \textit{r} \textbf{<} o \textit{r}' \textbf{and} \textit{r}' \textbf{\leq} o \textit{r}''
\textit{shows} \textit{r} \textbf{<} o \textit{r}''
\textit{proof} –
\textbf{have} \textit{Well-order r} \land \textit{Well-order r}''
\textit{using} \textbf{assms} unfolding \textit{ordLeq-def ordLess-def} by auto
\textbf{thus} \textbf{\?thesis} using \textbf{assms} unfolding \textit{ordLeq-def ordLess-def}
\textit{using embedS-comp-embed} by blast
\textbf{qed}

\textbf{lemma} \textit{ordLeq-ordIso-trans}[trans]:
assumes $r \leq o r'$ and $r' = o r''$
shows $r \leq o r''$
proof
  have Well-order $r$ $\land$ Well-order $r''$
    using assms unfolding ordLeq-def ordIso-def by auto
  thus $\lnot$thesis using assms unfolding ordLeq-def ordIso-def
    using embed-comp-iso by blast
qed

lemma ordIso-ordLeq-trans[trans]:
assumes $r = o r'$ $\land$ $r' \leq o r''$
shows $r \leq o r''$
proof
  have Well-order $r$ $\land$ Well-order $r''$
    using assms unfolding ordLeq-def ordIso-def by auto
  thus $\lnot$thesis using assms unfolding ordLeq-def ordIso-def
    using iso-comp-embed by blast
qed

lemma ordLess-ordIso-trans[trans]:
assumes $r < o r'$ $\land$ $r' = o r''$
shows $r < o r''$
proof
  have Well-order $r$ $\land$ Well-order $r''$
    using assms unfolding ordLess-def ordIso-def by auto
  thus $\lnot$thesis using assms unfolding ordLess-def ordIso-def
    using embedS-comp-iso by blast
qed

lemma ordIso-ordLess-trans[trans]:
assumes $r = o r'$ $\land$ $r' < o r''$
shows $r < o r''$
proof
  have Well-order $r$ $\land$ Well-order $r''$
    using assms unfolding ordLess-def ordIso-def by auto
  thus $\lnot$thesis using assms unfolding ordLess-def ordIso-def
    using iso-comp-embedS by blast
qed

lemma ordLess-not-embed:
assumes $r < o r'$
shows $\lnot$($\exists f'. \text{ embed } r' r f'$)
proof
  obtain $f$ where 1: Well-order $r$ $\land$ Well-order $r'$ $\land$ $2$: embed $r r' f$ and
    3: $\lnot$ bij-betw $f$ (Field $r$) (Field $r'$)
  using assms unfolding ordLess-def by (auto simp add: embedS-def)
  { fix $f'$ assume $*: \text{ embed } r' r f'$
    hence bij-betw $f$ (Field $r$) (Field $r'$) using 1 2
    by (simp add: embed-bothWays-Field-bij-betw) }
with 3 have False by contradiction
}
thus ?thesis by blast
qed

lemma ordLess-Field:
assumes OL: r1 <o r2 and EMB: embed r1 r2 f
shows ¬ (f'(Field r1) = Field r2)
proof¬
  let ?A1 = Field r1  let ?A2 = Field r2
  obtain g where
  0: Well-order r1 ∧ Well-order r2 and
  1: embed r1 r2 g ∧ ¬(bij-betw g ?A1 ?A2)
  using OL unfolding ordLess-def by (auto simp add: embedS-def)
  hence ∀ a ∈ ?A1. f a = g a
  using 0 EMB embed-unique[of r1] by auto
  hence ¬(bij-betw f ?A1 ?A2)
  using 1 bij-betw-cong[of ?A1] by blast
moreover
  have inj-on f ?A1 using EMB 0 by (simp add: embed-inj-on)
ultimately show ?thesis by (simp add: bij-betw-def)
qed

lemma ordLess-iff:
r <o r' = (Well-order r ∧ Well-order r' ∧ ¬(∃ f'. embed r' r f'))
proof
  assume *: r <o r'
  hence ¬(∃ f'. embed r' r f') using ordLess-not-embed[of r r'] by simp
  with * show Well-order r ∧ Well-order r' ∧ ¬(∃ f'. embed r' r f')
  unfolding ordLess-def by auto
next
  assume *: Well-order r ∧ Well-order r' ∧ ¬(∃ f'. embed r' r f')
  then obtain f where 1: embed r r' f
  using wellorders-totally-ordered[of r r'] by blast
moreover
  {assume bij-betw f (Field r) (Field r')
   with * 1 have embed r' r (inv-into (Field r) f)
   using inv-into-Field-embed-bij-betw[of r r' f] by auto
   with * have False by blast
  }
ultimately show (r,r') ∈ ordLess
unfolding ordLess-def using * by (fastforce simp add: embedS-def)
qed

lemma ordLess-irreflexive:  ¬ r <o r
proof
  assume r <o r
  hence Well-order r ∧ ¬(∃ f. embed r r f)
  unfolding ordLess-iff ..
moreover have $\text{embed } r \ id \ using \ id\text{-embed}[\text{of } r]$.
ultimately show $\text{False by blast}$
qed

\textbf{lemma} \textit{ordLeq-iff-ordLess-or-ordIso}:
$r \leq o \ r' = (r < o \ r' \lor r = o \ r')$
\textit{unfolding ordRels-def embedS-defs iso-defs by blast}

\textbf{lemma} \textit{ordIso-iff-ordLeq}:
$(r = o \ r') = (r \leq o \ r' \land r' \leq o \ r)$
\textit{proof}
assume $r = o \ r'$
then obtain $f$ where \textit{1}: Well-order $r$ \land Well-order $r'$ \land $\text{embed } r \ r' \ f \land \text{bij-betw } f \ (\text{Field } r) \ (\text{Field } r')$
\textit{unfolding ordIso-def iso-defs by auto}
hence $\text{embed } r \ r' \ f \land \text{embed } r' \ r \ g$
\textit{by (simp add: inv-into-Field-embed-bij-betw)}
thus $r \leq o \ r' \land r' \leq o \ r$
\textit{unfolding ordLeq-def using 1 by auto}
next
assume $r \leq o \ r' \land r' \leq o \ r$
then obtain $f$ and $g$ where \textit{1}: Well-order $r$ \land Well-order $r'$ \land $\text{embed } r \ r' \ f \land \text{embed } r' \ r \ g$
\textit{unfolding ordLeq-def by auto}
hence $\text{iso } r \ r'$ \text{f by (auto simp add: embed-bothWays-iso)}
thus $r = o \ r'$ \textit{unfolding ordIso-def using 1 by auto}
qed

\textbf{lemma} \textit{not-ordLess-ordLeq}:
$r < o \ r' \Rightarrow \neg r' \leq o \ r$
\textit{using ordLess-ordLeq-trans ordLess-irreflexive by blast}

\textbf{lemma} \textit{ordLess-or-ordLeq}:
\textit{assumes WELL: Well-order $r$ and WELL': Well-order $r'$}
shows $r < o \ r' \lor r' \leq o \ r$
\textit{proof–}
have $r \leq o \ r' \lor r' \leq o \ r$
\textit{using assms by (simp add: ordLeq-total)}
moreover
{\textit{assume }$\neg r < o \ r' \land r \leq o \ r'$
hence $r = o \ r'$ \textit{using ordLeq-iff-ordLess-or-ordIso by blast}
hence $r' \leq o \ r$ \textit{using ordIso-symmetric ordIso-iff-ordLeq by blast}}
ultimately show $\text{thesis by blast}$
qed

\textbf{lemma} \textit{not-ordLess-ordIso}:
$r < o \ r' \Rightarrow \neg r = o \ r'$
\textit{using ordLess-ordIso-trans ordIso-symmetric ordLess-irreflexive by blast}
lemma not-ordLeq-iff-ordLess:
assumes WELL: Well-order r and WELL': Well-order r'
shows \((\neg r' \leq o r) = (r < o r')\)
using assms not-ordLess-ordLeq ordLess-or-ordLeq by blast

lemma not-ordLess-iff-ordLeq:
assumes WELL: Well-order r and WELL': Well-order r'
shows \((\neg r' < o r) = (r \leq o r')\)
using assms not-ordLess-ordLeq ordLess-or-ordLeq by blast

lemma ordLess-transitive[trans]:
\[[r < o r'; r' < o r''] \implies r < o r''\]
using ordLess-ordLeq-trans ordLeq-iff-ordLess-or-ordIso by blast

corollary ordLess-trans: trans ordLess
unfolding trans-def using ordLess-transitive by blast

lemmas ordIso-equivalence = ordIso-transitive ordIso-reflexive ordIso-symmetric

lemma ordIso-imp-ordLeq:
r = o r' \implies r \leq o r'
using ordIso-iff-ordLeq by blast

lemma ordLess-imp-ordLeq:
r < o r' \implies r \leq o r'
using ordLeq-iff-ordLess-or-ordIso by blast

lemma ofilter-subset-ordLeq:
assumes WELL: Well-order r and OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
shows \((A \leq B) = (\text{Restr } r A \leq o \text{Restr } r B)\)
proof
assume A \leq B
thus \text{Restr } r A \leq o \text{Restr } r B
unfolding ordLeq-def using assms
Well-order-Restr Well-order-Restr ofilter-subset-embed by blast
next
assume *: \text{Restr } r A \leq o \text{Restr } r B
then obtain f where embed \((\text{Restr } r A) (\text{Restr } r B) f\)
unfolding ordLeq-def by blast
{assume B < A
hence \text{Restr } r B < o \text{Restr } r A
unfolding ordLess-def using assms
Well-order-Restr Well-order-Restr ofilter-subset-embedS by blast
hence False using * not-ordLess-ordLeq by blast}
} thus A \leq B using OFA OFB WELL
wo-rel-def[of r] wo-rel.ofilter-linord[of r A B] by blast
lemma ofilter-subset-ordLess:
assumes WELL: Well-order r and
 OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
shows \((A < B) = (\text{Restr } r A <_o \text{Restr } r B)\)
proof –
  let ?rA = Restr r A let ?rB = Restr r B
  have 1: Well-order ?rA ∧ Well-order ?rB
  using WELL Well-order-Restr by blast
  have \((A < B) = (\neg B \leq A)\) using assms
  wo-rel-def wo-rel.ofilter-linord[of r A B] by blast
  also have \(\ldots = (\neg \text{Restr } r B \leq_0 \text{Restr } r A)\)
  using assms ofilter-subset-ordLeq by blast
also have \(\ldots = (\text{Restr } r A <_o \text{Restr } r B)\)
using 1 not-ordLeq-iff-ordLess by blast
finally show ?thesis
qed

lemma ofilter-ordLess:
\[ [\text{Well-order } r; \text{wo-rel.ofilter } r A] \implies (A < Field r) = (\text{Restr } r A <_o r) \] 
by (simp add: ofilter-subset-ordLess wo-rel.Field-ofilter
 wo-rel-def Restr-Field)

corollary underS-Restr-ordLess:
assumes Well-order r and Field r ≠ \{
shows \(\text{Restr } r (\text{underS } r a) <_o r\)
proof –
  have underS r a < Field r using assms
  by (simp add: underS-Field3)
  thus ?thesis using assms
  by (simp add: ofilter-ordLess wo-rel.underS-ofilter wo-rel-def)
qed

lemma embed-ordLess-ofilterIncl:
assumes
 OL12: \(r1 <_o r2\) and OL23: \(r2 <_o r3\) and
 EMB13: embed r1 r3 f13 and EMB23: embed r2 r3 f23
shows \((f13'(\text{Field } r1), f23'(\text{Field } r2)) \in (\text{ofilterIncl } r3)\)
proof –
  have OL13: \(r1 <_o r3\)
  using OL12 OL23 using ordLess-transitive by auto
  let ?A1 = Field r1 let ?A2 =Field r2 let ?A3 =Field r3
  obtain f12 g23 where
  \(0: \text{Well-order } r1 \land \text{Well-order } r2 \land \text{Well-order } r3\) and
  1: embed r1 r2 f12 ∧ ¬(bij-betw f12 ?A1 ?A2) and
  2: embed r2 r3 g23 ∧ ¬(bij-betw g23 ?A2 ?A3)
  using OL12 OL23 by (auto simp add: ordLess-def embedS-def)
hence \(\forall a \in ?A2. f23 a = g23 a\)
THEORY "BNF-Wellorder-Constructions"

using EMB23 embed-unique[of r2 r3] by blast
hence 3: ¬(bij-betw f23 ?A2 ?A3)
using 2 bij-betw-cong[of ?A2 f23 g23] by blast

have 4: wo-rel.ofilter r2 (f12 ' ?A1) ∧ f12 ' ?A1 ≠ ?A2
using 0 1 OL12 by (simp add: embed-Field-ofilter ordLess-Field)
have 5: wo-rel.ofilter r3 (f23 ' ?A2) ∧ f23 ' ?A2 ≠ ?A3
using 0 EMB23 OL23 by (simp add: embed-Field-ofilter ordLess-Field)
using 0 EMB13 OL13 by (simp add: embed-Field-ofilter ordLess-Field)

have f12 ' ?A1 < ?A2
using 0 4 by (auto simp add: wo-rel-ofilter-def wo-rel_def)
moreover have inj-on f23 ?A2
using EMB23 0 by (simp add: embed-inj-on)
ultimately
have f23 ' (f12 ' ?A1) < f23 ' ?A2 by (simp add: inj-on-strict-subset)
moreover
{ have embed r1 r3 (f23 o f12)
  using 1 EMB23 0 by (auto simp add: comp-embed)
  hence ∃a ∈ ?A1. f23(f12 a) = f13 a
  using EMB13 0 embed-unique[of r1 r3 f23 o f12 f13] by auto
  hence f23 ' (f12 ' ?A1) = f13 ' ?A1 by force
}
ultimately
have f13 ' ?A1 < f23 ' ?A2 by simp

with 5 6 show ?thesis
unfolding ofilterIncl-def by auto
qed

lemma ordLess-iff-ordIso-Restr:
assumes WELL: Well-order r and WELL': Well-order r'
shows (r' <o r) = (∃a ∈ Field r. r'=o Restr r (underS r a))
proof(auto)
  fix a assume *: a ∈ Field r and **: r'=o Restr r (underS r a)
  hence Restr r (underS r a) <o r using WELL underS-Restr-ordLess[of r] by blast
  thus r' <o r using ** ordIso-ordLess-trans by blast
next
  assume r' <o r
  then obtain f where 1: Well-order r ∧ Well-order r' and
  2: embed r' f r ∧ f ' (Field r') ≠ Field r
  unfolding ordLess-def embedS-def[abs-def] bij-betw-def using embed-inj-on by blast
  hence wo-rel.ofilter r (f ' (Field r')) using embed-Field-ofilter by blast
  then obtain a where 3: a ∈ Field r and 4: underS r a = f ' (Field r')
  using 1 2 by (auto simp add: wo-rel.ofilter-Field underS-wo-rel_def)
  have iso r' (Restr r (f ' (Field r'))) f
using embed-implies-iso-Restr 2 assms by blast
moreover have Well-order (Restr r (f ′ (Field r′)))
using WELL Well-order-Restr by blast
ultimately have r′ = o Restr r (f ′ (Field r′))
using WELL unfolding ordIso-def by auto
hence r′ = o Restr r (underS r a) using 4 by auto
thus ∃a ∈ Field r. r′ = o Restr r (underS r a) using 3 by auto
qed

lemma internalize-ordLess:
(r′ < o r) = (∃p. Field p < Field r ∧ r′ = o p ∧ p < o r)
proof
assume ∗: r′ < o r
hence 0: Well-order r ∧ Well-order r′ unfolding ordLess-def by auto
with ∗ obtain a where 1: a ∈ Field r and 2: r′ = o Restr r (underS r a)
using ordLess-iff-ordIso-Restr by blast
let ?p = Restr r (underS r a)
have wo-rel.ofilter r (underS r a) using 0
by (simp add: wo-rel-def wo-rel.ofilter)
by (simp add: wo-rel-def wo-rel.ofilter)
hence Field ?p = underS r a using 0 Field-Restr-ofilter by blast
hence Field ?p < Field r using underS-FIELD 1 by fast
moreover have ?p < o r using underS-Restr-ordLess[of r a] 0 1 by fast
ultimately
ultimately have ?p < o r using underS-Restr-ordLess[of r a] 0 1 by fast
ultimately
ultimately show ∃p. Field p < Field r ∧ r′ = o p ∧ p < o r using 2 by blast
next
assume ∃p. Field p < Field r ∧ r′ = o p ∧ p < o r
thus r′ < o r using ordIso-ordLess-trans by blast
qed

lemma internalize-ordLeq:
(r′ ≤ o r) = (∃p. Field p ≤ Field r ∧ r′ = o p ∧ p ≤ o r)
proof
assume ∗: r′ ≤ o r
moreover
{assume r′ < o r
then obtain p where Field p < Field r ∧ r′ = o p ∧ p < o r
using internalize-ordLess[of r′ r] by blast
hence ∃p. Field p < Field r ∧ r′ = o p ∧ p < o r
using ordLeq-iff-ordLess-or-ordIso by blast
}
moreover
have r ≤ o r using * ordLeq-def ordLeq-reflexive by blast
ultimately show ∃p. Field p ≤ Field r ∧ r′ = o p ∧ p ≤ o r
using ordLeq-iff-ordLess-or-ordIso by blast
next
assume ∃p. Field p ≤ Field r ∧ r′ = o p ∧ p ≤ o r
thus r′ ≤ o r using ordIso-ordLeq-trans by blast
qed
lemma ordLeq-iff-ordLess-Restr:
assumes WELL: Well-order r and WELL′: Well-order r′
shows \((r \trianglelefteq o r′) = (\forall a \in \text{Field } r. \text{Restr } r (\underS r a) < o r′)\)
proof(auto)
  assume \(*\): \(r \leq o r′\)
  fix \(a\) assume \(a \in \text{Field } r\)
  hence \(\text{Restr } r (\underS r a) < o r′\)
  using WELL underS-Restr-ordLess[of r] by blast
  thus \(\text{Restr } r (\underS r a) < o r′\)
  using \(*\) ordLess-ordLeq-trans by blast
next
  assume \(*\): \(\forall a \in \text{Field } r. \text{Restr } r (\underS r a) < o r′\)
  {assume \(r′ \leq o r\)
   then obtain \(h\) where \(h \in \text{Field } r \land r′ = o \text{Restr } r (\underS r a)\)
   using assms ordLess-iff-ordIso-Restr by blast
   hence \(\text{False}\) using \(*\) not-ordLess-ordIso ordIso-symmetric by blast
  } thus \(r \leq o r′\) using ordLess-or-ordLeq assms by blast
qed

lemma finite-ordLess-infinite:
assumes WELL: Well-order r and WELL′: Well-order r′ and
FIN: finite(\(\text{Field } r\)) and INF: \(\neg\)finite(\(\text{Field } r′\))
shows \(r < o r′\)
proof
  {assume \(r′ \leq o r\)
   then obtain \(h\) where \(\text{inj-on } h (\text{Field } r′) \land h′ (\text{Field } r′) \leq \text{Field } r\)
   unfolding ordLeq-def using assms embed-inj-on embed-Field by blast
   hence \(\text{False}\) using finite-imageD finite-subset FIN INF by blast
  }
  thus \(?\text{thesis}\) using WELL WELL′ ordLess-or-ordLeq by blast
qed

lemma finite-well-order-on-ordIso:
assumes FIN: finite \(A\) and
WELL: well-order-on \(A\) \(r\) and WELL′: well-order-on \(A\) \(r′\)
shows \(r = o r′\)
proof
  have \(0\): Well-order \(r\) \land Well-order \(r′\) \land \(\text{Field } r = A \land \text{Field } r′ = A\)
  using assms well-order-on-Well-order by blast
moreover
  have \(\forall r r′\). well-order-on \(A\) \(r\) \land well-order-on \(A\) \(r′\) \land \(r \leq o r′\)
  hence \(\Rightarrow r = o r′\)
proof(clarify)
  fix \(r r′\) assume \(*\): well-order-on \(A\) \(r\) and \(**\): well-order-on \(A\) \(r′\)
  have \(2\): Well-order \(r\) \land Well-order \(r′\) \land \(\text{Field } r = A \land \text{Field } r′ = A\)
  using \(*\) \(**\) well-order-on-Well-order by blast
  assume \(r \leq o r′\)
  then obtain \(f\) where \(1\): embed \(r r′ \) \(f\) and
inj-on f A \land f \upharpoonright A \leq A

unfolding \textit{ordLeq-def} using \textit{2 embed-inj-on embed-Field} by \textit{blast}

hence bij-btw f A A unfolding bij-btw-def using \textit{FIN endo-inj-surj} by \textit{blast}

thus r = o r' unfolding \textit{ordIso-def iso-def[abs-def]} using \textit{1 2} by \textit{auto}

qed

ultimately show \textit{thesis} using \textit{assms ordLeq-total ordIso-symmetric} by \textit{blast}

qed

27.5 \( < o \) is well-founded

Of course, it only makes sense to state that the \( < o \) is well-founded on the restricted type \( 'a \ rel \ rel \). We prove this by first showing that, for any set of well-orders all embedded in a fixed well-order, the function mapping each well-order in the set to an order filter of the fixed well-order is compatible w.r.t. to \( < o \) versus \textit{strict inclusion}; and we already know that \textit{strict inclusion} of order filters is well-founded.

definition \textit{ord-to-filter} :: \( 'a \ rel \Rightarrow 'a \ rel \Rightarrow 'a \ set \)

where \textit{ord-to-filter} \( r0 \ r \equiv (\text{SOME} f. \text{embed} r \ r0 f) \uparrow (\text{Field} r) \)

lemma \textit{ord-to-filter-compat}:\textit{compat}(\textit{ordLess Int} (\textit{ordLess}^{-1} \{r0\} \times \textit{ordLess}^{-1} \{r0\}))

(of\textit{filterIncl} r0)

(of\textit{filter} → \textit{ord-less}\textit{compted})

proof(unfold \textit{compat-def} \textit{ord-to-filter-def}, clarify)

fix \( r1::'a \ rel \ and \ r2::'a \ rel \)

let \( ?A1 = \text{Field} r1 \ let \ ?A2 = \text{Field} r2 \ let \ ?A0 = \text{Field} r0 \)

let \( ?\phi10 = \lambda f10. \text{embed} r1 r0 f10 \ let \ ?f10 = \text{SOME} f. ?\phi10 f \)

let \( ?\phi20 = \lambda f20. \text{embed} r2 r0 f20 \ let \ ?f20 = \text{SOME} f. ?\phi20 f \)

assume \( \ast \): \( r1 < o r0 \ r2 < o r0 \ and \ \ast \ast \): \( r1 < o r2 \)

hence \( (\exists f. ?\phi10 f) \uparrow (\exists f. ?\phi20 f) \)

by (\textit{auto simp add: ordLess-def embedS-def})

hence \( ?\phi10 ?f10 \uparrow ?\phi20 ?f20 \) by (\textit{auto simp add: someI-ex})

thus \( (?f10 \uparrow ?A1, ?f20 \uparrow ?A2) \in \textit{of\textit{filterIncl} r0} \)

using \( \ast \ast \) by (\textit{simp add: embed-ordLess-of\textit{filterIncl}})

qed

theorem \textit{wf-ordLess}: \textit{wf ordLess}

proof\textit{–}

\( \{ \text{fix} \ r0 :: (\textit{'}a \times \textit{'}a) \ \text{set} \} \)

let \( ?\textit{ordLess} = \textit{ordLess-def}('d \ rel \ast 'd \ rel) \ \text{set} \)

let \( ?R = ?\textit{ordLess Int} (?\textit{ordLess}^{-1} \{r0\} \times ?\textit{ordLess}^{-1} \{r0\}) \)

\( \{ \text{assume Case1: Well-order} r0 \} \)

hence \( \textit{wf ?R} \)

using \( \textit{wf-of\textit{filterIncl}[of r0]} \)

\textit{compat-wf}[of \ ?R of\textit{filterIncl} r0 \textit{ord-to-filter} r0]

\textit{ord-to-filter-compat}[of r0] by \textit{auto}
moreover
{ assume Case2: ¬ Well-order r0
  hence ?R = {} unfolding ordLess-def by auto
  hence wf ?R using wf-empty by simp
}
ultimately have wf ?R by blast
thus ?thesis by (simp add: trans-wf-iff ordLess-trans)
qed

corollary exists-minim-Well-order:
assumes NE: R ≠ {} and WELL: ∀ r ∈ R. Well-order r
shows ∃ r ∈ R. ∀ r' ∈ R. r ≤o r'
proof -
  obtain r where r ∈ R ∧ (∀ r' ∈ R. ¬ r' <o r)
using NE spec[OF spec[OF subst[OF wf-eq-minimal, of %x. x, OF wf-ordLess]], of - R]
equalsI[of R] by blast
with not-ordLeq-iff-ordLess WELL show ?thesis by blast
qed

27.6 Copy via direct images
The direct image operator is the dual of the inverse image operator \textit{inv-image}
from Relation.thy. It is useful for transporting a well-order between different
types.

definition dir-image :: 'a rel ⇒ ('a ⇒ 'a') ⇒ 'a' rel
  where
  dir-image r f = \{(f a, f b)\ | a b. (a,b) ∈ r\}

lemma dir-image-Field:
Field(dir-image r f) = f ` (Field r)
unfolding dir-image-def Field-def Range-def Domain-def by fast

lemma dir-image-minus-Id:
inj-on f (Field r) ⟹ (dir-image r f) - Id = dir-image (r - Id) f
unfolding inj-on-def Field-def dir-image-def by auto

lemma Refl-dir-image:
assumes Refl r
shows Refl(dir-image r f)
proof -
  { fix a' b'
    assume (a',b') ∈ dir-image r f
    then obtain a b where 1: a' = f a ∧ b' = f b ∧ (a,b) ∈ r
    unfolding dir-image-def by blast
    hence a ∈ Field r ∧ b ∈ Field r using Field-def by fastforce
    hence (a,a) ∈ r ∧ (b,b) ∈ r using assms by (simp add: refl-on-def)
    with 1 have (a',a') ∈ dir-image r f ∧ (b',b') ∈ dir-image r f
  }
unfolding dir-image-def by auto

} thus thesis
by (unfold refl-on-def Field-def Domain-def Range-def, auto)
qed

lemma trans-dir-image:
assumes TRANS: trans r and INJ: inj-on f (Field r)
shows trans (dir-image r f)
proof (unfold trans-def, auto)
fix a' b' c'
assume (a', b') ∈ dir-image r f (b', c') ∈ dir-image r f
then obtain a b1 b2 c where 1: a' = f a ∧ b' = f b1 ∧ c' = f c
and
2: (a, b1) ∈ r ∧ (b2, c) ∈ r

unfolding dir-image-def by blast
hence b1 ∈ Field r ∧ b2 ∈ Field r

unfolding Field-def by auto
hence b1 = b2 using 1 INJ unfolding inj-on-def by auto
hence (a, c) ∈ r using 2 TRANS unfolding trans-def by blast
thus (a', c') ∈ dir-image r f
unfolding dir-image-def using 1 by auto
qed

lemma Preorder-dir-image:
[ Preorder r; inj-on f (Field r) ] ⇒ Preorder (dir-image r f)
by (simp add: preorder-on-def Refl-dir-image trans-dir-image)

lemma antisym-dir-image:
assumes AN: antisym r and INJ: inj-on f (Field r)
shows antisym (dir-image r f)
proof (unfold antisym-def, auto)
fix a' b'
assume (a', b') ∈ dir-image r f (b', a') ∈ dir-image r f
then obtain a1 b1 a2 b2 where 1: a' = f a1 ∧ a' = f a2 ∧ b' = f b1 ∧ b' = f b2
and
2: (a1, b1) ∈ r ∧ (b2, a2) ∈ r

unfolding dir-image-def Field-def by blast
hence a1 = a2 ∧ b1 = b2 using 1 INJ unfolding inj-on-def by auto
hence a1 = b2 using 2 AN unfolding antisym-def by auto
thus a' = b' using 1 by auto
qed

lemma Partial-order-dir-image:
[ Partial-order r; inj-on f (Field r) ] ⇒ Partial-order (dir-image r f)
by (simp add: partial-order-on-def Preorder-dir-image antisym-dir-image)

lemma Total-dir-image:
assumes \( \text{TOT}: \text{Total} \ r \ \text{and} \ \text{INJ}: \text{inj-on} \ f (\text{Field} \ r) \)
shows \( \text{Total}(\text{dir-image} \ r \ f) \)
proof (unfold \text{total-on-def}, \text{intro ballI impI})
  fix \( a' \ b' \)
  assume \( a' \in \text{Field} \ (\text{dir-image} \ r \ f) \ b' \in \text{Field} \ (\text{dir-image} \ r \ f) \)
then obtain \( a \ \text{and} \ b \) where \( 1: a \in \text{Field} \ r \land b \in \text{Field} \ r \land f \ a = a' \land f \ b = b' \)
  unfolding \( \text{dir-image-Field}[of \ r \ f] \) by blast
moreover assume \( a' \neq b' \)
ultimately have \( a \neq b \) using \( \text{INJ} \ unfolding \ \text{inj-on-def} \) by auto
hence \( (a,b) \in r \lor (b,a) \in r \) using \( \text{1 TOT} \ unfolding \ \text{total-on-def} \) by auto
thus \( (a',b') \in \text{dir-image} \ r \ f \lor (b',a') \in \text{dir-image} \ r \ f \)
using \( \text{1 unfolding} \ \text{dir-image-def} \) by auto
qed

lemma \( \text{Linear-order-dir-image}: \)
\[ [\text{Linear-order} \ r; \ \text{inj-on} \ f (\text{Field} \ r)] \implies \text{Linear-order} \ (\text{dir-image} \ r \ f) \]
by (simp add: linear-order-on-def Partial-order-dir-image Total-dir-image)

lemma \( \text{wf-dir-image}: \)
assumes \( \text{WF}: \text{wf} \ r \ \text{and} \ \text{INJ}: \text{inj-on} \ f (\text{Field} \ r) \)
shows \( \text{wf}(\text{dir-image} \ r \ f) \)
proof (unfold \( \text{wf-eq-minimal2} \), intro allI impI, elim conjE)
  fix \( A':\b \ \text{set} \)
  assume \( \text{SUB}: A' \subseteq \text{Field}(\text{dir-image} \ r \ f) \ \text{and} \ \text{NE}: A' \neq \{\} \)
  obtain \( A \) where \( \text{A-def}: A = \{a \in \text{Field} \ r. f \ a \in A'\} \) by blast
  have \( A \neq \{\} \land A \subseteq \text{Field} \ r \) using \( \text{A-def} \ \text{SUB} \ \text{NE} \) by (auto simp: \text{dir-image-Field})
  then obtain \( a \) where \( 1: a \in A \land (\forall b \in A. (b,a) \notin r) \)
  using \( \text{spec[OF WF[unfolded \text{wf-eq-minimal2}], of A]} \) by blast
  have \( \forall b' \in A'. (b', f a) \notin \text{dir-image} \ r \ f \)
  proof (clarify)
    fix \( b'\) assume \( \ast: b' \in A' \) and \( \ast\ast: (b', f a) \in \text{dir-image} \ r \ f \)
    obtain \( b1 a1 \) where \( 2: b' = f b1 \land f a = f a1 \) and
    \( 3: (b1,a1) \in r \land \{a1,b1\} \subseteq \text{Field} \ r \)
    using \( \ast\ast \) unfolding \( \text{dir-image-def Field-def} \) by blast
    hence \( a = a1 \) using \( 1 \ \text{A-def} \ \text{INJ} \ unfolding \ \text{inj-on-def} \) by auto
    hence \( b1 \in A \land (b1,a1) \in r \) using \( 2 \ 3 \ \text{A-def} \ \ast \) by auto
    with \( 1 \) show \( \text{False} \) by auto
  qed
  thus \( \exists a' \in A'. \forall b' \in A'. (b', a') \notin \text{dir-image} \ r \ f \)
  using \( \text{A-def} \ \text{1} \) by blast
  qed

lemma \( \text{Well-order-dir-image}: \)
\[ [\text{Well-order} \ r; \ \text{inj-on} \ f (\text{Field} \ r)] \implies \text{Well-order} \ (\text{dir-image} \ r \ f) \]
unfolding \( \text{well-order-on-def} \)
using \( \text{Linear-order-dir-image[of r f] \ \text{wf-dir-image[of r Id f]} \}
\text{dir-image-minus-Id[of f r] \ \text{subset-inj-on[of f Field r Field(r Id)]}} \)
\text{mono-Field[of r Id r] by auto}
lemma dir-image-bij-betw:
\[ \text{inj-on } f (\text{Field } r) \implies \text{bij-betw } f (\text{Field } r) (\text{Field } (\text{dir-image } r f)) \]

unfolding bij-betw-def by (simp add: dir-image-Field order-on-defs)

lemma dir-image-compat:
compat r (dir-image r f) f

unfolding compat-def dir-image-def by auto

lemma dir-image-iso:
\[ \text{Well-order } r ; \text{inj-on } f (\text{Field } r) \implies \text{iso } r (\text{dir-image } r f) f \]

using iso-iff3 dir-image-compat dir-image-bij-betw Well-order-dir-image by blast

lemma dir-image-ordIso:
\[ \text{Well-order } r ; \text{inj-on } f (\text{Field } r) \implies r = o \text{ dir-image } r f \]

unfolding ordIso-def dir-image-iso Well-order-dir-image by blast

lemma Well-order-iso-copy:
assumes WELL: \text{well-order-on } A r \text{ and BIJ: } \text{bij-betw } f A A'

shows \( \exists r', \text{well-order-on } A' r' \land r = o r' \)

proof –

let \(?r' = \text{dir-image } r f\)

have 1: \( A = \text{Field } r \land \text{Well-order } r \)

using WELL well-order-on-Well-order by blast

hence 2: \( \text{iso } r ?r' f \)

using dir-image-iso using BIJ unfolding bij-betw-def by auto

hence \( f ' (\text{Field } r) = \text{Field } ?r' \) using 1 iso-iff[of r ?r'] by blast

hence Field ?r' = A'

using 1 BIJ unfolding bij-betw-def by auto

moreover have \( \text{Well-order } ?r' \)

using 1 Well-order-dir-image BIJ unfolding bij-betw-def by blast

ultimately show \( \exists \text{thesis unfolding ordIso-def using } 1 \ 2 \) by blast

qed

27.7 Bounded square

This construction essentially defines, for an order relation \( r \), a lexicographic order \( \text{bsqr } r \) on \( (\text{Field } r) \times (\text{Field } r) \), applying the following criteria (in this order):

- compare the maximums;
- compare the first components;
- compare the second components.

The only application of this construction that we are aware of is at proving that the square of an infinite set has the same cardinal as that set. The essential property required there (and which is ensured by this construction)
is that any proper order filter of the product order is included in a rectangle, i.e., in a product of proper filters on the original relation (assumed to be a well-order).

**definition** bsqr :: 'a rel => ('a * 'a)rel
where
bsqr r = \{(a1,a2),(b1,b2)\).
    \{a1,a2,b1,b2\} ≤ Field r ∧
    (a1 = b1 ∧ a2 = b2 ∨
     wo-rel.max2 r a1 a2, wo-rel.max2 r b1 b2) ∈ r − Id ∨
    wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2 ∧ (a1,b1) ∈ r − Id ∨
    wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2 ∧ a1 = b1 ∧ (a2,b2) ∈ r
− Id
\}

**lemma** Field-bsqr:
Field (bsqr r) = Field r × Field r
**proof**
  show Field (bsqr r) ≤ Field r × Field r
  **proof**−
  \{fix a1 a2 assume (a1,a2) ∈ Field (bsqr r)
      moreover
      have \(b1,b2,(a1,a2),(b1,b2)\) ∈ bsqr r ∨ ((b1,b2),(a1,a2)) ∈ bsqr r =⇒
      a1 ∈ Field r ∧ a2 ∈ Field r unfolding bsqr-def by auto
  \}
  ultimately have a1 ∈ Field r ∧ a2 ∈ Field r unfolding Field-def by auto
  thus ?thesis unfolding Field-def by force
  qed
next
  show Field r × Field r ≤ Field (bsqr r)
  **proof**(auto)
    fix a1 a2 assume a1 ∈ Field r and a2 ∈ Field r
    hence ((a1,a2),(a1,a2)) ∈ bsqr r unfolding bsqr-def by blast
    thus (a1,a2) ∈ Field (bsqr r) unfolding Field-def by auto
  qed

**lemma** bsqr-Refl:
Refl(bsqr r)
by(unfold refl-on-def Field-bsqr, auto simp add: bsqr-def)

**lemma** bsqr-Trans:
assumes Well-order r
shows trans (bsqr r)
**proof**(unfold trans-def, auto)

  have Well: wo-rel r using assms wo-rel-def by auto
  hence Trans: trans r using wo-rel.TRANS by auto
  have Anti: antisym r using wo-rel.ANTISYM Well by auto
  hence TransS: trans(r − Id) using Trans by (simp add: trans-diff-Id)
\[ \text{fix } a_1 a_2 b_1 b_2 c_1 c_2 \]
\[ \text{assume } \vdash ((a_1,a_2),(b_1,b_2)) \in \text{bsqr } r \text{ and } \vdash ((b_1,b_2),(c_1,c_2)) \in \text{bsqr } r \]
\[ \text{hence } \theta : \{a_1,a_2,b_1,b_2,c_1,c_2\} \leq \text{Field } r \text{ unfolding bsqr-def by auto} \]
\[ \text{have } 1: a_1 = b_1 \wedge a_2 = b_2 \vee (\text{wo-rel.max2 } r \ a_1 a_2, \text{wo-rel.max2 } r \ b_1 b_2) \in r - Id \]
\[ \vdash \text{wo-rel.max2 } r \ a_1 a_2 = \text{wo-rel.max2 } r \ b_1 b_2 \wedge (a_1,b_1) \in r - Id \lor \]
\[ \vdash \text{wo-rel.max2 } r \ a_1 a_2 = \text{wo-rel.max2 } r \ b_1 b_2 \wedge a_1 = b_1 \wedge (a_2,b_2) \in r - \]
\[ \vdash \text{using } \ast \text{ unfolding bsqr-def by auto} \]
\[ \text{have } 2: b_1 = c_1 \wedge b_2 = c_2 \vee (\text{wo-rel.max2 } r \ b_1 b_2, \text{wo-rel.max2 } r \ c_1 c_2) \in r - Id \]
\[ \vdash \text{wo-rel.max2 } r \ b_1 b_2 = \text{wo-rel.max2 } r \ c_1 c_2 \wedge (b_1,c_1) \in r - Id \lor \]
\[ \vdash \text{wo-rel.max2 } r \ b_1 b_2 = \text{wo-rel.max2 } r \ c_1 c_2 \wedge b_1 = c_1 \wedge (b_2,c_2) \in r - \]
\[ \vdash \text{Id} \text{ using } \ast \text{ unfolding bsqr-def by auto} \]
\[ \text{show } ((a_1,a_2),(c_1,c_2)) \in \text{bsqr } r \]
\[ \text{proof -} \]
\[ \{\text{assume Case1: } a_1 = b_1 \wedge a_2 = b_2 \}
\[ \text{hence } \theta \text{thesis using } \ast \text{ by simp} \}
\]
\[ \text{moreover} \]
\[ \{\text{assume Case2: } (\text{wo-rel.max2 } r \ a_1 a_2, \text{wo-rel.max2 } r \ b_1 b_2) \in r - Id \}
\[ \{\text{assume Case21: } b_1 = c_1 \wedge b_2 = c_2 \}
\[ \text{hence } \theta \text{thesis using } \ast \text{ by simp} \}
\]
\[ \text{moreover} \]
\[ \{\text{assume Case22: } (\text{wo-rel.max2 } r \ b_1 b_2, \text{wo-rel.max2 } r \ c_1 c_2) \in r - Id \}
\[ \text{hence } (\text{wo-rel.max2 } r \ a_1 a_2, \text{wo-rel.max2 } r \ c_1 c_2) \in r - Id \]
\[ \text{using Case2 TransS trans-def[of r - Id] by blast} \]
\[ \text{hence } \theta \text{thesis using } 0 \text{ unfolding bsqr-def by auto} \}
\]
\[ \text{moreover} \]
\[ \{\text{assume Case23-4: } \text{wo-rel.max2 } r \ b_1 b_2 = \text{wo-rel.max2 } r \ c_1 c_2 \}
\[ \text{hence } \theta \text{thesis using Case2 0 unfolding bsqr-def by auto} \}
\]
\[ \text{moreover} \]
\[ \{\text{assume Case3: } \text{wo-rel.max2 } r \ a_1 a_2 = \text{wo-rel.max2 } r \ b_1 b_2 \wedge (a_1,b_1) \in r - Id \}
\[ \{\text{assume Case31: } b_1 = c_1 \wedge b_2 = c_2 \}
\[ \text{hence } \theta \text{thesis using } \ast \text{ by simp} \}
\]
\[ \text{moreover} \]
\[ \{\text{assume Case32: } (\text{wo-rel.max2 } r \ b_1 b_2, \text{wo-rel.max2 } r \ c_1 c_2) \in r - Id \}
\[ \text{hence } \theta \text{thesis using Case3 0 unfolding bsqr-def by auto} \}
\]
\[ \text{moreover} \]
\[ \{\text{assume Case33: } \text{wo-rel.max2 } r \ b_1 b_2 = \text{wo-rel.max2 } r \ c_1 c_2 \wedge (b_1,c_1) \in r \}
\]
THEORY “BNF-Wellorder-Constructions”

− Id
  hence \((a_1, c_1) \in r − Id\)
  using Case3 TransS trans-def[of r − Id] by blast
  hence ?thesis using Case3 Case33 0 unfolding bsqr-def by auto
}\nmoreover
{assume Case33: wo-rel.max2 r b1 b2 = wo-rel.max2 r c1 c2 ∧ b1 = c1
  hence ?thesis using Case3 0 unfolding bsqr-def by auto
}
ultimately have ?thesis using 0 2 by auto
}\nultimately have ?thesis using 0 1 by auto
qed

lemma bsqr-antisym:
assumes Well-order r
shows antisym (bsqr r)
proof(unfold antisym-def, clarify)

  have Well: wo-rel r using assms wo-rel-def by auto
  hence Trans: trans r using wo-rel.TRANS by auto
  have Anti: antisym r using wo-rel.ANTISYM Well by auto
  hence TransS: \((r − Id)\) using Trans by (simp add: trans-diff-Id)
  hence IrrS: \(∀ a b. ¬((a, b)) \in r − Id ∧ (b, a) \in r − Id\)
using \( \text{Anti trans-def[of } r \text{ - Id]} \) \( \text{antisym-def[of } r \text{ - Id]} \) by blast

fix \( a1 a2 b1 b2 \)
assume \( *: ((a1,a2),(b1,b2)) \in \text{bsqr } r \) and \( **: ((b1,b2),(a1,a2)) \in \text{bsqr } r \)
hence \( \theta: \{a1,a2,b1,b2\} \leq \text{Field } r \) unfolding \( \text{bsqr-def by auto} \)
have \( 1: a1 = b1 \land a2 = b2 \lor (\text{wo-rel.max2 } r \ a1 \ a2, \text{wo-rel.max2 } r \ b1 \ b2) \in r \land \text{Id} \lor \)
\( \text{wo-rel.max2 } r \ a1 \ a2 = \text{wo-rel.max2 } r \ b1 \ b2 \land (a1,b1) \in r \land \text{Id} \lor \)
\( \text{wo-rel.max2 } r \ a1 \ a2 = \text{wo-rel.max2 } r \ b1 \ b2 \land a1 = b1 \land (a2,b2) \in r \land \text{Id} \)
using \( * \) unfolding \( \text{bsqr-def by auto} \)
have \( 2: b1 = a1 \land b2 = a2 \lor (\text{wo-rel.max2 } r \ b1 \ b2, \text{wo-rel.max2 } r \ a1 \ a2) \in r \land \text{Id} \lor \)
\( \text{wo-rel.max2 } r \ b1 \ b2 = \text{wo-rel.max2 } r \ a1 \ a2 \land (b1,a1) \in r \land \text{Id} \lor \)
\( \text{wo-rel.max2 } r \ b1 \ b2 = \text{wo-rel.max2 } r \ a1 \ a2 \land b1 = a1 \land (b2,a2) \in r \land \text{Id} \)
using \( ** \) unfolding \( \text{bsqr-def by auto} \)
show \( a1 = b1 \land a2 = b2 \)

proof –
\{ assume Case1: \( (\text{wo-rel.max2 } r \ a1 \ a2, \text{wo-rel.max2 } r \ b1 \ b2) \in r \land \text{Id} \)
\{ assume Case11: \( (\text{wo-rel.max2 } r \ b1 \ b2, \text{wo-rel.max2 } r \ a1 \ a2) \in r \land \text{Id} \)
  hence \( \text{False using Case1 IrrS by blast} \) \}
moreover
\{ assume Case12-3: \( \text{wo-rel.max2 } r \ b1 \ b2 = \text{wo-rel.max2 } r \ a1 \ a2 \)
  hence \( \text{False using Case1 by auto} \) \}
ultimately have \( \text{thesis using } \theta \ 2 \text{ by auto} \) \}
moreover
\{ assume Case2: \( \text{wo-rel.max2 } r \ a1 \ a2 = \text{wo-rel.max2 } r \ b1 \ b2 \land (a1,b1) \in r \land \text{Id} \)
\{ assume Case21: \( (\text{wo-rel.max2 } r \ b1 \ b2, \text{wo-rel.max2 } r \ a1 \ a2) \in r \land \text{Id} \)
  hence \( \text{False using Case2 by auto} \) \}
moreover
\{ assume Case22: \( (b1,a1) \in r \land \text{Id} \)
  hence \( \text{False using Case2 IrrS by blast} \) \}
moreover
\{ assume Case23: \( b1 = a1 \)
  hence \( \text{False using Case2 by auto} \) \}
ultimately have \( \text{thesis using } \theta \ 2 \text{ by auto} \) \}
moreover
\{ assume Case3: \( \text{wo-rel.max2 } r \ a1 \ a2 = \text{wo-rel.max2 } r \ b1 \ b2 \land a1 = b1 \land (a2,b2) \in r \land \text{Id} \)
moreover
{assume Case31: \((\text{wo-rel}\cdot\text{max2}\ r\ b1\ b2,\ \text{wo-rel}\cdot\text{max2}\ r\ a1\ a2) \in r - \text{Id})
  hence False using Case3 by auto }

moreover
{assume Case32: \((b1,a1) \in r - \text{Id})
  hence False using Case3 by auto }

moreover
{assume Case33: \((b2,a2) \in r - \text{Id})
  hence False using Case3 \text{IrrS by blast} }

ultimately have \(?\text{thesis using 0 2 by auto}\)
}

ultimately show \(?\text{thesis using 0 1 by blast}\)
qed

lemma bsqr-Total:
assumes \text{Well-order } r
shows \text{Total}(\text{bsqr } r)
proof –

have \text{Well: wo-rel } r \text{ using assms wo-rel-def by auto}
hence \text{Total: } \forall a \in \text{Field } r. \forall b \in \text{Field } r. (a,b) \in r \lor (b,a) \in r
using \text{wo-rel.TOTALS by auto}

{fix \(a1\ a2\ b1\ b2\) assume \(\{(a1,a2), (b1,b2)\} \leq \text{Field(bsqr } r)\)
hence \(a1 \in \text{Field } r \land a2 \in \text{Field } r \land b1 \in \text{Field } r \land b2 \in \text{Field } r\)
using \text{Field-bsqr by blast}

have \((a1,a2) = (b1,b2) \lor ((a1,a2),(b1,b2)) \in \text{bsqr } r \lor ((b1,b2),(a1,a2)) \in \text{bsqr } r)\)
proof(rule \text{wo-rel.cases-Total[of } r\ a1\ a2\], clarsimp simp add: \text{Well, simp add: 0})

assume Case1: \((a1,a2) \in r\)
hence 1: \text{wo-rel.max2 } r\ a1\ a2 = a2
using \text{Well 0 by (simp add: \text{wo-rel.max2-equals2)}}
show \(?\text{thesis}\)
proof(rule \text{wo-rel.cases-Total[of } r\ b1\ b2\], clarsimp simp add: \text{Well, simp add: 0})

assume Case11: \((b1,b2) \in r\)
hence 2: \text{wo-rel.max2 } r\ b1\ b2 = b2
using \text{Well 0 by (simp add: \text{wo-rel.max2-equals2)}}
show \(?\text{thesis}\)
proof(rule \text{wo-rel.cases-Total3[of } r\ a2\ b2\], clarsimp simp add: \text{Well, simp add: 0})

assume Case111: \((a2,b2) \in r - \text{Id} \lor (b2,a2) \in r - \text{Id}\)
thus \(?\text{thesis using 0 1 2 unfolding } \text{bsqr-def by auto}\)
next
assume Case112: $a_2 = b_2$

show ?thesis

proof (rule wo-rel.cases-Total3[of $r\ a_1 b_1$], clarsimp simp add: Well, simp add: 0)

  assume Case1121: $(a_1,b_1) \in r - Id \lor (b_1,a_1) \in r - Id$
  thus ?thesis using 0 1 2 Case112 unfolding bsqr-def by auto

next

  assume Case1122: $a_1 = b_1$
  thus ?thesis using Case112 by auto

qed

next

assume Case112: $(b_2,b_1) \in r$

hence 3: wo-rel.max2 $r\ b_1 b_2 = b_1$ using Well 0 by (simp add: wo-rel.max2-equals1)

show ?thesis

proof (rule wo-rel.cases-Total3[of $r\ a_2 b_1$], clarsimp simp add: Well, simp add: 0)

  assume Case121: $(a_2,b_1) \in r - Id \lor (b_1,a_2) \in r - Id$
  thus ?thesis using 0 1 3 unfolding bsqr-def by auto

next

  assume Case122: $a_2 = b_1$
  show ?thesis

proof (rule wo-rel.cases-Total3[of $r\ a_1 b_1$], clarsimp simp add: Well, simp add: 0)

  assume Case1221: $(a_1,b_1) \in r - Id \lor (b_1,a_1) \in r - Id$
  thus ?thesis using 0 1 3 Case122 unfolding bsqr-def by auto

next

  assume Case1222: $a_1 = b_1$
  show ?thesis

proof (rule wo-rel.cases-Total3[of $r\ a_2 b_2$], clarsimp simp add: Well, simp add: 0)

  assume Case12221: $(a_2,b_2) \in r - Id \lor (b_2,a_2) \in r - Id$
  thus ?thesis using 0 1 3 Case122 Case1222 unfolding bsqr-def by auto

next

  assume Case12222: $a_2 = b_2$
  thus ?thesis using Case122 Case1222 by auto

qed

next

assume Case2: $(a_2,a_1) \in r$

hence 1: wo-rel.max2 $r\ a_1 a_2 = a_1$ using Well 0 by (simp add: wo-rel.max2-equals1)

show ?thesis

proof (rule wo-rel.cases-Total[of $r\ b_1 b_2$], clarsimp simp add: Well, simp add: 0)

  assume Case21: $(b_1,b_2) \in r$
  hence 2: wo-rel.max2 $r\ b_1 b_2 = b_2$ using Well 0 by (simp add: wo-rel.max2-equals2)
showthesis
proof(rule wo-rel-cases-Total3[of r a1 b2], clarsimp simp add: Well, simp add: 0)
  assume Case211: \((a1,b2) \in r - Id \lor (b2,a1) \in r - Id\)
  thus thesis using 0 1 2 unfolding bsqr-def by auto
next
  assume Case212: \(a1 = b2\)
  show thesis
proof(rule wo-rel-cases-Total3[of r a1 b1], clarsimp simp add: Well, simp add: 0)
  assume Case2121: \((a1,b1) \in r - Id \lor (b1,a1) \in r - Id\)
  thus thesis using 0 1 2 Case212 unfolding bsqr-def by auto
next
  assume Case2122: \(a1 = b1\)
  show thesis
proof(rule wo-rel-cases-Total3[of r a2 b2], clarsimp simp add: Well, simp add: 0)
  assume Case21221: \((a2,b2) \in r - Id \lor (b2,a2) \in r - Id\)
  thus thesis using 0 1 2 Case212 Case2122 unfolding bsqr-def by auto
next
  assume Case21222: \(a2 = b2\)
  thus thesis using Case2122 Case212 by auto
qed
qed
qed

assumecase 22: \((b2,b1) \in r\)

hence 3: wo-rel.max2 r b1 b2 = b1 using Well 0 by (simp add: wo-rel.max2-equals1)

show thesis
proof(rule wo-rel-cases-Total3[of r a1 b1], clarsimp simp add: Well, simp add: 0)
  assume Case221: \((a1,b1) \in r - Id \lor (b1,a1) \in r - Id\)
  thus thesis using 0 1 3 unfolding bsqr-def by auto
next
  assume Case222: a1 = b1
  show thesis
proof(rule wo-rel-cases-Total3[of r a2 b2], clarsimp simp add: Well, simp add: 0)
  assume Case2221: \((a2,b2) \in r - Id \lor (b2,a2) \in r - Id\)
  thus thesis using 0 1 3 Case222 unfolding bsqr-def by auto
next
  assume Case2222: a2 = b2
  thus thesis using Case222 by auto
qed
qed
qed

thus thesis unfolding total-on-def by fast qed

lemma bsqr-Linear-order:
assumes Well-order r
shows Linear-order(bsqr r)
unfolding order-on-defs
using assms bsqr-Refl bsqr-Trans bsqr-antisym bsqr-Total by blast

lemma bsqr-Well-order:
assumes Well-order r
shows Well-order(bsqr r)
using assms proof (simp add: bsqr-Linear-order Linear-order-Well-order-iff, intro allI impI)
have 0: \( \forall A \leq \text{Field } r. A \neq \{\} \longrightarrow (\exists a \in A. \forall a' \in A. (a,a') \in r)\)
using assms well-order-on-def Linear-order-Well-order-iff by blast
fix D assume \( \star: D \leq \text{Field } (bsqr r) \) and \( \star\star: D \neq \{\} \)
hence 1: \( D \leq \text{Field } r \times \text{Field } r \) unfolding Field-bsqr by simp
obtain M where M-def: \( M = \{\text{wo-rel}.\max2 r a1 a2| a1 a2. (a1,a2) \in D\} \) by blast
have M \( \neq \{\} \) using 1 M-def \( \star\star \) by auto
moreover have M \( \leq \text{Field } r \) unfolding M-def
using 1 assms wo-rel-def[of r] wo-rel.max2-among[of r] by fastforce
ultimately obtain m where m-min: \( m \in M \land (\forall a \in M. (m,a) \in r) \)
using 0 by blast
obtain A1 where A1-def: \( A1 = \{a1. \exists a2. (a1,a2) \in D \land \text{wo-rel}.\max2 r a1 a2 = m\} \) by blast
have A1 \( \leq \text{Field } r \) unfolding A1-def using 1 by auto
moreover have A1 \( \neq \{\} \) unfolding A1-def using m-min unfolding M-def by blast
ultimately obtain a1 where a1-min: \( a1 \in A1 \land (\forall a \in A1. (a1,a) \in r) \)
using 0 by blast
obtain A2 where A2-def: \( A2 = \{a2. (a1,a2) \in D \land \text{wo-rel}.\max2 r a1 a2 = m\} \) by blast
have A2 \( \leq \text{Field } r \) unfolding A2-def using 1 by auto
moreover have A2 \( \neq \{\} \) unfolding A2-def
using m-min a1-min unfolding A1-def M-def by blast
ultimately obtain a2 where a2-min: \( a2 \in A2 \land (\forall a \in A2. (a2,a) \in r) \)
using 0 by blast
have 2: wo-rel.max2 r a1 a2 = m
using a1-min a2-min unfolding A1-def A2-def by auto
have 3: \((a1,a2) \in D\) using a2-min unfolding A2-def by auto
moreover
{fix b1 b2 assume ***: (b1,b2) ∈ D
hence 4: {a1,a2,b1,b2} ≤ Field r using 1 3 by blast
have 5: (wo-rel.max2 r a1 a2, wo-rel.max2 r b1 b2) ∈ r
using *** a1-min a2-min m-min unfolding A1-def A2-def M-def by auto
have ((a1,a2),(b1,b2)) ∈ bsqr r
proof(cases wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2)
assume Case1: wo-rel.max2 r a1 a2 ≠ wo-rel.max2 r b1 b2
thus ?thesis unfolding bsqr-def using 4 5 by auto
next
assume Case2: wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2
hence b1 ∈ A1 unfolding A1-def using 2 *** by auto
hence 6: (a1,b1) ∈ r using a1-min by auto
show ?thesis
proof(cases a1 = b1)
assume Case21: a1 ≠ b1
thus ?thesis unfolding bsqr-def using 4 Case2 6 by auto
next
assume Case22: a1 = b1
hence b2 ∈ A2 unfolding A2-def using 2 *** Case2 by auto
hence 7: (a2,b2) ∈ r using a2-min by auto
thus ?thesis unfolding bsqr-def using 4 7 Case2 Case22 by auto
qed
qed
}
ultimately show ∃ d ∈ D. ∀ d′ ∈ D. (d,d′) ∈ bsqr r by fastforce
qed

lemma bsqr-max2:
assumes WELL: Well-order r and LEQ: ((a1,a2),(b1,b2)) ∈ bsqr r
shows (wo-rel.max2 r a1 a2, wo-rel.max2 r b1 b2) ∈ r
proof–
have {(a1,a2),(b1,b2)} ≤ Field(bsqr r)
using LEQ unfolding Field-def by auto
hence {a1,a2,b1,b2} ≤ Field r unfolding Field-bsqr by auto
hence {wo-rel.max2 r a1 a2, wo-rel.max2 r b1 b2} ≤ Field r
using WELL wo-rel-def[of r] wo-rel.max2-among[of r] by fastforce
moreover have (wo-rel.max2 r a1 a2, wo-rel.max2 r b1 b2) ∈ r ∨ wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2
using LEQ unfolding bsqr-def by auto
ultimately show ?thesis using WELL unfolding order-on-defs refl-on-def by auto
qed

lemma bsqr-ofilter:
assumes WELL: Well-order r and
OF: wo-rel.ofilter (bsqr r) D and SUB: D < Field r × Field r and
NE: ¬ (∃ a. Field r = under r a)
shows ∃ A. wo-rel.ofilter r A ∧ A < Field r ∧ D ≤ A × A
proof –
let \(?r' = bsqr r\)
have Well: wo-rel \(r\) using WELL wo-rel-def by blast
hence Trans: trans \(r\) using wo-rel.TRANS by blast
have Well': Well-order \(?r' \land wo-rel \(?r'\)
using WELL bsqr-Well-order wo-rel-def by blast
have \(D < Field \(?r'\)\) unfolding Field-bsqr using SUB .
with OF obtain \(a_1\) and \(a_2\) where
\((a_1,a_2) \in Field \(?r'\) and 1: \(D = underS \(?r' (a_1,a_2)\)
using Well' wo-rel.ofilter-underS-Field[of \(?r' D\)] by auto
hence 2: \(\{a_1,a_2\} \leq Field \(r\) unfolding Field-bsqr by auto
let \(?m = wo-rel.max2 r a_1 a_2\)
have \(D \leq (under r \(?m\)) \times (under r \(?m\))\)
proof\((\text{unfold } 1)\)
\{\fix \(b_1 b_2\)
let \(?n = wo-rel.max2 r b_1 b_2\)
assume \((b_1,b_2) \in underS \(?r' (a_1,a_2)\)
hence 3: \((b_1,b_2),(a_1,a_2)\) \(?r'\)
unfolding underS-def by blast
hence \((?n,?m) \in r using WELL by \((\text{simp add: bsqr-max2})\)
moreover
\{have \((b_1,b_2) \in Field \(r\) unfolding Field-def by auto
hence \(b_1,?n) \in r \land (b_2,?n) \in r\)
using Well by \((\text{simp add: wo-rel.max2-greater})\)\}
ultimately have \((b_1,?m) \in r \land (b_2,?m) \in r\)
using Trans trans-def[of \(r\)] by blast
hence \((b_1,b_2) \in (under r \(?m\)) \times (under r \(?m\)) unfolding under-def by simp}\)
thus underS \(?r' (a_1,a_2) \leq (under r \(?m\)) \times (under r \(?m\)) by auto
qed
moreover have wo-rel.ofilter \(r\) (under \(?m\))
using Well by \((\text{simp add: wo-rel.ofilter})\)
moreover have under r \(?m < Field \(r\)
using NE under-Field[of \(r\) \(?m\)] by blast
ultimately show \(?thesis by blast
qed

definition Func where
Func A B = \{f . (\forall a \in A. f a \in B) \land (\forall a. a \notin A \longrightarrow f a = \text{undefined})\}

lemma Func-empty:
Func \{\} B = \{\lambda x. undefined\}
unfolding Func-def by auto

lemma Func-elim:
assumes \(g \in Func A B\) and \(a \in A\)
shows \(\exists b. b \in B \land g a = b\)
using assms unfolding Func-def by (cases g a = undefined) auto

definition curr where
  curr A f ≡ λ a. if a ∈ A then λb. f (a, b) else undefined

lemma curr-in:
  assumes f: f ∈ Func (A × B) C
  shows curr A f ∈ Func A (Func B C)
  using assms unfolding curr-def Func-def by auto

lemma curr-inj:
  assumes f1 ∈ Func (A × B) C and f2 ∈ Func (A × B) C
  shows curr A f1 = curr A f2 ⟷ f1 = f2
  proof safe
    assume c: curr A f1 = curr A f2
    show f1 = f2
      proof (rule ext, clarify)
        fix a b show f1 (a, b) = f2 (a, b)
        proof (cases (a, b) ∈ A × B)
          case False
          thus ?thesis using assms unfolding Func-def by auto
        next
          case True hence a: a ∈ A and b: b ∈ B by auto
          thus ?thesis
            using c unfolding curr-def fun-eq-iff by elim allE[0f - a] simp
        qed
      qed
  qed

lemma curr-surj:
  assumes g ∈ Func A (Func B C)
  shows ∃ f ∈ Func (A × B) C. curr A f = g
  proof
    let ?f = λ ab. if fst ab ∈ A ∧ snd ab ∈ B then g (fst ab) (snd ab) else undefined
    show curr A ?f = g
      proof (rule ext)
        fix a show curr A ?f a = g a
        proof (cases a ∈ A)
          case False
          hence g a = undefined using assms unfolding Func-def by auto
          thus ?thesis unfolding curr-def using False by simp
        next
          case True
          obtain g1 where g1 ∈ Func B C and g a = g1
            using assms using Func-elim[0F assms True] by blast
          thus ?thesis using True unfolding Func-def curr-def by auto
        qed
      qed
    show ?f ∈ Func (A × B) C using assms unfolding Func-def mem-Collect-eq
by auto

qed

lemma bij-betw-curr:
  bij-betw (curr A) (Func (A × B) C) (Func A (Func B C))
unfolding bij-betw-def inj-on-def image-def
apply (intro impl conjI ballI)
apply (erule curr-inj[THEN iffD1], assumption+)
apply auto
apply (erule curr-in)
using curr-surj by blast

definition Func-map where
  Func-map B2 f1 f2 g b2 ≡ if b2 ∈ B2 then f1 (g (f2 b2)) else undefined

lemma Func-map:
  assumes g: g ∈ Func A2 A1 and f1: f1 ' A1 ⊆ B1 and f2: f2 ' B2 ⊆ A2
  shows Func-map B2 f1 f2 g ∈ Func B2 B1
using assms unfolding Func-def Func-map-def mem-Collect-eq by auto

lemma Func-non-emp:
  assumes B ≠ {} 
  shows Func A B ≠ {}
proof -
  obtain b where b: b ∈ B using assms by auto
  hence (λ a. if a ∈ A then b else undefined) ∈ Func A B unfolding Func-def
  by auto
  thus ?thesis by blast
qed

lemma Func-is-emp:
  Func A B = {} ←→ A ≠ {} ∧ B = {} (is ?L ↔ ?R)
proof
  assume L: ?L
  moreover {assume A = {} hence False using L Func-empty by auto}
  moreover {assume B ≠ {} hence False using L Func-non-emp[of B A] by simp }
  ultimately show ?R by blast
next
  assume R: ?R
  moreover
  {fix f assume f ∈ Func A B
   moreover obtain a where a ∈ A using R by blast
   ultimately obtain b where b ∈ B unfolding Func-def by blast
   with R have False by blast
  }
  thus ?L by blast
qed
lemma Func-map-surj:
assumes B1: f1 : A1 \rightarrow B1 and A2: inj-on f2 B2 f2 ' B2 \subseteq A2
and B2A2: B2 = {} \implies A2 = {}
.shows Func B2 B1 = Func-map B2 f1 f2 ' Func A2 A1
proof (cases B2 = {})
case True
  thus ?thesis using B2A2 by (auto simp: Func-empty Func-map-def)
next
case False
  note B2 = False
  show ?thesis
  proof (safe)
    fix h
    assume h: h \in Func B2 B1
    define j1 where j1 = inv-into A1 f1
    have \forall a2 \in f2 ' B2. \exists b2. b2 \in B2 \land f2 b2 = a2 by blast
    then obtain k where k: \forall a2 \in f2 ' B2. k a2 \in B2 \land (k a2) = a2
    by atomize-elim (rule bchoice)
    { fix b2
      assume b2: b2 \in B2
      hence f2 (k (f2 b2)) = f2 b2 using k A2(2) by auto
      moreover have k (f2 b2) \in B2 using b2 A2(2) k by auto
      ultimately have k (f2 b2) = b2 using b2 A2(1) unfolding inj-on-def by blast
    } note kk = this
    obtain j2 where j2: b22: b22 \in B2 using B2 by auto
    define j2 where [abs-def]: j2 a2 = (if a2 \in f2 ' B2 then k a2 else b22) for a2
    have j2A2: j2 ' A2 \subseteq B2 unfolding j2-def using k b22 by auto
    have j2: \forall b2. b2 \in B2 \implies j2 (f2 b2) = b2
    using kk unfolding j2-def by auto
    define g where g = Func-map A2 j1 j2 h
    have Func-map B2 f1 f2 g = h
    proof (rule ext)
      fix b2
      show Func-map B2 f1 f2 g b2 = h b2
      proof (cases b2 \in B2)
        case True
        show ?thesis
        proof (cases h b2 = undefined)
          case True
          hence b1: h b2 \in f1 ' A1 using h (b2 \in B2) unfolding B1 Func-def by auto
          show ?thesis using A2 f-inv-into-f[OF b1]
          unfolding True g-def Func-map-def j1-def j2[OF \ b2 \in B2] by auto
          qed(insert A2 True j2[OF True] h B1, unfold j1-def g-def Func-def Func-map-def,
          auto intro: f-inv-into-f)
          qed(insert h, unfold Func-def Func-map-def, auto)
          qed
        moreover have g \in Func A2 A1 unfolding g-def apply (rule Func-map[OF h])
        using j2A2 B1 A2 unfolding j1-def by (fast intro: inv-into-into)+
        ultimately show h \in Func-map B2 f1 f2 ' Func A2 A1
unfolding \textit{Func-map-def[abs-def]} by \textit{auto} \\
\textit{qed(insert B1 Func-map[OF - - A2(2)], auto)} \\
\textit{qed}

end

28 Cardinal-Order Relations as Needed by Bounded Natural Functors

\texttt{theory BNF-Cardinal-Order-Relation} \\
\texttt{imports Zorn BNF-Wellorder-Constructions} \\
\texttt{begin}

In this section, we define cardinal-order relations to be minim well-orders on their field. Then we define the cardinal of a set to be some cardinal-order relation on that set, which will be unique up to order isomorphism. Then we study the connection between cardinals and:

- standard set-theoretic constructions: products, sums, unions, lists, powersets, set-of finite sets operator;
- finiteness and infiniteness (in particular, with the numeric cardinal operator for finite sets, \texttt{card}, from the theory \texttt{Finite-Sets.thy}).

On the way, we define the canonical $\omega$ cardinal and finite cardinals. We also define (again, up to order isomorphism) the successor of a cardinal, and show that any cardinal admits a successor.

Main results of this section are the existence of cardinal relations and the facts that, in the presence of infiniteness, most of the standard set-theoretic constructions (except for the powerset) do not increase cardinality. In particular, e.g., the set of words/lists over any infinite set has the same cardinality (hence, is in bijection) with that set.

28.1 Cardinal orders

A cardinal order in our setting shall be a well-order \textit{minim} w.r.t. the order-embedding relation, $\leq_o$ (which is the same as being \textit{minimal} w.r.t. the strict order-embedding relation, $<o$), among all the well-orders on its field.

\texttt{definition card-order-on :: 'a set $\Rightarrow$ 'a rel $\Rightarrow$ bool} \\
\texttt{where} \\
\texttt{card-order-on A r $\equiv$ well-order-on A r $\wedge$ ($\forall$ r'. well-order-on A r' $\rightarrow$ r $\leq_o$ r')}

\texttt{abbreviation Card-order r $\equiv$ card-order-on (Field r) r} \\
\texttt{abbreviation card-order r $\equiv$ card-order-on UNIV r}
lemma card-order-on-well-order-on:
assumes card-order-on A r
shows well-order-on A r
using assms unfolding card-order-on-def by simp

lemma card-order-on-Card-order:
card-order-on A r ⇒ A = Field r ∧ Card-order r
unfolding card-order-on-def using well-order-on-Field by blast

The existence of a cardinal relation on any given set (which will mean that any set has a cardinal) follows from two facts:

- Zermelo’s theorem (proved in Zorn.thy as theorem well-order-on), which states that on any given set there exists a well-order;
- The well-founded-ness of \(<\circ\), ensuring that then there exists a minimal such well-order, i.e., a cardinal order.

theorem card-order-on: \(\exists\, r.\) card-order-on A r
proof -
obtain R where R-def: R = \{r. well-order-on A r\} by blast
have 1: R \(\neq\) {} ∧ (\(\forall\, r \in R.\) Well-order r)
using well-order-on[of A] R-def well-order-on-Well-order by blast
hence \(\exists\, r \in R.\, \forall\, r' \in R.\, r \leq o \, r'\)
using exists-minim-Well-order[of R] by auto
thus ?thesis using R-def unfolding card-order-on-def by auto
qed

lemma card-order-on-ordIso:
assumes CO: card-order-on A r and CO': card-order-on A r'
shows r = o r'
using assms unfolding card-order-on-def

lemma Card-order-ordIso:
assumes CO: Card-order r and ISO: r' = o r
shows Card-order r'
using ISO unfolding ordIso-def
proof
(unfold card-order-on-def, auto)
fix p' assume well-order-on (Field r') p'
hence 0: Well-order p' ∧ Field p' = Field r'
using well-order-on-Well-order by blast
obtain f where 1: iso r' r \(\circ\) f and 2: Well-order r ∧ Well-order r'
using ISO unfolding ordIso-def by auto
hence 3: inj-on f (Field r') ∧ \(f'\) (Field r') = Field r
by (auto simp add: iso-iff embed-inj-on)
let ?p = dir-image p' f
have 4: p' = o ?p ∧ Well-order ?p
using 0 2 3 by (auto simp add: dir-image-ordIso Well-order-dir-image)
moreover have Field ?p = Field r
using 0 ? by (auto simp add: dir-image-Field)
ultimately have well-order-on (Field r) ?p by auto
hence r ≤o ?p using CO unfolding card-order-on-def by auto
thus r' ≤o p' using ISO 4 ordLeq-ordIso-trans ordIso-ordLeq-trans ordIso-symmetric by blast
qed

lemma Card-order-ordIso2:
assumes CO: Card-order r and ISO: r =o r'
shows Card-order r'
using assms Card-order-ordIso ordIso-symmetric by blast

28.2 Cardinal of a set

We define the cardinal of set to be some cardinal order on that set. We shall prove that this notion is unique up to order isomorphism, meaning that order isomorphism shall be the true identity of cardinals.
definition card-of :: 'a set ⇒ 'a rel (| - |)
where card-of A = (SOME r. card-order-on A r)

lemma card-of-card-order-on: card-order-on A |A|
unfolding card-of-def by (auto simp add: card-order-on someI-ex)

lemma card-of-well-order-on: well-order-on A |A|
using card-of-card-order-on card-order-on-def by blast

lemma Field-card-of: Field |A| = A
using card-of-card-order-on[of A] unfolding card-order-on-def
using well-order-on-Field by blast

lemma card-of-Card-order: Card-order |A|
by (simp only: card-of-card-order-on Field-card-of)
corollary ordIso-card-of-imp-Card-order:
  r =o |A| ⇒ Card-order r
using card-of-Card-order Card-order-ordIso by blast

lemma card-of-Well-order: Well-order |A|
using card-of-Card-order unfolding card-order-on-def by auto

lemma card-of-refl: |A| =o |A|
using card-of-Well-order ordIso-reflexive by blast

lemma card-of-least: well-order-on A r =o |A| ≤o r
using card-of-card-order-on unfolding card-order-on-def by blast

lemma card-of-ordIso:
(∃ f. bij-betw f A B) = ( |A| =o |B| )
proof (auto)
fix f assume *: bij-betw f A B
then obtain r where well-order-on B r ⊔ |A| = o r
using Well-order-iso-copy card-of-well-order-on by blast
hence |B| ≤ o |A| using card-of-least
ordLeq-ordIso-trans ordIso-symmetric by blast
moreover
{let ?g = inv-into A f
have bij-betw ?g B A using * bij-betw-inv-into by blast
then obtain r where well-order-on A r ⊔ |B| = o r
using Well-order-iso-copy card-of-well-order-on by blast
hence |A| ≤ o |B| using card-of-least
ordLeq-ordIso-trans ordIso-symmetric by blast }
ultimately show |A| = o |B| using ordIso-iff-ordLeq by blast
next
assume *: |A| = o |B|
then obtain f where iso ( |A| ) ( |B| ) f
unfolding ordIso-def by auto
hence bij-betw f A B unfolding iso-def Field-card-of by simp
thus ∃ f. bij-betw f A B by auto
qed

lemma card-of-ordLeq:
(∃ f. inj-on f A ∧ f ‘ A ≤ B) = ( |A| ≤ o |B| )
proof (auto)
fix f assume *: inj-on f A and **: f ‘ A ≤ B
{assume |B| < o |A|
hence |B| ≤ o |A| using ordLeq-iff-ordLess-or-ordIso by blast
then obtain g where embed ( |B| ) ( |A| ) g
unfolding ordLeq-def by auto
obtain h where bij-betw h A B
using * ** 1 Schroeder-Bernstein[of f] by fastforce
hence |A| = o |B| using card-of-ordIso by blast
hence |A| ≤ o |B| using ordIso-iff-ordLeq by auto }
thus |A| ≤ o |B| using ordLess-or-ordLeq[of |B| |A|]
by (auto simp: card-of-Well-order)
next
assume *: |A| ≤ o |B|
obtain f where embed ( |A| ) ( |B| ) f
using * unfolding ordLeq-def by auto
thus ∃ f. inj-on f A ∧ f ‘ A ≤ B by auto
lemma card-of-ordLeq2:
\[ A \neq \{\} \implies (\exists g. g \cdot B = A) = (|A| \leq_o |B|) \]
using card-of-ordLeq[of A B] inj-on-iff-surj[of A B] by auto

lemma card-of-ordLess:
\[ (\neg(\exists f. \text{inj-on } f \cdot A \leq B)) = (|A| <_o |B|) \]
proof
  have \[ (\neg(\exists f. \text{inj-on } f \cdot A \leq B)) = (\neg |A| \leq_o |B|) \]
  using card-of-ordLeq by blast
  also have \[ \ldots = (|B| <_o |A|) \]

  not-ordLeq-iff-ordLess by blast
  finally show \[ \text{thesis} \].
qed

lemma card-of-ordLess2:
\[ B \neq \{\} \implies (\neg(\exists f. f \cdot A = B)) = (|A| <_o |B|) \]
using card-of-ordLess[of B A] inj-on-iff-surj[of B A] by auto

lemma card-of-ordIsoI:
assumes bij-betw f A B
shows \[ |A|=o|B| \]
using assms unfolding card-of-ordIso[symmetric] by auto

lemma card-of-ordLeqI:
assumes inj-on f A and \[ \forall a. a \in A \implies f a \in B \]
shows \[ |A| \leq_o |B| \]
using assms unfolding card-of-ordLeq[symmetric] by auto

lemma card-of-unique:
\[ \text{card-order-on } A \text{ r} \implies r =o|A| \]
by (simp only: card-order-on-ordIso card-of-card-order-on)

lemma card-of-mono1:
\[ A \leq B \implies |A| \leq_o |B| \]
using inj-on-id[of A] card-of-ordLeq[of A B] by fastforce

lemma card-of-mono2:
assumes \[ r \leq_o r' \]
shows \[ |\text{Field } r| \leq_o |\text{Field } r'| \]
proof
  obtain f where \[ f. \text{well-order-on } (\text{Field } r) \land \text{well-order-on } (\text{Field } r) \land \text{embed } r r' \]
  using assms unfolding ordLeq-def
  by (auto simp add: well-order-on-Well-order)
  hence inj-on f (\text{Field } r) \land f \cdot (\text{Field } r) \leq \text{Field } r'
  by (auto simp add: embed-inj-on embed-Field)
thus \(|\text{Field } r| \leq_o |\text{Field } r'|\) using card-of-ordLeq by blast
qed

lemma card-of-cong: \(r = o r' \implies |\text{Field } r| = o |\text{Field } r'|\)
by (simp add: ordIso-iff-ordLeq card-of-mono2)

lemma card-of-Field-ordLess: Well-order \(r \implies |\text{Field } r| \leq_o r\)
using card-of-least card-of-well-order-on well-order-on-Well-order by blast

lemma card-of-Field-ordIso:
assumes Card-order \(r\)
shows \(|\text{Field } r| = o r\)
proof −
  have card-order-on \((\text{Field } r) r\)
  using assms card-order-on-Card-order by blast
  moreover have card-order-on \((\text{Field } r) |\text{Field } r|\)
  using card-of-card-order-on by blast
  ultimately show (?thesis using card-order-on-ordIso by blast)
qed

lemma Card-order-iff-ordIso-card-of:
Card-order \(r\) = \((r = o |\text{Field } r|)\)
using ordIso-card-of-imp-Card-order card-of-Field-ordIso ordIso-symmetric by blast

lemma Card-order-iff-ordLeq-card-of:
Card-order \(r\) = \((r \leq_o |\text{Field } r|)\)
proof −
  have Card-order \(r\) = \((r = o |\text{Field } r|)\)
  unfolding Card-order-iff-ordIso-card-of by simp
  also have ...
    = \((r \leq_o |\text{Field } r|) \land |\text{Field } r| \leq_o r\)
  unfolding ordIso-iff-ordLeq by simp
  also have ...
    = \((r \leq_o |\text{Field } r|)\)
  using card-of-Field-ordLess
  by (auto simp: card-of-Field-ordLess ordLeq-Well-order-simp)
  finally show (?thesis).
qed

lemma Card-order-iff-Restr-underS:
assumes Well-order \(r\)
shows Card-order \(r\) = (\(\forall a \in \text{Field } r.\) Restr \(r\) \((\underS \ r \ a) < o |\text{Field } r|)\)
using assms unfolding Card-order-iff-ordLeq-card-of
using ordLeq-iff-ordLess-Restr card-of-Well-order by blast

lemma card-of-underS:
assumes \(r: \text{Card-order } r\) and \(a: a \in \text{Field } r\)
shows \(|\underS \ r \ a| < o r\)
proof −
  let \(?A = \underS \ r \ a\) let \(?r' = \text{Restr } r \ ?A\)
  have 1: Well-order \(r\)
using r unfolding card-order-on-def by simp
have Well-order ?r' using 1 Well-order-Restr by auto
moreover have card-order-on (Field ?r') |Field ?r'
using card-of-card-order-on .
ultimately have |Field ?r'| ≤o ?r'
unfolding card-order-on-def by simp
moreover have Field ?r' = ?A
using 1 wo-rel.underS-ofilter Field-Restr-ofilter
ultimately have |?A| ≤o ?r' by simp
also have ?r' <o |Field r| using 1 a r Card-order-iff-Restr-underS by blast
also have |Field r| =o r
using r ordIso-symmetric unfolding Card-order-iffordIso-card-of by auto
finally show ?thesis .
qed

lemma ordLess-Field:
assumes r <o r'
shows |Field r| <o r'
proof –
  have well-order-on (Field r) r using assms unfolding ordLess-def
  by (auto simp add: well-order-on-Well-order)
  hence |Field r| ≤o r using card-of-least by blast
  thus ?thesis using assms ordLeq-ordLess-trans by blast
qed

lemma internalize-card-of-ordLeq:
( |A| ≤o r ) = (∃ B ≤ Field r. |A| =o |B| ∧ |B| ≤o r )
proof
  assume |A| ≤o r
  then obtain p where 1: Field p ≤ Field r ∧ |A| =o p ∧ p ≤o r
  using internalize-ordLeq[of |A| r] by blast
  hence |Field p| =o p using card-of-Field-ordIso by blast
  hence |A| =o |Field p| ∧ |Field p| ≤o r
  using 1 ordIso-equivalence ordIso-ordLeq-trans by blast
  thus ∃ B ≤ Field r. |A| =o |B| ∧ |B| ≤o r using 1 by blast
next
  assume ∃ B ≤ Field r. |A| =o |B| ∧ |B| ≤o r
  thus |A| ≤o r using ordIso-ordLeq-trans by blast
qed

lemma internalize-card-of-ordLeq2:
( |A| ≤o |C| ) = (∃ B ≤ C. |A| =o |B| ∧ |B| ≤o |C| )
28.3 Cardinals versus set operations on arbitrary sets

Here we embark in a long journey of simple results showing that the standard set-theoretic operations are well-behaved w.r.t. the notion of cardinal – essentially, this means that they preserve the "cardinal identity" =o and are monotonic w.r.t. ≤o.

**Lemma card-of-empty:** \(|\{\}| \leq_o |A|

**Using** card-of-ordLeq inj-on-id by blast

**Lemma card-of-empty1:**
assumes Well-order r ∨ Card-order r
shows \(|\{\}| \leq_o r

**Proof:**
  have Well-order r using assms unfolding card-order-on-def by auto
  hence |Field r| \leq_o r
  using assms card-of-Field-ordLess by blast
  moreover have \(|\{\}| \leq_o |Field r| by (simp add: card-of-empty)
  ultimately show ?thesis using ordLeq-transitive by blast
  qed

**Corollary Card-order-empty:**
Card-order r \implies |\{\}| \leq_o r by (simp add: card-of-empty1)

**Lemma card-of-empty2:**
assumes LEQ: |A| =o |\{\}|
shows A = \{\}

**Using** assms card-of-ordIso[of A] bij-betw-empty2 by blast

**Lemma card-of-empty3:**
assumes LEQ: |A| \leq_o |\{\}|
shows A = \{\}

**Using** assms
by (simp add: ordIso-iff-ordLeq card-of-empty1 card-of-empty2
ordLeq-Well-order-simp)

**Lemma card-of-empty-ordIso:**
|\{\}| ::'a set = o |\{\}| ::'b set|
**Using** card-of-ordIso unfolding bij-betw-def inj-on-def by blast

**Lemma card-of-image:**
|f ‘ A| \leq_o |A|

**Proof:**
cases A = \{\}, simp add: card-of-empty)
  assume A ≠ \{\}
  hence f ‘ A ≠ \{\} by auto
  thus |f ‘ A| \leq_o |A|
  using card-of-ordLeq2[of f ‘ A A] by auto
  qed

**Lemma surj-imp-ordLeq:**
assumes $B \subseteq f \cdot A$
shows $|B| \leq o |A|$
proof
  have $|B| \leq o |f \cdot A|$ using assms card-of-mono1 by auto
  thus ?thesis using card-of-image ordLeq-transitive by blast
qed

lemma card-of-singl-ordLeq:
assumes $A \neq \{\}$
shows $|\{b\}| \leq o |A|$
proof
  obtain $a$ where $\ast$: $a \in A$ using assms by auto
  let $?h = \lambda b': b$, if $b' = b$ then $a$ else undefined
  have inj-on $?h \cdot \{b\} \wedge ?h \cdot \{b\} \leq A$
  using $\ast$ unfolding inj-on-def by auto
  thus ?thesis unfolding card-of-ordLeq[symmetric] by (intro exI)
qed

corollary Card-order-singl-ordLeq:
$[\{b\}] \leq o r \Rightarrow |\{b\}| \leq o r$
using card-of-singl-ordLeq[of Field $r$ $b$]
  card-of-Field-ordIso[of $r$] ordLeq-ordIso-trans by blast

lemma card-of-Pow: $|A| < o |Pow A|$
using card-of-ordLess2[of Pow $A$ $A$] Cantors-paradox[of $A$]
Pow-not-empty[of $A$] by auto

corollary Card-order-Pow:
Card-order $r \Rightarrow r < o |Pow(Field r)|$
using card-of-Pow card-of-Field-ordIso ordIso-ordLess-trans ordIso-symmetric by blast

lemma card-of-Plus1: $|A| \leq o |A <+> B|$
proof
  have $\text{Inl} \cdot A \leq A <+> B$ by auto
qed

corollary Card-order-Plus1:
Card-order $r \Rightarrow r \leq o |(Field r) <+> B|$
using card-of-Plus1 card-of-Field-ordIso ordIso-ordLeq-trans ordIso-symmetric by blast

lemma card-of-Plus2: $|B| \leq o |A <+> B|$
proof
  have $\text{Inr} \cdot B \leq A <+> B$ by auto
  thus ?thesis using inj-Inr[of $B$] card-of-ordLeq by blast
qed
corollary Card-order-Plus2:
Card-order \( r \implies r \leq o \mid A <+> \{\}\)  
using card-of-Plus2 card-of-Field-ordIso ordIso-ordLeq-trans ordIso-symmetric by blast

lemma card-of-Plus-empty1: \( |A| = o \mid A <+> \{\}\)  
proof – 
  have bij-betw Inl A (\( \{\}\) <+> \( \{\}\)) unfolding bij-betw-def inj-on-def by auto  
  thus \(?thesis\) using card-of-ordIso by auto  
qed

lemma card-of-Plus-empty2: \( |A| = o \mid {} <+> A\)  
proof – 
  have bij-betw Inr A (\( \{\}\) <+> \( A\)) unfolding bij-betw-def inj-on-def by auto  
  thus \(?thesis\) using card-of-ordIso by auto  
qed

lemma card-of-Plus-commute: \( |A <+> B| = o \mid B <+> A\)  
proof – 
  let \( ?f = \lambda (c::'a + 'b). \text{ case } c \text{ of } \text{Inl } a \Rightarrow \text{Inr } a\)  
  | \text{Inr } b \Rightarrow \text{Inl } b\)  
  have bij-betw ?f (\( A <+> B\)) (\( B <+> A\)) unfolding bij-betw-def inj-on-def by force  
  thus \(?thesis\) using card-of-ordIso by blast  
qed

lemma card-of-Plus-assoc:  
fixes A :: 'a set and B :: 'b set and C :: 'c set  
shows \( |(A <+> B) <+> C| = o \mid A <+> B <+> C\)  
proof – 
  define f :: ('a + 'b) + 'c ⇒ 'a + 'b + 'c  
  where \[abs-def\]: \( f k = \)  
  (case k of \( \text{Inl } ab \Rightarrow \)  
    \( \text{case } ab \text{ of } \text{Inl } a \Rightarrow \text{Inl } a\)  
    | \text{Inr } b \Rightarrow \text{Inr } (\text{Inl } b)\)  
    | \text{Inr } c \Rightarrow \text{Inr } (\text{Inr } c)\)  
  for k  
  have A <+> B <+> C \( \subseteq \) f \( ((A <+> B) <+> C)\)  
proof  
  fix x assume x: \( x \in A <+> B <+> C\)  
  show x \( \in f \) \( ((A <+> B) <+> C)\)  
proof(cases x)  
  case (Inl a)  
  hence a \( \in A \) x = f (Inl (Inl a))  
  using x unfolding f-def by auto  
  thus \(?thesis\) by auto  
next
case (Inr bc) note 1 = Inr show ?thesis
proof (cases bc)
  case (Inl b)
  hence \( b \in B \) \( x = f \) (Inl (Inr b))
  using \( x 1 \) unfolding \( f\)-def by auto
  thus ?thesis by auto
next
  case (Inr c)
  hence \( c \in C \) \( x = f \) (Inr c)
  using \( x 1 \) unfolding \( f\)-def by auto
  thus ?thesis by auto
qed

hence bij-betw \( f \) ((\( A \) <\>\( B \) ) <\>\( C \) ) (\( A \) <\>\( B \) <\>\( C \) )
  unfolding bij-betw-def inj-on-def \( f\)-def by fastforce
thus ?thesis using card-of-ordIso by blast
qed

lemma card-of-Plus-mono1:
assumes \( |A| \leq o \) \( |B| \)
shows \( |A <\> C| \leq o \) \( |B <\> C| \)
proof
  obtain \( f \) where \( 1 \): inj-on \( f \) \( A \) \( \land \) \( f \cdot A \leq B \)
  using assms card-of-ordLeq[of \( A \) ] by fastforce
  obtain \( g \) where \( g\)-def:
    \( g = (\lambda d. \text{case } d \text{ of } \text{Inl } a \Rightarrow \text{Inl}(f a) \mid \text{Inr } (c::'c) \Rightarrow \text{Inr } c) \) by blast
  have inj-on \( g \) \( (A <\> C) \) \( \land \) \( g \cdot (A <\> C) \leq (B <\> C) \)
  proof
    \{ fix \( d1 \) and \( d2 \) assume \( d1 \in A <\> C \) \( \land \) \( d2 \in A <\> C \) \( \land \) \( g d1 = g d2 \)
      hence \( d1 = d2 \) using \( 1 \) unfolding inj-on-def \( g\)-def by force \}
    moreover
    \{ fix \( d \) assume \( d \in A <\> C \)
      hence \( g d \in B <\> C \) using \( 1 \)
      by (cases \( d \) ) (auto simp add: \( g\)-def) \}
    ultimately show ?thesis unfolding inj-on-def by auto
  qed
thus ?thesis using card-of-ordLeq by blast
qed

corollary ordLeq-Plus-mono1:
assumes \( r \leq o \) \( r' \)
shows \( |(\text{Field } r) <\> C| \leq o \) \( |(\text{Field } r') <\> C| \)
using assms card-of-mono2 card-of-Plus-mono1 by blast

lemma card-of-Plus-mono2:
assumes $|A| \leq o |B|$
shows $|C <+> A| \leq o |C <+> B|$
using assms card-of-Plus-mono1[of $A B C$]
by blast
corollary ordLeq-Plus-mono2:
assumes $r \leq o r'$
shows $|A <+> (\text{Field } r)| \leq o |A <+> (\text{Field } r')|$
using assms card-of-mono2 card-of-Plus-mono2 by blast

lemma card-of-Plus-mono:
assumes $|A| \leq o |B|$ and $|C| \leq o |D|$
shows $|A <+> C| \leq o |B <+> D|$
ordLeq-transitive[of $|A <+> C|$] by blast
corollary ordLeq-Plus-mono:
assumes $r \leq o r'$ and $p \leq o p'$
shows $|(\text{Field } r) <+> (\text{Field } p)| \leq o |(\text{Field } r') <+> (\text{Field } p')|$

lemma card-of-Plus-cong1:
assumes $|A| = o |B|$
shows $|A <+> C| = o |B <+> C|$
using assms by (simp add: ordIso-iff-ordLeq card-of-Plus-mono1)
corollary ordIso-Plus-cong1:
assumes $r = o r'$
shows $|(\text{Field } r) <+> C| = o |(\text{Field } r') <+> C|$
using assms card-of-cong card-of-Plus-cong1 by blast

lemma card-of-Plus-cong2:
assumes $|A| = o |B|$
shows $|C <+> A| = o |C <+> B|$
using assms by (simp add: ordIso-iff-ordLeq card-of-Plus-mono2)
corollary ordIso-Plus-cong2:
assumes $r = o r'$
shows $|A <+> (\text{Field } r)| = o |A <+> (\text{Field } r')|$
using assms card-of-cong card-of-Plus-cong2 by blast

lemma card-of-Plus-cong:
assumes $|A| = o |B|$ and $|C| = o |D|$
shows $|A <+> C| = o |B <+> D|$
using assms by (simp add: ordIso-iff-ordLeq card-of-Plus-mono)
corollary ordIso-Plus-cong:
assumes $r = o r'$ and $p = o p'$
shows $|\text{Field } r| < + (\text{Field } p) = o (\text{Field } r') < + (\text{Field } p')$

lemma card-of-Un-Plus-ordLeq:
$|A \cup B| \leq o |A < + B|$
proof
  let $\lambda = \text{if } c \in A \text{ then Inl c else Inr c}$
  have inj-on $\lambda (A \cup B) \land \lambda (A \cup B) \leq A < + B$
  unfolding inj-on-def by auto
  thus $?thesis$ using card-of-ordLeq by blast
qed

lemma card-of-Times1:
assumes $A \neq \{\}$
shows $|B| \leq o |B \times A|$
proof(cases $B = \{\}$, simp add: card-of-empty)
  assume $*: B \neq \{\}$
  have $\text{fst} \ '(B \times A) = B$ using assms by auto
  thus $?thesis$ using card-of-ordLeq[of $B$ $B \times A$] by blast
qed

lemma card-of-Times-commute: $|A \times B| = o |B \times A|$
proof
  let $\lambda = \text{if } (a, b' : 'b) \text{ then } (b, a)$
  have bij-betw $\lambda (A \times B) (B \times A)$
  unfolding bij-betw-def inj-on-def by auto
  thus $?thesis$ using card-of-ordIso by blast
qed

lemma card-of-Times2:
assumes $A \neq \{\}$ shows $|B| \leq o |A \times B|$
ordLeq-ordIso-trans by blast

corollary Card-order-Times1:
$\{\text{Card-order } r; B \neq \{\}\} \Longrightarrow r \leq o (\text{Field } r) \times B$
using card-of-Times1[of $B$] card-of-Field-ordIso
ordIso-ordLeq-trans ordIso-symmetric by blast

corollary Card-order-Times2:
$\{\text{Card-order } r; A \neq \{\}\} \Longrightarrow r \leq o A \times (\text{Field } r)$
using card-of-Times2[of $A$] card-of-Field-ordIso
ordIso-ordLeq-trans ordIso-symmetric by blast

lemma card-of-Times3: $|A| \leq o |A \times A|$
using card-of-Times1[of $A$]
by (cases A = {}, simp add: card-of-empty, blast)

lemma card-of-Plus-Times-bool: \(|A <+> A| = o \|A \times (UNIV::bool set)\|
proof
let \(f = \lambda c::'a + 'a. case c of Inl a \Rightarrow (a,True)
|Inr a \Rightarrow (a,False)\)
have bij-betw \(f (A <+>) (A \times (UNIV::bool set))\)
proof
{fix \(c1\) and \(c2\) assume \(f c1 = f c2\)
hence \(c1 = c2\) by (cases c1; cases c2) auto}
moreover
{fix \(c\) assume \(c \in A <+> A\)
hence \(f c \in A \times (UNIV::bool set)\)
by (cases c) auto}
moreover
{fix \(a bl\) assume \(\ast\): \((a,bl) \in A \times (UNIV::bool set)\)
have \((a,bl) \in f' (A <+> A)\)
proof (cases bl)
assume \(bl\) hence \(f (Inl a) = (a,bl)\) by auto
thus \(\ast\) using \(\ast\) by force
next
assume \(\neg bl\) hence \(f (Inr a) = (a,bl)\) by auto
thus \(\ast\) using \(\ast\) by force
qed
}
ultimately show \(\ast\) unfolding bij-betw-def inj-on-def by auto
qed
thus \(\ast\) using card-of-ordIso by blast
qed

lemma card-of-Times-mono1:
assumes \(|A| \leq o \|B\|
shows \(|A \times C| \leq o \|B \times C|\)
proof
obtain \(f\) where \(f\): inj-on \(f A \wedge f' A \leq B\)
using assms card-of-ordLeq[of A] by fastforce
obtain \(g\) where \(g\)-def:
g = (λ(a,c::'c). (f a,c)) by blast
have inj-on \(g (A \times C) \cup g' (A \times C) \leq (B \times C)\)
using \(f\) unfolding inj-on-def using \(g\)-def by auto
thus \(\ast\) using card-of-ordLeq by blast
qed

corollary ordLeq-Times-mono1:
assumes \(r \leq o r'\)
shows \(|(Field r) \times C| \leq o |(Field r') \times C|\)
THEORY “BNF-Cardinal-Order-Relation”

using assms card-of-mono2 card-of-Times-mono1 by blast

lemma card-of-Times-mono2:
assumes \(|A| \leq o \ |B|
shows \(|C \times A| \leq o \ |C \times B|
using assms card-of-Times-mono1[|A| \leq o \ |B|]
card-of-Times-commute[|C| \leq o \ |B|]
card-of-Times-commute[|C| \leq o \ |B|]
ordLeq-ordIso-trans[|C| \leq o \ |B|]
by blast

corollary ordLeq-Times-mono2:
assumes \(r \leq o \ r'
shows \(|A \times (\text{Field } r)| \leq o \ |A \times (\text{Field } r')|
using assms card-of-mono2 card-of-Times-mono1 by blast

lemma card-of-Sigma-mono1:
assumes \(\forall i \in I. \ |A_i| \leq o \ |B_i|
shows \(|\bigwedge i \in I. A_i| \leq o \ |\bigwedge i \in I. A_i|
proof
have \(\forall i \in I. \exists f. \text{inj-on } f (A_i) \land f' (A_i) \leq B_i\)
using assms by (auto simp add: card-of-ordLeq)
with \(\lambda i f. \; i \in I \rightarrow \text{inj-on } f (A_i) \land f' (A_i) \leq B_i\)
obtain \(F\) where \(I: \forall i \in I. \text{inj-on } (F i) (A_i) \land (F i)' (A_i) \leq B_i\)
by atomize-elim (auto intro: choice)
using \(1\) unfolding inj-on-def using \(g\) def by blast
thus \(?thesis\) using card-of-ordLeq by blast
qed

lemma card-of-UNION-Sigma:
\(\bigcup i \in I. A_i| \leq o \ |\bigwedge i \in I. A_i|
using Ex-inj-on-UNION-Sigma of A I card-of-ordLeq by blast

lemma card-of-bool:
assumes \(a_1 \neq a_2\)
shows \(|\text{UNIV::bool set}| = o \ |\{a_1,a_2}\|
proof
let \(?f = \lambda bl. \text{case } bl \text{ of True } \Rightarrow a_1 \mid False \Rightarrow a_2\)
have bij-betw \(?f\) UNIV \(?\{a_1,a_2\}\)
by blast
proof
{fix bl1 and bl2 assume \(?f\) bl1 = \(?f\) bl2
hence bl1 = bl2 using assms by (cases bl1, cases bl2) auto}
moreover
{fix bl have \(?f\) bl \in \(?\{a_1,a_2\}\) by (cases bl) auto}
moreover
{fix a assume \(*: a \in \{a_1,a_2\}\)
have \( a \in ?f' \ UNIV \)

proof (cases \( a = a1 \))
  assume \( a = a1 \)
  hence \( ?f' \ True = a \) by auto  \thus ?thesis by blast

next
  assume \( a \neq a1 \) hence \( a = a2 \) using * by auto
  hence \( ?f' \ False = a \) by auto  \thus ?thesis by blast

qed

}\)

ultimately show ?thesis unfolding bij-betw_def inj-on_def by blast

qed

thus ?thesis using card-of-ordIso by blast

qed

lemma card-of-Plus-Times-aux:
assumes A2: \( a1 \neq a2 \land \{a1,a2\} \leq A \) and
  LEQ: \( |A| \leq |B| \)
shows \( |A <+> B| \leq o \ |A \times B| \)
proof –
  have 1: \( |UNIV::\text{bool set}| \leq o \ |A| \)
    using A2 card-of-mono1[af \( \{a1,a2\} \)] card-of-bool[af \( a1 a2 \)]
    ordIso-ordLeq-trans[of \( |UNIV::\text{bool set}| \)] by blast
  have \( |A <+> B| \leq o \ |B <+> B| \)
    using LEQ card-of-Plus-mono1 by blast
  moreover have \( |B <+> B| = o \ |B \times (UNIV::\text{bool set})| \)
    using 1 by (simp add: card-of-Times-mono2)
  moreover have \( |B \times (UNIV::\text{bool set})| \leq o \ |B \times A| \)
    using 1 by (simp add: card-of-Times-commute)
  moreover have \( |B \times A| = o \ |A \times B| \)
    using card-of-Times-commute by blast
  ultimately show \( |A <+> B| \leq o \ |A \times B| \)
    using ordLeq-ordIso-trans[of \( |A <+> B| \) \( |B <+> B| \) \( |B \times (UNIV::\text{bool set})| \) \( |B \times A| \) \( |B \times A| \) \( |A \times B| \) \( |A \times B| \)]
    ordLeq-transitive[of \( |A <+> B| \) \( |B \times (UNIV::\text{bool set})| \) \( |B \times A| \) \( |A \times B| \) \( |A \times B| \) \( |A \times B| \)]
    by blast

qed

lemma card-of-Plus-Times:
assumes A2: \( a1 \neq a2 \land \{a1,a2\} \leq A \) and
  B2: \( b1 \neq b2 \land \{b1,b2\} \leq B \)
shows \( |A <+> B| \leq o \ |A \times B| \)
proof –
  \{ assume \( |A| \leq o \ |B| \)
    hence \( ?thesis \) using assms by (auto simp add: card-of-Plus-Times-aux)
  \}
  moreover
  \{ assume \( |B| \leq o \ |A| \)
    hence \( |B <+> A| \leq o \ |B \times A| \)
  \}
using assms by (auto simp add: card-of-Plus-Times-aux)
hence ?thesis
using card-of-Plus-commute card-of-Times-commute
ordIso-ordLeq-trans ordLeq-ordIso-trans by blast
}
ultimately show ?thesis
ordLeq-total[of |A|] by blast
qed

lemma card-of-Times-Plus-distrib:
|A × (B <+> C)| =o |A × B <+> A × C| (is |?RHS| =o |?LHS|)

proof —
let ?f = λ(a, bc). case bc of Inl b ⇒ Inl (a, b) | Inr c ⇒ Inr (a, c)
have bij-betw ?f |?RHS| LHS unfolding bij-betw-def inj-on-def by force
thus ?thesis using card-of-ordIso by blast

qed

lemma card-of-ordLeq-finite:
assumes |A| ≤ o |B| and finite B
shows finite A
using assms unfolding ordLeq-def
using embed-inj-on[of |A|] [of |B|] embed-Field[of |A|] [of |B|]

lemma card-of-ordLeq-infinite:
assumes |A| ≤ o |B| and ¬ finite A
shows ¬ finite B
using assms card-of-ordLeq-finite by auto

lemma card-of-ordIso-finite:
assumes |A| = o |B|
shows finite A = finite B
using assms unfolding ordIso-def iso-def[abs-def]
by (auto simp: bij-betw-finite Field-card-of)

lemma card-of-ordIso-finite-Field:
assumes Card-order r and r =o |A|
shows finite(Field r) = finite A
using assms card-of-Field-ordIso card-of-ordIso-finite ordIso-equivalence by blast

28.4 Cardinals versus set operations involving infinite sets

Here we show that, for infinite sets, most set-theoretic constructions do not
increase the cardinality. The cornerstone for this is theorem Card-order-Times-same-infinite,
which states that self-product does not increase cardinality – the proof of
this fact adapts a standard set-theoretic argument, as presented, e.g., in the
proof of theorem 1.5.11 at page 47 in [4]. Then everything else follows fairly
 THEORY "BNF-Cardinal-Order-Relation"  

 easily.  

 lemma infinite-iff-card-of-nat:  
\[ \neg \text{finite } A \iff (|\text{UNIV::nat set}| \leq o |A|) \]  
 unfolding infinite-iff-countable-subset card-of-ordLeq ..  

 The next two results correspond to the ZF fact that all infinite cardinals are limit ordinals:  

 lemma Card-order-infinite-not-under:  
 assumes \(\text{CARD: Card-order } r \text{ and } \neg \text{finite } (\text{Field } r)\)  
 shows \(\neg (\exists a. \text{Field } r = \text{under } r a)\)  
 proof(auto)  
 have 0: \(\text{Well-order } r \land \text{wo-rel } r \land \text{Refl } r\)  
 using CARD unfolding wo-rel-def card-order-on-def order-on-defs by auto  
 fix a assume *: Field r = under r a  
 show False  
 proof(cases a ∈ Field r)  
 assume Case1: a /∈ Field r  
 hence under r a = {} unfolding Field-def under-def by auto  
 thus False using INF * by auto  
 next  
 let ?r' = Restr r (underS r a)  
 assume Case2: a ∈ Field r  
 hence 1: under r a = underS r a ∪ {a} ∧ a /∈ underS r a  
 using 0 Refl-under-underS[of r a] underS-notIn[of a r] by blast  
 have 2: wo-rel.ofilter r (underS r a) ∧ underS r a < Field r  
 using 0 wo-rel.underS-ofilter * 1 Case2 by fast  
 hence ?r' < o r using 0 using ofilter-ordLess by blast  
 moreover  
 have Field ?r' = underS r a ∧ Well-order ?r'  
 using 2 0 Field-Restr-ofilter[of r] Well-order-Restr[of r] by blast  
 ultimately have |underS r a| < o r using ordLess-Field[of ?r'] by auto  
 moreover have |under r a| = o r using * CARD card-of-Field-ordIso[of r] by auto  
 ultimately have |underS r a| < o |under r a|  
 using ordIso-symmetric ordLess-ordIso-trans by blast  
 moreover  
 \{have \(\exists f. \text{bij-betw } f (\text{under } r a) (\text{underS } r a)\)  
 using infinite-imp-bij-betw[of Field r a] INF * 1 by auto  
 hence |under r a| = o |underS r a| using card-of-ordIso by blast  
 \}  
 ultimately show False using not-ordLess-ordIso ordIso-symmetric by blast  
 qed  
 qed  

 lemma infinite-Card-order-limit:  
 assumes r: \(\text{Card-order } r \text{ and } \neg \text{finite } (\text{Field } r)\)  
 and a: a ∈ Field r  
 shows \(\exists b \in \text{Field } r. a \neq b \land (a,b) \in r\)  
 proof --
have Field $r \neq$ under $r a$
using assms Card-order-infinite-not-under by blast
moreover have under $r a \leq$ Field $r$
using under-Field.
ultimately have under $r a <$ Field $r$ by blast
then obtain $b$ where 1: $b \in$ Field $r \land \neg (b,a) \in r$
unfolding under-def by blast
moreover have $b:a: b \neq a$
using 1 $r$ unfolding card-order-on-def well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def refl-on-def by auto
ultimately have $(a,b) \in r$
using a $r$ unfolding card-order-on-def well-order-on-def linear-order-on-def total-on-def by blast
thus $\neg$thesis using 1 ba by auto
qed

theorem Card-order-Times-same-infinite:
assumes CO: Card-order $r$ and INF: $\neg$finite(Field $r$)
shows $\mid$Field $r \times$ Field $r\mid \leq o r$
proof –
obtain phi where phi-def:
$phi = (\lambda r::'a rel. \text{Card-order } r \land \neg$finite(Field $r$) \land
$\neg \mid$Field $r \times$ Field $r\mid \leq o r )$ by blast
have temp1: $\forall r. phi r \rightarrow$ Well-order $r$
unfolding phi-def card-order-on-def by auto
have Ft: $\neg(\exists r. phi r)$
proof
assume $\exists r. phi r$
hence $\{r. phi r\} \neq \{} \land \{r. phi r\} \leq \{r. \text{Well-order } r\}$
using temp1 by auto
then obtain $r$ where 1: phi $r$ and 2: $\forall r'. phi r' \rightarrow r \leq o r'$ and
3: Card-order $r \land$ Well-order $r$
using exists-minim-Well-order[of $\{r. phi r\}$] temp1 phi-def by blast
let $?A =$ Field $r$ let $?r' = bsqr $r$
have 4: Well-order $?r' \land$ Field $?r' = ?A \times ?A \land |?A| = o $r$
using 3 bsqr-Well-order Field-bsqr card-of-Field-ordIso by blast
have 5: Card-order $?A \times ?A\mid \land$ Well-order $?A \times ?A\mid$
using card-of-Card-order card-of-Well-order by blast
have $r < o \mid ?A \times ?A\mid$
using 1 3 5 ordLess-or-ordLeg unfolding phi-def by blast
moreover have $\mid ?A \times ?A\mid \leq o \mid ?r'\mid$
using card-of-least[of $?A \times ?A$] 4 by auto
ultimately have $r < o \mid ?r'\mid$ using ordLess-ordLeg-trans by auto
then obtain $f$ where 6: embed $r \mid ?r'\mid f$ and 7: $\neg$bij-betw $f ?A ( ?A \times ?A)$
unfolding ordLess-def embedS-def[abs-def]
by (auto simp add: Field-bsqr)
let $?B = f \cdot ?A$
have $|?A| = o |?B|$
using 3 6 embed-inj-on inj-on-imp-bij-tf card-of-ordIso by blast
hence 8: \( r = o |?B| \) using 4 ordIso-transitive ordIso-symmetric by blast

have wo-rel.ofilter ?r' ?B using 6 embed-Field-ofilter 3 4 by blast
hence wo-rel.ofilter ?r' ?B \& ?B \neq ?A \times ?A \& \neq Field ?r'
using 7 unfolding bij-tf-def using 6 3 embed-inj-on 4 by auto
have \( \neg (\exists a. \text{Field } r = \text{under } r a) \)
using 1 unfolding phi-def using Card-order-infinite-not-under[of r] by auto
using temp2 3 bsqr-ofilter[of r ?B] by blast
hence |?B| \leq o |A1 \times A1| using card-of-iso-transitive by blast
let \(?r1 = \text{Restr } r A1\)
have \( ?r1 < o r \) using temp3 ofilter-ordLess 3 by blast
moreover
\{ have well-order-on A1 \( ?r1 \) using 3 temp3 well-order-on-Restr by blast
hence |A1| \leq o ?r1 using 3 Well-order-Restr card-of-least by blast \}
ultimately have 11: |A1| < o r using ordLeq-ordLess-trans by blast

have \( \neg \text{finite } (\text{Field } r) \) using 1 unfolding phi-def by simp
hence \( \neg \text{finite } ?B \) using 8 3 card-of-iso-finite-Field[of r ?B] by blast
hence \( \neg \text{finite } A1 \) using 9 finite-cartesian-product finite-subset by blast
moreover have temp3: Field |A1| = A1 \& Well-order |A1| \& Card-order |A1|
by (simp add: Field-card-of)
moreover have \( \neg r \leq o | A1 | \)
using temp4 11 3 using not-ordLeq-iff-ordLess by blast
ultimately have \( \neg \text{finite } (\text{Field } |A1|) \& \text{Card-order } |A1| \& \neg r \leq o | A1 | \)
by (simp add: card-of-cardorder-on)
hence |Field |A1| \times Field |A1| | \leq o | A1|
using 2 unfolding phi-def by blast
hence |A1 \times A1| \leq o |A1| using temp4 by auto
hence r \leq o |A1| using 10 ordLeq-transitive by blast
thus False using 11 not-ordLess-ordLeq by auto
qed
thus \(?thesis using asms unfolding phi-def by blast qed quod

corollary card-of-Times-same-infinite:
assumes \( \neg \text{finite } A \)
shows |A \times A| = o |A|
proof
let \(?r = |A|
have Field ?r = A \& \text{Card-order } ?r
using Field-card-of card-of-Card-order[of A] by fastforce

hence $|A \times A| \leq \omega |A|

using Card-order-Times-same-infinite[of ?r] assms by auto

thus ?thesis using card-of-Times3 ordIso-iff-ordLeq by blast

qed

lemma card-of-Times-infinite:

assumes INF: ¬finite A and NE: B ≠ {} and LEQ: |B| \leq \omega |A|

shows $|A \times B| = \omega |A| \land |B \times A| = \omega |A|$

proof –

have $|A| \leq \omega |A \times B| \land |A| \leq \omega |B \times A|$

using assms by (simp add: card-of-Times1 card-of-Times2)

moreover

{ have $|A \times B| \leq \omega |A \times A| \land |B \times A| \leq \omega |A \times A|$ using LEQ card-of-Times-mono1 card-of-Times-mono2 by blast

moreover have $|A \times A| = \omega |A|$ using INF card-of-Times-same-infinite by blast

ultimately have $|A \times B| \leq \omega |A| \land |B \times A| \leq \omega |A|$ using INF card-of-Times-same-infinite by blast

ultimately show ?thesis using (simp add: ordIso-iff-ordLeq)

qed

corollary Card-order-Times-infinite:

assumes INF: ¬finite(Field r) and CARD: Card-order r and

NE: Field p ≠ {} and LEQ-I: |I| \leq \omega |B|

and LEQ: \forall i \in I. |A i| \leq \omega |B|

shows $|\SIGMA i : I. A i| \leq \omega |B|

proof –

have $|\SIGMA i : I. A i| \leq \omega |\ SIGMA i : I. A i| \land |\SIGMA i : I. A i| \leq \omega |\ SIGMA i : I. A i|$

using assms by (simp add: card-of-Times-infinite card-of-mono2)

thus ?thesis using assms of r ordIso-transitive[of |\ SIGMA i : I. A i|]

ordIso-transitive[of - |\ SIGMA i : I. A i| by blast

qed

lemma card-of-Sigma-ordLeq-infinite:

assumes INF: ¬finite B and

LEQ-I: |I| \leq \omega |B| and LEQ: \forall i \in I. |A i| \leq \omega |B|

shows $|\ SIGMA i : I. A i| \leq \omega |I \times B|

proof(cases I = {}, simp add: card-of-empty)

assume *; I ≠ {}

have $|\ SIGMA i : I. A i| \leq \omega |I \times B|$

using card-of-Sigma-mono1[of LEQ] by blast

moreover have $|I \times B| = \omega |B|$

using INF * LEQ-I by (auto simp add: card-of-Times-infinite)

ultimately show ?thesis using ordLeq-ordIso-trans by blast

qed
THEORY "BNF-Cardinal-Order-Relation"

lemma card-of-Sigma-ordLeq-infinite-Field:
assumes INF: ¬finite (Field r) and r: Card-order r and
LEQ-I: |I| ≤o r and LEQ: ∀ i ∈ I. |A i| ≤o r
shows |SIGMA i : I. A i| ≤o r
proof
  let ?B = Field r
have 1: r =o |?B| ∧ |?B| =o r using r card-of-Field-ordIso
ordIso-symmetric by blast
hence |I| ≤o |?B| ∀ i ∈ I. |A i| ≤o |?B|
using LEQ-I LEQ ordLeq-ordIso-trans by blast+
here |SIGMA i : I. A i| ≤o |?B| using INF LEQ
  card-of-Sigma-ordLeq-infinite by blast
thus ?thesis using 1 ordLeq-ordIso-trans by blast
qed

lemma card-of-Times-ordLeq-infinite-Field:
|¬finite (Field r); |A| ≤o r; |B| ≤o r; Card-order r|
⇒ |A × B| ≤o r
by(simp add: card-of-Sigma-ordLeq-infinite-Field)

lemma card-of-Times-infinite-simps:
|¬finite A; B ≠ {}; |B| ≤o |A| ⇒ |A × B| =o |A|
|¬finite A; B ≠ {}; |B| ≤o |A| ⇒ |A| =o |A × B|
|¬finite A; B ≠ {}; |B| ≤o |A| ⇒ |B × A| =o |A|
|¬finite A; B ≠ {}; |B| ≤o |A| ⇒ |A| =o |B × A|
by (auto simp add: card-of-Times-infinite ordIso-symmetric)

lemma card-of-UNION-ordLeq-infinite:
assumes INF: ¬finite B and
LEQ-I: |I| ≤o |B| and LEQ: ∀ i ∈ I. |A i| ≤o |B|
shows |⋃ i ∈ I. A i| ≤o |B|
proof(cases I = {}, simp add: card-of-empty)
  assume *: I ≠ {}
  have |⋃ i ∈ I. A i| ≤o |SIGMA i : I. A i|
  using card-of-UNION-Sigma by blast
  moreover have |SIGMA i : I. A i| ≤o |B|
  using assms card-of-Sigma-ordLeq-infinite by blast
  ultimately show ?thesis using ordLeq-transitive by blast
qed

corollary card-of-UNION-ordLeq-infinite-Field:
assumes INF: ¬finite (Field r) and r: Card-order r and
LEQ-I: |I| ≤o r and LEQ: ∀ i ∈ I. |A i| ≤o r
shows |⋃ i ∈ I. A i| ≤o r
proof
  let ?B = Field r
have 1: r =o |?B| ∧ |?B| =o r using r card-of-Field-ordIso
ordIso-symmetric by blast
hence |I| ≤o |?B| ∀ i ∈ I. |A i| ≤o |?B|
THEORY "BNF-Cardinal-Order-Relation"

using LEQ-1 LEQ ordLeq-ordIso-trans by blast
hence \( \bigcup_{i \in I} A 
\) for all \( B \) using INF LEQ
card-of-UNION-ordLeq-infinite by blast
thus \( ? \)thesis using \( I \) ordLeq-ordIso-trans by blast
qed

lemma card-of-Plus-infinite1:
assumes INF: \( \neg \)finite \( A \) and LEQ: \( \vert B \vert \leq o \vert A \vert \)
shows \( \vert A \plus B \vert = o \vert A \vert \)
proof(cases \( B = \{} \), simp add: card-of-Plus-empty1 card-of-Plus-empty2 ordIso-symmetric)
let \( ?Inl = \text{Inl} : 'a \Rightarrow 'a + 'b \) let \( ?Inr = \text{Inr} : 'b \Rightarrow 'a + 'b \)
assume \( \ast : B \neq \{} \)
then obtain \( b_1 \) where 1: \( b_1 \in B \) by blast
show \( ? \)thesis
proof(cases \( B = \{ b_1 \} \))
assume Case1: \( B = \{ b_1 \} \)
have 2: \( \text{bij-betw} ?Inl A (\{ ?Inl \ 'a \}) \)
unfolding bij-betw-def inj-on-def by auto
hence 3: \( \neg \)finite \( ( ?Inl \ 'a \) \)
using INF bij-betw-finite[of ?Inl A] by blast
let \( ?A' = ?Inl \ 'a \cup \{ ?Inr b_1 \} \)
obtain g where \( \text{bij-betw} g (\{ ?Inl \ 'a \}) ?A' \)
using 3 infinite-imp-bij-betw[of ?Inl A] by auto
moreover have \( ?A' = A \leftrightarrow B \) using Case1 by blast
ultimately have \( \text{bij-betw} (g \circ ?Inl) A (A \leftrightarrow B) \)
using 2 by (auto simp add: bij-betw-trans)
thus \( ? \)thesis using card-of-ordIso ordIso-symmetric by blast
next
assume Case2: \( B \neq \{ b_1 \} \)
with \( \ast \) obtain \( b_2 \) where 3: \( b_1 \neq b_2 \land \{ b_1, b_2 \} \leq B \) by fastforce
obtain f where \( \text{inj-on} f B \land f \cdot B \leq A \)
using LEQ card-of-ordLeq[of \( B \)] by blast
with \( \ast \) have \( f \cdot b_1 \neq f \cdot b_2 \land \{ f \cdot b_1, f \cdot b_2 \} \leq A \)
unfolding inj-on-def by auto
with \( \ast \) have \( A \leftrightarrow B \leq o \vert A \times B \vert \)
by (auto simp add: card-of-Plus-Times)
moreover have \( \vert A \times B \vert = o \vert A \vert \)
using assms \( \ast \) by (simp add: card-of-Plus-commute card-of-Plus-infinite1 ordIso-equivalence by blast
ultimately have \( A \leftrightarrow B \leq o \vert A \vert \) using ordLeq-ordIso-trans by blast
thus \( ? \)thesis using card-of-Plus1 ordIso-iff-ordLeq by blast
qed

lemma card-of-Plus-infinite2:
assumes INF: \( \neg \)finite \( A \) and LEQ: \( \vert B \vert \leq o \vert A \vert \)
shows \( \vert B \leftrightarrow A \vert = o \vert A \vert \)
using assms card-of-Plus-commute card-of-Plus-infinite1 ordIso-equivalence by blast
lemma card-of-Plus-infinite:
assumes INF: ¬finite A and LEQ: |B| ≤ o |A|
shows |A <+> B| = o |A| ∧ |B <+> A| = o |A|
using assms by (auto simp: card-of-Plus-infinite1 card-of-Plus-infinite2)

corollary Card-order-Plus-infinite:
assumes INF: ¬finite(Field r) and CARD: Card-order r and
LEQ: p ≤ o r
shows | (Field r) <+> (Field p) | = o r ∧ | (Field p) <+> (Field r) | = o r
proof
  have | Field r <+> Field p | = o r ∧ | Field p <+> Field r | = o r
  using assms by (simp add: card-of-Plus-infinite card-of-mono2)
thus ?thesis
using assms card-of-Field-ordIso[of r]
  ordIso-transitive[of |Field r <+> Field p|]
  ordIso-transitive[of - |Field r|] by blast
qed

28.5 The cardinal $\omega$ and the finite cardinals

The cardinal $\omega$, of natural numbers, shall be the standard non-strict order relation on nat, that we abbreviate by natLeq. The finite cardinals shall be the restrictions of these relations to the numbers smaller than fixed numbers $n$, that we abbreviate by natLeq-on $n$.

definition (natLeq::(nat * nat) set) ≡ \{(x,y). x ≤ y\}
definition (natLess::(nat * nat) set) ≡ \{(x,y). x < y\}

abbreviation natLeq-on :: nat ⇒ (nat * nat) set
where natLeq-on n ≡ \{(x,y). x < n ∧ y < n ∧ x ≤ y\}

lemma infinite-cartesian-product:
assumes ¬finite A ¬finite B
shows ¬finite (A × B)
proof
  assume finite (A × B)
  from assms(1) have A ≠ {} by auto
  with (finite (A × B)) have finite B using finite-cartesian-productD2 by auto
  with assms(2) show False by simp
qed

28.5.1 First as well-orders

lemma Field-natLeq: Field natLeq = (UNIV::nat set)
by(unfold Field-def natLeq-def, auto)

lemma natLeq-Refl: Refl natLeq
unfolding refl-on-def Field-def natLeq-def by auto
lemma natLeq-trans: trans natLeq
unfolding trans-def natLeq-def by auto

lemma natLeq-Preorder: Preorder natLeq
unfolding preorder-on-def
by (auto simp add: natLeq-Refl natLeq-trans)

lemma natLeq-antisym: antisym natLeq
unfolding antisym-def natLeq-def by auto

lemma natLeq-Partial-order: Partial-order natLeq
unfolding partial-order-on-def
by (auto simp add: natLeq-Preorder natLeq-antisym)

lemma natLeq-Total: Total natLeq
unfolding total-on-def natLeq-def by auto

lemma natLeq-Linear-order: Linear-order natLeq
unfolding linear-order-on-def
by (auto simp add: natLeq-Partial-order natLeq-Total)

lemma natLeq-natLess-Id: natLess = natLeq − Id
unfolding natLeq-def natLess-def by auto

lemma natLeq-Well-order: Well-order natLeq
unfolding well-order-on-def
using natLeq-Linear-order wf-less natLeq-natLess-Id natLeq-def natLess-def by auto

lemma Field-natLeq-on: Field (natLeq-on n) = \{x. x < n\}
unfolding Field-def by auto

lemma natLeq-underS-less: underS natLeq n = \{x. x < n\}
unfolding underS-def natLeq-def by auto

lemma Restr-natLeq: Restr natLeq \{x. x < n\} = natLeq-on n
unfolding natLeq-def by force

lemma Restr-natLeq2:
Restr natLeq (underS natLeq n) = natLeq-on n
by (auto simp add: Restr-natLeq natLeq-underS-less)

lemma natLeq-on-Well-order: Well-order(natLeq-on n)
using Restr-natLeq[of n] natLeq-Well-order
Well-order-Restr[of natLeq \{x. x < n\}] by auto

corollary natLeq-on-well-order-on: well-order-on \{x. x < n\} (natLeq-on n)
using natLeq-on-Well-order Field-natLeq-on by auto
lemma natLeq-on-wo-rel: wo-rel(natLeq-on n)
unfolding wo-rel-def using natLeq-on-Well-order.

28.5.2 Then as cardinals

lemma natLeq-Card-order: Card-order natLeq
proof(auto simp add: natLeq-Well-order
  Card-order-iff-Restr-underS Restr-natLeq2, simp add: Field-natLeq)
fix n have finite(Field (natLeq-on n)) by (auto simp: Field-def)
moreover have ¬finite(UNIV::nat set) by auto
ultimately show natLeq-on n <o UNIV::nat set
  Field-card-of[of UNIV::nat set]
qed
corollary card-of-Field-natLeq:
  |Field natLeq| =o natLeq
using Field-natLeq natLeq-Card-order Card-order-iff-ordIso-card-of[of natLeq]
  ordIso-symmetric[of natLeq] by blast
corollary card-of-nat:
  |UNIV::nat set| =o natLeq
using Field-natLeq card-of-Field-natLeq by auto
corollary infinite-iff-natLeq-ordLeq:
  ¬finite A = ( natLeq ≤o |A| )
  ordIso-ordLeq-trans ordLeq-ordIso-trans ordIso-symmetric by blast
corollary finite-iff-ordLess-natLeq:
  finite A = ( |A| <o natLeq)
using infinite-iff-natLeq-ordLeq not-ordLeq-iff-ordLess
  card-of-Well-order natLeq-Well-order by blast

28.6 The successor of a cardinal

First we define isCardSuc r r’, the notion of r’ being a successor cardinal of r. Although the definition does not require r to be a cardinal, only this case will be meaningful.

definition isCardSuc :: 'a rel ⇒ 'a set rel ⇒ bool
where
isCardSuc r r’ ≡
  Card-order r’ ∧ r <o r’ ∧
  (∀ (r’’: 'a set rel). Card-order r’’ ∧ r <o r’’ → r’ ≤o r’’)

Now we introduce the cardinal-successor operator cardSuc, by picking some cardinal-order relation fulfilling isCardSuc. Again, the picked item shall be
proved unique up to order-isomorphism.

**Definition**

\[
\text{cardSuc} :: 'a rel \Rightarrow 'a set rel
\]

where

\[
\text{cardSuc } r \equiv \text{SOME } r' . \text{isCardSuc } r r'
\]

**Lemma**

\[
\text{exists-minim-Card-order}:
\begin{align*}
\forall [\{ r \in R . \text{Card-order } r\} \implies \exists r \in R . \forall r' \in R . r <_o r']
\end{align*}
\]

**Unfolding**

\[
\text{card-order-on-def}
\]

**Lemma**

\[
\text{exists-isCardSuc}
\]

assumes

\[
\text{Card-order } r
\]

shows

\[
\exists r'. \text{isCardSuc } r r'
\]

**Proof**

- let \( ?R = \{ (r' :: 'a set rel). \text{Card-order } r' \land r < _o r' \} \)
- have \( \text{Pow} (\text{Field } r) \in ?R \land (\forall r \in ?R . \text{Card-order } r) \) using assms
  by (simp add: card-of-Card-order Card-order-Pow)
- then obtain \( r \) where \( r \in ?R \land (\forall r' \in ?R . r \leq _o r') \)
  using \( \text{exists-minim-Card-order[of } ?R \text{] by blast} \)
  thus \( \text{thesis} \) unfolding \( \text{isCardSuc-def by auto} \)
  qed

**Lemma**

\[
\text{cardSuc-isCardSuc}:
\]

assumes \( \text{Card-order } r \)

shows \( \text{isCardSuc } r (\text{cardSuc } r) \)

**Unfolding**

\[
\text{cardSuc-def}
\]

**Lemma**

\[
\text{cardSuc-Card-order}:
\]

\[
\text{Card-order } r \implies \text{Card-order}(\text{cardSuc } r)
\]

**Using**

\[
\text{cardSuc-isCardSuc unfolding isCardSuc-def by blast}
\]

**Lemma**

\[
\text{cardSuc-greater}:
\]

\[
\text{Card-order } r \implies r < _o \text{ cardSuc } r
\]

**Using**

\[
\text{cardSuc-isCardSuc unfolding isCardSuc-def by blast}
\]

**Lemma**

\[
\text{cardSuc-ordLeq}:
\]

\[
\text{Card-order } r \implies r \leq _o \text{ cardSuc } r
\]

**Using**

\[
\text{cardSuc-greater ordLeq-iff-ordLess-or-ordIso by blast}
\]

The minimality property of \( \text{cardSuc} \) originally present in its definition is local to the type \( 'a \text{ set rel} \), i.e., that of \( \text{cardSuc } r \):

**Lemma**

\[
\text{cardSuc-least-aux}:
\]

\[
\text{Card-order } (r :: 'a \text{ rel}); \text{Card-order } (r' :: 'a \text{ set rel}); r < _o r' \implies \text{cardSuc } r \leq _o r'
\]

**Using**

\[
\text{cardSuc-isCardSuc unfolding isCardSuc-def by blast}
\]

But from this we can infer general minimality:

**Lemma**

\[
\text{cardSuc-least}:
\]

**Assumes**

\[
\text{CARD: Card-order } r \text{ and CARD': Card-order } r' \text{ and LESS: } r < _o r'
\]
shows cardSuc r ≤o r'
proof -
  let ?p = cardSuc r
  have 0: Well-order ?p ∧ Well-order r'
    using assms cardSuc-Card-order unfolding card-order-on-def by blast
  { assume r' <o ?p
    then obtain r'' where 1: Field r'' < Field ?p and 2: r' =o r'' ∧ r'' <o ?p
      using internalize-ordLess[of r' ?p] by blast
      have Card-order r'' using CARD' Card-order-ordIso2 2 by blast
      moreover have r <o r'' using LESS 2 ordLess-ordIso-trans by blast
      ultimately have ?p ≤o r'' using cardSuc-least-aux CARD by blast
      hence False using 2 not-ordLess-ordLeq by blast }
  thus ?thesis using 0 ordLess-or-ordLeq by blast
qed

lemma cardSuc-ordLess-ordLeq:
assumes CARD: Card-order r and CARD': Card-order r'
shows (r <o r') = (cardSuc r ≤o r')
proof (auto simp add: assms cardSuc-least)
  assume cardSuc r ≤o r'
  thus r <o r' using assms cardSuc-greater ordLess-ordLeq-trans by blast
qed

lemma cardSuc-ordLeq-ordLess:
assumes CARD: Card-order r and CARD': Card-order r'
shows (r' <o cardSuc r) = (r' ≤o r)
proof -
  have Well-order r ∧ Well-order r'
    using assms unfolding card-order-on-def by auto
  moreover have Well-order(cardSuc r)
    using assms cardSuc-Card-order card-order-on-def by blast
  ultimately show ?thesis
    using assms cardSuc-ordLeq-ordLess-ordLeq[of r r'] not-ordLeq-iff-ordLess[of r r'] not-ordLeq-iff-ordLess[of r' cardSuc r] by blast
qed

lemma cardSuc-mono-ordLeq:
assumes CARD: Card-order r and CARD': Card-order r'
shows (cardSuc r ≤o cardSuc r') = (r ≤o r')
using assms cardSuc-ordLeq-ordLeq cardSuc-ordLeq-cardSuc-Card-order by blast

lemma cardSuc-invar-ordIso:
assumes CARD: Card-order r and CARD': Card-order r'
shows (cardSuc r =o cardSuc r') = (r =o r')
proof -
  have 0: Well-order r ∧ Well-order r' ∧ Well-order(cardSuc r) ∧ Well-order(cardSuc...
using assms by (simp add: card-order-on-well-order-on cardSuc-Card-order)
thus ?thesis using ordIso-iff-ordLeq[of r r'] ordIso-iff-ordLeq using cardSuc-mono-ordLeq[of r r'] cardSuc-mono-ordLeq[of r' r] assms by blast
qed

lemma card-of-cardSuc-finite:
finite(Field(cardSuc |A| )) = finite A
proof
  assume *: finite (Field (cardSuc |A| ))
  have 0: |Field(cardSuc |A| )| =o cardSuc |A|
    using card-of-Card-order cardSuc-Card-order card-of-Field-ordIso by blast
  hence |A| \leq_o |Field(cardSuc |A| )|
  thus finite A using * card-of-ordLeq-finite by blast
next
  assume finite A
  then have finite ( Field |Pow A| ) unfolding Field-card-of by simp
  then show finite (Field (cardSuc |A| ))
    proof (rule card-of-ordLeq-finite[OF card-of-mono2, rotated])
      show cardSuc |A| \leq_o |Pow A|
        by (rule iffD1[OF cardSuc-ordLess-ordLeq card-of-Pow]) (simp-all add: card-of-Card-order)
    qed
  qed

lemma cardSuc-finite:
  assumes Card-order r
  shows finite (Field (cardSuc r)) = finite (Field r)
proof-
  let ?A = Field r
  have |?A| =o r using assms by (simp add: card-of-Field-ordIso)
  hence cardSuc |?A| =o cardSuc r using assms
    by (simp add: card-of-Card-order cardSuc-invar-ordIso)
  moreover have |Field (cardSuc |?A| )| =o cardSuc |?A|
    by (simp add: card-of-cardSuc-finite Card-order)[cardSuc-cardSuc-cardSuc-Card-order]
  moreover
  { have |Field (cardSuc r)| =o cardSuc r
    using assms by (simp add: card-of-Field-ordIso)
    hence cardSuc r =o |Field (cardSuc r)|
      using ordIso-symmetric by blast
  }
  ultimately have |Field (cardSuc |?A| )| =o |Field (cardSuc r)|
    using ordIso-transitive by blast
  hence finite (Field (cardSuc |?A| )) = finite (Field (cardSuc r))
    using card-of-ordIso-finite by blast
  thus ?thesis by (simp only: card-of-cardSuc-finite)
qed
lemma card-of-Plus-ordLess-infinite:
assumes INF: ¬finite C and

\hspace{1cm} \text{LESS1: } |A| < o \ C \text{ and } \text{LESS2: } |B| < o \ C
shows \(|A <+> B| < o \ C|
proof(cases \(A = \{\}\) \vee \(B = \{\}\))
\hspace{1cm} assume Case1: \(A = \{\}\ \vee B = \{\}\)
\hspace{2cm} hence \(|A| = o \ \{A <+> B| \vee |B| = o \ \{A <+> B|
\hspace{1cm} using card-of-Plus-empty1 card-of-Plus-empty2 by blast
\hspace{1cm} hence \(|A <+> B| = o \ \{A| \vee |A <+> B| = o \ |B|
\hspace{1cm} using ordIso-symmetric[of \(|A|\) ordIso-symmetric[of \(|B|\)] by blast
\hspace{1cm} thus ?thesis using LESS1 LESS2
\hspace{1cm} ordIso-ordLess-trans[of \{|A <+> B| |A|\]
\hspace{1cm} ordIso-ordLess-trans[of \{|A <+> B| |B|\] by blast

next
assume Case2: \(\neg(A = \{\}\ \vee B = \{\}\)
\hspace{1cm} {assume \(*: \ |C| \leq o \ \{A <+> B|
\hspace{2cm} hence \(\neg\text{finite } (A <+> B)\) using INF card-of-ordLeq-finite by blast
\hspace{1cm} hence 1: \(\neg\text{finite } A \vee \neg\text{finite } B\) using finite-Plus by blast
\hspace{1cm} {assume Case21: \(|A| \leq o \ |B|
\hspace{2cm} hence \(\neg\text{finite } B\) using 1 card-of-ordLeq-finite by blast
\hspace{2cm} hence \(|A <+> B| = o \ |B|\) using Case2 Case21
\hspace{2cm} by (auto simp add: card-of-Plus-infinite)
\hspace{1cm} hence False using LESS2 not-ordLess-ordLeq * ordLeq-ordIso-trans by blast

\}
moreover
\hspace{1cm} {assume Case22: \(|B| \leq o \ |A|
\hspace{2cm} hence \(\neg\text{finite } A\) using 1 card-of-ordLeq-finite by blast
\hspace{2cm} hence \(|A <+> B| = o \ |A|\) using Case2 Case22
\hspace{2cm} by (auto simp add: card-of-Plus-infinite)
\hspace{1cm} hence False using LESS1 not-ordLess-ordLeq * ordLeq-ordIso-trans by blast

\}
ultimately have False using ordLeq-total card-of-Well-order[of A]
\hspace{1cm} card-of-Well-order[of B] by blast
\}
thus ?thesis using ordLess-or-ordLeq[of \{|A <+> B| |C|\]
\hspace{1cm} card-of-Well-order[of A <+> B] card-of-Well-order[of C] by auto
qed

lemma card-of-Plus-ordLess-infinite-Field:
assumes INF: ¬finite (Field r) and r: Card-order r and

\hspace{1cm} \text{LESS1: } |A| < o \ r \text{ and } \text{LESS2: } |B| < o \ r
shows \(|A <+> B| < o \ r|
proof-
\hspace{1cm} let \(\bar{r} = \text{Field } r\)
\hspace{1cm} have 1: \(r = o \ |\bar{C}| \wedge |\bar{C}| = o \ r\) using r card-of-Field-ordIso
\hspace{1cm} ordIso-symmetric by blast
\hspace{1cm} hence \(|A| < o \ |\bar{C}| \ \{B| < o \ |\bar{C}|
\hspace{1cm} using LESS1 LESS2 ordLess-ordIso-trans by blast
hence |A <+> B| <o |?C| using INF
card-of-Plus-ordLess-infinite by blast
thus ?thesis using 1 ordLess-ordIso-trans by blast
qed

lemma card-of-Plus-ordLeq-infinite-Field:
assumes r: ¬finite (Field r) and A: |A| ≤ o r and B: |B| ≤ o r
and c: Card-order r
shows |A <+> B| ≤ o r
proof −
  let ?r' = cardSuc r
  have Card-order ?r' ∧ ¬finite (Field ?r') using assms
    by (simp add: cardSuc-Card-order cardSuc-finite)
  moreover have |A| <o ?r' and |B| <o ?r' using A B c
    by (auto simp: card-of-card-order-on Field-card-of cardSuc-ordLeq-ordLess)
  ultimately have |A <+> B| <o ?r'
    using card-of-Plus-ordLess-infinite-Field by blast
  thus ?thesis using c r
    by (simp add: card-of-card-order-on Field-card-of cardSuc-ordLeq-ordLess)
qed

lemma card-of-Un-ordLeq-infinite-Field:
assumes C: ¬finite (Field r) and A: |A| ≤ o r and B: |B| ≤ o r
and r: Card-order r
shows |A Un B| ≤ o r
using assms card-of-Plus-ordLeq-infinite-Field card-of-Un-Plus-ordLeq
ordLeq-transitive by fast

28.7 Regular cardinals

definition cofinal where
cofinal A r ≡
  ∀a ∈ Field r. ∃b ∈ A. a ≠ b ∧ (a,b) ∈ r

definition regularCard where
regularCard r ≡
  ∀K. K ≤ Field r ∧ cofinal K r −→ |K| = o r

definition relChain where
relChain r As ≡
  ∀i j. (i,j) ∈ r −→ As i ≤ As j

lemma regularCard-UNION:
assumes r: Card-order r regularCard r
and As: relChain r As
and Bsub: B ≤ (⋃i ∈ Field r. As i)
and cardB: |B| <o r
shows ∃i ∈ Field r. B ≤ As i
proof −
let \( \phi b j = \lambda b j \cdot j \in \text{Field } r \wedge b \in \text{As } j \)

have \( \forall b \in B. \exists j. \phi b j \) using Bsub by blast

then obtain \( f \) where \( \forall b \in B. \phi b f(b) \)

using bchoice[of B ?phi] by blast

let \( K = f \cdot B \)

\{ assume 1: \( \forall i. i \in \text{Field } r \Rightarrow \neg B \leq \text{As } i \)

have 2: cofinal \( K r \)

unfolding cofinal-def proof auto

fix \( i \) assume \( i \in \text{Field } r \)

with 1 obtain \( b \) where \( b \in B \wedge b \notin \text{As } i \) by blast

hence \( i \neq f(b) \wedge (f(b), i) \in r \)

using As f unfolding relChain-def by auto

unfolding card-order-on-def well-order-on-def linear-order-on-def total-on-def using \( i f b \) by auto

with \( b \) show \( \exists b \in B. i \neq f b \wedge (i, f b) \in r \) by blast

qed

moreover have \( K \leq \text{Field } r \) using \( f \) by blast

ultimately have \( |K| = o r \)

unfolding regularCard-def by blast

moreover

\{ have \( |K| \leq o |B| \) using card-of-image .

hence \( |K| < o r \) using \( \text{ordLeq-ordLess-trans} \) by blast

ultimately have False using not-ordLess-ordIso by blast

\}

thus \( \phi \) by blast

qed

lemma infinite-cardSuc-regularCard:

assumes \( r\inf: \neg \text{finite (Field } r) \) and \( r\card: \text{Card-order } r \)

shows regularCard \( \langle \text{cardSuc } r \rangle \)

proof

let \( \phi r' = \text{cardSuc } r \)

have \( \phi r': \text{Card-order } \phi r' \)

!! p. Card-order \( p \rightarrow (p < o r) = (p < o \phi r') \)

using r-card by \( \langle \text{simp: cardSuc-Card-order cardSuc-ordLeq-ordLess} \rangle \)

show \( \phi \)

unfolding regularCard-def proof auto

fix \( K \) assume 1: \( K \leq \text{Field } \phi r' \) and 2: cofinal \( K \phi r' \)

hence \( |K| \leq o |\text{Field } \phi r'| \) by \( \langle \text{simp only: card-of-mono}\rangle \)

also have 22: \( |\text{Field } \phi r'| = o \phi r' \)

using r' by \( \langle \text{simp add: card-of-Field-ordIso[of } \phi r' \rangle \)

finally have \( |K| \leq o \phi r' \).

moreover

\{ have \( \phi L = \bigcup j \in K. \text{underS } \phi r' j \)

let \( \phi J = \text{Field } r \)

have \( rJ: r = o |\phi J| \)

using r-card card-of-Field-ordIso ordIso-symmetric by blast \}
assume $|K| < o \ ?r'$

hence $|K| < o r$ using $r'$ card-of-Card-order[of $K$] by blast

hence $|K| \leq o \ |?J|$ using $rJ \ ordLeq-ordIso-trans$ by blast

moreover

\{ have $\forall j \in K. \ underS \ ?r' j < o \ ?r'$

using $r' \ 1$ by (auto simp: card-of-underS)

hence $\forall j \in K. \ underS \ ?r' j \leq o \ r$

using $r' \ card-of-Card-order$ by blast

hence $\forall j \in K. \ underS \ ?r' j \leq o \ |?J|$

using $rJ \ ordLeq-ordIso-trans$ by blast

\}

ultimately have $|?L| \leq o \ |?J|$

using $r-inf \ card-of-UNION-ordLeq-infinite$ by blast

hence $|?L| \leq o \ r$ using $rJ \ ordIso-symmetric \ ordLeq-ordIso-trans$ by blast

hence $|?L| < o \ ?r'$ using $r' \ card-of-Card-order$ by blast

moreover

\{

have Field $?r' \leq ?L$

using 2 unfolding underS-def cofinal-def by auto

hence $[Field \ ?r'] \leq o \ |?L|$ by (simp add: card-of-mono1)

hence $?r' \leq o \ |?L|$

using 22 ordIso-ordLeq-trans ordIso-symmetric by blast

\}

ultimately have $|?L| < o \ |?L|$ using ordLess-ordLeq-trans by blast

hence False using ordLess-irreflexive by blast

\}

ultimately show $|K| = o \ ?r'$

unfolding ordLeq-iff-ordLess-or-ordIso by blast

qed

qed

lemma cardSuc-UNION:

assumes $r:\ Card-order \ r$ and $\neg finite \ (Field \ r)$

and As: relChain (cardSuc $r$) As

and Bsub: $B \leq (\bigcup \ i \in Field \ (cardSuc \ r). \ As \ i)$

and cardB: $|B| < o \ r$

shows $\exists i \in Field \ (cardSuc \ r). \ B \leq As \ i$

proof

let $?r' = cardSuc \ r$

have Card-order $?r' \land |B| < o \ ?r'$

using $r \ cardB \ cardSuc-ordLeq-ordLess \ cardSuc-Card-order$

card-of-Card-order by blast

moreover have regularCard $?r'$

using assms by(simp add: infinite-cardSuc-regularCard)

ultimately show $?thesis$

using As Bsub cardB regularCard-UNION by blast

qed
28.8 Others

**lemma** card-of-Func-Times:

\[ |\text{Func} (A \times B) C| = o |\text{Func} A (\text{Func} B C)| \]

**unfolding** card-of-ordIso[ symmetric ]

**using** bij-betw-curr by blast

**lemma** card-of-Pow- Func:

\[ |\text{Pow} A| = o |\text{Func} A (\text{UNIV}::\text{bool set})| \]

**proof** –

define \( F \) where [abs-def]:

\[ F \quad \text{a} = \]

\( \text{if a} \in A \text{ then } (\text{if a} \in A' \text{ then True else False } \text{ else undefined}) \text{ for } A' \text{ a} \)

**have** bij-betw \( F \) (Pow A) (Func A (UNIV::bool set))

**unfolding** bij-betw-def inj-on-def **proof** (intro ballI impI conjI)

fix \( A1 \ A2 \) assume \( A1 \in \text{Pow} A \ A2 \in \text{Pow} A \)

thus \( A1 = A2 \) unfolding F-def Pow-def fun-eq-iff by (auto split: if-split-asn)

next

show \( F' \) Pow A = Func A UNIV

**proof** safe

fix \( f \) assume \( f : f \in \text{Func} A (\text{UNIV}::\text{bool set}) \)

show \( \?A1 \in \{a \in A. f a = \text{True} \} \)

show \( f = F \?A1 \)

**unfolding** F-def apply (rule ext)

using \( f \) unfolding Func-def mem-Collect-eq by auto

**qed** auto

**qed** (unfold Func-def mem-Collect-eq F-def, auto)

**qed** thus \( \?\text{thesis} \) unfolding card-of-ordIso[ symmetric ] by blast

**qed**

**lemma** card-of-Func-UNIV:

\[ |\text{Func} (\text{UNIV}::\text{'}a set) (B::\text{'}b set)| = o |\{f::a \Rightarrow b. \text{range } f \subseteq B\}| \]

**apply** (rule ordIso-symmetric) **proof**(intro card-of-ordIso1)

let \( \?F = \lambda f : (a::a). (\{ f a::b\}) \)

show bij-betw \( \?F \) \{f. \text{range } f \subseteq B\} (Func UNIV B)

**unfolding** bij-betw-def inj-on-def **proof** safe

fix \( h : a' \Rightarrow b \) assume \( h : h \in \text{Func UNIV }B \)

hence \( \forall a. \exists b. h a = b \) **unfolding** Func-def by auto

then obtain \( f \) where \( f : \forall a. h a = f a \) by blast

hence range \( f \subseteq B \) using \( h \) unfolding Func-def by auto

thus \( h \in (\lambda f a. f a)^\prime \{f. \text{range } f \subseteq B\} \) using \( f \) by auto

**qed** (unfold Func-def fun-eq-iff, auto)

**qed**

**lemma** Func-Times-Range:

\[ |\text{Func} A (B \times C)| = o |\text{Func} A B \times \text{Func} A C| (\text{is } |?\text{LHS}| = o |?\text{RHS}|) \]

**proof** –

let \( ?F = \lambda fg. (x. \text{if } x \in A \text{ then } \text{fst } (fg x) \text{ else undefined}, \)

\( \text{\lambda x. } \text{if } x \in A \text{ then } \text{snd } (fg x) \text{ else undefined} \)
let \( ?G = \lambda (f, g) . \text{if } x \in A \text{ then } (f x, g x) \text{ else undefined} \)

have bij-betw \( ?F \ ?LHS \ ?RHS \) unfolding bij-betw-def inj-on-def

proof (intro conjI_impl ballI equalityI subsetI)
  fix \( f \) \( g \)
  assume \( \ast : f \in \text{Func} A (B \times C) \) \( g \in \text{Func} A (B \times C) \) \( ?F f = ?F g \)

  show \( f = g \)
  proof
    fix \( x \) from \( \ast \) have \( \text{fst} (f x) = \text{fst} (g x) \wedge \text{snd} (f x) = \text{snd} (g x) \)
    by (cases \( x \in A \)) (auto simp: Func-def fun-eq_iff split: if-splits)
    then show \( f x = g x \)
      by (subst (1 2) surjective-pairing) simp
  qed

next
  fix \( fg \)
  assume \( fg \in \text{Func} A B \times \text{Func} A C \)

  thus \( fg \in ?F ' \text{Func} A (B \times C) \)
  by (intro image-eqI[of - - ?G fg]) (auto simp: Func-def)

  qed (auto simp: Func-def fun-eq_iff)

thus \( ?\text{thesis} \) using card-of-ordIso by blast

qed

end

29 Cardinal Arithmetic as Needed by Bounded Natural Functors

theory BNF-Cardinal-Arithmetic
imports BNF-Cardinal-Order-Relation
begin

lemma dir-image: \( \exists \forall x y. (f x = f y) = (x = y); \text{Card-order } r \implies r = o \text{ dir-image } r f \)
by (rule dir-image-ordIso) (auto simp: inj-on_def card-order-on_def)

lemma card-order-dir-image:
  assumes bij: \( \text{bij } f \) and co: \( \text{card-order } r \)
  shows card-order (dir-image \( r f \))
proof –
  from assms have Field (dir-image \( r f \)) = UNIV
    using card-order-on-Card-order[of UNIV \( r \)] unfolding bij-def dir-image-Field by auto

moreover from bij have \( \exists \forall x y. (f x = f y) = (x = y) \)
  unfolding bij-def
  inj-on-def by auto

with co have Card-order (dir-image \( r f \))
  using card-order-on-Card-order[of UNIV \( r \)] Card-order-ordIso2[OF - dir-image]
  by blast

ultimately show \( ?\text{thesis} \)
by auto

qed

lemma ordIso-refl: \( \text{Card-order } r \implies r = o r \)
by (rule card-order-on-ordIso)
**THEORY “BNF-Cardinal-Arithmetic”**

lemma ordLeq-refl: Card-order r ⇒ r ≤ o r
by (rule ordIso-imp-ordLeq, rule card-order-on-ordIso)

lemma card-of-ordIso-subst: A = B ⇒ |A| = o |B|
by (simp only: ordIso-refl card-of-Card-order)

lemma Field-card-order: card-order r ⇒ Field r = UNIV
using card-order-on-Card-order[of UNIV r] by simp

29.1 Zero

definition czero where
czero = card-of { }

lemma czero-ordIso:
czero = o czero
using card-of-empty-ordIso by (simp add: czero-def)

lemma card-of-ordIso-czero-iff-empty:
|A| = o (czero :: 'b rel) ←→ A = ({ } :: 'a set)
unfolding czero-def by (rule iffI[OF card-of-empty2]) (auto simp: card-of-refl card-of-empty-ordIso)

abbreviation Cnotzero where
Cnotzero (r :: 'a rel) ≡ ¬(r = o (czero :: 'a rel)) ∧ Card-order r

lemma Cnotzero-imp-not-empty:
Cnotzero r ⇒ Field r ≠ { }
unfolding Card-order-iff-ordIso-card-of czero-def by force

lemma czeroI:
[Card-order r; Field r = { }] ⇒ r = o czero
using Cnotzero-imp-not-empty ordIso-transitive[of UNIV - czero-ordIso] by blast

lemma czeroE:
r = o czero ⇒ Field r = {}
unfolding czero-def
by (drule card-of-cong) (simp only: Field-card-of-card-of-empty2)

lemma Cnotzero-mono:
[ Cnotzero r; Card-order q; r ≤ o q ] ⇒ Cnotzero q
apply (rule ccontr)
apply auto
apply (drule czeroE)
apply (erule notE)
apply (erule czeroI)
apply (drule card-of-mono2)
29.2 (In)finite cardinals

definition cinfinite where
  cinfinite r = (∼ finite (Field r))

abbreviation Cinfinite where
  Cinfinite r ≡ cinfinite r ∧ Card-order r

definition cfinite where
cfinite r = finite (Field r)

abbreviation Cfinite where
  Cfinite r ≡ cfinite r ∧ Card-order r

lemma Cfinite-ordLess-Cinfinite: [ Cfinite r; Cinfinite s ] ⇒ r <o s
  unfolding cfinite-def cinfinite-def
  by (blast intro: finite-ordLess-infinite card-order-on-well-order-on)

lemmas natLeq-card-order = natLeq-Card-order [unfolded Field-natLeq]

lemma natLeq-cinfinite: cinfinite natLeq
  unfolding cinfinite-def Field-natLeq by (rule infinite-UNIV-nat)

lemma natLeq-ordLeq-cinfinite:
  assumes inf: Cinfinite r
  shows natLeq ≤o r
proof −
  from inf have natLeq ≤o |Field r| unfolding cfinite-def
    using infinite-iff-natLeq-ordLeq by blast
  also from inf have |Field r| =o r by (simp add: card-of-unique ordIso-symmetric)
  finally show ?thesis .
  qed

lemma cinfinite-not-czero: cinfinite r ⇒ ∼(r =o czero :: 'a rel))
  unfolding cinfinite-def by (cases Field r = {}) (auto dest: czeroE)

lemma Cinfinite-Cnotzero: Cinfinite r ⇒ Cnotzero r
  by (rule conjI[OF cinfinite-not-czero]) simp-all

lemma Cinfinite-cong: [r1 =a r2; Cinfinite r1] ⇒ Cinfinite r2
  using Card-order-ordIso2[of r1 r2] unfolding cinfinite-def ordIso-iff-ordLeq
  by (auto dest: card-of-ordLeq-infinite[OF card-of-mono2])

lemma cinfinite-mono: [r1 ≤o r2; cinfinite r1] ⇒ cinfinite r2
  unfolding cinfinite-def by (auto dest: card-of-ordLeq-infinite[OF card-of-mono2])
29.3 Binary sum

definition csum (infixr +c 65) where
  r1 +c r2 ≡ |Field r1 <+> Field r2|

lemma Field-csum: Field (r +c s) = Inl ' Field r ∪ Inr ' Field s
  unfolding csum-def Field-card-of by auto

lemma Card-order-csum:
  Card-order (r1 +c r2)
  unfolding csum-def by (simp add: card-of-Card-order)

lemma csum-Cnotzero1:
  Cnotzero r1 =⇒ Cnotzero (r1 +c r2)
  unfolding csum-def using Cnotzero-imp-not-empty[of r1] Plus-eq-empty-conv[of
  Field r1 Field r2]
  card-of-ordIso-czero-iff-empty[of Field r1 <+> Field r2] by (auto intro: card-of-Card-order)

lemma card-order-csum:
  assumes card-order r1 card-order r2
  shows card-order (r1 +c r2)
  proof
    have Field r1 = UNIV Field r2 = UNIV using assms card-order-on-Card-order
    by auto
    thus thesis unfolding csum-def by (auto simp: card-of-card-order-on)
  qed

lemma csum-Cinfinite:
  assumes Cinfinite r1 Cinfinite r2
  shows Cinfinite (r1 +c r2)
  unfolding csum-def using (rule conjI[OF - card-of-Card-order]) (auto simp: Field-card-of)

lemma Cinfinite-csum-
  Cinfinite r1 Cinfinite r2 =⇒ Cinfinite (r1 +c r2)
  unfolding csum-def by (rule conjI[OF - card-of-Card-order]) (auto simp: Field-card-of)

lemma Cinfinite-csum-weak:
  [Cinfinite r1; Cinfinite r2] =⇒ Cinfinite (r1 +c r2)
  by (erule Cinfinite-csum1)

lemma csum-cong: [p1 =o r1; p2 =o r2] =⇒ p1 +c p2 =o r1 +c r2
  by (simp only: csum-def ordIso-Plus-cong)

lemma csum-cong1: p1 =o r1 =⇒ p1 +c q =o r1 +c q
  by (simp only: csum-def ordIso-Plus-cong1)
**THEORY “BNF-Cardinal-Arithmetic”**

lemma `csum-cong2`: \( p2 = o r2 \implies q + c p2 = o q + c r2 \)
by (simp only: `csum-def ordIso-Plus-cong2`)

lemma `csum-mono`: \( p1 \leq o r1; p2 \leq o r2 \implies p1 + c p2 \leq o r1 + c r2 \)
by (simp only: `csum-def ordIso-Plus-mono`)

lemma `csum-mono1`: \( p1 \leq o r1 \implies p1 + c q \leq o r1 + c q \)
by (simp only: `csum-def ordLeq-Plus-mono1`)

lemma `csum-mono2`: \( p2 \leq o r2 \implies q + c p2 \leq o q + c r2 \)
by (simp only: `csum-def ordLeq-Plus-mono2`)

lemma `ordLeq-csum1`: \( \text{Card-order } p1 \implies p1 \leq o p1 + c p2 \)
by (simp only: `csum-def Card-order-Plus1`)

lemma `ordLeq-csum2`: \( \text{Card-order } p2 \implies p2 \leq o p1 + c p2 \)
by (simp only: `csum-def Card-order-Plus2`)

lemma `csum-com`: \( p1 + c p2 = o p2 + c p1 \)
by (simp only: `csum-def card-of-Plus-commute`)

lemma `csum-assoc`: \( (p1 + c p2) + c p3 = o p1 + c(p2 + c p3) \)
by (simp only: `csum-def Field-card-of card-of-Plus-assoc`)

lemma `Cfinite-csum`: \( \text{Cfinite } r; \text{Cfinite } s \implies \text{Cfinite } (r + c s) \)
unfolding `cfinite-def csum-def Field-card-of` using `card-of-card-order-on` by simp

lemma `csum-csum`: \( (r1 + c r2) + c (r3 + c r4) = o (r1 + c r3) + c (r2 + c r4) \)
proof –
have \( r1 + c r2 + c (r3 + c r4) = o r1 + c r2 + c (r3 + c r4) \)
by (rule `csum-assoc`)
also have \( r1 + c r2 + c (r3 + c r4) = o r1 + c (r2 + c r3) + c r4 \)
by (intro `csum-assoc csum-cong2 ordIso-symmetric`)
also have \( r1 + c (r2 + c r3) + c r4 = o r1 + c (r3 + c r2) + c r4 \)
by (intro `csum-comm csum-cong1 csum-cong2`)
also have \( r1 + c (r3 + c r2) + c r4 = o r1 + c r3 + c r2 + c r4 \)
by (intro `csum-assoc csum-cong2 ordIso-symmetric`)
also have \( r1 + c r3 + c r2 + c r4 = o (r1 + c r3) + c (r2 + c r4) \)
by (intro `csum-assoc ordIso-symmetric`)
finally show `?thesis`.
qed

lemma `Plus-csum`: \( |A <+> B| = o |A| + c |B| \)
by (simp only: `csum-def Field-card-of card-of-refl`)

lemma `Un-csum`: \( |A \cup B| \leq o |A| + c |B| \)
using `ordLeq-ordIso-trans[OF card-of-Un-Plus-ordLeq Plus-csum]` by blast
29.4 One
definition cone where  
cone = card-of {()}

lemma Card-order-cone: Card-order cone
unfolding cone-def by (rule card-of-Card-order)

lemma Cfinite-cone: Cfinite cone
unfolding cfinite-def by (simp add: Card-order-cone)

lemma cone-not-czero: ¬ (cone =o czero)
unfolding czero-def cone-def ordIso-iff-ordLeq using card-of-empty3 empty-not-insert
by blast

lemma cone-ordLeq-Cnotzero: Cnotzero r =⇒ cone ≤ o r
unfolding cone-def by (rule Card-order-singl-ordLeq) (auto intro: czeroI)

29.5 Two
definition ctwo where  
ctwo = |UNIV :: bool set|

lemma Card-order-ctwo: Card-order ctwo
unfolding ctwo-def by (rule card-of-Card-order)

lemma ctwo-not-czero: ¬ (ctwo =o czero)
using card-of-empty3 [of UNIV :: bool set] ordIso-iff-ordLeq
unfolding czero-def ctwo-def using UNIV-not-empty by auto

lemma ctwo-Cnotzero: Cnotzero ctwo
by (simp add: ctwo-not-czero Card-order-ctwo)

29.6 Family sum
definition Csum where  
Csum r rs ≡ |SIGMA i : Field r. Field (rs i)|

syntax -Csum ::
  |pttrn ⇒ (′a * ′a) set ⇒ ′b * ′b set ⇒ ((′a * ′b) * (′a * ′b)) set
((3CSUM ::= -) [0, 51, 10] 10)

translations
  CSUM i:r. rs == CONST Csum r (%i. rs)

lemma SIGMA-CSUM: |SIGMA i : I. As i| = (CSUM i : |I|. |As i| )
by (auto simp: Csum-def Field-card-of)
29.7 Product

definition cprod (infixr ∗ 80) where
  \( r1 ∗ c r2 = |\text{Field } r1 \times \text{Field } r2| \)

lemma card-order-cprod:
  assumes card-order r1 card-order r2
  shows card-order (r1 ∗ c r2)
proof –
  have Field r1 = UNIV Field r2 = UNIV using assms card-order-on-Card-order
  by auto
  thus ?thesis by (auto simp: cprod-def card-of-card-order-on)
qed

lemma Card-order-cprod: Card-order (r1 ∗ c r2)
by (simp only: cprod-def Field-card-of card-of-card-order-on)

lemma cprod-mono1: \( p1 ≤ o r1 =⇒ p1 ∗ c q ≤ o r1 ∗ c q \)
by (simp only: cprod-def ordLeq-Times-mono1)

lemma cprod-mono2: \( p2 ≤ o r2 =⇒ q ∗ c p2 ≤ o q ∗ c r2 \)
by (simp only: cprod-def ordLeq-Times-mono2)

lemma cprod-mono: \[ p1 ≤ o r1; p2 ≤ o r2 \] =⇒ \( p1 ∗ c p2 ≤ o r1 ∗ c r2 \)
by (rule ordLeq-transitive[OF cprod-mono1 cprod-mono2])

lemma ordLeq-cprod2: \[ Cnotzero p1; Cinfinite p2 \] =⇒ \( p2 ≤ o p1 ∗ c p2 \)
unfolding cprod-def by (rule Card-order-Times2) (auto intro: czeroI)

lemma cinfinite-cprod: \[ cinfinite r1; cinfinite r2 \] =⇒ cinfinite (r1 ∗ c r2)
by (simp add: cinfinite-def cprod-def Field-card-of infinite-cartesian-product)

lemma cinfinite-cprod2: \[ Cnotzero r1; Cinfinite r2 \] =⇒ Cinfinite (r1 ∗ c r2)
by (rule cinfinite-mono) (auto intro: ordLeq-cprod2)

lemma Cinfinite-cprod2: \[ Cnotzero r1; Cinfinite r2 \] =⇒ Cinfinite (r1 ∗ c r2)
by (blast intro: cinfinite-cprod2 Card-order-cprod)

lemma cprod-cong: \[ p1 = o r1; p2 = o r2 \] =⇒ \( p1 ∗ c p2 = o r1 ∗ c r2 \)
unfolding ordIso-iff-ordLeq by (blast intro: cprod-mono)

lemma cprod-cong1: \[ p1 = o r1 \] =⇒ \( p1 ∗ c p2 = o r1 ∗ c p2 \)
unfolding ordIso-iff-ordLeq by (blast intro: cprod-mono1)

lemma cprod-cong2: \( p2 = o r2 =⇒ q ∗ c p2 = o q ∗ c r2 \)
unfolding ordIso-iff-ordLeq by (blast intro: cprod-mono2)

lemma cprod-com: \( p1 ∗ c p2 = o p2 ∗ c p1 \)
by (simp only: cprod-def card-of-Times-commute)
**THEORY “BNF-Cardinal-Arithmetic”**

lemma card-of-Csum-Times:
\[ \forall i \in I. \#A_i \leq o \#B \Rightarrow (CSUM i : |I|, |A_i| \leq o |I| * c |B|) \]
by (simp only: Csum-def cprod-def Field-card-of card-of-Sigma-mono1)

lemma card-of-Csum-Times' :
assumes Card-order r \forall i \in I. \#A_i \leq o r
shows (CSUM i : |I|, |A_i| \leq o |I| * c |Field r|)
proof
  from assms(1) have \( r = o |Field r| \) by (simp add: card-of-unique)
with assms(2) have \( \forall i \in I. \#A_i \leq o |Field r| \) by (blast intro: ordLeq-ordIso-trans)
hence \( (CSUM i : |I|, \#A_i \leq o |I| * c |Field r| \) by (simp only: card-of-Csum-Times)
also from \( * \) have \( |I| * c |Field r| \leq o |I| * c r \)
  by (simp only: Field-card-of card-of-refl cprod-def ordIso-imp-ordLeq)
finally show ?thesis .
qed

lemma cprod-csum-distrib1 :
\( r1 * c r2 + c r1 * c r3 = o r1 \)
unfolding csum-def cprod-def by (simp only: Card-order-Plus-infinite)

lemma csum-absorb2 :
\[ \#Cinfinite r2; \#r1 \leq o r2; \#cinfinite r1 \lor \#cinfinite r2 \] \Rightarrow \r1 + c r2 = o r2
unfolding csum-def by (rule conjunct2[OF Card-order-Plus-infinite])
(auto simp: cinfinite-def dest: cinfinite-mono)

lemma csum-absorb1' :
assumes card: Card-order r2
and r12 : \#r1 \leq o r2 and \#cinfinite r1 \lor \#cinfinite r2
shows \( r2 + c r1 = o r2 \)
by (rule ordIso-transitive, rule csum-com, rule csum-absorb2', (simp only: assms)+)

lemma csum-absorb1 :
\[ \#Cinfinite r2; \#r1 \leq o r2 \] \Rightarrow \#r2 + c r1 = o r2
by (rule csum-absorb1')

29.8 Exponentiation

definition cexp (infixr \( ^c \)) where
\( r1 ^c r2 \equiv |\text{Func (Field r2) (Field r1)}| \)

lemma Card-order-cexp: Card-order (\( r1 \) ^c r2)
unfolding cexp-def by (rule card-of-Card-order)

lemma cexp-mono' :
assumes 1 : \#p1 \leq o r1 and 2 : \#p2 \leq o r2
and n : \text{Field p2} = {} \Rightarrow \text{Field r2} = {}
shows \( p1 \) ^c \( p2 \) \leq o r1 \( ^c \) r2
proof(cases Field p1 = {})
  case True
  hence Field p2 \( \neq \) {} \Rightarrow Func (Field p2) {} = {}
unfolding Func-is-emp by
simp
  with True have |Field |Func (Field p2) (Field p1)| ≤ o cone
  unfolding cone-def Field-card-of
  by (cases Field p2 = {}, auto intro: surj-ordLeq simp: Func-empty)
  hence |Func (Field p2) (Field p1)| ≤ o cone by (simp add: Field-card-of cexp-def)
  hence p1 ^c p2 ≤ o cone unfolding cexp-def
  thus ?thesis
proof (cases Field p2 = {})
  case True
  with n have Field r2 = {}
  hence cone ≤ o r1 ^c r2 unfolding cone-def cexp-def Func-def
  by (auto intro: card-of-ordLeqI \[ where f=λ -. undefined \])
  thus ?thesis using ⟨p1 ^c p2 ≤ o cone⟩ ordLeq-transitive by auto
next
  case False with True have |Field (p1 ^c p2)| = o czero
  unfolding card-of-ordIso-czero-iff-empty cexp-def Field-card-of Func-def
  by auto
  thus ?thesis unfolding cexp-def card-of-ordIso-czero-iff-empty Field-card-of
  by (simp add: card-of-empty)
  qed
next
case False
have 1: |Field p1| ≤ o |Field r1| and 2: |Field p2| ≤ o |Field r2|
  using 1 2 by (auto simp: card-of-mono2)
obtain f1 where f1 : Field r1 = Field p1
  using 1 unfolding card-of-ordLeq2[OF False, symmetric] by auto
obtain f2 where f2: inj-on f2 (Field p2) f2 : Field p2 ⊆ Field r2
  using 2 unfolding card-of-ordLeq[symmetric] by blast
have 0: Func-map (Field p2) f1 f2 : (Field (r1 ^c r2)) = Field (p1 ^c p2)
  unfolding cexp-def Field-card-of using Func-map-surj[OF f1 f2 n, symmetric]
  .
  have 00: Field (p1 ^c p2) ≠ {} unfolding cexp-def Field-card-of Func-is-emp
  using False by simp
  show ?thesis
  using 0 card-of-ordLeq2[OF 00] unfolding cexp-def Field-card-of by blast
  qed

lemma cexp-mono:
  assumes 1: p1 ≤ o r1 and 2: p2 ≤ o r2
  and n: p2 =o czero ⇒ r2 =o czero and card: Card-order p2
  shows p1 ^c p2 ≤ o r1 ^c r2
  by (rule cexp-mono[OF 1 2 czeroE[OF n][OF czeroI[OF card]]])

lemma cexp-mono1:
  assumes 1: p1 ≤ o r1 and q: Card-order q
  shows p1 ^c q ≤ o r1 ^c q
  using ordLeq-refl[OF q] by (rule cexp-mono[OF 1]) (auto simp: q)

lemma cexp-mono2':
assumes 2: \( p_2 \leq r_2 \) and \( q: \text{Card-order} \) 
and \( n: \text{Field} \ p_2 = \{\} \implies \text{Field} \ r_2 = \{\} 
shows \( q \upharpoonright c \ p_2 \leq o q \upharpoonright c \ r_2 \)
using \( \text{ordLeq-refl}[OF \ q] \) by \( \text{rule cexp-mono}''[OF - 2 \ n] \) auto

lemma \( \text{cexp-mono2} \):
assumes 2: \( p_2 \leq r_2 \) and \( q: \text{Card-order} \) 
and \( n: \text{Field} \ p_2 = \{\} = \{} = \{} \implies \text{Field} \ r_2 = \{\} 
shows \( q \upharpoonright c \ p_2 \leq o q \upharpoonright c \ r_2 \)
using \( \text{ordLeq-refl}[OF \ q] \) by \( \text{rule cexp-mono}[OF - 2 \ n \ \text{card}] \) auto

lemma \( \text{cexp-mono2-Cnotzero} \):
assumes \( p_2 \leq r_2 \) \text{Card-order} \ q \ Cnotzero \ p_2 
shows \( q \upharpoonright c \ p_2 \leq o q \upharpoonright c \ r_2 \)
using \( \text{assms}(3) \) \( \text{czerol} \) by \( \text{blast intro: cexp-mono2}''[OF \ \text{assms}(1,2)] \)

lemma \( \text{cexp-cong} \):
assumes 1: \( p_1 = o r_1 \) and 2: \( p_2 = o r_2 \) 
and \( \text{Cr}: \text{Card-order} \ r_2 \) 
and \( \text{Cp}: \text{Card-order} \ p_2 \) 
shows \( p_1 \upharpoonright c \ p_2 = o r_1 \upharpoonright c \ r_2 \)
proof -
obtain \( f \) where \( \text{bij-betw} \ f \) \( (\text{Field} \ p_2) \ (\text{Field} \ r_2) \)
using \( 2 \text{ card-of-ordIso}[OF \ \text{Field} \ p_2 \ \text{Field} \ r_2] \) \( \text{card-of-cong} \) by auto
hence \( \text{Field} \ p_2 = \{\} \iff \text{Field} \ r_2 = \{\} \) unfolding \( \text{bij-betw-def} \) by auto
have \( r: \text{Field} \ p_2 = \{\} \implies \text{Field} \ r_2 = \{\} = \{\} \)
and \( p: \text{Field} \ r_2 = \{\} = \{\} \implies \text{Field} \ p_2 = \{\} = \{\} \)
using \( \text{0 Cr Cp czeroE czeroI} \) by auto
show ?thesis using \( 0 1 2 \) unfolding \( \text{ordIso-iff-ordLeq} \)
using \( r \ p \ \text{cexp-mono}[OF - - - \text{Cp}] \) \( \text{cexp-mono}[OF - - - \text{Cr}] \) by blast
qed

lemma \( \text{cexp-cong1} \):
assumes 1: \( p_1 = o r_1 \) and \( q: \text{Card-order} \) 
shows \( p_1 \upharpoonright c \ q = o r_1 \upharpoonright c \ q \)
by \( \text{rule cexp-cong}[OF 1 - q \ q] \) \( \text{rule ordIso-refl}[OF \ q] \)

lemma \( \text{cexp-cong2} \):
assumes 2: \( p_2 = o r_2 \) and \( q: \text{Card-order} \ q \) and \( p: \text{Card-order} \ p_2 \) 
shows \( q \upharpoonright c p_2 = o q \upharpoonright c r_2 \)
by \( \text{rule cexp-cong}[OF - 2] \) \( \text{auto simp only: ordIso-refl Card-order-ordIso2}[OF \ p \ 2] \ q \ p] \)

lemma \( \text{cexp-cone} \):
assumes \( \text{Card-order} \ r \)
shows \( r \upharpoonright c \ \text{cone} = o r \)
proof -
have \( r \upharpoonright c \ \text{cone} = o \) \( \text{Field} \ r \)
unfolding \( \text{cexp-def cone-def Field-card-of Func-empty} \)
lemma \textit{cexp-cprod}:
assumes \( r1 : \text{Card-order} r1 \)
shows \( (r1 \hat{\cdot} c r2) \hat{\cdot} c r3 = o r1 \hat{\cdot} c (r2 \hat{\cdot} c r3) \) (is \( ?L = o ?R \))
proof –
have \( ?L = o r1 \hat{\cdot} c (r3 \hat{\cdot} c r2) \)
  unfolding \textit{cprod-def} \textit{cexp-def} \textit{Field-card-of}
  using \textit{card-of-Field-Times} by (rule \textit{ordIso-symmetric})
also have \( r1 \hat{\cdot} c (r3 \hat{\cdot} c r2) = o ?R \)
  apply (rule \textit{cexp-cong2}) using \textit{cprod-com r1} by (auto simp: \textit{Card-order-cprod})
finally show \( ?thesis \).
qed

lemma \textit{cprod-infinite1}' [
  Cinfinite r; Cnotzero p; p \leq o r ] \Longrightarrow r \hat{\cdot} c p = o r
unfolding \textit{cinfinite-def} \textit{cprod-def}
by (rule \textit{Card-order-Times-infinite}[THEN conjunct1]) (blast intro: czeroI)+
lemma \textit{cprod-infinite}: \( \text{Cinfinite} r \Longrightarrow r \hat{\cdot} c r = o r \)
using \textit{cprod-infinite1} Cinfinite-Cnotzero ordLeq-refl by blast

lemma \textit{cexp-cprod-ordLeq}:
assumes \( r1 : \text{Card-order} r1 \) and \( r2 : \text{Cinfinite} r2 \)
and \( r3 : \text{Cnotzero} r3 r3 \leq o r2 \)
shows \( (r1 \hat{\cdot} c r2) \hat{\cdot} c r3 = o r1 \hat{\cdot} c r2 \) (is \( ?L = o ?R \))
proof –
have \( ?L = o r1 \hat{\cdot} c (r2 \hat{\cdot} c r3) \) using \textit{cexp-cprod}[OF r1] .
also have \( r1 \hat{\cdot} c (r2 \hat{\cdot} c r3) = o ?R \)
apply (rule \textit{cexp-cong2})
apply (rule \textit{cprod-infinite1}'[OF r2 r3]) using r1 r2 by (fastforce simp: \textit{Card-order-cprod})+
finally show \( ?thesis \).
qed

lemma \textit{Cnotzero-UNIV}: Cnotzero \( |\text{UNIV}| \)
by (auto simp: \textit{card-of-Card-order} \textit{card-of-ordIso-czero-iff-empty})

lemma \textit{ordLess-ctwo-cexp}:
assumes \( \text{Card-order} r \)
shows \( r < o ctwo \hat{\cdot} c r \)
proof –
have \( r < o |\text{Pow} (\text{Field} r)| \) using \textit{assms} by (rule \textit{Card-order-Pow})
also have \( |\text{Pow} (\text{Field} r)| = o ctwo \hat{\cdot} c r \)
  unfolding \textit{ctwo-def} \textit{cexp-def} \textit{Field-card-of} by (rule \textit{card-of-Pow-Func})
finally show \( ?thesis \).
qed
lemma ordLeq-cexp1:
  assumes Cnotzero r Card-order q
  shows q ≤ o q c r
proof (cases q = o (czero :: 'a rel))
  case True thus ?thesis by (simp only: card-of-empty cexp-def czero-def ordIso-ordLeq-trans)
next
  case False
  thus ?thesis
  proof -
    apply (rule ordLess-imp-ordLeq)
    apply (rule ordLess-ordLeq-trans)
    apply (rule ordLess-ctwo-cexp)
    apply (rule assms (2))
    apply (rule assms (1))
    apply (rule assms (2))
    apply (rule notE)
    apply (rule cone-not-czero)
    apply assumption
    apply (rule Card-order-cone)
  qed
qed

lemma ordLeq-cexp2:
  assumes ctwo ≤ o q Card-order r
  shows r ≤ o q c r
proof (cases r = o (czero :: 'a rel))
  case True thus ?thesis by (simp only: card-of-empty cexp-def czero-def ordIso-ordLeq-trans)
next
  case False thus ?thesis
  proof -
    apply (rule ordLess-imp-ordLeq)
    apply (rule ordLess-ordLeq-trans)
    apply (rule ordLess-ctwo-cexp)
    apply (rule assms (2))
    apply (rule assms (1))
    apply (rule assms (2))
  qed
qed

lemma cinfinite-cexp: [ctwo ≤ o q; Cinfinite r] ⇒ cinfinite (q c r)
by (rule cinfinite-mono[OF ordLeq-cexp2]) simp-all

lemma Cinfinite-cexp:
  [ctwo ≤ o q; Cinfinite r] ⇒ Cinfinite (q c r)
by (simp add: cinfinite-cexp Card-order-cexp)
lemma ctwo-ordLess-natLeq: ctwo <o natLeq
unfolding ctwo-def using finite-UNIV natLeq-cinfinite natLeq-Card-order
by (intro Cfinite-ordLess-Cinfinite) (auto simp: cfinite-def card-of-Card-order)

lemma ctwo-ordLess-Cinfinite: Cfinite r ⇒ ctwo <o r
by (rule ordLess-ordLeq-trans [OF ctwo-ordLess-natLeq natLeq-ordLeq-cinfinite])

lemma ctwo-ordLeq-Cinfinite:
  assumes Cinfinite r
  shows ctwo ≤o r
by (rule ordLess-imp-ordLeq [OF ctwo-ordLess-Cinfinite [OF assms]])

lemma Un-Cinfinite-bound: [| A |≤ o r; | B |≤ o r; Cinfinite r |] ⇒ | A ∪ B |≤ o r
by (auto simp add: cinfinite-def card-of-Un-ordLeq-infinite-Field)

lemma UNION-Cinfinite-bound: [| | I |≤ o r; ∀ i ∈ I. | A i |≤ o r; Cinfinite r |] ⇒ |
∪ i ∈ I. A i |≤ o r
by (auto simp add: card-of-UNION-ordLeq-infinite-Field cinfinite-def)

lemma csum-cinfinite-bound:
  assumes p ≤o r q ≤o r Card-order p Card-order q Cinfinite r
  shows p +c q ≤o r
proof –
  from assms(1–4) have |Field p| ≤o r |Field q| ≤o r
    unfolding card-order-on-def using card-of-least ordLeq-transitive by blast+
  with assms show ?thesis unfolding cinfinite-def csum-def
  by (blast intro: card-of-Plus-ordLeq-infinite-Field)
qed

lemma cprod-cinfinite-bound:
  assumes p ≤o r q ≤o r Card-order p Card-order q Cinfinite r
  shows p *c q ≤o r
proof –
  from assms(1–4) have |Field p| ≤o r |Field q| ≤o r
    unfolding card-order-on-def using card-of-least ordLeq-transitive by blast+
  with assms show ?thesis unfolding cinfinite-def cprod-def
  by (blast intro: card-of-Times-ordLeq-infinite-Field)
qed

lemma cprod-csum-cexp:
  r1 *c r2 ≤o (r1 +c r2) ^c ctwo
unfolding cprod-def csum-def cexp-def ctwo-def Field-card-of
proof –
  let ᵈf = λ(a, b). %x. if x then Inl a else Inr b
  have inj-on ᵈf (Field r1 × Field r2) (is inj-on - ?LHS)
    by (auto simp: inj-on-def fun-eq-iff split: bool.split)
  moreover
  have ᵈf · ?LHS ⊆ Func (UNIV :: bool set) (Field r1 <+> Field r2) (is - ⊆
ultimately show |LHS| ≤ |RHS| using card-of-ordLeq by blast

lemma Cfinite-cprod-Cinfinite: [Cfinite r; Cinfinite s] → r * c s ≤ o s
by (intro cprod-cinfinite-bound)
(auto intro: ordLeq-refl ordLess-imp-ordLeq[OF Cfinite-ordLess-Cinfinite])

lemma cprod-cexp: (r * c s) ^ c t = o r ^ c t * c s ^ c t
unfolding cprod-def cexp-def Field-card-of by (rule Func-Times-Range)

lemma cprod-cexp-csum-cexp-Cinfinite:
assumes t: Cinfinite t
shows (r * c s) ^ c t ≤ o (r + c s) ^ c t
proof (cases r + c s ≤ o ctwo)
case True
thus ?thesis using t by (blast intro: cexp-mono1)
next
case False
hence ctwo ≤ o s using ordLeq-total[of s ctwo] Card-order-ctwo s
by (auto intro: card-order-on-well-order-on)
hence Cnotzero s using Cnotzero-mono[OF ctwo-Cnotzero] s by blast
hence st: Cnotzero (s * c t) by (intro Cfinite-Cnotzero[OF Cfinite-cprod-Cinfinite])
(auto simp: t)
have (ctwo ^ c s) ^ c t ≤ o (ctwo ^ c s) ^ c t
using assms by (blast intro: cexp-mono1 ordLess-imp-ordLeq[OF ordLess-ctwo-cexp])
also have (ctwo ^ c s) ^ c t = o ctwo ^ c (s * c t)
by (blast intro: Card-order-ctwo cexp-cprod)
also have ctwo ^ c (s * c t) ≤ o ctwo ^ c t
using assms st by (intro cexp-mono2-Cnotzero Cfinite-cprod-Cinfinite Card-order-ctwo)
finally show ?thesis.
qed

lemma csum-Cfinite-cexp-Cinfinite:
  assumes r: Card-order r and s: Cfinite s and t: Cinfinite t
  shows (r + c s) ^c t ≤ o (r + c ctwo) ^c t
proof (cases Cinfinite r)
  case True
  hence r + c s = o r by (intro csum-absorb1 ordLess-imp-ordLeq[OF Cfinite-ordLess-Cinfinite] s)
  hence (r + c s) ^c t = o r ^c t using t by (blast intro: cexp-cong1)
  also have r ^c t ≤ o (r + c ctwo) ^c t using t by (blast intro: cexp-mono1 ordLeq-csum1 r)
  finally show ?thesis .
next
  case False
  with r have Cfinite r unfolding cinfinite-def by auto
  hence Cfinite (r + c s) by (intro Cfinite-csum s)
  hence (r + c s) ^c t ≤ o ctwo ^c t by (intro Cfinite-cexp-Cinfinite t)
  also have ctwo ^c t ≤ o (r + c ctwo) ^c t using t
  by (blast intro: cexp-mono1 ordLeq-csum2 Card-order-ctwo)
  finally show ?thesis .
qed

lemma Cinfinite-cardSuc: Cinfinite r =⇒ Cinfinite (cardSuc r)
  by (simp add: cinfinite-def cardSuc-Card-order cardSuc-finite)

lemma cardSuc-UNION-Cinfinite:
  assumes Cinfinite r relChain (cardSuc r) As B ≤ (⋃ i ∈ Field (cardSuc r). As i) |B| <= o r
  shows ∃ i ∈ Field (cardSuc r). B ≤ As i
  using cardSuc-UNION assms unfolding cinfinite-def by blast

end

30 Function Definition Base

theory Fun-Def-Base
imports Ctr-Sugar Set Wellfounded
begin

ML-file (Tools/Function/function-lib.ML)
ML-file (Tools/Function/function-common.ML)
ML-file (Tools/Function/function-context-tree.ML)

attribute-setup fundef-cong =
  (Attrib.add_del Function-Context-Tree.cong_add Function-Context-Tree.cong_del)
THEORY "BNF-Def"

declaration of congruence rule for function definitions

ML-file ⟨Tools/Function/sum-tree.ML⟩ end

31 Definition of Bounded Natural Functors

theory BNF-Def imports BNF-Cardinal-Arithmetic Fun-Def-Base keywords print-bnfs :: diag and bnf :: thy-goal-defn begin

lemma Collect-case-prodD: \( x \in \text{Collect} (\text{case-prod} \ A) \implies A \ (\text{fst} \ x) (\text{snd} \ x) \)
  by auto

inductive rel-sum :: \((\alpha \Rightarrow \beta \Rightarrow \text{bool}) \Rightarrow (\gamma \Rightarrow \delta \Rightarrow \text{bool}) \Rightarrow \alpha + \gamma \Rightarrow \beta + \delta \Rightarrow \text{bool}\)
  for R1 R2
  where
  R1 a c \implies rel-sum R1 R2 (Inl a) (Inl c)
| R2 b d \implies rel-sum R1 R2 (Inr b) (Inr d)

definition rel-fun :: \((\alpha \Rightarrow \beta \Rightarrow \text{bool}) \Rightarrow (\gamma \Rightarrow \delta \Rightarrow \text{bool}) \Rightarrow \alpha \Rightarrow \gamma \Rightarrow \beta \Rightarrow \delta \Rightarrow \text{bool}\)
  where
  rel-fun A B = (\lambda f g. \forall x y. A x y \rightarrow B (f x) (g y))

lemma rel-funI [intro]:
  assumes \( \forall x y. A x y \implies B (f x) (g y) \)
  shows rel-fun A B f g
  using assms by (simp add: rel-fun-def)

lemma rel-funD:
  assumes rel-fun A B f g and A x y
  shows B (f x) (g y)
  using assms by (simp add: rel-fun-def)

lemma rel-fun-mono:
  \[ \text{rel-fun} X A f g; \forall x y. Y x y \rightarrow X x y; \forall x y. A x y \implies B x y \] \implies \text{rel-fun} Y B f g
  by(simp add: rel-fun-def)

lemma rel-fun-mono':
  \[ \forall x y. Y x y \rightarrow X x y; \forall x y. A x y \implies B x y \] \implies \text{rel-fun} X A f g \rightarrow \text{rel-fun} Y B f g
by (simp add: rel-fun-def)

definition rel-set :: 
  "'a ⇒ 'b ⇒ bool" ⇒ 'a set ⇒ 'b set ⇒ bool
where rel-set R = (λA B. (∀x∈A. ∃y∈B. R x y) ∧ (∀y∈B. ∃x∈A. R x y))

lemma rel-setI:
  assumes  ± x. x ∈ A =⇒ ∃y ∈ B. R x y
  assumes  ± y. y ∈ B =⇒ ∃x ∈ A. R x y
  shows rel-set R A B using assms unfolding rel-set-def by simp

lemma predicate2-transferD:
  [[rel-fun R1 (rel-fun R2 (=)) P Q; a ∈ A; b ∈ B; A ⊆ {(x, y). R1 x y}; B ⊆ {(x, y). R2 x y}] =⇒
  P (fst a) (fst b) ⇔ Q (snd a) (snd b)
unfolding rel-fun-def by (blast dest: Collect-case-prodD)

definition collect where
  collect F x = (⋃f ∈ F. f x)

lemma fstI: x = (y, z) =⇒ fst x = y
  by simp

lemma sndI: x = (y, z) =⇒ snd x = z
  by simp

lemma bijI: 
  ∀x y. (f x = f y) = (x = y);  ∀y. ∃x. y = f x
  =⇒ bij f
unfolding bij-def inj-on-def by auto blast

definition Gr A f = {(a, f a) | a. a ∈ A}

definition Grp A f = (λa b. b = f a ∧ a ∈ A)

definition vimage2p where
  vimage2p f g R = (λx y. R (f x) (g y))

lemma collect-comp: collect F o g = collect ((λf. f o g) o F)
  by (rule ext) (simp add: collect-def)

definition convol ((a, -)/) where
  (f, g) ≡ λa. (f a, g a)

lemma fst-convol: fst o (f, g) = f
  apply (rule ext)
unfolding convol-def by simp

lemma snd-convol: snd o (f, g) = g
  apply (rule ext)
unfolding  convol-def  by  simp

lemma  convol-mem-GrpI:
  \( x \in A \implies (\text{id}, g) x \in (\text{Collect} \ (\text{case-prod} \ (\text{Grp} A g))) \)
  unfolding  convol-def  Grp-def  by  auto

definition  csquare  where
  csquare A f1 f2 p1 p2  \(\iff (\forall \ a \in A. \ f1 \ (p1 \ a) = f2 \ (p2 \ a))\)

lemma  eq-alt: \(=\) = \(\text{Grp} \ UNIV \ \text{id} \)
  unfolding  Grp-def  by  auto

lemma  leq-conversepI: \(R = (=) =\) \(\implies R \leq R^{-1}^{-1} \)
  by  auto

lemma  leq-OOI: \(R = (=) =\) \(\implies R \leq R OO R \)
  by  auto

lemma  OO-Grp-alt: \((\text{Grp} A f)^{-1}^{-1} OO \text{Grp} A g = (\lambda x y. \exists z. z \in A \land f z = x \land g z = y)\)
  unfolding  Grp-def  by  auto

lemma  Grp-UNIV-id: \(f = \text{id} =\) \(\implies (\text{Grp} \ UNIV \ f)^{-1}^{-1} OO \text{Grp} \ UNIV \ f = \text{Grp} \ UNIV \ f \)
  unfolding  Grp-def  by  auto

lemma  Grp-UNIV-idI: \(x = y =\) \(\implies \text{Grp} \ UNIV \ \text{id} \ x \ y \)
  unfolding  Grp-def  by  auto

lemma  Grp-mono: \(A \leq B =\) \(\implies \text{Grp} A f \leq \text{Grp} B f \)
  unfolding  Grp-def  by  auto

lemma  GrpI: \([f x = y; x \in A] =\) \(\implies \text{Grp} A f \ x \ y \)
  unfolding  Grp-def  by  auto

lemma  GrpE: \(\text{Grp} A f \ x \ y \implies ([f x = y; x \in A] \implies R) \implies R \)
  unfolding  Grp-def  by  auto

lemma  Collect-case-prod-Grp-eqD: \(z \in \text{Collect} (\text{case-prod} \ (\text{Grp} A f)) =\) \((f \circ \text{fst}) z = \text{snd} z \)
  unfolding  Grp-def  comp-def  by  auto

lemma  Collect-case-prod-Grp-in: \(z \in \text{Collect} (\text{case-prod} \ (\text{Grp} A f)) =\) \(\text{fst} z \in A \)
  unfolding  Grp-def  comp-def  by  auto

definition  pick-middlep P Q a c = \(\text{SOME} \ b. \ P \ a \ b \land Q \ b \ c\)

lemma  pick-middlep:
  \((P OO Q) \ a \ c = (\text{SOME} \ b. \ P \ a \ (\text{pick-middlep} \ P \ Q \ a \ c) \land Q (\text{pick-middlep} \ P \ Q \ a \ c) \ c)\)
unfolding pick-middlep-def apply(rule someI-ex) by auto

definition fstOp where
  fstOp P Q ac = (fst ac, pick-middlep P Q (fst ac) (snd ac))

definition sndOp where
  sndOp P Q ac = (pick-middlep P Q (fst ac) (snd ac), (snd ac))

lemma fstOp-in: ac ∈ Collect (case-prod (P OO Q)) ⇒ fstOp P Q ac ∈ Collect (case-prod P)
  unfolding fstOp-def mem-Collect-eq
  by (subst (asm) surjective-pairing, unfold prod.case) (erule pick-middlep[THEN conjunct1])

lemma fst-fstOp: fst bc = (fst ◦ fstOp P Q) bc
  unfolding comp-def fstOp-def by simp

lemma snd-sndOp: snd bc = (snd ◦ sndOp P Q) bc
  unfolding comp-def sndOp-def by simp

lemma snd-sndOp-in: ac ∈ Collect (case-prod (P OO Q)) ⇒ sndOp P Q ac ∈ Collect (case-prod Q)
  unfolding sndOp-def mem-Collect-eq
  by (subst (asm) surjective-pairing, unfold prod.case) (erule pick-middlep[THEN conjunct2])

lemma csquare-fstOp-sndOp:
  csquare (Collect (f (P OO Q))) snd fst (fstOp P Q) (sndOp P Q)
  unfolding csquare-def fstOp-def sndOp-def using pick-middlep by simp

lemma snd-fst-flip: snd xy = (fst ◦ (%(x, y). (y, x))) xy
  by (simp split: prod.split)

lemma fst-snd-flip: fst xy = (snd ◦ (%(x, y). (y, x))) xy
  by (simp split: prod.split)

lemma flip-pred: A ⊆ Collect (case-prod (R ^-1)) ⇒ (%(x, y). (y, x)) ' A ⊆ Collect (case-prod R)
  by auto

lemma predicate2-eqD: A = B ⇒ A a b ↔ B a b
  by simp

lemma case-sum-o-inj: case-sum f g ◦ Inl = f case-sum f g ◦ Inr = g
  by auto

lemma map-sum-o-inj: map-sum f g ◦ Inl = Inl ◦ f map-sum f g ◦ Inr = Inr ◦ g
  by auto
lemma card-order-csum-cone-cexp-def:
card-order r \implies (\|A1\| + c \text{ cone}) \ast c r = |\text{Func\ UNIV} (\text{Inl} \cdot A1 \cup \{\text{Inr} \cdot \})| |\text{Field\-csum} \text{ Field\-order}|
unfolding cexp-def cone-def Field-csum Field-card-order by (auto dest: Field-card-order)

lemma If-the-inv-into-in-Func:
\begin{align*}
[\text{inj-on g C}; C \subseteq B \cup \{x\}] \implies \\
(\lambda i. \text{if } i \in g \cdot C \text{ then the-inv-into } C g i \text{ else } x) \in \text{Func\ UNIV} (B \cup \{x\})
\end{align*}
unfolding Func-def by (auto dest: the-inv-into)

lemma If-the-inv-into-f-f:
\begin{align*}
[i \in C; \text{inj-on g g}] \implies \\
((\lambda i. \text{if } i \in g \cdot C \text{ then the-inv-into } C g i \text{ else } x) \circ g) i = id i
\end{align*}
unfolding Func-def by (auto elim: the-inv-into-f-f)

lemma the-inv-f-o-f-id:
\begin{align*}
\text{inj f} \implies \text{the-inv f} \circ f z = id z
\end{align*}
by (simp add: the-inv-f-f)

lemma rel-fun-iff-leq-vimage2p:
\begin{align*}
(\text{rel-fun R S} f g) = (R \leq \text{vimage2p f g S})
\end{align*}
unfolding rel-fun-def vimage2p-def by auto

lemma convol-image-vimage2p:
\begin{align*}
\langle f \circ \text{fst}, g \circ \text{snd}\rangle \subseteq \text{Collect (case-prod (vimage2p f g R))} \subseteq \text{Collect (case-prod R)}
\end{align*}
unfolding vimage2p-def convol-def by auto

lemma vimage2p-Grp:
\begin{align*}
\text{vimage2p f g P} = \text{Grp UNIV f OO P OO (Grp UNIV g)}^{-1}\cdot^{-1}
\end{align*}
unfolding vimage2p-def Grp-def by auto

lemma subst-Pair:
\begin{align*}
P x y \implies a = (x, y) \implies P (\text{fst a}) (\text{snd a})
\end{align*}
by simp

lemma comp-apply-eq:
\begin{align*}
f (g x) = h (k x) \implies (f \circ g) x = (h \circ k) x
\end{align*}
unfolding comp-apply by assumption

lemma refl-ge-eq:
\begin{align*}
(\forall x. R x x) \implies (=) \leq R
\end{align*}
by auto

lemma ge-eq-refl:
\begin{align*}
(=) \leq R \implies R x x
\end{align*}
by auto

lemma reflp-eq:
\begin{align*}
(=) \leq R \implies \text{reflp} R
\end{align*}
by (auto simp: reflp-def fun-eq-iff)

lemma transp-recompp:
\begin{align*}
\text{transp r} \iff r \text{ OO } r \leq r
\end{align*}
by (auto simp: transp-def)
lemma symp-conversep: symp R = (R⁻¹⁻¹ ≤ R)  
  by (auto simp: symp-def fun-eq-iff)

lemma diag-imp-eq-le: (∀x. x ∈ A → R x x) → (∀x y. x ∈ A → y ∈ A → x = y → R x y)  
  by blast

definition eq-onp :: ('a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool  
  where eq-onp R = (λx y. R x ∧ x = y)

lemma eq-onp-Grp: eq-onp P = BNF-Def.Grp (Collect P) id  
  unfolding eq-onp-def Grp-def by auto

lemma eq-onp-to-eq: eq-onp P x y → x = y  
  by (simp add: eq-onp-def)

lemma eq-onp-top-eq-eq: eq-onp top = (=)  
  by (simp add: eq-onp-def)

lemma eq-onp-same-args: eq-onp P x x = P x  
  by (auto simp add: eq-onp-def)

lemma eq-onp-eqD: eq-onp P Q = Q x x → eq-onp Q x y  
  unfolding eq-onp-def by auto

lemma Ball-Collect: Ball A P = (A ⊆ (Collect P))  
  by auto

lemma eq-onp-mono0: ∀x∈A. P x → Q x → ∀x∈A. ∀y∈A. eq-onp P x y → eq-onp Q x y  
  unfolding eq-onp-def by auto

lemma eq-onp-True: eq-onp (λ-. True) = (=)  
  unfolding eq-onp-def by simp

lemma Ball-image-comp: Ball (f ' A) g = Ball A (g o f)  
  by auto

lemma rel-fun-Collect-case-prodD:  
  rel-fun A B f g X ⊆ Collect (case-prod A) x ∈ X → B (((f o fst) x) (g o snd) x)  
  unfolding rel-fun-def by auto

lemma eq-onp-mono-iff: eq-onp P ≤ eq-onp Q ←→ P ≤ Q  
  unfolding eq-onp-def by auto

ML-file ⟨Tools/BNF/bnf-util.ML⟩
ML-file ⟨Tools/BNF/bnf-tactics.ML⟩
ML-file ⟨Tools/BNF/bnf-def-tactics.ML⟩
32 Composition of Bounded Natural Functors

theory BNF-Composition
imports BNF-Def
begin

lemma ssubst-mem: \( [t = s; s \in X] \Rightarrow t \in X \)
  by simp

lemma empty-natural: \( (\lambda -.\{}\) \circ \mathit{f} = \mathit{image} \mathit{g} \circ (\lambda -.\{}\)
  by (rule ext) simp

lemma Union-natural: \( \mathit{Union} \circ \mathit{image} (\mathit{image} \mathit{f}) = \mathit{image} \mathit{f} \circ \mathit{Union} \)
  by (rule ext) (auto simp only: comp-apply)

lemma in-Union-o-assoc: \( x \in (\mathit{Union} \circ \mathit{gset} \circ \mathit{gmap}) \mathit{A} \Rightarrow x \in (\mathit{Union} \circ (\mathit{gset} \circ \mathit{gmap})) \mathit{A} \)
  by (unfold comp-assoc)

lemma comp-single-set-bd:
  assumes fbd-Card-order: \( \text{Card-order} \ fbd \) and
    fset-bd: \( \forall x. |\mathit{fset} \ x| \leq o fbd \) and
    gset-bd: \( \forall x. |\mathit{gset} \ x| \leq o gbd \)
  shows \( |\bigcup (\mathit{fset} ' \mathit{gset} \ x)\| \leq o gbd \ast c fbd \)
  apply simp
  apply (rule ordLeq-transitive)
  apply (rule card-of-UNION-Sigma)
  apply (rule subst SIGMA-CSUM)
  apply (rule ordLeq-transitive)
  apply (rule card-of-Csum-Times')
  apply (rule fbd-Card-order)
  apply (rule ballI)
  apply (rule fset-bd)
  apply (rule ordLeq-transitive)
  apply (rule cprod-mono1)
  apply (rule gset-bd)
  apply (rule ordIso-imp-ordLeq)
  apply (rule ordIso-refl)
  apply (rule Card-order-cprod)
  done

lemma csum-dup: cinfinite \( r \Rightarrow \text{Card-order} r \Rightarrow p + c \mathit{p}' = o r + c r \Rightarrow p + c \mathit{p}' \mathit{= o r} \)
  apply (erule ordIso-transitive)
  apply (frule csum-absorb2')
apply (erule ordLeq-refl)
by simp

lemma cprod-dup: cinfinite r ⇒ Card-order r ⇒ p * c p' = o r * c p ⇒ p * c p' = o r
apply (erule ordIso-transitive)
apply (rule cprod-infinite)
by simp

lemma Union-image-insert: \( \bigcup (f \mapsto \text{insert a } B) = f \cdot a \cup \bigcup (f \mapsto B) \)
by simp

lemma Union-image-empty: A \cup \bigcup (f \mapsto \{\}) = A
by simp

lemma image-o-collect: collect ((\lambda f. image g \circ f) \cdot F) = image g \circ collect F
by (rule ext) (auto simp add: collect-def)

lemma conj-subset-def: A \subseteq \{x. P x \land Q x\} = (A \subseteq \{x. P x\} \land A \subseteq \{x. Q x\})
by blast

lemma UN-image-subset: \( \bigcup (f \mapsto g x) \subseteq X = (g x \subseteq \{x. f x \subseteq X\}) \)
by blast

lemma comp-set-bd-Union-o-collect: |\bigcup (\bigcup (f \cdot x) \cdot X)| \leq o hbd \implies |(\bigcup \circ collect X) x| \leq o hbd
by (unfold comp-apply collect-def) simp

lemma Collect-inj: Collect P = Collect Q ⇒ P = Q
by blast

lemma Grp-fst-snd: (Grp (Collect (case-prod R)) fst)^{-1-1} OO Grp (Collect (case-prod R)) snd = R
unfolding Grp-def fun-eq-iff relcompp.simps by auto

lemma OO-Grp-cong: A = B ⇒ (Grp A f)^{-1-1} OO Grp A g = (Grp B f)^{-1-1}
OO Grp B g
by (rule arg-cong)

lemma vimage2p-relcompp-mono: R OO S \leq T \implies
vimage2p f g R OO vimage2p g h S \leq vimage2p f h T
unfolding vimage2p-def by auto

lemma type-copy-map-cong0: M (g x) = N (h x) ⇒ (f \circ M \circ g) x = (f \circ N \circ h) x
by auto

lemma type-copy-set-bd: (\forall y. |S y| \leq o bd) \implies |(S \circ Rep) x| \leq o bd
by auto
lemma vimage2p-cong: R = S \implies vimage2p f g R = vimage2p f g S
by simp

lemma Ball-comp-iff: (\lambda x. Ball (A x) f) \circ g = (\lambda x. Ball ((A \circ g) x) f)
unfolding o-def by auto

lemma conj-comp-iff: (\lambda x. P x \land Q x) \circ g = (\lambda x. (P \circ g) x \land (Q \circ g) x)
unfolding o-def by auto

context
fixes Rep Abs
assumes type-copy: type-definition Rep Abs UNIV
begin

lemma type-copy-map-id0: M = id \implies Abs \circ M \circ Rep = id
using type-definition.Rep-inverse[OF type-copy] by auto

lemma type-copy-map-comp0: M = M1 \circ M2 \implies f \circ M \circ g = (f \circ M1 \circ Rep) \circ (Abs \circ M2 \circ g)
using type-definition.Abs-inverse[OF type-copy UNIV-I] by auto

lemma type-copy-set-map0: S \circ M = image f \circ S' \implies (S \circ Rep) \circ (Abs \circ M \circ g) = image f \circ (S' \circ g)
using type-definition.Abs-inverse[OF type-copy UNIV-I] by (auto simp: o-def fun-eq-iff)

lemma type-copy-wit: x \in (S \circ Rep) (Abs y) \implies x \in S y
using type-definition.Abs-inverse[OF type-copy UNIV-I] by auto

lemma type-copy-vimage2p-Grp-Rep: vimage2p f Rep (Grp (Collect P) h) =
Grp (Collect (\lambda x. P (f x))) (Abs \circ h \circ f)
unfolding vimage2p-def Grp-def fun-eq-iff

lemma type-copy-vimage2p-Grp-Abs:
\forall h. vimage2p g Abs (Grp (Collect P) h) = Grp (Collect (\lambda x. P (g x))) (Rep \circ h \circ g)
unfolding vimage2p-def Grp-def fun-eq-iff

lemma type-copy-ex-RepI: (\exists b. F b) = (\exists b. F (Rep b))
proof safe
  fix b assume F b
  show \exists b'. F (Rep b')
  proof (rule exI)
    from F b show F (Rep (Abs b)) using type-definition.Abs-inverse[OF type-copy]
THEORY "BNF-Composition"

by auto
qed
qed blast

lemma vimage2p-relcompp-converse:
  vimage2p f g (R\(^{-1}\)) OO S = (vimage2p Rep f R\(^{-1}\)) OO vimage2p Rep g S
unfolding vimage2p-def relcompp.simps conversep.simps fun-eq-iff image-def
by (auto simp: type-copy-ex-RepI)
end

bnf DEADID: 'a
  map: id :: 'a \Rightarrow 'a
  bd: natLeq
  rel: (\_ :: 'a \Rightarrow bool
by (auto simp add: natLeq-card-order natLeq-cinfinite)
definition id-bnf :: 'a \Rightarrow 'a where
  id-bnf \equiv (\lambda x. x)
lemma id-bnf-apply: id-bnf x = x
  unfolding id-bnf-def by simp

bnf ID: 'a
  map: id-bnf :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b
  sets: \lambda x. \{x\}
  bd: natLeq
  rel: id-bnf :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool
  pred: id-bnf :: ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool
unfolding id-bnf-def
apply (auto simp: Grp-def fun-eq-iff relcompp.simps natLeq-card-order natLeq-cinfinite)
apply (rule ordLess-imp-ordLeq[OF finite-ordLess-infinite[OF - natLeq-Well-order]])
apply (auto simp add: Field-card-of Field-natLeq card-of-well-order-on)[\_]
done

lemma type-definition-id-bnf-UNIV: type-definition id-bnf id-bnf UNIV
  unfolding id-bnf-def by unfold-locales auto

ML-file (Tools/BNF/bnf-comp-tactics.ML)
ML-file (Tools/BNF/bnf-comp.ML)

hide-fact
  DEADID.inj-map DEADID.inj-map-strong DEADID.map-comp DEADID.map-cong
  DEADID.map-cong0
  DEADID.map-cong-simp DEADID.map-id DEADID.map-id0 DEADID.map-ident
  DEADID.map-transfer
  DEADID.rel-Grp DEADID.rel-compp DEADID.rel-compp-Grp DEADID.rel-conversep
  DEADID.rel-eq
  DEADID.rel-flip DEADID.rel-map DEADID.rel-monot DEADID.rel-transfer
33 Registration of Basic Types as Bounded Natural Functors

theory Basic-BNFs
imports BNF-Def

begin

inductive-set setl :: 'a + 'b ⇒ 'a set for s :: 'a + 'b where
s = Inl x ⇒ x ∈ setl s
inductive-set setr :: 'a + 'b ⇒ 'b set for s :: 'a + 'b where
s = Inr x ⇒ x ∈ setr s

lemma sum-set-defs[code]:
setl = (λx. case x of Inl z ⇒ {z} | _ ⇒ {})
setr = (λx. case x of Inr z ⇒ {z} | _ ⇒ {})
by (auto simp: fun-eq-iff intro: setl.intros setr.intros elim: setl.cases setr.cases
split: sum.splits)

lemma rel-sum-simps[code, simp]:
rel-sum R1 R2 (Inl a1) (Inl b1) = R1 a1 b1
rel-sum R1 R2 (Inl a1) (Inr b2) = False
rel-sum R1 R2 (Inr a2) (Inl b1) = False
rel-sum R1 R2 (Inr a2) (Inr b2) = R2 a2 b2
by (auto intro: rel-sum.intros elim: rel-sum.cases)

inductive
pred-sum :: ('a ⇒ bool) ⇒ ('b ⇒ bool) ⇒ 'a + 'b ⇒ bool for P1 P2
where
P1 a ⇒ pred-sum P1 P2 (Inl a)
| P2 b ⇒ pred-sum P1 P2 (Inr b)

lemma pred-sum-inject[code, simp]:
pred-sum P1 P2 (Inl a) ⪯ P1 a
pred-sum P1 P2 (Inr b) ⪯ P2 b
by (simp add: pred-sum.simps)+

bnf 'a + 'b
map: map-sum
sets: setl setr
bd: natLeq
wits: \( \text{Inl Inr} \)
rel: \( \text{rel-sum} \)
pred: \( \text{pred-sum} \)

**proof**
- show \( \text{map-sum id id} = \text{id} \) by (rule map-sum.id)

next
fix \( f1 :: 'o \Rightarrow 's \) and \( f2 :: 'p \Rightarrow 't \) and \( g1 :: 's \Rightarrow 'q \) and \( g2 :: 't \Rightarrow 'r \)
show \( \text{map-sum} (g1 \circ f1) (g2 \circ f2) = \text{map-sum} g1 \circ \text{map-sum} f1 \circ f2 \)
  by (rule map-sum.comp[Symmetric])

next
fix \( x \) and \( f1 :: 'o \Rightarrow 'q \) and \( f2 :: 'p \Rightarrow 'r \) and \( g1 \) and \( g2 \)

assume \( a1: \bigwedge z. z \in \text{setl} x \Rightarrow f1 z = g1 z \) and
\( a2: \bigwedge z. z \in \text{setr} x \Rightarrow f2 z = g2 z \)
thus \( \text{map-sum} f1 f2 x = \text{map-sum} g1 g2 x \)
proof (cases \( x \))
  case \( \text{Inl} \) thus \( ?\text{thesis using a1} \) by (clarsimp simp: sum-set-defs(1))
next
  case \( \text{Inr} \) thus \( ?\text{thesis using a2} \) by (clarsimp simp: sum-set-defs(2))
qed

next
fix \( f1 :: 'o \Rightarrow 'q \) and \( f2 :: 'p \Rightarrow 'r \)
show \( \text{setl} \circ \text{map-sum} f1 f2 = \text{image} f1 \circ \text{setl} \)
  by (rule ext, unfold o-apply) (simp add: sum-set-defs(1) split: sum.split)

next
fix \( f1 :: 'o \Rightarrow 'q \) and \( f2 :: 'p \Rightarrow 'r \)
show \( \text{setr} \circ \text{map-sum} f1 f2 = \text{image} f2 \circ \text{setr} \)
  by (rule ext, unfold o-apply) (simp add: sum-set-defs(2) split: sum.split)

next
show \( \text{card-order} \text{ natLeq} \) by (rule natLeq-card-order)
next
show \( \text{cinfinite} \text{ natLeq} \) by (rule natLeq-cinfinite)

next
fix \( x :: 'o + 'p \)
show \( |\text{setl} x| \leq o \text{ natLeq} \)
  apply (rule ordLess-imp-ordLeq)
  apply (rule finite-iff-ordLess-natLeq[THEN iffD1])
  by (simp add: sum-set-defs(1) split: sum.split)

next
fix \( x :: 'o + 'p \)
show \( |\text{setr} x| \leq o \text{ natLeq} \)
  apply (rule ordLess-imp-ordLeq)
  apply (rule finite-iff-ordLess-natLeq[THEN iffD1])
  by (simp add: sum-set-defs(2) split: sum.split)

next
fix \( R1 R2 S1 S2 \)
show \( \text{rel-sum} R1 R2 OO \text{ rel-sum} S1 S2 \leq \text{rel-sum} (R1 OO S1) (R2 OO S2) \)
  by (force elim: rel-sum.cases)

next
fix \( R S \)
show rel-sum $R \ S = (\lambda x \ y. \\ \exists z. (\text{setl } z \subseteq \{(x, y). R \ x \ y\} \land \text{setr } z \subseteq \{(x, y). S \ x \ y\}) \land \text{map-sum } \text{fst } \text{fst } z = x \land \text{map-sum } \text{snd } \text{snd } z = y)\\$

unfolding sum-set-defs relcompp.simps conversep.simps fun-eq-iff
by (fastforce elim: rel-sum.cases split: sum.splits)
qed (auto simp: sum-set-defs fun-eq-iff pred-sum.simps splits)

inductive-set fsts :: $'a \times 'b \Rightarrow 'a$ set
for $p :: 'a \times 'b$
where $\text{fst } p \in \text{fsts } p$

inductive-set snds :: $'a \times 'b \Rightarrow 'b$ set
for $p :: 'a \times 'b$
where $\text{snd } p \in \text{snds } p$

lemma prod-set-defs [code]: $\text{fsts } = (\lambda p. \{\text{fst } p\}) \ \text{snds } = (\lambda p. \{\text{snd } p\})$
by (auto intro: fsts.intros snds.intros elim: fsts.cases snds.cases)

inductive
rel-prod :: $(a \Rightarrow b \Rightarrow \text{bool}) \Rightarrow (c \Rightarrow d \Rightarrow \text{bool}) \Rightarrow a \times c \Rightarrow b \times d \Rightarrow \text{bool}$
for $R1 \ R2$
where $[R1 \ a \ b; R2 \ c \ d] \implies \text{rel-prod } R1 \ R2 \ (a, c) \ (b, d)\

inductive
pred-prod :: $(a \Rightarrow \text{bool}) \Rightarrow (b \Rightarrow \text{bool}) \Rightarrow a \times b \Rightarrow \text{bool}$ for $P1 \ P2$
where $[P1 \ a; P2 \ b] \implies \text{pred-prod } P1 \ P2 \ (a, b)\

lemma rel-prod-inject [code, simp]:
rel-prod $R1 \ R2 \ (a, b) \ (c, d) \iff R1 \ a \ c \ \land \ R2 \ b \ d$
by (auto intro: rel-prod.intros elim: rel-prod.cases)

lemma pred-prod-inject [code, simp]:
pred-prod $P1 \ P2 \ (a, b) \iff P1 \ a \ \land \ P2 \ b$
by (auto intro: pred-prod.intros elim: pred-prod.cases)

lemma rel-prod-conv:
rel-prod $R1 \ R2 = (\lambda(a, b) \ (c, d). R1 \ a \ c \ \land \ R2 \ b \ d)$
by (rule ext, rule ext) auto

definition
pred-fun :: $(a \Rightarrow \text{bool}) \Rightarrow (b \Rightarrow \text{bool}) \Rightarrow (a \Rightarrow 'b) \Rightarrow \text{bool}$
where $\text{pred-fun } A \ B = (\lambda f. \forall x. \ A \ x \ \rightarrow \ B \ (f \ x))$

lemma pred-funI: $(\forall x. \ A \ x \ \Rightarrow \ B \ (f \ x)) \Longrightarrow \text{pred-fun } A \ B \ f$
unfolding pred-fun-def by simp

bnf $'a \times 'b$
map: map-prod
sets: fsts snds
bd: natLeq
rel: rel-prod
prod: pred-prod

proof (unfold prod-set-defs)
  show map-prod id id = id by (rule map-prod.id)
next
fix f1 f2 g1 g2
  show map-prod (g1 o f1) (g2 o f2) = map-prod g1 g2 o map-prod f1 f2
    by (rule map-prod.comp[symmetric])
next
fix x f1 f2
  assume \( \forall z \in \{ \text{fst } x \} \Rightarrow f1 z = g1 z \ \ \ \forall z \in \{ \text{snd } x \} \Rightarrow f2 z = g2 z \)
  thus map-prod f1 f2 x = map-prod g1 g2 x by (cases x) simp
next
fix x f1 f2
  show (\( \lambda x. \{ \text{fst } x \} \)) o map-prod f1 f2 = image f1 o (\( \lambda x. \{ \text{fst } x \} \))
    by (rule ext, unfold o-apply) simp
next
fix x f1 f2
  show (\( \lambda x. \{ \text{snd } x \} \)) o map-prod f1 f2 = image f2 o (\( \lambda x. \{ \text{snd } x \} \))
    by (rule ext, unfold o-apply) simp
next
show card-order natLeq by (rule natLeq-card-order)
next
show cinfinite natLeq by (rule natLeq-cinfinite)
next
fix x
  show |\{ \text{fst } x \}| \leq o natLeq
    by (rule ordLess-imp-ordLeq) (simp add: finite-iff-ordLess-natLeq[symmetric])
next
fix x
  show |\{ \text{snd } x \}| \leq o natLeq
    by (rule ordLess-imp-ordLeq) (simp add: finite-iff-ordLess-natLeq[symmetric])
next
fix R1 R2 S1 S2
  show rel-prod R1 R2 OO rel-prod S1 S2 \leq rel-prod (R1 OO S1) (R2 OO S2)
    by auto
next
fix R S
  show rel-prod R S = (\( \lambda x. y. \))
    \( \exists z. (\{ \text{fst } z \} \subseteq \{ (x, y). R x y \} \wedge \{ \text{snd } z \} \subseteq \{ (x, y). S x y \}) \wedge \)
      map-prod fst fst z = x \wedge map-prod snd snd z = y)
    unfolding prod-set-defs rel-prod-inject relcompp.simps conversep.simps fun-eq-iff
    by auto
  qed auto

bnf 'a \Rightarrow 'b
  map: (\( o \))
  sets: range
THEORY “BNF-Fixpoint-Base”

bd: natLeq + c | UNIV :: 'a set|
rel: rel-fun (=)
pred: pred-fun (λ-. True)

proof
  fix f show id o f = id f by simp
next
  fix f g show (o) (g o f) = (o) g o (o) f
  unfolding comp-def[abs-def] ..
next
  fix x f g
  assume ⋀z. z ∈ range x =⇒ f z = g z
  thus f o x = g o x by auto
next
  show range o (o) f = (') f o range
    by (auto simp add: fun-eq-iff)
next
  show card-order (natLeq + c | UNIV |) (is - (+c ?U))
    apply (rule card-order-csum)
    apply (rule natLeq-card-order)
    by (rule card-of-card-order-on)
  
  show cinfinite (natLeq + c ?U)
    apply (rule cinfinite-csum)
    apply (rule disjI1)
    by (rule natLeq-cinfinite)
next
  fix f :: 'd =⇒ 'a
  have |range f| ≤ o | (UNIV::'d set) | (is - ≤o ?U) by (rule card-of-image)
  also have ?U ≤ o natLeq + c ?U by (rule ordLeq-csum2) (rule card-of-Card-order)
  finally show |range f| ≤ o natLeq +c ?U .
next
  fix R S
  show rel-fun (=) R OO rel-fun (=) S ≤ rel-fun (=) (R OO S) by (auto simp: rel-fun-def)
next
  fix R
  show rel-fun (=) R = (λx y. ∃z. range z ⊆ {(x, y). R x y} ∧ fst o z = x ∧ snd o z = y)
  unfolding rel-fun-def subset-iff by (force simp: fun-eq-iff[symmetric])
qed (auto simp: pred-fun-def)

end

34 Shared Fixpoint Operations on Bounded Natural Functors

theory BNF-Fixpoint-Base
imports BNF-Composition Basic-BNFs
begin

lemma conj-imp-eq-imp: \((P \land Q \implies PROP \, R) \equiv (P \implies Q \implies PROP \, R)\)
by standard simp-all

lemma predicate2D-conj: \(P \leq Q \land R \implies R \land (P \, x \, y \implies Q \, x \, y)\)
by blast

lemma eq-sym-Unity-conv: \((x = (() = ())) = x\)
by blast

lemma case-unit-Unity: \((case \, u \, of \, () \implies f) = f\)
by (cases u) (hypsubst, rule unit.case)

lemma case-prod-Pair-iden: \((case \, p \, of \, (x, y) \implies (x, y)) = p\)
by simp

lemma unit-all-impI: \((P () \implies Q ()) \implies \forall x. P \, x \implies Q \, x\)
by simp

lemma pointfree-idE:
\(f \circ g = id \implies f (g \, x) = x\)
unfolding comp-def fun-eq-iff by simp

lemma o-bij:
assumes gf: \(g \circ f = id\) and fg: \(f \circ g = id\)
shows bij f
unfolding bij-def inj-on-def surj-def proof safe
fix a1 a2 assume \(f \, a1 = f \, a2\)
hence \(g \, (f \, a1) = g \, (f \, a2)\) by simp
thus \(a1 = a2\) using gf unfolding fun-eq-iff by simp
next
fix b
have \(b = f \, (g \, b)\)
using fg unfolding fun-eq-iff by simp
thus \(\exists a. b = f \, a\) by blast
qed

lemma case-sum-step:
case-sum (case-sum \(f' \, g') \, (Inl \, p)\) = case-sum \(f' \, g' \, p\)
case-sum f (case-sum \(f' \, g') \, (Inr \, p)\) = case-sum \(f' \, g' \, p\)
by auto

lemma obj-one-pointE: \(\forall x. s = x \implies P \implies P\)
by blast

lemma type-copy-obj-one-point-absE:
assumes type-definition Rep Abs UNIV \(\forall x. s = Abs \, x \implies P\)
shows P
using type-definition.Rep-inverse[OF assms(1)]
by (intro mp[OF spec[OF assms(2), of Rep s]]) simp
lemma obj-sumE-f:
    assumes ∀ x. s = f (Inl x) → P ∀ x. s = f (Inr x) → P
    shows ∀ x. s = f x → P
proof
    fix x from assms show s = f x → P by (cases x) auto
qed

lemma case-sum-if:
    case-sum f g (if p then Inl x else Inr y) = (if p then f x else g y)
    by simp

lemma prod-set-simps[simp]:
    fsts (x, y) = {x}
    snds (x, y) = {y}
    unfolding prod-set-defs by simp+

lemma sum-set-simps[simp]:
    setl (Inl x) = {x}
    setl (Inr x) = {}
    setr (Inl x) = {x}
    setr (Inr x) = {}
    unfolding sum-set-defs by simp+

lemma Inl-Inr-False: (Inl x = Inr y) = False
    by simp

lemma Inr-Inl-False: (Inr x = Inl y) = False
    by simp

lemma spec2: ∀ x y. P x y → P x y
    by blast

lemma rewriteR-comp-comp: [g ∘ h = r] → f ∘ g ∘ h = f ∘ r
    unfolding comp-def fun-eq-iff by auto

lemma rewriteR-comp-comp2: [g ∘ h = r1 ∘ r2; f ∘ r1 = l] → f ∘ g ∘ h = l ∘ r2
    unfolding comp-def fun-eq-iff by auto

lemma rewriteL-comp-comp: [f ∘ g = l] → f ∘ (g ∘ h) = l ∘ h
    unfolding comp-def fun-eq-iff by auto

lemma rewriteL-comp-comp2: [f ∘ g = l1 ∘ l2; l2 ∘ h = r] → f ∘ (g ∘ h) = l1 ∘ r
    unfolding comp-def fun-eq-iff by auto

lemma convol-o: (f, g) ∘ h = (f ∘ h, g ∘ h)
    unfolding convol-def by auto
lemma map-prod-o-convol: map-prod h1 h2 o (f, g) = (h1 o f, h2 o g)
  unfolding convol-def by auto

lemma map-prod-o-convol-id: (map-prod f id o (id, g)) x = (id o f, g) x
  unfolding map-prod-o-convol id-comp comp-id ..

lemma o-case-sum: h o case-sum f g = case-sum (h o f) (h o g)
  unfolding comp-def by (auto split: sum.splits)

lemma case-sum-o-map-sum: case-sum f g o map-sum h1 h2 = case-sum (f o h1)
  (g o h2)
  unfolding comp-def by (auto split: sum.splits)

lemma case-sum-o-map-sum-id: (case-sum id g o map-sum f id) x = case-sum (f o id)
  g x
  unfolding case-sum-o-map-sum id-comp comp-id ..

lemma rel-fun-def-butlast:
  rel-fun R (rel-fun S T) f g = (∀ x y. R x y → (rel-fun S T) (f x) (g y))
  unfolding rel-fun-def ..

lemma subst-eq-imp: (∀ a b. a = b → P a b) ≡ (∀ a. P a a)
  by auto

lemma eq-subset: (=) ≤ (λa b. P a b ∨ a = b)
  by auto

lemma eq-le-Grp-id-iff: ((=) ≤ Grp (Collect R) id) = (All R)
  unfolding Grp-def id-apply by blast

lemma Grp-id-mono-subst: (∀x y. Grp P id x y → Grp Q id (f x) (f y)) ≡
  (∀x. x ∈ P → f x ∈ Q)
  unfolding Grp-def by rule auto

lemma vimage2p-mono: vimage2p f g R x y → R ≤ S → vimage2p f g S x y
  unfolding vimage2p-def by blast

lemma vimage2p-refl: (∀x. R x x) → vimage2p f f R x x
  unfolding vimage2p-def by auto

lemma
  assumes type-definition Rep Abs UNIV
  shows type-copy-Rep-o-Abs: Rep o Abs = id and type-copy-Abs-o-Rep: Abs o Rep = id
  unfolding fun-eq-iff comp-apply id-apply
lemma type-copy-map-comp0-undo:
assumes type-definition Rep Abs UNIV
type-definition Rep′ Abs′ UNIV
type-definition Rep″ Abs″ UNIV
shows Abs′ ∘ M ∘ Rep″ = (Abs′ ∘ M1 ∘ Rep) ∘ (Abs ∘ M2 ∘ Rep″) ⇒ M1 ∘ M2 = M
by (rule sym) (auto simp: fun-eq-iff type-definition.Abs-inject[OF assms(2) UNIV-I UNIV-I]
  type-definition.Abs-inverse[OF assms(1) UNIV-I]
  type-definition.Abs-inverse[OF assms(3) UNIV-I] dest: spec[of - Abs″ x for x])

lemma vimage2p-id: vimage2p id id R = R
unfolding vimage2p-def by auto

lemma vimage2p-comp: vimage2p (f1 ∘ f2) (g1 ∘ g2) = vimage2p f2 g2 ∘ vimage2p f1 g1
unfolding fun-eq-iff vimage2p-def o-apply by simp

lemma vimage2p-rel-fun: rel-fun (vimage2p f g R) R f g
unfolding rel-fun-def vimage2p-def by auto

lemma fun-cong-unused-0: f = (λx. g) ⇒ f (λx. 0) = g
by (erule arg-cong)

lemma inj-on-convol-ident: inj-on (λx. (x, f x)) X
unfolding inj-on-def by simp

lemma map-sum-if-distrib-then:
  ∀ g e x y. map-sum f g (if e then Inl x else y) = (if e then Inl (f x) else map-sum f g y)
  ∀ f e x y. map-sum f g (if e then Inr x else y) = (if e then Inr (g x) else map-sum f g y)
by simp-all

lemma map-sum-if-distrib-else:
  ∀ g e x y. map-sum f g (if e then x else Inl y) = (if e then map-sum f g x else Inl (f y))
  ∀ f e x y. map-sum f g (if e then x else Inr y) = (if e then map-sum f g x else Inr (g y))
by simp-all

lemma case-prod-app: case-prod f x y = case-prod (λl r. f l r y) x
by (cases x) simp

lemma case-sum-map-sum: case-sum l r (map-sum f g x) = case-sum (l ∘ f) (r ∘ g) x
by (cases x) simp-all

lemma case-sum-transfer:
rel-fun (rel-fun R T) (rel-fun (rel-fun S T) (rel-fun (rel-sum R S) T)) case-sum
unfolding rel-fun-def by (auto split: sum.splits)

lemma case-prod-map-prod: case-prod h (map-prod f g x) = case-prod (λl r. h (f l) (g r)) x
by (cases x) simp-all

lemma case-prod-o-map-prod: case-prod f o map-prod g1 g2 = case-prod (λl r. f (g1 l) (g2 r))
unfolding comp-def by auto

lemma case-prod-transfer:
(rel-fun (rel-fun A (rel-fun B C)) (rel-fun (rel-fun A B) C)) case-prod case-prod
unfolding rel-fun-def by simp

lemma eq-ifI: (P → t = u1) ⇒ (¬ P → t = u2) ⇒ t = (if P then u1 else u2)
by simp

lemma comp-transfer:
rel-fun (rel-fun B C) (rel-fun (rel-fun A B) (rel-fun A C)) (o) (o)
unfolding rel-fun-def by simp

lemma If-transfer: rel-fun (=) (rel-fun A (rel-fun A A)) If If
unfolding rel-fun-def by simp

lemma Abs-transfer:
assumes type-copy1: type-definition Rep1 Abs1 UNIV
assumes type-copy2: type-definition Rep2 Abs2 UNIV
shows rel-fun R (vimage2p Rep1 Rep2 R) Abs1 Abs2
unfolding vimage2p-def rel-fun-def
type-definition.Abs-inverse[OF type-copy1 UNIV-I]
type-definition.Abs-inverse[OF type-copy2 UNIV-I] by simp

lemma Inl-transfer:
rel-fun S (rel-sum S T) Inl Inl
by auto

lemma Inr-transfer:
rel-fun T (rel-sum S T) Inr Inr
by auto

lemma Pair-transfer: rel-fun A (rel-fun B (rel-prod A B)) Pair Pair
unfolding rel-fun-def by simp

lemma eq-omp-live-step: x = y ⇒ eq-omp P a a ∧ x ←→ P a ∧ y
by (simp only: eq-omp-same-args)
lemma top-conj: top x ∧ P ⟷ P ∧ top x ⟷ P
  by blast+

lemma fst-convol': fst (⟨f, g⟩ x) = f x
  using fst-convol unfolding convol-def by simp

lemma snd-convol': snd (⟨f, g⟩ x) = g x
  using snd-convol unfolding convol-def by simp

lemma convol-expand-snd: fst ◦ f = g =⇒ ⟨g, snd ◦ f⟩ = f
  unfolding convol-def fun-eq-iff
  ultimately show ?thesis by simp
  qed

lemma case-sum-expand-Inr-pointfree: f ◦ Inl = g =⇒ case-sum g (f ◦ Inr) = f
  by (auto split: sum.splits)

lemma case-sum-expand-Inr': f ◦ Inl = g =⇒ h = f ◦ Inr =⇒ case-sum g h = f
  by (rule iffI) (auto simp add: fun-eq-iff split: sum.splits)

lemma case-sum-expand-Inr: f ◦ Inl = g =⇒ f x = case-sum g (f ◦ Inr) x
  by (auto split: sum.splits)

lemma id-transfer: rel-fun A A id id
  unfolding rel-fun-def by simp

lemma fst-transfer: rel-fun (rel-prod A B) A fst
  unfolding rel-fun-def by simp

lemma snd-transfer: rel-fun (rel-prod A B) B snd snd
  unfolding rel-fun-def by simp

lemma convol-transfer:
  rel-fun (rel-fun R S) (rel-fun (rel-fun R T) (rel-fun R (rel-prod S T))) BNF-Def.convol
  unfolding rel-fun-def convol-def by auto

lemma Let-const: Let x (λ-. c) = c
  unfolding Let-def ..
ML-file ⟨Tools/BNF/bnf-fp-util-tactics.ML⟩
ML-file ⟨Tools/BNF/bnf-fp-util.ML⟩
ML-file ⟨Tools/BNF/bnf-fp-def-sugar-tactics.ML⟩
ML-file ⟨Tools/BNF/bnf-fp-def-sugar.ML⟩
ML-file ⟨Tools/BNF/bnf-fp-n2m-tactics.ML⟩
ML-file ⟨Tools/BNF/bnf-fp-n2m.ML⟩
ML-file ⟨Tools/BNF/bnf-fp-n2m-sugar.ML⟩
end

35 Least Fixpoint (Datatype) Operation on Bounded Natural Functors

theory BNF-Least-Fixpoint
imports BNF-Fixpoint-Base
keywords
datatype :: thy-defn and
datatype-compat :: thy-defn
begin

lemma subset-emptyI: \( \forall x. x \in A \implies False \) \implies A \subseteq \{ \}
  by blast

lemma image-Collect-subsetI: \( \forall x. P x \implies f x \in B \) \implies f \{ x. P x \} \subseteq B
  by blast

lemma Collect-restrict: \{ x. x \in X \land P x \} \subseteq X
  by auto

lemma prop-restrict: \[ [ x \in Z; Z \subseteq \{ x. x \in X \land P x \} ] ] \implies P x
  by auto

lemma underS-I: \[ [ i \neq j; (i, j) \in R ] \] \implies i \in underS R j
  unfolding underS-def by simp

lemma underS-E: i \in underS R j \implies i \neq j \land (i, j) \in R
  unfolding underS-def Field-def by auto

lemma underS-Field: i \in underS R j \implies i \in Field R
  unfolding underS-def Field-def by auto

lemma bij-betwE: bij-betw f A B \implies \forall a \in A. f a \in B
  unfolding bij-betw-def by auto

lemma f-the-inv-into-f-bij-betw:
  bij-betw f A B \implies (bij-betw f A B \implies x \in B) \implies f (the-inv-into A f x) = x
  unfolding bij-betw-def by (blast intro: f-the-inv-into-f)
lemma \texttt{ex-bij-betw}: $|A| \leq o (r :: 'b rel) \rightarrow \exists f B :: 'b set. \ bij-betw f B A$
by \texttt{(subst (asm) internalize-card-of-ordLeq) (auto dest: iffD2[OF card-of-ordIso ordIso-symmetric])}

lemma \texttt{bij-betwI}:
$\begin{array}{l}
\forall \ x \ y. \ x \in X; \ y \in X \rightarrow (f x = f y) = (x = y);\\
\forall y \ \in Y \rightarrow \exists x \in X. \ y = f x \rightarrow \bij-betw f X Y
\end{array}$

unfolding \texttt{bij-betw-def inj-on-def} \by \texttt{blast}

lemma \texttt{surj-fun-eq}:
assumes \texttt{surj-on}: $f \ X = UNIV$ and \texttt{eq-on}: $\forall x \ \in X. \ g1 \circ f \ x = (g2 \circ f) \ x$
shows $g1 = g2$
proof (rule ext)
fix $y$
from \texttt{surj-on} obtain $x$ \where $x \in X$ and $y = f x$ \by \texttt{blast}
thus $g1 \ y = g2 \ y$ using \texttt{eq-on} \by \texttt{simp}
qed

lemma \texttt{Card-order-wo-rel}: $\texttt{Card-order r \rightarrow wo-rel r}$
unfolding \texttt{wo-rel-def card-order-on-def} \by \texttt{blast}

lemma \texttt{Cinfinite-limit}:
$\begin{array}{l}
\forall x \ \in \ \text{Field r}; \ \text{Cinfinite r} \rightarrow \exists y \ \in \ \text{Field r}. \ x \neq y \land (x, y) \in r
\end{array}$

unfolding \texttt{cinfinite-def} \by \texttt{(auto simp add: infinite-Card-order-limit)}

lemma \texttt{Card-order-trans}:
$\begin{array}{l}
\forall x, y, z \ \in \ \text{Field r}; \ \text{Card-order r}; \ x \neq y; \ \text{trans r}; \ (x, y) \in r; \ y \neq z; \ (y, z) \in r \rightarrow x \neq z \land (x, z) \in r
\end{array}$

unfolding \texttt{card-order-on-def well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def trans-def antisym-def} \by \texttt{blast}

lemma \texttt{Cinfinite-limit2}:
assumes \texttt{x1: x1 \in Field r and x2: x2 \in Field r and r: Cinfinite r}
shows $\exists y \ \in \ \text{Field r}. \ (x1 \neq y \land (x1, y) \in r) \land (x2 \neq y \land (x2, y) \in r)$
proof --
from \texttt{r} \have \texttt{trans: trans r and total: Total r and antisym: antisym r}
unfolding \texttt{card-order-on-def well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def antisym-def} \by \texttt{auto}
obtain $y1$ \where $y1 = y1$ \in \text{Field r} \ x1 \neq y1 \ (x1, y1) \in r$
using \texttt{Cinfinite-limit[OF x1 r]} \by \texttt{blast}
obtain $y2$ \where $y2 = y2$ \in \text{Field r} \ x2 \neq y2 \ (x2, y2) \in r$
using \texttt{Cinfinite-limit[OF x2 r]} \by \texttt{blast}
show \texttt{?thesis}
proof (cases $y1 = y2$)
case True with $y1 \ y2$ \show \texttt{?thesis} \by \texttt{blast}
next
case False
with $y1(1) \ y2(1)$ \total have $y1 \ y2 \in r \lor (y2, y1) \in r$
unfolding total-on-def by auto
thus ?thesis
proof
  assume *: \((y_1, y_2) \in r\)
  with trans y1(3) have \((x_1, y_2) \in r\) unfolding trans-def by blast
  with False y1 y2 \(*\) antisym show ?thesis by (cases x1 = y2) (auto simp: antisym-def)
next
  assume *: \((y_2, y_1) \in r\)
  with trans y2(3) have \((x_2, y_1) \in r\) unfolding trans-def by blast
  with False y1 y2 \(*\) antisym show ?thesis by (cases x2 = y1) (auto simp: antisym-def)
qed

lemma Cinfinite-limit-finite:
\[
\begin{align*}
\text{finite } X & \subseteq \text{Field } r; \text{Cinfinite } r \implies \exists y \in \text{Field } r. \forall x \in X. (x \neq y \land (x, y) \\
& \in r)
\end{align*}
\]
proof (induct X rule: finite-induct)
case empty thus ?case unfolding cinfinite-def using ex-in-conv[of Field r]
finite.emptyI by auto
next
case (insert x X)
  then obtain y where y: \(y \in \text{Field } r. \forall x \in X. (x \neq y \land (x, y) \in r)\) by blast
then obtain z where z: \(z \in \text{Field } r. \forall x \in X. (x \neq z \land (x, z) \in r)\)
  unfolding insert-def using Cinfinite-limit2[OF - y(1) insert(5), of x] insert(4) by blast
show ?case
  apply (intro bexI ballI)
  apply (erule insertE)
  apply hypsubst
  apply (rule z(2))
  using Card-order-trans[OF insert(5) THEN conjunct2] y(2) z(3)
  apply blast
  apply (rule z(1))
  done
qed

lemma insert-subsetI: \(\{x \in A; X \subseteq A\} \implies \text{insert } x X \subseteq A\)
by auto

lemmas well-order-induct-imp = wo-rel.well-order-induct[of r \(\lambda x. x \in \text{Field } r \longrightarrow P x\) for r P]

lemma meta-spec2:
  assumes \((\forall x y. \text{PROP } P x y)\)
  shows \(\text{PROP } P x y\)
  by (rule assms)
lemma nchotomy-reacomppE:
assumes \( \forall y. \exists x. y = f x (r OO s) a c \quad \forall b. r a (f b) = s (f b) c \quad \Rightarrow \quad P \)
shows \( P \)
proof (rule reacompp.cases[OF assms(2)], hypsubst)
fix \( b \)
assume \( r a b s b c \)
moreover from assms(1) obtain \( b' = b \) where \( b = f b' \) by blast
ultimately show \( P \) by (blast intro: assms(3))
qed

lemma predicate2D-vimage2p: \[
\begin{align*}
R & \leq \text{vimage2p} f g S ; \quad R x y \quad \Rightarrow \quad S (f x) (g y)
\end{align*}
\]
unfolding vimage2p-def by auto

lemma ssubst-Pair-rhs: \[
\begin{align*}
(r, s) & \in R ; \quad s' = s \quad \Rightarrow \quad (r, s') \in R
\end{align*}
\]
by (rule ssubst)

lemma all-mem-range1: \[
\begin{align*}
(\forall y. y \in \text{range} f \quad \Rightarrow \quad P y) & \equiv (\exists x. P (f x))
\end{align*}
\]
by (rule equal-intr-rule) fast+

lemma all-mem-range2: \[
\begin{align*}
(\forall f a. f a \in \text{range} f \quad \Rightarrow \quad y \in \text{range} fa \quad \Rightarrow \quad P y) & \equiv (\exists x a. P (f x xa))
\end{align*}
\]
by (rule equal-intr-rule) fast+

lemma all-mem-range3: \[
\begin{align*}
(\forall f a f b. f a \in \text{range} f \quad \Rightarrow \quad f b \in \text{range} fa \quad \Rightarrow \quad y \in \text{range} fb \quad \Rightarrow \quad P y) & \equiv (\exists x a x b. P (f x xa xb))
\end{align*}
\]
by (rule equal-intr-rule) fast+

lemma all-mem-range4: \[
\begin{align*}
(\forall f a f b f c. f a \in \text{range} f \quad \Rightarrow \quad f b \in \text{range} fa \quad \Rightarrow \quad f c \in \text{range} fb \quad \Rightarrow \quad y \in \text{range} fc \quad \Rightarrow \quad P y) & \equiv (\exists x a x b x c. P (f x xa xb xc))
\end{align*}
\]
by (rule equal-intr-rule) fast+

lemma all-mem-range5: \[
\begin{align*}
(\forall f a f b f c f d. f a \in \text{range} f \quad \Rightarrow \quad f b \in \text{range} fa \quad \Rightarrow \quad f c \in \text{range} fb \quad \Rightarrow \quad f d \in \text{range} fc \quad \Rightarrow \quad y \in \text{range} fd \quad \Rightarrow \quad P y) & \equiv (\exists x a x b x c x d. P (f x xa xb xc xd))
\end{align*}
\]
by (rule equal-intr-rule) fast+

lemma all-mem-range6: \[
\begin{align*}
(\forall f a f b f c f d f e f f. f a \in \text{range} f \quad \Rightarrow \quad f b \in \text{range} fa \quad \Rightarrow \quad f c \in \text{range} fb \quad \Rightarrow \quad f d \in \text{range} fc \quad \Rightarrow \quad f e \in \text{range} fd \quad \Rightarrow \quad f f \in \text{range} fe \quad \Rightarrow \quad y \in \text{range} ff \quad \Rightarrow \quad P y) & \equiv (\exists x a x b x c x d x e x f. P (f x xa xb xc xd xe xf))
\end{align*}
\]
by (rule equal-intr-rule) (fastforce, fast)

lemma all-mem-range7:
\[(\forall fa \ fb \ fc \ fd \ fe \ ff \ fg \ y. \ fa \in \text{range } f \implies fb \in \text{range } fa \implies fc \in \text{range } fb \implies fd \\
\implies fe \in \text{range } fd \implies ff \in \text{range } fe \implies fg \in \text{range } ff \implies y \in \text{range } fg \implies P \ y)\]

\[
\equiv (\forall x \ xa \ xb \ xc \ xd \ xe \ xf \ xg \ y. \ f(x \ xa \ xb \ xc \ xd \ xe \ xf \ xg \ xh) \ P) \]

\[
\text{by (rule equal-intr-rule) (fastforce, fast)}
\]

\textbf{Lemma} \textit{all-mem-range8}:

\[(\forall fa \ fb \ fc \ fd \ fe \ ff \ fg \ fh \ y. \ fa \in \text{range } f \implies fb \in \text{range } fa \implies fc \in \text{range } fb \implies fd \\
\implies fe \in \text{range } fd \implies ff \in \text{range } fe \implies fg \in \text{range } ff \implies fh \in \text{range } fg \implies y \in \text{range } fh \implies P \ y)\]

\[
\equiv (\forall x \ xa \ xb \ xc \ xd \ xe \ xf \ xg \ xh \ y. \ f(x \ xa \ xb \ xc \ xd \ xe \ xf \ xg \ xh) \ P) \]

\[
\text{by (rule equal-intr-rule) (fastforce, fast)}
\]

\textbf{Lemmas} \textit{all-mem-range} = \textit{all-mem-range1 all-mem-range2 all-mem-range3 all-mem-range4 all-mem-range5 all-mem-range6 all-mem-range7 all-mem-range8}

\textbf{Lemma} \textit{pred-fun-True-id}: \textit{NO-MATCH} \text{ id } p \implies \textit{pred-fun} (\lambda x. \text{ True}) \ p f \equiv \textit{pred-fun} \\
(\lambda x. \text{ True}) \ido (p \circ f) \]

\textit{unfolding fun.pred-map unfolding comp-def id-def ..}

\textbf{ML-file} ⟨Tools/BNF/bnf-lfp-util.ML⟩
\textbf{ML-file} ⟨Tools/BNF/bnf-lfp-tactics.ML⟩
\textbf{ML-file} ⟨Tools/BNF/bnf-lfp.ML⟩
\textbf{ML-file} ⟨Tools/BNF/bnf-lfp-compat.ML⟩
\textbf{ML-file} ⟨Tools/BNF/bnf-lfp-rec-sugar-more.ML⟩
\textbf{ML-file} ⟨Tools/BNF/bnf-lfp-size.ML⟩

end

\textbf{Theory} Basic-BNF-LFPs
\textbf{Imports} BNF-Least-Fixpoint
\begin{flushleft}
begin
\end{flushleft}

\textbf{Definition} \textit{xtor} :: ' \textit{a} \Rightarrow ' \textit{a} \ where
\textit{xtor} \& \textit{x} = \textit{x}

\textbf{Lemma} \textit{xtor-map}: \textit{f} (\textit{xtor} \textit{x}) = \textit{xtor} (f \textit{x})
\textit{unfolding} \textit{xtor-def} \textbf{by} \ (\textit{rule refl})

\textbf{Lemma} \textit{xtor-map-unique}: \textit{u} \circ \textit{xtor} = \textit{xtor} \circ \textit{f} \implies \textit{u} = \textit{f}
\textit{unfolding} \textit{op-def} \textit{xtor-def} .

\textbf{Lemma} \textit{xtor-set}: \textit{f} (\textit{xtor} \textit{x}) = \textit{f} \textit{x}
\textit{unfolding} \textit{xtor-def} \textbf{by} \ (\textit{rule refl})
lemma xtor-rel: \( R (\text{xtor} x) (\text{xtor} y) = R x y \)
unfolding xtor-def by (rule refl)

lemma xtor-induct: \((\forall x. P (\text{xtor} x)) \Longrightarrow P z\)
unfolding xtor-def by assumption

lemma xtor-xtor: \(\text{xtor} (\text{xtor} x) = x\)
unfolding xtor-def by (rule refl)

lemmas xtor-inject = xtor-rel[of (=)]

lemma xtor-rel-induct: \((\forall x y. \text{vimage2p id-bnf id-bnf} R x y \Longrightarrow \text{IR} (\text{xtor} x) (\text{xtor} y)) \Longrightarrow R \leq \text{IR}\)
unfolding xtor-def vimage2p-def id-bnf-def ..

lemma xtor-iff-xtor: \(u = \text{xtor} w \iff \text{xtor} u = w\)
unfolding xtor-def ..

lemma Inl-def-alt: \(\text{Inl} \equiv (\lambda a. \text{xtor} (\text{id-bnf} (\text{Inl} a)))\)
unfolding xtor-def id-bnf-def by (rule reflexive)

lemma Inr-def-alt: \(\text{Inr} \equiv (\lambda a. \text{xtor} (\text{id-bnf} (\text{Inr} a)))\)
unfolding xtor-def id-bnf-def by (rule reflexive)

lemma Pair-def-alt: \(\text{Pair} \equiv (\lambda a b. \text{xtor} (\text{id-bnf} (a, b)))\)
unfolding xtor-def id-bnf-def by (rule reflexive)

definition ctor-rec :: 'a \Rightarrow 'a where
ctor-rec x = x

lemma ctor-rec: \(g = \text{id} \Longrightarrow \text{ctor-rec} f (\text{xtor} x) = f ((\text{id-bnf} \circ g \circ \text{id-bnf}) x)\)
unfolding ctor-rec-def id-bnf-def xtor-def comp-def id-def by hypsubst (rule refl)

lemma ctor-rec-unique: \(g = \text{id} \Longrightarrow f \circ \text{ctor} = s \circ (\text{id-bnf} \circ g \circ \text{id-bnf}) \Longrightarrow f = \text{ctor-rec} s\)
unfolding ctor-rec-def id-bnf-def xtor-def comp-def id-def by hypsubst (rule refl)

lemma ctor-rec-def-alt: \(f = \text{ctor-rec} (f \circ \text{id-bnf})\)
unfolding ctor-rec-def id-bnf-def comp-def by (rule refl)

lemma ctor-rec-o-map: \(\text{ctor-rec} f \circ g = \text{ctor-rec} (f \circ (\text{id-bnf} \circ g \circ \text{id-bnf}))\)
unfolding ctor-rec-def id-bnf-def comp-def by (rule refl)

lemma ctor-rec-transfer: \(\text{rel-fun} (\text{rel-fun} (\text{vimage2p id-bnf id-bnf} R) S) (\text{rel-fun} R S) \text{ctor-rec} \text{ctor-rec}\)
unfolding rel-fun-def vimage2p-def id-bnf-def ctor-rec-def by simp

lemma eq-fst-iff: \(a = \text{fst} p \iff (\exists b. p = (a, b))\)
by (cases p) auto
lemma eq-snd-iff: \( b = \text{snd} \, p \iff (\exists \, a. \, p = (a, b)) \)
by (cases p) auto

lemma ex-neg-all-pos: \((\exists \, x. \, P \, x) \implies Q \) \equiv (\forall \, x. \, P \, x \implies Q) 
by standard blast

lemma hypsubst-in-prems: \((\forall \, x. \, P \, x) = Q \) \equiv (\forall x. (P \, x = Q)) 
by standard blast

lemma isl-map-sum: 
isl (map-sum f g s) = isl s 
by (cases s) simp-all

lemma map-sum-sel: 
isl s \implies \text{projl} (map-sum f g s) = f \, (\text{projl} s) 
\neg \, isl \, s \implies \text{projr} (map-sum f g s) = g \, (\text{projr} \, s) 
by (cases s; simp)+

lemma set-sum-sel: 
isl s \implies \text{projl} \, s \in \text{setl} \, s 
\neg \, isl \, s \implies \text{projr} \, s \in \text{setr} \, s 
by (cases s; auto intro: setl.intros setr.intros)+

ML-file ⟨Tools/BNF/bnf-lfp-basic-sugar.ML⟩
declare prod.size [no-atp]

hide-const (open) xtor ctor-rec

hide-fact (open) 
xtor-def xtor-map xtor-set xtor-rel xtor-induct xtor-xtor xtor-inject ctor-rec-def ctor-rec
ctor-rec-def-alt ctor-rec-o-map xtor-rel-induct Inl-def-alt Inr-def-alt Pair-def-alt}
end
36 MESON Proof Method

de Morgan laws

lemma not-conjD: \(\neg (P \land Q) \implies \neg P \lor \neg Q\)
and not-disjD: \(\neg (P \lor Q) \implies \neg P \land \neg Q\)
and not-notD: \(\neg \neg P \implies P\)
and not-allD: \(\forall x. \neg P(x) \implies \exists x. \neg P(x)\)
and not-exD: \(\exists x. \neg P(x) \implies \forall x. \neg P(x)\)
by fast+

Removal of \(\rightarrow\) and \(\iff\) (positive and negative occurrences)

lemma imp-to-disjD: \(P \implies Q \implies \neg P \lor Q\)
and not-impD: \(\neg (P \implies Q) \implies P \land \neg Q\)
and iff-to-disjD: \(P \iff Q \implies (\neg P \lor Q) \land (\neg Q \lor P)\)
and not-iffD: \(\neg (P \iff Q) \implies (P \lor Q) \land (\neg P \lor \neg Q)\)
— Much more efficient than \(P \land \neg Q \lor Q \land \neg P\) for computing CNF
and not-refl-disj-D: \(x \neq x \lor P \implies P\)
by fast+

36.2 Pulling out the existential quantifiers

Conjunction

lemma conj-exD1: \(\forall P Q. (\exists x. P(x)) \land Q \implies \exists x. P(x) \land Q\)
and conj-exD2: \(\forall P Q. P \land (\exists x. Q(x)) \implies \exists x. P \land Q(x)\)
by fast+

Disjunction

lemma disj-exD: \(\forall P Q. (\exists x. P(x)) \lor (\exists x. Q(x)) \implies \exists x. P(x) \lor Q(x)\)
— DO NOT USE with forall-Skolemization: makes fewer schematic variables!!
— With ex-Skolemization, makes fewer Skolem constants
and disj-exD1: \(\forall P Q. (\exists x. P(x)) \lor Q \implies \exists x. P(x) \lor Q\)
and disj-exD2: \(\forall P Q. P \lor (\exists x. Q(x)) \implies \exists x. P \lor Q(x)\)
by fast+

lemma disj-assoc: \((P \lor Q) \lor R \implies P \lor (Q \lor R)\)
and disj-comm: \(P \lor Q \implies Q \lor P\)
and disj-FalseD1: \(\bot \lor P \implies P\)
and disj-FalseD2: \(P \lor \bot \implies P\)
by fast+

Generation of contrapositives
Inserts negated disjunct after removing the negation; P is a literal. Model elimination requires assuming the negation of every attempted subgoal, hence the negated disjuncts.

**Lemma** make-neg-rule: $\neg P \lor Q \implies ((\neg P \implies P) \implies Q)$
by blast

Version for Plaisted’s ”Postive refinement” of the Meson procedure

**Lemma** make-refined-neg-rule: $\neg P \lor Q \implies (P \implies Q)$
by blast

$P$ should be a literal

**Lemma** make-pos-rule: $P \lor Q \implies ((P \implies \neg P) \implies Q)$
by blast

Versions of make-neg-rule and make-pos-rule that don’t insert new assumptions, for ordinary resolution.

**Lemmas** make-neg-rule’ = make-refined-neg-rule

**Lemma** make-pos-rule’: $[P \lor Q; \neg P] \implies Q$
by blast

Generation of a goal clause – put away the final literal

**Lemma** make-neg-goal: $\neg P \implies ((\neg P \implies P) \implies False)$
by blast

**Lemma** make-pos-goal: $P \implies ((P \implies \neg P) \implies False)$
by blast

36.3 Lemmas for Forward Proof

There is a similarity to congruence rules. They are also useful in ordinary proofs.

**Lemma** conj-forward: $[P' \land Q'; P' \implies P; Q' \implies Q] \implies P \land Q$
by blast

**Lemma** disj-forward: $[P' \lor Q'; P' \implies P; Q' \implies Q] \implies P \lor Q$
by blast

**Lemma** imp-forward: $[P' \implies Q'; P \implies P'; Q' \implies Q] \implies P \implies Q$
by blast

**Lemma** imp-forward2: $[P' \implies Q'; P \implies P'; P' \implies Q' \implies Q] \implies P \implies Q$
by blast

**Lemma** disj-forward2: $[P' \lor Q'; P' \implies P; [Q'; P \implies False] \implies Q] \implies P \lor Q$
apply blast
done

lemma all-forward: \[ |∀ x. P'(x); !! x. P'(x) \implies P(x) | \implies ∀ x. P(x) \]
by blast

lemma ex-forward: \[ |∃ x. P'(x); !! x. P'(x) \implies P(x) | \implies ∃ x. P(x) \]
by blast

36.4 Clausification helper

lemma TruepropI: P ≡ Q \implies Trueprop P ≡ Trueprop Q
by simp

lemma ext-cong-neq: F g \neq F h \implies F g \neq F h ∧ (∃ x. g x \neq h x)
apply (erule contrapos-np)
apply clarsimp
apply (rule cong[where f = F])
by auto

Combinator translation helpers

definition COMBI :: 'a ⇒ 'a where
COMBI P = P

definition COMBK :: 'a ⇒ 'b ⇒ 'a where
COMBK P Q = P

definition COMBB :: ('b ⇒ 'c) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'c where
COMBB P Q R = P (Q R)

definition COMBC :: ('a ⇒ 'b ⇒ 'c) ⇒ 'b ⇒ 'a ⇒ 'c where
COMBC P Q R = P R (Q R)

definition COMBS :: ('a ⇒ 'b ⇒ 'c) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'c where
COMBS P Q R = P R (Q R)

lemma abs-S: λx. (f x) (g x) ≡ COMBS f g
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBS-def)
done

lemma abs-I: λx. x ≡ COMBI
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBI-def)
done

lemma abs-K: λx. y ≡ COMBK y
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBK-def)
done

lemma abs-B: λx. a (g x) ≡ COMBB a g
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBB-def)
done

lemma abs-C: λx. (f x) b ≡ COMBC f b
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBC-def)
done

36.5 Skolemization helpers

definition skolem :: 'a ⇒ 'a where
  skolem = (λx. x)

lemma skolem-COMBK-iff: P ≡ skolem (COMBK P (i::nat))
unfolding skolem-def COMBK-def by (rule refl)

lemmas skolem-COMBK-I = iffD1 [OF skolem-COMBK-iff]
lemmas skolem-COMBK-D = iffD2 [OF skolem-COMBK-iff]

36.6 Meson package

ML-file ⟨Tools/Meson/meson.ML⟩
ML-file ⟨Tools/Meson/meson-clausify.ML⟩
ML-file ⟨Tools/Meson/meson-tactic.ML⟩

hide-const (open) COMBI COMBB COMBC COMBS skolem
hide-fact (open) not-conjD not-disjD not-notD not-allD not-exD imp-to-disjD
  not-impD iff-to-disjD not-iffD not-refl-disj-D conj-exD1 conj-exD2 disj-exD
  disj-exD1 disj-exD2 disj-assoc disj-comm disj-FalseD1 disj-FalseD2 TruepropI
  ext-cong-neq COMBI-def COMBK-def COMBB-def COMBC-def COMBS-def
  abs-I abs-K
  abs-B abs-C abs-S skolem-def skolem-COMBK-iff skolem-COMBK-I skolem-COMBK-D

end

37 Automatic Theorem Provers (ATPs)

theory ATP
  imports Meson
begin
37.1 ATP problems and proofs

ML-file ⟨Tools/ATP/atp-util.ML⟩
ML-file ⟨Tools/ATP/atp-problem.ML⟩
ML-file ⟨Tools/ATP/atp-proof.ML⟩
ML-file ⟨Tools/ATP/atp-proof-redirect.ML⟩
ML-file ⟨Tools/ATP/atp-satallax.ML⟩

37.2 Higher-order reasoning helpers

definition fFalse :: bool where
  fFalse ←→ False

definition fTrue :: bool where
  fTrue ←→ True

definition fNot :: bool ⇒ bool where
  fNot P ←→ ¬ P

definition fComp :: ('a ⇒ bool) ⇒ 'a ⇒ bool where
  fComp P = (λx. ¬ P x)

definition fconj :: bool ⇒ bool ⇒ bool where
  fconj P Q ←→ P ∧ Q

definition fdisj :: bool ⇒ bool ⇒ bool where
  fdisj P Q ←→ P ∨ Q

definition fimplies :: bool ⇒ bool ⇒ bool where
  fimplies P Q ←→ (P → Q)

definition fAll :: ('a ⇒ bool) ⇒ bool where
  fAll P ←→ All P

definition fEx :: ('a ⇒ bool) ⇒ bool where
  fEx P ←→ Ex P

definition fequal :: 'a ⇒ 'a ⇒ bool where
  fequal x y ←→ (x = y)

lemma fTrue-ne-fFalse: fFalse ≠ fTrue
  unfolding fFalse-def fTrue-def by simp

lemma fNot-table:
  fNot fFalse = fTrue
  fNot fTrue = fFalse
  unfolding fFalse-def fTrue-def fNot-def by auto

lemma fconj-table:
  fconj fFalse P = fFalse
fconj P fFalse = fFalse
fconj fTrue fTrue = fTrue
unfolding fFalse-def fTrue-def fconj-def by auto

lemma fdisj-table:
fdisj fTrue P = fTrue
fdisj P fTrue = fTrue
fdisj fFalse fFalse = fFalse
unfolding fFalse-def fTrue-def fdisj-def by auto

lemma fimplies-table:
fimplies P fTrue = fTrue
fimplies fFalse P = fTrue
fimplies fTrue fFalse = fFalse
unfolding fFalse-def fTrue-def fimplies-def by auto

lemma fAll-table:
Ex (fComp P) ∨ fAll P = fTrue
All P ∨ fAll P = fFalse
unfolding fFalse-def fTrue-def fComp-def fAll-def by auto

lemma fEx-table:
All (fComp P) ∨ fEx P = fTrue
Ex P ∨ fEx P = fFalse
unfolding fFalse-def fTrue-def fComp-def fEx-def by auto

lemma fequal-table:
fequal x x = fTrue
x = y ∨ fequal x y = fFalse
unfolding fFalse-def fTrue-def fequal-def by auto

lemma fNot-law:
fNot P ≠ P
unfolding fNot-def by auto

lemma fComp-law:
fComp P x ←→ ¬ P x
unfolding fComp-def ..

lemma fconj-laws:
fconj P P ←→ P
fconj P Q ←→ fconj Q P
fNot (fconj P Q) ←→ fdisj (fNot P) (fNot Q)
unfolding fNot-def fconj-def fdisj-def by auto

lemma fdisj-laws:
fdisj P P ←→ P
fdisj P Q ←→ fdisj Q P
fNot (fdisj P Q) ←→ fconj (fNot P) (fNot Q)
unfolding fNot-def fconj-def fdisj-def by auto

lemma fimplies-laws:
\[ f\text{implies } P Q \leftrightarrow f\text{disj } (\neg P) Q \]
unfolding fNot-def fconj-def fdisj-def fimplies-def by auto

lemma fAll-law:
\[ f\text{Not } (f\text{All } R) \leftrightarrow f\text{Ex } (f\text{Comp } R) \]
unfolding fNot-def fComp-def fAll-def fEx-def by auto

lemma fEx-law:
\[ f\text{Not } (f\text{Ex } R) \leftrightarrow f\text{All } (f\text{Comp } R) \]
unfolding fNot-def fComp-def fAll-def fEx-def by auto

lemma fequal-laws:
\[ fequal x y = fequal y x \]
\[ fequal x y = fFalse \lor fequal y z = fFalse \lor fequal x z = fTrue \]
\[ fequal x y = fFalse \lor fequal (f x) (f y) = fTrue \]
unfolding fFalse-def fTrue-def fequal-def by auto

37.3 Basic connection between ATPs and HOL

ML-file ⟨Tools/lambda-lifting.ML⟩
ML-file ⟨Tools/monomorph.ML⟩
ML-file ⟨Tools/ATP/atp-problem-generate.ML⟩
ML-file ⟨Tools/ATP/atp-proof-reconstruct.ML⟩
ML-file ⟨Tools/ATP/atp-systems.ML⟩
end

38 Metis Proof Method

theory Metis
imports ATP
begin

ML-file ⟨~/src/Tools/Metis/metis.ML⟩

38.1 Literal selection and lambda-lifting helpers

definition select :: 'a ⇒ 'a where
\[ select = (\lambda x. x) \]
lemma not-atomize: \((\neg A \implies False) \equiv Trueprop A\)
by (cut-tac atomize-not [of \(\neg A\)]) simp

lemma atomize-not-select: \((A \implies select False) \equiv Trueprop (\neg A)\)
unfolding select-def by (rule atomize-not)
Theorem "Transfer"

Lemma not-atomize-select: \( \neg A \Rightarrow \select False \equiv \Trueprop A \)
Unfolding select-def by (rule not-atomize-def)

Lemma select-FalseI: False \( \Rightarrow \select False \) by simp

Definition lambda :: 'a \Rightarrow 'a where
lambda = (\x. x)

Lemma eq-lambdaI: \(\equiv y \Rightarrow x \equiv \lambda y \)
Unfolding lambda-def by assumption

38.2 Metis package

ML-file ⟨Tools/Metis/metis-generate.ML⟩
ML-file ⟨Tools/Metis/metis-reconstruct.ML⟩
ML-file ⟨Tools/Metis/metis-tactic.ML⟩

Hide-const (open) select fFalse fTrue fNot fComp fconj fdisj fimplies fAll fEx
fequal lambda
Hide-fact (open) select-def not-atomize atomize-not-select not-atomize-select select-FalseI
fFalse-def fTrue-def fNot-def fconj-def fdisj-def fimplies-def fAll-def fEx-def fequal-def
fTrue-ne-fFalse fNot-table fconj-table fdisj-table fimplies-table fAll-table fEx-table
fequal-table fAll-table fEx-table fNot-law fComp-law fconj-laws fdisj-laws fimplies-laws
fequal-laws fAll-law fEx-law lambda-def eq-lambdaI

End

39 Generic theorem transfer using relations

Theory Transfer
Imports Basic-BNF-LFPs Hilbert-Choice Metis
Begin

39.1 Relator for function space

Bundle lifting-syntax
Begin

Notation rel-fun (infixr \(===>\) 55)
Notation map-fun (infixr \(===>\) 55)
End

Context includes lifting-syntax
Begin

Lemma rel-funD2:
Assumes rel-fun A B f g and A x x
Shows B (f x) (g x)
Using assms by (rule rel-funD)
lemma rel-funE:
  assumes rel-fun \( A \leq B \leq f \leq g \) and \( A \times x \times y \)
  obtains \( B \leq (f \times x) \leq (g \times y) \)
  using assms by (simp add: rel-fun-def)

lemmas rel-fun-eq = fun.rel-eq

lemma rel-fun-eq-rel:
  shows rel-fun \( = \) \( R \equiv (\lambda f \times g. \forall x. R(f \times x)(g \times x)) \)
  by (simp add: rel-fun-def)

39.2 Transfer method

Explicit tag for relation membership allows for backward proof methods.

definition Rel :: \( 'a \Rightarrow 'b \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow 'b \Rightarrow \text{bool} \)
  where \( \text{Rel } r \equiv r \)

Handling of equality relations

definition is-equality :: \( 'a \Rightarrow 'a \Rightarrow \text{bool} \Rightarrow \text{bool} \)
  where \( \text{is-equality } R \leftarrow\rightarrow R = (=) \)

lemma is-equality-eq: is-equality \( = \)
  unfolding is-equality-def by simp

Reverse implication for monotonicity rules

definition rev-implies where
  rev-implies \( x \times y \leftarrow\rightarrow (y \rightarrow x) \)

Handling of meta-logic connectives

definition transfer-forall where
  transfer-forall \( \equiv \text{All} \)

definition transfer-implies where
  transfer-implies \( \equiv (\rightarrow) \)

definition transfer-bforall :: \( 'a \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow \text{bool} \Rightarrow \text{bool} \)
  where \( \text{transfer-bforall } \equiv (\lambda P \times Q. \forall x. P x \rightarrow Q x) \)

lemma transfer-bforall-unfold: \( \text{Trueprop } (\text{transfer-bforall } (\lambda x. P x)) \)
  unfolding atomize-all transfer-bforall-def ..

lemma transfer-implies-eq: \( A \Rightarrow B \) \( \equiv \text{Trueprop } (\text{transfer-implies } A B) \)
  unfolding atomize-imp transfer-implies-def ..

lemma transfer-bforall-unfold:
  \( \text{Trueprop } (\text{transfer-bforall } P (\lambda x. Q x)) \equiv (\forall x. P x \Rightarrow Q x) \)
  unfolding transfer-bforall-def atomize-imp atomize-all ..
THEORY "Transfer"

lemma transfer-start: \[ [P; \text{Rel} (=) P Q] \implies Q \]
  unfolding Rel-def by simp

lemma transfer-start': \[ [P; \text{Rel} (\rightarrow) P Q] \implies Q \]
  unfolding Rel-def by simp

lemma transfer-prover-start: \[ [x = x'; \text{Rel} \; R \; x \; y] \implies \text{Rel} \; R \; x \; y \]
  by simp

lemma untransfer-start: \[ [Q; \text{Rel} (=) P Q] \implies P \]
  unfolding Rel-def by simp

lemma Rel-eq-refl: \text{Rel} (=) x x
  unfolding Rel-def ..

lemma Rel-app:
  assumes \text{Rel} (A ===> B) f g and \text{Rel} A x y
  shows \text{Rel} B (f x) (g y)
  using assms unfolding Rel-def rel-fun-def by fast

lemma Rel-abs:
  assumes \( \forall x \ y. \text{Rel} \; A \; x \; y \implies \text{Rel} \; B \; (f \; x) \; (g \; y) \)
  shows \text{Rel} (A ===> B) (\lambda x. f \; x) (\lambda y. g \; y)
  using assms unfolding Rel-def rel-fun-def by fast

39.3 Predicates on relations, i.e. “class constraints”

definition left-total :: ('a => 'b => bool) => bool
  where left-total R <-> (\forall x. \exists y. R \; x \; y)

definition left-unique :: ('a => 'b => bool) => bool
  where left-unique R <-> (\forall x \ y \ z. R \; x \; z \implies R \; y \; z \implies x = y)

definition right-total :: ('a => 'b => bool) => bool
  where right-total R <-> (\forall y. \exists x. R \; x \; y)

definition right-unique :: ('a => 'b => bool) => bool
  where right-unique R <-> (\forall x \ y \ z. R \; x \; z \implies y = z)

definition bi-total :: ('a => 'b => bool) => bool
  where bi-total R <-> (\forall x \ y \ z. R \; x \; y \implies R \; x \; z \implies y = z)

definition bi-unique :: ('a => 'b => bool) => bool
  where bi-unique R <-> (\forall x \ y \ z. R \; x \; y \implies R \; x \; z \implies y = z) \land (\forall x \ y \ z. R \; x \; z \implies R \; y \; z \implies x = y)

lemma left-unique1: (\forall x \ y \ z. \[ A \; x \; z; A \; y \; z \] \implies x = y) \implies \text{left-unique} \; A
unfolding \texttt{left-unique-def} by \texttt{blast}

\textbf{lemma} \texttt{left-uniqueD}: \( \{ \text{left-unique } A; A x z; A y z \} \implies x = y \)

unfolding \texttt{left-unique-def} by \texttt{blast}

\textbf{lemma} \texttt{left-totalI}:
\((\forall x. \exists y. R x y) \implies \text{left-total } R\)

unfolding \texttt{left-total-def} by \texttt{blast}

\textbf{lemma} \texttt{left-totalE}:
\begin{itemize}
\item assumes \texttt{left-total } R
\item obtains \((\forall x. \exists y. R x y)\)
\end{itemize}

using \texttt{assms} unfolding \texttt{left-total-def} by \texttt{blast}

\textbf{lemma} \texttt{bi-uniqueDr}: \(\{ \text{bi-unique } A; A x y; A x z \} \implies y = z\)
by (\texttt{simp add: bi-unique-def})

\textbf{lemma} \texttt{bi-uniqueDl}: \(\{ \text{bi-unique } A; A x y; A z y \} \implies x = z\)
by (\texttt{simp add: bi-unique-def})

\textbf{lemma} \texttt{right-uniqueI}:
\((\forall x y z. \{ A x y; A x z \} \implies y = z) \implies \text{right-unique } A\)

unfolding \texttt{right-unique-def} by \texttt{fast}

\textbf{lemma} \texttt{right-uniqueD}: \(\{ \text{right-unique } A; A x y; A x z \} \implies y = z\)

unfolding \texttt{right-unique-def} by \texttt{fast}

\textbf{lemma} \texttt{right-totalI}:
\((\forall y. \exists x. A x y) \implies \text{right-total } A\)

by (\texttt{simp add: right-total-def})

\textbf{lemma} \texttt{right-totalE}:
\begin{itemize}
\item assumes \texttt{right-total } A
\item obtains \texttt{x where } A x y
\end{itemize}

using \texttt{assms} by (\texttt{auto simp add: right-total-def})

\textbf{lemma} \texttt{right-total-alt-def2}:
\begin{itemize}
\item \texttt{right-total } R \longleftrightarrow ((R =\gg\ (\rightarrow)) =\gg\ (\rightarrow)) \texttt{ All All}
\end{itemize}

unfolding \texttt{right-total-def rel-fun-def}

apply (\texttt{rule iffI, fast})
apply (\texttt{rule allI})
apply (\texttt{drule-tac } x=\lambda x. \texttt{True in spec})
apply (\texttt{drule-tac } x=\lambda y. \exists x. R x y \texttt{ in spec})
apply \texttt{fast}
done

\textbf{lemma} \texttt{right-unique-alt-def2}:
\begin{itemize}
\item \texttt{right-unique } R \longleftrightarrow ((R =\gg\ R =\gg\ (\rightarrow)) (\rightarrow)) (\rightarrow)
\end{itemize}

unfolding \texttt{right-unique-def rel-fun-def} by \texttt{auto}

\textbf{lemma} \texttt{bi-total-alt-def2}:
bi-total \( R \leftrightarrow ((R \Rightarrow \Rightarrow (\Rightarrow)) \Rightarrow \Rightarrow (\Rightarrow)) \) All All

unfolding bi-total-def rel-fun-def

apply (rule iffI, fast)
apply safe
apply (drule_tac \( x=x \), \( \exists \ y \). \( R \ x \ y \) in spec)
apply (drule_tac \( x=y \), True in spec)
apply fast
apply (drule_tac \( x=x \), \( \exists \ y \). \( R \ x \ y \) in spec)
apply fast
done

lemma bi-unique-alt-def2:
bi-unique \( R \leftrightarrow (R \Rightarrow \Rightarrow (\Rightarrow)) \Rightarrow \Rightarrow (\Rightarrow) \) (\( \Rightarrow \) (\( \Rightarrow \))

unfolding bi-unique-def rel-fun-def by auto

lemma [simp]:
shows left-unique-conversep: left-unique \( A \ x \ y \) \( x \) \( y \)
and right-unique-conversep: right-unique \( A \ x \ y \) \( x \) \( y \)
by(auto simp add: left-unique-def right-unique-def)

lemma [simp]:
shows left-total-conversep: left-total \( A \ x \ y \) \( x \) \( y \)
and right-total-conversep: right-total \( A \ x \ y \) \( x \) \( y \)
by(simp-all add: left-total-def right-total-def)

lemma bi-unique-conversep [simp]: bi-unique \( R \ x \) \( x \) \( y \)
by(auto simp add: bi-unique-def)

lemma bi-total-conversep [simp]: bi-total \( R \ x \) \( x \) \( y \)
by(auto simp add: bi-total-def)

lemma right-unique-alt-def: right-unique \( R = (\text{conversep} \ R \ OO \ R \leq (\Rightarrow)) \) unfolding right-unique-def by blast
lemma left-unique-alt-def: left-unique \( R = (R \ OO \ \text{conversep} \ R \leq (\Rightarrow)) \) unfolding left-unique-def by blast

lemma right-total-alt-def: right-total \( R = (\text{conversep} \ R \ OO \ R \geq (\Rightarrow)) \) unfolding right-total-def by blast
lemma left-total-alt-def: left-total \( R = (R \ OO \ \text{conversep} \ R \geq (\Rightarrow)) \) unfolding left-total-def by blast

lemma bi-total-alt-def: bi-total \( R = (\text{left-total} \ A \land \ \text{right-total} \ A) \) unfolding left-total-def right-total-def bi-total-def by blast

lemma bi-unique-alt-def: bi-unique \( R = (\text{left-unique} \ A \land \ \text{right-unique} \ A) \) unfolding left-unique-def right-unique-def bi-unique-def by blast

lemma bi-totalI: left-total \( R \Rightarrow \text{right-total} \ R \Rightarrow \text{bi-total} \ R \)
THEORY "Transfer"

unfolding bi-total-alt-def ..

lemma bi-uniqueI: left-unique R \rightarrow right-unique R \rightarrow bi-unique R
unfolding bi-unique-alt-def ..
end

lemma is-equality-lemma: (∀ R. is-equality R \implies PROP (P R)) ≡ PROP (P (=))
apply (unfold is-equality-def)
apply (rule equal-intr-rule)
apply (drule meta-spec)
apply (erule meta-mp)
apply (rule refl)
apply simp
done

lemma Domainp-lemma: (∀ T. Domainp T = R \implies PROP (P R)) ≡ PROP (P (Domainp T))
apply (rule equal-intr-rule)
apply (drule meta-spec)
apply (erule meta-mp)
apply (rule refl)
apply simp
done

ML-file ⟨Tools/Transfer/transfer.ML⟩
declare refl [transfer-rule]

hide-const (open) Rel

context includes lifting-syntax
begin

Handling of domains

lemma Domainp-iff: Domainp T x \iff (∃ y. T x y)
by auto

lemma Domainp-refl[transfer-domain-rule]:
Domainp T = Domainp T ..

lemma Domain-eq-top[transfer-domain-rule]: Domainp (=) = top by auto

lemma Domainp-pred-fun-eq[relator-domain]:
assumes left-unique T
shows Domainp (T ===> S) = pred-fun (Domainp T) (Domainp S)
using assms unfolding rel-fun-def Domainp-iff [abs-def] left-unique-def fun-eq-iff
pred-fun-def
apply safe
apply blast
apply (subst all-comm)
apply (rule choice)
apply blast
done

Properties are preserved by relation composition.

lemma OO-def: \( R \ OO S = (\lambda x z. \exists y. \ R \ x \ y \land S \ y \ z) \)
   by auto

lemma bi-total-OO: \([\text{bi-total } A; \text{bi-total } B] \implies \text{bi-total } (A \ OO \ B)\)
   unfolding bi-total-def OO-def by fast

lemma bi-unique-OO: \([\text{bi-unique } A; \text{bi-unique } B] \implies \text{bi-unique } (A \ OO \ B)\)
   unfolding bi-unique-def OO-def by blast

lemma right-total-OO: \([\text{right-total } A; \text{right-total } B] \implies \text{right-total } (A \ OO \ B)\)
   unfolding right-total-def OO-def by fast

lemma right-unique-OO: \([\text{right-unique } A; \text{right-unique } B] \implies \text{right-unique } (A \ OO \ B)\)
   unfolding right-unique-def OO-def by fast

lemma left-total-OO: \( \text{left-total } R \implies \text{left-total } S \implies \text{left-total } (R \ OO \ S)\)
   unfolding left-total-def OO-def by fast

lemma left-unique-OO: \( \text{left-unique } R \implies \text{left-unique } S \implies \text{left-unique } (R \ OO \ S)\)
   unfolding left-unique-def OO-def by blast

39.4 Properties of relators

lemma left-total-eq[transfer-rule]: \( \text{left-total } (=)\)
   unfolding left-total-def by blast

lemma left-unique-eq[transfer-rule]: \( \text{left-unique } (=)\)
   unfolding left-unique-def by blast

lemma right-total-eq [transfer-rule]: \( \text{right-total } (=)\)
   unfolding right-total-def by simp

lemma right-unique-eq [transfer-rule]: \( \text{right-unique } (=)\)
   unfolding right-unique-def by simp

lemma bi-total-eq[transfer-rule]: \( \text{bi-total } (=)\)
   unfolding bi-total-def by simp

lemma bi-unique-eq[transfer-rule]: \( \text{bi-unique } (=)\)
   unfolding bi-unique-def by simp
lemma left-total-fun[transfer-rule]:
\left-total A; \left-total B \implies \left-total (A \Rightarrow B)
unfolding left-total-def rel-fun-def
apply (rule allI, rename-tac f)
apply (rule-tac x=\lambda y. SOME z. B (f (THE x. A x y)) z in exI)
apply clarify
apply (subgoal-tac (THE x. A x y) = x, simp)
apply (rule someI-ex)
apply (simp)
apply (rule the-equality)
apply assumption
apply (simp add: left-unique-def)
done

lemma left-unique-fun[transfer-rule]:
\left-unique A; \left-unique B \implies \left-unique (A \Rightarrow B)
unfolding left-total-def left-unique-def rel-fun-def
by (clarify, rule ext, fast)

lemma right-total-fun [transfer-rule]:
\right-total A; \right-total B \implies \right-total (A \Rightarrow B)
unfolding right-total-def rel-fun-def
apply (rule allI, rename-tac g)
apply (rule-tac x=\lambda x. SOME z. B z (g (THE y. A x y)) in exI)
apply clarify
apply (subgoal-tac (THE y. A x y) = y, simp)
apply (rule someI-ex)
apply (simp)
apply (rule the-equality)
apply assumption
apply (simp add: right-unique-def)
done

lemma right-unique-fun [transfer-rule]:
\right-total A; \right-unique B \implies \right-unique (A \Rightarrow B)
unfolding right-total-def right-unique-def rel-fun-def
by (clarify, rule ext, fast)

lemma bi-total-fun[transfer-rule]:
\bi-unique A; \bi-total B \implies \bi-total (A \Rightarrow B)
unfolding bi-unique-all-def bi-total-alt-def
by (blast intro: right-total-fun left-total-fun)

lemma bi-unique-fun[transfer-rule]:
\bi-total A; \bi-unique B \implies \bi-unique (A \Rightarrow B)
unfolding bi-unique-all-def bi-total-alt-def
by (blast intro: right-unique-fun left-unique-fun)
end

lemma if-conn:
  (if P ∧ Q then t else e) = (if P then if Q then t else e else e)
  (if P ∨ Q then t else e) = (if P then t else if Q then t else e)
  (if P → Q then t else e) = (if P then if Q then t else e else t)
  (if ¬ P then t else e) = (if P then e else t)
by auto

ML-file (Tools/Transfer/transfer-bnf.ML)
ML-file (Tools/BNF/bnf-fp-rec-sugar-transfer.ML)

declare pred-fun-def [simp]
declare rel-fun-eq [relator-eq]
declare fun.Domainp-rel[relator-domain del]

39.5 Transfer rules
context includes lifting-syntax
begin

lemma Domainp-forall-transfer [transfer-rule]:
  assumes right-total A
  shows ((A ===> (=)) ===> (=))
  (transfer-bforall (Domainp A)) transfer-forall
using assms unfolding right-total-def
unfolding transfer-forall-def transfer-bforall-def rel-fun-def Domainp-iff
by fast

Transfer rules using implication instead of equality on booleans.

lemma transfer-forall-transfer [transfer-rule]:
  bi-total A =⇒ ((A ===> (=)) ===> (=)) transfer-forall transfer-forall
  right-total A =⇒ ((A ===> (=)) ===> implies) transfer-forall transfer-transfer
  right-total A =⇒ ((A ===> implies) ===> implies) transfer-forall transfer-transfer
  bi-total A =⇒ ((A ===> (=)) ===> rev-implies) transfer-transfer transfer-transfer
  bi-total A =⇒ ((A ===> rev-implies) ===> rev-implies) transfer-transfer transfer-transfer
unfolding transfer-forall-def rev-implies-def rel-fun-def right-total-def
by fast+

lemma transfer-implies-transfer [transfer-rule]:
  (¯¯)
  (rev-implies ===> implies) ===> implies
  (rev-implies ===> (=)) ===> implies
  (¯¯) ===> implies ===> implies
  (¯¯) ===> (=) ===> implies
  (implies ===> rev-implies ===> rev-implies) ===> implies
  (implies ===> (=) ===> rev-implies) ===> implies
  (implies ===> (=) ===> rev-implies) ===> implies
  (implies ===> (=) ===> rev-implies) ===> implies
  (implies ===> (=) ===> rev-implies) ===> implies
lemma eq-imp-transfer [transfer-rule]:
right-unique A \Rightarrow (A \implies A \implies (\neg \rightarrow)) (=) (=)
unfolding right-unique-alt-def2.

Transfer rules using equality.

lemma left-unique-transfer [transfer-rule]:
assumes right-total A
assumes right-total B
assumes bi-unique A
shows ((A \implies B \implies (=)) \implies left-unique left-unique
using assms unfolding left-unique-def [abs-def] right-total-def bi-unique-def rel-fun-def
by metis

lemma eq-transfer [transfer-rule]:
assumes bi-unique A
shows (A \implies A \implies (=)) (=) (=)
using assms unfolding bi-unique-def rel-fun-def by auto

lemma right-total-Ex-transfer [transfer-rule]:
assumes right-total A
shows ((A \implies (=)) \implies (Ex (Collect (Domainp A)))) Ex
using assms unfolding right-total-def Ex-def rel-fun-def Domainp-iff [abs-def]
by fast

lemma right-total-All-transfer [transfer-rule]:
assumes right-total A
shows ((A \implies (=)) \implies (All (Collect (Domainp A)))) All
using assms unfolding right-total-def All-def rel-fun-def Domainp-iff [abs-def]
by fast

context
  includes lifting-syntax
begin

lemma right-total-fun-eq-transfer:
assumes [transfer-rule]: right-total A bi-unique B
shows ((A \implies B) \implies (A \implies B) \implies (\lambda f g. \forall x \in Collect(Domainp A). f x = g x) (=)
unfolding fun-eq-iff
by transfer-prover

end

lemma All-transfer [transfer-rule]:
assumes bi-total A
shows \((A \implies (=)) \implies (=))\) All All
using assms unfolding bi-total-def rel-fun-def by fast

lemma Ex-transfer [transfer-rule]:
assumes bi-total A
shows \((A \implies (=)) \implies (=))\) Ex Ex
using assms unfolding bi-total-def rel-fun-def by fast

lemma Ex1-parametric [transfer-rule]:
assumes [transfer-rule]: bi-unique A bi-total A
shows \((A \implies (=)) \implies (=))\) Ex1 Ex1
unfolding Ex1-def [abs-def] by transfer-prover

declare If-transfer [transfer-rule]

lemma Let-transfer [transfer-rule]: \((A \implies (A \implies B)) \implies B\) Let Let
unfolding rel-fun-def by simp

declare id-transfer [transfer-rule]

declare comp-transfer [transfer-rule]

lemma curry-transfer [transfer-rule]:
\((\text{rel-prod} A B \implies C) \implies (\implies (\implies A \implies B))\implies (\implies C)\) curry curry
unfolding curry-def by transfer-prover

lemma fun-upd-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A
shows \((A \implies B) \implies A \implies B \implies A \implies B\) fun-upd fun-upd
unfolding fun-upd-def [abs-def] by transfer-prover

lemma case-nat-transfer [transfer-rule]:
\((A \implies \text{((=) \implies A) \implies (=) \implies A})\) case-nat case-nat
unfolding rel-fun-def by (simp split: nat.split)

lemma rec-nat-transfer [transfer-rule]:
\((A \implies \text{((=) \implies A \implies A) \implies (=) \implies A})\) rec-nat rec-nat
unfolding rel-fun-def by (clarsimp, rename-tac n, induct-tac n, simp-all)

lemma funpow-transfer [transfer-rule]:
\((=) \implies \text{(A \implies A) \implies (A \implies A)})\) compow compow
unfolding funpow-def by transfer-prover

lemma mono-transfer [transfer-rule]:
assumes [transfer-rule]: bi-total A
assumes [transfer-rule]: \((A \implies A \implies (=)) \leq \leq\)
assumes [transfer-rule]: \((B \implies B \implies (=)) \leq \leq\)
shows \((A \implies B) \implies (=))\) mono mono
unfolding mono-def [abs-def] by transfer-prover
THEORY “Transfer”

lemma right-total-relcompp-transfer[transfer-rule]:
  assumes [transfer-rule]: right-total B
  shows \((A \implies B \implies (=)) \implies (B \implies C \implies (=)) \implies A\)
  \((\lambda R S x z. \exists y \in \text{Collect}(\text{Domain} B). \ R x y \land S y z) \ (\text{OO})\)
unfolding \text{OO-def} \ [\text{abs-def}] \ by \ transfer-prover

lemma relcompp-transfer[transfer-rule]:
  assumes [transfer-rule]: bi-total B
  shows \((A \implies B \implies (=)) \implies A \implies (=)) \implies (B \implies C \implies (=)) \implies A\)
  \((\text{OO} \ (\text{OO})\)
unfolding \text{OO-def} \ [\text{abs-def}] \ by \ transfer-prover

lemma right-total-Domainp-transfer[transfer-rule]:
  assumes [transfer-rule]: right-total B
  shows \((A \implies B \implies (=)) \implies A \implies (=)) \implies (\lambda T x. \exists y \in \text{Collect}(\text{Domain} B). T x y) \ \text{Domainp}\)
apply\((\text{subst}(2) \ \text{Domainp-iff}[\text{abs-def}])\) \ by \ transfer-prover

lemma Domainp-transfer[transfer-rule]:
  assumes [transfer-rule]: bi-total B
  shows \((A \implies B \implies (=)) \implies A \implies (=)) \ \text{Domainp} \ \text{Domainp}\)
unfolding \text{Domainp-iff}[\text{abs-def}] \ by \ transfer-prover

lemma reflp-transfer[transfer-rule]:
  \text{bi-total} A \implies ((A \implies A \implies (=)) \implies (=)) \ \text{reflp} \ \text{reflp}
  \text{right-total} A \implies ((A \implies A \implies \text{implies}) \implies \implies) \ \text{reflp} \ \text{reflp}
  \text{right-total} A \implies ((A \implies A \implies (=)) \implies \implies) \ \text{reflp} \ \text{reflp}
  \text{bi-total} A \implies ((A \implies A \implies \text{rev-implies}) \implies \implies) \ \text{rev-implies} \ \text{reflp} \ \text{reflp}
  \text{bi-total} A \implies ((A \implies A \implies (=)) \implies \implies) \ \text{rev-implies} \ \text{reflp} \ \text{reflp}
unfolding \text{reflp-def}[\text{abs-def}] \ \text{rev-implies-def} \ \text{bi-total-def} \ \text{right-total-def} \ \text{rel-fun-def}
by \ \text{fast} +

lemma right-unique-transfer [transfer-rule]:
  \[ \text{right-total} A; \ \text{right-total} B; \ \text{bi-unique} B \]\n  \implies ((A \implies B \implies (=)) \implies \implies) \ \text{right-unique} \ \text{right-unique}
unfolding \text{right-unique-def}[\text{abs-def}] \ \text{right-total-def} \ \text{bi-unique-def} \ \text{rel-fun-def}
by \ \text{metis}

lemma left-total-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-total A bi-total B
  shows \((A \implies B \implies (=)) \implies \implies) \ \text{left-total} \ \text{left-total}
unfolding \text{left-total-def}[\text{abs-def}] \ by \ \text{transfer-prover}

lemma right-total-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-total A bi-total B
  shows \((A \implies B \implies (=)) \implies \implies) \ \text{right-total} \ \text{right-total}
unfolding \text{right-total-def}[\text{abs-def}] \ by \ \text{transfer-prover}
lemma left-unique-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-unique A bi-total A bi-total B
  shows \(((A ==\Longrightarrow B ==\Longrightarrow (=)) ==\Longrightarrow (=))\) left-unique left-unique
unfolding left-unique-def[abs-def] by transfer-prover

lemma prod-pred-parametric [transfer-rule]:
  \(((A ==\Longrightarrow (=)) ==\Longrightarrow (B ==\Longrightarrow (=))) ==\Longrightarrow rel-prod A B ==\Longrightarrow (=))\)
unfolding prod.prd-set[abs-def] Basic-BNFs.fsts-def Basic-BNFs.snds-def fstsp.simps sndsp.simps
by simp transfer-prover

lemma apfst-parametric [transfer-rule]:
  \(((A ==\Longrightarrow B) ==\Longrightarrow rel-prod A C ==\Longrightarrow rel-prod B C) \) apfst apfst
unfolding apfst-def[abs-def] by transfer-prover

lemma rel-fun-eq-eq-onp: ((=) ==\Longrightarrow eq-onp P) = eq-onp (λf. ∀ x. P(f x))
unfolding eq-onp-def rel-fun-def by auto

lemma eq-onp-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows \(((A ==\Longrightarrow (=)) ==\Longrightarrow (A ==\Longrightarrow (=)))) eq-onp eq-onp
unfolding eq-onp-def[abs-def] by transfer-prover

lemma rtranclp-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-unique A bi-total A
  shows \(((A ==\Longrightarrow A ==\Longrightarrow (=)) ==\Longrightarrow A ==\Longrightarrow A ==\Longrightarrow (=)))\) rtranclp
rtranclp
proof(rule rel-funI iffI)+
  fix R :: 'a ⇒ 'a ⇒ bool and R' x y x' y'
  assume R: \((A ==\Longrightarrow A ==\Longrightarrow (=)) \) R R' and A x x'
\{
  assume R** x y A y'
  thus R** x' y'
  proof(induction arbitrary: y')
  case base
  with \(\langle bi-unique A \rangle \langle A x x' \rangle \) have x' = y' by rule bi-uniqueDr
  thus ?case by simp
  next
  case (step y z z')
  from \(\langle bi-total A \rangle \) obtain y' where A y y' unfolding bi-total-def by blast
  hence R** x' y' by(rule step.IH)
  moreover from R \(\langle A y y' \rangle \langle A z z' \rangle \langle R y z \rangle \)
  have R' y' z' by(auto dest: rel-funD)
ultimately show \textit{case} ..
qed
next
assume $R^{**} x' y' A y'$
thus $R^{**} x y$
proof (induction arbitrary: $y$)
case base
with $\langle \text{bi-unique } A \rangle \langle A x x' \rangle$
have $x = y$ by (rule bi-uniqueDl)
thus \textit{case} by simp
next
case (step $y' z' z$)
from $\langle \text{bi-total } A \rangle$
obtain $y$ where $A y y'$ unfolding bi-total-def by blast
hence $R^{**} x y$ by (rule step.IH)
moreover from $R \langle A y y' \rangle \langle A z z' \rangle \langle R' y' z' \rangle$
have $R y z$ by (auto dest: rel-funD)
ultimately show \textit{case} ..
qed

\textbf{lemma} right-unique-parametric [transfer-rule]:
\textbf{assumes} [transfer-rule]: bi-total $A$ bi-unique $B$ bi-total $B$
\textbf{shows} $( (A =\Rightarrow B) =\Rightarrow (\sim) ) =\Rightarrow (\sim)$ right-unique right-unique
unfolding right-unique-def[abs-def] by transfer-prover

\textbf{lemma} map-fun-parametric [transfer-rule]:
$((A =\Rightarrow B) =\Rightarrow (C =\Rightarrow D) =\Rightarrow (B =\Rightarrow C) =\Rightarrow A =\Rightarrow D)$ map-fun map-fun
unfolding map-fun-def[abs-def] by transfer-prover

end

39.6 of-bool and of-nat

context
  includes lifting-syntax

begin

\textbf{lemma} transfer-rule-of-bool:
$((\sim\equiv\equiv) =\Rightarrow (\equiv))$ of-bool of-bool
if [transfer-rule]: $\langle 0 \equiv 0 \rangle \langle 1 \equiv 1 \rangle$
for $R :: \langle 'a::zero-neq-one \Rightarrow 'b::zero-neq-one \Rightarrow bool \rangle$ (infix $\equiv 50$)
by (unfold of-bool-def [abs-def]) transfer-prover

\textbf{lemma} transfer-rule-of-nat:
$((\equiv) =\Rightarrow (\equiv))$ of-nat of-nat
if [transfer-rule]: $\langle 0 \equiv 0 \rangle \langle 1 \equiv 1 \rangle$
$\langle (\equiv) =\Rightarrow (\equiv) =\Rightarrow (\equiv) \rangle (+) (+)$
for $R :: \langle 'a::semiring-1 \Rightarrow 'b::semiring-1 \Rightarrow bool \rangle$ (infix $\equiv 50$)
40 Binary Numerals

theory Num
  imports BNF-Least-Fixpoint Transfer
begin

40.1 The num type

datatype num = One | Bit0 num | Bit1 num

Increment function for type num

primrec inc :: num ⇒ num
  where
  inc One = Bit0 One
  | inc (Bit0 x) = Bit1 x
  | inc (Bit1 x) = Bit0 (inc x)

Converting between type num and type nat

primrec nat-of-num :: num ⇒ nat
  where
  nat-of-num One = Suc 0
  | nat-of-num (Bit0 x) = nat-of-num x + nat-of-num x
  | nat-of-num (Bit1 x) = Suc (nat-of-num x + nat-of-num x)

primrec num-of-nat :: nat ⇒ num
  where
  num-of-nat 0 = One
  | num-of-nat (Suc n) = (if 0 < n then inc (num-of-nat n) else One)

lemma nat-of-num-pos: 0 < nat-of-num x
  by (induct x) simp-all

lemma nat-of-num-neq-0: nat-of-num x ≠ 0
  by (induct x) simp-all

lemma nat-of-num-inc: nat-of-num (inc x) = Suc (nat-of-num x)
  by (induct x) simp-all

lemma nat-of-num-double: 0 < n ⇒ nat-of-num (n + n) = Bit0 (nat-of-num n)
  by (induct n) simp-all

Type num is isomorphic to the strictly positive natural numbers.
lemma nat-of-num-inverse: num-of-nat (nat-of-num x) = x
  by (induct x) (simp-all add: num-of-nat-double nat-of-num-pos)

lemma num-of-nat-inverse: 0 < n ⇒ nat-of-num (num-of-nat n) = n
  by (induct n) (simp-all add: nat-of-num-inc)

lemma num-eq-iff: x = y ⇔ nat-of-num x = nat-of-num y
  apply safe
  apply (drule arg-cong [where f=num-of-nat])
  apply (simp add: nat-of-num-inverse)
  done

lemma num-induct [case-names One inc]:
  fixes P :: num ⇒ bool
  assumes One: P One
  and inc: ∀x. P x ⇒ P (inc x)
  shows P x
proof –
  obtain n where n: Suc n = nat-of-num x
    by (cases nat-of-num x) (simp-all add: nat-of-num-neq-0)
  have P (num-of-nat (Suc n))
  proof (induct n)
    case 0
    from One show ?case by simp
  next
    case (Suc n)
    then have P (inc (num-of-nat (Suc n))) by (rule inc)
    then show P (num-of-nat (Suc (Suc n))) by simp
  qed
  with n show P x
  by (simp add: nat-of-num-inverse)
qed

From now on, there are two possible models for num: as positive naturals (rule num-induct) and as digit representation (rules num.induct, num.cases).

40.2 Numeral operations

instantiation num :: {plus,times,linorder}
begin

definition [code del]: m + n = num-of-nat (nat-of-num m + nat-of-num n)
definition [code del]: m * n = num-of-nat (nat-of-num m * nat-of-num n)
definition [code del]: m ≤ n ⇔ nat-of-num m ≤ nat-of-num n
definition [code del]: m < n ⇔ nat-of-num m < nat-of-num n
instance
  by standard (auto simp add: less-num-def less-eq-num-def num-eq-iff)

end

lemma nat-of-num-add: nat-of-num (x + y) = nat-of-num x + nat-of-num y
  unfolding plus-num-def
  by (intro num-of-nat-inverse add-pos-pos nat-of-num-pos)

lemma nat-of-num-mult: nat-of-num (x * y) = nat-of-num x * nat-of-num y
  unfolding times-num-def
  by (intro num-of-nat-inverse mult-pos-pos nat-of-num-pos)

lemma add-num-simps [simp, code]:
  One + One = Bit0 One
  One + Bit0 n = Bit1 n
  One + Bit1 n = Bit0 (n + One)
  Bit0 m + One = Bit1 m
  Bit0 m + Bit0 n = Bit0 (m + n)
  Bit0 m + Bit1 n = Bit1 (m + n)
  Bit1 m + One = Bit0 (m + One)
  Bit1 m + Bit0 n = Bit1 (m + n)
  Bit1 m + Bit1 n = Bit0 (m + n + One)
  by (simp-all add: num-eq-iff nat-of-num-add)

lemma mult-num-simps [simp, code]:
  m * One = m
  One * n = n
  Bit0 m * Bit0 n = Bit0 (Bit0 (m * n))
  Bit0 m * Bit1 n = Bit0 (m * Bit1 n)
  Bit1 m * Bit0 n = Bit0 (Bit1 m * n)
  Bit1 m * Bit1 n = Bit1 (m + n + Bit0 (m * n))
  by (simp-all add: num-eq-iff nat-of-num-add nat-of-num-mult distrib-right distrib-left)

lemma eq-num-simps:
  One = One ↔ True
  One = Bit0 n ↔ False
  One = Bit1 n ↔ False
  Bit0 m = One ↔ False
  Bit1 m = One ↔ False
  Bit0 m = Bit0 n ↔ m = n
  Bit0 m = Bit1 n ↔ False
  Bit1 m = Bit0 n ↔ False
  Bit1 m = Bit1 n ↔ m = n
  by simp-all

lemma le-num-simps [simp, code]:
  One ≤ n ↔ True
  Bit0 m ≤ One ↔ False
Bit1 \( m \leq One \iff False \)
Bit0 \( m \leq Bit0 \ n \iff m \leq n \)
Bit0 \( m \leq Bit1 \ n \iff m \leq n \)
Bit1 \( m \leq Bit1 \ n \iff m \leq n \)
Bit1 \( m \leq Bit0 \ n \iff m < n \)
using nat-of-num-pos \([of \ n]\) nat-of-num-pos \([of \ m]\)
by (auto simp add: less-eq-num-def less-num-def)

**lemma** less-num-simps [simp, code]:

\( m < One \iff False \)
\( One < Bit0 \ n \iff True \)
\( One < Bit1 \ n \iff True \)
Bit0 \( m < Bit0 \ n \iff m < n \)
Bit0 \( m < Bit1 \ n \iff m \leq n \)
Bit1 \( m < Bit1 \ n \iff m < n \)
Bit1 \( m < Bit0 \ n \iff m < n \)
using nat-of-num-pos \([of \ n]\) nat-of-num-pos \([of \ m]\)
by (auto simp add: less-eq-num-def less-num-def)

**lemma** le-num-One-iff: \( x \leq num.One \iff x = num.One \)
by (simp add: antisym-conv)

Rules using \( One \) and \( inc \) as constructors.

**lemma** add-One: \( x + One = inc \ x \)
by (simp add: num-eq-iff nat-of-num-add nat-of-num-inc)

**lemma** add-One-commute: \( One + n = n + One \)
by (induct \( n \)) simp-all

**lemma** add-inc: \( x + inc \ y = inc \ (x + y) \)
by (simp add: num-eq-iff nat-of-num-add nat-of-num-inc)

**lemma** mult-inc: \( x \ast inc \ y = x \ast y + x \)

The \( num-of-nat \) conversion.

**lemma** num-of-nat-One: \( n \leq 1 \Rightarrow num-of-nat \ n = One \)
by (cases \( n \)) simp-all

**lemma** num-of-nat-plus-distrib:
\( \emptyset \Rightarrow \emptyset < n \Rightarrow num-of-nat \ (m + n) = num-of-nat \ m + num-of-nat \ n \)
by (induct \( n \)) (auto simp add: add-One add-One-commute add-inc)

A double-and-decrement function.

**primrec** BitM :: \( num \Rightarrow num \)
where
BitM One = One
| BitM (Bit0 \ n) = Bit1 (BitM \ n)
| BitM (Bit1 \ n) = Bit1 (Bit0 \ n)
lemma BitM-plus-one: \( \text{BitM } n + \text{One} = \text{Bit0 } n \)
by (induct \( n \)) simp-all

lemma one-plus-BitM: \( \text{One} + \text{BitM } n = \text{Bit0 } n \)
unfolding add-One-commute BitM-plus-one ..

Squaring and exponentiation.
primrec sqr :: num \( \Rightarrow \) num
where
\( \text{sqr } \text{One} = \text{One} \)
| \( \text{sqr } (\text{Bit0 } n) = \text{Bit0 } (\text{Bit0 } (\text{sqr } n)) \)
| \( \text{sqr } (\text{Bit1 } n) = \text{Bit1 } (\text{Bit0 } (\text{sqr } n + n)) \)

primrec pow :: num \( \Rightarrow \) num \( \Rightarrow \) num
where
\( \text{pow } x \text{ One} = x \)
| \( \text{pow } x (\text{Bit0 } y) = \text{sqr } (\text{pow } x y) \)
| \( \text{pow } x (\text{Bit1 } y) = \text{sqr } (\text{pow } x y) * x \)

lemma nat-of-num-sqr: \( \text{nat-of-num } (\text{sqr } x) = \text{nat-of-num } x * \text{nat-of-num } x \)
by (induct \( x \)) (simp-all add: algebra-simps nat-of-num-add)

lemma sqr-conv-mult: \( \text{sqr } x = x * x \)
by (simp add: num-eq-iff nat-of-num-sqr nat-of-num-mult)

lemma num-double [simp]:
\( \text{num.Bit0 } \text{num.One} * n = \text{num.Bit0 } n \)
by (simp add: num-eq-iff nat-of-num-mult)

40.3 Binary numerals

We embed binary representations into a generic algebraic structure using numeral.

class numeral = one + semigroup-add
begin

primrec numeral :: num \( \Rightarrow \) 'a
where
\( \text{numeral-One} : \text{numeral } \text{One} = 1 \)
| \( \text{numeral-Bit0} : \text{numeral } (\text{Bit0 } n) = \text{numeral } n + \text{numeral } n \)
| \( \text{numeral-Bit1} : \text{numeral } (\text{Bit1 } n) = \text{numeral } n + \text{numeral } n + 1 \)

lemma numeral-code [code]:
\( \text{numeral One} = 1 \)
\( \text{numeral } (\text{Bit0 } n) = (\text{let } m = \text{numeral } n \text{ in } m + m) \)
\( \text{numeral } (\text{Bit1 } n) = (\text{let } m = \text{numeral } n \text{ in } m + m + 1) \)
by (simp-all add: Let-def)
lemma one-plus-numeral-commute: \( 1 + \text{numeral } x = \text{numeral } x + 1 \)
proof (induct \( x \))
case One
  then show \(?\) by simp
next
case Bit0
  then show \(?\) by (simp add: add.assoc [symmetric]) (simp add: add.assoc)
next
case Bit1
  then show \(?\) by (simp add: add.assoc [symmetric]) (simp add: add.assoc)
qed

lemma numeral-inc: \( \text{numeral } (\text{inc } x) = \text{numeral } x + 1 \)
proof (induct \( x \))
case One
  then show \(?\) by simp
next
case Bit0
  then show \(?\) by simp
next
case (Bit1 \( x \))
  have \( \text{numeral } x + (1 + \text{numeral } x) + 1 = \text{numeral } x + (\text{numeral } x + 1) + 1 \)
    by (simp only: one-plus-numeral-commute)
  with Bit1 show \(?\) by (simp add: add.assoc)
qed

declare numeral_simps [simp del]

abbreviation Numeral1 \( \equiv \text{numeral } One \)
declare numeral-One [code-post]
end

Numeral syntax.
syntax
-\text{Numeral} : \text{num-const} \Rightarrow 'a 

ML-file (Tools/numeral.ML)

parse-translation 
let
  fun numeral-tr [(c as Const (syntx-const \(-\text{constrain}, -\)) $ t $ u) =
    c $ numeral-tr \[t\] $ u
| numeral-tr [Const (num, -)] =
    (Numeral.mk-number-syntax o \#\text{value} o \text{Lexicon.read-num}) num
| numeral-tr ts = raise TERM (numeral-tr, ts);
40.4 Class-specific numeral rules

numeral is a morphism.

40.4.1 Structures with addition: class numeral

classical numeric

begin numeral

lemma numeral-add: numeral (m + n) = numeral m + numeral n
by (induct n rule: num-induct)
(simp-all only: numeral-One add-One add-inc numeral-inc add assoc)

lemma numeral-plus-numeral: numeral m + numeral n = numeral (m + n)
by (rule numeral-add [symmetric])

lemma numeral-plus-one: numeral n + 1 = numeral (n + One)
using numeral-add [of n One] by (simp add: numeral-One)

lemma one-plus-numeral: 1 + numeral n = numeral (One + n)
using numeral-add [of One n] by (simp add: numeral-One)

lemma one-add-one: 1 + 1 = 2
using numeral-add [of One One] by (simp add: numeral-One)
lemmas add-numeral-special =
    numeral-plus-one one-plus-numeral one-add-one
end

40.4.2 Structures with negation: class neg-numeral

class neg-numeral = numeral + group-add
begin

lemma uminus-numeral-One: - Numeral1 = - 1
    by (simp add: numeral-One)

Numerals form an abelian subgroup.

inductive is-num :: 'a ⇒ bool
where
  is-num 1
| is-num x ⇒ is-num (- x)
| is-num x ⇒ is-num y ⇒ is-num (x + y)

lemma is-num-numeral: is-num (numeral k)
    by (induct k) (simp-all add: numeral.simps is-num.intros)

lemma is-num-add-commute: is-num x ⇒ is-num y ⇒ x + y = y + x
    apply (induct x rule: is-num.induct)
      apply (induct y rule: is-num.induct)
        apply simp
      apply (rule-tac a=x in add-left-imp-eq)
      apply (rule-tac a=x in add-right-imp-eq)
      apply (simp add: add.assoc)
      apply (simp add: add.assoc [symmetric])
      apply (simp add: add.assoc)
      apply (rule-tac a=x in add-left-imp-eq)
      apply (rule-tac a=x in add-right-imp-eq)
      apply (simp add: add.assoc)
      apply (simp add: add.assoc [symmetric])
    done

lemma is-num-add-left-commute: is-num x ⇒ is-num y ⇒ x + (y + z) = y + (x + z)
    by (simp only: add.assoc [symmetric] is-num-add-commute)

lemmas is-num-normalize =
    add.assoc is-num-add-commute is-num-add-left-commute
    is-num.intros is-num-numeral
    minus-add
definition dbl :: 'a ⇒ 'a
  where dbl x = x + x

definition dbl-inc :: 'a ⇒ 'a
  where dbl-inc x = x + x + 1

definition dbl-dec :: 'a ⇒ 'a
  where dbl-dec x = x + x - 1

definition sub :: num ⇒ num ⇒ 'a
  where sub k l = numeral k - numeral l

lemma numeral-BitM: numeral (BitM n) = numeral (Bit0 n) - 1
  by (simp only: BitM-plus-one [symmetric] numeral-add numeral-One eq-diff-eq)

lemma dbl-simps [simp]:
  dbl (- numeral k) = - dbl (numeral k)
  dbl 0 = 0
  dbl 1 = 2
  dbl (- 1) = - 2
  dbl (numeral k) = numeral (Bit0 k)
  by (simp-all add: dbl-def numeral.simps minus-add)

lemma dbl-inc-simps [simp]:
  dbl-inc (- numeral k) = - dbl-dec (numeral k)
  dbl-inc 0 = 1
  dbl-inc 1 = 3
  dbl-inc (- 1) = - 1
  dbl-inc (numeral k) = numeral (Bit1 k)
  by (simp-all add: dbl-inc-def dbl-dec-def numeral.simps numeral-BitM is-num-normalize

lemma dbl-dec-simps [simp]:
  dbl-dec (- numeral k) = - dbl-inc (numeral k)
  dbl-dec 0 = - 1
  dbl-dec 1 = 1
  dbl-dec (- 1) = - 3
  dbl-dec (numeral k) = numeral (BitM k)
  by (simp-all add: dbl-dec-def dbl-inc-def numeral.simps numeral-BitM is-num-normalize)

lemma sub-num-simps [simp]:
  sub One One = 0
  sub One (Bit0 l) = - numeral (BitM l)
  sub One (Bit1 l) = - numeral (Bit0 l)
  sub (Bit0 k) One = numeral (BitM k)
  sub (Bit1 k) One = numeral (Bit0 k)
  sub (Bit0 k) (Bit0 l) = dbl (sub k l)
  sub (Bit0 k) (Bit1 l) = dbl-dec (sub k l)
THEORY "Num"

sub (Bit1 k) (Bit0 l) = dbl-inc (sub k l)
sub (Bit1 k) (Bit1 l) = dbl (sub k l)
by (simp-all add: dbl-def dbl-dec-def dbl-inc-def sub-def numeral.simps
numeral-BitM is-num-normalize del: add-uminus-conv-diff add: diff-conv-add-a minus)

lemma add-neg-numeral-simps:
numeral m + − numeral n = sub m n
− numeral m + numeral n = sub n m
− numeral m + − numeral n = − (numeral m + numeral n)
by (simp-all add: sub-def numeral-add numeral.simps is-num-normalize
del: add-uminus-conv-diff add: diff-conv-add-a minus)

lemma add-neg-numeral-special:
1 + − numeral m = sub One m
− numeral m + 1 = sub One m
numeral m + − 1 = sub m One
− 1 + − numeral n = sub n One
− 1 + − numeral n = − numeral (inc n)
− numeral m + − 1 = − numeral (inc m)
1 + − 1 = 0
− 1 + 1 = 0
− 1 + − 1 = − 2
by (simp-all add: sub-def numeral-add numeral.simps is-num-normalize right-minus
numeral-inc
del: add-uminus-conv-diff add: diff-conv-add-a minus)

lemma diff-numeral-simps:
numeral m − numeral n = sub m n
numeral m − − numeral n = numeral (m + n)
− numeral m − numeral n = − numeral (m + n)
− numeral m − − numeral n = sub n m
by (simp-all add: sub-def numeral-add numeral.simps is-num-normalize
del: add-uminus-conv-diff add: diff-conv-add-a minus)

lemma diff-numeral-special:
1 − numeral n = sub One n
numeral m − 1 = sub m One
1 − − numeral n = numeral (One + n)
− numeral m − 1 = − numeral (m + One)
− 1 − − numeral n = − numeral (inc n)
numeral m − − 1 = numeral (inc m)
− 1 − − numeral n = sub n One
− numeral m − − 1 = sub One m
1 − 1 = 0
− 1 − 1 = − 2
1 − − 1 = 2
− 1 − − 1 = 0
by (simp-all add: sub-def numeral-add numeral.simps is-num-normalize numeral-inc
del: add-uminus-conv-diff add: diff-conv-add-a minus)
40.4.3 Structures with multiplication: class semiring-numeral

class semiring-numeral = semiring + monoid-mult
begin
subclass numeral ..

lemma numeral-mult: numeral (m * n) = numeral m * numeral n
  by (induct n rule: num-induct)
    (simp-all add: numeral-One mult-inc numeral-inc numeral-add distrib-left)

lemma numeral-times-numeral: numeral m * numeral n = numeral (m * n)
  by (rule numeral-mult [symmetric])

lemma mult-2: 2 * z = z + z
  by (simp add: one-add-one [symmetric] distrib-right)

lemma mult-2-right: z * 2 = z + z
  by (simp add: one-add-one [symmetric] distrib-left)

lemma left-add-twice:
  a + (a + b) = 2 * a + b
  by (simp add: mult-2 ac-simps)
end

40.4.4 Structures with a zero: class semiring-1

context semiring-1
begin
subclass semiring-numeral ..

lemma of-nat-numeral [simp]: of-nat (numeral n) = numeral n
  by (induct n) (simp-all only: numeral.simps numeral-class numeral.simps of-nat-add of-nat-1)

lemma numeral-unfold-funpow:
  numeral k = ((+) 1 ^^ numeral k) 0
  unfolding of-nat-def [symmetric] by simp
end
context
  includes lifting-syntax
begin
lemma transfer-rule-numeral:
  \( ((\equiv) \equiv \Rightarrow R) \text{ numeral numeral} \)
  if [transfer-rule]: \( R \equiv 0 \equiv R 1 \equiv 1 \)
  \( (R \equiv \Rightarrow R \equiv \Rightarrow R) \equiv (\equiv) \equiv \Rightarrow \)
  for \( R :: 'a::semiring-1 \Rightarrow 'b::semiring-1 \Rightarrow \text{ bool} \)

proof -
  have \( ((\equiv) \equiv \Rightarrow R) \lambda k. \((\equiv) \equiv 1 \equiv \text{ numeral } k) \equiv 0 \)
  by transfer-prover
  then show ?thesis
  by (simp flip: numeral-unfold-funpow [abs-def])

qed

end

lemma nat-of-num-numeral [code-abbrev]: \( \text{ nat-of-num } = \text{ numeral } \)

proof
  fix\( n \)
  have \( \text{ numeral } n = \text{ nat-of-num } n \)
    by (induct n) (simp-all add: numeral.simps)
  then show \( \text{ nat-of-num } n = \text{ numeral } n \)
    by simp

qed

lemma nat-of-num-code [code]:
  \( \text{ nat-of-num } \text{ One } = 1 \)
  \( \text{ nat-of-num } (\text{ Bit0 } n) = (\text{ let } m = \text{ nat-of-num } n \text{ in } m + m) \)
  \( \text{ nat-of-num } (\text{ Bit1 } n) = (\text{ let } m = \text{ nat-of-num } n \text{ in } \text{ Suc } (m + m)) \)
  by (simp-all add: Let-def)

40.4.5 Equality: class semiring-char-0

context semiring-char-0

begin

lemma numeral-eq-iff: numeral \( m = \text{ numeral } n \longleftrightarrow m = n \)
  by (simp only: of-nat-numeral [symmetric] nat-of-num-numeral [symmetric]
    of-nat-eq-iff num-eq-iff)

lemma numeral-eq-one-iff: numeral \( n = 1 \longleftrightarrow n = \text{ One } \)
  by (rule numeral-eq-iff [of n One, unfolded numeral-One])

lemma one-eq-numeral-iff: \( 1 = \text{ numeral } n \longleftrightarrow \text{ One } = n \)
  by (rule numeral-eq-iff [of One n, unfolded numeral-One])

lemma numeral-neq-zero: numeral \( n \neq 0 \)

lemma zero-neq-numeral: \( 0 \neq \text{ numeral } n \)
unfolding eq-commute [of 0] by (rule numeral-neq-zero)

lemmas eq-numeral-simps [simp] =
  numeral-eq-iff
  numeral-eq-one-iff
  one-eq-numeral-iff
  numeral-neq-zero
  zero-neq-numeral

end

40.4.6 Comparisons: class linordered-nonzero-semiring

class linordered-nonzero-semiring

begin

lemma numeral-le-iff: numeral m ≤ numeral n ←→ m ≤ n
proof
  have of-nat (numeral m) ≤ of-nat (numeral n) ←→ m ≤ n
  by (simp only: less-eq-num-def nat-of-num-numeral of-nat-le-iff)
  then show ?thesis by simp
qed

lemma one-le-numeral: 1 ≤ numeral n
  using numeral-le-iff [of num.One n] by (simp add: numeral-One)

lemma numeral-le-one-iff: numeral n ≤ 1 ←→ n ≤ num.One
  using numeral-le-iff [of n num.One] by (simp add: numeral-One)

lemma numeral-less-iff: numeral m < numeral n ←→ m < n
proof
  have of-nat (numeral m) < of-nat (numeral n) ←→ m < n
  unfolding less-num-def nat-of-num-numeral of-nat-less-iff
  then show ?thesis by simp
qed

lemma not-numeral-less-one: ¬ numeral n < 1
  using numeral-less-iff [of n num.One] by (simp add: numeral-One)

lemma one-less-numeral-iff: 1 < numeral n ←→ num.One < n
  using numeral-less-iff [of num.One n] by (simp add: numeral-One)

lemma zero-le-numeral: 0 ≤ numeral n
  using dual-order.trans one-le-numeral zero-le-one by blast

lemma zero-less-numeral: 0 < numeral n
  using less-linear not-numeral-less-one order.strict-trans zero-less-one by blast

lemma not-numeral-le-zero: ¬ numeral n ≤ 0
by (simp add: not-le zero-less-numeral)

lemma not-numeral-less-zero: ¬ numeral n < 0
  by (simp add: not-less zero-le-numeral)

lemmas le-numeral-extra =
  zero-le-one not-one-le-zero
  order-refl [of 0] order-refl [of 1]

lemmas less-numeral-extra =
  zero-less-one not-one-less-zero
  less-irrefl [of 0] less-irrefl [of 1]

lemmas le-numeral-simps [simp] =
  numeral-le-iff
  one-le-numeral
  numeral-le-one-iff
  zero-le-numeral
  not-numeral-le-zero

lemmas less-numeral-simps [simp] =
  numeral-less-iff
  one-less-numeral-iff
  not-numeral-less-one
  zero-less-numeral
  not-numeral-less-zero

lemma min-0-1 [simp]:
  fixes min' :: 'a ⇒ 'a ⇒ 'a
  defines min' ≡ min
  shows
    min' 0 1 = 0
    min' 1 0 = 0
    min' 0 (numeral x) = 0
    min' (numeral x) 0 = 0
    min' 1 (numeral x) = 1
    min' (numeral x) 1 = 1
  by (simp-all add: min'-def min-def le-num-One-iff)

lemma max-0-1 [simp]:
  fixes max' :: 'a ⇒ 'a ⇒ 'a
  defines max' ≡ max
  shows
    max' 0 1 = 1
    max' 1 0 = 1
    max' 0 (numeral x) = numeral x
    max' (numeral x) 0 = numeral x
    max' 1 (numeral x) = numeral x
    max' (numeral x) 1 = numeral x
by (simp-all add: max'-def max-def le-num-One-iff)
end

Unfold min and max on numerals.

lemmas max-number-of [simp] =
  max-def [of numeral u numeral v]
  max-def [of numeral u - numeral v]
  max-def [of - numeral u numeral v]
  max-def [of - numeral u - numeral v] for u v

lemmas min-number-of [simp] =
  min-def [of numeral u numeral v]
  min-def [of numeral u - numeral v]
  min-def [of - numeral u numeral v]
  min-def [of - numeral u - numeral v] for u v

40.4.7 Multiplication and negation: class ring-1

context ring-1
begin

subclass neg-numeral ..

lemma mult-neg-numeral-simps:
  - numeral m * - numeral n = numeral (m * n)
  - numeral m * numeral n = - numeral (m * n)
  numeral m * - numeral n = - numeral (m * n)
by (simp-all only: mult-minus-left mult-minus-right minus-minus numeral-mult)

lemma mult-minus1 [simp]: - 1 * z = - z
by (simp add: numeral.simps)

lemma mult-minus1-right [simp]: z * - 1 = - z
by (simp add: numeral.simps)
end

40.4.8 Equality using iszero for rings with non-zero characteristic

context ring-1
begin

definition iszero :: 'a ⇒ bool
  where iszero z ⟷ z = 0

lemma iszero-0 [simp]: iszero 0
by (simp add: iszero-def)

lemma not-iszero-1 [simp]: ¬ iszero 1
by (simp add: iszero-def)

lemma not-iszero-Numeral1: ¬ iszero Numeral1
  by (simp add: numeral-One)

lemma not-iszero-neg-1 [simp]: ¬ iszero (− 1)
  by (simp add: iszero-def)

lemma not-iszero-neg-Numeral1: ¬ iszero (− Numeral1)
  by (simp add: numeral-One)

lemma iszero-neg-numeral [simp]: iszero (− numeral w) ⟷ iszero (numeral w)
  unfolding iszero-def by (rule neg-equal-0-iff-equal)

lemma eq-iff-iszero-diff: x = y ⟷ iszero (x − y)
  unfolding iszero-def by (rule eq-iff-diff-eq-0)

The eq-numeral-iff-iszero lemmas are not declared [simp] by default, because
for rings of characteristic zero, better simp rules are possible. For a type like
integers mod n, type-instantiated versions of these rules should be added to
the simplifier, along with a type-specific rule for deciding propositions of the
form iszero (numeral w).

bh: Maybe it would not be so bad to just declare these as simp rules anyway?
I should test whether these rules take precedence over the ring-char-0 rules
in the simplifier.

lemma eq-numeral-iff-iszero:
  numeral x = numeral y ⟷ iszero (sub x y)
  numeral x = − numeral y ⟷ iszero (numeral (x + y))
  − numeral x = numeral y ⟷ iszero (numeral (x + y))
  − numeral x = − numeral y ⟷ iszero (sub x y)
  numeral x = 1 ⟷ iszero (sub x One)
  1 = numeral y ⟷ iszero (sub One y)
  − numeral x = 1 ⟷ iszero (numeral (x + One))
  1 = − numeral y ⟷ iszero (numeral (One + y))
  numeral x = 0 ⟷ iszero (numeral x)
  0 = numeral y ⟷ iszero (numeral y)
  − numeral x = 0 ⟷ iszero (numeral x)
  0 = − numeral y ⟷ iszero (numeral y)
  unfolding eq-iff-iszero-diff diff-numeral-simps diff-numeral-special
  by simp-all

end

40.4.9 Equality and negation: class ring-char-0

context ring-char-0
begin
lemma not-iszero-numeral [simp]: ¬ iszero (numeral w)
  by (simp add: iszero-def)

lemma neg-numeral-eq-iff: ¬ numeral m = ¬ numeral n ←→ m = n
  by simp

lemma numeral-neq-neg-numeral: numeral m ≠ numeral n
  by (simp add: eq-neg-iff-add-eq-0 numeral-plus-numeral)

lemma neg-numeral-neq-numeral: ¬ numeral m ≠ numeral n
  by (rule numeral-neq-neg-numeral [symmetric])

lemma zero-neq-neg-numeral: 0 ≠ numeral n
  by simp

lemma neg-numeral-neq-zero: ¬ numeral n ≠ 0
  by simp

lemma one-neq-neg-numeral: 1 ≠ numeral n
  using numeral-neq-neg-numeral [of One n] by (simp add: numeral-One)

lemma neg-one-neq-numeral: ¬ 1 ≠ numeral n
  using neg-numeral-neq-numeral [of One n] by (simp add: numeral-One)

lemma numeral-neq-neg-one: numeral n ≠ -1
  using numeral-neq-neg-one [of One n] by (simp add: numeral-One)

lemma neg-one-neq-numeral-one: ¬ 1 ≠ numeral One
  using neg-one-neq-numeral-one [of numeral One] by (auto simp add: numeral-One)

lemma numeral-neq-one: numeral n ≠ 0
  using numeral-neq-one [of One n] by (simp add: numeral-One)

lemma neg-one-neq-zero: ¬ 1 ≠ 0
  by simp

lemma zero-neq-neg-one: 0 ≠ -1
  by simp

lemma one-neq-one: ¬ 1 ≠ 1
  using one-neq-one [of One One] by (simp only: numeral-One not-False-eq-True)

lemma not-one-neq-one: ¬ 1 ≠ 1
  using one-neq-one [of One One] by (simp only: numeral-One not-False-eq-True)

lemmas eq-numeral-simps [simp] =
40.4.10 Structures with negation and order: class linordered-idom

context linordered-idom
begin

subclass ring-char-0 ..

lemma neg-numeral-le-iff: \(-\numeral m \leq \numeral n \iff n \leq m\)
  by (simp only: neg-le-iff-le numeral-le-iff)

lemma neg-numeral-less-iff: \(-\numeral m < \numeral n \iff n < m\)
  by (simp only: neg-less-iff-less numeral-less-iff)

lemma neg-numeral-less-zero: \(-\numeral n < 0\)
  by (simp only: neg-less-0-iff-less zero-less-numeral)

lemma neg-numeral-le-zero: \(-\numeral n \leq 0\)
  by (simp only: neg-le-0-iff-le zero-le-numeral)

lemma not-zero-less-neg-numeral: \(\neg 0 < -\numeral n\)
  by (simp only: not-less neg-numeral-le-zero)

lemma not-zero-le-neg-numeral: \(\neg 0 \leq -\numeral n\)
  by (simp only: not-le neg-numeral-less-zero)

lemma neg-numeral-less-numeral: \(-\numeral m < \numeral n\)
  using neg-numeral-less-zero less-numeral-le-zero by (rule less-trans)

lemma neg-numeral-le-numeral: \(-\numeral m \leq \numeral n\)
  by (simp only: less-imp-le neg-numeral-less-numeral)

lemma not-numeral-less-neg-numeral: \(\neg \numeral m < -\numeral n\)
  by (simp only: not-less neg-numeral-le-numeral)

lemma not-numeral-le-neg-numeral: \(\neg \numeral m \leq -\numeral n\)
  by (simp only: not-le neg-numeral-less-numeral)

lemma neg-numeral-less-one: \(-\numeral m < 1\)
by (rule neg-numeral-less-numeral [of \( m \) One, unfolded numeral-One])

lemma neg-numeral-le-one: \(-\) numeral \( m \) \( \leq \) 1
  by (rule neg-numeral-le-numeral [of \( m \) One, unfolded numeral-One])

lemma not-one-less-neg-numeral: \( -1 < -\) numeral \( m \)
  by (simp only: not-less neg-numeral-le-one)

lemma not-one-le-neg-numeral: \( -1 \leq -\) numeral \( m \)
  by (simp only: not-le neg-numeral-less-one)

lemma not-numeral-less-neg-one:
  \( \neg \) numeral \( m \) \(<\) \(-1\)
  using not-numeral-less-neg-numeral [of \( m \) One]
  by (simp add: numeral-One)

lemma not-numeral-le-neg-one:
  \( \neg \) numeral \( m \) \( \leq \) \(-1\)
  using not-numeral-le-neg-numeral [of \( m \) One]
  by (simp add: numeral-One)

lemma neg-one-less-numeral:
  \(-1 <\) numeral \( m \)
  using neg-numeral-less-numeral [of One \( m \)]
  by (simp add: numeral-One)

lemma neg-one-le-numeral:
  \(-1 \leq \) numeral \( m \)
  using neg-numeral-le-numeral [of One \( m \)]
  by (simp add: numeral-One)

lemma neg-numeral-less-neg-one-iff:
  \( -\) numeral \( m \) \(<\) \(-1\) \( \iff \) \( m \) \( \neq \) One
  by (cases \( m \)) simp-all

lemma neg-numeral-le-neg-one:
  \( -\) numeral \( m \) \( \leq \) \(-1\)
  by simp

lemma not-neg-one-less-neg-numeral:
  \( \neg -1 < -\) numeral \( m \)
  by simp

lemma not-neg-one-le-neg-numeral-iff:
  \( -1 \leq -\) numeral \( m \) \( \iff \) \( m \) \( \neq \) One
  by (cases \( m \)) simp-all

lemma sub-non-negative:
  sub \( n \) \( m \) \( \geq \) 0 \( \iff \) \( n \) \( \geq \) \( m \)
  by (simp only: sub-def le-diff-eq) simp

lemma sub-positive:
  sub \( n \) \( m \) \( > \) 0 \( \iff \) \( n \) \( > \) \( m \)
  by (simp only: sub-def less-diff-eq) simp

lemma sub-non-positive:
  sub \( n \) \( m \) \( \leq \) 0 \( \iff \) \( n \) \( \leq \) \( m \)
  by (simp only: sub-def diff-le-eq) simp

lemma sub-negative:
  sub \( n \) \( m \) \( < \) 0 \( \iff \) \( n \) \( < \) \( m \)
  by (simp only: sub-def diff-less-eq) simp

lemmas le-numeral-simps [simp] =
  neg-numeral-le-iff
neg-numeral-le-numeral not-numeral-le-neg-numeral
neg-numeral-le-zero not-zero-le-neg-numeral
neg-numeral-le-one not-one-le-neg-numeral
neg-one-le-numeral not-numeral-le-neg-one
neg-numeral-le-neg-one not-neg-one-le-neg-numeral-iff

lemma le-minus-one-simps [simp]:
\[
-1 \leq 0
-1 \leq 1
\neg 0 \leq -1
\neg 1 \leq -1
\]
by simp-all

lemmas less-neg-numeral-simps [simp] =
neg-numeral-less-iff
neg-numeral-less-numeral not-numeral-less-neg-numeral
neg-numeral-less-zero not-zero-less-neg-numeral
neg-numeral-less-one not-one-less-neg-numeral
neg-one-less-numeral not-numeral-less-neg-one
neg-numeral-less-neg-one-iff not-neg-one-less-neg-numeral

lemma less-minus-one-simps [simp]:
\[
-1 < 0
-1 < 1
\neg 0 < -1
\neg 1 < -1
\]
by (simp-all add: less-le)

lemma abs-numeral [simp]: \(|\text{numeral } n\) = numeral n
by simp

lemma abs-neg-numeral [simp]: \(|-\text{numeral } n\) = numeral n
by (simp only: abs-minus-cancel abs-numeral)

lemma abs-neg-one [simp]: \(|-1\) = 1
by simp

end

40.4.11 Natural numbers

lemma numeral-num-of-nat:
\text{numeral} \ (\text{num-of-nat } n) = n if n > 0
using that nat-of-num-numeral num-of-nat-inverse by simp

lemma Suc-1 [simp]: Suc 1 = 2
unfolding Suc-eq-plus1 by (rule one-add-one)

lemma Suc-numeral [simp]: Suc (numeral n) = numeral (n + One)
unfolding Suc-eq-plus1 by (rule numeral-plus-one)

definition pred-numeral :: num ⇒ nat
  where pred-numeral k = numeral k - 1

declare [[code drop: pred-numeral]]

lemma numeral-eq-Suc: numeral k = Suc (pred-numeral k)
  by (simp add: pred-numeral-def)

lemma eval-nat-numeral:
  numeral One = Suc 0
  numeral (Bit0 n) = Suc (numeral (BitM n))
  numeral (Bit1 n) = Suc (numeral (Bit0 n))
  by (simp-all add: numeral.simps BitM-plus-one)

lemma pred-numeral-simps [simp]:
  pred-numeral One = 0
  pred-numeral (Bit0 k) = numeral (BitM k)
  pred-numeral (Bit1 k) = numeral (Bit0 k)
  by (simp-all only: pred-numeral-def eval-nat-numeral diff-Suc-Suc diff-0)

lemma pred-numeral-inc [simp]:
  pred-numeral (Num.inc k) = numeral k
  by (simp only: pred-numeral-def numeral-inc diff-add-inverse2)

lemma numeral-2-eq-2: 2 = Suc (Suc 0)
  by (simp add: eval-nat-numeral)

lemma numeral-3-eq-3: 3 = Suc (Suc 0))
  by (simp add: eval-nat-numeral)

lemma numeral-1-eq-Suc-0: Numeral1 = Suc 0
  by (simp only: numeral-One One-nat-def)

lemma Suc-nat-number-of-add: Suc (numeral v + n) = numeral (v + One) + n
  by simp

lemma numerals: Numeral1 = (1::nat) 2 = Suc (Suc 0)
  by (rule numeral-One) (rule numeral-2-eq-2)

lemmas numeral-nat = eval-nat-numeral BitM.simps One-nat-def

Comparisons involving Suc.

lemma eq-numeral-Suc [simp]: numeral k = Suc n ←→ pred-numeral k = n
  by (simp add: numeral-eq-Suc)

lemma Suc-eq-numeral [simp]: Suc n = numeral k ←→ n = pred-numeral k
  by (simp add: numeral-eq-Suc)
lemma less-numeral-Suc [simp]: numeral k < Suc n ←→ pred-numeral k < n
by (simp add: numeral-eq-Suc)

lemma less-Suc-numeral [simp]: Suc n < numeral k ←→ n < pred-numeral k
by (simp add: numeral-eq-Suc)

lemma le-numeral-Suc [simp]: numeral k ≤ Suc n ←→ pred-numeral k ≤ n
by (simp add: numeral-eq-Suc)

lemma le-Suc-numeral [simp]: Suc n ≤ numeral k ←→ n ≤ pred-numeral k
by (simp add: numeral-eq-Suc)

lemma diff-Suc-numeral [simp]: Suc n − numeral k = n − pred-numeral k
by (simp add: numeral-eq-Suc)

lemma diff-numeral-Suc [simp]: numeral k − Suc n = pred-numeral k − n
by (simp add: numeral-eq-Suc)

lemma max-Suc-numeral [simp]: max (Suc n) (numeral k) = Suc (max n (pred-numeral k))
by (simp add: numeral-eq-Suc)

lemma max-numeral-Suc [simp]: max (numeral k) (Suc n) = Suc (max (pred-numeral k) n)
by (simp add: numeral-eq-Suc)

lemma min-Suc-numeral [simp]: min (Suc n) (numeral k) = Suc (min n (pred-numeral k))
by (simp add: numeral-eq-Suc)

lemma min-numeral-Suc [simp]: min (numeral k) (Suc n) = Suc (min (pred-numeral k) n)
by (simp add: numeral-eq-Suc)

For case-nat and rec-nat.

lemma case-nat-numeral [simp]: case-nat a f (numeral v) = (let pv = pred-numeral v in f pv)
by (simp add: numeral-eq-Suc)

lemma case-nat-add-eq-if [simp]:
case-nat a f ((numeral v) + n) = (let pv = pred-numeral v in f (pv + n))
by (simp add: numeral-eq-Suc)

lemma rec-nat-numeral [simp]:
rec-nat a f (numeral v) = (let pv = pred-numeral v in f pv (rec-nat a f pv))
by (simp add: numeral-eq-Suc Let-def)

lemma rec-nat-add-eq-if [simp]:

rec-nat a f (numeral v + n) = (let pv = pred-numeral v in f (pv + n))
  by (simp add: numeral-eq-Suc Let-def)

Case analysis on \( n < (2::'a) \).

\[
\text{lemma } \text{less-2-cases}: n < 2 \implies n = 0 \lor n = \text{Suc 0}
\]
  by (auto simp add: numeral-2-eq-2)

\[
\text{lemma } \text{less-2-cases-iff}: n < 2 \iff n = 0 \lor n = \text{Suc 0}
\]
  by (auto simp add: numeral-2-eq-2)

Removal of Small Numerals: 0, 1 and (in additive positions) 2.

bh: Are these rules really a good idea? LCP: well, it already happens for 0 and 1!

\[
\text{lemma } \text{add-2-eq-Suc} \quad \text{[simp] } 2 + n = \text{Suc (Suc n)}
\]
  by simp

\[
\text{lemma } \text{add-2-eq-Suc'} \quad \text{[simp] } n + 2 = \text{Suc (Suc n)}
\]
  by simp

Can be used to eliminate long strings of Sucs, but not by default.

\[
\text{lemma } \text{Suc3-eq-add-3}: \text{Suc (Suc (Suc n))} = 3 + n
\]
  by simp

\[
\text{lemmas } \text{nat-1-add-1} = \text{one-add-one} \quad \text{[where } 'a=nat]\)
\]

40.5 Particular lemmas concerning \( 2::'a \)

context linordered-field

begin

subclass field-char-0 ..

\[
\text{lemma } \text{half-gt-zero-iff}: 0 < a / 2 \iff 0 < a
\]
  by (auto simp add: field-simps)

\[
\text{lemma } \text{half-gt-zero} \quad \text{[simp] } 0 < a \implies 0 < a / 2
\]
  by (simp add: half-gt-zero-iff)

end

40.6 Numerical equations as default simplification rules

declare (in numeral) numeral-One [simp]
declare (in numeral) numeral-plus-numeral [simp]
declare (in numeral) add-numeral-special [simp]
declare (in neg-numeral) add-neg-numeral-simps [simp]
declare (in neg-numeral) add-neg-numeral-special [simp]
40.6.1 Special Simplification for Constants

These distributive laws move literals inside sums and differences.

\begin{itemize}
  \item \textbf{lemmas} \texttt{distrib-right-numeral [simp] = distrib-right \ [of \ - \ numeral \ v]} \textcolor{red}{\textbf{for \ v}}
  \item \textbf{lemmas} \texttt{distrib-left-numeral [simp] = distrib-left \ [of \ numeral \ v]} \textcolor{red}{\textbf{for \ v}}
  \item \textbf{lemmas} \texttt{left-diff-distrib-numeral [simp] = left-diff-distrib \ [of \ - \ numeral \ v]} \textcolor{red}{\textbf{for \ v}}
  \item \textbf{lemmas} \texttt{right-diff-distrib-numeral [simp] = right-diff-distrib \ [of \ numeral \ v]} \textcolor{red}{\textbf{for \ v}}
\end{itemize}

These are actually for fields, like real

\begin{itemize}
  \item \textbf{lemmas} \texttt{zero-less-divide-iff-numeral [simp, no-atp] = zero-less-divide-iff \ [of \ numeral \ w]} \textcolor{red}{\textbf{for \ w}}
  \item \textbf{lemmas} \texttt{divide-less-0-iff-numeral [simp, no-atp] = divide-less-0-iff \ [of \ numeral \ w]} \textcolor{red}{\textbf{for \ w}}
  \item \textbf{lemmas} \texttt{zero-le-divide-iff-numeral [simp, no-atp] = zero-le-divide-iff \ [of \ numeral \ w]} \textcolor{red}{\textbf{for \ w}}
  \item \textbf{lemmas} \texttt{divide-le-0-iff-numeral [simp, no-atp] = divide-le-0-iff \ [of \ numeral \ w]} \textcolor{red}{\textbf{for \ w}}
\end{itemize}

Replaces \texttt{inverse \ #nn} by \texttt{1/\#nn}. It looks strange, but then other simprocs simplify the quotient.

\begin{itemize}
  \item \textbf{lemmas} \texttt{inverse-eq-divide-numeral [simp] = inverse-eq-divide \ [of \ numeral \ w]} \textcolor{red}{\textbf{for \ w}}
  \item \textbf{lemmas} \texttt{inverse-eq-divide-neg-numeral [simp] = inverse-eq-divide \ [of \ - \ numeral \ w]} \textcolor{red}{\textbf{for \ w}}
\end{itemize}

These laws simplify inequalities, moving unary minus from a term into the literal.

\begin{itemize}
  \item \textbf{lemmas} \texttt{equation-minus-iff-numeral [no-atp] = equation-minus-iff \ [of \ numeral \ v]} \textcolor{red}{\textbf{for \ v}}
  \item \textbf{lemmas} \texttt{minus-equation-iff-numeral [no-atp] = minus-equation-iff \ [of \ - \ numeral \ v]} \textcolor{red}{\textbf{for \ v}}
  \item \textbf{lemmas} \texttt{le-minus-iff-numeral [no-atp] = le-minus-iff \ [of \ numeral \ v]} \textcolor{red}{\textbf{for \ v}}
  \item \textbf{lemmas} \texttt{minus-le-iff-numeral [no-atp] = minus-le-iff \ [of \ numeral \ v]} \textcolor{red}{\textbf{for \ v}}
  \item \textbf{lemmas} \texttt{less-minus-iff-numeral [no-atp] = less-minus-iff \ [of \ numeral \ v]} \textcolor{red}{\textbf{for \ v}}
\end{itemize}
lemmas minus-less-iff-numeral \[\text{no-atp}\] = minus-less-iff \[\of - \text{numeral } v\] for \(v\)

Cancellation of constant factors in comparisons (< and \(\leq\))

lemmas mult-less-cancel-left-numeral \[\simp, \text{no-atp}\] = mult-less-cancel-left \[\of \text{numeral } v\] for \(v\)
lemmas mult-less-cancel-right-numeral \[\simp, \text{no-atp}\] = mult-less-cancel-right \[\of - \text{numeral } v\] for \(v\)
lemmas mult-le-cancel-left-numeral \[\simp, \text{no-atp}\] = mult-le-cancel-left \[\of \text{numeral } v\] for \(v\)
lemmas mult-le-cancel-right-numeral \[\simp, \text{no-atp}\] = mult-le-cancel-right \[\of - \text{numeral } v\] for \(v\)

Multiplying out constant divisors in comparisons (<, \(\leq\) and =)

called-theorems divide-const-simps simplification rules to simplify comparisons involving constant divisors

lemmas le-divide-eq-numeral1 \[\simp, \text{divide-const-simps}\] =
pos-le-divide-eq \[\of \text{numeral } w, \ OF \text{zero-less-numeral}\]
neg-le-divide-eq \[\of - \text{numeral } w, \ OF \neg\text{-numeral-less-zero}\] for \(w\)

lemmas divide-le-eq-numeral1 \[\simp, \text{divide-const-simps}\] =
pos-divide-le-eq \[\of \text{numeral } w, \ OF \text{zero-less-numeral}\]
neg-divide-le-eq \[\of - \text{numeral } w, \ OF \neg\text{-numeral-less-zero}\] for \(w\)

lemmas less-divide-eq-numeral1 \[\simp, \text{divide-const-simps}\] =
pos-less-divide-eq \[\of \text{numeral } w, \ OF \text{zero-less-numeral}\]
neg-less-divide-eq \[\of - \text{numeral } w, \ OF \neg\text{-numeral-less-zero}\] for \(w\)

lemmas divide-less-eq-numeral1 \[\simp, \text{divide-const-simps}\] =
pos-divide-less-eq \[\of \text{numeral } w, \ OF \text{zero-less-numeral}\]
neg-divide-less-eq \[\of - \text{numeral } w, \ OF \neg\text{-numeral-less-zero}\] for \(w\)

lemmas eq-divide-eq-numeral1 \[\simp, \text{divide-const-simps}\] =
\(eq\)-divide-eq \[\of - - \text{numeral } w\]
\(eq\)-divide-eq \[\of - - - \text{numeral } w\] for \(w\)

lemmas divide-eq-eq-numeral1 \[\simp, \text{divide-const-simps}\] =
divide-eq-eq \[\of \text{- numeral } w\]
divide-eq-eq \[\of - - \text{numeral } w\] for \(w\)

40.6.2 Optional Simplification Rules Involving Constants

Simplify quotients that are compared with a literal constant.

lemmas le-divide-eq-numeral \[\text{divide-const-simps}\] =
le-divide-eq \[\of \text{numeral } w\]
le-divide-eq \[\of - \text{numeral } w\] for \(w\)
lemmas \texttt{divide-le-eq-numeral} [divide-const-simps] =  
\texttt{divide-le-eq} \ [\texttt{if} \ - \ - \ \texttt{numeral} \ \texttt{w}] \ \textbf{for} \ \texttt{w}  

lemmas \texttt{less-divide-eq-numeral} [divide-const-simps] =  
\texttt{less-divide-eq} \ [\texttt{of} \ \texttt{numeral} \ \texttt{w}] \ \textbf{for} \ \texttt{w}  

lemmas \texttt{divide-less-eq-numeral} [divide-const-simps] =  
\texttt{divide-less-eq} \ [\texttt{of} \ - \ \texttt{numeral} \ \texttt{w}] \ \textbf{for} \ \texttt{w}  

lemmas \texttt{eq-divide-eq-numeral} [divide-const-simps] =  
\texttt{eq-divide-eq} \ [\texttt{of} \ \texttt{numeral} \ \texttt{w}] \ \textbf{for} \ \texttt{w}  

lemmas \texttt{divide-eq-eq-numeral} [divide-const-simps] =  
\texttt{divide-eq-eq} \ [\texttt{of} \ - \ \texttt{numeral} \ \texttt{w}] \ \textbf{for} \ \texttt{w}  

Not good as automatic simprules because they cause case splits.

lemmas [divide-const-simps] =  
le-divide-eq-1 \ \texttt{divide-le-eq-1} \ \texttt{less-divide-eq-1} \ \texttt{divide-less-eq-1}

40.7 Setting up simprocs

lemma \texttt{mult-numeral-1}: \texttt{Numeral1} \ast \ a = a  
\textbf{for} \ \texttt{a} :: \texttt{'}a::semiring-numeral  
\textbf{by} \ \texttt{simp}  

lemma \texttt{mult-numeral-1-right}: \ a \ast \texttt{Numeral1} = a  
\textbf{for} \ \texttt{a} :: \texttt{'}a::semiring-numeral  
\textbf{by} \ \texttt{simp}  

lemma \texttt{divide-numeral-1}: \ a \ / \ \texttt{Numeral1} = a  
\textbf{for} \ \texttt{a} :: \texttt{'}a::field  
\textbf{by} \ \texttt{simp}  

lemma \texttt{inverse-numeral-1}: \ \texttt{inverse} \ \texttt{Numeral1} = (\texttt{Numeral1}::\texttt{'a::division-ring})  
\textbf{by} \ \texttt{simp}  

Theorem lists for the cancellation simprocs. The use of a binary numeral for 1 reduces the number of special cases.

lemma \texttt{mult-1s-semiring-numeral}:  
\texttt{Numeral1} \ast \ a = a  
a \ast \texttt{Numeral1} = a  
\textbf{for} \ \texttt{a} :: \texttt{'}a::semiring-numeral  
\textbf{by} \ \texttt{simp-all}
THEORY "Num"

lemma mult-1s-ring-1:
- Numeral1 * b = - b
b * - Numeral1 = - b
for b :: 'a::ring-1
by simp-all

lemmas mult-1s = mult-1s-semiring-numeral mult-1s-ring-1

setup:
Reorient-Proc.add
(fn Const (const-name numeral, -) $ - => true
 | Const (const-name uminus, -) $ (Const (const-name numeral, -) $ -)
 => true
 | - => false) 

simproc-setup reorient-numeral (numeral w = x | - numeral w = y) =
Reorient-Proc.proc

40.7.1 Simplification of arithmetic operations on integer constants

lemmas arith-special =
add-numeral-special add-neg-numeral-special
diff-numeral-special

lemmas arith-extra-simps =
numeral-plus-numeral add-neg-numeral-simps add-0-left add-0-right
minus-zero
diff-numeral-simps diff-0 diff-0-right
numeral-times-numeral mult-neg-numeral-simps
mult-zero-left mult-zero-right
abs-numeral abs-neg-numeral

For making a minimal simpset, one must include these default simprules.
Also include simp-thms.

lemmas arith-simps =
add-num-simps mult-num-simps sub-num-simps
BitM.simps dbl-simps dbl-inc-simps dbl-dec-simps
abs-zero abs-one arith-extra-simps

lemmas more-arith-simps =
neg-le-iff-le
minus-zero left-minus right-minus
mult-1-left mult-1-right
mult-minus-left mult-minus-right
minus-add-distrib minus-minus mult.assoc

lemmas of-nat-simps =
of-nat-0 of-nat-1 of-nat-Suc of-nat-add of-nat-mult

Simplification of relational operations.

**lemmas** eq-numeral-extra =
zero-neq-one one-neq-zero

**lemmas** rel-simps =
le-num-simps less-num-simps eq-num-simps
le-numeral-simps le-neg-numeral-simps le-minus-one-simps le-numeral-extra
less-numeral-simps less-neg-numeral-simps less-minus-one-simps less-numeral-extra
eq-numeral-simps eq-neg-numeral-simps eq-numeral-extra

**lemma** Let-numeral [simp]: Let (numeral v) f = f (numeral v)
— Unfold all lets involving constants
**unfolding** Let-def ..

**lemma** Let-neg-numeral [simp]: Let (− numeral v) f = f (− numeral v)
— Unfold all lets involving constants
**unfolding** Let-def ..

**declaration**

```
let fun number-of ctxt T n =
  if not (Sign.of-sort (Proof-Context.theory-of ctxt) (T, sort ⟨numeral⟩))
  then raise CTERM (number-of, [])
  else Numeral.mk-cnumber (Thm.ctyp-of ctxt T) n;
```

```
in K (Lin-Arith.set-number-of number-of
#> (Lin-Arith.add-simps
@{thms arith-simps more-arith-simps rel-simps pred-numeral-simps
  arith-special numeral-One of-nat-simps uminus-numeral-One
  Suc-numeral Let-numeral Let-neg-numeral Let-0 Let-1
  le-Suc-numeral le-numeral-Suc less-Suc-numeral less-numeral-Suc
  Suc-eq-numeral eq-numeral-Suc mult-Suc mult-Suc-right-of-nat-numeral})
end)
```

### 40.7.2 Simplification of arithmetic when nested to the right

**lemma** add-numeral-left [simp]: numeral v + (numeral w + z) = (numeral(v + w) + z)
  by (simp-all add: add.assoc [symmetric])

**lemma** add-neg-numeral-left [simp]:
  numeral v + (− numeral w + y) = (sub v w + y)
  − numeral v + (numeral w + y) = (sub w v + y)
  − numeral v + (− numeral w + y) = (− numeral(v + w) + y)
  by (simp-all add: add.assoc [symmetric])
lemma mult-numeral-left-semiring-numeral:
numeral v * (numeral w * z) = (numeral(v * w) * z :: 'a::semiring-numeral)
by (simp add: mult.assoc [symmetric])

lemma mult-numeral-left-ring-1:
- numeral v * (numeral w * y) = (- numeral(v * w) * y :: 'a::ring-1)
numeral v * (- numeral w * y) = (- numeral(v * w) * y :: 'a::ring-1)
- numeral v * (- numeral w * y) = (numeral(v * w) * y :: 'a::ring-1)
by (simp-all add: mult.assoc [symmetric])

lemmas mult-numeral-left [simp] =
mult-numeral-left-semiring-numeral
mult-numeral-left-ring-1

hide-const (open) One Bit0 Bit1 BitM inc pow sqr sub dbl dbl-inc dbl-dec

40.8 Code module namespace

code-identifier
code-module Num ⇒ (SML) Arith and (OCaml) Arith and (Haskell) Arith

40.9 Printing of evaluated natural numbers as numerals

lemma [code-post]:
Suc 0 = 1
Suc 1 = 2
Suc (numeral n) = numeral (Num.inc n)
by (simp-all add: numeral-inc)

lemmas [code-post] = Num.inc.simps

end

41 Exponentiation

theory Power
  imports Num
begin

41.1 Powers for Arbitrary Monoids

class power = one + times
begin

primrec power :: 'a ⇒ nat ⇒ 'a (infixr "\^ 80")
where
  power-0: a ^ 0 = 1
| power-Suc: a ^ Suc n = a * a ^ n
**THEORY** “Power”  

**notation** (latex output)  
\( \text{power}\ ((^\cdot) [1000] 1000) \)

Special syntax for squares.

**abbreviation** power2 :: 'a ⇒ 'a 
\((^2) [1000] 999) \)

where \(x^2 \equiv x \cdot 2\)

**context** includes lifting-syntax  
begin

**lemma** power-transfer [transfer-rule]:
\((R ===> (=) ===> R) \quad (\cdot) \quad (\cdot)\)

if [transfer-rule]: \(\langle R \ 1 \ \rangle\)
\((R ===> R ===> R) \quad (\ast) \quad (\ast)\)

for \(R :: \langle a::\text{power} \Rightarrow b::\text{power} \Rightarrow \text{bool}\)
by (simp only: power-def [abs-def]) transfer-prover

end

**context** monoid-mult  
begin

subclass power .  

**lemma** power-one [simp]: \(1 \cdot n = 1\)
by (induct \(n\)) simp-all

**lemma** power-one-right [simp]: \(a \cdot 1 = a\)
by simp

**lemma** power-Suc0-right [simp]: \(a \cdot \text{Suc } 0 = a\)
by simp

**lemma** power-commutes: \(a \cdot n \ast a = a \ast a \cdot n\)
by (induct \(n\)) (simp-all add: mult.assoc)

**lemma** power-Suc2: \(a \cdot \text{Suc } n = a \cdot n \ast a\)
by (simp add: power-commutes)

**lemma** power-add: \(a \cdot (m + n) = a \cdot m \ast a \cdot n\)
by (induct \(m\)) (simp-all add: algebra-simps)

**lemma** power-mult: \(a \cdot (m \ast n) = (a \cdot m) \cdot n\)
by (induct \(n\)) (simp-all add: power-add)
lemma power-even-eq: \( a^{(2 \cdot n)} = (a^n)^2 \)
by (subst mult.commute) (simp add: power-mult)

lemma power-odd-eq: \( a^{Suc \cdot (2\cdot n)} = a \cdot (a^n)^2 \)
by (simp add: power-even-eq)

lemma power-numeral-even: \( z^{numeral \cdot (Num.Bit0 \cdot w)} = (let \ w = z^{numeral \cdot w} in w \cdot w) \)
by (simp only: numeral-Bit0 power-add Let-def)

lemma power-numeral-odd: \( z^{numeral \cdot (Num.Bit1 \cdot w)} = (let \ w = z^{numeral \cdot w} in z \cdot w \cdot w) \)
by (simp only: numeral-Bit1 One-nat-def add-Suc-right add-0-right power-Suc power-add Let-def mult.assoc)

lemma power2-eq-square: \( a^2 = a \cdot a \)
by (simp add: numeral-2-eq-2)

lemma power3-eq-cube: \( a^3 = a \cdot a \cdot a \)
by (simp add: numeral-3-eq-3 mult.assoc)

lemma power4-eq-xxxx: \( x^4 = x \cdot x \cdot x \cdot x \)
by (simp add: mult.assoc power-numeral-even)

lemma funpow-times-power: \( \times x^{f \cdot x} = \times (x^{f \cdot x}) \)
proof (induct f x arbitrary: f)
  case 0
  then show \( ?\text{case} \) by (simp add: fun-eq-iff)
next
  case (Suc n)
  define g where \( g \_x = f \_x - 1 \) for \( x \)
  with Suc have \( n = g \_x \) by simp
  with Suc have \( times x^{g \_x} = times (x \cdot g \_x) \) by simp
  moreover from Suc g-def have \( f \_x = g \_x + 1 \) by simp
  ultimately show \( ?\text{case} \)
  by (simp add: power-def funpow-add fun-eq-iff mult.assoc)
qed

lemma power-commuting-commutes:
  assumes \( x \cdot y = y \cdot x \)
  shows \( x^n \cdot y = y \cdot x^n \)
proof (induct n)
  case 0
  then show \( ?\text{case} \) by simp
next
  case (Suc n)
  have \( x^{Suc \cdot n} \cdot y = x^n \cdot y \cdot x \)
  by (subst power-Suc2) (simp add: assms ac-simps)
  also have \( \ldots = y \cdot x^{Suc \cdot n} \)
theory "Power"

by (simp only: Suc power-Suc2) (simp add: ac-simps)
finally show ?case.
qed

lemma power-minus-mult: 0 < n \implies a ^ (n - 1) * a = a ^ n
by (simp add: power-commutes split: nat-diff-split)

lemma left-right-inverse-power:
assumes x * y = 1
shows x ^ n * y ^ n = 1
proof (induct n)
case (Suc n)
moreover have x ^ Suc n * y ^ Suc n = x ^ n * (x * y) * y ^ n
by (simp add: power-Suc2[symmetric] mult.assoc[symmetric])
ultimately show ?case by (simp add: assms)
qed simp

end

context comm-monoid-mult
begin

lemma power-mult-distrib [algebra-simps, algebra-split-simps, field-simps, field-split-simps, divide-simps]:
(a * b) ^ n = (a ^ n) * (b ^ n)
by (induction n) (simp-all add: ac-simps)

end

Extract constant factors from powers.

declare power-mult-distrib [where a = numeral w for w, simp]
declare power-mult-distrib [where b = numeral w for w, simp]

lemma power-add-numeral [simp]: a ^ numeral m * a ^ numeral n = a ^ numeral (m + n)
for a :: 'a::monoid-mult
by (simp add: power-add [symmetric])

lemma power-add-numeral2 [simp]: a ^ numeral m * (a ^ numeral n * b) = a ^ numeral (m + n) * b
for a :: 'a::monoid-mult
by (simp add: mult.assoc [symmetric])

lemma power-mult-numeral [simp]: (a ^ numeral m) ^ numeral n = a ^ numeral (m * n)
for a :: 'a::monoid-mult
by (simp only: numeral-mult power-mult)

context semiring-numeral
begin

lemma numeral-sqr: numeral (Num.sqr k) = numeral k * numeral k
by (simp only: sqr-conv-mult numeral-mult)

lemma numeral-pow: numeral (Num.pow k l) = numeral k ^ numeral l
by (induct l)
(simp-all only: numeral-class.numeral.simps pow.simps
 numeral-sqr numeral-mult power-add power-one-right)

lemma power-numeral [simp]: numeral k ^ numeral l = numeral (Num.pow k l)
by (rule numeral-pow [symmetric])

end

context semiring-1
begin

lemma of-nat-power [simp]: of-nat (m ^ n) = of-nat m ^ n
by (induct n) simp-all

lemma zero-power: 0 < n ==> 0 ^ n = 0
by (cases n) simp-all

lemma power-zero-numeral [simp]: 0 ^ numeral k = 0
by (simp add: numeral-eq-Suc)

lemma zero-power2: 0^2 = 0
by (rule power-zero-numeral)

lemma one-power2: 1^2 = 1
by (rule power-one)

lemma power-0-Suc [simp]: 0 ^ Suc n = 0
by simp

It looks plausible as a simprule, but its effect can be strange.

lemma power-0-left: 0 ^ n = (if n = 0 then 1 else 0)
by (cases n) simp-all

end

context semiring-char-0 begin

lemma numeral-power-eq-of-nat-cancel-iff [simp]:
umeral x ^ n = of-nat y <-> numeral x ^ n = y
using of-nat-eq-iff by fastforce

lemma real-of-nat-eq-numeral-power-cancel-iff [simp]:
THEORY “Power”

of-naty \equiv y = \text{numeral } x \cdot n

using numeral-power-eq-of-nat-cancel-iff [of \ x \ n \ y] by (metis (mono-tags))

lemma of-nat-eq-of-nat-power-cancel-iff[simp]: (of-nat b) \cdot w = of-nat x \iff b \cdot w = x
  by (metis of-nat-power of-nat-eq-iff)

lemma of-nat-power-eq-of-nat-cancel-iff[simp]: of-nat x = (of-nat b) \cdot w \iff x = b \cdot w
  by (metis of-nat-eq-of-nat-power-cancel-iff)

end

context comm-semiring-1
begin

The divides relation.

lemma le-imp-power-dvd:
  assumes m \leq n
  shows a \cdot m \ dvd a \cdot n
proof
  from assms have a \cdot n = a \cdot (m + (n - m)) by simp
  also have \ldots = a \cdot m \cdot a \cdot (n - m) by (rule power-add)
  finally show a \cdot n = a \cdot m \cdot a \cdot (n - m).
qed

lemma power-le-dvd: a \cdot n \ dvd b \implies m \leq n \implies a \cdot m \ dvd b
  by (rule dvd-trans [OF le-imp-power-dvd])

lemma dvd-power-same: x \ dvd y \implies x \cdot n \ dvd y \cdot n
  by (induct n) (auto simp add: mult-dvd-mono)

lemma dvd-power-le: x \ dvd y \implies m \geq n \implies x \cdot n \ dvd y \cdot m
  by (rule power-le-dvd [OF dvd-power-same])

lemma dvd-power [simp]:
  fixes n :: nat
  assumes n > 0 \lor x = 1
  shows x \ dvd (x \cdot n)
  using assms
proof
  assume 0 < n
  then have x \cdot n = x \cdot Suc (n - 1) by simp
  then show x \ dvd (x \cdot n) by simp
next
  assume x = 1
  then show x \ dvd (x \cdot n) by simp
qed
begin

context semiring-1-no-zero-divisors
begin

subclass power.

lemma power-eq-0-iff [simp]:\(a \cdot n = 0 \iff a = 0 \land n > 0\)
by (induct n) auto

lemma power-not-zero: \(a \neq 0 \implies a \cdot n \neq 0\)
by (induct n) auto

lemma zero-eq-power2 [simp]: \(a^2 = 0 \iff a = 0\)
unfolding power2-eq-square by simp

end

context ring-1
begin

lemma power-minus: \((-a) \cdot n = (-1) \cdot n \cdot a \cdot n\)
proof (induct n)
  case 0
  show ?case by simp
next
  case (Suc n)
  then show ?case
  by (simp del: power-Suc add: power-Suc2 mult.assoc)
qed

lemma power-minus': NO-MATCH 1 x \implies (-x) \cdot n = (-1) \cdot n \cdot x \cdot n
by (rule power-minus)

lemma power-minus-Bit0: \((-x) \cdot \text{numeral (Num.Bit0 \(k\))} = \text{numeral (Num.Bit0 \(k\))}\)
by (induct k, simp-all only: numeral-class.numeral.simps power-add
  power-one-right mult-minus-left mult-minus-right minus-minus)

lemma power-minus-Bit1: \((-x) \cdot \text{numeral (Num.Bit1 \(k\))} = - (x \cdot \text{numeral (Num.Bit1 \(k\)))}\)
by (simp only: eval-nat-numeral(3) power-Suc power-minus-Bit0 mult-minus-left)

lemma power2-minus [simp]: \((-a)^2 = a^2\)
by (fact power-minus-Bit0)

lemma power-minus1-even [simp]: \((-1) \cdot (2n) = 1\)
proof (induct n)
  case 0
show ?case by simp
next
case (Suc n)
then show ?case by (simp add: power-add power2-eq-square)
qed

lemma power-minus1-odd: \((-1)^{\text{Suc} (2 \cdot n)} = -1\)
by simp

lemma power-minus-even [simp]: \((-a)^{2 \cdot n} = a^{2 \cdot n}\)
by (simp add: power-minus [of a])
end

context ring-1-no-zero-divisors
begin
lemma power2-eq-1-iff: 
a^2 = 1 \iff a = 1 \lor a = -1
using square-eq-1-iff [of a] by (simp add: power2-eq-square)
end

context idom
begin
lemma power2-eq-iff: 
x^2 = y^2 \iff x = y \lor x = -y
unfolding power2-eq-square by (rule square-eq-iff)
end

context semidom-divide
begin
lemma power-diff:
\(a^{m-n} = (a^m) \div (a^n)\) if \(a \neq 0\) and \(n \leq m\)
proof -
define q where q = m - n
with \(n \leq m\) have m = q + n by simp
with \(a \neq 0\): q-def show ?thesis
  by (simp add: power-add)
qed
end

context algebraic-semidom
begin
lemma div-power: \(b \dvd a \implies (a \div b)^n = a^n \div b^n\)
by (induct n) (simp-all add: div-mult-if-dvd dvd-power-same)
lemma is-unit-power-iff: is-unit \( (a \cdot n) \rightarrow is-unit a \lor n = 0 \)
    by (induct n) (auto simp add: is-unit-mult-iff)

lemma dvd-power-iff:
    assumes \( x \neq 0 \)
    shows \( x \cdot m \text{ dvd } x \cdot n \leftrightarrow is-unit x \lor m \leq n \)
    proof
      assume \(\ast\): \( x \cdot m \text{ dvd } x \cdot n \)
      { 
        assume \( m > n \)
        note \(\ast\) also have \( x \cdot n = x \cdot n \cdot 1 \) by simp
        also from \( m > n \) have \( m = n + (m - n) \) by simp
        also have \( x \cdot \ldots = x \cdot n \cdot x \cdot (m - n) \) by (rule power-add)
        finally have \( x \cdot (m - n) \text{ dvd } 1 \)
          by (subst (asm) dvd-times-left-cancel-iff) (insert assms, simp-all)
        with \( m > n \) have \( is-unit x \) by (simp add: is-unit-power-iff)
      }
      thus \( is-unit x \lor m \leq n \) by force
    qed (auto intro: unit-imp-dvd simp: is-unit-power-iff le-imp-power-dvd)

end

context normalization-semidom-multiplicative
begin

lemma normalize-power: normalize \( (a \cdot n) \) = normalize \( a \cdot n \)
    by (induct n) (simp-all add: normalize-mult)

lemma unit-factor-power: unit-factor \( (a \cdot n) \) = unit-factor \( a \cdot n \)
    by (induct n) (simp-all add: unit-factor-mult)

end

context division-ring
begin

Perhaps these should be simprules.

lemma power-inverse [field-simps, field-split-simps, divide-simps]: inverse \( a \cdot n \) = inverse \( a \cdot n \)
    proof (cases \( a = 0 \))
      case True
      then show \(?thesis\) by (simp add: power-0-left)
    next
      case False
      then have \( inverse (a \cdot n) = inverse a \cdot n \)
          by (induct n) (simp-all add: nonzero-inverse-mult-distrib power-commutes)

end
then show \( \text{thesis by simp} \)

qed

lemma power-one-over [field-simps, field-split-simps, divide-simps]: \((1 / a) ^ n = 1 / a ^ n\)
using power-inverse [of a] by (simp add: divide-inverse)

end

context field
begin

lemma power-divide [field-simps, field-split-simps, divide-simps]: \((a / b) ^ n = a ^ n / b ^ n\)
by (induct n) simp-all

end

41.2 Exponentiation on ordered types

context linordered-semidom
begin

lemma zero-less-power [simp]: \(0 < a \implies 0 < a ^ n\)
by (induct n) simp-all

lemma zero-le-power [simp]: \(0 \leq a \implies 0 \leq a ^ n\)
by (induct n) simp-all

lemma power-mono: \(a \leq b \implies 0 \leq a \implies a ^ n \leq b ^ n\)
by (induct n) (auto intro: mult-mono order-trans [of 0 a b])

lemma one-le-power [simp]: \(1 \leq a \implies 1 \leq a ^ n\)
using power-mono [of 1 a n] by simp

lemma power-le-one: \(0 \leq a \implies a \leq 1 \implies a ^ n \leq 1\)
using power-mono [of a 1 n] by simp

lemma power-gt1-lemma:
assumes gt1: \(1 < a\)
shows \(1 < a * a ^ n\)
proof –
from gt1 have \(0 < a\)
  by (fact order-trans [OF zero-le-one less-imp-le])
from gt1 have \(1 * 1 < a * 1\) by simp
also from gt1 have \(\ldots \leq a * a ^ n\)
  by (simp only: mult-mono [of \(1 \leq a\) one-le-power order-less-imp-le zero-le-one order-refl])
finally show \(\text{thesis by simp}\)
qed

lemma power-gt1: \(1 < a \implies 1 < a \cdot \text{Suc } n\)
  by (simp add: power-gt1-lemma)

lemma one-less-power [simp]: \(1 < a \implies 0 < n \implies 1 < a \cdot n\)
  by (cases n) (simp-all add: power-gt1-lemma)

lemma power-le-imp-le-exp:
  assumes gt1: \(1 < a\)
  shows \(\text{a}^m \leq \text{a}^n \implies m \leq n\)
proof (induct m arbitrary: n)
  case 0
  show ?case by simp
  next
  case (Suc m)
  show ?case
proof (cases n)
  case 0
  with Suc have \(a \ast a^m \leq 1\) by simp
  with gt1 show ?thesis
    by (force simp only: power-gt1-lemma not-less [symmetric])
  next
  case (Suc n)
  with Suc.prems Suc.hyps show ?thesis
    by (force dest: mult-left-le-imp-le simp add: less-trans [OF zero-less-one gt1])
qed
qed

lemma of-nat-zero-less-power-iff [simp]: \(\text{of-nat } x \cdot n > 0 \iff x > 0 \lor n = 0\)
  by (induct n) auto

Surely we can strengthen this? It holds for \(0 < a < 1\) too.

lemma power-inject-exp [simp]: \(1 < a \implies \text{a}^m = \text{a}^n \iff m = n\)
  by (force simp add: order-antisym power-le-imp-le-exp)

Can relax the first premise to \((0::\'a) < a\) in the case of the natural numbers.

lemma power-less-imp-less-exp: \(1 < a \implies \text{a}^m < \text{a}^n \implies m < n\)
  by (simp add: order-less-le [of m n] less-le [of a\^m a\^n] power-le-imp-le-exp)

lemma power-strict-mono [rule-format]: \(a < b \implies 0 \leq a \implies 0 < n \implies a \cdot n < b \cdot n\)
  by (induct n) (auto simp: mult-strict-mono le-less-trans [of 0 a b])

lemma power-mono-iff [simp]:
  shows \([a \geq 0; b \geq 0; n > 0]\) \(\implies a \cdot n \leq b \cdot n \iff a \leq b\)
  using power-mono [of a b] power-strict-mono [of b a] not-le by auto

Lemma for power-strict-decreasing
lemma power-Suc-less: $0 < a \Rightarrow a < 1 \Rightarrow a \cdot a ^ \sim n < a ^ \sim n$
  by (induct n) (auto simp: mult-strict-left-mono)

lemma power-strict-decreasing [rule-format]: $n < N \Rightarrow 0 < a \Rightarrow a < 1 \Rightarrow a ^ \sim N < a ^ \sim n$
  proof (induct N)
    case 0
    then show ?case by simp
  next
    case (Suc N)
    then show ?case
      apply (auto simp add: power-Suc-less less-Suc-eq)
      apply (subgoal-tac a * a ^ N < 1 * a ^ n)
      apply simp
      apply (rule mult-strict-mono)
      apply auto
      done
  qed

Proof resembles that of power-strict-decreasing.

lemma power-decreasing: $n \leq N \Rightarrow 0 \leq a \Rightarrow a \leq 1 \Rightarrow a ^ \sim N \leq a ^ \sim n$
  proof (induct N)
    case 0
    then show ?case by simp
  next
    case (Suc N)
    then show ?case
      apply (auto simp add: le-Suc-eq)
      apply (subgoal-tac a * a ^ N \leq 1 * a ^ n)
      apply simp
      apply (rule mult-mono)
      apply auto
      done
  qed

lemma power-decreasing-iff [simp]: $[0 < b; b < 1] \Rightarrow b ^ \sim m \leq b ^ \sim n \longleftrightarrow n \leq m$
  using power-strict-decreasing [of m n b]
  by (auto intro: power-decreasing ccontr)

lemma power-strict-decreasing-iff [simp]: $[0 < b; b < 1] \Rightarrow b ^ \sim m < b ^ \sim n \longleftrightarrow n < m$
  using power-decreasing-iff [of b m n] unfolding le-less
  by (auto dest: power-strict-decreasing le-neq-implies-less)

lemma power-Suc-less-one: $0 < a \Rightarrow a < 1 \Rightarrow a ^ \sim Suc n < 1$
  using power-strict-decreasing [of 0 Suc n a] by simp

Proof again resembles that of power-strict-decreasing.
lemma power-increasing: \( n \leq N \Rightarrow 1 \leq a \Rightarrow a \cdot n \leq a \cdot N \)
proof (induct N)
  case \( \emptyset \)
  then show ?case by simp
next
  case (Suc N)
  then show ?case
apply (auto simp add: le-Suc-eq)
apply (subgoal-tac \( 1 \cdot a \cdot n \leq a \cdot a \cdot N \))
simp
apply (rule mult-mono)
  apply (auto simp add: order-trans \[OF \zero_less_one\])
done
qed

Lemma for power-strict-increasing.

lemma power-less-power-Suc: \( 1 < a \Rightarrow a \cdot n < a \cdot a \cdot n \)
by (induct n) (auto simp: mult-strict-left_mono less-trans \[OF \zero_less_one\])

lemma power-strict-increasing: \( n < N \Rightarrow 1 < a \Rightarrow a \cdot n < a \cdot a \cdot N \)
proof (induct N)
  case \( \emptyset \)
  then show ?case by simp
next
  case (Suc N)
  then show ?case
apply (auto simp add: power-less-power-Suc less-Suc-eq)
apply (subgoal-tac \( 1 \cdot a \cdot n < a \cdot a \cdot N \))
simp
apply (rule mult-strict-mono)
apply (auto simp add: less-trans \[OF \zero_less_one\] less-imp-le)
done
qed

lemma power-increasing-iff [simp]: \( 1 < b \Rightarrow b \cdot x \leq b \cdot y \iff x \leq y \)
by (blast intro: power-le-imp-le-exp power-increasing less-imp-le)

lemma power-strict-increasing-iff [simp]: \( 1 < b \Rightarrow b \cdot x < b \cdot y \iff x < y \)
by (blast intro: power-less-imp-strict-exp power-strict-increasing)

lemma power-le-imp-le-base:
  assumes le: \( a \cdot \text{Suc} \cdot n \leq b \cdot \text{Suc} \cdot n \)
  and \( \emptyset \leq b \)
  shows \( a \leq b \)
proof (rule ccontr)
  assume \( \neg \text{thesis} \)
then have \( b < a \) by (simp only: linorder-not-le)
then have \( b \cdot \text{Suc} \cdot n < a \cdot \text{Suc} \cdot n \)
  by (simp only: assms(2) power-strict-mono)
with le show False
  by (simp add: linorder-not-less [symmetric])
qed

lemma power-less-imp-less-base:
  assumes less: a ^ n < b ^ n
  assumes nonneg: 0 ≤ b
  shows a < b
proof (rule contrapos-pp [OF less])
  assume ¬ thesis
  then have b ≤ a by (simp only: linorder-not-less)
  from this nonneg have b ^ n ≤ a ^ n by (rule power-mono)
  then show ¬ a ^ n < b ^ n by (simp only: linorder-not-less)
qed

lemma power-inject-base: a ^ Suc n = b ^ Suc n ≤⇒ 0 ≤ a ≤⇒ 0 ≤ b ≤⇒ a = b
by (blast intro: power-le-imp-le-base antisym eq-refl sym)

lemma power-eq-imp-eq-base: a ^ n = b ^ n ≤⇒ 0 ≤ a ≤⇒ 0 ≤ b ≤⇒ 0 < n ≤⇒ a = b
by (cases n) (simp-all del: power-Suc, rule power-inject-base)

lemma power-eq-iff-eq-base: 0 < n ≤⇒ 0 ≤ a ≤⇒ 0 ≤ b ≤⇒ a ^ n = b ^ n ←→ a = b
  using power-eq-imp-eq-base [of a n b] by auto

lemma power2-le-imp-le: x^2 ≤ y^2 ≤⇒ 0 ≤ y ≤⇒ x ≤ y
unfolding numeral-2-eq-2 by (rule power-le-imp-le-base)

lemma power2-less-imp-less: x^2 < y^2 ≤⇒ 0 ≤ y ≤⇒ x < y
by (rule power-less-imp-less-base)

lemma power2-eq-imp-eq: x^2 = y^2 ≤⇒ 0 ≤ x ≤⇒ 0 ≤ y ≤⇒ x = y
unfolding numeral-2-eq-2 by (erule (2) power-eq-imp-eq-base) simp

lemma power-Suc-le-self: 0 ≤ a ≤⇒ a ≤ 1 ≤⇒ a ^ Suc n ≤ a
using power-decreasing [of 1 Suc n a] by simp

lemma power2-eq-iff-nonneg [simp]:
  assumes 0 ≤ x 0 ≤ y
  shows (x ^ 2 = y ^ 2) ←→ x = y
using assms power2-eq-imp-eq by blast

lemma of-nat-less-numeral-power-cancel-iff [simp]:
of-nat x < numeral i ^ n ←→ x < numeral i ^ n
using of-nat-less-iff [of x numeral i ^ n, unfolded of-nat-numeral of-nat-power].

lemma of-nat-le-numeral-power-cancel-iff [simp]:
of-nat x ≤ numeral i ^ n ←→ x ≤ numeral i ^ n
using \text{of-nat-le-iff}[of x \text{ numeral } i \sim n, \text{unfolded of-nat-numeral of-nat-power}] .

\textbf{Lemma numeral-power-less-of-nat-cancel-iff}[simp]:
\text{numeral } i \sim n < \text{of-nat } x \leftrightarrow \text{numeral } i \sim n < x
\text{using \text{of-nat-less-iff}[of numeral } i \sim n x, \text{unfolded of-nat-numeral of-nat-power}] .

\textbf{Lemma numeral-power-le-of-nat-cancel-iff}[simp]:
\text{numeral } i \sim n \leq \text{of-nat } x \leftrightarrow \text{numeral } i \sim n \leq x
\text{using \text{of-nat-le-iff}[of numeral } i \sim n x, \text{unfolded of-nat-numeral of-nat-power}] .

\textbf{Lemma of-nat-le-of-nat-power-cancel-iff}[simp]: \( (\text{of-nat } b)^w \leq \text{of-nat } x \leftrightarrow b^w \leq x \)
\text{by (metis of-nat-le-iff of-nat-power)}

\textbf{Lemma of-nat-power-le-of-nat-cancel-iff}[simp]: \( \text{of-nat } x \leq (\text{of-nat } b)^w \leftrightarrow x \leq b^w \)
\text{by (metis of-nat-le-iff of-nat-power)}

\textbf{Lemma of-nat-less-of-nat-power-cancel-iff}[simp]: \( (\text{of-nat } b)^w < \text{of-nat } x \leftrightarrow b^w < x \)
\text{by (metis of-nat-less-iff of-nat-power)}

\textbf{Lemma of-nat-power-less-of-nat-cancel-iff}[simp]: \( \text{of-nat } x < (\text{of-nat } b)^w \leftrightarrow x < b^w \)
\text{by (metis of-nat-less-iff of-nat-power)}

end

Some \texttt{nat}-specific lemmas:

\textbf{Lemma mono-ge2-power-minus-self}:
\texttt{assumes } k \geq 2 \texttt{ shows mono (\lambda m. k \sim m − m)}
\texttt{unfolding mono-iff-le-Suc}
\texttt{proof}
\texttt{fix } n
\texttt{have } k^n < k^\text{Suc } n \texttt{ using power-strict-increasing-iff[of } k n \text{ Suc } n \texttt{]} \texttt{assms by linarith}
\texttt{thus } k^n − n \leq k^\text{Suc } n − \text{Suc } n \texttt{ by linarith}
\texttt{qed}

\textbf{Lemma self-le-ge2-pow[simp]}:
\texttt{assumes } k \geq 2 \texttt{ shows } m \leq k^m
\texttt{proof (induction } m\texttt{)}
\texttt{case 0 show ?case by simp}
\texttt{next}
\texttt{case } (\text{Suc } m)\texttt{ hence } \text{Suc } m \leq \text{Suc } (k^m) \texttt{ by simp}
\texttt{also have } \ldots \leq k^m + k^m \texttt{ using one-le-power[of } k \text{]} \texttt{assms by linarith}
\texttt{also have } \ldots \leq k \cdot k^m \texttt{ by (metis mult-2 mult-le-mono1[OF assms])}
\texttt{finally show ?case by simp}
qed

lemma diff-le-diff-pow [simp]:
  assumes k ≥ 2 shows m − n ≤ k ^ m − k ^ n
proof (cases n ≤ m)
  case True
  thus ?thesis
  using monoD [OF mono-ge2-power-minus-self [OF assms] True] self-le-ge2-pow [OF assms, of m]
  by (simp add: le-diff-conv le-diff-conv2)
qed auto

context linordered-ring-strict
begin

lemma sum-squares-eq-zero-iff: x * x + y * y = 0 ←→ x = 0 ∧ y = 0
by (simp add: add-nonneg-eq-0-iff)

lemma sum-squares-le-zero-iff: x * x + y * y ≤ 0 ←→ x = 0 ∧ y = 0
by (simp add: le-less not-sum-squares-lt-zero sum-squares-eq-zero-iff)

lemma sum-squares-gt-zero-iff: 0 < x * x + y * y ←→ x ≠ 0 ∨ y ≠ 0
by (simp add: not-le [symmetric] sum-squares-le-zero-iff)

end

category linordered-idom

begin

lemma zero-le-power2 [simp]: 0 ≤ a^2
by (simp add: power2-eq-square)

lemma zero-less-power2 [simp]: 0 < a^2 ←→ a ≠ 0
by (force simp add: power2-eq-square zero-less-mult-iff linorder-neq-iff)

lemma power2-less-0 [simp]: ~ a^2 < 0
by (force simp add: power2-eq-square mult-less-0-iff)

lemma power-abs: |a ^ n| = |a| ^ n — FIXME simp?
by (induct n) (simp-all add: abs-mult)

lemma power-sgn [simp]: sgn (a ^ n) = sgn a ^ n
by (induct n) (simp-all add: sgn-mult)

lemma abs-power-minus [simp]: |(- a) ^ n| = |a ^ n|
by (simp add: power-args)

lemma zero-less-power-abs-iff [simp]: 0 < |a| ^ n ←→ a ≠ 0 ∨ n = 0
proof (induct n)
  case 0
  show ?case by simp
next
  case Suc
  then show ?case by (auto simp: zero-less-mult-iff)
qed

lemma zero-le-power-abs [simp]: \(0 \leq |a|^n\)
  by (rule zero-le-power [OF abs-ge-zero])

lemma power2-less-eq-zero-iff [simp]: \(a^2 \leq 0 \iff a = 0\)
  by (simp add: le-less)

lemma abs-power2 [simp]: \(|a|^2 = a^2\)
  by (simp add: power2-eq-square)

lemma power2-abs [simp]: \(|a|^2 = a^2\)
  by (simp add: power2-eq-square)

lemma odd-power-less-zero: \(a < 0 \implies a \cdot \text{Suc} (2 \cdot n) < 0\)
proof (induct n)
  case 0
  show ?case by simp
next
  case (Suc n)
  have \(a \cdot \text{Suc} (2 \cdot \text{Suc} n) = (a \cdot a) \cdot \text{Suc}(2 \cdot n)\)
    by (simp add: ac-simps power-add power2-eq-square)
  then show ?case
    by (simp del: power-Suc add: Suc mult-less-0-iff mult-neg-neg)
qed

lemma odd-0-le-power-imp-0-le: \(0 \leq a \cdot \text{Suc} (2 \cdot n) \implies 0 \leq a\)
  using odd-power-less-zero [of a n]
  by (force simp add: linorder-not-less [symmetric])

lemma zero-le-even-power [simp]: \(0 \leq a \cdot (2 \cdot n)\)
proof (induct n)
  case 0
  show ?case by simp
next
  case (Suc n)
  have \(a \cdot (2 \cdot \text{Suc} n) = (a \cdot a) \cdot a \cdot (2 \cdot n)\)
    by (simp add: ac-simps power-add power2-eq-square)
  then show ?case
    by (simp add: Suc zero-le-mult-iff)
qed

lemma sum-power2-ge-zero: \(0 \leq x^2 + y^2\)
by (intro add-nonneg-nonneg zero-le-power2)

lemma not-sum-power2-lt-zero: \( \neg x^2 + y^2 < 0 \)
  unfolding not-less by (rule sum-power2-ge-zero)

lemma sum-power2-eq-zero-iff: \( x^2 + y^2 = 0 \iff x = 0 \land y = 0 \)
  unfolding power2-eq-square by (simp add: add-nonneg-eq-0-iff)

lemma sum-power2-le-zero-iff: \( x^2 + y^2 \leq 0 \iff x = 0 \land y = 0 \)
  by (simp add: le-less sum-power2-eq-zero-iff not-sum-power2-lt-zero)

lemma sum-power2-gt-zero-iff: \( 0 < x^2 + y^2 \iff x \neq 0 \lor y \neq 0 \)
  unfolding not-le [symmetric] by (simp add: sum-power2-le-zero-iff)

lemma abs-square-le-1: \( |x|^2 \leq 1 \iff |x| \leq 1 \)
  using abs-square-eq-1 \( |x|^2 = 1 \iff |x| = 1 \)
  by (auto simp add: abs-if power2-eq-1-iff)

lemma power2-eq-square: \( a^2 = 1 \iff a = 1 \lor a = -1 \)
  using power-increasing [of 1 1 a] by auto

end

41.3 Miscellaneous rules

lemma (in linordered-semidom) self-le-power: \( 1 \leq a \implies 0 < n \implies a \leq a^{-n} \)
  using power-increasing [of 1 n a] power-one-right [of a] by auto
lemma (in power) power-eq-if: \( p^m = (\text{if } m = 0 \text{ then } 1 \text{ else } p \cdot (m - 1)) \)
unfolding One-nat-def by (cases m) simp-all

lemma (in comm-semiring-1) power2-sum: \((x + y)^2 = x^2 + y^2 + 2 \cdot x \cdot y\)
by (simp add: algebra-simps power2-eq-square mult-2-right)

context comm-ring-1
begin

lemma power2-diff: \((x - y)^2 = x^2 + y^2 - 2 \cdot x \cdot y\)
by (simp add: algebra-simps power2-eq-square mult-2-right)

lemma power2-commute: \((x - y)^2 = (y - x)^2\)
by (simp add: algebra-simps power2-eq-square)

lemma minus-power-mult-self: \((- a)^n \cdot (- a)^n = a^{2 \cdot n}\)
by (simp add: power-mult-distrib [symmetric])

lemma minus-one-mult-self [simp]: \((- 1)^n \cdot (- 1)^n = 1\)
using minus-power-mult-self [of 1 n] by simp

lemma left-minus-one-mult-self [simp]: \((- 1)^n \cdot ((- 1)^n \cdot a) = a\)
by (simp add: mult.assoc [symmetric])

end

Simplrules for comparisons where common factors can be cancelled.

lemmas zero-compare-simps =
add-strict-increasing add-strict-increasing2 add-increasing
zero-le-mult-iff zero-le-divide-iff
zero-less-mult-iff zero-less-divide-iff
mult-le-0-iff divide-le-0-iff
mult-less-0-iff divide-less-0-iff
zero-le-power2 power2-less-0

41.4  Exponentiation for the Natural Numbers

lemma nat-one-le-power [simp]: \(\text{Suc 0} \leq i \Rightarrow \text{Suc 0} \leq i \cdot n\)
by (rule one-le-power [of i n, unfolded One-nat-def])

lemma nat-zero-less-power-iff [simp]: \(x \cdot n > 0 \iff x > 0 \vee n = 0\)
for \(x :: \text{nat}\)
by (induct n) auto

lemma nat-power-eq-Suc-0-iff [simp]: \(x \cdot m = \text{Suc 0} \iff m = 0 \vee x = \text{Suc 0}\)
by (induct m) auto

lemma power-Suc-0 [simp]: \(\text{Suc 0} \cdot n = \text{Suc 0}\)
Valid for the naturals, but what if $0 < i < 1$? Premises cannot be weakened: consider the case where $i = 0$, $m = 1$ and $n = 0$. 

**Lemma nat-power-less-imp-less:**
- **Fixes** $i :: \text{naturals}$
- **Assumes** nonneg: $0 < i$
- **Assumes** less: $i ^ m < i ^ n$
- **Shows** $m < n$

**Proof (cases $i = 1$)**
- **Case** True
  - with less power-one [where $'a = \text{nat}$] show thesis by simp
- **Next**
  - **Case** False
    - with nonneg have $1 < i$ by auto
    - from power-strict-increasing-iff [OF this] less show thesis ..

**Qed**

**Lemma power-gt-expt:** $n > \text{Suc } 0 \Rightarrow n ^ k > k$
- by (induction $k$) (auto simp: less-trans-Suc n-less-m-mult-n)

**Lemma power-dvd-imp-le:**
- **Fixes** $i :: \text{naturals}$
- **Assumes** $i ^ m \text{ dvd } i ^ n I < i$
- **Shows** $m \leq n$

**Using assms** by (auto intro: power-le-imp-le-exp [OF $\langle 1 < i \rangle$ dvd-imp-le])

**Lemma dvd-power-iff-le:**
- **Fixes** $k :: \text{naturals}$
- **Shows** $2 \leq k \Rightarrow ((k ^ m) \text{ dvd } (k ^ n) \longleftrightarrow m \leq n)$

**Using** le-imp-power-dvd power-dvd-imp-le by force

**Lemma power2-nat-le-eq-le:** $m ^ 2 \leq n ^ 2 \longleftrightarrow m \leq n$
- for $m n :: \text{naturals}$
- by (auto intro: power2-le-imp-le power-mono)

**Lemma power2-nat-le-imp-le:**
- **Fixes** $m n :: \text{naturals}$
- **Assumes** $m ^ 2 \leq n$
- **Shows** $m \leq n$

**Proof (cases $m$)**
- **Case** 0
  - then show thesis by simp
- **Next**
  - **Case** (Suc $k$)
    - show thesis
    - **Proof** (rule ccontr)
      - assume $\neg$ thesis
      - then have $n < m$ by simp
with assms Suc show False
  by (simp add: power2-eq-square)
qed

lemma ex-power-ivl1: fixes b k :: nat assumes b ≥ 2
shows k ≥ 1 =⇒ ∃ n. b^n ≤ k ∧ k < b^(n+1) (is - =⇒ ∃ n. ?P k n)
proof (induction k)
  case 0 thus ?case by simp
next
  case (Suc k)
  show ?case
  proof
    cases k = b^(n+1) - 1
    case True
    hence ?P (Suc k) (n+1) using assms by simp
    thus ?case ..
  next
    assume k≠0
    with Suc obtain n where IH: ?P k n by auto
    show ?case
    proof (cases k = b^(n+1) - 1)
      case True
      hence ?P (Suc k) (n+1) using assms
        by (simp add: power-less-power-Suc)
      thus ?thesis ..
    next
      case False
      hence ?P (Suc k) n using IH by auto
      thus ?thesis ..
    qed
  qed
qed

lemma ex-power-ivl2: fixes b k :: nat assumes b ≥ 2 k ≥ 2
shows ∃ n. b^n < k ∧ k ≤ b^(n+1)
proof
  have 1 ≤ k - 1 using assms(2) by arith
  from ex-power-ivl1[OF assms(1) this]
  obtain n where b^n ≤ k - 1 ∧ k - 1 < b^(n+1) ..
  hence b^n < k ∧ k ≤ b^(n+1) using assms by auto
  thus ?thesis ..
qed

41.4.1 Cardinality of the Powerset

lemma card-UNIV-bool [simp]: card (UNIV :: bool set) = 2
  unfolding UNIV-bool by simp

lemma card-Pow: finite A =⇒ card (Pow A) = 2 ^ card A
proof (induct rule: finite-induct)
  case empty
  show ?case by simp
next
  case (insert x A)
  from ⟨x ∉ A⟩ have disjoint: Pow A ∩ insert x ' Pow A = {} by blast
  from ⟨x ∉ A⟩ have inj-on: inj-on (insert x) (Pow A)
    unfolding inj-on-def by auto

  have card (Pow (insert x A)) = card (Pow A ∪ insert x ' Pow A)
    by (simp only: Pow-insert)
  also have ... = card (Pow A) + card (insert x ' Pow A)
    by (rule card-Un-disjoint) (use ⟨finite A⟩ disjoint in simp-all)
  also from inj-on have card (insert x ' Pow A) = card (Pow A)
    by (rule card-image)
  also have ... + ... = 2 * ... by (simp add: mult-2)
  also from insert(3) have ... = 2 ^ Suc (card A) by simp
  also from insert(1,2) have Suc (card A) = card (insert x A)
    by (rule card-insert-disjoint [symmetric])
  finally show ?case .
qed

41.5 Code generator tweak

code-identifier
code-module Power → (SML) Arith and (OCaml) Arith and (Haskell) Arith

end

42 Big sum and product over finite (non-empty) sets

theory Groups-Big
  imports Power
begin

42.1 Generic monoid operation over a set

locale comm-monoid-set = comm-monoid
begin

42.1.1 Standard sum or product indexed by a finite set

interpretation comp-fun-commute f
  by standard (simp add: fun-eq-iff left-commute)

interpretation comp?: comp-fun-commute f ∘ g
  by (fact comp-comp-fun-commute)
definition $F :: (\{b \Rightarrow 'a\} \Rightarrow \{b \Rightarrow \} \Rightarrow \) \Rightarrow \{b \Rightarrow \} \Rightarrow \{a \Rightarrow \}$

where \textit{eq-fold}: $F g A = \text{Finite-Set.fold} (f \circ g) \ 1 \ A$

\textit{lemma infinite [simp]}; \neg \text{finite} A \Longrightarrow F g A = \ 1$
by (simp add: eq-fold)

\textit{lemma empty [simp]}; $F g \ \{\} = \ 1$
by (simp add: eq-fold)

\textit{lemma insert [simp]}; \text{finite} A \Longrightarrow x \notin A \Longrightarrow F g (\text{insert} x A) = g x \cdot F g A$
by (simp add: eq-fold)

\textit{lemma remove};
\text{assumes finite} A \ \text{and} \ x \in A$
\text{shows} $F g A = g x \cdot F g (A - \{x\})$
\text{proof} –
from $x \in A$; obtain $B$ where $B = \text{insert} x B$ \text{and} $x \notin B$
by (auto dest: \text{mk-disjoint-insert})
moreover from \text{finite} A \ \text{B have} \ \text{finite} B \ \text{by simp}$
ultimately show \text{thesis} by simp
qed

\textit{lemma insert-remove}; \text{finite} A \Longrightarrow F g (\text{insert} x A) = g x \cdot F g (A - \{x\})$
by (cases $x \in A$) (simp-all add: remove-insert-absorb)

\textit{lemma insert-if}; \text{finite} A \Longrightarrow F g (\text{insert} x A) = (if x \in A \ \text{then} F g A \ \text{else} g x \cdot F g A)$
by (cases $x \in A$) (simp-all add: insert-absorb)

\textit{lemma neutral}; \forall x \in A. g x = \ 1 \Longrightarrow F g A = \ 1$
by (induct A rule: \text{infinite-finite-induct}) simp-all

\textit{lemma neutral-const [simp]}; $F (\lambda -. \ 1) A = \ 1$
by (simp add: neutral)

\textit{lemma union-inter};
\text{assumes finite} A \ \text{and} \ finite B$
\text{shows} $F g (A \cup B) \cdot F g (A \cap B) = F g A \cdot F g B$
— The reversed orientation looks more natural, but LOOPS as a simprule!
using \text{assms}$
\text{proof} (induct A)$
case empty
then show \text{case} by simp
next
case (\text{insert} x A)
then show \text{case}$
by (auto simp: insert-absorb Int-insert-left commute [of - g x] assoc left-commute)
qed
corollary union-inter-neutral:
  assumes finite A and finite B
  and \( \forall x \in A \cap B, g(x) = 1 \)
  shows \( F g (A \cup B) = F g A \ast F g B \)
  using assms by (simp add: union-inter [symmetric] neutral)

corollary union-disjoint:
  assumes finite A and finite B
  assumes \( A \cap B = \{\} \)
  shows \( F g (A \cup B) = F g A \ast F g B \)
  using assms by (simp add: union-inter-neutral)

lemma union-diff2:
  assumes finite A and finite B
  shows \( F g (A \cup B) = F g (A - B) \ast F g (B - A) \ast F g (A \cap B) \)
  proof
    have \( A \cup B = A - B \cup (B - A) \cup A \cap B \)
      by auto
    with assms show ?thesis
      by simp (subst union-disjoint, auto)
  qed

lemma subset-diff:
  assumes \( B \subseteq A \) and finite A
  shows \( F g A = F g (A - B) \ast F g B \)
  proof
    from assms have finite \((A - B)\) by auto
    moreover from assms have finite \(B\) by (rule finite-subset)
    moreover from assms have \((A - B) \cap B = \{\}\) by auto
    ultimately have \( F g (A - B \cup B) = F g (A - B) \ast F g B \) by (rule union-disjoint)
    moreover from assms have \( A \cup B = A \) by auto
    ultimately show ?thesis by simp
  qed

lemma Int-Diff:
  assumes finite A
  shows \( F g A = F g (A \cap B) \ast F g (A - B) \)
  by (subst subset-diff [where \( B = A - B \)]) (auto simp: Diff-Diff-Int assms)

lemma setdiff-irrelevant:
  assumes finite A
  shows \( F g (A - \{x. g x = z\}) = F g A \)
  using assms by (induct A) (simp-all add: insert-Diff-if)

lemma not-neutral-contains-not-neutral:
  assumes \( F g A \neq 1 \)
  obtains \( a \) where \( a \in A \) and \( g a \neq 1 \)
  proof
from assms have \( \exists a \in A. \ g \ a \neq 1 \)
proof (induct A rule: infinite-finite-induct)
  case infinite
  then show \(?case\) by simp
next
  case empty
  then show \(?case\) by simp
next
  case (insert a A)
  then show \(?case\) by fastforce
qed

with that show thesis by blast
qed

lemma reindex:
  assumes inj-on \( h \) A
  shows \( F \ g \ (h \cdot A) = F \ (g \circ h) \ A \)
proof (cases finite A)
  case True
  with assms show \(?thesis\)
  by (simp add: eq-fold fold-image comp-assoc)
next
  case False
  with assms have \( \neg\ )finite \( (h \cdot A) \) by (blast dest: finite-imageD)
  with False show \(?thesis\) by simp
qed

lemma cong [fundef-cong]:
  assumes \( A = B \)
  assumes g-h: \( \forall x. \ x \in B \Longrightarrow g \ x = h \ x \)
  shows \( F \ g \ A = F \ (g \circ h) \ B \)
  using g-h unfolding \( \langle A = B \rangle \)
  by (induct B rule: infinite-finite-induct) auto
lemma cong-simp [cong]:
  \[ \begin{array}{l}
  \forall A = B; \ \forall x. \ x \in B =simp=> g \ x = h \ x \ \Longrightarrow F (\lambda x. \ g \ x) A = F (\lambda x. \ h \ x) B
  \end{array} \]
  by (rule cong) (simp-all add: simp-implies-def)

lemma reindex-cong:
  assumes inj-on \( l \) B
  assumes \( A = l \cdot B \)
  assumes \( \forall x. \ x \in B \Longrightarrow g \ (l \ x) = h \ x \)
  shows \( F \ g \ A = F \ h \ B \)
  using assms by (simp add: reindex)

lemma UNION-disjoint:
  assumes finite \( I \) and \( \forall i \in I. \ )finite \( \langle A i \rangle \)
  \( \)and \( \forall i \in I. \ \forall j \in I. \ i \neq j \Longrightarrow A i \cap A j = {} \)
  shows \( F \ g \ (\bigcup (A \cdot I)) = F \ (\lambda x. \ g \ (A x)) \ I \)
using assms
proof (induction rule: finite-induct)
  case (insert i I)
  then have \( \forall j \in I. \, j \neq i \)  
    by blast
  with insert.prems have \( A \cap \bigcup (A \setminus I) = {} \)  
    by blast
  with insert show \( \text{?case} \)  
    by (simp add: union-disjoint)
qed auto

lemma Union-disjoint:
  assumes \( \forall A \in C. \, \text{finite } A \).
  \( \forall A \in C. \forall B \in C. \, A \neq B \rightarrow A \cap B = {} \)
proof (cases finite C)
  case True
  from UNION-disjoint [OF this assms] show \( \text{?thesis} \) by simp
next
  case False
  then show \( \text{?thesis} \) by (auto dest: finite-UnionD intro: infinite)
qed

lemma distrib: \( F (\lambda x. \, g x \ast h x) A = F g A \ast F h A \)
by (induct A rule: infinite-finite-induct) (simp-all add: assoc commute left-commute)

lemma Sigma:
  assumes finite A \( \forall x \in A. \, \text{finite } (B x) \)
  shows \( F (\lambda x. \, F (g x) (B x)) A = F (\text{case-prod } g) (\text{SIGMA } x A. \, B x) \)
proof (subst UNION-disjoint)
  show \( F (\lambda x. \, F (g x) (B x)) A = F (\lambda x. \, F (\lambda(x, y). \, g x y) (\bigcup y \in B x. \, \{(x, y)\})) \)
  A
  proof (rule cong [OF refl])
    show \( F (g x) (B x) = F (\lambda(x, y). \, g x y) (\bigcup y \in B x. \, \{(x, y)\}) \)
      if \( x \in A \) for \( x \)
    using that assms by (simp add: UNION-disjoint)
  qed
qed (use assms in auto)

lemma related:
  assumes Re: \( R 1 1 \)
  and Rop: \( \forall x1 y1 x2 y2. \, R x1 x2 \land R y1 y2 \rightarrow R (x1 \ast y1) (x2 \ast y2) \)
  and fin: \( \text{finite } S \)
  and R-h-g: \( \forall x \in S. \, R (h x) (g x) \)
  shows \( R (F h S) (F g S) \)
  using fin by (rule finite-subset-induct) (use assms in auto)

lemma mono-neutral-cong-left:
  assumes finite T
and $S \subseteq T$
and $\forall i \in T - S. \ h i = 1$
and $\land x. \ x \in S \implies \ h x = h x$
shows $F g S = F h T$

proof
- have eq: $T = S \cup (T - S)$ using ($S \subseteq T$) by blast
- have d: $S \cap (T - S) = \{}$ using ($S \subseteq T$) by blast
- from $\langle$finite $T \rangle$ $\langle S \subseteq T \rangle$ have $f$: finite $S$ finite $(T - S)$
  by (auto intro: finite-subset)
- show $\langle$thesis using assms(4)$\rangle$
  by (simp add: union-disjoint [OF d f], unfolded eq [symmetric])
qed

lemma mono-neutral-cong-right:
finite $T \implies S \subseteq T \implies \forall i \in T - S. \ g i = 1 \implies (\land x. \ x \in S \implies \ h x = h x)$
implies $F g T = F h S$
by (auto intro!: mono-neutral-cong-left [symmetric])

lemma mono-neutral-left: finite $T \implies S \subseteq T \implies \forall i \in T - S. \ g i = 1 \implies F g S = F g T$
by (blast intro: mono-neutral-cong-left)

lemma mono-neutral-right: finite $T \implies S \subseteq T \implies \forall i \in T - S. \ g i = 1 \implies F g S = F g T$
by (blast intro!: mono-neutral-cong-left [symmetric])

lemma mono-neutral-cong:
assumes [simp]: finite $T$ finite $S$
and $*: \land i. \ i \in T - S \implies \ h i = 1 \land i. \ i \in S - T \implies \ g i = 1$
and gh: $\land x. \ x \in S \cap T \implies \ h x = h x$
shows $F g S = F h T$

proof
- have $F g S = F g (S \cap T)$
  by (rule mono-neutral-right)(auto intro: *)
- also have $\ldots = F h (S \cap T)$ using refl gh by (rule cong)
- also have $\ldots = F h T$
  by (rule mono-neutral-left)(auto intro: *)
- finally show $\langle$thesis $\rangle$.
qed

lemma reindex-bij-betw: bij-betw $h S T \implies F (\lambda x. \ g (h x)) S = F g T$
by (auto simp: bij-betw-def reindex)

lemma reindex-bij-witness:
assumes witness:
$\land a. \ a \in S \implies (j a) = a$
$\land a. \ a \in S \implies j a \in T$
\[ b \in T \implies j (i b) = b \]
\[ b \in T \implies i b \in S \]
assumes \( eq \):
\[ a, a \in S \implies h (j a) = g a \]
shows \( F g S = F h T \)

proof
- have \( \text{bij-betw j S T} \)
  using \( \text{bij-betw-byWitness[where A=S and f=j and f'=i and A'=T]} \)
witness by auto
moreover have \( F g S = F (\lambda x. h (j x)) S \)
  by \( \text{intro cong} \) (auto simp: eq)
ultimately show \( ?\text{thesis} \)
  by \( \text{simp add: reindex-bij-betw} \)
qed

lemma \( \text{reindex-bij-betw-not-neutral} \):
assumes \( \text{fin: finite S' finite T'} \)
assumes \( \text{bij: bij-betw h (S - S') (T - T')} \)
assumes \( \text{nn:} \)
\[ a, a \in S' \implies g (h a) = z \]
\[ b, b \in T' \implies g b = z \]
shows \( F (\lambda x. g (h x)) S = F g T \)

proof
- have \( \text{[simp]: finite S \longleftrightarrow finite T} \)
  using \( \text{bij-betw-finite[OF bij]} \) by auto
show \( ?\text{thesis} \)
  proof (cases finite S)
  case True
  with \( \text{nn} \) have \( F (\lambda x. g (h x)) S = F (\lambda x. g (h x)) (S - S') \)
    by \( \text{intro mono-neutral-cong-right} \) auto
  also have \( \ldots = F g (T - T') \)
    using \( \text{bij} \) by \( \text{rule reindex-bij-betw} \)
  also have \( \ldots = F g T \)
    using \( \text{nn} \) (finite S) by \( \text{intro mono-neutral-cong-left} \) auto
  finally show \( ?\text{thesis} \).
next
  case False
  then show \( ?\text{thesis} \) by simp
qed

lemma \( \text{reindex-nontrivial} \):
assumes \( \text{finite A} \)
  and \( \text{nz:} \)
\[ \{ x, y: x \in A \implies y \in A \implies x \neq y \implies h x = h y \implies g (h x) = 1 \} \]
shows \( F g (h ' A) = F (g \circ h) A \)
proof (subst reindex-bij-betw-not-neutral [symmetric])
  show \( \text{bij-betw h (A - \{ x \in A. (g \circ h) x = 1 \}) (h ' A - h ' \{ x \in A. (g \circ h) x = 1 \})} \)
    using \( \text{nz} \) by \( \text{auto intro! inj-onI simp: bij-betw-def} \)
lemma reindex-bij-witness-not-neutral:
  assumes fin: finite S' finite T'
  assumes witness:
    \( \forall a. a \in S - S' \implies i (j a) = a \)
    \( \forall a. a \in S - S' \implies j a \in T - T' \)
    \( \forall b. b \in T - T' \implies j (i b) = b \)
    \( \forall b. b \in T - T' \implies i b \in S - S' \)
  assumes nn:
    \( \forall a. a \in S' \implies g a = z \)
    \( \forall b. b \in T' \implies h b = z \)
  assumes eq:
    \( \forall a. a \in S' \implies h (j a) = g a \)
  shows F g S = F h T
proof
  have bij: bij-betw j (S - (S' \cap S)) (T - (T' \cap T))
    using witness by (intro bij-betw-byWitness[where f'\{=i\}]) auto
  have F-eq: F g S = F (\( \lambda x. h (j x) \)) S
    by (intro cong) (auto simp: eq)
  show \(?thesis\)
    unfolding F-eq using fin nn eq
    by (intro reindex-bij-betw-not-neutral[OF - - bij]) auto
qed

lemma delta-remove:
  assumes fS: finite S
  shows F (\( \lambda k. \) if k = a then b k else c k) S = (if a \in S then b a \* F c (S -\{a\})
  else F c (S -\{a\}))
proof
  let \(?f\) = (\( \lambda k. \) if k = a then b k else c k)
  show \(?thesis\)
    (cases a \in S)
    case False
    then have \( \forall k \in S. \) \(?f\) k = c k by simp
    with False show \(?thesis\) by simp
next
  case True
  let \(?A\) = S - \{a\}
  let \(?B\) = \{a\}
  from True have eq: S = \(?A\) \cup \(?B\) by blast
  have dj: \(?A\) \cap \(?B\) = \{\}
    by simp
  from fS have fAB: finite \(?A\) finite \(?B\) by auto
  have f \(\) if S = F \(\) if \(\) ?A \* F \(\) if \(\) ?B
    using union-disjoint \[OF fAB dj, of \(?f\), unfolded eq \[symmetric\] \] by simp
  with True show \(?thesis\)
    using comm-monoid-set.remove comm-monoid-set-axioms fS by fastforce
qed
lemma delta [simp]:
assumes fS: finite S
shows \( F (\lambda k. \text{if } k = a \text{ then } b \ k \text{ else } 1) \) \( S = (\text{if } a \in S \text{ then } b \ a \text{ else } 1) \)
by (simp add: delta-remove [OF assms])

lemma delta' [simp]:
assumes fin: finite S
shows \( F (\lambda k. \text{if } a = k \text{ then } b \ k \text{ else } 1) \) \( S = (\text{if } a \in S \text{ then } b \ a \text{ else } 1) \)
using delta [OF fin, of a b] symmetric by (auto intro: cong)

lemma If-cases:
fixes P :: 'b \Rightarrow bool and g h :: 'b \Rightarrow 'a
assumes fin: finite A
shows \( F (\lambda x. \text{if } P x \text{ then } h x \text{ else } g x) \) \( A = F h \) \( (A \cap \{ x. P x \}) \times F g \) \( (A \cap -\{ x. P x \}) \)
proof
  have a: \( A = A \cap \{ x. P x \} \cup A \cap -\{ x. P x \} \cap (A \cap -\{ x. P x \}) = \{ \} \)
    by blast+
  from fin have f: finite \( (A \cap \{ x. P x \}) \times \{ x. P x \} \cap \{ x. P x \} \)
    by auto
  let ?g = \( \lambda x. \text{if } P x \text{ then } h x \text{ else } g x \)
  from union-disjoint [OF f a(2), of ?g] a(1) show ?thesis
  by (subst (1 2)) simp-all
qed

lemma cartesian-product: \( F (\lambda x. F (g x) B) A = F \) \( (\text{case-prod } g) \) \( (A \times B) \)
proof (cases A = {} \lor B = {})
  case True
  then show ?thesis
  by auto
next
  case False
  then have A \( \neq \) {} B \( \neq \) {} by auto
  show ?thesis
  proof (cases finite A \land finite B)
    case True
    then show ?thesis
    by (simp add: Sigma)
  next
    case False
    then consider infinite A \| infinite B by auto
    then have infinite \( (A \times B) \)
      by cases (use \( A \neq \{ \} \); \( B \neq \{ \} \)) in (auto dest: finite-cartesian-productD1 finite-cartesian-productD2)
    then show ?thesis
    using False by auto
  qed
qed
lemma inter-restrict:
  assumes finite A
  shows \( F \circ g (A \cap B) = F (\lambda x. \text{if } x \in B \text{ then } g x \text{ else } 1) \) A
proof –
  let \(?g = \lambda x. \text{if } x \in A \cap B \text{ then } g x \text{ else } 1\)
  have \( \forall i \in A - A \cap B. (\text{if } i \in A \cap B \text{ then } g i \text{ else } 1) = 1 \) by simp
  moreover have \( A \cap B \subseteq A \) by blast
  ultimately have \( F ?g (A \cap B) = F ?g A \) using \( \langle \text{finite A} \rangle \) by (intro mono-neutral-left) auto
then show \(?thesis by simp
qed

lemma inter-filter:
  finite A = \( \Rightarrow \ F g \{ x \in A. P x \} = F (\lambda x. \text{if } P x \text{ then } g x \text{ else } 1) \) A
by (simp add: inter-restrict [symmetric, of A \{x. P x\} g, simplified mem-Collect-eq] Int-def)

lemma Union-comp:
  assumes \( \forall A \in B. \text{finite } A \)
  and \( \bigwedge A1 A2 x. A1 \in B \implies A2 \in B \implies A1 \neq A2 \implies x \in A1 \implies x \in A2 \)
  \( \implies g x = 1 \)
  shows \( F g (\bigcup B) = (F \circ F) g B \)
using assms
proof (induct B rule: infinite-finite-induct)
  case (infinite A)
  then have \( \neg \text{finite } (\bigcup A) \) by (blast dest: finite-UnionD)
  with infinite show \(?case by simp
next
  case empty
  then show \(?case by simp
next
  case (insert A B)
  then have finite A finite B finite (\bigcup B) A \notin B
      and \( \forall x \in A \cap \bigcup B. g x = 1 \)
      and H: \( F g (\bigcup B) = (F \circ F) g B \) by auto
  then have \( F g (A \cup \bigcup B) = F g A \ast F g (\bigcup B) \)
      by (simp add: union-inter-neutral)
  with \( \neg \text{finite } B \) \( A \notin B \) show \(?case
      by (simp add: H)
qed

lemma swap: \( F (\lambda i. F (g i) B) A = F (\lambda j. F (\lambda i. g i j) A) B \)
unfolding cartesian-product
by (rule reindex-bij-witness [where i = \lambda(i, j). (j, i) and j = \lambda(i, j). (j, i)]) auto

lemma swap-restrict:
  finite A \( \Rightarrow \) finite B \( \Rightarrow \)
THEORY “Groups-Big”

828

\[F (\lambda x. F (g x) \{y. y \in B \land R \, x \, y\}) A = F (\lambda y. F (\lambda x. g x \, y) \{x. x \in A \land R \, x \, y\}) B\]

by (simp add: inter-filter) (rule swap)

lemma image-gen:
  assumes fin: finite S
  shows F h S = F (\lambda x. F (\lambda y. h \, x \, y) \{y. y \in g \, S \land g \, x \, y\}) (g \cdot S)
proof -
  have \{y. y \in g '\, S \land g \, x \, y\} = \{g \, x\} if x \in S for x
    using that by auto
  then have F h S = \{\lambda x. F (\lambda y. h \, x \, y) \{y. y \in g \, S \land g \, x \, y\}\} S
    by simp
  also have \ldots = F (\lambda y. F (\lambda x. x \in S \land g \, x \, y)\} (g \cdot S)
    by (rule swap-restrict [OF fin finite-imageI [OF fin]])
  finally show ?thesis .
qed

lemma group:
  assumes fS: finite S and fT: finite T and fST: g \cdot S \subseteq T
  shows F (\lambda g. f T h \{x. x \in S \land g \, x \, y\}) T = F h S
unfolding image-gen[OF fS, of h g]
by (auto intro: neutral mono-neutral-right[OF fT fST])

lemma Plus:
  fixes A :: 'b set and B :: 'c set
  assumes fin: finite A finite B
  shows F g (A <+> B) = F (g \circ Inl) A \ast F (g \circ Inr) B
proof -
  have A <+> B = Inl \cdot A \cup Inr \cdot B by auto
  moreover from fin have finite (Inl \cdot A) finite (Inr \cdot B) by auto
  moreover have Inl \cdot A \cap Inr \cdot B = {} by auto
  moreover have inj-on Inl A inj-on Inr B by (auto intro: inj-onI)
  ultimately show ?thesis
    using fin by (simp add: union-disjoint reindex)
qed

lemma same-carrier:
  assumes finite C
  assumes subset: A \subseteq C B \subseteq C
  assumes trivial: \{a. a \in C - A \Rightarrow g \, a = 1 \land b \in C - B \Rightarrow h \, b = 1\}
  shows F g A = F h B \iff F g C = F h C
proof -
  have finite A and finite B and finite (C - A) and finite (C - B)
    using (finite C) subset by (auto elim: finite-subset)
  from subset have [simp]: A - (C - A) = A by auto
  from subset have [simp]: B - (C - B) = B by auto
  from subset have C = A \cup (C - A) by auto
  then have F g C = F g (A \cup (C - A)) by simp
  also have \ldots = F g (A - (C - A)) \ast F g (C - A - A) \ast F g (A \cap (C - A))
using (finite A) (finite (C − A)) by (simp only: union-diff2)
finally have *: F g C = F g A using trivial by simp
from subset have C = B ∪ (C − B) by auto
then have F h C = F h (B ∪ (C − B)) by simp
also have ... = F h (B − (C − B)) ∗ F h (C − B − B) ∗ F h (B ∩ (C − B))
using (finite B) (finite (C − B)) by (simp only: union-diff2)
finally have F h C = F h B using trivial by simp
with * show ?thesis by simp
qed

lemma same-carrierI:
assumes finite C
assumes subset: A ⊆ C B ⊆ C
assumes trivial: ∀a. a ∈ C − A ⇒ g a = 1 ∀b. b ∈ C − B ⇒ h b = 1
assumes F g C = F h C
shows F g A = F h B
using assms same-carrier [of C A B] by simp

lemma eq-general:
assumes B: ∀y. y ∈ B ⇒ ∃!x. x ∈ A ∧ h x = y and A: ∀x. x ∈ A ⇒ h x ∈ B ∧ γ(h x) = ϕ x
shows F ϕ A = F γ B
proof –
  have eq: B = h ‘ A
    by (auto dest: assms)
  have h: inj-on h A
    using assms by (blast intro: inj-onI)
  have F ϕ A = F (γ ◦ h) A
    using A by auto
  also have ... = F γ B
    by (simp add: eq reindex h)
  finally show ?thesis .
qed

lemma eq-general-inverses:
assumes B: ∀y. y ∈ B ⇒ k y ∈ A ∧ h(k y) = y and A: ∀x. x ∈ A ⇒ h x ∈ B ∧ k(h x) = x ∧ γ(h x) = ϕ x
shows F ϕ A = F γ B
by (rule eq-general [where h=h]) (force intro: dest: A B)+

42.1.2 HOL Light variant: sum/product indexed by the non-neutral subset

NB only a subset of the properties above are proved

definition G :: ['b ⇒ 'a, 'b set] ⇒ 'a
  where G p I ≡ if finite {x ∈ I. p x ≠ 1} then F p {x ∈ I. p x ≠ 1} else 1
lemma finite-Collect-op:
  shows \[
  \text{finite } \{ i \in I. \ x i \neq 1 \} \Rightarrow \text{finite } \{ i \in I. \ x i \ast \ y i \neq 1 \}
  \]
  apply (rule finite-subset [where \( B = \{ i \in I. \ x i \neq 1 \} \cup \{ i \in I. \ y i \neq 1 \} \)])
  using left-neutral by force+

lemma empty' [simp]: \( G \ p \ \{ \} = 1 \)
  by (auto simp: G-def)

lemma eq-sum [simp]: finite I \( \Rightarrow \) \( G \ p \ I = F \ p \ I \)
  by (auto simp: G-def intro: mono-neutral-cong-left)

lemma insert' [simp]:
  assumes finite \( \{ x \in I. \ p x \neq 1 \} \)
  shows \( G \ p \ (\text{insert } i I) = (\text{if } i \in I \text{ then } G \ p \ I \text{ else } p i \ast G \ p \ I) \)
proof −
  have \( \{ x. \ x = i \land p x \neq 1 \lor x \in I \land p x \neq 1 \} = (\text{if } p i = 1 \text{ then } \{ x \in I. \ p x \neq 1 \} \text{ else insert } i \ { x \in I. \ p x \neq 1 } \)\)
    by auto
  then show \?thesis using assms by (simp add: G-def conj-disj-distribR insert-absorb)
qed

lemma distrib-triv':
  assumes finite \( I \)
  shows \( G \ (\lambda i. \ g i \ast h i) I = G g I \ast G h I \)
  by (simp add: assms local.distrib)

lemma non-neutral': \( G \ g \ { x \in I. \ g x \neq 1 } = G g I \)
  by (simp add: G-def)

lemma distrib':
  assumes finite \( \{ x \in I. \ g x \neq 1 \} \) finite \( \{ x \in I. \ h x \neq 1 \} \)
  shows \( G \ (\lambda i. \ g i \ast h i) I = G g I \ast G h I \)
proof −
  have \( a \ast a \neq a \Rightarrow a \neq 1 \) for \( a \)
    by auto
  then have \( G \ (\lambda i. \ g i \ast h i) I = G (\lambda i. \ g i \ast h i) \ (\{ i \in I. \ g i \neq 1 \} \cup \{ i \in I. \ h i \neq 1 \}) \)
    using assms by (force simp: G-def finite-Collect-op intro!: mono-neutral-cong)
  also have \( \ldots = G g I \ast G h I \)
proof −
  have \( F g \ (\{ i \in I. \ g i \neq 1 \} \cup \{ i \in I. \ h i \neq 1 \}) \ast G g I \)
    \( F h \ (\{ i \in I. \ g i \neq 1 \} \cup \{ i \in I. \ h i \neq 1 \}) \ast G h I \)
    by (auto simp: G-def assms intro: mono-neutral-right)
  then show \?thesis using assms by (simp add: distrib)
qed

finally show \?thesis .
qed

lemma cong':
  assumes A = B
  assumes g-h: \( \forall x. x \in B \Rightarrow g x = h x \)
  shows G g A = G h B
  using assms by (auto simp: G-def cong: conj-cong intro: cong)

lemma mono-neutral-cong-left':
  assumes S \subseteq T
  and \( \forall i. i \in T - S \Rightarrow h i = 1 \)
  and \( \forall x. x \in S \Rightarrow g x = h x \)
  shows G g S = G h T
proof -
  have *: \( \{ x \in S. g x \neq 1 \} = \{ x \in T. h x \neq 1 \} \)
  using assms by (metis DiffI subset-eq)
  then have finite \( \{ x \in S. g x \neq 1 \} = \{ x \in T. h x \neq 1 \} \)
    by simp
  then show ?thesis
    using assms by (auto simp add: G-def * intro: cong)
qed

lemma mono-neutral-cong-right':
  S \subseteq T \Rightarrow \forall i \in T - S. g i = 1 \Rightarrow (\forall x. x \in S \Rightarrow g x = h x) \Rightarrow G g T = G h S
by (auto intro!: mono-neutral-cong-left' [symmetric])

lemma mono-neutral-left': S \subseteq T \Rightarrow \forall i \in T - S. g i = 1 \Rightarrow G g S = G g T
by (blast intro: mono-neutral-cong-left')

lemma mono-neutral-right': S \subseteq T \Rightarrow \forall i \in T - S. g i = 1 \Rightarrow G g T = G g S
by (blast intro!: mono-neutral-left' [symmetric])
end

42.2 Generalized summation over a set

class comm_monoid_add
begin

abbreviation Sum (\( \sum \)) where \( \sum \equiv \text{sum} (\lambda x. x) \)
end
THEORY "Groups-Big"

Now: lots of fancy syntax. First, \( \sum (\lambda x. \, e) \, A \) is written \( \sum x \in A. \, e \).

**syntax (ASCII)**

\[-\text{sum} :: \text{pttrn} \Rightarrow 'a \, \text{set} \Rightarrow 'b \Rightarrow 'b::\text{comm-monoid-add} \quad ((\text{3SUM} \ (-/\cdot)/ \cdot) \, [0, 51, 10])\]

**translations** — Beware of argument permutation!

\[ \sum x \in A. \, b \Rightarrow \text{CONST} \, \text{sum} \, (\lambda i. \, b) \, A \]

Instead of \( \sum x \in \{x. \, P\}. \, e \) we introduce the shorter \( \sum x | P. \, e \).

**syntax (ASCII)**

\[-\text{qsum} :: \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow 'a \quad ((\text{SUM} \ (-/\cdot)/ \cdot) \, [0, 0, 10])\]

**translations**

\[ \sum x | P. \, t \Rightarrow \text{CONST} \, \text{sum} \, (\lambda x. \, t) \, \{x. \, P\} \]

**print-translation**

let

\[ \text{fun sum-tr'} [\text{Abs} \, (x, \, Tx, \, t), \, \text{Const} \, (\text{const-synt x} \, \text{Collect}, \, -) \, \$ \, \text{Abs} \, (y, \, Ty, \, P)]] = \]

\[ \text{if } x \not<\not> y \text{ then raise Match } \]

\[ \text{else } \]

\[ \text{let } \]

\[ \text{val } x' = \text{Syntax-Trans.mark-bound-body} \, (x, \, Tx); \]

\[ \text{val } t' = \text{subst-bound} \, (x', \, t); \]

\[ \text{val } P' = \text{subst-bound} \, (x', \, P); \]

\[ \text{in } \]

\[ \text{Syntax.const synt x} \, \text{-qsum} \, \$ \, \text{Syntax-Trans.mark-bound-abs} \, (x, \, Tx) \, \$ \, P' \, \$ \, t' \]

\[ \text{end } \]

\[ \mid \text{sum-tr'} - = \text{raise Match}; \]

\[ \text{in } [(\text{const-synt x} \, \text{sum}, \, K \, \text{sum-tr'}]] \text{ end } \]

### 42.2.1 Properties in more restricted classes of structures

**lemma** \( \text{sum-Un} \):  

\[ \text{finite } A \Rightarrow \text{finite } B \Rightarrow \text{sum } f \, (A \cup B) = \text{sum } f \, A + \text{sum } f \, B - \text{sum } f \, (A \cap B) \]

for \( f :: 'b \Rightarrow 'a::\text{ab-group-add} \)  
by (\text{subst sum.union-inter [symmetric]} \, (\text{auto simp add: algebra-simps} \))

**lemma** \( \text{sum-Un2} \):

\[ \text{assumes } \text{finite } (A \cup B) \]

\[ \text{shows } \text{sum } f \, (A \cup B) = \text{sum } f \, (A - B) + \text{sum } f \, (B - A) + \text{sum } f \, (A \cap B) \]

**proof** —

\[ \text{have } A \cup B = A - B \cup (B - A) \cup A \cap B \]
by auto
with assms show \( ?\text{thesis} \)
  by simp (subst sum, union-disjoint, auto)+
qed

lemma sum-diff1:
  fixes \( f : 'b \Rightarrow 'a::ab-group-add \)
  assumes finite \( A \)
  shows \( \sum f \ (A - \{a\}) = (if \ a \in A \ then \ \sum f A - f \ a \ else \ \sum f A) \)
  using assms by induct (auto simp: insert-Diff-if)

lemma sum-diff:
  fixes \( f : 'b \Rightarrow 'a::ab-group-add \)
  assumes finite \( A \ B \subseteq A \)
  shows \( \sum f \ (A - B) = \sum f A - \sum f B \)
proof
  from assms(2,1) have finite \( B \) by (rule finite-subset)
  from this \( (B \subseteq A) \)
  show \( ?\text{thesis} \)
  proof
    induct
    case empty
    thus \( ?\text{case} \) by simp
  next
    case (insert \( x \) \( F \))
    with \( \langle \)finite \( A \) \( \rangle \) \( \langle \)finite \( B \) \( \rangle \) show \( ?\text{case} \)
      by (simp add: Diff-insert[where \( a=x \) and \( B=F \)] sum-diff insert-absorb)
  qed
qed

lemma sum-diff1'-aux:
  fixes \( f : 'a \Rightarrow 'b::ab-group-add \)
  assumes finite \( F \ \{i \in I. f i \neq 0\} \subseteq F \)
  shows \( \sum' f \ (I - \{i\}) = (if \ i \in I \ then \ \sum' f I - f \ i \ else \ \sum' f I) \)
  using assms
proof
  induct
  case (insert \( x \) \( F \))
  have 1: finite \( \{x \in I. f x \neq 0\} \Rightarrow \) finite \( \{x \in I. x \neq i \land f x \neq 0\} \)
    by (erule rev-finite-subset) auto
  have 2: finite \( \{x \in I. x \neq i \land f x \neq 0\} \Rightarrow \) finite \( \{x \in I. f x \neq 0\} \)
    apply (erule finite-insert[THEN iffD2])
    by (erule rev-finite-subset) auto
  have 3: finite \( \{i \in I. f i \neq 0\} \)
    using finite-subset insert by blast
  show \( ?\text{case} \)
    using insert sum-diff1 [of \( \{i \in I. f i \neq 0\} \) \( f i \)]
    by (auto simp: sum.G-def 1 2 3 set-diff-eq conj-ac)
  qed (simp add: sum.G-def)

lemma sum-diff1':
fixes \( f :: \alpha \Rightarrow 'b::{ab-group-add} \)
assumes finite \( \{i \in I. f i \neq 0\} \)
shows \( \sum' f \{I - \{i\}\} = (if i \in I then \sum' f I - f i else \sum' f I) \)
by (rule sum-diff1'-'aux [OF assms order-refl])

lemma (in ordered-comm-monoid-add) sum-mono:
\( (\forall i \in K. f i \leq g i) \Rightarrow (\sum i \in K. f i) \leq (\sum i \in K. g i) \)
by (induct K rule: infinite-finite-induct) (use add-mono in auto)

lemma (in strict-ordered-comm-monoid-add) sum-strict-mono:
assumes finite A and \( \forall x \in A. f x < g x \)
shows \( \sum f A < \sum g A \)
proof (induct rule: finite-ne-induct)
  case singleton then show ?case by simp
  next
  case insert then show ?case by (auto simp: add-strict-mono)
qed

lemma sum-strict-mono-ex1:
fixes \( f, g :: 'a::{ordered-cancel-comm-monoid-add} \)
assumes finite A and \( \forall x \in A. f x \leq g x \) and \( \exists a \in A. f a < g a \)
shows \( \sum f A < \sum g A \)
proof
  from assms(3) obtain a where a: a \in A \( f a < g a \) by blast
  have sum f A = sum f \((A - \{a\}) \cup \{a\}\)
  by (simp add: insert-absorb[of a \( a \in A\)])
  also have \( \ldots = \sum f (A - \{a\}) + \sum f \{a\} \)
  using finite A by (subst sum.union_disjoint) auto
  also have sum f \((A - \{a\})\) \( \leq \sum g (A - \{a\}) \)
  by (rule sum_mono) (simp add: assms(2))
  also from a have sum f \{a\} \( < \sum g \{a\} \) by simp
  also have sum g \((A - \{a\})\) \( + \sum g \{a\} = \sum g(A - \{a\}) \cup \{a\} \)
  using finite A by (subst sum.union_disjoint) auto
  also have \( \ldots = \sum g A \) by (simp add: insert-absorb[of a \( a \in A\)])
  finally show ?thesis by (auto simp add: add-right_mono add-strict-left_mono)
qed

lemma sum-mono-inv:
fixes \( f, g :: 'i \Rightarrow 'a::{ordered-cancel-comm-monoid-add} \)
assumes eq: \( \sum f I = \sum g I \)
assumes le: \( \forall i \in I. f i \leq g i \)
assumes i: i \in I

assumes $I$: finite $I$
shows $f_i = g_i$
proof (rule contr)
  assume $\neg \ ?thesis$
  with $le[\ OF\ i\ ]$ have $f_i < g_i$ by simp
  with $i$ have $\exists i \in I. f_i < g_i$ ..
  from sum-strict-mono-ex1 [OF $I$ - this] le have $\sum f I < \sum g I$
    by blast
  with eq show False by simp
qed

lemma member-le-sum:
  fixes $f :: 'b::{semiring-1, ordered-comm-monoid-add}$
  assumes $i \in A$
    and $le: \ \land x. x \in A - \{i\} \implies 0 \leq f x$
    and finite $A$
shows $f_i \leq \sum f A$
proof
  have $f_i \leq \sum f (A \cap \{i\})$
    by (simp add: assms)
  also have $.. = (\sum x\in A. if x \in \{i\} then f x else 0)$
    using assms sum.inter-restrict by blast
  also have $.. \leq \sum f A$
    apply (rule sum-mono)
    apply (auto simp: le)
  done
finally show $\ ?thesis$ .
qed

lemma sum-negf: $(\sum x\in A. - f x) = - (\sum x\in A. f x)$
  for $f :: 'b => 'a::ab-group-add$
  by (induct $A$ rule: infinite-finite-induct) auto

lemma sum-subtractf: $(\sum x\in A. f x - g x) = (\sum x\in A. f x) - (\sum x\in A. g x)$
  for $f \ g :: 'b => 'a::ab-group-add$
  using sum.distrib [of $f - g A$] by (simp add: sum-negf)

lemma sum-subtractf-nat:
  $(\land x. x \in A \implies g x \leq f x) \implies (\sum x\in A. f x - g x) = (\sum x\in A. f x) - (\sum x\in A. g x)$
  for $f \ g :: 'a => nat$
  by (induct $A$ rule: infinite-finite-induct) (auto simp: sum-mono)

context ordered-comm-monoid-add
begin

lemma sum-nonneg: $(\land x. x \in A \implies 0 \leq f x) \implies 0 \leq \sum f A$
proof (induct $A$ rule: infinite-finite-induct)
  case infinite
then show \( ?\text{case} \) by simp
next
case empty
then show \( ?\text{case} \) by simp
next
case \((\text{insert } x \ F)\)
then have \( 0 + 0 \leq f x + \sum\ f F \) by (blast intro: add-mono)
with \text{insert} show \( ?\text{case} \) by simp
qed

lemma \text{sum-nonpos}: \((\forall x. x \in A \implies f x \leq 0) \implies \sum f A \leq 0\)
proof (induct A rule: infinite-finite-induct)
case infinite
then show \( ?\text{case} \) by simp
next
case empty
then show \( ?\text{case} \) by simp
next
case \((\text{insert } x \ F)\)
then have \( f x + \sum f F \leq 0 + 0 \) by (blast intro: add-mono)
with \text{insert} show \( ?\text{case} \) by simp
qed

lemma \text{sum-nonneg-eq-0-iff}:
finite A \implies \((\forall i. i \in s \implies f i \geq 0) \implies (\sum i \in s. f i) = 0 \implies i \in s \implies f i = 0\)
by (simp add: sum-nonneg-eq-0-iff sum-nonneg)

lemma \text{sum-nonneg-leq-bound}:
assumes finite s \( (\forall i. i \in s \implies f i \geq 0) \implies (\sum i \in s. f i) \leq B \implies i \in s \implies f i \leq B\)
proof 
from \text{assms} have \( f i \leq f i + (\sum i \in s - \{i\}. f i) \)
by (intro add-increasing2 sum-nonneg) auto
also have \( \ldots = B \)
using sum.remove[of s f] \text{assms} by simp
finally show \( ?\text{thesis} \) by auto
qed

lemma \text{sum-mono2}:
assumes fin: \text{finite } B
and sub: \( A \subseteq B\)
and nn: \( \forall b. b \in B - A \implies 0 \leq f b\)
shows \( \sum f A \leq \sum f B\)
proof –
have \( \sum f A \leq \sum f A + \sum f (B - A) \)
by (auto intro: add-increasing2 [OF sum-nonneg] nn)
also from fin finite-subset[OF sub fin] have \( \ldots = \sum f (A \cup (B - A)) \)
by (simp add: sum.union-disjoint del: Un-Diff-cancel)
also from sub have \( A \cup (B - A) = B \) by blast
finally show \( \text{thesis} \).
qed

lemma sum-le-included:
assumes finite s finite t
and \( \forall y \in t. 0 \leq g y \) \( (\forall x \in s. \exists y \in t. i y = x \land f x \leq g y) \)
says \( \sum f s \leq \sum g t \)
proof
− have \( \sum f s \leq \sum (\lambda y. \sum g \{x. x \in t \land i x = y\}) s \)
proof (rule sum-mono)
fix y
assume \( y \in s \)
with assms obtain \( z \) where \( z \in t \ y = i z \ y \leq g z \) by auto
with assms show \( f y \leq \sum g \{x \in t. i x = y\} \) \( (\forall \ A y \leq B y) \)
using order-trans[of ?A \( i z \) \( \sum g \{z\} \) \( ?B (i z), \text{intro} \)]
by (auto intro!: sum-mono2)
qed
also have \( \ldots \leq \sum (\lambda y. \sum g \{x \in t \land i x = y\}) \) \( (i \ ' t) \)
using assms(2-4) by (auto intro!: sum-mono2 sum-nonneg)
also have \( \ldots \leq \sum g t \)
using assms by (auto simp: sum.image-gen[symmetric])
finally show \( \text{thesis} \).
qed
end

lemma (in canonically-ordered-monoid-add) sum-eq-0-iff [simp]:
finite F \( \implies \) \( (\sum f F = 0) = (\forall a \in F. f a = 0) \)
by (intro ballI sum-nonneg-eq-0-iff zero-le)

context semiring-0
begin

lemma sum-distrib-left: \( r \ast \sum f A = (\sum n \in A. r \ast f n) \)
by (induct A rule: infinite-finite-induct) (simp-all add: algebra-simps)

lemma sum-distrib-right: \( \sum f A \ast r = (\sum n \in A. f n \ast r) \)
by (induct A rule: infinite-finite-induct) (simp-all add: algebra-simps)

end

lemma sum-divide-distrib: \( \sum f A / r = (\sum n \in A. f n / r) \)
for \( r :: 'a::field \)
proof (induct A rule: infinite-finite-induct)
case infinite
  then show ?case by simp
next
  case empty
  then show ?case by simp
next
  case insert
  then show ?case by (simp add: add-divide-distrib)
qed

lemma sum-abs[iff]: \(|\sum f A| \leq \sum (\lambda i. |f i|) A\)
  for f :: 'a ⇒ 'b::ordered-ab-group-add-abs
proof (induct A rule: infinite-finite-induct)
  case infinite
  then show ?case by simp
next
  case empty
  then show ?case by simp
next
  case (insert a A)
  then have \(|\sum a\in insert a A. |f a|\) = \(||f a| + \sum a\in A. |f a|\) by simp
  also from insert have \(\ldots = ||f a| + \sum a\in A. |f a||\) by simp
  also have \(\ldots = |f a| + \sum a\in A. |f a|\) by (simp del: abs-of-nonneg)
  also from insert have \(\ldots = (\sum a\in insert a A. |f a|)\) by simp
  finally show ?case .
qed

lemma sum-product:
  fixes f :: 'a ⇒ 'b::semiring-0
  shows \(\sum f A * \sum g B = (\sum i\in A. \sum j\in B. f i * g j)\)
  by (simp add: sum-distrib-left sum-distrib-right) (rule sum_swap)
lemma sum-mult-sum-if-inj:
  fixes f :: 'a ⇒ 'b::semiring-0
  shows inj-on (λ(a, b). f a * g b) (A × B) ⇒
      sum f A * sum g B = sum id { f a * g b | a b. a ∈ A ∧ b ∈ B}
by(auto simp: sum-product sum.cartesian-product intro!: sum.reindex-cong[symmetric])

lemma sum-SucD: sum f A = Suc n ⇒ ∃ a∈A. 0 < f a
by (induct A rule: infinite-finite-induct) auto

lemma sum-eq-Suc0-iff:
  finite A ⇒ sum f A = Suc 0 ↔ (∃ a∈A. f a = Suc 0 ∧ (∀ b∈A. a ≠ b → f b = 0))
by (induct A rule: finite-induct) (auto simp add: add-is-1)

lemmas sum-eq-1-iff = sum-eq-Suc0-iff[simplified One-nat-def][symmetric]

lemma sum-Un-nat:
  finite A ⇒ finite B ⇒ sum f (A ∪ B) = sum f A + sum f B − sum f (A ∩ B)
  for f :: 'a ⇒ nat
— For the natural numbers, we have subtraction.
by (subst sum.union-inter [symmetric]) (auto simp: algebra-simps)

lemma sum-diff1-nat: sum f (A − {a}) = (if a ∈ A then sum f A − f a else sum f A)
  for f :: 'a ⇒ nat
proof (induct A rule: infinite-finite-induct)
case infinite
  then show ?case by simp
next
case empty
  then show ?case by simp
next
case insert
  then show ?case
    apply (auto simp: insert-Diff-if)
    apply (drule mk-disjoint-insert)
    apply auto
    done
  qed

lemma sum-diff-nat:
  fixes f :: 'a ⇒ nat
  assumes finite B and B ⊆ A
  shows sum f (A − B) = sum f A − sum f B
using assms
proof induct
case empty
  then show ?case by simp
next
case (insert x F)

note IH = \( F \subseteq A \implies \sum f (A - F) = \sum f A - \sum f F \) from (insert x F \subseteq A) have \( x \in A - F \) by simp then have \( A: \sum f ((A - F) - \{x\}) = \sum f (A - F) - f x \) by (simp add: sum-diff1-nat) from (insert x F \subseteq A) have \( F \subseteq A \) by simp with IH have \( \sum f (A - F) = \sum f A - \sum f F \) by simp with \( A \) have \( B: \sum f ((A - F) - \{x\}) = \sum f A - \sum f F - f x \) by simp from (finite F) (x /\in F) have \( A - \text{insert } x F = (A - F) - \{x\} \) by auto with \( B \) have \( C: \sum f (A - \text{insert } x F) = \sum f A - \sum f (\text{insert } x F) \) by simp then show \( ?\) by simp

qed

lemma sum-comp-morphism:
\( h 0 = 0 \implies (\forall x y. h (x + y) = h x + h y) \implies \sum (h \circ g) A = h (\sum g A) \)
by (induct A rule: infinite-finite-induct) simp-all

lemma (in comm-semiring-1) ded-sum: \( (\forall a. a \in A \implies d \vdash f a) \implies d \vdash \sum f A \)
by (induct A rule: infinite-finite-induct) simp-all

lemma (in ordered-comm-monoid-add) sum-pos:
\( \text{finite } I \implies I \neq \{\} \implies (\forall i. i \in I \implies 0 < f i) \implies 0 < \sum f I \)
by (induct I rule: finite-ne-induct) (auto intro: add-pos-pos)

lemma (in ordered-comm-monoid-add) sum-pos2:
assumes \( I: \text{finite } I \subseteq I 0 < f i \forall i. i \in I \implies 0 \leq f i \)
shows \( 0 < \sum f I \)
proof
have \( 0 < f i + \sum f (I - \{i\}) \)
  using assms by (intro add-pos-nonneg sum-nonneg) auto
also have \( \ldots = \sum f I \)
  using assms by (simp add: sum.remove)
finally show \( ?\) by simp
qed

lemma sum-cong-Suc:
assumes \( 0 /\in A \forall x. \text{Suc } x \in A \implies f (\text{Suc } x) = g (\text{Suc } x) \)
shows \( \sum f A = \sum g A \)
proof (rule sum.cong)
fix \( x \)
assume \( x \in A \)
with assms(1) show \( f x = g x \)
  by (cases \( x \)) (auto intro!: assms(2))
qed simp-all

42.2.2 Cardinality as special case of \( \sum \)

lemma card-eq-sum: \( \text{card } A = \sum (\lambda x. 1) A \)
proof -
  have plus \( \circ (\lambda x. \text{Suc } 0) = (\lambda x. \text{Suc}) \)
    by (simp add: fun-eq_iff)
  then have Finite-Set.fold (plus \( \circ (\lambda x. \text{Suc } 0) \)) = Finite-Set.fold (\( \lambda x. \text{Suc} \)) 0 A
    by (rule arg-cong)
  then have Finite-Set.fold (plus \( \circ (\lambda x. \text{Suc } 0) \)) \( \text{of-nat}(\text{card } A) \) = Finite-Set.fold (\( \lambda x. \text{Suc} \)) \( \text{of-nat}(\text{card } A) \)
    by (blast intro: fun-cong)
  then show \(?thesis \)
    by (simp add: card.eq_fold sum.eq_fold)
qed

context semiring-1

begin

lemma sum-constant [simp]:
  \( \sum x \in A. y = \text{of-nat}(\text{card } A) \times y \)
  by (induct A rule: infinite-finite-induct) (simp-all add: algebra-simps)

end

lemma sum-Suc: \( \sum (\lambda x. \text{Suc}(f x)) A = \sum f A + \text{card } A \)
using sum.distrib[of f \( \lambda x. 1 \) A] by simp

lemma sum-bounded-above:
  fixes \( K :: 'a::{semiring-1,ordered-comm-monoid-add} \)
  assumes le: \( \forall i. i \in A \rightarrow f i \leq K \)
  shows sum f A \( \leq \text{of-nat}(\text{card } A) \times K \)
proof (cases finite A)
  case True
  then show \(?thesis \)
    using le sum-mono[where \( K=A \) and \( g = \lambda x. K \)] by simp
next
  case False
  then show \(?thesis \) by simp
qed

lemma sum-bounded-above-divide:
  fixes \( K :: 'a::linordered-field \)
  assumes le: \( \forall i. i \in A \rightarrow f i \leq K / \text{of-nat}(\text{card } A) \) and fin: finite \( A \) \( A \neq \emptyset \)
  shows sum f A \( \leq K \)
using sum-bounded-above [of A f K / \text{of-nat}(\text{card } A), OF le] fin by simp
lemma sum-bounded-above-strict:
fixes \( K \) :: 'a::{ordered-cancel-comm-monoid-add,semiring-1}
assumes \( \forall i. \ i \in A \implies f_i < K \ \text{card} \ A > 0 \)
shows \( \text{sum} \ f \ A < \text{of-nat} (\text{card} \ A) * K \)
using assms sum-strict-mono[where \( A = A \) and \( g = \lambda x. \ K \)]
by (simp add: card-gt-0-iff)

lemma sum-bounded-below:
fixes \( K \) :: 'a::{semiring-1,ordered-comm-monoid-add}
assumes \( \le \): \( \forall i. \ i \in A \implies K \le f_i \)
shows \( \text{of-nat} (\text{card} \ A) * K \le \text{sum} \ f \ A \)
proof (cases finite \( A \))
case True
then show \( \text{thesis} \)
using \( \le \) sum-mono[where \( K = A \) and \( f = \lambda x. \ K \)]
by simp
next
case False
then show \( \text{thesis} \)
by simp
qed

lemma convex-sum-bound-le:
fixes \( x \) :: 'a\Rightarrow 'b::linordered-idom
assumes \( 0 \): \( \forall i. \ i \in I \implies 0 \le x_i \) and \( 1 \): \( \text{sum} \ x \ I = 1 \)
and \( \delta \): \( \forall i. \ i \in I \implies |a_i - b| \le \delta \)
shows \( |(\sum i \in I. \ a_i * x_i) - b| \le \delta \)
proof -
have [simp]: \((\sum i \in I. \ c * x_i) = c\) for \( c \)
by (simp flip: sum-distrib-left 1)
then have \(|(\sum i \in I. \ a_i * x_i) - b| = |(\sum i \in I. \ (a_i - b) * x_i)|\)
by (simp add: sum-subtractf left-diff-distrib)
also have \( \ldots \le (\sum i \in I. \ |(a_i - b) * x_i|) \)
using abs-abs abs-of-nonneg by blast
also have \( \ldots \le (\sum i \in I. \ |(a_i - b)| * x_i) \)
by (simp add: abs-mult 0)
also have \( \ldots \le (\sum i \in I. \ \delta * x_i) \)
by (rule sum-mono) (use \( \delta \) 0 mult-right-mono in blast)
also have \( \ldots = \delta \)
by simp
finally show \( \text{thesis} \).
qed

lemma card-UN-disjoint:
assumes \( \text{finite} \ I \) and \( \forall i \in I. \ \text{finite} \ (A \ i) \)
and \( \forall i \in I. \ \forall j \in I. \ i \neq j \implies A \ i \cap A \ j = \{\} \)
sshows \( \text{card} \ (\bigcup (A \ i)) = (\sum i \in I. \ \text{card}(A \ i)) \)
proof -
have \((\sum i \in I. \ \text{card} \ (A \ i)) = (\sum i \in I. \ \sum x \in A \ i. \ 1)\)
by simp
with assms show ?thesis
  by (simp add: card-eq-sum sum.UNION-disjoint del: sum-constant)
qed

lemma card-Union-disjoint:
  assumes pairwise disjoint C and fin: \(\forall A. A \in C \implies \text{finite } A\)
  shows \(\text{card } (\bigcup C) = \text{sum card } C\)
proof (cases finite C)
  case True
  then show ?thesis
  using card-UN-disjoint [OF True, of \(\lambda x. x\)] assms
  by (simp add: disjoint-def fin pairwise-def)
next
  case False
  then show ?thesis
  using assms card-eq-0-iff finite-UnionD by fastforce
qed

lemma card-UN-le-sum-card:
  fixes \(U : \text{a set set}\)
  assumes \(\forall u \in U. \text{finite } u\)
  shows \(\text{card } (\bigcup U) \leq \text{sum card } U\)
proof (cases finite U)
  case False
  then show card \((\bigcup U) \leq \text{sum card } U\)
  using card-eq-0-iff finite-UnionD by auto
next
  case True
  then show card \((\bigcup U) \leq \text{sum card } U\)
  proof (induct U rule: finite-induct)
    case empty
    then show ?case by auto
next
  case (insert x F)
  then have \(\text{card } (\bigcup (\text{insert } x F)) \leq \text{card } x + \text{card } (\bigcup F)\) using card-Un-le by auto
  also have \(\ldots \leq \text{card } x + \text{sum card } F\) using insert.hyps by auto
  also have \(\ldots = \text{sum card } (\text{insert } x F)\) using sum.insert-if and insert.hyps by auto
  finally show ?case .
  qed
qed

lemma card-UN-le:
  assumes finite I
  shows \(\text{card } (\bigcup i \in I. A \ i) \leq \sum i \in I. \text{card } (A \ i)\)
using assms
proof induction
  case (insert i I)
then show \(?\)case
  using card-Un-le nat-add-left-cancel-le by (force intro: order-trans)
qed auto

lemma sum-multicount-gen:
  assumes finite s finite t \(\forall j \in t. (\text{card } \{i \in s. R i j\} = k j)\)
  shows sum (\(\lambda i. (\text{card } \{j \in t. R i j\})\)) s = sum k t
  (is \(?l = ?r\))
proof –
  have \(?l = \text{sum } (\lambda i. \text{sum } (\lambda x. 1) \{j \in t. R i j\})\) s
    by auto
  also have \(\ldots = ?r\)
  unfolding sum.swap-restrict [OF assms(1-2)]
  using assms(3) by auto
  finally show \(?\)thesis .
qed

lemma sum-multicount:
  assumes finite S finite T \(\forall j \in T. (\text{card } \{i \in S. R i j\} = k)\)
  shows sum (\(\lambda i. \text{card } \{j \in T. R i j\}\)) S = k * card T
  (is \(?l = ?r\))
proof –
  have \(?l = \text{sum } (\lambda i. k)\) T
    by (rule sum-multicount-gen) (auto simp: assms)
  also have \(\ldots = ?r\) by (simp add: mult.commute)
  finally show \(?\)thesis by auto
qed

lemma sum-card-image:
  assumes finite A
  assumes pairwise \((\lambda s t. \text{disjnt } (f s) (f t))\) A
  shows sum card (f ' A) = sum (\(\lambda a. \text{card } (f a)\)) A
  using assms
proof (induct A)
  case (insert a A)
  show \(?\)case
  proof cases
    assume f a = {} 
    with insert show \(?\)case
    by (subst sum.mono-neutral-right[where \(S=f ' A\)]) (auto simp: pairwise-insert)
  next
    assume f a \(\neq\) {}
    then have \(\text{sum } \text{card } (\text{insert } (f a) (f ' A)) = \text{card } (f a) + \text{sum } \text{card } (f ' A)\)
      using insert
      by (subst sum.insert) (auto simp: pairwise-insert)
    with insert show \(?\)case by (simp add: pairwise-insert)
  qed
  qed simp
42.2.3 Cardinality of products

**Lemma** $\text{card-SigmaI [simp]}$:

finite $A \implies \forall a \in A$. finite $(B a) \implies \text{card } (\Sigma x : A. B x) = (\sum a \in A. \text{card } (B a))$

by $(\text{simp add: card-eq-sum sum}.\Sigma \text{del: sum-constant})$

**Lemma** $\text{card-cartesian-product}$: $\text{card } (A \times B) = \text{card } A * \text{card } B$

by $(\text{cases finite } A \land \text{finite } B)$ $(\text{auto simp add: card-eq-0-iff dest: finite-cartesian-productD1 finite-cartesian-productD2})$

**Lemma** $\text{card-cartesian-product-singleton}$: $\text{card } \{x\} \times A = \text{card } A$

by $(\text{simp add: card-cartesian-product})$

42.3 Generalized product over a set

context $\text{comm-monoid-mult}$

begin

sublocale prod $\text{comm-monoid-set times 1}$

defines $\text{prod } = \text{prod}.F$

and $\text{prod'} = \text{prod}.G$ ..

abbreviation $\text{Prod }$ $\equiv \prod \lambda x. x$

end

**Syntax (ASCII)**

$\prod$ $\text{pttrn} = > 'a \text{ set} = > 'b = > 'b::\text{comm-monoid-mult } ((4\text{PROD }/-)/-)[0, 51, 10] 10$

**Syntax**

$\prod$ $\text{pttrn} = > 'a \text{ set} = > 'b = > 'b::\text{comm-monoid-mult } ((2\prod (-)\in)/-) [0, 51, 10] 10$

**Translations** — Beware of argument permutation!

$\prod i \in A. b = = \text{CONST prod } (\lambda i. b) A$

Instead of $\prod x \in \{x. P\}$. $e$ we introduce the shorter $\prod x | P. e$.

**Syntax (ASCII)**

$\prod'p = | (4\text{PROD }-)/-)[0, 0, 10] 10$

**Syntax**

$\prod'p = | (2\prod (-))/-) [0, 0, 10] 10$

**Translations**

$\prod x | P. t = > \text{CONST prod } (\lambda x. t) \{x. P\}$

context $\text{comm-monoid-mult}$

begin

**Lemma** $\text{prod-dvd-prod}$: $(\forall a. a \in A \implies f a dvd g a) \implies prod f A dvd prod g A$
proof (induct A rule: infinite-finite-induct)
  case infinite
  then show ?case by (auto intro: dvdI)
next
  case empty
  then show ?case by (auto intro: dvdI)
next
  case (insert a A)
  then have f a dvd g a and prod f A dvd prod g A
      by simp
  then obtain r s where g a = f a * r and prod g A = prod f A * s
      by (auto elim: dvdE)
  then have g a * prod g A = f a * prod f A * (r * s)
      by (simp add: ac-simps)
  with insert.hyps show ?case
      by (auto intro: dvdI)
qed

lemma prod-dvd-prod-subset: finite B ⇒ A ⊆ B ⇒ prod f A dvd prod f B
  by (auto simp add: prod_subset_diff ac-simps intro: dvdI)

end

42.3.1 Properties in more restricted classes of structures

context linordered-nonzero-semiring

lemma prod-ge-1: (∀x. x ∈ A ⇒ 1 ≤ f x) ⇒ 1 ≤ prod f A
proof (induct A rule: infinite-finite-induct)
  case infinite
  then show ?case by simp
next
  case empty
  then show ?case by simp
next
  case (insert x F)
  have 1 * 1 ≤ f x * prod f F
      by (rule mult-mono') (use insert in auto)
  with insert show ?case by simp
qed

lemma prod-le-1:
  fixes f :: 'a ⇒ 'a
  assumes (∀x. x ∈ A ⇒ 0 ≤ f x ∧ f x ≤ 1)
  shows prod f A ≤ 1
    using assms
proof (induct A rule: infinite-finite-induct)
  case infinite
then show ?case by simp
next
case empty
then show ?case by simp
next
case (insert x F)
then show ?case by (force simp: mult.commute intro: dest: mult-le-one) qed

end

context comm-semiring-1
begin

lemma dvd-prod-eqI [intro]:
  assumes finite A and a ∈ A and b = f a
  shows b dvd prod f A
proof −
  from ⟨finite A⟩ have prod f (insert a (A − {a})) = f a * prod f (A − {a})
    by (intro prod.insert) auto
also from ⟨a ∈ A⟩ have insert a (A − {a}) = A
    by blast
finally have prod f A = f a * prod f (A − {a})
  with ⟨b = f a⟩ show ?thesis
    by simp
qed

lemma dvd-prodI [intro]: finite A ⇒ a ∈ A ⇒ f a dvd prod f A
  by auto

lemma prod-zero:
  assumes finite A and ∃ a ∈ A. f a = 0
  shows prod f A = 0
  using assms
proof (induct A)
case empty
then show ?case by simp
next
case (insert a A)
then have f a = 0 ∨ (∃ a ∈ A. f a = 0) by simp
then have f a * prod f A = 0 by rule (simp-all add: insert)
with insert show ?case by simp
qed

lemma prod-dvd-prod-subset2:
  assumes finite B and A ⊆ B and a ∈ A ⇒ f a dvd g a
  shows prod f A dvd prod g B
proof −
  from assms have prod f A dvd prod g A
by (auto intro: prod-dvd-prod)
moreover from assms have prod g A dvd prod g B
  by (auto intro: prod-dvd-prod-subset)
ultimately show ?thesis by (rule dvd-trans)
qed

end

lemma (in semidom) prod-zero-iff [simp]:
  fixes f :: 'b ⇒ 'a
  assumes finite A
  shows prod f A = 0 ←→ (∃a∈A. f a = 0)
  using assms by (induct A) (auto simp: no-zero-divisors)

lemma (in semidom-divide) prod-diff1:
  assumes finite A and f a ≠ 0
  shows prod f (A − {a}) = (if a ∈ A then prod f A div f a else prod f A)
proof (cases a ∉ A)
  case True
  then show ?thesis by simp
next
  case False
  with assms show ?thesis
proof induct
    case empty
    then show ?case by simp
next
    case (insert b B)
    then show ?case
proof (cases a = b)
      case True
      with insert show ?thesis by simp
next
      case False
      with insert have a ∈ B by simp
      define C where C = B − {a}
      with (finite B) (a ∈ B) have B = insert a C finite C a ∉ C
      by auto
      with insert show ?thesis
      by (auto simp add: insert-commute ac-simps)
    qed
  qed

lemma sum-zero-power [simp]: (∑i∈A. c i * 0 ^ i) = (if finite A ∧ 0 ∈ A then c 0 else 0)
  for c :: nat ⇒ 'a::division-ring
  by (induct A rule: infinite-finite-induct) auto
lemma sum-zero-power' [simp]:
(\sum_{i \in A} c \cdot i \cdot 0^{i / d_i}) = (\text{if finite } A \land 0 \in A \text{ then } c_0 / d_0 \text{ else } 0)
for \(c :: \text{nat} \Rightarrow 'a::field\)
using sum-zero-power [of \(\lambda i. c \cdot i / d_i A\)] by auto

lemma (in field) prod-inversef: \(\prod (\text{inverse} \circ f) A = \text{inverse} (\prod f A)\)
proof (cases finite A)
  case True
  then show ?thesis
  by (induct A rule: finite-induct) simp-all
next
  case False
  then show ?thesis
  by auto
qed

lemma (in field) prod-dividef: \((\prod_{x \in A} f x / g x) = \prod f A / \prod g A\)
using prod-inversef [of g A] by (simp add: divide-inverse prod distrib)

lemma prod-Un:
fixes f :: 'b \Rightarrow 'a :: field
assumes finite A and finite B and \(\forall x \in A \cap B. f x \neq 0\)
shows \(\prod f (A \cup B) = \prod f A \ast \prod f B / \prod f (A \cap B)\)
proof
  from assms have \(\prod f A \ast \prod f B = \prod f (A \cup B) \ast \prod f (A \cap B)\)
  by (simp add: prod.union-inter [symmetric, of A B])
  with assms show ?thesis
  by simp
qed

context linordered-semidom
begin

lemma prod-nonneg: \((\forall a \in A. 0 \leq f a) \Longrightarrow 0 \leq \prod f A\)
by (induct A rule: infinite-finite-induct) simp-all

lemma prod-pos: \((\forall a \in A. 0 < f a) \Longrightarrow 0 < \prod f A\)
by (induct A rule: infinite-finite-induct) simp-all

lemma prod-mono:
(\(\forall i. i \in A \Rightarrow 0 \leq f_i \land f_i \leq g_i\) \(\Longrightarrow \prod f A \leq \prod g A\))
by (induct A rule: infinite-finite-induct) (force intro: prod-nonneg mult-mono)+

lemma prod-mono-strict:
assumes finite A \(\forall i. i \in A \Rightarrow 0 \leq f_i \land f_i < g_i \neq \{\}\)
shows \(\prod f A < \prod g A\)
using assms
proof (induct A rule: finite-induct)
THEORY "Groups-Big"

case empty
then show ?case by simp
next
  case insert
  then show ?case by (force intro: mult-strict-mono prod-nonneg)
qed

end

lemma prod-mono2:
  fixes f :: 'a ⇒ 'b :: linordered-idom
  assumes fin: finite B
  and sub: A ⊆ B
  and nn: \( \forall b. b \in B - A \implies 1 \leq f b \)
  and A: \( \forall a. a \in A \implies 0 \leq f a \)
  shows prod f A \leq prod f B
proof
  have prod f A \leq prod f A * prod f (B - A)
    by (metis prod-ge-1 A mult-le-cancel-left1 nn not-less prod-nonneg)
  also from fin finite-subset[OF sub fin] have ... = prod f (A \cup (B - A))
    by (simp add: prod.union-disjoint del: Un-Diff-cancel)
  also from sub have A \cup (B - A) = B by blast
  finally show ?thesis.
qed

lemma less-1-prod:
  fixes f :: 'a ⇒ 'b::linordered-idom
  shows finite I \implies I \neq {} \implies (\\( \forall i. i \in I \implies 1 < f i \)) \implies 1 < prod f I
  by (induct I rule: finite-ne-induct) (auto intro: less-1-mult)
lemma less-1-prod2:
  fixes f :: 'a ⇒ 'b::linordered-idom
  assumes I: finite I i \in I \implies 1 < f i \land I. i \in I \implies 1 \leq f i
  shows I \leq prod f I
proof
  have 1 < f i * prod f (I - {i})
    using assms
    by (meson DiffD1 leI less-1-mult less-le-trans mult-le-cancel-left1 prod-ge-1)
  also have ... = prod f I
    using assms by (simp add: prod.remove)
  finally show ?thesis.
qed

lemma (in linordered-field) abs-prod: \(|\prod x\in A. f x|\) = (\prod x\in A. |f x|)
  by (induct A rule: infinite-finite-induct) (simp-all add: abs-mult)

lemma prod-eq-1-iff [simp]: finite A \implies prod f A = 1 \iff (\forall a\in A. f a = 1)
  for f :: 'a ⇒ nat
by (induct A rule: finite-induct) simp-all
lemma prod-pos-nat-iff [simp]: finite A ⇒ prod f A > 0 ←→ (∀ a∈A. f a > 0)
for f :: 'a ⇒ nat
using prod-zero-iff by (simp del: neq0-conv add: zero-less_iff_neq_zero)

lemma prod-constant [simp]: (∏ x∈ A. y) = y ^ card A
for y :: 'a::comm-monoid-mult
by (induct A rule: infinite-finite-induct) simp-all

lemma prod-power-distrib: prod f A ^ n = prod (λ x. (f x) ^ n) A
for f :: 'a ⇒ 'b::comm-semiring-1
by (induct A rule: infinite-finite-induct) (auto simp add: power-mult-distrib)

lemma power-sum: c ^ (∑ a∈A. f a) = (∏ a∈A. c ^ f a)
by (induct A rule: infinite-finite-induct) (simp-all add: power-add)

lemma prod-gen-delta:
fixes b :: 'b⇒'a::comm-monoid-mult
assumes fin: finite S
shows prod (λ k. if k = a then b k else c) S = (if a ∈ S then b a * c ^ (card S - 1) else c ^ card S)
proof
  let ?f = (λ k. if k = a then b k else c)
  show ?thesis
  proof (cases a ∈ S)
    case False
    then have ∀ k∈ S. ?f k = c by simp
    with False show ?thesis by (simp add: prod-constant)
  next
    case True
    let ?A = S - {a}
    let ?B = {a}
    from True have eq: S = ?A ∪ ?B by blast
    have disjoint: ?A ∩ ?B = {} by simp
    from fin have fin': finite ?A finite ?B by auto
    have f-A0: prod ?f ?A = prod (λ x. c) ?A
      by (rule prod.cong) auto
    from fin True have card-A: card ?A = card S - 1 by auto
    have f-A1: prod ?f ?A = c ^ card ?A
      unfolding f-A0 by (rule prod-constant)
    have prod ?f ?A = prod ?f ?B = prod ?f S
      using prod.union_disjoint[OF fin' disjoint, of ?f, unfolded eq[symmetric]]
      by simp
    with True card-A show ?thesis
      by (simp add: f-A1 field_simps cong add: prod.cong cong del: if_weak_cong)
  qed
  qed

lemma sum-image-le:
fixes $g :: 'a ⇒ 'b$::ordered-comm-monoid-add
assumes finite $I \land \forall i \in I \rightarrow 0 \leq g(f\ i)$
shows $\text{sum } g(f\ i) \leq \text{sum } (g \circ f)\ I$
using assms
proof induction
  case empty
  then show ?case by auto
next
case $(\text{insert } x\ F)$
  from insertII have $0 \leq g(f\ x)$ by (rule insert)
  hence 1: $\text{sum } g(f\ F) \leq g(f\ x) + \text{sum } g(f\ F)$ using add-increasing by blast
  have 2: $\text{sum } g(f\ F) \leq \text{sum } (g \circ f)\ F$ using insert by blast
  have $\text{sum } g(f\ \text{insert } x\ F) = \text{sum } g(\text{insert } (f\ F)\ x)$ by simp
  also have $\ldots \leq g(f\ x) + \text{sum } g(f\ F)$ by (simp add: 1 insert sum.insert-if)
  also from 2 have $\ldots \leq g(f\ x) + \text{sum } (g \circ f)\ F$ by (rule add-left-mono)
  also from insert(1, 2) have $\ldots = \text{sum } (g \circ f)(\text{insert } x\ F)$ by (simp add: sum.insert-if)
  finally show ?case .
qed

end

43 Equivalence Relations in Higher-Order Set Theory

theory Equiv-Relations imports Groups-Big begin

43.1 Equivalence relations – set version

definition equiv :: 'a set ⇒ ('a × 'a) set ⇒ bool
  where equiv $A\ r \iff \text{refl-on } A\ r \land \text{sym } r \land \text{trans } r$

lemma equivI: refl-on $A\ r \implies \text{sym } r \implies \text{trans } r \implies \text{equiv } A\ r$
bysimp (simp add: equiv-def)

lemma equivE:
  assumes equiv $A\ r$
  obtains refl-on $A\ r$ and sym $r$ and trans $r$
  using assms by (simp add: equiv-def)

Suppes, Theorem 70: $r$ is an equiv relation iff $r^{-1}\ O\ r = r$.
First half: $\text{equiv } A\ r \implies r^{-1}\ O\ r = r$.

lemma sym-trans-comp-subset: $\text{sym } r \implies \text{trans } r \implies r^{-1}\ O\ r \subseteq r$
unfolding trans-def sym-def converse-unfold by blast

lemma refl-on-comp-subset: refl-on $A\ r \implies r \subseteq r^{-1}\ O\ r$
THEORY “Equiv-Relations”

unfolding refl-on-def by blast

lemma equiv-comp-eq: equiv A r \implies r^{-1} O r = r
  apply (unfold equiv-def)
  apply clarify
  apply (rule equalityI)
  apply (iprover intro: sym-trans-comp-subset refl-on-comp-subset)+
  done

Second half.

lemma comp-equivI: r^{-1} O r = r \implies Domain r = A \implies equiv A r
  apply (unfold equiv-def refl-on-def sym-def trans-def)
  apply (erule equalityE)
  apply (subgoal-tac \forall x y. (x, y) \in r \Longrightarrow (y, x) \in r)
  apply fast
  apply fast
  done

43.2 Equivalence classes

lemma equiv-class-subset: equiv A r \Longrightarrow (a, b) \in r \Longrightarrow r'^{-1}\{a\} \subseteq r'^{-1}\{b\}
  — lemma for the next result
  unfolding equiv-def trans-def sym-def by blast

theorem equiv-class-eq: equiv A r \Longrightarrow (a, b) \in r \Longrightarrow r'^{-1}\{a\} = r'^{-1}\{b\}
  apply (assumption | rule equalityI equiv-class-subset)+
  apply (unfold equiv-def sym-def)
  apply blast
  done

lemma equiv-class-self: equiv A r \Longrightarrow a \in A \Longrightarrow a \in r'^{-1}\{a\}
  unfolding equiv-def refl-on-def by blast

lemma subset-equiv-class: equiv A r \Longrightarrow r'^{-1}\{b\} \subseteq r'^{-1}\{a\} \Longrightarrow b \in A \Longrightarrow (a, b) \in r
  — lemma for the next result
  unfolding equiv-def refl-on-def by blast

lemma eq-equiv-class: r'^{-1}\{a\} = r'^{-1}\{b\} \Longrightarrow equiv A r \Longrightarrow b \in A \Longrightarrow (a, b) \in r
  by (iprover intro: equalityD2 subset-equiv-class)

lemma equiv-class-nondisjoint: equiv A r \Longrightarrow x \in (r'^{-1}\{a\} \cap r'^{-1}\{b\}) \Longrightarrow (a, b) \in r
  unfolding equiv-def trans-def sym-def by blast

lemma equiv-type: equiv A r \Longrightarrow r \subseteq A \times A
  unfolding equiv-def refl-on-def by blast

lemma equiv-class-eq-iff: equiv A r \Longrightarrow (x, y) \in r \longleftrightarrow r'^{-1}\{x\} = r'^{-1}\{y\} \wedge x \in A
∧ y ∈ A
  by (blast intro!: equiv-class-eq dest: eq-equiv-class equiv-type)

lemma eq-equiv-class-iff: equiv A r ⇒ x ∈ A ⇒ y ∈ A ⇒ r''{x} = r''{y}
  ⨆ (x, y) ∈ r
  by (blast intro!: equiv-class-eq dest: eq-equiv-class equiv-type)

43.3 Quotients

definition quotient :: 'a set ⇒ ('a × 'a) set ⇒ 'a set set (infixl '//'/ 90)
  where A//r = (∪x ∈ A. {r''{x}}) — set of equiv classes

lemma quotientI: x ∈ A ⇒ r''{x} ∈ A//r
  unfolding quotient-def by blast

lemma quotientE: X ∈ A//r ⇒ (∀x. X = r''{x} ⇒ x ∈ A ⇒ P) ⇒ P
  unfolding quotient-def by blast

lemma Union-quotient: equiv A r ⇒ ∪(A//r) = A
  unfolding equiv-def refl-on-def quotient-def by blast

lemma quotient-disj: equiv A r ⇒ X ∈ A//r ⇒ Y ∈ A//r ⇒ X = Y ∨ X ∩ Y = {}
  apply (unfold quotient-def)
  apply clarify
  apply (rule equiv-class-eq)
  apply assumption
  apply (unfold equiv-def trans-def sym-def)
  apply blast
  done

lemma quotient-eqI:
  equiv A r ⇒ X ∈ A//r ⇒ Y ∈ A//r ⇒ x ∈ X ⇒ y ∈ Y ⇒ (x, y) ∈ r
  ⇒ X = Y
  apply (clarify elim!: quotientE)
  apply (rule equiv-class-eq)
  apply assumption
  apply (unfold equiv-def sym-def trans-def)
  apply blast
  done

lemma quotient-eq-iff:
  equiv A r ⇒ X ∈ A//r ⇒ Y ∈ A//r ⇒ x ∈ X ⇒ y ∈ Y ⇒ X = Y ↔
  (x, y) ∈ r
  apply (rule iffI)
  prefer 2
  apply (blast del: equalityI intro: quotient-eqI)
  apply (clarify elim!: quotientE)
  apply (unfold equiv-def sym-def trans-def)
  done
apply blast
done

lemma eq-equiv-class-iff2: equiv A r \iff x \in A \Rightarrow y \in A \Rightarrow \{x\} /\ r = \{y\} /\ r
by (simp add: quotient-def eq-equiv-class-iff)

lemma quotient-empty [simp]: \{\} /\ r = \{}
by (simp add: quotient-def)

lemma quotient-is-empty [iff]: A /\ r = {} \iff A = {}
by (simp add: quotient-def)

lemma quotient-is-empty2 [iff]: \{\} = A /\ r \iff A = {}
by (simp add: quotient-def)

lemma singleton-quotient: \{x\} /\ r = \{r '' \{x\}\}
by (simp add: quotient-def)

lemma quotient-diff1: inj-on (\lambda a. \{a\} /\ r) A = \Rightarrow a \in A = \Rightarrow (A - \{a\}) /\ r = A /\ r - \{a\} /\ r
by (auto simp: quotient-def inj-on-def)

43.4 Refinement of one equivalence relation WRT another

lemma refines-equiv-class-eq: R \subseteq S \Rightarrow equiv A R \Rightarrow equiv A S \Rightarrow R''(S''\{a\})
= S''\{a\}
by (auto simp: equiv-class-eq-iff)

lemma refines-equiv-class-eq2: R \subseteq S \Rightarrow equiv A R \Rightarrow equiv A S \Rightarrow S''(R''\{a\})
= S''\{a\}
by (auto simp: equiv-class-eq-iff)

lemma refines-equiv-image-eq: R \subseteq S \Rightarrow equiv A R \Rightarrow equiv A S \Rightarrow (\lambda X. S''X) ' (A /\ R) = A /\ S
by (auto simp: quotient-def image-UN refines-equiv-class-eq2)

lemma finite-refines-finite:
finite (A /\ R) \Rightarrow R \subseteq S \Rightarrow equiv A R \Rightarrow equiv A S \Rightarrow finite (A /\ S)
by (erule finite-surj [where f = \lambda X. S''X]) (simp add: refines-equiv-image-eq)

lemma finite-refines-card-le:
finite (A /\ R) \Rightarrow R \subseteq S \Rightarrow equiv A R \Rightarrow equiv A S \Rightarrow card (A /\ S) \leq card (A /\ R)
by (subst refines-equiv-image-eq [of R S A, symmetric])
(auto simp: card-image-le [where f = \lambda X. S''X])
43.5 Defining unary operations upon equivalence classes

A congruence-preserving function.

\[ \text{definition } \text{congruent} :: (\prime \ a \times \prime \ a) \text{ set } \Rightarrow (\prime \ a \Rightarrow \prime \ b \Rightarrow \text{bool}) \]

where \[ \text{congruent } r \ f \longleftrightarrow (\forall (y, z) \in r. \ f y = f z) \]

\[ \text{lemma } \text{congruentI} : (\forall y z. (y, z) \in r \Rightarrow f y = f z) \Rightarrow \text{congruent } r \ f \]

by (auto simp add: congruent-def)

\[ \text{lemma } \text{congruentD} : \text{congruent } r \ f = \Rightarrow (y, z) \in r \Rightarrow f y = f z \]

by (auto simp add: congruent-def)

\[ \text{abbreviation } \text{RESPECTS} :: (\prime \ a \Rightarrow \prime \ b) \Rightarrow (\prime \ a \times \prime \ a) \text{ set } \Rightarrow \text{bool} \]

\[ \text{where } f \text{ respects } r \equiv \text{congruent } r \ f \]

\[ \text{lemma } \text{UN-constant-eq} : a \in A \Rightarrow \forall y \in A. f y = c \Rightarrow (\bigcup y \in A. f y) = c \]

— lemma required to prove \( \text{UN-equiv-class} \)

by auto

\[ \text{lemma } \text{UN-equiv-class} : \text{equiv } A r \Rightarrow f \text{ respects } r \Rightarrow a \in A \Rightarrow (\bigcup x \in r``\{a\}. f x) = f a \]

— Conversion rule

apply (rule equiv-class-self [THEN UN-constant-eq])

apply assumption

apply assumption

apply (unfold equiv-def congruent-def sym-def)

apply (blast del: equalityI)

done

\[ \text{lemma } \text{UN-equiv-class-type} : \]

\[ \text{equiv } A r \Rightarrow f \text{ respects } r \Rightarrow X \in A//r \Rightarrow (\bigcup x \in r``\{a\}. f x) \in X. f x) \in B \]

apply (unfold quotient-def)

apply clarify

apply (subst UN-equiv-class)

apply auto

done

Sufficient conditions for injectiveness. Could weaken premises! major premise could be an inclusion; bcong could be \( \forall y. y \in A \Rightarrow f y \in B \).

\[ \text{lemma } \text{UN-equiv-class-inject} : \]

\[ \text{equiv } A r \Rightarrow f \text{ respects } r \Rightarrow \]

\[ (\bigcup x \in X. f x) = (\bigcup y \in Y. f y) \Rightarrow X \in A//r \Rightarrow Y \in A//r \]

\[ \Rightarrow (\forall x y. x \in A \Rightarrow y \in A \Rightarrow f x = f y \Rightarrow (x, y) \in r) \]

\[ \Rightarrow X = Y \]

apply (unfold quotient-def)

apply clarify
apply (rule equiv-class-eq)
apply assumption
apply (subgoal-tac f x = f xa)
apply blast
apply (erule box-equals)
apply (assumption | rule UN-equiv-class)+
done

43.6 Defining binary operations upon equivalence classes

A congruence-preserving function of two arguments.

definition congruent2 :: ('a × 'a) set ⇒ ('b × 'b) set ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ bool
  where congruent2 r1 r2 f ←→ (∀ (y1 z1) ∈ r1. ∀ (y2 z2) ∈ r2. f y1 y2 = f z1 z2)

lemma congruent2I':
  assumes (∀ y1 z1. ∀ y2 z2. (y1, z1) ∈ r1 ⇒ (y2, z2) ∈ r2 ⇒ f y1 y2 = f z1 z2
  shows congruent2 r1 r2 f
using assms by (auto simp add: congruent2-def)

lemma congruent2D: congruent2 r1 r2 f ⇒ (y1, z1) ∈ r1 ⇒ (y2, z2) ∈ r2 ⇒ f y1 y2 = f z1 z2
  by (auto simp add: congruent2-def)

Abbreviation for the common case where the relations are identical.

abbreviation RESPECTS2:: ('a ⇒ 'a ⇒ 'b) ⇒ ('a × 'a) set ⇒ bool (infix respects2 80)
  where f respects2 r ≡ congruent2 r r f

lemma congruent2-implies-congruent:
equiv A r1 =⇒ congruent2 r1 r2 f =⇒ a ∈ A =⇒ congruent2 f a
  unfolding congruent-def congruent2-def equiv-def refl-on-def by blast

lemma congruent2-implies-congruent-UN:
equiv A1 r1 =⇒ equiv A2 r2 =⇒ congruent2 r1 r2 f =⇒ a ∈ A2 =⇒
  congruent (λx1. ∪ x2 ∈ r2''{a}. f x1 x2)
apply (unfold congruent-def)
apply clarify
apply (rule equiv-type [THEN subsetD, THEN SigmaE2], assumption+)  
apply (simp add: UN-equiv-class congruent2-implies-congruent)
apply (unfold congruent2-def equiv-def refl-on-def)
apply (blast del: equalityI)
done

lemma UN-equiv-class2:
equiv A1 r1 =⇒ equiv A2 r2 =⇒ congruent2 r1 r2 f =⇒ a1 ∈ A1 =⇒ a2 ∈ A2
  =⇒ (∪ x1 ∈ r1''{a1}. ∪ x2 ∈ r2''{a2}. f x1 x2) = f a1 a2
THEORY “Equiv-Relations”

by (simp add: UN-equiv-class congruent2-implies-congruent congruent2-implies-congruent-UN)

lemma UN-equiv-class-type2:
equiv A1 r1 ⇒equiv A2 r2 ⇒congruent2 r1 r2 f
⇒ X1 ∈ A1//r1 ⇒ X2 ∈ A2//r2
⇒ (∀x1 x2. x1 ∈ A1 ⇒ x2 ∈ A2 ⇒ f x1 x2 ∈ B)
⇒ (∪x1 ∈ X1. ∪x2 ∈ X2. f x1 x2) ∈ B
apply (unfold quotient-def)
apply clarify
apply (blast intro: UN-equiv-class-type congruent2-implies-congruent-UN congruent2-implies-congruent quotientI)
done

lemma UN-UN-split-split-eq:
(∪(x1, x2) ∈ X. ∪(y1, y2) ∈ Y. A x1 x2 y1 y2) =
(∪x ∈ X. ∪y ∈ Y. (λ(x1, x2). (λ(y1, y2). A x1 x2 y1 y2) y) x)
— Allows a natural expression of binary operators,
— without explicit calls to split
by auto

lemma congruent2I:
equiv A1 r1 ⇒equiv A2 r2
⇒ (∀y z w. w ∈ A2 ⇒ (y,z) ∈ r1 ⇒ f y w = f z w)
⇒ (∀y z w. w ∈ A1 ⇒ (y,z) ∈ r2 ⇒ f w y = f w z)
⇒ congruent2 r1 r2 f
— Suggested by John Harrison – the two subproofs may be
— much simpler than the direct proof.
apply (unfold congruent2-def equiv-def refl-on-def)
apply clarify
apply (blast intro: trans)
done

lemma congruent2-commuteI:
assumes equivA: equiv A r
and commute: ∀y z. y ∈ A ⇒ z ∈ A ⇒ f y z = f z y
and congt: ∀y z w. w ∈ A ⇒ (y,z) ∈ r ⇒ f w y = f w z
shows f respects2 r
apply (rule congruent2I [OF equivA equivA])
apply (rule commute [THEN trans])
apply (rule-tac [3] commute [THEN trans, symmetric])
apply (rule-tac [5] sym)
apply (rule congt | assumption |
erule equivA [THEN equiv-type, THEN subsetD, THEN SigmaE2])+
done

43.7 Quotients and finiteness

Suggested by Florian Kammler

lemma finite-quotient: finite A ⇒ r ⊆ A × A ⇒ finite (A//r)
  apply (rule finite-subset)
  apply (erule-tac [2] finite-Pow-iff [THEN iffD2])
  apply (unfold quotient-def)
  apply blast
  done

lemma finite-equiv-class: finite A =⇒ r ⊆ A × A =⇒ X ∈ A//r =⇒ finite X
  apply (unfold quotient-def)
  apply (rule finite-subset)
  prefer 2 apply assumption
  apply blast
  done

lemma equiv-imp-dvd-card: finite A =⇒ equiv A r =⇒ ∀X ∈ A//r. k dvd card X
  apply (rule Union-quotient [THEN subst [where P=λA. k dvd card A]])
  apply assumption
  apply (rule dvd-partition)
  prefer 3 apply (blast dest: quotient-disj)
  apply (simp-all add: Union-quotient equiv-type)
  done

lemma card-quotient-disjoint: finite A =⇒ inj-on (λx. {x} // r) A =⇒ card (A//r) = card A
  apply (simp add:quotient-def)
  apply (subst card-UN-disjoint)
  apply assumption
  apply simp
  apply (fastforce simp add:inj-on-def)
  apply simp
  done

43.8 Projection

definition proj :: ('b × 'a) set ⇒ 'b ⇒ 'a set
  where proj r x = r "\{x\}

lemma proj-preserves: x ∈ A =⇒ proj r x ∈ A//r
  unfolding proj-def by (rule quotientI)

lemma proj-in-iff:
  assumes equiv A r
  shows proj r x ∈ A//r ⇐⇒ x ∈ A
    (is ?lhs ⇐⇒ ?rhs)
  proof
    assume ?rhs
    then show ?lhs by (simp add: proj-preserves)
  next
THEORY “Equiv-Relations”

assume ?lhs
then show ?rhs
  unfolding proj-def quotient-def
proof clarsimp
  fix y
  assume y: y ∈ A and r " {x} = r " {y}
  moreover have y ∈ r " {y}
    using assms unfolding equiv-def refl-on-def by blast
  ultimately have (x, y) ∈ r by blast
  then show x ∈ A
    using assms unfolding equiv-def refl-on-def by blast
qed
qed

lemma proj-iff: equiv A r → {x, y} ⊆ A ≡ proj r x = proj r y ≡ (x, y) ∈ r
  by (simp add: proj-def eq-equiv-class-iff)

lemma proj-image: proj r ' A = A//r
  unfolding proj-def[abs-def] quotient-def by blast

lemma in-quotient-imp-non-empty: equiv A r → X ∈ A//r → X ≠ {}
  unfolding quotient-def using equiv-class-self by fast

lemma in-quotient-imp-in-rel: equiv A r → X ∈ A//r → {x, y} ⊆ X → (x, y) ∈ r
  using quotient-eq-iff[THEN iffD1] by fastforce

lemma in-quotient-imp-closed: equiv A r → X ∈ A//r → x ∈ X → (x, y) ∈ r → y ∈ X
  unfolding quotient-def equiv-def trans-def by blast

lemma in-quotient-imp-subset: equiv A r → X ∈ A//r → X ⊆ A
  using in-quotient-imp-in-rel equiv-type by fastforce

43.9 Equivalence relations – predicate version

Partial equivalences.

definition part-equivp :: ('a ⇒ 'a ⇒ bool) ⇒ bool
  where part-equivp R ≡ (∃ x. R x x) ∧ (∀ x y. R x y ≡ R x x ∧ R y y ∧ R x = R y)
  — John-Harrison-style characterization

lemma part-equivpI: (∃ x. R x x) ⇒ symp R ⇒ transp R ⇒ part-equivp R
  by (auto simp add: part-equivp-def) (auto elim: sympE transpE)

lemma part-equivpE:
  assumes part-equivp R
obtains $x$ where $R x x$ and $\text{symp } R$ and $\text{transp } R$

proof 
from assms have 1: $\exists x. R x x$
and 2: $\forall x y. R x y \iff R x x \land R y y \land R x = R y$
unfolding part-equivp-def by blast+
from 1 obtain $x$ where $R x x$ ..
moreover have $\text{symp } R$
proof (rule sympI)
fix $x y$
assume $R x y$
with 2 [of $x y$] show $R y x$ by auto
qed
moreover have $\text{transp } R$
proof (rule transpI)
fix $x y z$
assume $R x y$ and $R y z$
with 2 [of $x y$] [of $y z$] show $R x z$ by auto
qed
ultimately show thesis by (rule that)
qed

lemma part-equivp-refl-symp-transp: $\text{part-equivp } R \iff (\exists x. R x x) \land \text{symp } R \land \text{transp } R$
by (auto intro: part-equivpI elim: part-equivpE)

lemma part-equivp-symp: $\text{part-equivp } R \Rightarrow R x y \Rightarrow R y x$
by (erule part-equivpE, erule sympE)

lemma part-equivp-transp: $\text{part-equivp } R \Rightarrow R x y \Rightarrow R y z \Rightarrow R x z$
by (erule part-equivpE, erule transpE)

lemma part-equivp-typedef: $\text{part-equivp } R \Rightarrow \exists d. d \in \{ c. \exists x. R x x \land c = \text{Collect } (R x) \}$
by (auto elim: part-equivpE)

Total equivalences.

definition equivp :: $\forall 'a \Rightarrow 'a \Rightarrow \text{bool} \Rightarrow \text{bool}$
where $\text{equivp } R \iff (\forall x y. R x y = (R x = R y))$ — John-Harrison-style characterization

lemma equivpI: $\text{reflp } R \Rightarrow \text{symp } R \Rightarrow \text{transp } R \Rightarrow \text{equivp } R$
by (auto elim: reflpE sympE transpE simp add: equivp-def)

lemma equivpE:
assumes $\text{equivp } R$
obtains $\text{reflp } R$ and $\text{symp } R$ and $\text{transp } R$
using assms by (auto intro!: that reflpI sympI transpI simp add: equivp-def)

lemma equivp-implies-part-equivp: $\text{equivp } R \Rightarrow \text{part-equivp } R$
THEORY "Equiv-Relations"

lemma equivp-equiv: equiv UNIV A ↔ equivp (λx y. (x, y) ∈ A)
  by (auto intro!: equivI equivpI to-set elim!: equivE equivpE)

lemma equivp-reflp-symp-transp: equivp R ↔ reflp R ∧ symp R ∧ transp R
  by (auto intro!: equivpI elim!: equivE equivpE)

lemma identity-equivp: equivp (=)
  by (auto intro!: equivpI reflpI sympI transpI)

lemma equivp-reflp: equivp R =⇒ R x x
  by (erule equivpE, erule reflpE)

lemma equivp-symp: equivp R =⇒ R x y =⇒ R y x
  by (erule equivpE, erule sympE)

lemma equivp-transp: equivp R =⇒ R x y =⇒ R y z =⇒ R x z
  by (erule equivpE, erule transpE)

lemma equivp-rtranclp: symp r =⇒ equivp r**
  by (intro equivpI reflpI sympI transpI)(auto dest: sympD[OF symp-rtranclp])

lemmas equivp-rtranclp-symclp [simp] = equivp-rtranclp[OF symp-symclp]

lemma equivp-vimage2p: equivp R =⇒ equivp (vimage2p f f R)
  by (auto simp add: equivp-def vimage2p-def dest: fun-cong)

lemma equivp-imp-transp: equivp R =⇒ transp R
  by (simp add: equivp-reflp-symp-transp)

43.10 Equivalence closure

definition equivclp :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool where
  equivclp r = (symclp r)**

lemma transp-equivclp [simp]: transp (equivclp r)
  by (simp add: equivclp-def)

lemma reflp-equivclp [simp]: reflp (equivclp r)
  by (simp add: equivclp-def)

lemma symp-equivclp [simp]: symp (equivclp r)
  by (simp add: equivclp-def)

lemma equivp-evquivclp [simp]: equivp (equivclp r)
  by (simp add: equivpI)

lemma tranclp-equivclp [simp]: (equivclp r)** = equivclp r
by (simp add: equivclp-def)

**lemma** rtranclp-equivclp [simp]: (equivclp r)** = equivclp r
  by (simp add: equivclp-def)

**lemma** symclp-equivclp [simp]: symclp (equivclp r) = equivclp r
  by (simp add: equivclp-def symp-symclp-eq)

**lemma** equivclp-symclp [simp]: equivclp (symclp r) = equivclp r
  by (simp add: equivclp-def)

**lemma** equivclp-conversep [simp]: equivclp (conversep r) = equivclp r
  by (simp add: equivclp-def)

**lemma** equivclp-sym [simp]: equivclp r x y =⇒ equivclp r y x
  by (rule sympD [OF symp-equivclp])

**lemma** equivclp-OO-equivclp-le-equivclp: equivclp r OO equivclp r ≤ equivclp r
  by (rule transp-relcompp-less-OF transp-equivclp)

**lemma** rtranlcp-le-equivclp: r∗∗ ≤ equivclp r
  unfolding equivclp-def by (rule rtranclp-mono) (simp add: symclp-pointfree)

**lemma** rtranclp-conversep-le-equivclp: r−1−1∗∗ ≤ equivclp r
  unfolding equivclp-def by (rule rtranclp-mono) (simp add: symclp-pointfree)

**lemma** symclp-rtranclp-le-equivclp: symclp r∗∗ ≤ equivclp r
  unfolding symclp-pointfree
  by (rule le-supI) (simp-all add: rtranclp-conversep[symmetric] rtranclp-OO-equivclp rtranclp-conversep-le-equivclp)

**lemma** converse-equivclp-induct [consumes 1, case-names base step, induct pred: equivclp]:
  assumes a: equivclp r a b
  and cases: P a ∨ y z. equivclp r a y =⇒ r y z =⇒ r z y =⇒ P y =⇒ P z
  shows P b
  using a unfolding equivclp-def
  by (induction rule: rtranclp-induct; fold equivclp-def; blast intro: cases elim: symclpE)

**lemma** converse-equivclp-induct [consumes 1, case-names base step]:
  assumes major: equivclp r a b
  and cases: P b ∨ y z. r y z =⇒ r z y =⇒ equivclp r z b =⇒ P z =⇒ P y
  shows P a
  using major unfolding equivclp-def
by (induction rule: converse-rtranclp-induct; fold equivclp-def; blast intro: cases elim: symclpE)

lemma equivclp-refl [simp]: equivclp r x x
  by (rule reflpD[OF reflp-equivclp])

lemma r-into-equivclp [intro]: r x y =⇒ equivclp r x y
  unfolding equivclp-def by (blast intro: symclpI)

lemma converse-r-into-equivclp [intro]: r y x =⇒ equivclp r x y
  unfolding equivclp-def by (blast intro: symclpI)

lemma rtranclp-into-equivclp: r∗∗ x y =⇒ equivclp r x y
  using rtranclp-le-equivclp[of r] by blast

lemma converse-rtranclp-into-equivclp: r∗∗ y x =⇒ equivclp r x y
  by (blast intro: equivclp-sym rtranclp-into-equivclp)

lemma equivclp-into-equivclp: [ equivclp r a b; r b c ∨ r c b ] =⇒ equivclp r a c
  unfolding equivclp-def by (erule rtranclp.rtrancl-into-rtrancl)(auto intro: symclpI)

lemma equivclp-trans [trans]: [ equivclp r a b; equivclp r b c ] =⇒ equivclp r a c
  using equivclp-OO-equivclp-le-equivclp[of r] by blast

hide-const (open) proj

end

44 Lifting package

theory Lifting
imports Equiv-Relations Transfer
keywords
  parametric and
  print-quot-maps print-quotients :: diag and
  lift-definition :: thy-goal-defn and
  setup-lifting lifting-forget lifting-update :: thy-decl
begin

44.1 Function map

context includes lifting-syntax
begin

lemma map-fun-id:
  (id −−→ id) = id
  by (simp add: fun-eq-iff)

end
44.2 Quotient Predicate

definition
Quotient R Abs Rep T \iff
(\forall a. \text{Abs (Rep a)} = a) \land
(\forall a. R (\text{Rep a}) (\text{Rep a})) \land
(\forall r s. R r s \iff R r r \land R s s \land \text{Abs r} = \text{Abs s}) \land
T = (\lambda x y. R x x \land \text{Abs x} = y)

lemma QuotientI:
assumes \land a. \text{Abs (Rep a)} = a
\land \land a. R (\text{Rep a}) (\text{Rep a})
\land \land r s. R r s \iff R r r \land R s s \land \text{Abs r} = \text{Abs s}
\land T = (\lambda x y. R x x \land \text{Abs x} = y)
shows Quotient R Abs Rep T
using assms unfolding Quotient-def by blast

context
fixes R Abs Rep T
assumes a: Quotient R Abs Rep T
begin

lemma Quotient-abs-rep: \text{Abs (Rep a)} = a
using a unfolding Quotient-def by simp

lemma Quotient-rep-reflp: R (\text{Rep a}) (\text{Rep a})
using a unfolding Quotient-def by blast

lemma Quotient-rel:
R r r \land R s s \land \text{Abs r} = \text{Abs s} \iff R r s — orientation does not loop on
rewriting
using a unfolding Quotient-def
by blast

lemma Quotient-cr-rel: T = (\lambda x y. R x x \land \text{Abs x} = y)
using a unfolding Quotient-def
by blast

lemma Quotient-refl1: R r s \implies R r r
using a unfolding Quotient-def
by fast

lemma Quotient-refl2: R r s \implies R s s
using a unfolding Quotient-def
by fast

lemma Quotient-rel-rep: R (\text{Rep a}) (\text{Rep b)} \iff a = b
using a unfolding Quotient-def
by metis

lemma Quotient-rep-abs: R r r \rightarrow R (Rep (Abs r)) r
  using a unfolding Quotient-def
  by blast

lemma Quotient-rep-abs-eq: R t t \rightarrow R \leq (=) \rightarrow Rep (Abs t) = t
  using a unfolding Quotient-def
  by blast

lemma Quotient-rep-abs-fold-unmap:
  assumes x' \equiv Abs x and R x x and Rep x' \equiv Rep' x'
  shows R (Rep' x') x
proof
  have R (Rep x') x using assms(1-2) Quotient-rep-abs by auto
  then show ?thesis using assms(3) by simp
qed

lemma Quotient-Rep-eq:
  assumes x' \equiv Abs x
  shows Rep x' \equiv Rep x'
  by simp

lemma Quotient-rel-abs: R r s \rightarrow Abs r = Abs s
  using a unfolding Quotient-def
  by blast

lemma Quotient-rel-abs2:
  assumes R (Rep x) y
  shows x = Abs y
proof
  from assms have Abs (Rep x) = Abs y by (auto intro: Quotient-rel-abs)
  then show ?thesis using assms(1) by (simp add: Quotient-abs-rep)
qed

lemma Quotient-symp: symp R
  using a unfolding Quotient-def using sympI by (metis (full-types))

lemma Quotient-transp: transp R
  using a unfolding Quotient-def using transpI by (metis (full-types))

lemma Quotient-part-equivp: part-equivp R
  by (metis Quotient-rep-reflp Quotient-symp Quotient-transp part-equivpI)
end

lemma identity-quotient: Quotient (=) id id (=)
  unfolding Quotient-def by simp

TODO: Use one of these alternatives as the real definition.
lemma Quotient-alt-def:
Quotient R Abs Rep T ⟷
(∀ a b. T a b ⟹ Abs a = b) ∧
(∀ b. T (Rep b) b) ∧
(∀ x y. R x y ⟹ T x (Abs x) ∧ T y (Abs y) ∧ Abs x = Abs y)
apply safe
apply (simp (no-asmp-use) only: Quotient-def, fast)
apply (simp (no-asmp-use) only: Quotient-def, fast)
apply (simp (no-asmp-use) only: Quotient-def, fast)
apply (simp (no-asmp-use) only: Quotient-def, fast)
apply (simp (no-asmp-use) only: Quotient-def, fast)
apply (rule QuotientI)
apply simp
apply metis
apply simp
apply (rule ext, rule ext, metis)
done

lemma Quotient-alt-def2:
Quotient R Abs Rep T ⟷
(∀ a b. T a b ⟹ Abs a = b) ∧
(∀ b. T (Rep b) b) ∧
(∀ x y. R x y ⟹ T x (Abs y) ∧ T y (Abs x))
unfolding Quotient-alt-def by (safe, metis+)

lemma Quotient-alt-def3:
Quotient R Abs Rep T ⟷
(∀ a b. T a b ⟹ Abs a = b) ∧ (∀ b. T (Rep b) b) ∧
(∀ x y. R x y ⟹ (∃ z. T x z ∧ T y z))
unfolding Quotient-alt-def2 by (safe, metis+)

lemma Quotient-alt-def4:
Quotient R Abs Rep T ⟷
(∀ a b. T a b ⟹ Abs a = b) ∧ (∀ b. T (Rep b) b) ∧ R = T OO conversep T
unfolding Quotient-alt-def3 fun-eq-iff by auto

lemma Quotient-alt-def5:
Quotient R Abs Rep T ⟷
T ≤ BNF-Def.Grp UNIV Abs ∧ BNF-Def.Grp UNIV Rep ≤ T⁻¹⁻¹ ∧ R = T OO T⁻¹⁻¹
unfolding Quotient-alt-def4 Grp-def by blast

lemma fun-quotient:
  assumes 1: Quotient R1 abs1 rep1 T1
  assumes 2: Quotient R2 abs2 rep2 T2
  shows Quotient (R1 ===> R2) (rep1 ===> abs2) (abs1 ===> rep2) (T1 ===> T2)
  using assms unfolding Quotient-alt-def2
unfolding rel-fun-def fun-eq-iff map-fun-apply by (safe, metis+)

lemma apply-rsp:
  fixes f g :: 'a ⇒ 'c
  assumes q: Quotient R1 Abs1 Rep1 T1
  and a: (R1 ===> R2) f g R1 x y
  shows R2 (f x) (g y)
  using a by (auto elim: rel-funE)

lemma apply-rsp':
  assumes a: (R1 ===> R2) f g R1 x y
  shows R2 (f x) (g y)
  using a by (auto elim: rel-funE)

lemma apply-rsp'':
  assumes Quotient R Abs Rep T
  and (R === S) f f
  shows S (f (Rep x)) (f (Rep x))
  proof
    from assms(1) have R (Rep x) (Rep x) by (rule Quotient-rep-reflp)
    then show ?thesis using assms(2) by (auto intro: apply-rsp')
  qed

44.3 Quotient composition

lemma Quotient-compose:
  assumes 1: Quotient R1 Abs1 Rep1 T1
  assumes 2: Quotient R2 Abs2 Rep2 T2
  shows Quotient (T1 OO R2 OO conversep T1) (Abs2 ◦ Abs1) (Rep1 ◦ Rep2)
           (T1 OO T2)
  using assms unfolding Quotient-alt-def4 by fastforce

lemma equivp-reflp2:
  equivp R ⇒ reflp R
  by (erule equivpE)

44.4 Respects predicate

definition Respects :: ('a ⇒ 'a ⇒ bool) ⇒ 'a set
  where Respects R = {x. R x x}

lemma in-respects: x ∈ Respects R ←→ R x x
  unfolding Respects-def by simp

lemma UNIV-typedef-to-Quotient:
  assumes type-definition Rep Abs UNIV
  and T-def: T ≡ (λx y. x = Rep y)
  shows Quotient (=) Abs Rep T
  proof –
interpret type-definition Rep Abs UNIV by fact
from Abs-inject Rep-inverse Abs-inverse T-def show ?thesis
  by (fastforce intro!: QuotientI fun-eq-iff)
qed

lemma UNIV-typedef-to-equivp:
  fixes Abs :: 'a ⇒ 'b
  and Rep :: 'b ⇒ 'a
  assumes type-definition Rep Abs (UNIV::'a set)
  shows equivp ((=) ::'a⇒'a⇒bool)
by (rule identity-equivp)

lemma typedef-to-Quotient:
  assumes type-definition Rep Abs S
  and T-def: T ≡ (λx y. x = Rep y)
  shows Quotient (eq-onp (λx. x ∈ S)) Abs Rep T
proof -
  interpret type-definition Rep Abs S by fact
  from Rep Abs-inject Rep-inverse Abs-inverse T-def show ?thesis
    by (auto intro!: QuotientI simp: eq-onp-def fun-eq-iff)
qed

lemma typedef-to-part-equivp:
  assumes type-definition Rep Abs {x. P x}
  shows part-equivp (eq-onp (λx. x ∈ S))
proof (intro part-equivpI)
  interpret type-definition Rep Abs S by fact
  show ∃x. eq-onp (λx. x ∈ S) x x using Rep by (auto simp: eq-onp-def)
next
  show symp (eq-onp (λx. x ∈ S)) by (auto intro: sympI simp: eq-onp-def)
next
  show transp (eq-onp (λx. x ∈ S)) by (auto intro: transpI simp: eq-onp-def)
qed

lemma open-typedef-to-Quotient:
  assumes type-definition Rep Abs {x. P x}
  and T-def: T ≡ (λx y. x = Rep y)
  shows Quotient (eq-onp P) Abs Rep T
  using typedef-to-Quotient [OF assms] by simp

lemma open-typedef-to-part-equivp:
  assumes type-definition Rep Abs {x. P x}
  shows part-equivp (eq-onp P)
  using typedef-to-part-equivp [OF assms] by simp

lemma type-definition-Quotient-not-empty: Quotient (eq-onp P) Abs Rep T ⊢
  ∃x. P x
  unfolding eq-onp-def by (drule Quotient-reflp) blast
Generating transfer rules for quotients.

context  
fixes $R$, $Abs$, $Rep$, $T$  
assumes 1: Quotient $R$ $Abs$ $Rep$ $T$
begin

lemma Quotient-right-unique: right-unique $T$
  using 1 unfolding Quotient-alt-def right-unique-def by metis

lemma Quotient-right-total: right-total $T$
  using 1 unfolding Quotient-alt-def right-total-def by metis

lemma Quotient-rel-eq-transfer: $(T =\rightarrow T =\rightarrow (=))$ $R$ $(=)$
  using 1 unfolding Quotient-alt-def rel-fun-def by simp

lemma Quotient-abs-induct:
  assumes $\forall y. R y y =\rightarrow P (Abs y)$ shows $P x$
  using 1 assms unfolding Quotient-def by metis
end

Generating transfer rules for total quotients.

category  
fixes $R$, $Abs$, $Rep$, $T$  
assumes 1: Quotient $R$ $Abs$ $Rep$ $T$ and 2: reflp $R$
begin

lemma Quotient-left-total: left-total $T$
  using 1 2 unfolding Quotient-alt-def left-total-def reflp-def by auto

lemma Quotient-bi-total: bi-total $T$
  using 1 2 unfolding Quotient-alt-def bi-total-def reflp-def by auto

lemma Quotient-id-abs-transfer: $(=\rightarrow (\lambda x. x))$ $Abs$
  using 1 2 unfolding Quotient-alt-def reflp-def rel-fun-def by simp

lemma Quotient-total-abs-induct: $(\forall y. P (Abs y)) =\rightarrow P x$
  using 1 2 unfolding Quotient-alt-def reflp-def by metis

lemma Quotient-total-abs-eq-iff: $Abs x = Abs y \iff R x y$
  using Quotient-rel [OF 1] 2 unfolding reflp-def by simp
end

Generating transfer rules for a type defined with typedef.
context
fixes Rep Abs A T
assumes type: type-definition Rep Abs A
assumes T-def: \( T \equiv (\lambda (x::'a) (y::'b). x = \text{Rep} y) \)
begin

lemma typedef-left-unique: left-unique T
unfolding left-unique-def T-def
by (simp add: type-definition Rep-inject [OF type])

lemma typedef-bi-unique: bi-unique T
unfolding bi-unique-def T-def
by (simp add: type-definition Rep-inject [OF type])

lemma typedef-right-unique: right-unique T
using T-def type Quotient-right-unique typedef-to-Quotient
by blast

lemma typedef-right-total: right-total T
using T-def type Quotient-right-total typedef-to-Quotient
by blast

lemma typedef-rep-transfer: \( (T \equiv \rightarrow (\equiv)) (\lambda x. x) \text{Rep} \)
unfolding rel-fun-def T-def by simp

end

Generating the correspondence rule for a constant defined with lift-definition.

lemma Quotient-to-transfer:
assumes Quotient R Abs Rep T and R c c and c' \( \equiv \) Abs c
shows T c c'
using assms by (auto dest: Quotient-cr-rel)

Proving reflexivity

lemma Quotient-to-left-total:
assumes q: Quotient R Abs Rep T
and r-R: reflp R
shows left-total T
using r-R Quotient-cr-rel[OF q] unfolding left-total-def by (auto elim: reflpE)

lemma Quotient-composition-ge-eq:
assumes left-total T
assumes R \( \geq \) (=)
shows \( (T \circ R \circ T^{-1}) \geq (=) \)
using assms unfolding left-total-def by fast

lemma Quotient-composition-le-eq:
assumes left-unique $T$
assumes $R \leq (=)$
shows $(T \OO R \OO T^{-1}-1) \leq (=)$
using assms unfolding left-unique-def by blast

lemma eq-onp-le-eq:
\[ eq-onp P \leq (=) \] unfolding eq-onp-def by blast

lemma reflp-ge-eq:
\[ reflp R \Rightarrow R \geq (=) \] unfolding reflp-def by blast

Proving a parametrized correspondence relation

definition $POS :: \left( \forall a \Rightarrow \forall b \Rightarrow bool \right) \Rightarrow bool$ where
\[ POS A B \equiv A \leq B \]
definition $NEG :: \left( \forall a \Rightarrow \forall b \Rightarrow bool \right) \Rightarrow bool$ where
\[ NEG A B \equiv B \leq A \]

lemma pos-OO-eq:
\[ \text{shows } POS (A \OO (=)) A \]
unfolding POS-def OO-def by blast

lemma pos-eq-OO:
\[ \text{shows } POS ((=) \OO A) A \]
unfolding POS-def OO-def by blast

lemma pos-eq-OO:
\[ \text{shows } POS ((=) \OO A) A \]
unfolding POS-def OO-def by blast

lemma neg-OO-eq:
\[ \text{shows } NEG (A \OO (=)) A \]
unfolding NEG-def OO-def by auto

lemma neg-eq-OO:
\[ \text{shows } NEG ((=) \OO A) A \]
unfolding NEG-def OO-def by blast

lemma POS-trans:
\[ \text{assumes } POS A B \]
\[ \text{assumes } POS B C \]
\[ \text{shows } POS A C \]
using assms unfolding POS-def by auto

lemma NEG-trans:
\[ \text{assumes } NEG A B \]
\[ \text{assumes } NEG B C \]
\[ \text{shows } NEG A C \]
using assms unfolding NEG-def by auto

lemma POS-NEG:
\[ POS A B \equiv NEG B A \]
unfolding POS-def NEG-def by auto
**THEORY “Lifting”**

**lemma NEG-POS:**
\[ \neg A \equiv \neg B \equiv \neg B \equiv \neg A \]

**unfolding** POS-def NEG-def **by** auto

**lemma POS-per-rule:**
- **assumes** \( \text{POS} (A OO B) \ C \)
- **shows** \( \text{POS} (A OO B OO X) \ (C OO X) \)
- **using** assms **unfolding** POS-def OO-def **by** blast

**lemma NEG-per-rule:**
- **assumes** \( \text{NEG} (A OO B) \ C \)
- **shows** \( \text{NEG} (A OO B OO X) \ (C OO X) \)
- **using** assms **unfolding** NEG-def OO-def **by** blast

**lemma POS-apply:**
- **assumes** \( \text{POS} R R' \)
- **assumes** \( R f g \)
- **shows** \( R' f g \)
- **using** assms **unfolding** POS-def **by** auto

Proving a parametrized correspondence relation

**lemma fun-mono:**
- **assumes** \( A \geq C \)
- **assumes** \( B \leq D \)
- **shows** \( (A \equiv \rightarrow B) \leq (C \equiv \rightarrow D) \)
- **using** assms **unfolding** rel-fun-def **by** blast

**lemma pos-fun-distr:**
\[ ((R \equiv \rightarrow S) \ OO (R' \equiv \rightarrow S')) \leq ((R \ OO R') \equiv \rightarrow (S \ OO S')) \]

**unfolding** OO-def rel-fun-def **by** blast

**lemma functional-relation:**
- **right-unique** \( R \implies \leftarrow \text{left-total} \)
- **\\forall x, \exists!y, R x y**

**unfolding** right-unique-def left-total-def **by** blast

**lemma functional-converse-relation:**
- **left-unique** \( R \implies \rightarrow \text{right-total} \)
- **\\forall y, \exists!x, R x y**

**unfolding** left-unique-def right-total-def **by** blast

**lemma neg-fun-distr1:**
- **assumes** \( 1: \text{left-unique} R \text{ right-total} R \)
- **assumes** \( 2: \text{right-unique} R' \text{ left-total} R' \)
- **shows** \( (R \ OO R' \equiv \rightarrow S \ OO S') \leq ((R \equiv \rightarrow S) \ OO (R' \equiv \rightarrow S')) \)
- **apply** clarify
- **apply** (subst all-comm)
- **apply** (subst all-conj-distrib[symmetric])
- **apply** (intro choice)
by metis

lemma neg-fun-distr2:
assumes 1: right-unique R' left-total R'
assumes 2: left-unique S' right-total S'
shows (R OO R' ===> S OO S') \leq ((R ===> S) OO (R' ===> S'))
unfolding rel-fun-def OO-def
apply clarify
apply (subst all-comm)
apply (subst all-conj-distrib[symmetric])
apply (intro choice)
by metis

44.5 Domains

lemma composed-equiv-rel-eq-onp:
assumes left-unique R
assumes (R ===> (=)) P P'
assumes Domainp R = P''
shows (R OO eq-onp P' OO R^{-1-1}) = eq-onp (inf P'' P)
using assms unfolding OO-def conversep-iff Domainp-iff[abs-def] left-unique-def
rel-fun-def eq-onp-def
fun-eq-iff by blast

lemma composed-equiv-rel-eq-eq-onp:
assumes left-unique R
assumes Domainp R = P
shows (R OO (=) OO R^{-1-1}) = eq-onp P
using assms unfolding OO-def conversep-iff Domainp-iff[abs-def] left-unique-def
eq-onp-def
fun-eq-iff is-equality-def by metis

lemma per-Domainp-par-left-total:
assumes Domainp B = P
assumes left-total A
assumes (A ===> (=)) P' P
shows Domainp (A OO B) = P'
using assms
unfolding Domainp-iff[abs-def] OO-def bi-unique-def left-total-def rel-fun-def
by (fast intro: fun-eq-iff)

lemma per-Domainp-par:
assumes Domainp B = P2
assumes Domainp A = P1
assumes (A ===> (=)) P2' P2
shows Domainp (A OO B) = (inf P1 P2')
using assms unfolding rel-fun-def Domainp-iff[abs-def] OO-def
by (fast intro: fun-eq-iff)
def \text{rel-pred-comp} :: (\text{'}a \to \text{'}b \to \text{bool}) \to (\text{'}b \to \text{bool}) \to \text{'}a \to \text{bool}

\text{where} \quad \text{rel-pred-comp} ~ R ~ P \equiv \lambda x. \exists y. ~ R x y \land P y

\text{lemma} \quad \text{per-Domainp}:
\begin{align*}
\text{assumes} \quad & \text{Domainp} \ B = P \\
\text{shows} \quad & \text{Domainp} \ (A \ O O \ B) = (\lambda x. \exists y. \ A x y \land P y) \\
\text{using} \quad & \text{assms by blast}
\end{align*}

\text{lemma} \quad \text{per-Domainp-total}:
\begin{align*}
\text{assumes} \quad & \text{left-total} \ B \\
\text{assumes} \quad & \text{Domainp} \ A = P \\
\text{shows} \quad & \text{Domainp} \ (A \ O O \ B) = P \\
\text{using} \quad & \text{assms unfolding left-total-def} \\
\text{by} \quad & \text{fast}
\end{align*}

\text{lemma} \quad \text{Quotient-to-Domainp}:
\begin{align*}
\text{assumes} \quad & \text{Quotient} \ R \ A B \ T \\
\text{shows} \quad & \text{Domainp} \ T = (\lambda x. \ R x x) \\
\text{by} \quad & (\text{simp add: Domainp-iff[abs-def]} \ \text{Quotient-cr-rel[OF assms]})
\end{align*}

\text{lemma} \quad \text{eq-onp-to-Domainp}:
\begin{align*}
\text{assumes} \quad & \text{Quotient} \ (\text{eq-onp} \ P) \ A B \ T \\
\text{shows} \quad & \text{Domainp} \ T = P \\
\text{by} \quad & (\text{simp add: eq-onp-def Domainp-iff[abs-def]} \ \text{Quotient-cr-rel[OF assms]})
\end{align*}

\text{end}

\text{lemma} \quad \text{right-total-UNIV-transfer}:
\begin{align*}
\text{assumes} \quad & \text{right-total} \ A \\
\text{shows} \quad & \text{rel-set} \ A \ \text{(Collect (Domainp A))} \ \text{UNIV} \\
\text{using} \quad & \text{assms unfolding right-total-def rel-set-def Domainp-iff} \ \text{by blast}
\end{align*}

44.6 ML setup

\text{ML-file} \quad (\text{Tools/Lifting/lifting-util.ML})

\text{named-theorems} \quad \text{relator-eq-onp}

\text{theorems that a relator of an eq-onp is an eq-onp of the corresponding predicate}

\text{ML-file} \quad (\text{Tools/Lifting/lifting-info.ML})

\text{declare} \quad \text{fun-quotient[quot-map]}
\text{declare} \quad \text{fun-mono[relator-mono]}
\text{lemmas} \quad \text{[relator-distr] = pos-fun-distr neg-fun-distr1 neg-fun-distr2}

\text{ML-file} \quad (\text{Tools/Lifting/lifting-bnf.ML})
\text{ML-file} \quad (\text{Tools/Lifting/lifting-term.ML})
45 Definition of Quotient Types

Basic definition for equivalence relations that are represented by predicates.

Composition of Relations

abbreviation
rel-conj :: (′a ⇒ ′b ⇒ bool) ⇒ (′b ⇒ ′a ⇒ bool) ⇒ ′a ⇒ ′b ⇒ bool (infixr OOO 75)

where
r1 OOO r2 ≡ r1 OO r2 OO r1

lemma eq-comp-r:
shows ((=) OOO R) = R
by (auto simp add: fun-eq-iff)

45.1 Quotient Predicate

definition
Quotient3 R Abs Rep ⟷
(∀ a. Abs (Rep a) = a) ∧ (∀ a. R (Rep a) (Rep a)) ∧
(∀ r s. R r s ⟷ R r r ∧ R s s ∧ Abs r = Abs s)
lemma Quotient3I:
  assumes \( \forall a. \text{Abs} (\text{Rep} a) = a \)
  and \( \forall a. R (\text{Rep} a) (\text{Rep} a) \)
  and \( \forall r s. R r s \iff R r r \land R s s \land \text{Abs} r = \text{Abs} s \)
  shows Quotient3 R Abs Rep
  using assms unfolding Quotient3-def by blast

context
  fixes R Abs Rep
  assumes a: Quotient3 R Abs Rep
begin

lemma Quotient3-abs-rep:
  Abs (Rep a) = a
  using a
  unfolding Quotient3-def
  by simp

lemma Quotient3-rep-reflp:
  R (Rep a) (Rep a)
  using a
  unfolding Quotient3-def
  by blast

lemma Quotient3-rel:
  R r r \land R s s \land Abs r = Abs s \iff R r s — orientation does not loop on
  rewriting
  using a
  unfolding Quotient3-def
  by blast

lemma Quotient3-refl1:
  R r s \implies R r r
  using a unfolding Quotient3-def
  by fast

lemma Quotient3-refl2:
  R r s \implies R s s
  using a unfolding Quotient3-def
  by fast

lemma Quotient3-rel-rep:
  R (Rep a) (Rep b) \iff a = b
  using a
  unfolding Quotient3-def
  by metis

lemma Quotient3-rep-abs:
THEORY “Quotient”

\[ R \, r \, r \implies R \, (\text{Rep} \, (\text{Abs} \, r)) \, r \]
\[ \text{using a unfolding Quotient3-def} \]
\[ \text{by blast} \]

lemma Quotient3-rel-abs:
\[ R \, r \, s \implies \text{Abs} \, r = \text{Abs} \, s \]
\[ \text{using a unfolding Quotient3-def} \]
\[ \text{by blast} \]

lemma Quotient3-symp:
\[ \text{symp} \, R \]
\[ \text{using a unfolding Quotient3-def} \text{ using sympI by metis} \]

lemma Quotient3-transp:
\[ \text{transp} \, R \]
\[ \text{using a unfolding Quotient3-def} \text{ using transpI by (metis (full-types))} \]

lemma Quotient3-part-eqvlp:
\[ \text{part-eqvlp} \, R \]
\[ \text{by (metis Quotient3-rep-reflp Quotient3-symp Quotient3-transp part-eqvlpI)} \]

lemma abs-o-rep:
\[ \text{Abs} \circ \text{Rep} = \text{id} \]
\[ \text{unfolding fun-eq-iff} \]
\[ \text{by (simp add: Quotient3-abs-rep)} \]

lemma equals-rsp:
\[ \text{assumes b: } R \, x_a \, x_b \, R \, y_a \, y_b \]
\[ \text{shows } R \, x_a \, y_a = R \, x_b \, y_b \]
\[ \text{using } b \, \text{Quotient3-symp Quotient3-transp} \]
\[ \text{by (blast elim: sympE transpE)} \]

lemma rep-abs-rsp:
\[ \text{assumes b: } R \, x_1 \, x_2 \]
\[ \text{shows } R \, x_1 \, (\text{Rep} \, (\text{Abs} \, x_2)) \]
\[ \text{using } b \, \text{Quotient3-rel Quotient3-abs-rep Quotient3-rep-reflp} \]
\[ \text{by metis} \]

lemma rep-abs-rsp-left:
\[ \text{assumes b: } R \, x_1 \, x_2 \]
\[ \text{shows } R \, (\text{Rep} \, (\text{Abs} \, x_1)) \, x_2 \]
\[ \text{using } b \, \text{Quotient3-rel Quotient3-abs-rep Quotient3-rep-reflp} \]
\[ \text{by metis} \]

end

lemma identity-quotient3:
\[ \text{Quotient3} \, (=) \, \text{id} \, \text{id} \]
\[ \text{unfolding Quotient3-def id-def} \]
THEORY "Quotient" 879

by blast

lemma fun-quotient3:
  assumes q1: Quotient3 R1 abs1 rep1
  and q2: Quotient3 R2 abs2 rep2
  shows Quotient3 (R1 ===> R2) (rep1 ===> abs2) (abs1 ===> rep2)
proof
  have (rep1 ===> abs2) ((abs1 ===> rep2) a) = a for a
    using q1 q2 by (simp add: Quotient3-def fun-eq-iff)
  moreover have (R1 ===> R2) ((abs1 ===> rep2) a) ((abs1 ===> rep2) a) for a
    by (rule rel-funI)
    (insert q1 q2 Quotient3-rel-abs [of R1 abs1 rep1] Quotient3-rel-rep [of R2 abs2 rep2],
    simp (no_asm) add: Quotient3-def, simp)
  moreover have (R1 ===> R2) r s = ((R1 ===> R2) r r ∧ (R1 ===> R2) s s ∧
    (rep1 ===> abs2) r = (rep1 ===> abs2) s) for r s
    unfolding rel-fun-def
    using Quotient3-part-equivp[OF q1] Quotient3-part-equivp[OF q2]
    by (metis (full-types) part-equivp-def)
  moreover have (R1 ===> R2) r s = (rep1 ===> abs2) r = (rep1 ===> abs2) s)
    =⇒ (R1 ===> R2) r s
    unfolding rel-fun-def
    by (auto simp add: rel-fun-def fun-eq-iff)
    (use q1 q2 in (unfold Quotient3-def, metis))
  moreover have ((R1 ===> R2) r r ∧ (R1 ===> R2) s s ∧
    (rep1 ===> abs2) r = (rep1 ===> abs2) s) =⇒ (R1 ===> R2) r s
    by (auto simp add: rel-fun-def fun-eq-iff)
    (use q1 q2 in (unfold Quotient3-def, metis map-fun-apply))
ultimately show ?thesis by blast
qed

ultimately show ?thesis by (intro Quotient3I) (assumption+)
qed

lemma lambda-prs:
  assumes q1: Quotient3 R1 Abs1 Rep1
  and q2: Quotient3 R2 Abs2 Rep2
  shows (Rep1 ===> Abs2) (λx. Rep2 (f (Abs1 x))) = (λx. f x)
unfolding fun-eq-iff
using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]
by simp

lemma lambda-prs1:
  assumes q1: Quotient3 R1 Abs1 Rep1
and \( q_2 : \text{Quotient3} R_2 \text{Abs2} \text{Rep2} \)
shows \((\text{Rep1} \Longrightarrow \text{Abs2}) \ (\lambda x. (\text{Abs1} \Longrightarrow \text{Rep2}) \ f \ x) = (\lambda f. f x)\)
unfolding fun-eq-iff
using Quotient3-abs-rep[OF \( q_1 \)] Quotient3-abs-rep[OF \( q_2 \)]
by simp

In the following theorem \( R_1 \) can be instantiated with anything, but we know some of the types of the \( \text{Rep} \) and \( \text{Abs} \) functions; so by solving \( \text{Quotient} \) assumptions we can get a unique \( R_1 \) that will be provable; which is why we need to use apply-rsp and not the primed version

lemma apply-rspQ3:
  fixes \( f\ g :: 'a \Rightarrow 'c \)
  assumes \( q : \text{Quotient3} R_1 \text{Abs1} \text{Rep1} \)
  and \( a : (R_1 \Longrightarrow R_2) \ f\ g\ R_1\ x\ y \)
  shows \( R_2(\ f\ x)\ (\ g\ y) \)
  using \( a \) by (auto elim: rel-funE)

lemma apply-rspQ3'':
  assumes \( \text{Quotient3} R \text{Abs} \text{Rep} \)
  and \((R \Longrightarrow S) \ f\ f\)
  shows \( S(\ f\ (\text{Rep}\ x))\ (\ f\ (\text{Rep}\ x))\)
proof –
  from \( \text{assms}(1) \) have \( R\ (\text{Rep}\ x)\ (\text{Rep}\ x) \) by \( \text{rule Quotient3-rep-reflp} \)
then show \( ?\text{thesis} \) using \( \text{assms}(2) \) by \( \text{auto intro: apply-rsp'} \)
qed

45.2 lemmas for regularisation of ball and \( \text{bex} \)

lemma ball-reg-equiv:
  fixes \( P :: 'a \Rightarrow \text{bool} \)
  assumes \( a : \text{equivp} R \)
  shows \( \text{Ball} (\text{Respects} \ R) \ P = (\text{All} P) \)
  using \( a \)
  unfolding equivp-def
  by \( \text{auto simp add: in-respects} \)

lemma bex-reg-equiv:
  fixes \( P :: 'a \Rightarrow \text{bool} \)
  assumes \( a : \text{equivp} R \)
  shows \( \text{Bex} (\text{Respects} \ R) \ P = (\text{Ex} P) \)
  using \( a \)
  unfolding equivp-def
  by \( \text{auto simp add: in-respects} \)

lemma ball-reg-right:
  assumes \( a : \forall x. x \in R \Longrightarrow P \ x \Longrightarrow Q \ x \)
  shows \( \text{All} P \Longrightarrow \text{Ball} R \ P \)
  using \( a \) by \( \text{fast} \)
lemma bex-reg-left:
assumes a: \( \forall x. x \in R \Rightarrow Q x \rightarrow P x \)
shows \( Bex R Q \rightarrow Ex P \)
using a by fast

lemma ball-reg-left:
assumes a: equivp R
shows \( (\forall x. (Q x \rightarrow P x)) \Rightarrow Ball (Respects R) Q \rightarrow All P \)
using a by (metis equivp-reflp in-respects)

lemma bex-reg-right:
assumes a: equivp R
shows \( (\forall x. (Q x \rightarrow P x)) \Rightarrow Ex Q \rightarrow Bex (Respects R) P \)
using a by (metis equivp-reflp in-respects)

lemma ball-reg-eqv-range:
fixes P::'a \Rightarrow bool
and x::'a
assumes a: equivp R2
shows \( (Ball (Respects (R1 \implies R2))) (\lambda f. P (f x)) = All (\lambda f. P (f x)) \)
apply(rule iffI)
apply(rule allI)
apply(rule_tac x=\lambda y. f x in bspec)
apply(simp add: in-respects rel-fun-def)
apply(rule impl)
using a equivp-reflp-symp-transp[of R2]
apply (auto elim: equivpE reflpE)
done

lemma bex-reg-eqv-range:
assumes a: equivp R2
shows \( (Bex (Respects (R1 \implies R2))) (\lambda f. P (f x)) = Ex (\lambda f. P (f x)) \)
apply(auto)
apply(rule-tac x=\lambda y. f x in bexI)
apply(simp)
apply(simp add: Respects-def in-respects rel-fun-def)
apply(rule impl)
using a equivp-reflp-symp-transp[of R2]
apply (auto elim: equivpE reflpE)
done

lemma all-reg:
assumes a: \( \forall x :: 'a. (P x \rightarrow Q x) \)
and b: All P
shows All Q
using a b by fast

lemma ex-reg:
assumes $a: \forall x :: 'a. (P x \rightarrow Q x)$
and $b: \text{Ex } P$
shows $\text{Ex } Q$
using $a$ $b$ by fast

lemma ball-reg:
assumes $a: \forall x :: 'a. (x \in R \rightarrow P x \rightarrow Q x)$
and $b: \text{Ball } R P$
shows $\text{Ball } R Q$
using $a$ $b$ by fast

lemma bex-reg:
assumes $a: \forall x :: 'a. (x \in R \rightarrow P x \rightarrow Q x)$
and $b: \text{Bex } R P$
shows $\text{Bex } R Q$
using $a$ $b$ by fast

lemma ball-all-comm:
assumes $\forall y. (\forall x \in P. A x y) \rightarrow (\forall x. B x y)$
shows $(\forall x \in P. \forall y. A x y) \rightarrow (\forall x. \forall y. B x y)$
using $\text{assms by auto}$

lemma bex-ex-comm:
assumes $(\exists y. \exists x. A x y) \rightarrow (\exists y. \exists x \in P. B x y)$
shows $(\exists x. \exists y. A x y) \rightarrow (\exists x \in P. \exists y. B x y)$
using $\text{assms by auto}$

45.3 Bounded abstraction

definition
$Babs :: 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
where
$x \in p \Rightarrow Babs \ p \ m \ x = m \ x$

lemma babs-rsp:
assumes $q: \text{Quotient3 } R1 \ \text{Abs1 } \text{Rep1}$
and $a: (R1 \Longrightarrow R2) \ f \ g$
shows $(R1 \Longrightarrow R2) \ (\text{Babs } (\text{Respects } R1) \ f) \ (\text{Babs } (\text{Respects } R1) \ g)$
apply (auto simp add: $\text{Babs-def}$ in-respects $\text{rel-fun-def}$)
apply (subgoal-tac $x \in \text{Respects } R1 \ \&\ y \in \text{Respects } R1$)
using $a$ apply (simp add: $\text{Babs-def}$ $\text{rel-fun-def}$)
apply (simp add: $\text{in-respects}$ $\text{rel-fun-def}$)
using $\text{Quotient3-rel}[\text{OF } q]$
by metis

lemma babs-prs:
assumes $q1: \text{Quotient3 } R1 \ \text{Abs1 } \text{Rep1}$
and $q2: \text{Quotient3 } R2 \ \text{Abs2 } \text{Rep2}$
shows \((\text{Rep}1 \longrightarrow \text{Abs}2 \ (\text{Babs} \ (\text{Respects} \ R1) \ (\text{Abs}1 \longrightarrow \text{Rep}2) \ f))) = f\)

apply (rule ext)
apply (simp add:)
apply (subgoal_tac \text{Rep}1 \ x \in \text{Respects} \ R1)
apply (simp add: \text{Babs-def} \ Quotient3-abs-rep[\text{OF} \ q1] \ Quotient3-abs-rep[\text{OF} \ q2])
apply (simp add: in-respects \text{Quotient3-rel-rep}[\text{OF} \ q1])
done

lemma \text{babs-simp}:  
assumes \(q\) : \text{Quotient3} \ R1 \ Abs \ Rep  
shows \( ((R1 \Longrightarrow R2) \ (\text{Babs} \ (\text{Respects} \ R1) \ f) \ (\text{Babs} \ (\text{Respects} \ R1) \ g)) = ((R1 \Longrightarrow R2) \ f \ g)\)  
apply (rule iffI)  
apply (simp-all only: \text{babs-rsp}[\text{OF} \ q])  
apply (auto simp add: \text{Babs-def} \ rel-fun-def)  
apply (metis \text{Babs-def} \ in-respects \text{Quotient3-rel}[\text{OF} \ q])
done

lemma \text{babs-reg-eqv}:  
shows \( \text{equivp} \ R \Longrightarrow \text{Babs} \ (\text{Respects} \ R) \ P = P \)
by (simp add: \text{fun-eq-iff} \text{Babs-def} \ in-respects \text{equivp-reflp})

lemma \text{ball-rsp}:  
assumes \(a\) : \( (R \Longrightarrow (=)) \ f \ g \)
shows \( \text{Ball} \ (\text{Respects} \ R) \ f = \text{Ball} \ (\text{Respects} \ R) \ g \)
using \(a\) by (auto simp add: \text{Ball-def} \ in-respects \text{elim: rel-funE})

lemma \text{bex-rsp}:  
assumes \(a\) : \( (R \Longrightarrow (=)) \ f \ g \)
shows \( \text{Bex} \ (\text{Respects} \ R) \ f = \text{Bex} \ (\text{Respects} \ R) \ g \)
using \(a\) by (auto simp add: \text{Bex-def} \ in-respects \text{elim: rel-funE})

lemma \text{bex1-rsp}:  
assumes \(a\) : \( (R \Longrightarrow (=)) \ f \ g \)
shows \( \text{Ex1} \ (\lambda x. \ x \in \text{Respects} \ R \land f x) = \text{Ex1} \ (\lambda x. \ x \in \text{Respects} \ R \land g x) \)
using \(a\) by (auto elim: rel-funE simp add: \text{Ex1-def} \ in-respects)

lemma \text{all-prs}:  
assumes \(a\) : \text{Quotient3} \ R \ absf \ repf
shows \( \text{Ball} \ (\text{Respects} \ R) \ ((\text{absf} \longrightarrow \text{id}) \ f) = \text{All} \ f \)
using \(a\) unfolding \text{Quotient3-def} \text{Ball-def} \ in-respects \text{id-apply} \text{comp-def} \text{map-fun-def}  
by \text{metis}

lemma \text{ex-prs}:
THEORY “Quotient”

assumes a: Quotient3 R absf repf
shows Bex (Respects R) ((absf --- id) f) = Ex f
using a unfolding Quotient3-def Bex-def in-respects id-apply comp-def map-fun-def
by metis

45.4 Bex1-rel quantifier

definition
Bex1-rel :: ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ bool
where
Bex1-rel R P ←→ (∃ x ∈ Respects R. P x) ∧ (∀ x ∈ Respects R. ∀ y ∈ Respects R. ((P x ∨ P y) → (R x y)))

lemma bex1-rel-aux:
[∀ xa ya. R xa ya → x xa = y ya; Bex1-rel R x] ⇒ Bex1-rel R y
unfolding Bex1-rel-def
by (metis in-respects)

lemma bex1-rel-aux2:
[∀ xa ya. R xa ya → x xa = y ya; Bex1-rel R y] ⇒ Bex1-rel R x
unfolding Bex1-rel-def
by (metis in-respects)

lemma bex1-rel-rsp:
assumes a: Quotient3 R absf repf
shows ((R === (==)) === (==)) (Bex1-rel R) (Bex1-rel R)
unfolding rel-fun-def by (metis bex1-rel-aux bex1-rel-aux2)

lemma ex1-prs:
assumes Quotient3 R absf repf
shows ((absf --- id) --- id) (Bex1-rel R) f = Ex1 f
apply (auto simp: Bex1-rel-def Respects-def)
apply (metis Quotient3-def assms)
apply (metis full-types Quotient3-def assms)
by (meson Quotient3-rel assms)

lemma bex1-bexeq-reg:
shows (∃! x ∈ Respects R. P x) → (Bex1-rel R (λ x. P x))
by (auto simp add: Ex1-def Bex1-rel-def Bex1-rel-def in-respects)

lemma bex1-bexeq-reg-eqv:
assumes a: equivp R
shows (∃! x. P x) → Bex1-rel R P
using equivp-reflp[OF a]
by (metis full-types Bex1-rel-def in-respects)

45.5 Various respects and preserve lemmas

lemma quot-rel-rsp:
assumes a: Quotient3 R Abs Rep
shows \((R \Longrightarrow R \Longrightarrow (=)) R R\)
apply (rule rel-funI)+
by (meson assms equals-rsp)

lemma o-prs:
assumes q1: Quotient3 R1 Abs1 Rep1
and q2: Quotient3 R2 Abs2 Rep2
and q3: Quotient3 R3 Abs3 Rep3
shows ((Abs2 \Longrightarrow Rep3) \Longrightarrow (Abs1 \Longrightarrow Rep2) \Longrightarrow (Rep1 \Longrightarrow Abs3)) (\circ) = (\circ)
and (id \Longrightarrow (Abs1 \Longrightarrow id) \Longrightarrow Rep1 \Longrightarrow id) (\circ) = (\circ)
using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2] Quotient3-abs-rep[OF q3]
by (simp-all add: fun-eq-iff)

lemma o-rsp:
\(((R1 \Longrightarrow R3) \Longrightarrow (R1 \Longrightarrow R2) \Longrightarrow (R1 \Longrightarrow R3)) (\circ) (\circ)\n\(((=) \Longrightarrow (R1 \Longrightarrow (=)) \Longrightarrow R1 \Longrightarrow (=)) (\circ) (\circ)\nby (force elim: rel-funE)+

lemma cond-prs:
assumes a: Quotient3 R absf repf
shows absf (if a then repf b else repf c) = (if a then b else c)
using a unfolding Quotient3-def by auto

lemma if-prs:
assumes q: Quotient3 R Abs Rep
shows (id \Longrightarrow Rep \Longrightarrow Rep \Longrightarrow Abs) If = If
using Quotient3-abs-rep[OF q]
by (auto simp add: fun-eq-iff)

lemma if-rsp:
assumes q: Quotient3 R Abs Rep
shows ((=) \Longrightarrow R \Longrightarrow R \Longrightarrow R) If If
by force

lemma let-prs:
assumes q1: Quotient3 R1 Abs1 Rep1
and q2: Quotient3 R2 Abs2 Rep2
shows (Rep2 \Longrightarrow (Abs2 \Longrightarrow Rep1) \Longrightarrow Abs1) Let = Let
using Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]
by (auto simp add: fun-eq-iff)

lemma let-rsp:
shows (R1 \Longrightarrow R1 \Longrightarrow R2) \Longrightarrow (R1 \Longrightarrow R2) Let Let
by (force elim: rel-funE)

lemma id-rsp:
shows (R \Longrightarrow R) id id
by auto

lemma id-prs:
  assumes a: Quotient3 R Abs Rep
  shows (Rep −−−> Abs) id = id
  by (simp add: fun-eq-iff Quotient3-abs-rep [OF a])
end

locale quot-type =
  fixes R :: 'a ⇒ 'a ⇒ bool
  and Abs :: 'a set ⇒ 'b
  and Rep :: 'b ⇒ 'a set
  assumes equivp: part-equivp R
  and rep-prop: ∀ y. ∃ x. R x x ∧ Rep y = Collect (R x)
  and rep-inverse: ∀ x. Abs (Rep x) = x
  and abs-inverse: ∀ c. (∃ x. ((R x x) ∧ (c = Collect (R x)))) ⇒ (Rep (Abs c)) = c
  and rep-inject: ∀ x y. (Rep x = Rep y) = (x = y)
begin

definition abs :: 'a ⇒ 'b where
  abs x = Abs (Collect (R x))

definition rep :: 'b ⇒ 'a where
  rep a = (SOME x. x ∈ Rep a)

lemma some-collect:
  assumes R r r
  shows (SOME x. x ∈ Collect (R r)) = R r
  apply simp
  by (metis assms exE-some equivp[simplified part-equivp-def])

lemma Quotient:
  shows Quotient3 R abs rep
  unfolding Quotient3-def abs-def rep-def
  proof (intro conjI allI)
    fix a r s
    show x: R (SOME x. x ∈ Rep a) (SOME x. x ∈ Rep a) proof –
      obtain x where r: R x x and rep: Rep a = Collect (R x) using rep-prop[of a] by auto
      have R (SOME x. x ∈ Rep a) x using r rep some-collect by metis
      then have R x (SOME x. x ∈ Rep a) using part-equivp-symp[of R equivp] by fast
      then show R (SOME x. x ∈ Rep a) (SOME x. x ∈ Rep a)
  end
using part-equiv-transp[OF equivp] by (metis 'R (SOME x. x ∈ Rep a)
x)
qed

have Collect (R (SOME x. x ∈ Rep a)) = (Rep a) by (metis some-collect rep-prop)
then show Abs (Collect (R (SOME x. x ∈ Rep a))) = a using rep-inverse by auto

have R r r ⇒ R s s ⇒ Abs (Collect (R r)) = Abs (Collect (R s)) ←→ R r = R s
proof −
assume R r r and R s s
then have Abs (Collect (R r)) = Abs (Collect (R s)) ←→ Collect (R r) = Collect (R s)
by (metis abs-inverse)
also have Collect (R r) = Collect (R s) ←→ (λA x. x ∈ A) (Collect (R r))
= (λA x. x ∈ A) (Collect (R s))
by rule simp-all
finally show Abs (Collect (R r)) = Abs (Collect (R s)) ←→ R r = R s by simp
qed

then show R r s ←→ R r r ∧ R s s ∧ (Abs (Collect (R r)) = Abs (Collect (R s)))
using equivp[simplified part-equiv-def] by metis
qed

end

45.6 Quotient composition

lemma OOO-quotient3:
fixes R1 :: 'a ⇒ 'a ⇒ bool
fixes Abs1 :: 'a ⇒ 'b and Rep1 :: 'b ⇒ 'a
fixes Abs2 :: 'b ⇒ 'c and Rep2 :: 'c ⇒ 'b
fixes R2' :: 'a ⇒ 'a ⇒ bool
fixes R2 :: 'b ⇒ 'b ⇒ bool
assumes R1: Quotient3 R1 Abs1 Rep1
assumes R2: Quotient3 R2 Abs2 Rep2
assumes Abs1: ∃x y. R2' x y ⇒ R1 x x ⇒ R1 y y ⇒ R2 (Abs1 x) (Abs1 y)
assumes Rep1: ∃x y. R2 x y ⇒ R2' (Rep1 x) (Rep1 y)
shows Quotient3 (R1 OO R2' OO R1) (Abs2 oo Abs1) (Rep1 oo Rep2)
proof −
have *: (R1 OOO R2') r r ∧ (R1 OOO R2') s s ∧ (Abs2 oo Abs1) r = (Abs2 oo Abs1) s
  ⇒ (R1 OOO R2') r s for r s
apply safe
subgoal for a b c d
  apply simp
  apply (rule-tac b=Rep1 (Abs1 r) in relcomppI)
using Quotient3-refl1 R1 rep-abs-rsp apply fastforce
apply (rule-tac b=Rep1 (Abs1 s) in relcomppI)
apply (metis (full-types) Rep1 Abs1 Quotient3-rel R2 Quotient3-refl1 [OF R1] Quotient3-refl2 [OF R1] Quotient3-rel-abs [OF R1])
by (metis Quotient3-rel R1 rep-abs-rsp-left)

subgoal for x y
  apply (drule Abs1)
  apply (erule Quotient3-refl2 [OF R1])
  apply (erule Quotient3-refl1 [OF R1])
  apply (drule Quotient3-refl2 [OF R2], drule Rep1)
by (metis (full-types) Quotient3-def R1 relcompp.relcomppI)

subgoal for x y
  by simp (metis (full-types) Abs1 Quotient3-rel R1 R2)
done

show ?thesis
  apply (rule Quotient3I)
using * apply (simp-all add: o-def Quotient3-abs-rep [OF R2] Quotient3-abs-rep [OF R1])
  apply (metis Quotient3-rep-reflp R1 R2 Rep1 relcompp.relcomppI)
done

qed

lemma OOO-eq-quotient3:
  fixes R1 :: 'a ⇒ 'a ⇒ bool
  fixes Abs1 :: 'a ⇒ 'b and Rep1 :: 'b ⇒ 'a
  fixes Abs2 :: 'b ⇒ 'c and Rep2 :: 'c ⇒ 'b
  assumes R1: Quotient3 R1 Abs1 Rep1
  assumes R2: Quotient3 (=) Abs2 Rep2
  shows Quotient3 (R1 OOO (=)) (Abs2 ◦ Abs1) (Rep1 ◦ Rep2)
using assms
by (rule OOO-quotient3) auto

45.7 Quotient3 to Quotient

lemma Quotient3-to-Quotient:
  assumes Quotient3 R Abs Rep
  and T ≡ λx y. R x x ∧ Abs x = y
  shows Quotient R Abs Rep T
using assms unfolding Quotient3-def by (intro QuotientI) blast+

lemma Quotient3-to-Quotient-equivp:
  assumes q: Quotient3 R Abs Rep
  and T-def: T ≡ λx y. Abs x = y
and \( eR \): equivp \( R \)

shows Quotient \( R \) Abs Rep \( T \)

proof (intro QuotientI)
  fix \( a \)
  show \( \text{Abs} \ (\text{Rep} \ a) = a \) using \( q \) by (rule Quotient3-abs-rep)

next
  fix \( a \)
  show \( \text{R} \ (\text{Rep} \ a) \ (\text{Rep} \ a) \) using \( q \) by (rule Quotient3-rep-reflp)

next
  fix \( r \ s \)
  show \( \text{R} \ r \ s = (\text{R} \ r \ r \land \text{R} \ s \ s \land \text{Abs} \ r = \text{Abs} \ s) \) using \( q \) by (rule Quotient3-rel[ symmetric ])

next
  show \( T = (\lambda x \ y. \ R \ x \ x \land \text{Abs} \ x = y) \) using \( T\text{-def equivp-reflp}[OF } eR \) by simp

qed

45.8 ML setup

Auxiliary data for the quotient package

named-theorems quot-equiv equivalence relation theorems
  and quot-respect respectfulness theorems
  and quot-preserve preservation theorems
  and id-simps identity simp rules for maps
  and quot-thm quotient theorems
ML-file ⟨Tools/Quotient/quotient-info.ML⟩

declare [[mapQ3 fun = (rel-fun, fun-quotient3)]]

lemmas [quot-thm] = fun-quotient3
lemmas [quot-respect] = quot-rel-rsp if-rsp o-rsp let-rsp id-rsp
lemmas [quot-preserve] = if-prs o-prs let-prs id-prs
lemmas [quot-equiv] = identity-equivp

Lemmas about simplifying id’s.

lemmas [id-simps] =
id-def[ symmetric ]
map-fun-id
id-apply
id-o
o-id
eq-comp-r
vimage-id

Translation functions for the lifting process.
ML-file ⟨Tools/Quotient/quotient-term.ML⟩

Definitions of the quotient types.
ML-file ⟨Tools/Quotient/quotient-type.ML⟩

Definitions for quotient constants.
An auxiliary constant for recording some information about the lifted theorem in a tactic.

**Definition**

Quot-True :: 'a ⇒ bool

where

Quot-True x ⟷ True

**Lemma**

shows QT-all: Quot-True (All P) ⟷ Quot-True P

and QT-ex: Quot-True (Ex P) ⟷ Quot-True P

and QT-ex1: Quot-True (Ex1 P) ⟷ Quot-True P

and QT-lam: Quot-True (λx. P x) ⟷ (λx. Quot-True (P x))

and QT-ext: (λx. Quot-True (a x) ⟷ f x = g x) ⟷ (Quot-True a ⟷ f = g)

by (simp-all add: Quot-True-def ext)

**Lemma** QT-imp: Quot-True a ≡ Quot-True b

by (simp add: Quot-True-def)

**Context** includes lifting-syntax

**Begin**

Tactics for proving the lifted theorems

**ML-file** (Tools/Quotient/quotient-tacs.ML)

**End**

### 45.9 Methods / Interface

**Method-setup** lifting =

Attrib.thms >> (fn thms => fn ctxt =>)

SIMPLE-METHOD' (Quotient-Tacs.lift-tac ctxt [] thms))

(lift theorems to quotient types)

**Method-setup** lifting-setup =

Attrib.thm >> (fn thm => fn ctxt =>)

SIMPLE-METHOD' (Quotient-Tacs.lift-procedure-tac ctxt [] thm))

(set up the three goals for the quotient lifting procedure)

**Method-setup** descending =

Scan.succeed (fn ctxt => SIMPLE-METHOD' (Quotient-Tacs.descend-tac ctxt [])))

(descend theorems to the raw level)

**Method-setup** descending-setup =

Scan.succeed (fn ctxt => SIMPLE-METHOD' (Quotient-Tacs.descend-procedure-tac ctxt [])))
THEORY “Quotient”

(set up the three goals for the decending theorems)

method-setup partiality-descending =
<Scan.succeed (fn ctxt => SIMPLE-METHOD' (Quotient-Tacs.partiality-descend-tac ctxt []))>
(descend theorems to the raw level)

method-setup partiality-descending-setup =
<Scan.succeed (fn ctxt => SIMPLE-METHOD' (Quotient-Tacs.partiality-descend-procedure-tac ctxt []))>
(set up the three goals for the decending theorems)

method-setup regularize =
<Scan.succeed (fn ctxt => SIMPLE-METHOD' (Quotient-Tacs.regularize-tac ctxt))>
(prove the regularization goals from the quotient lifting procedure)

method-setup injection =
<Scan.succeed (fn ctxt => SIMPLE-METHOD' (Quotient-Tacs.all-injection-tac ctxt))>
(prove the rep/abs injection goals from the quotient lifting procedure)

method-setup cleaning =
<Scan.succeed (fn ctxt => SIMPLE-METHOD' (Quotient-Tacs.clean-tac ctxt))>
(prove the cleaning goals from the quotient lifting procedure)

attribute-setup quot-lifted =
<Scan.succeed Quotient-Tacs.lifted-attrib>
(lift theorems to quotient types)

no-notation
rel-conj (infixr OOO 75)

46 Lifting of BNFs

lemma sum-insert-Inl-unit: x ∈ A ⇒ (∀ y. x = Inr y ⇒ Inr y ∈ B) ⇒ x ∈ insert (Inl ()) B
 by (cases x) (simp-all)

lemma lift-sum-unit-vimage-commute:
 insert (Inl ()) (Inr ' f ' A) = map-sum id f ' insert (Inl ()) (Inr ' A)
 by (auto simp: map-sum-def split: sum.splits)

lemma insert-Inl-int-map-sum-unit: insert (Inl ()) A ∩ range (map-sum id f) ≠ {}
 by (auto simp: map-sum-def split: sum.splits)

lemma image-map-sum-unit-subset:
 A ⊆ insert (Inl ()) (Inr ' B) ⇒ map-sum id f ' A ⊆ insert (Inl ()) (Inr ' f ' B)
by auto

lemma subset-lift-sum-unitD: A ⊆ insert (Inl ()) (Inr ' B) ⇒ Inr x ∈ A ⇒ x ∈ B
  unfolding insert-def by auto

lemma UNIV-sum-unit-conv: insert (Inl ()) (range Inr) = UNIV
  unfolding UNIV-sum UNIV-unit image-insert image-empty Un-insert-left sup-bot.left-neutral..

lemma subset-vimage-image-subset: A ⊆ f − ' B ⇒ f ' A ⊆ B
  by auto

lemma relcompp-mem-Grp-neq-bot: A ∩ range f ≠ {} ⇒ (λx y. x ∈ A ∧ y ∈ A) OO (Grp UNIV f)^−1−1 ≠ bot
  unfolding Grp-def relcompp-apply fun-eq-iff by blast

lemma comp-projr-Inr: projr ∘ Inr = id
  by auto

lemma in-rel-sum-in-image-projr:
  B ⊆ {(x,y). rel-sum (=) :: unit ⇒ unit ⇒ bool} A x y ⇒
  Inr ' C = fst ' B ⇒ snd ' B = Inr ' D ⇒ map-prod projr projr ' B ⊆ {(x,y).
  A x y}
  by (force simp: projr-def image-iff dest: spec[of - Inl ()] split: sum.splits)

lemma subset-rel-sumI: B ⊆ {(x,y). A x y} ⇒ rel-sum (=) :: unit ⇒ unit ⇒ bool) A
  (if x ∈ B then Inr (fst x) else Inl ())
  (if x ∈ B then Inr (snd x) else Inl ())
  by auto

lemma relcompp-eq-Grp-neq-bot: (=) OO (Grp UNIV f)^−1−1 ≠ bot
  unfolding Grp-def relcompp-apply fun-eq-iff by blast

lemma rel-fun-rel-OO1: (rel-fun Q (rel-fun R (=))) A B ⇒ conversep Q OO A OO R ≤ B
  by (auto simp: rel-fun-def)

lemma rel-fun-rel-OO2: (rel-fun Q (rel-fun R (=))) A B ⇒ Q OO B OO conversep R ≤ A
  by (auto simp: rel-fun-def)

lemma rel-sum-eq2-nonempty: rel-sum (=) A OO rel-sum (=) B ≠ bot
  by (auto simp: fun-eq-iff relcompp-apply intro!: exI[of - Inl ()])

lemma rel-sum-eq3-nonempty: rel-sum (=) A OO (rel-sum (=) B OO rel-sum (=) C) ≠ bot
  by (auto simp: fun-eq-iff relcompp-apply intro!: exI[of - Inl ()])
lemma hypsubst: \( A = B \implies x \in B \implies (x \in A \implies P) \implies P \) by simp

lemma Quotient-crel-quotient: Quotient \( R \) Abs Rep T \( \implies \) equivp \( R \implies T \equiv (\lambda x y. \text{Abs } x = y) \)
  by (drule Quotient-cr-rel) (auto simp: fun-eq-iff equivp-reflp intro: eq-reflection)

lemma Quotient-crel-typedef: Quotient \((\text{eq-onp } P)\) Abs Rep T \( \implies \) \( T \equiv (\lambda x y. x = \text{Rep } y) \)
  unfolding Quotient-def by (auto simp: fun-eq-iff fun-eq-iff symp-transp transp-relcompp)

lemma Quotient-crel-typecopy: Quotient \((\tau)\) Abs Rep T \( \implies \) \( T \equiv (\lambda x y. x = \text{Rep } y) \)
  unfolding Quotient-def by (auto)

lemma equivp-add-relconj:
  assumes \( \text{equiv: equivp } R \text{ equivp } R' \text{ and } \le: S \text{ OO } T \text{ OO } U \le R \text{ OO } STU \text{ OO } R' \)
  shows \( R \text{ OO } S \text{ OO } T \text{ OO } U \text{ OO } R' \le R \text{ OO } STU \text{ OO } R' \)
proof
t  have trans: \( R \text{ OO } R \le R \text{ R' } \text{ OO } R' \le R' \)
    using equiv unfolding equivp-reflp-symp-transp transp-relcompp by blast+
  have \( R \text{ OO } S \text{ OO } T \text{ OO } U \text{ OO } R' = R \text{ OO } (S \text{ OO } T \text{ OO } U) \text{ OO } R' \)
    unfolding relcompp-assoc ..
  also have \( \ldots \le R \text{ OO } (R \text{ OO } STU \text{ OO } R') \text{ OO } R' \)
    by (intro le relcompp-monoid order-refl)
  also have \( \ldots \le (R \text{ OO } R') \text{ OO } STU \text{ OO } (R' \text{ OO } R') \)
    unfolding relcompp-assoc ..
  also have \( \ldots \le R \text{ OO } STU \text{ OO } R' \)
    by (intro trans relcompp-monoid order-refl)
  finally show \( \text{thesis} \).
qed

lemma Grp-conversep-eq-onp: \( ((\text{BNF-Def.Grp UNIV } f)^{-1})^{-1} \text{ OO } \text{BNF-Def.Grp UNIV } f = \text{eq-onp } (\lambda x. x \in \text{range } f) \)
  by (auto simp: fun-eq-iff Grp-def eq-onp-def image-iff)

lemma Grp-conversep-nonempty: \( ((\text{BNF-Def.Grp UNIV } f)^{-1})^{-1} \text{ OO } \text{BNF-Def.Grp UNIV } f \neq \text{bot} \)
  by (auto simp: fun-eq-iff Grp-def)

lemma relcomppI2: \( r \ a \ b \implies s \ b \ c \implies t \ c \ d \implies (r \text{ OO } s \text{ OO } t) \ a \ d \)
  by (auto)

lemma rel-conj-eq-onp: \( \text{equivp } R \implies \text{rel-conj } R \text{ (eq-onp } P) \leq R \)
  by (auto simp: eq-onp-def transp-def equivp-def)

lemma Quotient-Quotient3: Quotient \( R \) Abs Rep T \( \implies \) Quotient3 \( R \) Abs Rep
  unfolding Quotient-def Quotient3-def by blast
THEORY “Complete-Partial-Order”

lemma Quotient-reflp-imp-equivp: Quotient R Abs Rep T ⇒ reflp R ⇒ equivp R
  using Quotient-symp Quotient-transp equivpI by blast

lemma Quotient-eq-onp-typedef:
  Quotient (eq-onp P) Abs Rep cr ⇒ type-definition Rep Abs {x. P x}
  unfolding Quotient-def eq-onp-def
  by unfold-locales auto

lemma Quotient-eq-onp-type-copy:
  Quotient (=) Abs Rep cr ⇒ type-definition Rep Abs UNIV
  unfolding Quotient-def eq-onp-def
  by unfold-locales auto

ML-file Tools/BNF/bnf-lift.ML

hide-fact
  sum-insert-Inl-unit lift-sum-unit-vimage-commute insert-Inl-int-map-sum-unit
  image-map-sum-unit-subset subset-lift-sum-unitD UNIV-sum-unit-conv subset-vimage-image-subset
  relcomp-mem-Grp-neq-bot comp-projr-Inr in-rel-sum-in-image-projr subset-rel-sumI
  relcompp-eq-Grp-neq-bot rel-fun-rel-OO1 rel-fun-rel-OO2 rel-sum-eq2-nonempty
  rel-sum-eq3-nonempty
  hypsubst equivp-add-relconj Grp-conversep-eq-onp Grp-conversep-nonempty rel-
  comppI2 rel-conj-eq-onp
  Quotient-reflp-imp-equivp Quotient-Quotient3

end

47 Chain-complete partial orders and their fix-points

theory Complete-Partial-Order
  imports Product-Type
begin

47.1 Monotone functions

Dictionary-passing version of mono.

definition monotone :: (′a ⇒ ′a ⇒ bool) ⇒ (′b ⇒ ′b ⇒ bool) ⇒ (′a ⇒ ′b) ⇒ bool
  where monotone orda ordb f ≡ (∀x y. orda x y ⇒ ordb (f x) (f y))

lemma monotoneI[intro?]: (∀x y. orda x y ⇒ ordb (f x) (f y)) ⇒ monotone orda ordb f
  unfolding monotone-def by iprover

lemma monotoneD[dest?]: monotone orda ordb f ⇒ orda x y ⇒ ordb (f x) (f y)
unfolding monotone-def by iprover

47.2 Chains

A chain is a totally-ordered set. Chains are parameterized over the order for maximal flexibility, since type classes are not enough.

definition chain :: ('a ⇒ 'a ⇒ bool) ⇒ 'a set ⇒ bool
  where chain ord S ←→ (∀x∈S. ∀y∈S. ord x y ∨ ord y x)

lemma chainI:
  assumes (∀x y. x ∈ S ⇒ y ∈ S ⇒ ord x y ∨ ord y x)
  shows chain ord S
  using assms unfolding chain-def by fast

lemma chainD:
  assumes chain ord S and x ∈ S and y ∈ S
  shows ord x y ∨ ord y x
  using assms unfolding chain-def by fast

lemma chainE:
  assumes chain ord S and x ∈ S and y ∈ S
  obtains ord x y | ord y x
  using assms unfolding chain-def by fast

lemma chain-empty: chain ord {} by (simp add: chain-def)

lemma chain-equality: chain (=) A ←→ (∀x∈A. ∀y∈A. x = y) by (auto simp add: chain-def)

lemma chain-subset: chain ord A ⇒ B ⊆ A ⇒ chain ord B by (rule chainI) (blast dest: chainD)

lemma chain-imageI:
  assumes chain: chain le-a Y
  and mono: (∃x y. x ∈ Y ⇒ y ∈ Y ⇒ le-a x y ⇒ le-b (f x) (f y))
  shows chain le-b (f ' Y)
  by (blast intro: chainI dest: chainD[OF chain] mono)

47.3 Chain-complete partial orders

A ccpo has a least upper bound for any chain. In particular, the empty set is a chain, so every ccpo must have a bottom element.

class ccpo = order + Sup +
  assumes ccpo-Sup-upper: chain (≤) A ⇒ x ∈ A ⇒ x ≤ Sup A
  assumes ccpo-Sup-least: chain (≤) A ⇒ (∀x. x ∈ A ⇒ x ≤ z) ⇒ Sup A ≤ z
begin
lemmas

chain-singleton: Complete-Partial-Order.chain (≤) {x}
  by (rule chainI) simp

ccpo-Sup-singleton [simp]: \{x\} = x
  by (rule antisym) (auto intro: ccpo-Sup-least ccpo-Sup-upper simp add: chain-singleton)

47.4 Transfinite iteration of a function

context notes [[inductive-internals]]
begin

inductive-set iterates :: ('a ⇒ 'a) ⇒ 'a set
  for f :: 'a ⇒ 'a
  where
    step: x ∈ iterates f ⇒ f x ∈ iterates f
    | Sup: chain (≤) M ⇒ (∀x ∈ M. x ∈ iterates f ⇒ Sup M ∈ iterates f)

end

lemma iterates-le-f: x ∈ iterates f ⇒ monotone (≤) (≤) f ⇒ x ≤ f x
  by (induct x rule: iterates.induct)
    (force dest: monotoneD intro: ccpo-Sup-upper ccpo-Sup-least)+

lemmas

chain-iterates:
  assumes f: monotone (≤) (≤) f
  shows chain (≤) (iterates f) (is chain - ?C)
proof (rule chainI)
  fix x y
  assume x ∈ ?C y ∈ ?C
  then show x ≤ y ∨ y ≤ x
    proof (induct x arbitrary: y rule: iterates.induct)
      fix x y
      assume y: y ∈ ?C
      and IH: (∀z. z ∈ ?C ⇒ x ≤ z ∨ z ≤ x)
      from y show f x ≤ y ∨ y ≤ f x
        proof (induct y rule: iterates.induct)
          case (step y)
          with IH show ?case by (auto dest: monotoneD)
        next
          case (Sup M)
          then have chM: chain (≤) M
          and IH': (∃z. z ∈ M ⇒ f x ≤ z ∨ z ≤ f x) by auto
          show f x ≤ Sup M ∨ Sup M ≤ f x
            proof (cases z ∈ M. f x ≤ z)
              case True
              then have f x ≤ Sup M
                apply rule
              apply (erule order-trans)
apply (rule ccpo-Sup-upper[OF chM])
apply assumption
done
then show ?thesis ..
next
case False
with IH' show ?thesis
  by (auto intro: ccpo-Sup-least[OF chM])
qed
qed

next
case (Sup M y)
show ?case
proof (cases ∃x∈M. y ≤ x)
case True
then have y ≤ Sup M
  apply rule
  apply (erule order-trans)
  apply (rule ccpo-Sup-upper[OF Sup(1)])
  apply assumption
  done
then show ?thesis ..
next
case False with Sup
show ?thesis by (auto intro: ccpo-Sup-least)
qed
qed

lemma bot-in-iterates: Sup {} ∈ iterates f
  by (auto intro: iterates.Sup simp add: chain-empty)

47.5 Fixpoint combinator

definition fixp :: ('a ⇒ 'a) ⇒ 'a
  where fixp f = Sup (iterates f)

lemma iterates-fixp:
  assumes f: monotone (≤) (≤) f
  shows fixp f ∈ iterates f
unfolding fixp-def
  by (simp add: iterates.Sup chain-iterates f)

lemma fixp-unfold:
  assumes f: monotone (≤) (≤) f
  shows fixp f = f (fixp f)
proof (rule antisym)
  show fixp f ≤ f (fixp f)
    by (intro iterates-le-f iterates-fixp f)
THEORY “Complete-Partial-Order”

have \( f (\text{fixp } f) \leq \text{Sup} (\text{iterates } f) \)
  by (intro ccpo-Sup-upper chain-iterates f iterates.step iterates-fixp)
then show \( f (\text{fixp } f) \leq \text{fixp } f \)
  by (simp only: fixp-def)
qed

lemma fixp-lowerbound:
  assumes \( f : \text{monotone } (\leq) (\leq) f \)
  and \( z : f z \leq z \)
  shows \( \text{fixp } f \leq z \)
unfolding fixp-def
proof (rule ccpo-Sup-least[OF chain-iterates[OF f]])
  fix \( x \)
  assume \( x \in \text{iterates } f \)
  then show \( x \leq z \)
  proof (induct \( x \) rule: iterates.induct)
    case (step \( x \))
    from \( f \langle x \leq z \rangle \) have \( f x \leq f z \) by (rule monotoneD)
    also note \( z \)
    finally show \( f x \leq z \).
  next
  case (Sup \( M \))
  then show \?case
    by (auto intro: ccpo-Sup-least)
qend
qend

47.6 Fixpoint induction

setup ⟨Sign.map-naming (Name-Space.mandatory-path ccpo)⟩

definition admisible :: (‘a set ⇒ ‘a) ⇒ (‘a ⇒ ‘a ⇒ bool) ⇒ (‘a ⇒ bool) ⇒ bool
  where admisible lub ord P ←→ (∀ A. chain ord A =⇒ A ≠ {} =⇒ (∀ x∈A. P x) =⇒ P (lub A))

lemma admisibleI:
  assumes \( \forall A. \text{chain ord } A =⇒ A ≠ {} \) =⇒ \( \forall x\in A. P x =⇒ P (\text{lub } A) \)
  shows ccpo.admissible lub ord P
using assms unfolding ccpo.admissible-def by fast

lemma admisibleD:
  assumes ccpo.admissible lub ord P
  assumes chain ord A
  assumes \( A ≠ {} \)
  assumes \( \forall x. x \in A =⇒ P x \)
  shows \( P (\text{lub } A) \)
using assms by (auto simp: ccpo.admissible-def)
setup (Sign.map-naming Name-Space.parent-path)

lemma (in ccpo) fixp-induct:
assumes adm: ccpo.admissible Sup (\leq) P
assumes mono: monotone (\leq) (\leq) f
assumes bot: P (Sup \{\})
assumes step: \forall x. P x \implies P (f x)
shows P (fixp f)
unfolding fixp-def
using adm chain-iterates[OF mono]
proof (rule ccpo.admissibleD)
  show iterates f \neq {}
    using bot-in-iterates by auto
next
fix x
assume x \in iterates f
then show P x
proof (induct rule: iterates.induct)
  case prems: (step x)
  from this(2) show ?case by (rule step)
next
  case (Sup M)
  then show ?case by (cases M = {}) (auto intro: step bot ccpo.admissibleD adm)
qd
qed

lemma admissible-True: ccpo.admissible lub ord (\lambda x. True)
unfolding ccpo.admissible-def by simp

lemma admissible-const: ccpo.admissible lub ord (\lambda x. t)
by (auto intro: ccpo.admissibleI)

lemma admissible-conj:
assumes ccpo.admissible lub ord (\lambda x. P x)
assumes ccpo.admissible lub ord (\lambda x. Q x)
sshows ccpo.admissible lub ord (\lambda x. P x \land Q x)
using assms unfolding ccpo.admissible-def by simp

lemma admissible-all:
assumes \forall y. ccpo.admissible lub ord (\lambda x. P x y)
sshows ccpo.admissible lub ord (\lambda x. \forall y. P x y)
using assms unfolding ccpo.admissible-def by fast

lemma admissible-ball:
assumes \forall y. y \in A \implies ccpo.admissible lub ord (\lambda x. P x y)
sshows ccpo.admissible lub ord (\lambda x. \forall y \in A. P x y)
using assms unfolding ccpo.admissible-def by fast

lemma chain-compr: chain ord A ⇒ chain ord \{x ∈ A. P x\}
  unfolding chain-def by fast

color context ccpo
begin

lemma admissible-disj:
  fixes P Q :: 'a ⇒ bool
  assumes P: ccpo.admissible Sup (≤) (λx. P x)
  assumes Q: ccpo.admissible Sup (≤) (λx. Q x)
  shows ccpo.admissible Sup (≤) (λx. P x ∨ Q x)
proof (rule ccpo.admissibleI)
  fix A :: 'a set
  assume chain: chain (≤) A
  assume A: A ≠ {} and P-Q: ∀x∈A. P x ∨ Q x
  have (∃x∈A. P x) ∧ (∀x∈A. ∃y∈A. x ≤ y ∧ P y) ∨ (∃x∈A. Q x) ∧ (∀x∈A. ∃y∈A. x ≤ y ∧ Q y)
    (is ?P ∨ ?Q is ?P1 ∧ ?P2 ∨ -)
proof (rule disjCI)
  assume ¬ ?Q
  then consider ∀x∈A. ¬ Q x | a where a ∈ A ∀y∈A. a ≤ y → ¬ Q y
    by blast
  then show ?P
proof cases
  case 1
    with P-Q have ∀x∈A. P x by blast
    with A show ?P by blast
  next
  case 2
  note a = ⟨a ∈ A⟩
  show ?P
proof
  from P-Q 2 have *: ∀y∈A. a ≤ y → P y by blast
  with a have P a by blast
  with a show ?P1 by blast
  show ?P2
proof
  fix x
  assume x: x ∈ A
  with chain a show ∃y∈A. x ≤ y ∧ P y
proof (rule chainE)
  assume le: a ≤ x
  with * a x have P x by blast
  with x le show ?thesis by blast
  next
  assume a ≥ x
  with a ⟨P a⟩ show ?thesis by blast
moreover
have $\text{Sup } A = \text{Sup } \{x \in A. P x\}$ if $\forall x \in A. \exists y \in A. x \leq y \land P y$ for $P$
proof (rule antisym)
  have chain-P; chain ($\leq$) $\{x \in A. P x\}$
    by (rule chain-compr [OF chain])
  show $\text{Sup } A \leq \text{Sup } \{x \in A. P x\}$
    apply (rule ccpo-Sup-least [OF chain])
    apply clarify
    apply (erule order-trans)
    apply (simp add: ccpo-Sup-upper [OF chain-P])
    done
  show $\{x \in A. P x\} \leq \text{Sup } A$
    apply (rule ccpo-Sup-least [OF chain-P])
    apply clarify
    apply (simp add: ccpo-Sup-upper [OF chain])
    done
qed
ultimately
consider $\exists x. x \in A \land P x \text{ Sup } A = \text{Sup } \{x \in A. P x\}$
| $\exists x. x \in A \land Q x \text{ Sup } A = \text{Sup } \{x \in A. Q x\}$
by blast
then show $P (\text{Sup } A) \lor Q (\text{Sup } A)$
  apply cases
  apply simp-all
  apply (rule disjI1)
  apply (rule ccpo.admissibleD [OF P chain-compr [OF chain]]; simp)
  apply (rule disjI2)
  apply (rule ccpo.admissibleD [OF Q chain-compr [OF chain]]; simp)
  done
qed

end

instance complete-lattice $\subseteq$ ccpo
  by standard (fast intro: Sup-upper Sup-least)+

lemma lfp-eq-fixp:
  assumes mono: mono $f$
  shows $lfp f = fixp f$
proof (rule antisym)
  from mono have $f^\prime$: monotone ($\leq$) ($\leq$) $f$
    unfolding mono-def monotone-def .
  show $lfp f \leq fixp f$

by (rule fixp-lowerbound, subst fixp-unfold [OF f'], rule order-refl)
show fixp f ≤ lfp f
  by (rule fixp-lowerbound [OF f']) (simp add: lfp-fixpoint [OF mono])
qed

hide-const (open) iterates fixp

declare fixp

48 Datatype option

theory Option
  imports Lifting
begin

datatype 'a option =
  None
| Some (the: 'a)

datatype-compat option

lemma [case-names None Some, cases type: option]:
  — for backward compatibility - names of variables differ
  (y = None ⇒ P) ⇒ (∀ a. y = Some a ⇒ P) ⇒ P
  by (rule option.exhaust)

lemma [case-names None Some, induct type: option]:
  — for backward compatibility - names of variables differ
  P None ⇒ (∀ option. P (Some option)) ⇒ P option
  by (rule option.induct)

Compatibility:
setup ⟨Sign.mandatory-path option⟩
lemmas inducts = option.induct
lemmas cases = option.case
setup ⟨Sign.parent-path⟩

lemma not-None-eq [iff]: x ≠ None ↔ (∃ y. x = Some y)
  by (induct x) auto

lemma not-Some-eq [iff]: (∀ y. x ≠ Some y) ↔ x = None
  by (induct x) auto

lemma comp-the-Some[simp]: the o Some = id
  by auto

Although it may appear that both of these equalities are helpful only when
applied to assumptions, in practice it seems better to give them the uniform
iff attribute.
THEORY "Option"

lemma inj-Some [simp]: inj-on Some A
  by (rule inj-onI) simp

lemma case-optionE:
  assumes c: (case x of None ⇒ P | Some y ⇒ Q y)
  obtains (None) x = None and P
         | (Some) y where x = Some y and Q y
  using c by (cases x) simp-all

lemma split-option-all: (∀ x. P x) ↔ P None ∧ (∀ x. P (Some x))
  by (auto intro: option.induct)

lemma split-option-ex: (∃ x. P x) ↔ P None ∨ (∃ x. P (Some x))
  using split-option-all[of λx. ¬ P x] by blast

lemma UNIV-option-conv: UNIV = insert None (range Some)
  by (auto intro: classical)

lemma rel-option-None1 [simp]: rel-option P None x ↔ x = None
  by (cases x) simp-all

lemma rel-option-None2 [simp]: rel-option P x None ↔ x = None
  by (cases x) simp-all

lemma option-rel-Some1: rel-option A (Some x) y ↔ (∃ y'. y = Some y' ∧ A x y')
  by (cases y) simp-all

lemma option-rel-Some2: rel-option A x (Some y) ↔ (∃ x'. x = Some x' ∧ A x' y)
  by (cases x) simp-all

lemma rel-option-inf: inf (rel-option A) (rel-option B) = rel-option (inf A B)
  (is ?lhs = ?rhs)
proof (rule antisym)
  show ?lhs ≤ ?rhs by (auto elim: option.rel-cases)
  show ?rhs ≤ ?lhs by (auto elim: option.rel-mono-strong)
qed

lemma rel-option-reflI:
  (∀ x ∈ set-option y ⇒ P x x) ⇒ rel-option P y y
  by (cases y) auto

48.0.1 Operations

lemma ospec [dest]: (∀ x∈set-option A. P x) ⇒ A = Some x ⇒ P x
  by simp
setup ⟨map-theory-claset (fn ctxt => ctxt addSD2 (ospec, @\{thm ospec\}))⟩

lemma elem-set [iff]: (x ∈ set-option xo) = (xo = Some x)
  by (cases xo) auto

lemma set-empty-eq [simp]: (set-option xo = {}) = (xo = None)
  by (cases xo) auto

lemma map-option-case: map-option f y = (case y of None ⇒ None | Some x ⇒ Some (f x))
  by (auto split: option.split)

lemma map-option-is-None [iff]: (map-option f opt = None) = (opt = None)
  by (simp add: map-option-case split: option.split)

lemma None-eq-map-option-iff: None = map-option f x ⇐⇒ x = None
  by (cases x) simp-all

lemma map-option-eq-Some [iff]: (map-option f xo = Some y) = (∃z. xo = Some z ∧ f z = y)
  by (simp add: map-option-case split: option.split)

lemma map-option-o-case-sum [simp]:
  map-option f ∘ case-sum g h = case-sum (map-option f ∘ g) (map-option f ∘ h)
  by (rule o-case-sum)

lemma map-option-cong: x = y ⇒ (∀a. y = Some a ⇒ f a = g a) ⇒ map-option f x = map-option g y
  by (cases x) auto

lemma map-option-idI: (∀y. y ∈ set-option x ⇒ f y = y) ⇒ map-option f x = x
  by (cases x) (simp-all)

functor map-option: map-option
  by (simp-all add: option.map-comp fun-iff option.map-id)

lemma case-map-option [simp]: case-option g h (map-option f x) = case-option g (h ∘ f) x
  by (cases x) simp-all

lemma None-notin-image-Some [simp]: None ∉ Some · A
  by auto

lemma notin-range-Some: x ∉ range Some ⇐⇒ x = None
  by (cases x) auto

lemma rel-option-iff:
rel-option \( R x y = \text{(case } (x, y) \text{ of } (\text{None, None}) \Rightarrow \text{True} \) \\
\text{| (Some x, Some y) } \Rightarrow R x y \\
\text{| - } \Rightarrow \text{False} \)

by (auto split: prod.split option.split)

\text{definition} combine-options :: 'a option \Rightarrow 'a option \Rightarrow 'a option \Rightarrow 'a option \\
\text{where} combine-options f x y = \\
\text{(case } x \text{ of } \text{None } \Rightarrow \text{y } | \text{Some } x \Rightarrow \text{(case } y \text{ of } \text{None } \Rightarrow \text{Some } x \text{ | Some } y \Rightarrow \text{Some } (f x y))\)

\text{lemma} combine-options-simps [simp]:
combine-options f None y = y  
combine-options f x None = x  
combine-options f (Some a) (Some b) = Some (f a b)  
by (simp-all add: combine-options-def split: option.splits)

\text{lemma} combine-options-cases [case-names None1 None2 Some]:
\text{(x = None } \Rightarrow P x y) \Rightarrow (y = None } \Rightarrow P x y) \Rightarrow 
\text{(} \forall a \text{ b. } x = \text{Some } a \Rightarrow y = \text{Some } b \Rightarrow P x y) \Rightarrow P x y  
by (cases x; cases y) simp-all

\text{lemma} combine-options-commute:
(\forall x y. f x y = f y x) \Rightarrow combine-options f x y = combine-options f y x 
using combine-options-cases[of x]  
by (induction x y rule: combine-options-cases) simp-all

\text{lemma} combine-options-assoc:
(\forall x y z. f (f x y) z = f x (f y z)) \Rightarrow 
combine-options f (combine-options f x y) z = 
combine-options f x (combine-options f y z)  
by (auto simp: combine-options-def split: option.splits)

\text{lemma} combine-options-left-commute:
(\forall x y. f x y = f y x) \Rightarrow (\forall x y z. f (f x y) z = f x (f y z)) \Rightarrow 
combine-options f y (combine-options f x z) = 
combine-options f x (combine-options f y z)  
by (auto simp: combine-options-def split: option.splits)

\text{lemmas} combine-options-ac = 
combine-options-commute combine-options-assoc combine-options-left-commute

\text{context}
\text{begin}

\text{qualified definition} is-none :: 'a option \Rightarrow bool \\
\text{where} [\text{code-post}]: \text{is-none } x \leftrightarrow x = \text{None}
lemma is-none-simps [simp]:
  is-none None
  \neg is-none (Some x)
by (simp-all add: is-none-def)

lemma is-none-code [code]:
  is-none None = True
  is-none (Some x) = False
by simp-all

lemma rel-option-unfold:
  rel-option R x y \longleftrightarrow
  (is-none x \longleftrightarrow is-none y) \land (\neg is-none x \longrightarrow \neg is-none y \longrightarrow R (the x) (the y))
by (simp add: rel-option-iff split: option.split)

lemma rel-optionI:
[ [ is-none x \longleftrightarrow is-none y; [ \neg is-none x; \neg is-none y ] \Longrightarrow P (the x) (the y) ]
\Longrightarrow rel-option P x y
by (simp add: rel-option-unfold)

lemma is-none-map-option [simp]: is-none (map-option f x) \longleftrightarrow is-none x
by (simp add: is-none-def)

lemma the-map-option: \neg is-none x \Longrightarrow the (map-option f x) = f (the x)
by (auto simp add: is-none-def)

qualified primrec bind :: 'a option \Rightarrow ('b ⇒ option) \Rightarrow 'b option
where
  bind-lzero: bind None f = None
  | bind-lunit: bind (Some x) f = f x

lemma is-none-bind: is-none (bind f g) \longleftrightarrow is-none f \lor is-none (g (the f))
by (cases f) simp-all

lemma bind-runit[simp]: bind x Some = x
by (cases x) auto

lemma bind-assoc[simp]: bind (bind x f) g = bind x (\lambda y. bind (f y) g)
by (cases x) auto

lemma bind-rzero[simp]: bind x (\lambda x. None) = None
by (cases x) auto

qualified lemma bind-cong: x = y \Longrightarrow (\forall a. y = Some a \Longrightarrow f a = g a) \Longrightarrow
bind x f = bind y g
by (cases x) auto
lemma bind-split: \( P (\text{bind} \ m \ f) \leftrightarrow (m = \text{None} \implies P \text{None}) \land (\forall v. \ m = \text{Some} \ v \implies P (f v)) \)
  by (cases m) auto

lemma bind-split-asn: \( P (\text{bind} \ m \ f) \leftrightarrow \neg (m = \text{None} \land \neg P \text{None} \lor (\exists x. \ m = \text{Some} \ x \land \neg P (f x))) \)
  by (cases m) auto

lemmas bind-splits = bind-split bind-split-asn

lemma bind-eq-None-conv: \( \text{Option}.\text{bind} \ a \ f = \text{None} \leftrightarrow a = \text{None} \lor f (\text{the} \ a) = \text{None} \)
  by (cases a) simp-all

lemma map-option-bind: \( \text{map-option} \ f (\text{bind} \ x \ g) = \text{bind} \ x (\text{map-option} \circ g) \)
  by (cases x) simp-all

lemma bind-option-cong:
  \[
  \[ x = y; \forall z. z \in \text{set-option} y \implies f z = g z \] \implies \text{bind} x f = \text{bind} y g
  \]
  by (cases y) simp-all

lemma bind-option-cong-simp:
  \[
  \[ x = y; \forall z. z \in \text{set-option} y =\text{simp=}\implies f z = g z \] \implies \text{bind} x f = \text{bind} y g
  \]
  unfolding simp-implies-def by (rule bind-option-cong)

lemma bind-option-cong-code: \( x = y \implies \text{bind} x f = \text{bind} y f \)
  by simp

lemma bind-map-option: \( \text{bind} \ (\text{map-option} f x) g = \text{bind} x (g \circ f) \)
  by (cases x) simp-all

lemma set-bind-option [simp]: \( \text{set-option} (\text{bind} x f) = (\bigcup ((\text{set-option} \circ f) \cdot \text{set-option} x)) \)
  by (cases x) auto

lemma map-conv-bind-option: \( \text{map-option} f x = \text{Option}.\text{bind} x (\text{Some} \circ f) \)
  by (cases x) simp-all

end

setup (Code-Simp.map-sls (Simplifier.add-cong @{thm bind-option-cong-code}));

case
begin
qualified definition these :: 'a option set ⇒ 'a set
  where these A = the ′{x ∈ A. x ≠ None}′

lemma these-empty [simp]: these {} = {}
  by (simp add: these-def)

lemma these-insert-None [simp]: these (insert None A) = these A
  by (auto simp add: these-def)

lemma these-insert-Some [simp]: these (insert (Some x) A) = insert x (these A)
proof –
  have {y ∈ insert (Some x) A. y ≠ None} = insert (Some x) {y ∈ A. y ≠ None}
    by auto
  then show ?thesis by (simp add: these-def)
qed

lemma in-these-eq: x ∈ these A ←→ Some x ∈ A
proof
  assume Some x ∈ A
  then obtain B where A = insert (Some x) B by auto
  then show x ∈ these A by (auto simp add: these-def intro!: image-eqI)
next
  assume x ∈ these A
  then show Some x ∈ A by (auto simp add: these-def)
qed

lemma these-image-Some-eq [simp]: these (Some ' A) = A
  by (auto simp add: these-def intro!: image-eqI)

lemma Some-image-these-eq: Some ' these A = {x ∈ A. x ≠ None}
  by (auto simp add: these-def image-image intro!: image-eqI)

lemma these-empty-eq: these B = {} ←→ B = {} ∨ B = {None}
  by (auto simp add: these-def)

lemma these-not-empty-eq: these B ≠ {} ←→ B ≠ {} ∧ B ≠ {None}
  by (auto simp add: these-empty-eq)
end

lemma finite-range-Some: finite (range (Some :: 'a ⇒ 'a option)) = finite (UNIV :: 'a set)
  by (auto dest: finite-imageD intro: inj-Some)

48.1 Transfer rules for the Transfer package
context includes lifting-syntax
begin
lemma option-bind-transfer [transfer-rule]:
(rel-option A ===> (A ===> rel-option B) ===> rel-option B)
  Option.bind Option.bind
unfolding rel-fun-def split-option-all by simp

lemma pred-option-parametric [transfer-rule]:
((A ===> (=)) ===> rel-option A ===> (=)) pred-option pred-option
by (rule rel-funI)+ (auto simp add: rel-option-unfold Option.is-none-def dest: rel-funD)
end

48.1.1 Interaction with finite sets

lemma finite-option-UNIV [simp]:
finite (UNIV :: 'a option set) = finite (UNIV :: 'a set)
by (auto simp add: UNIV-option-conv elim: finite-imageD intro: inj-Some)

instance option :: (finite) finite
by standard (simp add: UNIV-option-conv)

48.1.2 Code generator setup

lemma equal-None-code-unfold [code-unfold]:
HOL.equal x None ←→ Option.is-none x
HOL.equal None = Option.is-none
by (auto simp add: equal Option.is-none-def)

code-printing
  type-constructor option →
    (SML) - option
    and (OCaml) - option
    and (Haskell) Maybe -
    and (Scala) !Option[(=)]
  | constant None →
    (SML) NONE
    and (OCaml) None
    and (Haskell) Nothing
    and (Scala) !None
  | constant Some →
    (SML) SOME
    and (OCaml) Some -
    and (Haskell) Just
    and (Scala) Some
  | class-instance option :: equal →
    (Haskell) -
  | constant HOL.equal :: 'a option ⇒ 'a option ⇒ bool →
    (Haskell) infix 4 ==
49 Partial Function Definitions

theory Partial-Function
  imports Complete-Partial-Order Option
  keywords partial-function :: thy-defn
begin

named-theorems partial-function-mono monotonicity rules for partial function definitions
ML-file ⟨Tools/Function/partial-function.ML⟩

lemma (in ccpo) in-chain-finite:
  assumes Complete-Partial-Order.chain (≤) A finite A A ≠ {}
  shows ⨆ A ∈ A
  using assms(2,1,3)
proof induction
  case empty thus ?case by simp
next
  case (insert x A)
  note chain = ⟨Complete-Partial-Order.chain (≤) (insert x A)⟩
  show ?case
  proof (cases A = {})
    case True thus ?thesis by simp
  next
    case False
    from chain have chain': Complete-Partial-Order.chain (≤) A
      by (rule chain-subset) blast
    hence ⨆ A ∈ A using False by (rule insert.IH)
    show ?thesis
    proof (cases x ≤ ⨆ A)
      case True
      have ⨆ (insert x A) ≤ ⨆ A using chain
        by (rule ccpo-Sup-upper)(auto simp add: True intro: ccpo-Sup-upper[OF chain'])
      hence ⨆ (insert x A) = ⨆ A
        by (rule antisym)(blast intro: ccpo-Sup-upper[OF chain] ccpo-Sup-least[OF chain'])
      with ⨆ A ∈ A show ?thesis by simp
THEORY "Partial-Function"

next

*case True*

with chainD[OF chain, of x \ U A] | U A \in A

have U(insert x A) = x


thus \ ?thesis by simp

qed

qed

lemma (in ccpo) admissible-chfin:

(\forall S. Complete-Partial-Order.chain (\leq) S \rightarrow finite S)

\implies ccpo.admissible Sup (\leq) P

using in-chain-finite by (blast intro: ccpo.admissibleI)

49.1 Aximatic setup

This technical locale constains the requirements for function definitions with ccpo fixed points.

**Definition** fun-ord ord f g \iff (\forall x. ord(f x) (g x))

**Definition** fun-lub L A = (\lambda x. L \{ y. \exists f \in A. y = f x \})

**Definition** img-ord f ord = (\lambda x y. ord(f x) (f y))

**Definition** img-lub f g Lub = (\lambda A. g (Lub(f A)))

lemma (in ccpo) chain-fun:

assumes A: chain (fun-ord ord) A

shows chain ord \{ y. \exists f \in A. y = f a \} is chain ord \ ?C

proof (rule chainI)

fix x y assume x \in \ ?C y \in \ ?C

then obtain f g where fg: f \in A g \in A

and \ [simp]: x = f a y = g a by blast

from chainD[OF A fg]

show ord x y \lor ord y x unfolding fun-ord-def by auto

qed

lemma (in ccpo) call-mono[partial-function-mono]: monotone (fun-ord ord) ord (\lambda f. f t)

by (rule monotoneI) (auto simp: fun-ord-def)

lemma (in ccpo) let-mono[partial-function-mono]:

(\forall x. monotone orda ordb (\lambda f. b f x))

\implies monotone orda ordb (\lambda f. Let t (b f))

by (simp add: Let-def)

lemma (in ccpo) if-mono[partial-function-mono]: monotone orda ordb F

\implies monotone orda ordb G

\implies monotone orda ordb (\lambda f. if c then F f else G f)

unfolding monotone-def by simp
definition mk-less $R = (\lambda x\ y. R\ x\ y \land \neg R\ y\ x)$

locale partial-function-definitions =
  fixes $\text{leq} :: 'a \Rightarrow 'a \Rightarrow \text{bool}$
  fixes $\text{lub} :: 'a\ set \Rightarrow 'a$
  assumes leq-refl: $\text{leq} x\ x$
  assumes leq-trans: $\text{leq} x\ y$ $\Longrightarrow$ $\text{leq} y\ z$ $\Longrightarrow$ $\text{leq} x\ z$
  assumes leq-antisym: $\text{leq} x\ y$ $\Longrightarrow$ $\text{leq} y\ x$ $\Longrightarrow$ $x = y$
  assumes lub-upper: chain $\text{leq} A$ $\Longrightarrow$ $x \in A$ $\Longrightarrow$ $\text{leq} x\ (\text{lub} A)$
  assumes lub-least: chain $\text{leq} A$ $\Longrightarrow$ $(\forall x.\ x \in A \Longrightarrow \text{leq} x\ z)$ $\Longrightarrow$ $\text{leq} (\text{lub} A)\ z$

lemma partial-function-lift:
  assumes partial-function-definitions ord lb
  shows partial-function-definitions $\text{fun-ord ord}$ $\text{fun-lub lb}$ (is partial-function-definitions $\text{ordf}$ $\text{lubf}$)
  proof –
    interpret partial-function-definitions ord lb by fact
    show $\text{thesis}$
    proof
      fix $x$ show $\text{ordf}\ x\ x$
        unfolding $\text{fun-ord-def}$ by (auto simp: leq-refl)
    next
      fix $x\ y\ z$ assume $\text{ordf}\ x\ y$ $\text{ordf}\ y\ z$
      thus $\text{ordf}\ x\ z$
        unfolding $\text{fun-ord-def}$
        by (force dest: leq-trans)
    next
      fix $x\ y$ assume $\text{ordf}\ x\ y$ $\text{ordf}\ y\ x$
      thus $x = y$
        unfolding $\text{fun-ord-def}$
        by (force intro!: dest: leq-antisym)
    next
      fix $A\ f$ assume $f: f \in A$ and $A$: chain $\text{ordf} A$
      thus $\text{ordf} f$ ($\text{lubf} A$)
        unfolding $\text{fun-lub-def}$ $\text{fun-ord-def}$
        by (blast intro: lub-upper chain-fun[OF $A$] $f$)
    next
      fix $A :: ('b \Rightarrow 'a)$ set and $g :: 'b \Rightarrow 'a$
      assume $A$: chain $\text{ordf} A$ and $g$: $\forall f.\ f \in A$ $\Longrightarrow$ $\text{ordf} f\ g$
      show $\text{ordf}$ ($\text{lubf} A$) $g$ unfolding $\text{fun-lub-def}$ $\text{fun-ord-def}$
        by (blast intro: lub-least chain-fun[OF $A$] dest: $g[$unfolded fun-ord-def$])
    qed
  qed

lemma ccpo: assumes partial-function-definitions ord lb
  shows class.ccpo lb ord (mk-less ord)
  using assms unfolding partial-function-definitions-def mk-less-def
  by unfold-locales blast+

lemma partial-function-image:
assumes partial-function-definitions ord Lub
assumes inj: \(\forall x, y. f x = f y \Rightarrow x = y\)
assumes inv: \(\forall x. f (g x) = x\)
shows partial-function-definitions (img-ord f ord) (img-lub f g Lub)
proof –
let \(?iord\) = img-ord f ord
let \(?ilub\) = img-lub f g Lub
interpret partial-function-definitions ord Lub by fact
show \(\?thesis\)
proof
  fix A x assume chain \(?iord\) A x \(\in\) A
  then have chain ord \((f ' A) f x \in f ' A\)
  by (auto simp: img-ord-def intro: chainI dest: chainD)
  thus \(?iord\) x \((?ilub\) A)
  unfolding inv img-lub-def img-ord-def by (rule lub-upper)
next
  fix A x assume chain \(?iord\) A
  and \(1\): \(\forall z. z \in A \Rightarrow \?iord\) z x
  then have chain ord \((f ' A)\)
  by (auto simp: img-ord-def intro: chainI dest: chainD)
  thus \(?iord\) \((?ilub\) A) x
  unfolding inv img-lub-def img-ord-def
  by (rule lub-least) (auto dest: \(1[\text{unfolded img-ord-def}]\))
qed (auto simp: img-ord-def intro: leq-refl dest: leq-trans leq-antisym inj)
qed

context partial-function-definitions begin

abbreviation le-fun \(\equiv\) fun-ord leq
abbreviation lub-fun \(\equiv\) fun-lub lub
abbreviation fixp-fun \(\equiv\) ccpo.fixp lub-fun le-fun
abbreviation mono-body \(\equiv\) monotone le-fun leq
abbreviation admissible \(\equiv\) ccpo.admissible lub-fun le-fun

Interpret manually, to avoid flooding everything with facts about orders

lemma ccpo: class ccpo lub-fun le-fun (mk-less le-fun)
apply (rule ccpo)
apply (rule partial-function-lift)
apply (rule partial-function-definitions-axioms)
done

The crucial fixed-point theorem

lemma mono-body-fixp:
  \(\forall x. \text{mono-body} (\lambda f. F x) \Rightarrow \text{fixp-fun} F = F (\text{fixp-fun} F)\)
by (rule ccpo.fixp-unfold[OF ccpo]) (auto simp: monotone-def fun-ord-def)

Version with curry/uncurry combinators, to be used by package
lemma fixp-rule-uc:
fixes $F :: \mathcal{C} \Rightarrow \mathcal{C}$ and
$U :: \mathcal{C} \Rightarrow \mathcal{B} \Rightarrow \mathcal{A}$ and
$C :: (\mathcal{B} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{C}$
assumes mono: $\forall x. \text{mono-body} (\lambda f. U (F (C f)) x)$
assumes eq: $f \equiv C \circ \text{fixp-fun} (\lambda f. U (F (C f))))$
assumes inverse: $\forall f. C (U f) = f$
shows $f = F f$
proof
have $f = C \circ \text{fixp-fun} (\lambda f. U (F (C f))))$ by (simp add: eq)
also have $... = C (U (F (C \circ \text{fixp-fun} (\lambda f. U (F (C f))))))$
bysubst mono-body-fixp[of $f$. U (F (C f))], OF mono] (rule refl)
also have $... = F (C \circ \text{fixp-fun} (\lambda f. U (F (C f))))$ by (rule inverse)
also have $... = F f$ by (simp add: eq)
finally show $f = F f$.
qed

Fixpoint induction rule

lemma fixp-induct-uc:
fixes $F :: \mathcal{C} \Rightarrow \mathcal{C}$
and $U :: \mathcal{C} \Rightarrow \mathcal{B} \Rightarrow \mathcal{A}$
and $C :: (\mathcal{B} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{C}$
and $P :: (\mathcal{B} \Rightarrow \mathcal{A}) \Rightarrow \text{bool}$
assumes mono: $\forall x. \text{mono-body} (\lambda f. U (F (C f)) x)$
and eq: $f \equiv C \circ \text{fixp-fun} (\lambda f. U (F (C f))))$
and inverse: $\forall f. C (U f) = f$
and adm: ccpo.admissible lub-fun le-fun P
and bot: $P (\lambda - . \text{lub} \{\})$
and step: $\forall f. P (U f) \Rightarrow P (U (F f))$
shows $P (U f)$
unfolding eq inverse
apply (rule ccpo.fixp-induct[OF ccpo adm])
apply (insert mono, auto simp: monotone-def fun-ord-def bot fun-lub-def)[2]
apply (rule_tac f5=C x in step)
apply (simp add: inverse)
done

Rules for mono-body:

lemma const-mono[partial-function-mono]: monotone ord leq (\lambda f. c)
by (rule monotoneI) (rule leq-refl)

end

49.2 Flat interpretation: tailrec and option

definition flatOrd b x y \iff x = b \lor x = y

definition
\text{lemma flat-interpretation:}
\text{partial-function-definitions (flat-ord b) (flat-lub b)}
\text{proof}
\text{fix } A \ x \ \text{assume } 1: \text{chain (flat-ord b)} \ A \ x \in A
\text{show flat-ord b} \ x \ (\text{flat-lub b} \ A)
\text{proof cases}
\text{assume } x = b
\text{thus } \neg \text{thesis by (simp add: flat-ord-def)}
\text{next}
\text{assume } x \neq b
\text{with } 1 \text{ have } A - \{b\} = \{x\}
\text{by (auto elim: chainE simp: flat-ord-def)}
\text{then have flat-lub b} \ A = x
\text{by (auto simp: flat-lub-def)}
\text{thus } \neg \text{thesis by (auto simp: flat-ord-def)}
\text{qed}
\text{next}
\text{fix } A \ z \ \text{assume } A: \text{chain (flat-ord b)} \ A
\text{and } z: \bigwedge x. x \in A \implies \text{flat-ord b} \ x \ z
\text{show flat-ord b} (\text{flat-lub b} \ A) \ z
\text{proof cases}
\text{assume } A \subseteq \{b\}
\text{thus } \neg \text{thesis by (auto simp: flat-lub-def flat-ord-def)}
\text{next}
\text{assume } nb: \neg A \subseteq \{b\}
\text{then obtain } y \text{ where } y: y \in A y \neq b \text{ by auto}
\text{with } A \text{ have } A - \{b\} = \{y\}
\text{by (auto elim: chainE simp: flat-ord-def)}
\text{with } nb \text{ have flat-lub b} \ A = y
\text{by (auto simp: flat-lub-def)}
\text{with } z \ y \text{ show } \neg \text{thesis by auto}
\text{qed}
\text{qed (auto simp: flat-ord-def)}

\text{lemma flat-ordI: } (x \neq a \implies x = y) \implies \text{flat-ord a} \ x \ y
\text{by (auto simp add: flat-ord-def)}

\text{lemma flat-ord-antisym: } [\text{flat-ord a} \ x \ y; \text{flat-ord a} \ y \ x] \implies x = y
\text{by (auto simp add: flat-ord-def)}

\text{lemma antisymp-flat-ord: antisymp (flat-ord a)}
\text{by (rule antisympI) (auto dest: flat-ord-antisym)}

\text{interpretation tailrec:}
\text{partial-function-definitions flat-ord undefined flat-lub undefined}
\text{rewrites flat-lub undefined } \emptyset \equiv \text{undefined}
by (rule flat-interpretation)(simp add: flat-lub-def)

interpretation option:
  partial-function-definitions flat-ord None flat-lub None
rewrites flat-lub None \{\} ≡ None
by (rule flat-interpretation)(simp add: flat-lub-def)

abbreviation tailrec-ord ≡ flat-ord undefined
abbreviation mono-tailrec ≡ monotone (fun-ord tailrec-ord) tailrec-ord

lemma tailrec-admissible:
  ccpo.admissible (fun-lub (flat-lub c)) (fun-ord (flat-ord c))
  (\x. \. a x ≠ c → P x (a x))
proof (intro ccpo.admissibleI strip)
  fix A x
  assume chain: Complete-Partial-Order.chain (fun-ord (flat-ord c)) A
  and P [rule-formal]: \f \in A. \. f x ≠ c → P x (f x)
  and defined: fun-lub (flat-lub c) A x ≠ c
  from defined obtain f where f: f \in A f x ≠ c
  hence P x (f x) by (rule P)
moreover from chain f have \A. f' x = c \or f' x = f x
  by (auto 4 4 simp add: Complete-Partial-Order.chain-def flat-ord-def fun-ord-def)
  using f by (auto simp add: fun-lub-def flat-lub-def)
ultimately show P x (fun-lub (flat-lub c) A x)
proof
  have \A. U f x = y → y ≠ c → P x y
    by (rule partial-function-definitions.fixp-induct-uc[OF flat-interpretation, of U F C, OF mono eq inverse2])
(auto intro: step tailrec-admissible simp add: fun-lub-def flat-lub-def)

thus thesis using result defined by blast
qed

lemma admissible-image:
assumes pfun: partial-function-definitions le lub
assumes adm: ccpo.admissible lub le (P o g)
assumes inj: \( \forall x y. f x = f y \Rightarrow x = y \)
assumes inv: \( \forall x. f (g x) = x \)
shows ccpo.admissible (img-lub f g lub) (img-ord f le) P
proof (rule ccpo.admissibleI)
  fix A assume chain (img-ord f le) A
  then have ch': chain le (f ' A)
    by (auto simp: img-ord-def intro: chainI dest: chainD)
  assume A \( \neq \{\} \)
  assume P-A: \( \forall x \in A. P x \)
  have (P o g) (lub (f ' A)) using adm ch'
  proof (rule ccpo.admissibleD)
    fix x assume x \( \in f ' A \)
    with P-A show (P o g) x by (auto simp: inj[OF inv])
  qed simp add: \( \{\} \neq \{\} \)
  thus P (img-lub f g lub A) unfolding img-lub-def by simp
qed

lemma admissible-fun:
assumes pfun: partial-function-definitions le lub
assumes adm: \( \forall x. ccpo.admissible lub le (Q x) \)
shows ccpo.admissible (fun-lub lub) (fun-ord le) (\( \lambda f. \forall x. Q x (f x) \))
proof (rule ccpo.admissibleI)
  fix A :: (\( \forall b \Rightarrow a \)) set
  assume Q: \( \forall f \in A. \forall x. Q x (f x) \)
  assume ch: chain (fun-ord le) A
  assume A \( \neq \{\} \)
  hence non-empty: \( \forall a. \{ y. \exists f \in A. y = f a \} \neq \{\} \) by auto
  show \( \forall x. Q x (fun-lub lub A x) \)
    unfolding fun-lub-def
    by (rule allI, rule ccpo.admissibleD[OF adm chain-fun[OF ch] non-empty])
    (auto simp: Q)
qed

abbreviation option-ord \( \equiv \) flat-ord None
abbreviation mono-option \( \equiv \) monotone (fun-ord option-ord) option-ord

lemma bind-mono[partial-function-mono]:
assumes mf: mono-option B and mg: \( \forall y. mono-option (\lambda f. C y f) \)
shows mono-option (\( \lambda f. Option.bind (B f) (\lambda y. C y f) \))
proof (rule monotoneI)
  fix f g :: \( \forall a \Rightarrow b \) option assume fg: fun-ord option-ord f g
with \( \text{mf} \)
have \( \text{option-ord} \ (B \ f) \ (B \ g) \) by (rule \( \text{monotoneD}[\text{of } - - - f \ g] \))
then have \( \text{option-ord} \ (\text{Option.bind} \ (B \ f) \ (\lambda y. C \ y \ f)) \ (\text{Option.bind} \ (B \ g) \ (\lambda y. C \ y \ f)) \)
  unfolding \( \text{flat-ord-def} \) by \text{auto}
also from \( \text{mg} \)
have \( \forall y'. \ \text{option-ord} \ (C \ y' \ f) \ (C \ y' \ g) \)
by (rule \( \text{monotoneD} \ (\text{rule } fg) \))
then have \( \text{option-ord} \ (\text{Option.bind} \ (B \ g) \ (\lambda y'. C \ y' \ f)) \ (\text{Option.bind} \ (B \ g) \ (\lambda y'. C \ y' \ g)) \)
  unfolding \( \text{flat-ord-def} \) by (cases \( B \ g \)) \text{auto}
finally (\text{option.leq-trans})
show \( \text{option-ord} \ (\text{Option.bind} \ (B \ f) \ (\lambda y. C \ y \ f)) \ (\text{Option.bind} \ (B \ g) \ (\lambda y. C \ y \ f)) \).
qed

lemma \( \text{flat-lub-in-chain} \):
assumes \( \text{ch} : \text{chain} \ (\text{flat-ord } b) \ A \)
assumes \( \text{lab} : \text{flat-lub } b \ A = a \)
shows \( a = b \lor a \in A \)
proof (cases \( A \subseteq \{b\} \))
  case \( \text{True} \)
  then have \( \text{flat-lub } b \ A = b \) unfolding \( \text{flat-lub-def} \) by \text{simp}
with \( \text{lab} \) show \(?thesis \) by \text{simp}
next
  case \( \text{False} \)
  then obtain \( c \) where \( c \in A \) and \( c \neq b \) by \text{auto}
  { fix \( z \) assume \( z \in A \)
    from \( \text{chainD}[\text{OF } \text{ch} \ : \ c \in A \text{ this}] \)
    have \( z = c \lor z = b \)
    unfolding \( \text{flat-ord-def} \) using \( c \neq b \) by \text{auto} }
with \( \text{False} \) have \( A \setminus \{b\} = \{c\} \) by \text{auto}
with \( \text{False} \) have \( \text{flat-lub } b \ A = c \) (auto \text{ simp}; \text{flat-lub-def})
with \( \langle c \in A \rangle \)
  \( \text{lab} \) show \(?thesis \) by \text{simp}
qed

lemma \( \text{option-admissible} \): \( \text{option.admissible} \ (\forall (f::'a \Rightarrow 'b \text{ option}). \ (\forall x. \ f \ xx = \text{Some } y \longrightarrow P \ xx)) \)
proof (rule \( \text{ccpo.admissibleI} \))
fix \( A \) :: \( ('a \Rightarrow 'b \text{ option}) \) set
  assume \( \text{ch} : \text{chain } \text{option.le-fun } A \)
and \( \text{IH} : \forall f \in A. \forall x. \ f \ xx = \text{Some } y \longrightarrow P \ xx \)
from \( \text{ch} \) have \( \text{ch'} : \forall x. \text{chain option-ord } \{y. \exists f \in A. y = f \ xx\} \) by (rule \( \text{chain-fun} \))
show \( \forall x. y. \ \text{option.lub-fun } A \ xx = \text{Some } y \longrightarrow P \ xx \)
proof (intro \( \text{allI } \text{impl} \))
  fix \( x \) \( y \) assume \( \text{option.lub-fun } A \ xx = \text{Some } y \)
  from \( \text{flat-lub-in-chain}[\text{OF } \text{ch'} \ \text{this[unfolded fun-lub-def]]} \)
  have \( \text{Some } y \in \{y. \exists f \in A. y = f \ xx\} \) by \text{simp}
then have \( \exists f \in A. f \ xx = \text{Some } y \) by \text{auto}
with \( \text{IH} \) show \( P \ xx \) by \text{auto}

lemma fixp-induct-option:
  fixes F :: 'c ⇒ 'c and
  U :: 'c ⇒ 'a option and
  C :: ('b ⇒ 'a option) ⇒ 'c and
  P :: 'b ⇒ 'a ⇒ bool
assumes mono: ∀x. mono-option (λf. U (F (C f)) x)
assumes eq: f ≡ C (ccpo.fixp (fun-lub (flat-lub None)) (fun-ord option-ord) (λf. U (F (C f))))
assumes inverse2: ∀f. U (C f) = f
assumes step: ∀x y. (∀x y. U f x = Some y ⇒ P x y) ⇒ U (F f) x = Some y ⇒ P x y
assumes defined: U f x = Some y
shows P x y
using step defined option.fixp-induct-uc[of U F C, OF mono eq inverse2 option-admissible]
unfolding fun-lub-def flat-lub-def by(auto 9 2)
declaration (Partial-Function.init tailrec term (tailrec.fixp-fixp-fun)
term (tailrec.mono-body) @{thm tailrec.fixp-rule-uc} @{thm tailrec.fixp-induct-uc}
(SOME @{thm fixp-induct-tailrec[where c = undefined]}))
declaration (Partial-Function.init option term (option.fixp-fixp-fun)
term (option.mono-body) @{thm option.fixp-rule-uc} @{thm option.fixp-induct-uc}
(SOME @{thm fixp-induct-option}))-hide-const (open) chain
end
theory Argo
imports HOL
begin
ML-file (~~/src/Tools/Argo/argo-expr.ML)
ML-file (~~/src/Tools/Argo/argo-term.ML)
ML-file (~~/src/Tools/Argo/argo-lit.ML)
ML-file (~~/src/Tools/Argo/argo-proof.ML)
ML-file (~~/src/Tools/Argo/argo-rewr.ML)
ML-file (~~/src/Tools/Argo/argo-cls.ML)
ML-file (~~/src/Tools/Argo/argo-common.ML)
ML-file (~~/src/Tools/Argo/argo-cc.ML)
ML-file (~~/src/Tools/Argo/argo-simplex.ML)
ML-file (~~/src/Tools/Argo/argo-thy.ML)
ML-file (~~/src/Tools/Argo/argo-heap.ML)
ML-file (~~/src/Tools/Argo/argo-cdcl.ML)
ML-file (~~/src/Tools/Argo/argo-core.ML)
50 Reconstructing external resolution proofs for propositional logic

theory SAT
imports Argo
begin

ML-file ⟨Tools/Argo/argo-clausify.ML⟩
ML-file ⟨Tools/Argo/argo-solver.ML⟩
ML-file ⟨Tools/Argo/argo-tactic.ML⟩
end

method-setup sat = ⟨Scan.succeed (SIMPLE-METHOD' o SAT.sat-tac)⟩
SAT solver

method-setup satx = ⟨Scan.succeed (SIMPLE-METHOD' o SAT.satx-tac)⟩
SAT solver (with definitional CNF)

51 Function Definitions and Termination Proofs

theory Fun-Def
imports Basic-BNF-LFPs Partial-Function SAT
keywords
  function termination :: thy-goal-defn and
  fun fun-cases :: thy-defn
begin

51.1 Definitions with default value

definition THE-default :: 'a ⇒ ('a ⇒ bool) ⇒ 'a
  where THE-default d P = (if (∃!x. P x) then (THE x. P x) else d)

lemma THE-defaultI: ∃!x. P x ⇒ P (THE-default d P)
  by (simp add: theI' THE-default-def)

lemma THE-default1-equality: ∃!x. P x ⇒ P a ⇒ THE-default d P = a
  by (simp add: the1-equality THE-default-def)

lemma THE-default-none: (∃!x. P x) ⇒ THE-default d P = d
lemma fundef-ex1-existence:
assumes f-def: \( f \equiv (\lambda x::'a. \text{THE-default} (d x) (\lambda y. G x y)) \)
assumes ex1: \( \exists! y. G x y \)
shows G x (f x)
apply (simp only: f-def)
apply (rule THE-defaultI)
apply (rule ex1)
done

lemma fundef-ex1-uniqueness:
assumes f-def: \( f \equiv (\lambda x::'a. \text{THE-default} (d x) (\lambda y. G x y)) \)
assumes ex1: \( \exists! y. G x y \)
assumes elm: \( G x (h x) \)
shows h x = f x
apply (simp only: f-def)
apply (rule THE-default1-equality [symmetric])
apply (rule ex1)
apply (rule elm)
done

lemma fundef-ex1-iff:
assumes f-def: \( f \equiv (\lambda x::'a. \text{THE-default} (d x) (\lambda y. G x y)) \)
assumes ex1: \( \exists! y. G x y \)
shows \( (G x y) = (f x = y) \)
apply (auto simp:ex1 f-def THE-default1-equality)
apply (rule THE-defaultI)
apply (rule ex1)
done

lemma fundef-default-value:
assumes f-def: \( f \equiv (\lambda x::'a. \text{THE-default} (d x) (\lambda y. G x y)) \)
assumes graph: \( \forall x y. G x y \Rightarrow D x \)
assumes \( \neg D x \)
shows f x = d x
proof
  have \( \neg(\exists y. G x y) \)
  proof
    assume \( \exists y. G x y \)
    then have \( D x \) using graph ..
    with \( \neg D x \) show False ..
  qed
  then have \( \neg(\exists! y. G x y) \) by blast
  then show \( ?thesis \)
  unfolding f-def by (rule THE-default-none)
qed
**THEORY “Fun-Def”**

**definition** `in-rel-def[simp]: in-rel R x y ≡ (x, y) ∈ R`

**lemma** `wf-in-rel`: `wf R ⟹ wfP (in-rel R)`
  by `(simp add: wfP-def)`

**ML-file** `(Tools/Function/function-core.ML)`
**ML-file** `(Tools/Function/mutual.ML)`
**ML-file** `(Tools/Function/pattern-split.ML)`
**ML-file** `(Tools/Function/relation.ML)`
**ML-file** `(Tools/Function/function-elims.ML)`

**method-setup** `relation = ⟨`
  `Args.term >> (fn t => fnctxt => SIMPLE-METHOD’ (Function-Relation.relation-infer-tac ctxt t))`
  `⟩` prove termination using a user-specified wellfounded relation

**ML-file** `(Tools/Function/function.ML)`
**ML-file** `(Tools/Function/pat-completeness.ML)`

**method-setup** `pat-completeness = ⟨`
  `Scan.succeed (SIMPLE-METHOD’ o Pat-Completeness.pat-completeness-tac)`
  `⟩` prove completeness of (co)datatype patterns

**ML-file** `(Tools/Function/fun.ML)`
**ML-file** `(Tools/Function/induction-schema.ML)`

**method-setup** `induction-schema = ⟨`
  `Scan.succeed (CONTEXT-TACTIC oo Induction-Schema.induction-schema-tac)`
  `⟩` prove an induction principle

### 51.2 Measure functions

**inductive** `is-measure :: (‘a ⇒ nat) ⇒ bool`
  **where** `is-measure-trivial: is-measure f`

**named-theorems** measure-function rules that guide the heuristic generation of measure functions

**ML-file** `(Tools/Function/measure-functions.ML)`

**lemma** `measure-size[measure-function]: is-measure size`
  by `(rule is-measure-trivial)`

**lemma** `measure-fst[measure-function]: is-measure f ⟹ is-measure (λp. f (fst p))`
  by `(rule is-measure-trivial)`

**lemma** `measure-snd[measure-function]: is-measure f ⟹ is-measure (λp. f (snd p))`
  by `(rule is-measure-trivial)`
51.3 Congruence rules

lemma let-cong [fundef-cong]: $M = N \Rightarrow (\forall x. x = N \Rightarrow f x = g x) \Rightarrow \text{Let}
\begin{align*}
M f = \text{Let} N g
\end{align*}
\text{unfolding Let-def by blast}

lemmas [fundef-cong] =
\begin{align*}
\text{if-cong image-cong bex-cong ball-cong imp-cong map-option-cong Option.bind-cong}
\end{align*}
lemma split-cong [fundef-cong]:
\begin{align*}
(\forall x y. (x, y) = q \Rightarrow f x y = g x y) \Rightarrow p = q \Rightarrow \text{case-prod f p = case-prod g q}
\end{align*}
\text{by (auto simp: split-def)}

lemma comp-cong [fundef-cong]:
\begin{align*}
f (g x) = f' (g' x') \Rightarrow (f \circ g) x = (f' \circ g') x'
\end{align*}
\text{by (simp only: o-apply)}

51.4 Simp rules for termination proofs

declare
\begin{align*}
\text{trans-less-add1[termination-simp]}
\text{trans-less-add2[termination-simp]}
\text{trans-le-add1[termination-simp]}
\text{trans-le-add2[termination-simp]}
\text{less-imp-le-nat[termination-simp]}
\text{le-imp-less-Suc[termination-simp]}
\end{align*}
lemma size-prod-simp[termination-simp]: $\text{size-prod f g p} = f (\text{fst p}) + g (\text{snd p}) + \text{Suc 0}$
\text{by (induct p) auto}

51.5 Decomposition

lemma less-by-empty: $A = \{\} \Rightarrow A \subseteq B$
and union-comp-emptyL: $A \ O C = \{\} \Rightarrow B \ O C = \{\} \Rightarrow (A \cup B) \ O C = \{\}$
and union-comp-emptyR: $A \ O B = \{\} \Rightarrow A \ O C = \{\} \Rightarrow A \ O (B \cup C) = \{\}$
and wf-no-loop: $R \ O R = \{\} \Rightarrow \text{wf R}$
\text{by (auto simp add: wf-comp-self [of R])}

51.6 Reduction pairs
definition reduction-pair P ≜ \text{wf (fst P) \land fst P \ O snd P \subseteq fst P}
THEORY “Fun-Def”

lemma reduction-pairI[intro]: wf R ⇒ R O S ⊆ R ⇒ reduction-pair (R, S)
by (auto simp: reduction-pair-def)

lemma reduction-pair-lemma:
assumes rp: reduction-pair P
assumes R ⊆ fst P
assumes S ⊆ snd P
assumes wf S
shows wf (R ∪ S)
proof –
from rp :S ⊆ snd P; have wf (fst P)fst P O S ⊆ fst P
unfolding reduction-pair-def by auto
with ⟨wf S⟩ have wf (fst P ∪ S)
by (auto intro: wf-union-compatible)
moreover from ⟨R ⊆ fst P; have R ∪ S ⊆ fst P ∪ S by auto
ultimately show thesis by (rule wf-subset)
qed

definition rp-inv-image = (λ(R, S). (inv-image R f, inv-image S f))

lemma rp-inv-image-rp: reduction-pair P ⇒ reduction-pair (rp-inv-image P f)
unfolding reduction-pair-def rp-inv-image-def split-def by force

51.7 Concrete orders for SCNP termination proofs

definition pair-less = less-than <**lex**> less-than

definition pair-leq = pair-less

definition max-strict = max-ext pair-less

definition max-weak = max-ext pair-leq ∪ {{}, {}}

definition min-strict = min-ext pair-less

definition min-weak = min-ext pair-leq ∪ {{}, {}}

lemma wf-pair-less[simp]: wf pair-less
by (auto simp: pair-less-def)

lemma total-pair-less [iff]: total-on A pair-less and trans-pair-less [iff]: trans
pair-less
by (auto simp: total-on-def pair-less-def)

Introduction rules for pair-less/pair-leq

lemma pair-leqI1: a < b ⇒ ((a, s), (b, t)) ∈ pair-leq
and pair-leqI2: a ≤ b ⇒ s ≤ t ⇒ ((a, s), (b, t)) ∈ pair-leq
and pair-lessI1: a < b ⇒ ((a, s), (b, t)) ∈ pair-less
and pair-lessI2: a ≤ b ⇒ s < t ⇒ ((a, s), (b, t)) ∈ pair-less
by (auto simp: pair-leq-def pair-less-def)

Introduction rules for max

lemma smax-emptyI: finite Y ⇒ Y ≠ {} ⇒ ({}, Y) ∈ max-strict
and smax-insertI:
  \( y \in Y \implies (x, y) \in \text{pair-less} \implies (X, Y) \in \text{max-strict} \implies (\text{insert } x X, Y) \in \text{max-strict} \)

and wmax-emptyI: finite \( X \implies (\{\}, X) \in \text{max-weak} \)

and wmax-insertI:
  \( y \in YS \implies (x, y) \in \text{pair-leq} \implies (XS, YS) \in \text{max-weak} \implies (\text{insert } x XS, YS) \in \text{max-weak} \)

by (auto simp: max-strict-def max-weak-def elim: max-ext)

Introduction rules for min

lemma smin-emptyI: \( X \neq \{\} \implies (X, \{\}) \in \text{min-strict} \)

and smin-insertI:
  \( x \in XS \implies (x, y) \in \text{pair-less} \implies (XS, YS) \in \text{min-strict} \implies (XS, \text{insert } y YS) \in \text{min-strict} \)

and wmin-emptyI: \( (X, \{\}) \in \text{min-weak} \)

and wmin-insertI:
  \( x \in XS \implies (x, y) \in \text{pair-leq} \implies (XS, YS) \in \text{min-weak} \implies (XS, \text{insert } y YS) \in \text{min-weak} \)

by (auto simp: min-strict-def min-weak-def min-ext-def)

Reduction Pairs.

lemma max-ext-compat:
  assumes \( R \circ S \subseteq R \)
  shows \( \text{max-ext } R \circ O (\text{max-ext } S \cup \{\{\}, \{\}\}) \subseteq \text{max-ext } R \)
  using assms
  apply (auto simp: max-ext-def)
  done

lemma max-rpair-set: reduction-pair (max-strict, max-weak)
  unfolding max-strict-def max-weak-def
  apply (intro reduction-pairI max-ext-af)
  apply simp
  apply (rule max-ext-compat)
  apply (auto simp: pair-less-def pair-leq-def)
  done

lemma min-ext-compat:
  assumes \( R \circ S \subseteq R \)
  shows \( \text{min-ext } R \circ O (\text{min-ext } S \cup \{\{\}, \{\}\}) \subseteq \text{min-ext } R \)
  using assms
  apply (auto simp: min-ext-def)
  done
apply (drule-tac x=ya in bspec, assumption)
apply (erule bspec)
apply (drule-tac x=xc in bspec)
  apply assumption
apply auto
done

lemma min-rpair-set: reduction-pair (min-strict, min-weak)
  unfolding min-strict-def min-weak-def
  apply (intro reduction-pairI min-ext-wf)
  apply simp
  apply (rule min-ext-compat)
  apply (auto simp: pair-less-def pair-leq-def)
done

51.8 Yet another induction principle on the natural numbers

lemma nat-descend-induct [case-names base descend]:
  fixes P :: nat ⇒ bool
  assumes H1: ∀k. k > n ⇒ P k
  assumes H2: ∀k. k ≤ n ⇒ (∀i. i > k ⇒ P i) ⇒ P k
  shows P m
using assms by induction-schema (force intro!: wf_measure [of λk. Suc n – k])+

51.9 Tool setup

ML-file ⟨Tools/Function/termination.ML⟩
ML-file ⟨Tools/Function/scnp-solve.ML⟩
ML-file ⟨Tools/Function/scnp-reconstruct.ML⟩
ML-file ⟨Tools/Function/fun-cases.ML⟩

ML-val — setup inactive
  ‹
    Context.theory-map (Function-Common.set-termination-prover
      (K (ScnpReconstruct.decomp-scnp-tac [ScnpSolve.MAX, ScnpSolve.MIN, ScnpSolve.MS])))
  ›
end

52 The Integers as Equivalence Classes over Pairs of Natural Numbers

theory Int
  imports Equiv-Relations Power Quotient Fun-Def
begin
52.1 Definition of integers as a quotient type

**Definition** intrel :: (nat × nat) ⇒ (nat × nat) ⇒ bool
where intrel = (λ(x, y) (u, v). x + v = u + y)

**Lemma** intrel-iff [simp]: intrel (x, y) (u, v) ↔ x + v = u + y
by (simp add: intrel-def)

**Quotient-type** int = nat × nat / intrel
**Morphisms** Rep-Integ Abs-Integ
**Proof** (rule equivpI)
  show reflp intrel by (auto simp: reflp-def)
  show symp intrel by (auto simp: symp-def)
  show transp intrel by (auto simp: transp-def)
qed

**Lemma** eq-Abs-Integ [case-names Abs-Integ, cases type: int]:
(∀x y z = Abs-Integ (x, y) ⟹ P) ⟹ P
by (induct z) auto

52.2 Integers form a commutative ring

**Instantiation** int :: comm-ring-1
begin

**Lift-definition** zero-int :: int is (0, 0).

**Lift-definition** one-int :: int is (1, 0).

**Lift-definition** plus-int :: int ⇒ int ⇒ int
is λ(x, y) (u, v). (x + u, y + v)
by clarsimp

**Lift-definition** uminus-int :: int ⇒ int
is λ(x, y) (y, x)
by clarsimp

**Lift-definition** minus-int :: int ⇒ int ⇒ int
is λ(x, y) (u, v). (x + v, y + u)
by clarsimp

**Lift-definition** times-int :: int ⇒ int ⇒ int
is λ(x, y) (u, v). (x*u + y*v, x*v + y*u)
**Proof** (clarsimp)
fix s t u v w x y z :: nat
assume s + v = u + t and w + z = y + x
then have (s + v) * w + (u + t) * x + u * (w + z) + v * (y + x) =
  (u + t) * w + (s + v) * x + u * (y + x) + v * (w + z)
by simp
then show (s * w + t * x) + (u * z + v * y) = (u * y + v * z) + (s * x + t
* w) 
  by (simp add: algebra-simps)
qed

instance 
  by standard (transfer; clarsimp simp: algebra-simps)+
end

abbreviation int :: nat ⇒ int 
  where int ≡ of-nat

lemma int-def: int n = Abs-Integ (n, 0) 
  by (induct n) (simp add: zero-int.abs-eq, simp add: one-int.abs-eq plus-int.abs-eq)

lemma int-transfer [transfer-rule]:
  includes lifting-syntax 
  shows rel-fun (=) pcr-int (λn. (n, 0)) int 
  by (simp add: rel-fun-def int.pcr-cr-eq cr-int-def int-def)

lemma int-diff-cases: obtains (diff) m n where z = int m − int n 
  by transfer clarsimp

52.3 Integers are totally ordered

instantiation int :: linorder 
begin

lift-definition less-eq-int :: int ⇒ int ⇒ bool 
  is λ(x, y) (u, v). x + v ≤ u + y 
  by auto

lift-definition less-int :: int ⇒ int ⇒ bool 
  is λ(x, y) (u, v). x + v < u + y 
  by auto

instance 
  by standard (transfer, force)+
end

instantiation int :: distrib-lattice 
begin

definition (inf :: int ⇒ int ⇒ int) = min

definition (sup :: int ⇒ int ⇒ int) = max

instance
by standard (auto simp add: inf-int-def sup-int-def max-min-distrib2)

end

52.4 Ordering properties of arithmetic operations

instance \textit{int} :: ordered-cancel-ab-semigroup-add

proof
  fix \texttt{i \ j \ k :: int}
  show \texttt{i \leq \ j \implies k + i \leq k + j}
  by transfer clarsimp
qed

Strict Monotonicity of Multiplication.

Strict, in 1st argument; proof is by induction on \texttt{k \ > \ 0}.

lemma \textit{zmult-zless-mono2-lemma}: \texttt{i \ < \ j \implies 0 \ < \ k \implies int k * i \ < \ int k * j}
  for \texttt{i \ j :: int}

proof (induct \texttt{k})
  case \texttt{0}
  then show \texttt{?case} by simp

next
  case \texttt{(Suc \ k)}
  then show \texttt{?case}
    by (cases \texttt{k \ = \ 0}) (simp-all add: distrib-right add-strict-mono)
qed

lemma \textit{zero-le-imp-eq-int}: \texttt{0 \ \leq \ k \implies \exists \ n. \ k \ = \ int n}
  for \texttt{k :: int}

apply transfer
apply clarsimp
apply (rule-tac \texttt{x=a - b in exI})
apply simp
done

lemma \textit{zero-less-imp-eq-int}: \texttt{0 \ < \ k \implies \exists \ n>0. \ k \ = \ int n}
  for \texttt{k :: int}

apply transfer
apply clarsimp
apply (rule-tac \texttt{x=a - b in exI})
apply simp
done

lemma \textit{zmult-zless-mono2}: \texttt{i \ < \ j \implies 0 \ < \ k \implies k * i \ < \ k * j}
  for \texttt{i \ j \ k :: int}

by (drule zero-less-imp-eq-int) (auto simp add: zmult-zless-mono2-lemma)

The integers form an ordered integral domain.

instantiation \textit{int} :: linordered-idom
THEORY “Int”

begin

definition zabs-def: \(|i::\text{int}| = (\text{if } i < 0 \text{ then } -i \text{ else } i)\)

definition zsgn-def: \(\text{sgn (} i::\text{int}) = (\text{if } i = 0 \text{ then } 0 \text{ else if } 0 < i \text{ then } 1 \text{ else } -1)\)

instance
proof
fix i j k :: int
show \(i < j \Rightarrow 0 < k \Rightarrow k * i < k * j\)
by (rule zmult-zless-mono2)
show \(|i| = (\text{if } i < 0 \text{ then } -i \text{ else } i)\)
by (simp only: zabs-def)
show \(\text{sgn (} i::\text{int}) = (\text{if } i=0 \text{ then } 0 \text{ else if } 0<i \text{ then } 1 \text{ else } -1)\)
by (simp only: zsgn-def)
qed

end

lemma zless-imp-add1-zle: \(w < z \Rightarrow w + 1 \leq z\)
for \(w z :: \text{int}\)
by transfer clarsimp

lemma zless-iff-Suc-zadd: \(w < z \longleftrightarrow (\exists n. z = w + \text{int (Suc n)})\)
for \(w z :: \text{int}\)
apply transfer
apply auto
apply (rename-tac a b c d)
apply (rule-tac x=c+b - Suc(a+d) in exI)
apply arith
done

lemma zabs-less-one-iff [simp]: \(|z| < 1 \longleftrightarrow z = 0\) (is \(?lhs \longleftrightarrow ?rhs\)
for \(z :: \text{int}\)
proof
assume \(?rhs\)
then show \(?lhs\) by simp
next
assume \(?lhs\)
with zless-imp-add1-zle [of \(|z| 1\] have \(|z| + 1 \leq 1\) by simp
then have \(|z| \leq 0\) by simp
then show \(?rhs\) by simp
qed

52.5 Embedding of the Integers into any ring-1: of-int
context ring-1
begin
lift-definition of-int :: int ⇒ 'a
  is λ(i, j). of-nat i − of-nat j
  by (clarsimp simp add: diff-eq-eq eq-diff-eq diff-add-eq

lemma of-int-0 [simp]: of-int 0 = 0
  by transfer simp

lemma of-int-1 [simp]: of-int 1 = 1
  by transfer simp

lemma of-int-add [simp]: of-int (w + z) = of-int w + of-int z
  by transfer (clarsimp simp add: algebra-simps)

lemma of-int-minus [simp]: of-int (− z) = − (of-int z)
  by (transfer fixing: uminus) clarsimp

lemma of-int-diff [simp]: of-int (w − z) = of-int w − of-int z
  using of-int-add [of w − z] by simp

lemma of-int-mult [simp]: of-int (w * z) = of-int w * of-int z
  by (transfer fixing: times) (clarsimp simp add: algebra-simps)

lemma mult-of-int-commute: of-int x * y = y * of-int x
  by (transfer fixing: times) (auto simp: algebra-simps mult-of-nat-commute)

Collapse nested embeddings.

lemma of-int-of-nat-eq [simp]: of-int (int n) = of-nat n
  by (induct n) auto

lemma of-int-numeral [simp, code-post]: of-int (numeral k) = numeral k
  by (simp add: of-nat-numeral [symmetric] of-int-of-nat-eq [symmetric])

lemma of-int-neg-numeral [code-post]: of-int (− numeral k) = − numeral k
  by simp

lemma of-int-power [simp]: of-int (z ^ n) = of-int z ^ n
  by (induct n) simp-all

lemma of-int-of-bool [simp]:
  of-int (of-bool P) = of-bool P
  by auto

end
context ring-char-0
begin

lemma of-int-eq-iff [simp]: of-int w = of-int z ⇔ w = z

THEORY "Int"

932


Special cases where either operand is zero.

lemma of-int-eq-0-iff [simp]: of-int z = 0 ⟷ z = 0
  using of-int-eq-iff [of z 0] by simp

lemma of-int-0-eq-iff [simp]: 0 = of-int z ⟷ z = 0
  using of-int-eq-iff [of 0 z] by simp

lemma of-int-eq-1-iff [iff]: of-int z = 1 ⟷ z = 1
  using of-int-eq-iff [of z 1] by simp

lemma numeral-power-eq-of-int-cancel-iff [simp]:
  numeral x ^ n = of-int y ⟷ numeral x ^ n = y
  using of-int-eq-iff [of numeral x ^ n y, unfolded of-int-numeral of-int-power].

lemma of-int-eq-numeral-power-cancel-iff [simp]:
  of-int y = numeral x ^ n ⟷ y = numeral x ^ n
  using numeral-power-eq-of-int-cancel-iff [of x n y] by (metis (mono-tags))

lemma neg-numeral-power-eq-of-int-cancel-iff [simp]:
  (− numeral x) ^ n = of-int y ⟷ (− numeral x) ^ n = y
  using of-int-eq-iff [of (− numeral x) ^ n y] by simp

lemma of-int-eq-of-int-power-cancel-iff [simp]:
  (of-int b) ^ w = of-int x ⟷ b ^ w = x
  by (metis of-int-power of-int-eq-iff)

lemma of-int-power-eq-of-int-cancel-iff [simp]:
  of-int x = (of-int b) ^ w ⟷ x = b ^ w
  by (metis of-int-eq-of-int-power-cancel-iff)

end

context linordered-idom
begin

Every linordered-idom has characteristic zero.

subclass ring-char-0 ..

lemma of-int-le-iff [simp]: of-int w ≤ of-int z ⟷ w ≤ z
  by (transfer fixing: less-eq)
THEORY "Int"

lemma of-int-less_iff [simp]: of-int w < of-int z ←→ w < z
  by (simp add: less_le order_less_le)

lemma of-int-0-le_iff [simp]: 0 ≤ of-int z ←→ 0 ≤ z
  using of-int-le_iff [of 0 z] by simp

lemma of-int-le-0_iff [simp]: of-int z ≤ 0 ←→ z ≤ 0
  using of-int-le_iff [of z 0] by simp

lemma of-int-0-less_iff [simp]: 0 < of-int z ←→ 0 < z
  using of-int-less_iff [of 0 z] by simp

lemma of-int-less-0_iff [simp]: of-int z < 0 ←→ z < 0
  using of-int-less_iff [of z 0] by simp

lemma of-int-1-le_iff [simp]: 1 ≤ of-int z ←→ 1 ≤ z
  using of-int-le_iff [of 1 z] by simp

lemma of-int-le-1_iff [simp]: of-int z ≤ 1 ←→ z ≤ 1
  using of-int-le_iff [of z 1] by simp

lemma of-int-1-less_iff [simp]: 1 < of-int z ←→ 1 < z
  using of-int-less_iff [of 1 z] by simp

lemma of-int-less-1_iff [simp]: of-int z < 1 ←→ z < 1
  using of-int-less_iff [of z 1] by simp

lemma of-int-pos: z > 0 ⇒ of-int z > 0
  by simp

lemma of-int-nonneg: z ≥ 0 ⇒ of-int z ≥ 0
  by simp

lemma of-int-abs [simp]: of-int |x| = |of-int x|
  by (auto simp add: abs_if)

lemma of-int-lessD:
  assumes |of-int n| < x
  shows n = 0 ∨ x > 1
proof (cases n = 0)
  case True
  then show ?thesis by simp
next
  case False
  then have |n| ≠ 0 by simp
  then have |n| > 0 by simp
  then have |n| ≥ 1
    using zless_imp_add1_zle [of 0 |n|] by simp
then have $|\text{of-int } n| \geq 1$

unfolding of-int-1-le-iff $[|n|, \text{symmetric}]$ by simp
then have $1 < x$ using assms by (rule le-less-trans)
then show ?thesis ..

qed

lemma of-int-leD:

assumes $|\text{of-int } n| \leq x$

shows $n = 0 \lor 1 \leq x$

proof (cases $n = 0$)
  case True
  then show ?thesis by simp

next
  case False
  then have $|n| \neq 0$ by simp
  then have $|n| > 0$ by simp
  then have $|n| \geq 1$
    using zless-imp-add1-zle $[0, |n|]$ by simp
  then have $|\text{of-int } n| \geq 1$
    unfolding of-int-1-le-iff $[|n|, \text{symmetric}]$ by simp
  then have $1 \leq x$ using assms by (rule order-trans)
  then show ?thesis ..

qed

lemma numeral-power-le-of-int-cancel-iff [simp]:

numeral $x$ $^n$ $\leq$ of-int $a$ $\iff$ numeral $x$ $^n$ $\leq a$

by (metis (mono-tags) local.of-int-eq-numeral-power-cancel-iff of-int-le-iff)

lemma of-int-le-numeral-power-cancel-iff [simp]:

of-int $a$ $\leq$ numeral $x$ $^n$ $\iff$ a $\leq$ numeral $x$ $^n$

by (metis (mono-tags) local.numeral-power-eq-of-int-cancel-iff of-int-le-iff)

lemma numeral-power-less-of-int-cancel-iff [simp]:

numeral $x$ $^n$ $<$ of-int $a$ $\iff$ numeral $x$ $^n$ $<$ a

by (metis (mono-tags) local.of-int-eq-numeral-power-cancel-iff of-int-less-iff)

lemma of-int-less-numeral-power-cancel-iff [simp]:

of-int $a$ $<$ numeral $x$ $^n$ $\iff$ a $<$ numeral $x$ $^n$

by (metis (mono-tags) local.of-int-eq-numeral-power-cancel-iff of-int-less-iff)

lemma neg-numeral-power-le-of-int-cancel-iff [simp]:

$(-$ numeral $x$ $)^n$ $\leq$ of-int $a$ $\iff$ $(-$ numeral $x$ $)^n$ $\leq a$

by (metis (mono-tags) of-int-le-iff of-int-neg-numeral of-int-power)

lemma of-int-le-neg-numeral-power-cancel-iff [simp]:

of-int $a$ $\leq$ $(-$ numeral $x$ $)^n$ $\iff$ a $\leq$ $(-$ numeral $x$ $)^n$

by (metis (mono-tags) of-int-le-iff of-int-neg-numeral of-int-power)

lemma neg-numeral-power-less-of-int-cancel-iff [simp]:
Comparisons involving of-int. 

**Lemma of-int-eq-numeral-iff** [iff]: of-int z = (numeral n :: 'a::ring-char-0) ↔ z = numeral n 
using of-int-eq-iff by fastforce

**Lemma of-int-le-numeral-iff** [simp]: 
of-int z ≤ (numeral n :: 'a::linordered-idom) ↔ z ≤ numeral n 
using of-int-le-iff [of z numeral n] by simp
lemma of-int-numeral-le-iff [simp]:
\[(\text{numeral } n :: 'a::linordered-idom) \leq \text{of-int } z \iff \text{numeral } n \leq z\]
using of-int-le-iff [of numeral n] by simp

lemma of-int-less-numeral-iff [simp]:
\[\text{of-int } z < (\text{numeral } n :: 'a::linordered-idom) \iff z < \text{numeral } n\]
using of-int-less-iff [of numeral n] by simp

lemma of-int-numeral-less-iff [simp]:
\[(\text{numeral } n :: 'a::linordered-idom) < \text{of-int } z \iff \text{numeral } n < z\]
using of-int-less-iff [of numeral n z] by simp

lemma of-nat-less-of-int-iff: (of-nat n :: 'a::linordered-idom) < of-int x \iff int n < x
by (metis of-int-of-nat-eq of-int-less-iff)

lemma of-int-eq-id [simp]: of-int = id
proof
  show of-int z = id z for z
  by (cases z rule: int-diff-cases) simp
qed

instance int :: no-top
apply standard
apply (rule tac x=x + 1 in exI)
apply simp
done

instance int :: no-bot
apply standard
apply (rule tac x=x - 1 in exI)
apply simp
done

52.6 Magnitude of an Integer, as a Natural Number: nat

lift-definition nat :: int \Rightarrow nat is \(\lambda(x, y). x - y\)
by auto

lemma nat-int [simp]: nat (int n) = n
by transfer simp

lemma int-nat-eq [simp]: int (nat z) = (if 0 \leq z then z else 0)
by transfer clarsimp

lemma nat-0-le: 0 \leq z \Rightarrow int (nat z) = z
by simp
lemma nat-le-0 [simp]: \( z \leq 0 \implies \text{nat } z = 0 \)
  by transfer clarsimp

lemma nat-le-eq-zle: \( 0 < w \lor 0 \leq z \implies \text{nat } w \leq \text{nat } z \iff w \leq z \)
  by transfer (clarsimp, arith)

An alternative condition is \((0::'a) \leq w\).

lemma nat-mono-iff: \( 0 < z \implies \text{nat } w < \text{nat } z \iff w < z \)
  by (simp add: nat-le-eq-zle linorder-not-le [symmetric])

lemma nat-less-eq-zless: \( 0 \leq w \implies \text{nat } w < \text{nat } z \iff w < z \)
  by (simp add: nat-le-eq-zle linorder-not-le [symmetric])

lemma zless-nat-conj [simp]: \( \text{nat } w < \text{nat } z \iff 0 < z \land w < z \)
  by transfer (clarsimp, arith)

lemma nonneg-int-cases:
  assumes \( 0 \leq k \)
  obtains \( n \) where \( k = \text{int } n \)
proof –
  from assms have \( k = \text{int } (\text{nat } k) \)
    by simp
  then show thesis
    by (rule that)
qed

lemma pos-int-cases:
  assumes \( 0 < k \)
  obtains \( n \) where \( k = \text{int } n \) and \( n > 0 \)
proof –
  from assms have \( 0 \leq k \)
    by simp
  then obtain \( n \) where \( k = \text{int } n \)
    by (rule nonneg-int-cases)
  moreover have \( n > 0 \)
    using \( k = \text{int } n \) assms by simp
  ultimately show thesis
    by (rule that)
qed

lemma nonpos-int-cases:
  assumes \( k \leq 0 \)
  obtains \( n \) where \( k = - \text{int } n \)
proof –
  from assms have \( -k \geq 0 \)
    by simp
  then obtain \( n \) where \( -k = \text{int } n \)
    by (rule nonneg-int-cases)
  then have \( k = - \text{int } n \)
by simp
then show thesis
  by (rule that)
qed

lemma neg-int-cases:
  assumes k < 0
  obtains n where k = - int n and n > 0
proof –
  from assms have - k > 0
    by simp
  then obtain n where - k = int n and - k > 0
    by (blast elim: pos-int-cases)
  then have k = - int n and n > 0
    by simp-all
  then show thesis
    by (rule that)
qed

lemma nat-eq-iff:
  nat w = m ↔ (if 0 ≤ w then w = int m else m = 0)
by transfer (clarsimp simp add: le-imp-diff-is-add)

lemma nat-eq-iff2:
  m = nat w ↔ (if 0 ≤ w then w = int m else m = 0)
using nat-eq-iff [of w m] by auto

lemma nat-0 [simp]: nat 0 = 0
  by (simp add: nat-eq-iff)

lemma nat-1 [simp]: nat 1 = Suc 0
  by (simp add: nat-eq-iff)

lemma nat-numeral [simp]: nat (numeral k) = numeral k
  by (simp add: nat-eq-iff)

lemma nat-neg-numeral [simp]: nat (- numeral k) = 0
  by simp

lemma nat-2: nat 2 = Suc (Suc 0)
  by simp

lemma nat-less-iff: 0 ≤ w ==> nat w < m ↔ w < of-nat m
  by transfer (clarsimp simp add: linorder_not_less)

lemma nat-le-iff: nat x ≤ n ↔ x ≤ int n
  by transfer (clarsimp simp add: le_iff_diff_is_add)

lemma nat-mono: x ≤ y ==> nat x ≤ nat y
  by transfer auto
lemma nat-0-iff [simp]: nat i = 0 ⟷ i ≤ 0
  for i :: int
  by transfer clarsimp

lemma int-eq-iff: of-nat m = z ⟷ m = nat z ∧ 0 ≤ z
  by (auto simp add: nat-eq-iff2)

lemma zero-less-nat-eq [simp]: 0 < nat z ⟷ 0 < z
  using zless-nat-conj [of 0] by auto

lemma nat-add-distrib: 0 ≤ z ⟷ 0 ≤ z' ⟷ nat (z + z') = nat z + nat z'
  by transfer clarsimp

lemma nat-diff-distrib': 0 ≤ x ⟷ 0 ≤ y ⟷ nat (x - y) = nat x - nat y
  by transfer clarsimp

lemma nat-diff-distrib: 0 ≤ z' ⟷ z' ≤ z ⟷ nat (z - z') = nat z - nat z'
  by (rule nat-diff-distrib') auto

lemma nat-zminus-int [simp]: nat (- int n) = 0
  by transfer simp

lemma le-nat-iff: k ≥ 0 ⟷ n ≤ nat k ⟷ int n ≤ k
  by transfer auto

lemma zless-nat-eq-int-zless: m < nat z ⟷ int m < z
  by transfer (clarsimp simp add: less-diff-conv)

lemma (in ring-1) of-nat-nat [simp]: 0 ≤ z ⟷ of-nat (nat z) = of-int z
  by transfer (clarsimp simp add: of-nat-diff)

lemma diff-nat-numeral [simp]: (numeral v :: nat) - numeral v' = nat (numeral v - numeral v')
  by (simp only: nat-diff-distrib' zero-le-numeral nat-numeral)

lemma nat-abs-triangle-ineq:
  nat |k + l| ≤ nat |k| + nat |l|
  by (simp add: nat-add-distrib [symmetric] nat-le-eq-zle abs-triangle-ineq)

lemma nat-of-bool [simp]:
  nat (of-bool P) = of-bool P
  by auto

lemma split-nat [arith-split]: P (nat i) ⟷ ((∀ n. i = int n ⟷ P n) ∧ (i < 0 ⟷ P 0))
  (is ?P = (?L ∧ ?R))
  for i :: int
  proof (cases i < 0)
    case True
then show ?thesis 
  by auto
next
case False
have ?P = ?L
proof
  assume ?P
  then show ?L using False by auto
next
  assume ?L
  moreover from False have int (nat i) = i 
    by (simp add: not-less)
  ultimately show ?P
    by simp
qed
with False show ?thesis by simp
qed

lemma all-nat: (∀ x. P x) ←→ (∀ x≥0. P (nat x))
  by (auto split: split-nat)

lemma ex-nat: (∃ x. P x) ←→ (∃ x. 0 ≤ x ∧ P (nat x))
proof
  assume ∃ x. P x
  then obtain x where P x ..
  then have int x ≥ 0 ∧ P (nat (int x)) by simp
  then show ∃ x≥0. P (nat x) ..
next
  assume ∃ x≥0. P (nat x)
  then show ∃ x. P x by auto
qed

For termination proofs:

lemma measure-function-int[measure-function]: is-measure (nat o abs) ..

52.7 Lemmas about the Function of-nat and Orderings

lemma negative-zless-0: – (int (Suc n)) < (0 :: int)
  by (simp add: order-less-le del: of-nat-Suc)

lemma negative-zless [iff]: – (int (Suc n)) < int m
  by (rule negative-zless-0 [THEN order-less-le-trans], simp)

lemma negative-zle-0: – int n ≤ 0
  by (simp add: minus-le-iff)

lemma negative-zle [iff]: – int n ≤ int m
  by (rule order-trans {OF negative-zle-0 of-nat-0-le-iff})
lemma not-zle-0-negative [simp]: ¬ 0 ≤ − int (Suc n)
  by (subst le-minus-iff) (simp del: of-nat-Suc)

lemma int-zle-neg: int n ≤ − int m ⟷ n = 0 ∧ m = 0
  by transfer simp

lemma not-int-zless-negative [simp]: ¬ int n < − int m
  by (simp add: linorder-not-less)

lemma negative-eq-positive [simp]: − int n = of-nat m ⟷ n = 0 ∧ m = 0
  by (force simp add: order-eq-iff [of − of-nat n] int-zle-neg)

lemma zle-iff-zadd: w ≤ z ⟷ (∃ n. z = w + int n)
  (is ?lhs ⟷ ?rhs)
proof
  assume ?rhs
  then show ?lhs by auto
next
  assume ?lhs
  then have 0 ≤ z − w by simp
  then obtain n where z − w = int n
    using zero-le-imp-eq-int [of z − w] by blast
  then have z = w + int n by simp
  then show ?rhs ..
qed

lemma zadd-int-left: int m + (int n + z) = int (m + n) + z
  by simp

This version is proved for all ordered rings, not just integers! It is proved here because attribute arith-split is not available in theory Rings. But is it really better than just rewriting with abs-if?

lemma abs-split [arith-split, no-atp]: P |a| ⟷ (0 ≤ a ⟷ P a) ∧ (a < 0 ⟷ P (− a))
  for a :: 'a::linordered-idom
  by (force dest: order-less-le-trans simp add: abs-if linorder-not-less)

lemma negD: x < 0 ⟹ ∃ n. x = − (int (Suc n))
apply transfer
apply clarsimp
apply (rule_tac x=b − Suc a in exI)
apply arith
done

52.8 Cases and induction

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.
This version is symmetric in the two subgoals.

**lemma int-cases2** [case-names nonneg nonpos, cases type: int]:
\[
(\forall n. \text{z} = \text{int } n \Rightarrow P) \Rightarrow (\forall n. \text{z} = - (\text{int } n) \Rightarrow P) \Rightarrow P
\]
by (cases \text{z} < 0) (auto simp add: linorder-not-less dest!: negD nat-0-le [THEN sym])

This is the default, with a negative case.

**lemma int-cases** [case-names nonneg neg, cases type: int]:
\[
(\forall n. \text{z} = \text{int } n \Rightarrow P) \Rightarrow (\forall n. \text{z} = - (\text{int } (\text{Suc } n)) \Rightarrow P) \Rightarrow P
\]
apply (cases \text{z} < 0)
  apply (blast dest!: negD)
apply (simp add: linorder-not-less dest: of-nat-Suc)
apply auto
apply (blast dest: nat-0-le [THEN sym])
done

**lemma int-cases3** [case-names zero pos neg]:
fixes \text{k} :: int
assumes \text{k} = 0 \Rightarrow P and \(\forall n. \text{k} = \text{int } n \Rightarrow n > 0 \Rightarrow P\)
shows P
proof (cases \text{k} 0::int rule: linorder-cases)
case equal
  with assms (1) show P by simp
next
case greater
  then have *: nat \text{k} > 0 by simp
  moreover from * have \text{k} = \text{int } (\text{nat } \text{k}) by auto
  ultimately show P using assms (2) by blast
next
case less
  then have *: nat \text{(- k)} > 0 by simp
  moreover from * have \text{k} = - \text{int } (\text{nat } \text{(- k)}) by auto
  ultimately show P using assms (3) by blast
qed

**lemma int-of-nat-induct** [case-names nonneg neg, induct type: int]:
\[
(\forall n. \text{P } (\text{int } n)) \Longrightarrow (\forall n. \text{P } (\text{int } (\text{Suc } n))) \Longrightarrow P \text{ z}
\]
by (cases \text{z}) auto

**lemma sgn-mult-dvd-iff** [simp]:
\[
\text{sgn } r \ast l \text{ dvd } k \longleftrightarrow l \text{ dvd } k \land (r = 0 \rightarrow k = 0)
\]
for \text{k l r :: int}
by (cases \text{r rule: int-cases3}) auto

**lemma mult-sgn-dvd-iff** [simp]:
\[
l \ast \text{sgn } r \text{ dvd } k \longleftrightarrow l \text{ dvd } k \land (r = 0 \rightarrow k = 0)
\]
for \text{k l r :: int}
using sgn-mult-dvd-iff [of \text{r l k}] by (simp add: ac-simps)

**lemma dvd-sgn-mult-iff** [simp]:
l dvd sgn r \leftrightarrow l dvd k \lor r = 0 \text{ for } k l r :: \text{int}

by (cases r rule: int-cases3) simp-all

lemma dvd-mult-sgn-iff [simp]:
l dvd k * sgn r \leftrightarrow l dvd k \lor r = 0 \text{ for } k l r :: \text{int}
using dvd-sgn-mult-iff [of l r k] by (simp add: ac-simps)

lemma int-sgnE:
fixes k :: int
obtains n and l where k = sgn l * int n
proof –
have k = sgn k * int (nat |k|)
  by (simp add: sgn-mult-abs)
then show ?thesis ..
qed

52.8.1 Binary comparisons

Preliminaries

lemma le-imp-0-less:
fixes z :: int
assumes le: 0 \leq z
shows 0 < 1 + z
proof –
have 0 \leq z by fact
also have \ldots < z + 1 by (rule less-add-one)
also have \ldots = 1 + z by (simp add: ac-simps)
finally show 0 < 1 + z.
qed

lemma odd-less-0-iff: 1 + z + z \neq 0 \leftrightarrow z < 0
for z :: int
proof (cases z)
case (nonneg n)
then show ?thesis
  by (simp add: linorder-not-less add.assoc add-increasing le-imp-0-less [THEN order-less-imp-le])
next
case (neg n)
then show ?thesis
  by (simp del: of-nat-Suc of-nat-add of-nat-1
    add: algebra-simps of-nat-1 [where 'a=int, symmetric] of-nat-add [symmetric])
qed

52.8.2 Comparisons, for Ordered Rings

lemma odd-nonzero: 1 + z + z \neq 0
for z :: int
proof (cases z)
case (nonneg n)
  have le: \(0 \leq z + z\)
    by (simp add: nonneg add-increasing)
  then show ?thesis
    using le-imp-0-less [OF le] by (auto simp: ac-simps)
next
case (neg n)
  show ?thesis
    proof
      assume eq: \(1 + z + z = 0\)
      have \(0 < 1 + (\text{int } n + \text{int } n)\)
        by (simp add: le-imp-0-less add-increasing)
      also have \(\ldots = - (1 + z + z)\)
        by (simp add: neg add.assoc [symmetric])
      also have \(\ldots = 0\)
        by (simp add: eq)
      finally have \(0 < 0\)
      then show False by blast
    qed
qed

52.9 The Set of Integers
context ring-1
begin

definition Ints :: 'a set (\(\mathbb{Z}\))
  where \(\mathbb{Z} = \text{range of-int}\)

lemma Ints-of-int [simp]: of-int \(z\) \(\in\) \(\mathbb{Z}\)
  by (simp add: Ints-def)

lemma Ints-of-nat [simp]: of-nat \(n\) \(\in\) \(\mathbb{Z}\)
  using Ints-of-int [of of-nat \(n\)] by simp

lemma Ints-0 [simp]: \(0\) \(\in\) \(\mathbb{Z}\)
  using Ints-of-int [of 0] by simp

lemma Ints-1 [simp]: \(1\) \(\in\) \(\mathbb{Z}\)
  using Ints-of-int [of 1] by simp

lemma Ints-numeral [simp]: numeral \(n\) \(\in\) \(\mathbb{Z}\)
  by (subst of-nat-numeral [symmetric], rule Ints-of-nat)

lemma Ints-add [simp]: \(a \in \mathbb{Z} \rightleftharpoons\) \(b \in \mathbb{Z} \rightleftharpoons\) \(a + b \in \mathbb{Z}\)
  apply (auto simp add: Ints-def)
  apply (rule range-eql)
  apply (rule of-int-add [symmetric])
  done
lemma Ints-minus [simp]: \( a \in \mathbb{Z} \implies -a \in \mathbb{Z} \)
apply (auto simp add: Ints-def)
apply (rule range-eqI)
apply (rule of-int-minus [symmetric])
done

lemma minus-in-Ints-iff: \(-x \in \mathbb{Z} \iff x \in \mathbb{Z}\)
using Ints-minus [of x]
Ints-minus [of \(-x\)]
by auto

lemma Ints-diff [simp]: \( a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a - b \in \mathbb{Z} \)
apply (auto simp add: Ints-def)
apply (rule range-eqI)
apply (rule of-int-diff [symmetric])
done

lemma Ints-mult [simp]: \( a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a * b \in \mathbb{Z} \)
apply (auto simp add: Ints-def)
apply (rule range-eqI)
apply (rule of-int-mult [symmetric])
done

lemma Ints-power [simp]: \( a \in \mathbb{Z} \implies a ^ n \in \mathbb{Z} \)
by (induct n) simp-all

lemma Ints-cases [cases set: Ints]:
assumes q \( \in \mathbb{Z} \)
obtains (of-int) z where q = of-int z
unfolding Ints-def
proof –
from \( q \in \mathbb{Z} \): have q \( \in \) range of-int unfolding Ints-def .
then obtain z where q = of-int z ..
then show thesis ..
qed

lemma Ints-induct [case-names of-int, induct set: Ints]:
q \( \in \mathbb{Z} \) \( \\implies \) (\( \forall \) z. P (of-int z)) \( \\implies \) P q
by (rule Ints-cases) auto

lemma Nats-subset-Ints: \( \mathbb{N} \subseteq \mathbb{Z} \)
unfolding Nats-def Ints-def
by (rule subsetI, elim imageE, hypsubst, subst of-int-of-nat-eq[symmetric], rule imageI) simp-all

lemma Nats-altdef1: \( \mathbb{N} = \{ \text{of-int } n \mid n. n \geq 0 \} \)
proof (intro subsetI equalityI)
fix x :: 'a
assume x \( \in \) \( \{ \text{of-int } n \mid n. n \geq 0 \} \)
then obtain n where x = of-int n n \( \geq \) 0
by (auto elim!: Ints-cases)
then have \( x = \text{of-nat} \ (\text{nat} \ n) \)
  by (subst of-nat-nat) simp-all
then show \( x \in \mathbb{N} \)
  by simp

next
fix \( x :: 'a \)
assume \( x \in \mathbb{N} \)
then obtain \( n \) where \( x = \text{of-nat} \ n \)
  by (auto elim!: Nats-cases)
then have \( x = \text{of-int} \ (\text{int} \ n) \) by simp
also have \( \text{int} \ n \geq 0 \) by simp
then have \( \text{of-int} \ (\text{int} \ n) \in \{ \text{of-int} \ n \ | n \geq 0 \} \) by blast
finally show \( x \in \{ \text{of-int} \ n \ | n \geq 0 \} \).
qed

end

lemma (in linordered-idom) Ints-abs [simp]:
  shows \( a \in \mathbb{Z} \implies \text{abs} \ a \in \mathbb{Z} \)
by (auto simp: abs-if)

lemma (in linordered-idom) Nats-altdef2: \( \mathbb{N} = \{ n \in \mathbb{Z} . n \geq 0 \} \)
proof (intro subsetI equalityI)
fix \( x :: 'a \)
assume \( x \in \{ n \in \mathbb{Z} . n \geq 0 \} \)
then obtain \( n \) where \( x = \text{of-int} \ n \ n \geq 0 \)
  by (auto elim!: Ints-cases)
then have \( x = \text{of-nat} \ (\text{nat} \ n) \)
  by (subst of-nat-nat) simp-all
then show \( x \in \mathbb{N} \)
  by simp
qed (auto elim!: Nats-cases)

lemma (in idom-divide) of-int-divide-in-Ints:
  of-int \( a \) div of-int \( b \in \mathbb{Z} \) if \( b \) dvd \( a \)
proof –
  from that obtain \( c \) where \( a = b \ast c \) ..
  then show \( \?thesis \)
    by (cases of-int \( b = 0 \)) simp-all
qed

The premise involving \( \mathbb{Z} \) prevents \( a = (1::'a) / (2::'a) \).

lemma Ints-double-eq-0-iff:
  fixes \( a :: 'a::ring-char-0 \)
  assumes in-Ints: \( a \in \mathbb{Z} \)
  shows \( a + a = 0 \iff a = 0 \)
    (is \( ?lhs \iff ?rhs \))
proof –
  from in-Ints have \( a \in \text{range of-int} \)
unfolding Ints-def [symmetric] .
then obtain z where a: a = of-int z ..
show ?thesis
proof
  assume ?rhs
  then show ?lhs by simp
next
  assume ?lhs
  with a have of-int (z + z) = (of-int 0 :: 'a) by simp
  then have z + z = 0 by (simp only: of-int-eq-iff)
  then have z = 0 by (simp only: double-zero)
  with a show ?rhs by simp
qed
qed

lemma Ints-odd-nonzero:
  fixes a :: 'a::ring-char-0
  assumes in-Ints: a ∈ ℤ
  shows 1 + a + a ≠ 0
proof
  from in-Ints have a ∈ range of-int unfolding Ints-def [symmetric] .
  then obtain z where a: a = of-int z ..
  show ?thesis
  proof
    assume 1 + a + a = 0
    with a have of-int (1 + z + z) = (of-int 0 :: 'a) by simp
    then have 1 + z + z = 0 by (simp only: of-int-eq-iff)
    with odd-nonzero show False by blast
  qed
qed

lemma Nats-numeral [simp]: numeral w ∈ ℕ
  using of-nat-in-Nats [of numeral w] by simp

lemma Ints-odd-less-0:
  fixes a :: 'a::linordered-idom
  assumes in-Ints: a ∈ ℤ
  shows 1 + a + a < 0 ←→ a < 0
proof
  from in-Ints have a ∈ range of-int unfolding Ints-def [symmetric] .
  then obtain z where a: a = of-int z ..
  with a have 1 + a + a < 0 ←→ of-int (1 + z + z) < (of-int 0 :: 'a)
    by simp
  also have .. ←→ z < 0
    by (simp only: of-int-less-iff odd-less-0-iff)
  also have .. ←→ a < 0
    by (simp add: a)
finally show ?thesis .

qed

52.10 sum and prod

context semiring-1

begin

lemma of-nat-sum [simp]:
  of-nat (sum f A) = (∑ x ∈ A. of-nat (f x))
  by (induction A rule: infinite-finite-induct) auto

end

context ring-1

begin

lemma of-int-sum [simp]:
  of-int (sum f A) = (∑ x ∈ A. of-int (f x))
  by (induction A rule: infinite-finite-induct) auto

end

context comm-semiring-1

begin

lemma of-nat-prod [simp]:
  of-nat (prod f A) = (∏ x ∈ A. of-nat (f x))
  by (induction A rule: infinite-finite-induct) auto

end

context comm-ring-1

begin

lemma of-int-prod [simp]:
  of-int (prod f A) = (∏ x ∈ A. of-int (f x))
  by (induction A rule: infinite-finite-induct) auto

end

52.11 Setting up simplification procedures

ML-file (Tools/int-arith.ML)

declaration : K (  
Lin-Arith.add-discrete-type type-name (Int.int)  
#> Lin-Arith.add-lessD @{thm zless_imp_add1_zle}  
#> Lin-Arith.add-inj-thms @{thms of-nat-le_iff [THEN iffD2] of-nat-eq_iff [THEN iffD2]}  
)
THEORY “Int”

```isar
#> Lin-Arith.add-inj-const (const-name of-nat, typ (nat ⇒ int))
#> Lin-Arith.add-simps
@{thms of-int-0 of-int-1 of-int-add of-int-mult of-int-numeral of-int-neg-numeral nat-0 nat-1 diff-nat-numeral nat-numeral
  neg-less-iff-less
  True-implies-equals
  distrib-left [where a = numeral v for v]
  distrib-left [where a = − numeral v for v]
  div-by-1 div-0
  times-divide-eq-right times-divide-eq-left
  minus-divide-left [THEN sym] minus-divide-right [THEN sym]
  add-divide-distrib diff-divide-distrib
  of-int-minus of-int-diff
  of-int-of-nat-eq
}@ {K Lin-Arith.simproc}

52.12 More Inequality Reasoning

lemma zless-add1-eq: w < z + 1 ⟷ w < z ∨ w = z
  for w z :: int
  by arith

lemma add1-zle-eq: w + 1 ≤ z ⟷ w < z
  for w z :: int
  by arith

lemma zle-diff1-eq [simp]: w ≤ z − 1 ⟷ w < z
  for w z :: int
  by arith

lemma zle-add1-eq-le [simp]: w < z + 1 ⟷ w ≤ z
  for w z :: int
  by arith

lemma int-one-le-iff-zero-less: 1 ≤ z ⟷ 0 < z
  for z :: int
  by arith

lemma Ints-nonzero-abs-ge1:
  fixes x :: linordered-idom
  assumes x ∈ Ints x ≠ 0
  shows 1 ≤ abs x
```
proof (rule Ints-cases [OF \( x \in \text{Ints} \)])
fix \( z :: \text{int} \)
assume \( x = \text{of-int } z \)
with \( x \neq 0 \)
show \( 1 \leq |x| \)
apply (auto simp add: abs-if)
by (metis diff-0 of-int-1 of-int-le-iff of-int-minus zle-diff1-eq)
qed

lemma Ints-nonzero-abs-less1:
fixes \( x :: 'a :: \text{linordered-idom} \)
shows \( [x \in \text{Ints}; \text{abs } x < 1] \Longrightarrow x = 0 \)
using Ints-nonzero-abs-ge1 [of \( x \)] by auto

lemma Ints-eq-abs-less1:
fixes \( x :: 'a :: \text{linordered-idom} \)
shows \( [x \in \text{Ints}; y \in \text{Ints}] \Longrightarrow x = y \iff \text{abs } (x-y) < 1 \)
using eq-iff-diff-eq-0 by (fastforce intro: Ints-nonzero-abs-less1)

52.13 The functions \textit{nat} and \textit{int}

Simplify the term \( w + -z \).

lemma one-less-nat-eq [simp]: \( \text{Suc } 0 < \text{nat } z \iff 1 < z \)
using zless-nat-conj [of 1 z] by auto

lemma int-eq-iff-numeral [simp];
\( \text{int } m = \text{numeral } v \iff m = \text{numeral } v \)
by (simp add: int-eq-iff)

lemma nat-abs-int-diff:
\( \text{nat } |\text{int } a - \text{int } b| = (\text{if } a \leq b \text{ then } b - a \text{ else } a - b) \)
by auto

lemma nat-int-add; \( \text{nat } (\text{int } a + \text{int } b) = a + b \)
by auto

context ring-1
begin

lemma of-int-of-nat [nitpick-simp];
of-int \( k = (\text{if } k < 0 \text{ then } -\text{of-nat } (\text{nat } (-k)) \text{ else } \text{of-nat } (\text{nat } k)) \)
proof (cases \( k < 0 \))
case True
then have \( 0 \leq -k \) by simp
then have \( \text{of-nat } (\text{nat } (-k)) = \text{of-int } (-k) \) by (rule of-nat-nat)
with True show \( ?\text{thesis} \) by simp
next
case False
then show \( ?\text{thesis} \) by (simp add: not-less)
THEORY “Int”

qed

end

lemma transfer-rule-of-int:
  includes lifting-syntax
  fixes R :: 'a::ring-1 ⇒ 'b::ring-1 ⇒ bool
  assumes [transfer-rule]: R 0 0 R 1 1
          (R ===> R ===> R) (+) (+)
          (R ===> R) uminus uminus
  shows ((=) ===> R) of-int of-int
proof –
  note assms
  note transfer-rule-of-nat [transfer-rule]
  have [transfer-rule]: ((=) ===> R) of-nat of-nat
    by transfer-prover
  show ?thesis
    by (unfold of-int-of-nat [abs-def]) transfer-prover
qed

lemma nat-mult-distrib:
  fixes z z' :: int
  assumes 0 ≤ z
  shows nat (z * z') = nat z * nat z'
proof (cases 0 ≤ z')
  case False
    with assms have z * z' ≤ 0
      by (simp add: not-le mult-le-0-iff)
    then have nat (z * z') = 0 by simp
    moreover from False have nat z' = 0 by simp
    ultimately show ?thesis by simp
  next
  case True
    with assms have ge-0: z * z' ≥ 0 by (simp add: zero-le-mult-iff)
    show ?thesis
      by (rule injD [of of-nat :: nat ⇒ int, OF inj-of-nat])
      (simp only: of-nat-mult of-nat-nat [OF assms] of-nat-nat [OF ge-0], simp)
qed

lemma nat-mult-distrib-neg: z ≤ 0 ===> nat (z * z') = nat (- z) * nat (- z')
  for z z' :: int
  apply (rule trans)
  apply (rule-tac [2] nat-mult-distrib)
  apply auto
  done

lemma nat-abs-mult-distrib: nat |w * z| = nat |w| * nat |z|
  by (cases z = 0 ∨ w = 0)

lemma int-in-range-abs [simp]: int n ∈ range abs
proof (rule range-eqI)
  show int n = |int n| by simp
qed

lemma range-abs-Nats [simp]: range abs = (\text{\textN} :: int set)
proof
  have \(|k| \in \text{\textN} \text{ for } k :: \text{int}\)
    by (cases k) simp-all
  moreover have \(k \in \text{range abs} \text{ if } k \in \text{\textN} \text{ for } k :: \text{int}\)
    using that by induct simp
  ultimately show \(?thesis by blast
qed

lemma Suc-nat-eq-nat-zadd1: \(0 \leq z \Rightarrow Suc (nat z) = nat (1 + z)\)
for \(z :: \text{int}\)
by (rule sym) (simp add: nat-eq-iff)

lemma diff-nat-eq-if:
  \(nat z - nat z' =\)
  (if \(z' < 0\) then nat z
  else
    let \(d = z - z'\)
in if \(d < 0\) then 0 else nat d)
  by (simp add: Let-def nat-diff-distrib [symmetric])

lemma nat-numeral-diff-1 [simp]: \(\text{numeral v - (1::nat)} = \text{nat (numeral v - 1)}\)
using diff-nat-numeral [of v Num.One] by simp

52.14 Induction principles for int

Well-founded segments of the integers.

definition int-ge-less-than :: int ⇒ (int × int) set
where int-ge-less-than d = \{\(\langle z', z\rangle. \ d \leq z' \land z' < z\)\}

lemma wf-int-ge-less-than: wf (int-ge-less-than d)
proof
  have \(int-ge-less-than d \subseteq \text{measure (\lambda z. nat (z - d))}\)
    by (auto simp add: int-ge-less-than-def)
  then show \(?thesis
    by (rule wf-subset [OF wf-measure])
qed

This variant looks odd, but is typical of the relations suggested by Rank-Finder.

definition int-ge-less-than2 :: int ⇒ (int × int) set
where \(\text{int-ge-less-than2 } d = \{(z,z'). \ d \leq z \land z' < z\}\)

lemma \(\text{wf-int-ge-less-than2} : \text{wf (int-ge-less-than2 } d\) proof
  have \(\text{int-ge-less-than2 } d \subseteq \text{measure } (\lambda z. \text{nat } (1 + z - d))\)
    by \((\text{auto simp add: int-ge-less-than2-def})\)
  then show \(!\text{thesis}\)
    by \((\text{rule wf-subset } [OF \text{wf-measure}])\)
qed

theorem \(\text{int-ge-induct}\)\\[\text{case-names base step, induct set: int}]:\\
  fixes \(i :: \text{int}\)\\
  assumes \(\text{ge: } k \leq i\)
    and \(\text{base: } P k\)
    and \(\text{step: } \forall i. k \leq i \implies P i \implies P (i + 1)\)\\
  shows \(P i\) proof
    have \(\forall i :: \text{int}. \ n = \text{nat } (i - k) \implies k \leq i \implies P i\) for \(n\)
    proof \((\text{induct } n)\)
      case \(\theta\)
        then have \(i = k\) by \(\text{arith}\)
        with \(\text{base}\) show \(P i\) by \(\text{simp}\)
      next
      case \(\text{Suc } n\)
        then have \(n = \text{nat } ((i - 1) - k)\) by \(\text{arith}\)
        moreover have \(k \leq i - 1\) using \(\text{Suc.prems}\) by \(\text{arith}\)
        ultimately have \(P (i - 1)\) by \((\text{rule Suc.hyps})\)
        from \(\text{step } [OF k \text{ this}]\) show \(!\text{case}\) by \(\text{simp}\)
    qed
    with \(\text{ge}\) show \(!\text{thesis} by \text{fast}\)
qed

theorem \(\text{int-gr-induct}\)\\[\text{case-names base step, induct set: int}]:\\
  fixes \(i k :: \text{int}\)\\
  assumes \(\text{gr: } k < i\)
    and \(\text{base: } P (k + 1)\)
    and \(\text{step: } \forall i. k < i \implies P i \implies P (i + 1)\)\\
  shows \(P i\) apply \((\text{rule int-ge-induct}[of } k + 1]\))
    using \(\text{gr}\) apply \(\text{arith}\)
    apply \((\text{rule base})\)
    apply \((\text{rule step})\)
    apply \(\text{simp-all}\)
  done

theorem \(\text{int-le-induct}\)\\[\text{consumes 1, case-names base step}]:\\
  fixes \(i k :: \text{int}\)
assumes \( \text{le: } i \leq k \)
and \( \text{base: } P k \)
and \( \text{step: } \bigwedge_{i}. \ i \leq k \implies P i \implies P (i - 1) \)
shows \( P i \)

proof –
have \( \bigwedge_{i::\text{int}.} n = \text{nat}(k-i) \implies i \leq k \implies P i \) for \( n \)
proof (induct \( n \))
case 0
then have \( i = k \) by arith
with base show \( P i \) by simp
next
case (Suc \( n \))
then have \( n = \text{nat}(k - (i + 1)) \) by arith
moreover have \( k: \ i + 1 \leq k \) using Suc.prems by arith
ultimately have \( P (i + 1) \) by (rule Suc.hyps)
from step[OF \( k \) this] show \( \text{?case} \) by simp
qed
with le show \( \text{?thesis} \) by fast
qed

theorem \text{int-less-induct} \[\text{consumes 1, case-names base step}];
fixes \( i \) \( k \) :: int
assumes \( \text{less: } i < k \)
and \( \text{base: } P (k - 1) \)
and \( \text{step: } \bigwedge_{i}. \ i < k \implies P i \implies P (i - 1) \)
shows \( P i \)
apply (rule \text{int-le-induct}![of \( - k - 1 \)])
using less apply arith
apply (rule base)
apply (rule step)
apply simp-all
done

theorem \text{int-induct} \[\text{case-names base step1 step2}];
fixes \( k \) :: int
assumes \( \text{base: } P k \)
and \( \text{step1: } \bigwedge_{i}. \ k \leq i \implies P i \implies P (i + 1) \)
and \( \text{step2: } \bigwedge_{i}. \ k \geq i \implies P i \implies P (i - 1) \)
shows \( P i \)
proof –
have \( i \leq k \lor i \geq k \) by arith
then show \( \text{?thesis} \)
proof
assume \( i \geq k \)
then show \( \text{?thesis} \)
using base by (rule \text{int-ge-induct}) (fact \( \text{step1} \))
next
assume \( i \leq k \)
then show \( \text{?thesis} \)
THEORY "Int"

using base by (rule int-le-duct) (fact step2)

qed

52.15 Intermediate value theorems

lemma nat-intermed-int-val:
\exists i, m \leq i \land i \leq n \land f i = k  
if \forall i, m \leq i \land i < n \rightarrow |f (Suc i) - f i| \leq 1  
m \leq n f m \leq k k \leq f n  
for m n :: nat and k :: int

proof -

have (\forall i \leq n. |f (Suc i) - f i| \leq 1) \implies f 0 \leq k \implies k \leq f n  
\implies (\exists i \leq n. f i = k)  
for n :: nat and f

apply (induct n)
apply auto
apply (erule-tac x = n in allE)
apply (case-tac k = f (Suc n))
apply (auto simp add: abs-if split:if-split-asm intro:le-SucI)
done

from this [of n m f plus m] that show \?thesis

apply auto
apply (rule-tac x = m + i in exI)
apply auto
done

qed

lemma nat0-intermed-int-val:
\exists i \leq n. f i = k  
if \forall i \leq n. |f (i + 1) - f i| \leq 1 f 0 \leq k k \leq f n  
for n :: nat and k :: int

using nat-intermed-int-val [of 0 n f k] that by auto

52.16 Products and 1, by T. M. Rasmussen

lemma abs-zmult-eq-1:

fixes m n :: int
assumes mn: \|m \ast n\| = 1
shows \|m\| = 1

proof -

from mn have 0: m \neq 0 n \neq 0 by auto

have \neg 2 \leq |m|

proof

assume 2 \leq |m|

then have 2 \ast |n| \leq |m| \ast |n| by (simp add: mult-mono 0)

also have \ldots = |m \ast n| by (simp add: abs-mult)

also from mn have \ldots = 1 by simp

finally have 2 \ast |n| \leq 1 .

with 0 show False by arith
qed
with \( \theta \) show \( \text{thesis} \) by auto
qed

lemma pos-zmult-eq-1-iff-lemma: \( m \ast n = 1 \implies m = 1 \lor m = -1 \)
for \( m \simplex n :: \int \)
using abs-zmult-eq-1 [of \( m \ simplex n \)] by arith

lemma pos-zmult-eq-1-iff:
fixes \( m \ simplex n :: \int \)
assumes \( 0 < m \)
shows \( m \ast n = 1 \iff m = 1 \land n = 1 \)
proof –
from \( \text{assms} \) have \( m \ast n = 1 \implies m = 1 \)
by (auto dest: pos-zmult-eq-1-iff-lemma)
then show \( \text{thesis} \)
by (auto dest: pos-zmult-eq-1-iff-lemma)
qed

lemma zmult-eq-1-iff: \( m \ast n = 1 \iff (m = 1 \land n = 1) \lor (m = -1 \land n = -1) \)
for \( m \ simplex n :: \int \)
apply (rule iffI)
apply (frule pos-zmult-eq-1-iff-lemma)
apply (simp add: mult.assoc zero-le-mult-iff zmult-eq-1-iff)
apply auto

done

lemma infinite-UNIV-int [simp]: \( \neg \text{finite} (UNIV::\int \ simplex) \)
proof
assume finite (UNIV::\int \ simplex)
moreover have inj (\( \lambda i :: \int. \ 2 \ast i \))
by (rule injI) simp
ultimately have surj (\( \lambda i :: \int. \ 2 \ast i \))
by (rule finite-UNIV-inj-surj)
then obtain \( i :: \int \) where \( 1 = 2 \ast i \) by (rule surjE)
then show False by (simp add: pos-zmult-eq-1-iff)
qed

\section{52.17 The divides relation}

lemma zdvd-antisym-nonneg: \( 0 \leq m \implies 0 \leq n \implies m \vdots n \implies n \vdots m \implies m = n \)
for \( m \ simplex n :: \int \)
by (auto simp add: dvd-def mult.assoc zero-le-mult-iff zmult-eq-1-iff)

lemma zdvd-antisym-abs:
fixes \( a \ simplex b :: \int \)
assumes $a$ dvd $b$ and $b$ dvd $a$
shows $|a| = |b|$
proof (cases $a = 0$)
  case True
  with assms show ?thesis by simp
next
  case False
  from $(a$ dvd $b)$ obtain $k$ where $k$: $b = a * k$
    unfolding dvd-def by blast
  from $(b$ dvd $a)$ obtain $k'$ where $k'$: $a = b * k'$
    unfolding dvd-def by blast
  from $k$ $k'$ have $a = a * k * k'$ by simp
  with $mult-cancel-left1$ [where $c=a$ and $b=k*k'$] have $kk'$: $k * k' = 1$
    using $(a \neq 0)$ by (simp add: mult_assoc)
  then have $k = 1 \land k' = 1 \lor k = -1 \land k' = -1$
    by (simp add: zmult_eq_1_iff)
  with $k$ $k'$ show ?thesis by auto
qed

lemma zdvd-zdiffD: $k$ dvd $m - n$ $\Rightarrow$ $k$ dvd $n$ $\Rightarrow$ $k$ dvd $m$
  for $k$ $m$ $n$ :: int
  using dvd-add-right-iff [of $k$ $- n$ $m$] by simp

lemma zdvd-reduce: $k$ dvd $n + k * m$ $\iff$ $k$ dvd $n$
  for $k$ $m$ $n$ :: int
  using dvd-add-times-triv-right-iff [of $k$ $n$ $m$] by (simp add: ac-simps)

lemma dvd-imp-le-int:
  fixes $d$ $i$ :: int
  assumes $i \neq 0$ and $d$ dvd $i$
  shows $|d| \leq |i|$
proof
  from $(d$ dvd $i)$ obtain $k$ where $i = d * k$ ..
    with $(i \neq 0)$ have $k \neq 0$ by auto
    then have $1 \leq |k|$ and $0 \leq |d|$ by auto
    then have $|d| * 1 \leq |d| * |k|$ by (rule mult-left-mono)
    with $(i = d * k)$ show ?thesis by (simp add: abs_mult)
qed

lemma zdvd-not-zless:
  fixes $m$ $n$ :: int
  assumes $0 < m$ and $m < n$
  shows $\neg$ $n$ dvd $m$
proof
  from assms have $0 < n$ by auto
  assume $n$ dvd $m$ then obtain $k$ where $k$: $m = n * k$ ..
    with $(0 < m)$ have $0 < n * k$ by auto
    with $(0 < n)$ have $0 < k$ by (simp add: zero-less-mul-iff)
    with $k$ $0 < n$ $(m < n)$ have $n * k < n * 1$ by simp
with \(0 < n\) \(0 < k\) show False unfolding mult-less-cancel-left by auto

qed

lemma zdvd-mult-cancel:
  fixes k m n :: int
  assumes d: k * m dvd k * n
      and k \(\neq 0\)
  shows m dvd n

proof –
  from d obtain h where h: k * n = k * m * h
    unfolding dvd-def by blast
  have n = m * h
    proof (rule ccontr)
      assume \(\neg\) ?thesis
      with \(k \neq 0\) have k * n \(\neq k * (m * h)\) by simp
      with h show False
      by (simp add: mult.assoc)
    qed
  then show ?thesis by simp
qed

lemma int-dvd-int-iff [simp]:
  int m dvd int n \(\iff\) int m dvd n

proof –
  have m dvd n if int n = int m * k for k
    proof (cases k)
      case (nonneg q)
      with that have n = m * q
      by (simp del: of-nat-mult add: of-nat-mult [symmetric])
      then show ?thesis ..
    next
      case (neg q)
      with that have int n = int m * (- int (Suc q))
      by simp
      also have \(\ldots\) = - (int m * int (Suc q))
      by (simp only: mult-minus-right)
      also have \(\ldots\) = - int (m * Suc q)
      by (simp only: of-nat-mult [symmetric])
      finally have - int (m * Suc q) = int n ..
      then show ?thesis
      by (simp only: negative-eq-positive) auto
    qed
  then show ?thesis by (auto simp add: dvd-def)
qed

lemma dvd-nat-abs-iff [simp]:
  n dvd nat \(|k|\) \(\iff\) int n dvd \(|k|\)

proof –
  have n dvd nat \(|k|\) \(\iff\) int n dvd int \(|nat \(|k|\)\)
by (simp only: int-dvd-int-iff)
then show \(?\text{thesis}\)
  by simp
qed

lemma nat-abs-dvd-iff [simp]:
  nat \(|k| \vdash n \iff k \vdash \text{int } n\)
proof
  have nat \(|k| \vdash n \iff \text{int (nat } |k|) \vdash \text{int } n\)
    by (simp only: int-dvd-int-iff)
  then show \(?\text{thesis}\)
    by simp
qed

lemma zdvd1-eq [simp]: \(x \vdash 1 \iff |x| = 1\) (is \(?\text{lhs} \iff ?\text{rhs}\))
  for \(x :: \text{int}\)
proof
  assume \(?\text{lhs}\)
  then have nat \(|x| \vdash \text{nat } |1|\)
    by (simp only: nat-abs-dvd-iff) simp
  then have nat \(|x| = 1\)
    by simp
  then show \(?\text{rhs}\)
    by (cases \(x < 0\)) simp-all
next
  assume \(?\text{rhs}\)
  then have \(x = 1 \lor x = -1\)
    by auto
  then show \(?\text{lhs}\)
    by (auto intro: dvdI)
qed

lemma zdvd-mult-cancel1:
  fixes \(m :: \text{int}\)
  assumes mp: \(m \neq 0\)
  shows \(m * n \vdash m \iff |n| = 1\)
    (is \(?\text{lhs} \iff ?\text{rhs}\))
proof
  assume \(?\text{rhs}\)
  then show \(?\text{lhs}\)
    by (cases \(n > 0\)) (auto simp add: minus-equation-iff)
next
  assume \(?\text{lhs}\)
  then have \(m * n \vdash m * 1\) by simp
  from zdvd-mult-cancel[OF this mp] show \(?\text{rhs}\)
    by (simp only: zdvd1-eq)
qed

lemma nat-dvd-iff: nat \(z \vdash m \iff (0 \leq z \then z \vdash \text{int } m \else m = 0\))
using nat-abs-dvd-iff [of z m] by (cases z ≥ 0) auto

lemma eq-nat-nat-iff: 0 ≤ z → 0 ≤ z' → nat z = nat z' ↔ z = z'
by (auto elim: nonneg-int-cases)

lemma nat-power-eq: 0 ≤ z → nat (z ^ n) = nat z ^ n
by (induct n) (simp-all add: nat-mult-distrib)

lemma numeral-power-eq-nat-cancel-iff [simp]:
numeral x ^ n = nat y ↔ numeral x ^ n = y
using nat-eq-iff2 by auto

lemma nat-eq-numeral-power-cancel-iff [simp]:
nat y = numeral x ^ n ↔ y = numeral x ^ n
using numeral-power-eq-nat-cancel-iff[of x n y]
by (metis (mono-tags))

lemma numeral-power-le-nat-cancel-iff [simp]:
numeral x ^ n ≤ nat a ↔ numeral x ^ n ≤ a
using nat-le-eq-zle[of numeral x ^ n a]
by (auto simp: nat-power-eq)

lemma nat-le-numeral-power-cancel-iff [simp]:
nat a ≤ numeral x ^ n ↔ a ≤ numeral x ^ n
by (simp add: nat-le-iff)

lemma numeral-power-less-nat-cancel-iff [simp]:
numeral x ^ n < nat a ↔ numeral x ^ n < a
using nat-less-eq-zless[of numeral x ^ n a]
by (auto simp: nat-power-eq)

lemma nat-less-numeral-power-cancel-iff [simp]:
nat a < numeral x ^ n ↔ a < numeral x ^ n
using nat-less-eq-zless[of a numeral x ^ n]
by (cases a < 0) (auto simp: nat-power-eq less-le-trans[where y=0])

lemma zdvd-imp-le: z dvd n → 0 < n → z ≤ n
for n z :: int
apply (cases n)
apply auto
apply (cases z)
apply (auto simp add: dvd-imp-le)
done

lemma zdvd-period:
fixes a d :: int
assumes a dvd d
shows a dvd (x + t) ↔ a dvd ((x + c * d) + t)
(is ?lhs ↔ ?rhs)
proof
  from assms have \( a \text{ dvd } (x + t) \iff a \text{ dvd } ((x + t) + c \cdot d) \)
  by (simp add: dvd-add-left-iff)
then show \(?\text{thesis}\)
  by (simp add: ac-simps)
qed

52.18 Finiteness of intervals

lemma finite-interval-int1 \([\text{iff}]: \text{finite } \{ i :: \text{int}. \ a \leq i \land i \leq b \}\)
proof \((\text{cases } a \leq b)\)
  case True
  then show \(?\text{thesis}\)
    proof
      (induct b rule: int-ge-induct)
      case base
      have \(\{ i. \ a \leq i \land i \leq a \} = \{ a \}\) by auto
      then show \(?\text{case}\) by simp
    next
      case (step b)
      then have \(\{ i. \ a \leq i \land i \leq b + 1 \} = \{ i. \ a \leq i \land i \leq b \} \cup \{ b + 1 \}\) by auto
      with step show \(?\text{case}\) by simp
    qed
next
  case False
  then show \(?\text{thesis}\)
    by (metis (lifting, no-types) Collect-empty-eq finite.emptyI order-trans)
qed

lemma finite-interval-int2 \([\text{iff}]: \text{finite } \{ i :: \text{int}. \ a \leq i \land i < b \}\)
  by (rule rev-finite-subset[OF finite-interval-int1[of a b]]) auto

lemma finite-interval-int3 \([\text{iff}]: \text{finite } \{ i :: \text{int}. \ a < i \land i \leq b \}\)
  by (rule rev-finite-subset[OF finite-interval-int1[of a b]]) auto

lemma finite-interval-int4 \([\text{iff}]: \text{finite } \{ i :: \text{int}. \ a < i \land i < b \}\)
  by (rule rev-finite-subset[OF finite-interval-int1[of a b]]) auto

52.19 Configuration of the code generator

Constructors

definition Pos :: num \Rightarrow int
  where \([\text{simp, code-abbrev}]: \text{Pos } = \text{numeral}\)
definition Neg :: num \Rightarrow int
  where \([\text{simp, code-abbrev}]: \text{Neg } n = - (\text{Pos } n)\)

code-datatype 0::int Pos Neg

Auxiliary operations.
THEORY “Int”

**definition** dup :: int ⇒ int
  where [simp]: dup k = k + k

**lemma** dup-code [code]:
  dup 0 = 0
  dup (Pos n) = Pos (Num.Bit0 n)
  dup (Neg n) = Neg (Num.Bit0 n)
  by (simp-all add: numeral-Bit0)

**definition** sub :: num ⇒ num ⇒ int
  where [simp]: sub m n = numeral m - numeral n

**lemma** sub-code [code]:
  sub Num.One Num.One = 0
  sub (Num.Bit0 m) Num.One = Pos (Num.BitM m)
  sub (Num.Bit1 m) Num.One = Pos (Num.Bit0 m)
  sub Num.One (Num.Bit0 n) = Neg (Num.BitM n)
  sub Num.One (Num.Bit1 n) = Neg (Num.Bit0 n)
  sub (Num.Bit0 m) (Num.Bit0 n) = dup (sub m n)
  sub (Num.Bit1 m) (Num.Bit1 n) = dup (sub m n)
  sub (Num.Bit1 m) (Num.Bit0 n) = dup (sub m n) + 1
  sub (Num.Bit0 m) (Num.Bit1 n) = dup (sub m n) - 1
  by (simp-all only: sub-def dup-def numeral.simps Pos-def Neg-def numeral-BitM)

Implementations.

**lemma** one-int-code [code]: 1 = Pos Num.One
  by simp

**lemma** plus-int-code [code]:
  k + 0 = k
  0 + l = l
  Pos m + Pos n = Pos (m + n)
  Pos m + Neg n = sub m n
  Neg m + Pos n = sub n m
  Neg m + Neg n = Neg (m + n)
  for k l :: int
  by simp-all

**lemma** uminus-int-code [code]:
  uminus 0 = (0::int)
  uminus (Pos m) = Neg m
  uminus (Neg m) = Pos m
  by simp-all

**lemma** minus-int-code [code]:
  k - 0 = k
  0 - l = uminus l
  Pos m - Pos n = sub m n
  Pos m - Neg n = Pos (m + n)
\[ \text{Neg } m - \text{Pos } n = \text{Neg } (m + n) \]
\[ \text{Neg } m - \text{Neg } n = \text{sub } n \text{ m} \]
\[ 
\text{for } k \ l :: \text{int} \\
\text{by simp-all} \\
\]

\text{lemma times-int-code \ [code]}:
\[ k \ast 0 = 0 \]
\[ 0 \ast l = 0 \]
\[ \text{Pos } m \ast \text{Pos } n = \text{Pos } (m \ast n) \]
\[ \text{Pos } m \ast \text{Neg } n = \text{Neg } (m \ast n) \]
\[ \text{Neg } m \ast \text{Pos } n = \text{Neg } (m \ast n) \]
\[ \text{Neg } m \ast \text{Neg } n = \text{Pos } (m \ast n) \]
\[ \text{for } k \ l :: \text{int} \\
\text{by simp-all} \\
\]

\text{instantiation int :: equal} \\
\text{begin} \\
\text{definition HOL.equal } k \ l \leftarrow k = (l::\text{int}) \\
\text{instance} \\
\text{by standard (rule equal-int-def)} \\
\text{end} \\

\text{lemma equal-int-code \ [code]}:
\[ \text{HOL.equal } 0 \ (0::\text{int}) \leftarrow \text{True} \]
\[ \text{HOL.equal } 0 \ (\text{Pos } l) \leftarrow \text{False} \]
\[ \text{HOL.equal } 0 \ (\text{Neg } l) \leftarrow \text{False} \]
\[ \text{HOL.equal } (\text{Pos } k) \ (0) \leftarrow \text{False} \]
\[ \text{HOL.equal } (\text{Pos } k) \ (\text{Pos } l) \leftarrow \text{HOL.equal } k \ l \]
\[ \text{HOL.equal } (\text{Pos } k) \ (\text{Neg } l) \leftarrow \text{False} \]
\[ \text{HOL.equal } (\text{Neg } k) \ (0) \leftarrow \text{False} \]
\[ \text{HOL.equal } (\text{Neg } k) \ (\text{Pos } l) \leftarrow \text{False} \]
\[ \text{HOL.equal } (\text{Neg } k) \ (\text{Neg } l) \leftarrow \text{HOL.equal } k \ l \]
\text{by (auto simp add: equal)} \\

\text{lemma equal-int-refl \ [code nbe]}: \text{HOL.equal } k \ k \leftarrow \text{True} \\
\text{for } k :: \text{int} \\
\text{by (fact equal-refl)} \\

\text{lemma less-eq-int-code \ [code]}:
\[ 0 \leq (0::\text{int}) \leftarrow \text{True} \]
\[ 0 \leq \text{Pos } l \leftarrow \text{True} \]
\[ 0 \leq \text{Neg } l \leftarrow \text{False} \]
\[ \text{Pos } k \leq 0 \leftarrow \text{False} \]
\[ \text{Pos } k \leq \text{Pos } l \leftarrow k \leq l \]
\[ \text{Pos } k \leq \text{Neg } l \leftarrow \text{False} \]
\[ \text{Neg } k \leq 0 \leftarrow \text{True} \]
Neg k ≤ Pos l ↔ True
Neg k ≤ Neg l ↔ l ≤ k
by simp-all

lemma less-int-code [code]:
0 < (0::int) ↔ False
0 < Pos l ↔ True
0 < Neg l ↔ False
Pos k < 0 ↔ False
Pos k < Pos l ↔ k < l
Pos k < Neg l ↔ False
Neg k < 0 ↔ True
Neg k < Pos l ↔ True
Neg k < Neg l ↔ l < k
by simp-all

lemma nat-code [code]:
nat (Int.Neg k) = 0
nat 0 = 0
nat (Int.Pos k) = nat-of-num k
by (simp-all add: nat-of-num-numeral)

lemma (in ring-1) of-int-code [code]:
of-int (Int.Neg k) = − numeral k
of-int 0 = 0
of-int (Int.Pos k) = numeral k
by simp-all

Serializer setup.
code-identifier
code-module Int → (SML) Arith and (OCaml) Arith and (Haskell) Arith
quickcheck-params [default-type = int]
hide-const (open) Pos Neg sub dup
De-register int as a quotient type:
lifting-update int.lifting
lifting-forget int.lifting

52.20 Duplicates
lemmas int-sum = of-nat-sum [where 'a=int]
lemmas int-prod = of-nat-prod [where 'a=int]
lemmas zle-int = of-nat-le-iff [where 'a=int]
lemmas int-int-eq = of-nat-eq-iff [where 'a=int]
lemmas nonneg-eq-int = nonneg-int-cases
lemmas double-eq-0-iff = double-zero
lemmas int-distrib =
  distrib-right [of z1 z2 w]
  distrib-left [of w z1 z2]
  left-diff-distrib [of z1 z2 w]
  right-diff-distrib [of w z1 z2]
for z1 z2 w :: int

53 Big infimum (minimum) and supremum (maximum) over finite (non-empty) sets

theory Lattices-Big
  imports Option
begin

53.1 Generic lattice operations over a set

53.1.1 Without neutral element

locale semilattice-set = semilattice
begin

interpretation comp-fun-idem f
  by standard (simp-all add: fun-eq-iff left-commute)

definition F :: 'a set ⇒ 'a
where
  eq-fold': F A = the (Finite-Set.fold (λx y. Some (case y of None ⇒ x | Some z ⇒ f x z)) None A)

lemma eq-fold:
  assumes finite A
  shows F (insert x A) = Finite-Set.fold f x A
proof (rule sgm)
  let 'f = λx y. Some (case y of None ⇒ x | Some z ⇒ f x z)
  interpret comp-fun-idem 'f
    by standard (simp-all add: fun-eq-iff commute left-commute split: option.split)
  from assms show Finite-Set.fold f x A = F (insert x A)
proof induct
  case empty then show ?case by (simp add: eq-fold')
next
  case (insert y B) then show ?case by (simp add: insert-commute [of x] eq-fold')
qed

lemma singleton [simp]:
  F {x} = x
by (simp add: eq-fold)

lemma insert-not-elem:
  assumes finite A and x \notin A and A \neq {}
  shows F (insert x A) = x * F A
proof -
  from (A \neq {}) obtain b where b \in A by blast
  then obtain B where *: A = insert b B b \notin B by (blast dest: mk-disjoint-insert)
  with (finite A) and (x \notin A)
    have finite (insert x B) and b \notin insert x B by auto
  then have F (insert b (insert x B)) = x * F (insert b B)
    by (simp add: eq-fold)
  then show ?thesis by (simp add: * insert-commute)
qed

lemma in-idem:
  assumes finite A and x \in A
  shows x * F A = F A
proof -
  from assms have A \neq {} by auto
  with (finite A) show ?thesis using (x \in A)
    by (induct A rule: finite-ne-induct) (auto simp add: ac-simps insert-not-elem)
qed

lemma insert [simp]:
  assumes finite A and A \neq {}
  shows F (insert x A) = x * F A
using assms by (cases x \in A) (simp-all add: insert-absorb in-idem insert-not-elem)

lemma union:
  assumes finite A A \neq {} and finite B B \neq {}
  shows F (A \cup B) = F A * F B
using assms by (induct A rule: finite-ne-induct) (simp-all add: ac-simps)

lemma remove:
  assumes finite A and x \in A
  shows F A = (if A - {x} = {} then x else x * (A - {x}))
proof -
  from assms obtain B where A = insert x B and x \notin B by (blast dest: mk-disjoint-insert)
  with assms show ?thesis by simp
qed

lemma insert-remove:
  assumes finite A
  shows F (insert x A) = (if A - {x} = {} then x else x * (A - {x}))
using assms by (cases x \in A) (simp-all add: insert-absorb remove)

lemma subset:
assumes finite \( A B \neq \{\} \) and \( B \subseteq A \)
shows \( F B * F A = F A \)
proof –
from \( \text{assms} \) have \( A \neq \{\} \) and finite \( B \) by \( \text{auto dest: finite-subset} \)
with \( \text{assms} \) show \( \text{thesis} \) by \( \text{simp add: union [symmetric] \text{Un-absorb1}} \)
qed

lemma \( \text{closed} \):
assumes \( \text{finite } A A \neq \{\} \) and elem: \( \forall x y. x \ast y \in \{x, y\} \)
shows \( \text{F } A \in A \)
using \( \langle \text{finite } A \rangle \langle A \neq \{\} \rangle \) proof \( \text{induct rule: finite-ne-induct} \)
\begin{itemize}
  \item case \( \text{singleton} \) then show \( \text{case by simp} \)
  \item next \( \text{case } \text{insert with elem show } \text{case by force} \)
\end{itemize}
qed

lemma \( \text{hom-commute} \):
assumes hom: \( \forall x y. h (x \ast y) = h x \ast h y \)
and \( N: \text{finite } N N \neq \{\} \)
shows \( h (F N) = F (h ' N) \)
using \( N \) proof \( \text{induct rule: finite-ne-induct} \)
\begin{itemize}
  \item case \( \text{singleton} \) thus \( \text{case by simp} \)
  \item next \( \text{case } \text{(insert n N)} \)
    \begin{itemize}
      \item then have \( h (F (\text{insert n N})) = h (n \ast F N) \) by simp
      \item also have \( \ldots = h n \ast h (F N) \) by \( \text{rule hom} \)
      \item also have \( h (F N) = F (h ' N) \) by \( \text{rule insert} \)
      \item also have \( h n \ast \ldots = F (\text{insert } h n) (h ' N) \)
        using \( \text{insert by simp} \)
      \item also have \( \text{insert } (h n) (h ' N) = h ' \text{ insert n N by simp} \)
    \end{itemize}
finally show \( \text{case} \).
\end{itemize}
qed

lemma \( \text{infinite} \): \( \neg \text{finite } A \implies F A = \text{the None} \)
unfolding eq-fold' by \( \text{(cases finite } (\text{UNIV::'}a \text{ set})) \) \( \text{(auto intro: finite-subset fold-infinite)} \)

end

locale semilattice-order-set = binary?: semilattice-order + semilattice-set
begin

lemma \( \text{bounded-iff} \):
assumes finite \( A \) and \( A \neq \{\} \)
shows \( x \leq F A \iff (\forall a \in A. x \leq a) \)
using \( \text{assms by } \text{induct rule: finite-ne-induct} \) simp-all

lemma \( \text{boundedI} \):
assumes finite \( A \)
assumes $A \neq \{\}$  
assumes $\bigwedge a. \ a \in A \implies x \leq a$  
shows $x \leq F A$  
using assms by (simp add: bounded-iff)

lemma boundedE:  
assumes finite $A$ and $A \neq \{\}$ and $x \leq F A$  
obtains $\bigwedge a. \ a \in A \implies x \leq a$  
using assms by (simp add: bounded-iff)

lemma coboundedI:  
assumes finite $A$  
and $a \in A$  
shows $F A \leq a$  
proof –  
from assms have $A \neq \{\}$ by auto  
from ⟨finite $A$⟩ ⟨$A \neq \{\}$⟩ ⟨$a \in A$⟩ show ?thesis  
proof (induct rule: finite-ne-induct)  
  case singleton thus ?case by (simp add: refl)  
next  
  case (insert $x$ $B$)  
  from insert have $a = x \lor a \in B$ by simp  
  then show ?case using insert by (auto intro: coboundedI2)  
qed

lemma subset-imp:  
assumes $A \subseteq B$ and $A \neq \{\}$ and finite $B$  
shows $F B \leq F A$  
proof (cases $A = B$)  
  case True then show ?thesis by (simp add: refl)  
next  
  case False  
  have $B : B = A \cup (B - A)$ using ⟨$A \subseteq B$⟩ by blast  
  then have $F B = F (A \cup (B - A))$ by simp  
  also have $\ldots = F A * F (B - A)$ using False assms by (subst union) (auto intro: finite-subset)  
  also have $\ldots \leq F A$ by simp  
  finally show ?thesis .  
qed

end

53.1.2 With neutral element

locale semilattice-neutr-set = semilattice-neutr
begin

interpretation comp-fun-idem $f$
by standard (simp-all add: fun-eq-iff left-commute)
definition \( F :: \text{ 'a set } \Rightarrow \text{ 'a} \)
where
\[ \text{eq-fold: } F A = \text{Finite-Set.fold} f 1 A \]

lemma infinite [simp]:
\[ \neg \text{finite } A \Rightarrow F A = 1 \]
by (simp add: eq-fold)

lemma empty [simp]:
\[ F \{\} = 1 \]
by (simp add: eq-fold)

lemma insert [simp]:
assumes finite A
shows \( F (\text{insert } x A) = x \ast F A \)
using assms by (simp add: eq-fold)

lemma in-idem:
assumes finite A and \( x \in A \)
shows \( x \ast F A = F A \)
proof –
from assms have \( A \neq \{\} \) by auto
with \( \text{finite } A \) show ?thesis using \( x \in A \)
by (induct A rule: finite-ne-induct) (auto simp add: ac-simps)
qed

lemma union:
assumes finite A and finite B
shows \( F (A \cup B) = F A \ast F B \)
using assms by (induct A) (simp-all add: ac-simps)

lemma remove:
assumes finite A and \( x \in A \)
shows \( F A = x \ast F (A - \{x\}) \)
proof –
from assms obtain \( B \) where \( A = \text{insert } x B \) and \( x \notin B \)
by (blast dest: mk-disjoint-insert)
with assms show ?thesis by simp
qed

lemma insert-remove:
assumes finite A
shows \( F (\text{insert } x A) = x \ast F (A - \{x\}) \)
using assms by (cases \( x \in A \)) (simp-all add: insert-absorb remove)

lemma subset:
assumes finite A and \( B \subseteq A \)
shows $F B \star F A = F A$
proof
  from assms have finite $B$ by (auto dest: finite-subset)
  with assms show ?thesis by (simp add: union [symmetric] Un-absorb1)
qed

lemma closed:
  assumes finite $A$ $A \neq \{\}$ and elem: $\forall x \ y. \ x \star y \in \{x, y\}$
  shows $F A \in A$
using (finite $A$) $\langle A \neq \{\} \rangle$ proof (induct rule: finite-ne-induct)
  case singleton then show ?case by simp
  next
  case insert with elem show ?case by force
qed

end

locale semilattice-order-neutr-set = binary?: semilattice-neutr-order + semilattice-neutr-set
begin

lemma bounded-iff:
  assumes finite $A$
  shows $x \leq F A \iff (\forall a \in A. \ x \leq a)$
  using assms by (induct $A$) simp-all

lemma boundedI:
  assumes finite $A$
  assumes $\forall a. \ a \in A \Rightarrow x \leq a$
  shows $x \leq F A$
  using assms by (simp add: bounded-iff)

lemma boundedE:
  assumes finite $A$ and $x \leq F A$
  obtains $\forall a. \ a \in A \Rightarrow x \leq a$
  using assms by (simp add: bounded-iff)

lemma coboundedI:
  assumes finite $A$
  and $a \in A$
  shows $F A \leq a$
proof
  from assms have $A \neq \{\}$ by auto
  from (finite $A$) $\langle A \neq \{\} \rangle$ $\langle a \in A \rangle$ show ?thesis
  proof (induct rule: finite-ne-induct)
    case singleton thus ?case by (simp add: refl)
  next
    case (insert $x$ $B$)
    from insert have $a = x \vee a \in B$ by simp
    then show ?case using insert by (auto intro: coboundedI2)
  qed
qed

lemma subset-imp:
assumes A ⊆ B and finite B
shows F B ≤ F A
proof (cases A = B)
case True then show ?thesis by (simp add: refl)
next
case False
have B: B = A ∪ (B − A) using (A ⊆ B) by blast
then have F B = F (A ∪ (B − A)) by simp
also have ... = F A * F (B − A) using False assms by (subst union) (auto intro: finite-subset)
also have ... ≤ F A by simp
finally show ?thesis .
qed

end

53.2 Lattice operations on finite sets

context semilattice-inf
begin

sublocale Inf-fin: semilattice-order-set inf less-eq less
defines Inf-fin (∩ fin - [900 ] 900 ) = Inf-fin.F ..
end

context semilattice-sup
begin

sublocale Sup-fin: semilattice-order-set sup greater-eq greater
defines Sup-fin (∪ fin - [900 ] 900 ) = Sup-fin.F ..
end

53.3 Infimum and Supremum over non-empty sets

context lattice
begin

lemma Inf-fin-le-Sup-fin [simp]:
assumes finite A and A ≠ { }
shows ∩ fin.A ≤ ∪ fin.A
proof –
from (A ≠ { }) obtain a where a ∈ A by blast
with \(\text{finite} \ A\) have \(\bigcap f_{\text{fin}} A \leq a\) by (rule \(\text{Inf-fin\_coboundedI}\))
moreover from \(\text{finite} \ A\) have \(a \leq \bigcup f_{\text{fin}} A\) by (rule \(\text{Sup-fin\_coboundedI}\))
ultimately show thesis by (rule order-trans)
qed

\[\text{lemma sup-Inf-absorb (simp)}:\]
\[\text{finite} \ A \implies a \in A \implies \bigcap f_{\text{fin}} A \sqcup a = a\]
by (rule \(\text{sup-absorb2}\)) (rule \(\text{Inf-fin\_coboundedI}\))

\[\text{lemma inf-Sup-absorb (simp)}:\]
\[\text{finite} \ A \implies a \in A \implies a \sqcap \bigcup f_{\text{fin}} A = a\]
by (rule \(\text{inf-absorb1}\)) (rule \(\text{Sup-fin\_coboundedI}\))

end

context distrib-lattice

begin

\[\text{lemma sup-Inf1-distrib:}\]
\[\text{assumes finite} \ A \text{ and} A \neq \{\}\]
\[\text{shows} \sup x (\bigcap f_{\text{fin}} A) = \bigcap f_{\text{fin}} \{\sup x a | a \in A\}\]
using \(\text{assms}\) by (simp add: image-def \(\text{Inf-fin\_hom-commute}\) [where \(h=\text{sup x}\), OF \(\text{sup-inf-distribI}\)]
(rule arg-cong [where \(f=\text{Inf-fin}\)], blast)

\[\text{lemma sup-Inf2-distrib:}\]
\[\text{assumes} \ A: \text{finite} \ A \neq \{\} \text{ and} B: \text{finite} \ B \neq \{\}\]
\[\text{shows} \sup (\bigcap f_{\text{fin}} A) (\bigcap f_{\text{fin}} B) = \bigcap f_{\text{fin}} \{\sup a b | a b \in A \land b \in B\}\]
using \(\text{A proof}\) (induct rule: finite-ne-induct)
case singleton then show ?case
by (simp add: sup-Inf1-distrib [OF \(B\)])

next
case \(\text{insert} \ x \ A\)
have finB: finite \(\{\sup x b | b \in B\}\)
by (rule finite-surj [where \(f=\text{sup x}\), OF \(B(1)\)], auto)
have finAB: finite \(\{\sup a b | a b \in A \land b \in B\}\)
proof
have \(\{\sup a b | a b \in A \land b \in B\} = (\bigcup a \in A. \bigcup b \in B. \{\sup a b\}\)
by blast
thus thesis by(simp add: insert(1) \(B(1)\))
qed
have ne: \(\sup a b | a b \in A \land b \in B\} \neq \{\}\) using \(\text{insert B by blast}\)
have sup \(\bigcap f_{\text{fin}} (\text{insert} \ x \ A)\) (\(\bigcap f_{\text{fin}} B\)) = sup \(\text{inf x} (\bigcap f_{\text{fin}} A) (\bigcap f_{\text{fin}} B)\)
using \(\text{insert by simp}\)
also have \(\ldots = \text{inf} (\sup x (\bigcap f_{\text{fin}} B)) (\sup (\bigcap f_{\text{fin}} A) (\bigcap f_{\text{fin}} B))\) by(rule sup-inf-distrib2)
also have \(\ldots = \text{inf} (\bigcap f_{\text{fin}} \{\sup x b | b \in B\}) (\bigcap f_{\text{fin}} \{\sup a b | a b \in A \land b \in B\})\)
using \(\text{insert by(simp add:sup-Inf1-distrib[OF B])}\)

end
also have \( \bigcap_{f \in \mathbb{N}} \{ \sup x \mid b. b \in B \} \cup \{ \sup a b \mid a b. a \in A \land b \in B \} \)  
(is \(-= \bigcap_{f \in \mathbb{N}}?M)  
using \( B \) insert  
by (simp add: \textit{Inf-fin.union} [OF finB - finAB \textit{ne}])  
also have \( ?M = \{ \sup a b \mid a b. a \in \text{insert} x A \land b \in B \} \)  
by blast  
finally show \( \textit{?case} \).
\end{proof}

lemma \textit{inf-Sup1-distrib}:
assumes \( \text{finite } A \text{ and } A \neq \{\} \)
shows \( \inf x (\bigcup_{f \in \mathbb{N}} A) = \bigcup_{f \in \mathbb{N}} \{ \inf x a. a \in A \} \)
using \( \text{assms by (simp add: image-def Sup-fin.hom-commute [where } h=\text{inf x}, \text{OF inf-sup-distrib1]} ) \)
(rule \textit{arg-cong} [where \( f=\text{Sup-fin}\)], blast)

lemma \textit{inf-Sup2-distrib}:
assumes \( A: \text{finite } A \neq \{\} \text{ and } B: \text{finite } B B \neq \{\} \)
shows \( \inf (\bigcup_{f \in \mathbb{N}} A) (\bigcup_{f \in \mathbb{N}} B) = \bigcup_{f \in \mathbb{N}} \{ \inf a b\mid a b. a \in A \land b \in B \} \)
using \( \text{proof (induct rule: finite-ne-induct)} \)
\( \text{case singleton thus } \textit{?case} \)
\( \text{by (simp add: inf-Sup1-distrib [OF } B\text{])} \)

next
\( \text{case (insert} x A) \)
\( \text{have } \text{finB: finite } \{ \inf x \mid b. b \in B \} \)
\( \text{by (rule finite-surj [where } f=\%b. \text{inf x b}, \text{OF B(1)]}, \text{auto}) \)
\( \text{have } \text{finAB: finite } \{ \inf a b\mid a b. a \in A \land b \in B \} \)
\( \text{proof} \)
\( \text{have } \{ \inf a b\mid a b. a \in A \land b \in B \} = (\bigcup a \in A. \bigcup b \in B. \{ \inf a b\}) \)
\( \text{by blast} \)
\( \text{thus } \textit{?thesis} \text{ by (simp add: insert(1) B(1))} \)
\end{proof}

qed
context complete-lattice
begin

lemma Inf-fin-Inf:
  assumes finite A and A ≠ {} 
  shows ⋂_f∈A = ⋂ A
proof –
  from assms obtain b B where A = insert b B and finite B by auto
  then show ?thesis 
    by (simp add: Inf-fin.eq-fold inf-inf-inf.fold-inf inf.commute [of b])
qed

lemma Sup-fin-Sup:
  assumes finite A and A ≠ {} 
  shows ⨆_f∈A = ⨆ A
proof –
  from assms obtain b B where A = insert b B and finite B by auto
  then show ?thesis 
    by (simp add: Sup-fin.eq-fold sup-sup-sup.fold-sup sup.commute [of b])
qed

end

53.4 Minimum and Maximum over non-empty sets

context linorder
begin

sublocale Min: semilattice-order-set min less-eq less
  + Max: semilattice-order-set max greater-eq greater
defines
  Min = Min.F and Max = Max.F ..
end

syntax
  -MIN1 :: pttrns ⇒ 'b ⇒ 'b 
    ((3MIN_-/-) [0, 10] 10)
  -MIN :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b 
    ((3MIN_∈/-) [0, 0, 10] 10)
  -MAX1 :: pttrns ⇒ 'b ⇒ 'b 
    ((3MAX_-/-) [0, 10] 10)
  -MAX :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b 
    ((3MAX_∈/-) [0, 0, 10] 10)

translations
  MIN x y. f ⇒ MIN x. MIN y. f
  MIN x. f ⇒ CONST Min (CONST range (λx. f))
  MIN x∈A. f ⇒ CONST Min ((λx. f) · A)
  MAX x y. f ⇒ MAX x. MAX y. f
  MAX x. f ⇒ CONST Max (CONST range (λx. f))
  MAX x∈A. f ⇒ CONST Max ((λx. f) · A)
An aside: $\text{Min}/\text{Max}$ on linear orders as special case of $\text{Inf-fin}/\text{Sup-fin}$

**Lemma** $\text{Inf-fin-Min}$:

$\text{Inf-fin} = (\text{Min} :: 'a::{\text{semilattice-inf, linorder}} \to 'a)$

**by** $(\text{simp add: Inf-fin-def Min-def inf-min})$

**Lemma** $\text{Sup-fin-Max}$:

$\text{Sup-fin} = (\text{Max} :: 'a::{\text{semilattice-sup, linorder}} \to 'a)$

**by** $(\text{simp add: Sup-fin-def Max-def sup-max})$

**Context** $\text{linorder}$

**Begin**

**Lemma** $\text{dual-min}$:

$\text{ord.min greater-eq} = \text{max}$

**by** $(\text{auto simp add: ord.min-def max-def fun-eq-iff})$

**Lemma** $\text{dual-max}$:

$\text{ord.max greater-eq} = \text{min}$

**by** $(\text{auto simp add: ord.max-def min-def fun-eq-iff})$

**Lemma** $\text{dual-Min}$:

$\text{linorder.Min greater-eq} = \text{Max}$

**Proof**

**Interpret** dual: $\text{linorder greater-eq} \text{greater}$ **by** $(\text{fact dual-linorder})$

**Show** $\text{?thesis}$ **by** $(\text{simp add: dual.Min-def dual-min Max-def})$

**Qed**

**Lemma** $\text{dual-Max}$:

$\text{linorder.Max greater-eq} = \text{Min}$

**Proof**

**Interpret** dual: $\text{linorder greater-eq} \text{greater}$ **by** $(\text{fact dual-linorder})$

**Show** $\text{?thesis}$ **by** $(\text{simp add: dual.Max-def dual-max Min-def})$

**Qed**

**Lemmas** $\text{Min-singleton} = \text{Min.singleton}$

**Lemmas** $\text{Max-singleton} = \text{Max.singleton}$

**Lemmas** $\text{Min-insert} = \text{Min.insert}$

**Lemmas** $\text{Max-insert} = \text{Max.insert}$

**Lemmas** $\text{Min-Un} = \text{Min.union}$

**Lemmas** $\text{Max-Un} = \text{Max.union}$

**Lemmas** $\text{hom-Min-commute} = \text{Min.hom-commute}$

**Lemmas** $\text{hom-Max-commute} = \text{Max.hom-commute}$

**Lemma** $\text{Min-in}$ **[simp]**:

**Assumes** finite $A$ **and** $A \neq \{\}$

**Shows** $\text{Min} A \in A$

**Using** assms **by** $(\text{auto simp add: min-def Min.closed})$

**Lemma** $\text{Max-in}$ **[simp]**:
assumes finite $A$ and $A \neq \{\}$
shows $\text{Max} \ A \in A$
using assms by (auto simp add: max-def Max.closed)

lemma Min-insert2:
assumes finite $A$ and $\forall b \in A : \Rightarrow a \leq b$
shows $\text{Min} \ (\text{insert} \ a \ A) = a$
proof (cases $A = \{\}$)
  case True
  then show ?thesis by simp
next
  case False
  with (finite $A$) have $\text{Min} \ (\text{insert} \ a \ A) = \text{min} \ a \ (\text{Min} \ A)$
  by simp
moreover from (finite $A$) $A \neq \{\}$ have $a \leq \text{Min} \ A$ by simp
ultimately show ?thesis by (simp add: min.absorb1)
qed

lemma Max-insert2:
assumes finite $A$ and $\forall b \in A : \Rightarrow b \leq a$
shows $\text{Max} \ (\text{insert} \ a \ A) = a$
proof (cases $A = \{\}$)
  case True
  then show ?thesis by simp
next
  case False
  with (finite $A$) have $\text{Max} \ (\text{insert} \ a \ A) = \text{max} \ a \ (\text{Max} \ A)$
  by simp
moreover from (finite $A$) $A \neq \{\}$ have $\text{Max} \ A \leq a$ by simp
ultimately show ?thesis by (simp add: max.absorb1)
qed

lemma Min-le [simp]:
assumes finite $A$ and $x \in A$
shows $\text{Min} \ A \leq x$
using assms by (fact Min.coboundedI)

lemma Max-ge [simp]:
assumes finite $A$ and $x \in A$
shows $x \leq \text{Max} \ A$
using assms by (fact Max.coboundedI)

lemma Min-eqI:
assumes finite $A$
assumes $\forall y : y \in A \Rightarrow y \geq x$
  and $x \in A$
shows $\text{Min} \ A = x$
proof (rule antisym)
from $x \in A$ have $A \neq \{\}$ by auto
with assms show $\text{Min } A \geq x$ by simp
next
from assms show $x \geq \text{Min } A$ by simp
qed

lemma $\text{Max-eqI}$:
assumes $\text{finite } A$
assumes $\forall y. y \in A \implies y \leq x$
and $x \in A$
shows $\text{Max } A = x$
proof (rule antisym)
from $x \in A$ have $A \neq \{\}$ by auto
with assms show $\text{Max } A \leq x$ by simp
next
from assms show $x \leq \text{Max } A$ by simp
qed

lemma $\text{eq-Min-iff}$:
[ [ $\text{finite } A$; $A \neq \{\}$ ] $\implies$ $m = \text{Min } A \iff m \in A \land (\forall a \in A. m \leq a)$ ]
by (meson Min-in Min-le antisym)

lemma $\text{Min-eq-iff}$:
[ [ $\text{finite } A$; $A \neq \{\}$ ] $\implies$ $\text{Min } A = m \iff m \in A \land (\forall a \in A. m \leq a)$ ]
by (meson Min-in Min-le antisym)

lemma $\text{eq-Max-iff}$:
[ [ $\text{finite } A$; $A \neq \{\}$ ] $\implies$ $m = \text{Max } A \iff m \in A \land (\forall a \in A. a \leq m)$ ]
by (meson Max-in Max-ge antisym)

lemma $\text{Max-eq-iff}$:
[ [ $\text{finite } A$; $A \neq \{\}$ ] $\implies$ $\text{Max } A = m \iff m \in A \land (\forall a \in A. a \leq m)$ ]
by (meson Max-in Max-ge antisym)

context
  fixes $A :: 'a$ set
  assumes fin-nonempty: $\text{finite } A \neq \{\}$
begin

lemma $\text{Min-ge-iff}$ [simp]:
$x \leq \text{Min } A \iff (\forall a \in A. x \leq a)$
using fin-nonempty by (fact Min.bounded-iff)

lemma $\text{Max-le-iff}$ [simp]:
$\text{Max } A \leq x \iff (\forall a \in A. a \leq x)$
using fin-nonempty by (fact Max.bounded-iff)

lemma $\text{Min-gr-iff}$ [simp]:
$x < \text{Min } A \iff (\forall a \in A. x < a)$
using fin-nonempty by (induct rule: finite-ne-induct) simp-all
lemma Max-less-iff [simp]:
Max A < x ↔ (∀ a∈A. a < x)
using fin-nonempty by (induct rule: finite-induct) simp-all

lemma Min-le-iff:
Min A ≤ x ↔ (∃ a∈A. a ≤ x)
using fin-nonempty by (induct rule: finite-induct) (simp add: min-iff-disj)

lemma Max-le-iff:
x ≤ Max A ↔ (∃ a∈A. x ≤ a)
using fin-nonempty by (induct rule: finite-induct) (simp add: le-max-iff-disj)

lemma Min-gr-iff:
Min A < x ↔ (∀ a∈A. a < x)
using fin-nonempty by (induct rule: finite-induct) (simp add: less-min-iff-disj)

lemma Max-gr-iff:
x < Max A ↔ (∃ a∈A. x < a)
using fin-nonempty by (induct rule: finite-induct) (simp add: less-max-iff-disj)

end

lemma Max-eq-if:
assumes finite A finite B ∀ a∈A. b∈B. a ≤ b ∀ b∈B. a < b
shows Max A = Max B
proof cases
  assume A = {} thus ?thesis using assms by simp
next
  assume A ≠ {} thus ?thesis using assms by (blast intro: antisym Max-in Max-ge-iff THEN iffD2)
qed

lemma Min-antimono:
assumes M ⊆ N and M ≠ {} and finite N
shows Min N ≤ Min M
using assms by (fact Min.subset-imp)

lemma Max-mono:
assumes M ⊆ N and M ≠ {} and finite N
shows Max M ≤ Max N
using assms by (fact Max.subset-imp)

end

context linorder
begin

lemma mono-Min-commute:
assumes \( \text{mono } f \)
assumes \( \text{finite } A \) and \( A \neq \emptyset \)
shows \( f (\text{Min } A) = \text{Min } (f ' A) \)

\textbf{proof} (rule linorder-class.Min-eqI \([\text{symmetric}]\))
from \( \text{finite } A \) show \( \text{finite } (f ' A) \) by simp
from \( f \) show \( f (\text{Min } A) \in f ' A \) by simp

fix \( x \)
assume \( x \in f ' A \)
then obtain \( y \) where \( y \in A \) and \( x = f y \)
with \( f \) have \( \text{Min } A \leq y \) by auto
with \( \text{mono } f \) have \( f (\text{Min } A) \leq f y \) by (rule monoE)
with \( x = f y \) show \( f (\text{Min } A) \leq x \) by simp

\textit{qed}

\textbf{lemma} mono-Max-commute:
assumes \( \text{mono } f \)
assumes \( \text{finite } A \) and \( A \neq \emptyset \)
shows \( f (\text{Max } A) = \text{Max } (f ' A) \)

\textbf{proof} (rule linorder-class.Max-eqI \([\text{symmetric}]\))
from \( \text{finite } A \) show \( \text{finite } (f ' A) \) by simp
from \( f \) show \( f (\text{Max } A) \in f ' A \) by simp

fix \( x \)
assume \( x \in f ' A \)
then obtain \( y \) where \( y \in A \) and \( x = f y \)
with \( f \) have \( y \leq \text{Max } A \) by auto
with \( \text{mono } f \) have \( f y \leq f (\text{Max } A) \) by (rule monoE)
with \( x = f y \) show \( x \leq f (\text{Max } A) \) by simp

\textit{qed}

\textbf{lemma} finite-linorder-max-induct \([\text{consumes } 1, \text{case-names } empty \text{ insert}]\):
assumes \( \text{fin: finite } A \)
and \( \text{empty: } P \{\} \)
and \( \text{insert: } \bigwedge b A. \text{finite } A \implies \forall a \in A. a < b \implies P A \implies P (\text{insert } b A) \)
shows \( P A \)
using \( \text{fin } empty \text{ insert} \)

\textbf{proof} (induct rule: finite-psubset-induct)
\begin{itemize}
  \item case \( (\text{psubset } A) \)
  \begin{itemize}
    \item have \( IH: \bigwedge B. [B \subset A; P \{}; (\bigwedge A b. [\text{finite } A; \forall a \in A. a < b; P A] \implies P (\text{insert } b A))] \implies P B \text{ by fact} \)
    \item have \( \text{fin: finite } A \text{ by fact} \)
    \item have \( \text{empty: } P \{} \text{ by fact} \)
    \item have \( \text{step: } \bigwedge b A. [\text{finite } A; \forall a \in A. a < b; P A] \implies P (\text{insert } b A) \text{ by fact} \)
  \end{itemize}
  \item show \( P A \)
  \begin{itemize}
    \item proof (cases \( A = \{} \))
    \begin{itemize}
      \item assume \( A = \{} \)
      \begin{itemize}
        \item then show \( P A \text{ using } (P \{}); \text{ by simp} \)
      \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}
next
\begin{itemize}
  \item let \( ?B = A - \{\text{Max } A\} \)
  \item let \( ?A = \text{insert } (\text{Max } A) \ ?B \)
\end{itemize}
have finite ?B using ⟨finite A⟩ by simp
assume A ≠ {} with ⟨finite A⟩ have Max A ∈ A by auto
then have A: ?A = A using insert-Diff-single insert-absorb by auto
then have P ?B using {P {}} step IH [of ?B] by blast
moreover have ∀a∈?B. a < Max A using Max-ge [OF ⟨finite A⟩] by fastforce
ultimately show P A using A insert-Diff-single step [OF ⟨finite ?B⟩] by fastforce
qed

lemma finite-linorder-min-induct [consumes 1, case-names empty insert]:
[finite A; P {}; ∀b A. [finite A; ∀a∈A. b < a; P (insert b A)] ⇒ P (insert b A)] ⇒ P A
by (rule linorder.finite-linorder-max-induct [OF dual-linorder])

lemma finite-ranking-induct [consumes 1, case-names empty insert]:
fixes f :: 'b ⇒ 'a
assumes finite S
assumes P {}
assumes ∀x S. finite S ⇒ (∀y. y ∈ S ⇒ f y ≤ f x) ⇒ P S ⇒ P (insert x S)
supports P S
using ⟨finite S⟩
proof (induction rule: finite-psubset-induct)
case (psubset A)
{ assume A ≠ {}
hence f ′ A ≠ {} and finite (f ′ A)
using psubset finite-image-iff by simp+
then obtain a where f a = Max (f ′ A) and a ∈ A
by (metis Max-in[of f ′ A] imageE)
then have P (A − {a})
using psubset member-remove by blast
moreover have ∀y. y ∈ A ⇒ f y ≤ f a
using f a = Max (f ′ A) ∨ finite (f ′ A) by simp
ultimately have ?case
by (metis (a ∈ A) DiffD1 insert-Diff assms(3) finite-Diff psubset.hyps)
}
thus ?case
using assms(2) by blast
qed

lemma Least-Min:
assumes finite {a. P a} and ∃a. P a
shows (LEAST a. P a) = Min {a. P a}
proof –
{ fix A :: 'a set
  assume A: finite A A ≠ {}
  have (LEAST a. a ∈ A) = Min A
  using A proof (induct A rule: finite-ne-induct)
    case singleton show ?case by (rule Least-equality) simp-all
  next
    case (insert a A)
    have (LEAST b. b = a ∨ b ∈ A) = min a (LEAST a. a ∈ A)
    by (auto intro!: Least-equality simp: min_def not_le Min-le-iff insert dest!: less_imp_le)
    with insert show ?case by simp
  qed
}
lemma infinite-growing:
  assumes X ≠ {}
  assumes ∗: (∀ x. x ∈ X ⇒ ∃ y ∈ X. y > x)
  shows ¬ finite X
proof
  assume finite X
  with ⟨X ≠ {} ⟩ have Max X ∈ X ∀ x ∈ X. x ≤ Max X
  by auto
  with ∗[of Max X] show False
  by auto
qed
end
context linordered-ab-semigroup-add
begin
lemma Min-add-commute:
  fixes k
  assumes finite S and S ≠ {}
  shows Min ((λx. f x + k) ' S) = Min(f ' S) + k
proof –
  have m: (∀x y. min x y + k = min (x+k) (y+k))
  by(simp add: min_def antisym add-right_mono)
  have (λx. f x + k) ' S = (λy. y + k) ' (f ' S) by auto
  also have Min . . = Min (f ' S) + k
  using assms hom-Min-commute [of λy. y+k f ' S, OF m, symmetric] by simp
  finally show ?thesis by simp
qed
lemma Max-add-commute:
  fixes k
  assumes finite S and S ≠ {}
shows $\max ((\lambda x. f x + k) \cdot S) = \max (f \cdot S) + k$

proof –

have $m : \forall x y. \max x y + k = \max (x+k) (y+k)$
  by (simp add: max_def antisym add-right_mono)

have $(\lambda x. f x + k) \cdot S = (\lambda y. y + k) \cdot (f \cdot S)$ by auto

also have $\max \ldots = \max (f \cdot S) + k$
  using assms hom-Max-commute [of $\lambda y. y + k f \cdot S$, OF $m$, symmetric] by simp

finally show ?thesis by simp

qed

end

context linordered-ab-group-add

begin

lemma minus-Max-eq-Min [simp]:
  finite S \Rightarrow S \neq \{\} \Rightarrow -\max S = \min (\uminus \cdot S)
  by (induct S rule: finite-ne-induct) (simp-all add: minus-max-eq-min)

lemma minus-Min-eq-Max [simp]:
  finite S \Rightarrow S \neq \{\} \Rightarrow -\min S = \max (\uminus \cdot S)
  by (induct S rule: finite-ne-induct) (simp-all add: minus-min-eq-max)

end

context complete-linorder

begin

lemma Min-Inf:
  assumes finite A and A \neq \{\}
  shows Min A = Inf A

proof –
  from assms obtain b B where A = insert b B and finite B by auto
  then show ?thesis
    by (simp add: Min.eq-fold complete-linorder-inf-min [symmetric] inf-Inf-fold-inf inf.commute [of b])

qed

lemma Max-Sup:
  assumes finite A and A \neq \{\}
  shows Max A = Sup A

proof –
  from assms obtain b B where A = insert b B and finite B by auto
  then show ?thesis
    by (simp add: Max.eq-fold complete-linorder-sup-max [symmetric] sup-Sup-fold-sup sup.commute [of b])

qed

end
53.5 Arg Min

definition is-arg-min :: ('a ⇒ 'b::ord) ⇒ ('a ⇒ bool) ⇒ 'a ⇒ bool where
  is-arg-min f P x = (P x ∧ ¬(∃y. P y ∧ f y < f x))

definition arg-min :: ('a ⇒ 'b::ord) ⇒ ('a ⇒ bool) ⇒ 'a where
  arg-min f P = (SOME x. is-arg-min f P x)

definition arg-min-on :: ('a ⇒ 'b::ord) ⇒ 'a set ⇒ 'a where
  arg-min-on f S = arg-min f (λx. x ∈ S)

syntax
  ARG-MIN f x. P ⇌ CONST arg-min f (λx. P)

translations

lemma is-arg-min-linorder: fixes f :: 'a ⇒ 'b :: linorder
  shows is-arg-min f P x = (P x ∧ (∀y. P y −→ f x ≤ f y))
  by (auto simp add: is-arg-min-def)

lemma is-arg-min-antimono: fixes f :: 'a ⇒ 'b::order
  shows [ [ is-arg-min f P x; f y ≤ f x; P y ] ] =⇒ is-arg-min f P y
  by (simp add: order.order-iff-strict is-arg-min-def)

lemma arg-minI:
  [ P x; 
    (∀y. P y ⇒ ¬ f y < f x); 
    (∀x. [ P x; ∀y. P y −→ ¬ f y < f x ] =⇒ Q x ]
    =⇒ Q (arg-min f P) ]

apply (simp add: arg-min-def is-arg-min-def)
aply blast
apply blast
done

lemma arg-min-equality:
  [ P k; (∀x. P x =⇒ f k ≤ f x ] =⇒ f (arg-min f P) = f k
  for f :: ⇒ 'a::order
apply (rule arg-minI)
aply assumption
apply (simp add: less-le-not-le)
by (metis le-less)

lemma wf-linord-ex-has-least:
  [ wf r; (∀x y. (x, y) ∈ r+ −→ (y, x) ∉ r*; P k ]
  =⇒ ∃x. P x ∧ (∀y. P y −→ (m x, m y) ∈ r*)
apply (drule wf-trancl [THEN wf-eq-minimal [THEN iffD1]])
aply (drule-tac x = m \ Collect P in spec)
by force
lemma ex-has-least-nat: \( P \, k \, \Rightarrow \, \exists \, x. \, P \, x \, \land \, (\forall \, y. \, P \, y \, \rightarrow \, m \, x \, \leq \, m \, y) \)
for \( m :: \, 'a \Rightarrow \, nat \)
apply (simp only: pred-nat-trancl-eq-le [symmetric])
apply (rule wf-pred-nat [THEN wf-linord-ex-has-least])
apply (simp add: less-eq linorder-not-le pred-nat-trancl-eq-le)
by assumption

lemma arg-min-nat-lemma: \( P \, k \, \Rightarrow \, P \, (\arg\text{-}min \, m \, P) \, \land \, (\forall \, y. \, P \, y \, \rightarrow \, m \, (\arg\text{-}min \, m \, P) \, \leq \, m \, y) \)
for \( m :: \, 'a \Rightarrow \, nat \)
apply (simp add: arg-min-def is-arg-min-linorder)
apply (erule ex-has-least-nat)
done

lemmas arg-min-natI = arg-min-nat-lemma [THEN conjunct1]

lemma is-arg-min-arg-min-nat: fixes \( m :: \, 'a \Rightarrow \, nat \)
shows \( P \, x \, \Rightarrow \, is\text{-}arg\text{-}min \, m \, P \, (\arg\text{-}min \, m \, P) \)
by (metis arg-min-nat-lemma is-arg-min-linorder)

lemma arg-min-nat-le: \( P \, x \, \Rightarrow \, m \, (\arg\text{-}min \, m \, P) \, \leq \, m \, x \)
for \( m :: \, 'a \Rightarrow \, nat \)
by (rule arg-min-nat-lemma [THEN conjunct2, THEN spec, THEN mp])

lemma ex-min-if-finite: \( \{ \, \text{finite} \, S; \, S \, \neq \, \{\} \, \} \, \Rightarrow \, \exists \, m \in S. \, \neg(\exists \, x \in S. \, x \, < \, (m::'a::order)) \)
by (induction rule: finite.induct) (auto intro: order.strict-trans)

lemma ex-is-arg-min-if-finite: fixes \( f :: \, 'a \Rightarrow \, 'b :: \, order \)
shows \( \{ \, \text{finite} \, S; \, S \, \neq \, \{\} \, \} \, \Rightarrow \, \exists \, x. \, is\text{-}arg\text{-}min \, f \, (\lambda x. \, x \in S) \, x \)
unfolding is-arg-min-def
using ex-min-if-finite[of \( f \, | \, S \)]
by auto

lemma arg-min-SOME-Min: \( \text{finite} \, S \, \Rightarrow \, \arg\text{-}min\text{-}on \, f \, S = \, (\text{SOME} \, y. \, y \in S \, \land \, f \, y \, = \, \text{Min}(f \, | \, S)) \)
unfolding arg-min-on-def arg-min-def is-arg-min-linorder
apply (rule arg-cong[where \( f = \text{Eps}\)])
apply (auto simp: fun-eq-iff intro: Min-eqI[ symmetric])
done

lemma arg-min-if-finite: fixes \( f :: \, 'a \Rightarrow \, 'b :: \, order \)
assumes finite \( S \, S \, \neq \, \{\} \)
shows \( \arg\text{-}min\text{-}on \, f \, S \, \in \, S \, \land \, \neg(\exists \, x \in S. \, f \, x \, < \, f \, (\arg\text{-}min\text{-}on \, f \, S)) \)
using ex-is-arg-min-if-finite[OF assms, of \( f \)]
unfolding arg-min-on-def arg-min-def is-arg-min-def
by (auto dest!: someI-ex)
lemma arg-min-least: fixes f :: 'a ⇒ 'b :: linorder
shows [ finite S; S ≠ {}; y ∈ S ] ⇒ f(arg-min-on f S) ≤ f y
by(simp add: arg-min-SOME-Min inv-into-def2[ symmetric ] f-inv-into-f)

lemma arg-min-inj-eq: fixes f :: 'a ⇒ 'b :: order
shows [ inj-on f { x. P x }; P a; ∀ y. P y −→ f a ≤ f y ] ⇒ arg-min f P = a
apply(simp add: arg-min-def is-arg-min-def)
apply(rule someI2[of - a])
apply (simp add: less-le-not-le)
by (metis inj-on-eq-iff less-le mem-Collect-eq)

53.6 Arg Max

definition is-arg-max :: ('a ⇒ 'b::ord) ⇒ ('a ⇒ bool) ⇒ 'a ⇒ bool where
is-arg-max f P x = (P x ∧ ¬ (∃ y. P y ∧ f y > f x))

definition arg-max :: ('a ⇒ 'b::ord) ⇒ ('a ⇒ bool) ⇒ 'a where
arg-max f P = (SOME x. is-arg-max f P x)

definition arg-max-on :: ('a ⇒ 'b::ord) ⇒ 'a set ⇒ 'a where
arg-max-on f S = arg-max f (λx. x ∈ S)

syntax
-arg-max :: ('a ⇒ 'b) ⇒ pttrn ⇒ bool ⇒ 'a
  ((λARG' MAX --/ -) [1000, 0, 10] 10)

translations
ARG-MAX f x. P ⇌ CONST arg-max f (λx. P)

lemma is-arg-max-linorder: fixes f :: 'a ⇒ 'b::linorder
shows is-arg-max f P x = (P x ∧ (∀ y. P y −→ f x ≥ f y))
by(auto simp add: is-arg-max-def)

lemma arg-maxI:
P x =⇒
  (∀ y. P y −→ ¬ f y > f x) =⇒
  (∀ x. P x =⇒ ∀ y. P y −→ ¬ f y > f x =⇒ Q x) =⇒
  Q (arg-max f P)
apply (simp add: arg-max-def is-arg-max-def)
apply (rule someI2-ex)
apply blast
apply blast
done

lemma arg-max-equality:
[ P k; ∀ x. P x =⇒ f x ≤ f k ] =⇒ f (arg-max f P) = f k
for f :: - ⇒ 'a::order
apply (rule arg-maxI [where f = f])
apply assumption
apply (simp add: less-le-not-le)
by (metis le-less)

lemma ex-has-greatest-nat-lemma:
P k \Rightarrow \forall x. P x \Rightarrow (\exists y. P y \land \neg f y \leq f x) \Rightarrow \exists y. P y \land \neg f y < f k + n
for f :: 'a \Rightarrow nat
by (induct n) (force simp: le-Suc-eq)+

lemma ex-has-greatest-nat:
P k \Rightarrow \forall y. P y \Rightarrow f y < b \Rightarrow \exists x. P x \land (\forall y. P y \Rightarrow f y \leq f x)
for f :: 'a \Rightarrow nat
apply (rule ccontr)
apply (cut-tac P = P and n = b - f k in ex-has-greatest-nat-lemma)
apply (subgoal-tac [3] f k \leq b)
apply auto
done

lemma arg-max-nat-lemma:
[ P k; \forall y. P y \Rightarrow f y < b ]
\Rightarrow P (arg-max f P) \land (\forall y. P y \Rightarrow f y \leq f (arg-max f P))
for f :: 'a \Rightarrow nat
apply (simp add: arg-max-def is-arg-max-linorder)
apply (rule someI-ex)
apply (erule (1) ex-has-greatest-nat)
done

lemmas arg-max-natI = arg-max-nat-lemma [THEN conjunct1]

lemma arg-max-nat-le: P x \Rightarrow \forall y. P y \Rightarrow f y < b \Rightarrow f x \leq f (arg-max f P)
for f :: 'a \Rightarrow nat
by (blast dest: arg-max-nat-lemma [THEN conjunct2, THEN spec, of P])

end

54 Division in euclidean (semi)rings

theory Euclidean-Division
  imports Int Lattices-Big
begin

54.1 Euclidean (semi)rings with explicit division and remainder

class euclidean-semiring = semidom-modulo +
fixes euclidean-size :: 'a \Rightarrow nat
assumes size-0 [simp]: euclidean-size 0 = 0
assumes mod-size-less:
  b \neq 0 \Rightarrow euclidean-size (a mod b) < euclidean-size b
assumes size-mult-mono:
\[ b \neq 0 \implies \text{euclidean-size } a \leq \text{euclidean-size } (a \ast b) \]

begin

lemma euclidean-size-eq-0-iff [simp]:
\[ \text{euclidean-size } b = 0 \iff b = 0 \]

proof
assume \( b = 0 \)
then show \( \text{euclidean-size } b = 0 \)
  by simp
next
assume \( \text{euclidean-size } b = 0 \)
show \( b = 0 \)
proof (rule ccontr)
assume \( b \neq 0 \)
with mod-size-less have \( \text{euclidean-size } (b \mod b) < \text{euclidean-size } b \).
with \( \text{euclidean-size } b = 0 \) show \( \text{False} \)
  by simp
qed
qed

lemma euclidean-size-greater-0-iff [simp]:
\[ \text{euclidean-size } b > 0 \iff b \neq 0 \]

using euclidean-size-eq-0-iff [symmetric, of \( b \)] by safe simp

lemma size-mult-mono': \( b \neq 0 \implies \text{euclidean-size } a \leq \text{euclidean-size } (b \ast a) \)

by (subst mult.commute) (rule size-mult-mono)

lemma dvd-euclidean-size-eq-imp-dvd:
assumes \( a \neq 0 \) and \( \text{euclidean-size } a = \text{euclidean-size } b \)
and \( b \text{ dvd } a \)
shows \( a \text{ dvd } b \)
proof (rule ccontr)
assume \( \neg \ a \text{ dvd } b \)
hence \( b \mod a \neq 0 \) using mod-0-imp-dvd [of \( b \ a \)] by blast
then have \( b \mod a \neq 0 \) by (simp add: mod-0-iff-dvd)
from \( \langle b \text{ dvd } a \rangle \) have \( b \text{ dvd } b \mod a \) by (simp add: dvd-mod-iff)
then obtain \( c \) where \( b \mod a = b \ast c \) unfolding dvd-def by blast
with \( \langle b \mod a \neq 0 \rangle \) have \( c \neq 0 \) by auto
with \( \langle b \mod a = b \ast c \rangle \) have \( \text{euclidean-size } (b \mod a) \geq \text{euclidean-size } b \)
using size-mult-mono by force
moreover from \( \langle \neg \ a \text{ dvd } b \rangle \) and \( \langle a \neq 0 \rangle \)
have \( \text{euclidean-size } (b \mod a) < \text{euclidean-size } a \)
using mod-size-less by blast
ultimately show \( \text{False} \) using \( \langle \text{euclidean-size } a = \text{euclidean-size } b \rangle \)
  by simp
qed

lemma euclidean-size-times-unit:
assumes is-unit \( a \)
shows \( \text{euclidean-size} \ (a \ast b) = \text{euclidean-size} \ b \)

proof (rule antisym)
from assms have [simp]: \( a \neq 0 \) by auto
thus \( \text{euclidean-size} \ (a \ast b) \geq \text{euclidean-size} \ b \) by (rule size-mult-mono')
from assms have is-unit \( (1 \div a) \) by simp
hence \( 1 \div a \neq 0 \) by (intro notI) simp-all
hence \( \text{euclidean-size} \ (a \ast b) \leq \text{euclidean-size} \ ((1 \div a) \ast (a \ast b)) \)
by (rule size-mult-mono')
also from assms have \( (1 \div a) \ast (a \ast b) = b \)
by (simp add: algebra-simps unit-div-mult-swap)
finally show \( \text{euclidean-size} \ (a \ast b) \leq \text{euclidean-size} \ b \).
qed

lemma euclidean-size-unit:
\( \text{is-unit} \ a \implies \text{euclidean-size} \ a = \text{euclidean-size} \ 1 \)
using euclidean-size-times-unit [of a 1] by simp

lemma unit-iff-euclidean-size:
\( \text{is-unit} \ a \iff \text{euclidean-size} \ a = \text{euclidean-size} \ 1 \land a \neq 0 \)
proof safe
assume A: \( a \neq 0 \) and B: \( \text{euclidean-size} \ a = \text{euclidean-size} \ 1 \)
show is-unit a
by (rule dvd-euclidean-size-eq-imp-dvd [OF A B]) simp-all
qed (auto intro: euclidean-size-unit)

lemma euclidean-size-times-nonunit:
assumes \( a \neq 0 \ \land \ b \neq 0 \ \land \ \neg \text{is-unit} \ a \)
shows \( \text{euclidean-size} \ b < \text{euclidean-size} \ (a \ast b) \)
proof (rule ccontr)
assume \( \neg \text{euclidean-size} \ b < \text{euclidean-size} \ (a \ast b) \)
with size-mult-mono [OF assms], of b
have eq: \( \text{euclidean-size} \ (a \ast b) = \text{euclidean-size} \ b \) by simp
have a \( \ast b \ \text{dvd} \ b \)
by (rule dvd-euclidean-size-imp-dvd [OF eq]) (insert assms, simp-all)
hence a \( \ast b \ \text{dvd} \ 1 \ \ast b \) by simp
with \( b \neq 0 \) have is-unit a by (subst (asm) dvd-times-right-cancel-iff)
with assms(3) show False by contradiction
qed

lemma dvd-imp-size-le:
assumes \( a \ \text{dvd} \ b \ \land \ b \neq 0 \)
shows \( \text{euclidean-size} \ a \leq \text{euclidean-size} \ b \)
using assms by (auto elim!: dvdE simp: size-mult-mono)

lemma dvd-proper-imp-size-less:
assumes \( a \ \text{dvd} \ b \ \land \ b \neq 0 \)
shows \( \text{euclidean-size} \ a < \text{euclidean-size} \ b \)
proof –
from assms(1) obtain c where \( b = a \ast c \) by (erule dvdE)
hence $z: b = c * a$ by (simp add: mult.commute)
from $z$ assms have $\neg$is-unit $c$ by (auto simp: mult.commute mult-unit-dvd-iff)
with $z$ assms show $?thesis$
  by (auto intro!: euclidean-size-times-nonunit)
qed

lemma unit-imp-mod-eq-0:
  $a \mod b = 0$ if is-unit $b$
using that by (simp add: mod-eq-0-iff-dvd unit-imp-dvd)

lemma mod-eq-self-iff-div-eq-0:
  $a \mod b = a \longleftrightarrow a \div b = 0$ (is $?P \longleftrightarrow ?Q$)
proof
  assume $?P$
  with div-mult-mod-eq $[\text{of } a, b]$ show $?Q$
    by auto
next
  assume $?Q$
  with div-mult-mod-eq $[\text{of } a, b]$ show $?P$
    by simp
qed

lemma coprime-mod-left-iff [simp]:
  coprime $(a \mod b)$ $b \longleftrightarrow$ coprime $a$ $b$ if $b \neq 0$
by (rule; rule coprimeI)
  (use that in ⟨auto dest!: dvd-mod-imp-dvd coprime-common-divisor simp add: dvd-mod-iff⟩)

lemma coprime-mod-right-iff [simp]:
  coprime $a$ $(b \mod a)$ $\longleftrightarrow$ coprime $a$ $b$ if $a \neq 0$
using that coprime-mod-left-iff $[\text{of } a, b]$ by (simp add: ac-simps)
end

class euclidean-ring = idom-modulo + euclidean-semiring
begin

lemma dvd-diff-commute [ac-simps]:
  $a$ dvd $c - b \longleftrightarrow a$ dvd $b - c$
proof
  have $a$ dvd $c - b \longleftrightarrow a$ dvd $(c - b) * - 1$
    by (subst dvd-mult-unit-iff) simp-all
  then show $?thesis$
    by simp
qed

end
54.2 Euclidean (semi)rings with cancel rules

class euclidean-semiring-cancel = euclidean-semiring + 
assumes div-mult-self1 [simp]: b ≠ 0 ⟹ (a + c * b) div b = c + a div b 
and div-mult-mult1 [simp]: c ≠ 0 ⟹ (c * a) div (c * b) = a div b 
begin

lemma div-mult-self2 [simp]:
assumes b ≠ 0
shows (a + b * c) div b = c + a div b
using assms div-mult-self1 [of b a c] by (simp add: mult.commute)

lemma div-mult-self3 [simp]:
assumes b ≠ 0
shows (c * b + a) div b = c + a div b
using assms by (simp add: add.commute)

lemma div-mult-self4 [simp]:
assumes b ≠ 0
shows (b * c + a) div b = c + a div b
using assms by (simp add: add.commute)

lemma mod-mult-self1 [simp]: (a + c * b) mod b = a mod b
proof (cases b = 0)
case True then show ?thesis by simp
next
  case False
  have a + c * b = (a + c * b) div b * b + (a + c * b) mod b
    by (simp add: div-mult-mod-eq)
  also from False div-mult-self1 [of b a c] have
    ... = (c + a div b) * b + (a + c * b) mod b
    by (simp add: algebra-simps)
  finally have a = a div b * b + (a + c * b) mod b
    by (simp add: add.commute [of a] add.assoc distrib-right)
  then have a div b * b + (a + c * b) mod b = a div b * b + a mod b
    by (simp add: div-mult-mod-eq)
  then show ?thesis by simp
qed

lemma mod-mult-self2 [simp]:
(a + b * c) mod b = a mod b
by (simp add: mult.commute [of b])

lemma mod-mult-self3 [simp]:
(c * b + a) mod b = a mod b
by (simp add: add.commute)

lemma mod-mult-self4 [simp]:
(b * c + a) mod b = a mod b
by (simp add: add.commute)
lemma mod-mult-self1-is-0 [simp]:
  \( b \cdot a \mod b = 0 \)
using mod-mult-self2 [of \( 0 \) \( b \) \( a \)] by simp

lemma mod-mult-self2-is-0 [simp]:
  \( a \cdot b \mod b = 0 \)
using mod-mult-self1 [of \( 0 \) \( a \) \( b \)] by simp

lemma div-add-self1:
assumes \( b \neq 0 \)
shows \((b + a) \div b = a \div b + 1\)
using assms div-mult-self1 [of \( b \) \( a \) \( 1 \)] by (simp add: add.commute)

lemma div-add-self2:
assumes \( b \neq 0 \)
shows \((a + b) \div b = a \div b + 1\)
using assms div-add-self1 [of \( b \) \( a \)] by (simp add: add.commute)

lemma mod-add-self1 [simp]:
  \((b + a) \mod b = a \mod b\)
using mod-mult-self1 [of \( a \) \( 1 \) \( b \)] by (simp add: add.commute)

lemma mod-add-self2 [simp]:
  \((a + b) \mod b = a \mod b\)
using mod-mult-self1 [of \( a \) \( 1 \) \( b \)] by simp

lemma mod-div-trivial [simp]:
  \( a \mod b \div b = 0 \)
proof (cases \( b = 0 \))
  assume \( b = 0 \)
  thus \(?thesis\) by simp
next
  assume \( b \neq 0 \)
  hence \( a \div b + a \mod b \div b = (a \mod b + a \div b \times b) \div b\)
    by (rule div-mult-self1 [symmetric])
  also have \( \ldots = a \div b\)
    by (simp only: mod-div-mult-eq)
  also have \( \ldots = a \div b + 0\)
    by simp
finally show \(?thesis\)
  by (rule add-left-imp-eq)
qed

lemma mod-mod-trivial [simp]:
  \( a \mod b \mod b = a \mod b \)
proof
  have \( a \mod b \mod b = (a \mod b + a \div b \times b) \mod b\)
    by (simp only: mod-mult-self1)
also have \ldots = a \mod b 
  by (simp only: mod-div-mult-eq)
finally show \ ?thesis .
qed

lemma mod-mod-cancel:
  assumes \ c \ dvd \ b 
  shows \ a \ mod \ b \ mod \ c = a \ mod \ c 
proof –
  from \ \langle \ c \ dvd \ b \ \rangle 
  obtain \ k 
  where \ b = c \cdot k 
  by (rule dvdE)
  have \ a \ mod \ b \ mod \ c = a \ mod \ (c \cdot k) 
  by (simp only: \ \langle \ b = c \cdot k \ \rangle )
  also have \ldots = (a \ mod \ (c \cdot k) + a \ div \ (c \cdot k) \cdot k \cdot c) \ mod \ c 
  by (simp only: mod-mult-self1)
  also have \ldots = (a \ div \ (c \cdot k) \cdot (c \cdot k) + a \ mod \ (c \cdot k)) \ mod \ c 
  by (simp only: ac-simps)
  also have \ldots = a \ mod \ c 
  by (simp only: div-mult-mod-eq)
finally show \ ?thesis .
qed

lemma div-mult-mult2 [simp]:
  \ c \neq 0 \implies (a \cdot c) \ div \ (b \cdot c) = a \ div \ b 
by (drule div-mult-mult1) (simp add: mult.commute)

lemma div-mult-mult1-if [simp]:
  \ (c \cdot a) \ div \ (c \cdot b) = (if \ c = 0 \ then \ 0 \ else \ a \ div \ b) 
by simp-all

lemma mod-mult-mult1:
  \ (c \cdot a) \ mod \ (c \cdot b) = c \cdot (a \ mod \ b) 
proof (cases \ c = 0)
  case True then show \ ?thesis by simp
next
  case False
  from \ div-mult-mod-eq 
  have \ (c \cdot a) \ div \ (c \cdot b) \cdot (c \cdot b) \cdot (c \cdot a) \ mod \ (c \cdot b) = c \cdot a . 
  with \ False have \ c \cdot ((a \ div \ b) \cdot b + a \ mod \ b) + (c \cdot a) \ mod \ (c \cdot b) 
  = c \cdot a + c \cdot (a \ mod \ b) \ by (simp add: algebra-simps) 
  with \ div-mult-mod-eq show \ ?thesis by simp
qed

lemma mod-mult-mult2:
  \ (a \cdot c) \ mod \ (b \cdot c) = (a \ mod \ b) \cdot c 
using \ mod-mult-mult1 [of \ c \ a \ b] \ by (simp add: mult.commute)

lemma mult-mod-left: \ (a \ mod \ b) \cdot c = (a \cdot c) \ mod \ (b \cdot c) 
by (fact mod-mult-mult2 [symmetric])
lemma mult-mod-right: \( c \ast (a \mod b) = (c \ast a) \mod (c \ast b) \)
by (fact mod-mult-mult1 [symmetric])

lemma dvd-mod: \( k \dvd m \Longrightarrow k \dvd n \Longrightarrow k \dvd (m \mod n) \)
unfolding dvd-def by (auto simp add: mod-mult-mult1)

lemma div-plus-div-distrib-dvd-left:
\( c \dvd a \Longrightarrow (a + b) \div c = a \div c + b \div c \)
by (cases \( c = 0 \)) (auto elim: dvdE)

lemma div-plus-div-distrib-dvd-right:
\( c \dvd b \Longrightarrow (a + b) \div c = a \div c + b \div c \)
using div-plus-div-distrib-dvd-left[of \( c \) \( b \) \( a \)]
by (simp add: ac-simps)

lemma sum-div-partition:
\( \langle \sum a \in A. f a \rangle \div b = (\sum a \in A \cap \{ a. b \dvd f a \}. f a \div b) + (\sum a \in A \cap \{ a. \neg b \dvd f a \}. f a) \rangle \div b \)
if \( \text{finite} \ A; \)
proof –
have \( \langle A = A \cap \{ a. b \dvd f a \} \cup A \cap \{ a. \neg b \dvd f a \} \rangle; \)
by auto
then have \( \langle \sum a \in A. f a \rangle = (\sum a \in A \cap \{ a. b \dvd f a \} \cup A \cap \{ a. \neg b \dvd f a \}. f a) \rangle; \)
by simp
also have \( \langle \ldots = (\sum a \in A \cap \{ a. b \dvd f a \}. f a) + (\sum a \in A \cap \{ a. \neg b \dvd f a \}. f a) \rangle; \)
using \( \text{finite} \ A; \)
by (auto intro: sum.union_inter_neutral)
finally have \( \ast: \langle \sum f A = \text{sum} f (A \cap \{ a. b \dvd f a \}) + \text{sum} f (A \cap \{ a. \neg b \dvd f a \}) \rangle \).

define \( B \) where \( \langle B = A \cap \{ a. b \dvd f a \} \rangle; \)
with \( \text{finite} \ A; \) have \( \text{finite} \ B; \)
and \( \langle a \in B \Longrightarrow b \dvd f a \rangle \) for \( a \)
by simp-all
then have \( \langle \sum a \in B. f a \rangle \div b = (\sum a \in B. \div b f a) \rangle \) and \( \langle b \dvd (\sum a \in B. f a) \rangle \)
by induction (simp-all add: div-plus-div-distrib-dvd-left)
then show \( \ast; \)
by (simp add: \( B \) div-plus-div-distrib-dvd-left)
qed

named-theorems mod-simps

Addition respects modular equivalence.

lemma mod-add-left-eq [mod-simps]:
\( (a \mod c + b) \mod c = (a + b) \mod c \)
proof –
have \( (a + b) \mod c = (a \div c \ast c + a \mod c + b) \mod c \)
by (simp only: div-mult-mod-eq)
also have \( \ldots = (a \mod c + b + a \div c \ast c) \mod c \)
by (simp only: ac-simps)
also have ... = (a mod c + b) mod c
  by (rule mod-mult-self1)
finally show \( ? \)thesis
  by (rule sym)
qed

lemma mod-add-right-eq [mod-simps]:
(a + b mod c) mod c = (a + b) mod c
  using mod-add-left-eq \([ b \, c \, a ]\) by (simp add: ac-simps)

lemma mod-add-eq:
(a mod c + b mod c) mod c = (a + b) mod c
  by (simp add: mod-add-left-eq mod-add-right-eq)

lemma mod-sum-eq [mod-simps]:
(\( \sum \) i\( \in A \). f i mod a) mod a = sum f A mod a
proof (induct A rule: infinite-finite-induct)
case (insert i A)
then have (\( \sum \) i\( \in insert \, i \, A \). f i mod a) mod a
  = (f i mod a + (\( \sum \) i\( \in A \). f i mod a)) mod a
  by simp
also have ... = (f i + (\( \sum \) i\( \in A \). f i mod a) mod a)
    by (simp add: mod-simps)
also have ... = (f i + (\( \sum \) i\( \in A \). f i) mod a) mod a
  by (simp add: insert.hyps)
finally show \( ? \)case
  by (simp add: insert.hyps mod-simps)
qed simp-all

lemma mod-add-cong:
  assumes a mod c = a' mod c
  assumes b mod c = b' mod c
  shows (a + b) mod c = (a' + b') mod c
proof
  have (a mod c + b mod c) mod c = (a' mod c + b' mod c) mod c
    unfolding assms ...
  then show ?thesis
    by (simp add: mod-add-eq)
qed

Multiplication respects modular equivalence.

lemma mod-mult-left-eq [mod-simps]:
((a mod c) * b) mod c = (a * b) mod c
proof
  have (a * b) mod c = ((a div c * c + a mod c) * b) mod c
    by (simp only: div-mult-mod-eq)
  also have ... = (a mod c * b + a div c * b * c) mod c
    by (simp only: algebra-simps)
also have \ldots = (a \mod c \ast b) \mod c  
  by (rule mod-mult-self1)  
finally show \thesis  
  by (rule sym)  
qed

lemma mod-mult-right-eq [mod-simps]:  
(a \ast (b \mod c)) \mod c = (a \ast b) \mod c  
using mod-mult-left-eq [of b c a] by (simp add: ac-simps)

lemma mod-mult-eq:  
((a \mod c) \ast (b \mod c)) \mod c = (a \ast b) \mod c  
by (simp add: mod-mult-left-eq mod-mult-right-eq)

lemma mod-prod-eq [mod-simps]:  
(\prod_{i \in A}. f i \mod a) \mod a = \prod_{i \in A}. f i \mod a  
proof (induct A rule: infinite-finite-induct)  
  case (insert i A)  
  then have \ldots = (f i \ast ((\prod_{i \in A}. f i \mod a) \mod a)) \mod a  
    by simp
also have \ldots = (f i \ast ((\prod_{i \in A}. f i \mod a) \mod a)) \mod a  
    by (simp add: mod-simps)
also have \ldots = (f i \ast ((\prod_{i \in A}. f i \mod a) \mod a)) \mod a  
    by (simp add: insert.hyps)
finally show \thesis  
    by (simp add: insert.hyps mod-simps)
qed simp-all

lemma mod-mult-cong:  
  assumes a \mod c = a' \mod c  
  assumes b \mod c = b' \mod c  
  shows (a \ast b) \mod c = (a' \ast b') \mod c  
proof
  have (a \mod c) \ast (b \mod c) \mod c = (a' \mod c) \ast (b' \mod c) \mod c  
    unfolding assms \ldots
  then show \thesis  
    by (simp add: mod-mult-eq)
qed

Exponentiation respects modular equivalence.

lemma power-mod [mod-simps]:  
(a \mod b) ^ n \mod b = (a ^ n) \mod b  
proof (induct n)  
  case 0  
  then show \thesis by simp
next  
  case (Suc n)  
  have (a \mod b) ^ Suc n \mod b = (a \mod b) ^ ((a \mod b) ^ n \mod b) \mod b
by (simp add: mod-mult-right-eq)
with Suc show ?case
  by (simp add: mod-mult-left-eq mod-mult-right-eq)
qed

lemma power-diff-power-eq:
〈a ^ m div a ^ n = (if n ≤ m then a ^ (m - n) else 1 div a ^ (n - m))〉
  if 〈a ≠ 0〉
proof (cases 〈n ≤ m〉)
  case True
  with that power-diff [symmetric, of a n m] show ?thesis by simp
next
  case False
  then obtain q where 〈n = m + Suc q〉
    by (auto simp add: not-le dest: less-imp-Suc-add)
  then have 〈a ^ m div a ^ n = (a ^ m * 1) div (a ^ m * a ^ Suc q)〉
    by (simp add: power-add ac-simps)
  moreover from that have 〈a ^ m ≠ 0〉
    by simp
  ultimately have 〈a ^ m div a ^ n = 1 div a ^ Suc q〉
    by (subst (asm) div-mult-mult1) simp
  with False n show ?thesis
    by simp
qed

class euclidean-ring-cancel = euclidean-ring + euclidean-semiring-cancel
begin
subclass idom-divide ..

lemma div-minus-minus [simp]: (− a) div (− b) = a div b
  using div-mult-mult1 [of − 1 a b] by simp

lemma mod-minus-minus [simp]: (− a) mod (− b) = − (a mod b)
  using mod-mult-mult1 [of − 1 a b] by simp

lemma div-minus-right: a div (− b) = (− a) div b
  using div-minus-minus [of − a b] by simp

lemma mod-minus-right: a mod (− b) = − ((− a) mod b)
  using mod-minus-minus [of − a b] by simp

lemma div-minus1-right [simp]: a div (− 1) = − a
  using div-minus-right [of a 1] by simp

lemma mod-minus1-right [simp]: a mod (− 1) = 0
using \texttt{mod-minus-right} [of a 1] by simp

Negation respects modular equivalence.

\textbf{lemma} \texttt{mod-minus-eq} [\texttt{mod-simps}]:
\[(\neg (a \mod b)) \mod b = (\neg a) \mod b\]
\textbf{proof} –
\begin{itemize}
\item have \((\neg a) \mod b = (\neg (a \div b \ast b + a \mod b)) \mod b\)
  \hspace{1em} by (simp only: \texttt{div-mult-mod-eq})
\item also have \ldots = \((\neg (a \mod b)) + \neg (a \div b) \ast b) \mod b\)
  \hspace{1em} by (simp add: \texttt{ac-simps})
\item also have \ldots = \((\neg (a \mod b)) \mod b\)
  \hspace{1em} by (rule \texttt{mod-mult-self1})
\end{itemize}
finally show \(?thesis\)
\hspace{1em} by (rule \texttt{sym})
\textbf{qed}

\textbf{lemma} \texttt{mod-minus-cong}:  
\begin{itemize}
\item assumes \(a \mod b = a' \mod b\)
\item shows \((\neg a) \mod b = (\neg a') \mod b\)
\end{itemize}
\textbf{proof} –
\begin{itemize}
\item have \((\neg (a \mod b)) \mod b = (\neg (a') \mod b)) \mod b\)
  \hspace{1em} unfolding \textit{assms} ..
\item then show \(?thesis\)
  \hspace{1em} by (simp add: \texttt{mod-minus-eq})
\end{itemize}
\textbf{qed}

Subtraction respects modular equivalence.

\textbf{lemma} \texttt{mod-diff-left-eq} [\texttt{mod-simps}]:
\[(a \mod c - b) \mod c = (a - b) \mod c\]
\textbf{using} \texttt{mod-add-cong} [of \(a \mod c c b\)]
\textbf{by} simp

\textbf{lemma} \texttt{mod-diff-right-eq} [\texttt{mod-simps}]:
\[(a - b \mod c) \mod c = (a - b) \mod c\]
\textbf{using} \texttt{mod-add-cong} [of \(a \mod c a - b - (b \mod c)\)] \texttt{mod-minus-cong} [of \(b \mod c c b\)]
\textbf{by} simp

\textbf{lemma} \texttt{mod-diff-eq}:  
\begin{itemize}
\item \((a \mod c - b \mod c) \mod c = (a - b) \mod c\)
\item \textbf{using} \texttt{mod-add-cong} [of \(a c a \mod c c b\)] \texttt{mod-minus-cong} [of \(b \mod c c b\)]
\item \textbf{by} simp
\end{itemize}

\textbf{lemma} \texttt{mod-diff-cong}:  
\begin{itemize}
\item \textbf{assumes} \(a \mod c = a' \mod c\)
\item \textbf{assumes} \(b \mod c = b' \mod c\)
\item \textbf{shows} \((a - b) \mod c = (a' - b') \mod c\)
\item \textbf{using} \textit{assms} \texttt{mod-add-cong} [of \(a c a' - b - (b \mod c)\)] \texttt{mod-minus-cong} [of \(b c b'\)]
\item \textbf{by} simp
lemma minus-mod-self2 [simp]:
(a − b) mod b = a mod b
using mod-diff-right-eq [of a b b]
by (simp add: mod-diff-right-eq)

lemma minus-mod-self1 [simp]:
(b - a) mod b = - a mod b
using mod-add-self2 [of - a b]
by simp

lemma mod-eq-dvd-iff:
a mod c = b mod c ←→ c dvd a − b (is ?P ←→ ?Q)
proof
assume ?P
then have (a mod c − b mod c) mod c = 0
  by simp
then show ?Q
  by (simp add: dvd-eq-mod-eq-0 mod-simps)
next
assume ?Q
then obtain d where d: a − b = c * d ..
then have a = c * d + b
  by (simp add: algebra-simps)
then show ?P by simp
qed

lemma mod-eqE:
assumes a mod c = b mod c
obtains d where b = a + c * d
proof
from assms have c dvd a − b
  by (simp add: mod-eq-dvd-iff)
then obtain d where a − b = c * d ..
then have b = a + c * d − d
  by (simp add: algebra-simps)
with that show thesis .
qed

lemma invertible-coprime:
coprime a c if a * b mod c = 1
by (rule coprimeI) (use that dvd-mod-iff [of - c a * b] in auto)
end

54.3 Uniquely determined division

class unique-euclidean-semiring = euclidean-semiring +
assumes euclidean-size-mult: euclidean-size (a * b) = euclidean-size a * euclidean-size b
THEORY “Euclidean-Division”

fixes division-segment :: 'a ⇒ 'a
assumes is-unit-division-segment [simp]: is-unit (division-segment)
and division-segment-mult:
  a ≠ 0 → b ≠ 0 → division-segment (a * b) = division-segment a * division-segment b
and division-segment-mod:
  b ≠ 0 → ¬ b dvd a → division-segment (a mod b) = division-segment b
assumes div-bounded:
  b ≠ 0 → division-segment r = division-segment b
  ⇒ euclidean-size r < euclidean-size b
  ⇒ (q * b + r) div b = q

begin

lemma division-segment-not-0 [simp]:
  division-segment a ≠ 0
using is-unit-division-segment [of a] is-unitE [of division-segment a] by blast

lemma divmod-cases [case-names divides remainder by0]:
  obtains
  (divides) q where b ≠ 0
  and a div b = q
  and a mod b = 0
  and a = q * b
  | (remainder) q r where b ≠ 0
  and division-segment r = division-segment b
  and euclidean-size r < euclidean-size b
  and r ≠ 0
  and a div b = q
  and a mod b = r
  and a = q * b + r
  | (by0) b = 0
proof (cases b = 0)
case True
then show thesis
by (rule by0)
next
case False
show thesis
proof (cases b dvd a)
case True
then obtain q where a = b * q ..
with (b ≠ 0) divides
show thesis
by (simp add: ac-simps)
next
case False
then have a mod b ≠ 0
by (simp add: mod-eq-0-iff-dvd)
moreover from (b ≠ 0) (¬ b dvd a) have division-segment (a mod b) =
THEORY “Euclidean-Division”

division-segment b
by (rule division-segment-mod)
moreover have euclidean-size (a mod b) < euclidean-size b
using (b ≠ 0) by (rule mod-size-less)
moreover have a = a div b * b + a mod b
by (simp add: div-mult-mod-eq)
ultimately show thesis
using (b ≠ 0) by (blast intro: remainder)
qed

lemma div-eqI:
a div b = q if b ≠ 0 division-segment r = division-segment b
euclidean-size r < euclidean-size b q * b + r = a
proof –
from that have (q * b + r) div b = q
by (auto intro: div-bounded)
with that show ?thesis
by simp
qed

lemma mod-eqI:
a mod b = r if b ≠ 0 division-segment r = division-segment b
euclidean-size r < euclidean-size b q * b + r = a
proof –
from that have a div b = q
by (rule div-eqI)
moreover have a div b * b + a mod b = a
by (fact div-mult-mod-eq)
ultimately have a div b * b + a mod b = a div b * b + r
using (q * b + r = a) by simp
then show ?thesis
by simp
qed

subclass euclidean-semiring-cancel
proof
show (a + c * b) div b = c + a div b if b ≠ 0 for a b c
proof (cases a b rule: divmod-cases)
case by0
with (b ≠ 0) show ?thesis
by simp
next
case (divides q)
then show ?thesis
by (simp add: ac-simps)
next
case (remainder q r)
then show ?thesis
by (auto intro: div-eqI simp add: algebra-simps)
qed
next

show $(c * a) \ div (c * b) = a \ div b$ if \( c \neq 0 \) for \( a \ b c \)
proof (cases \( a \ b \) rule: divmod-cases)
  case by0
  then show \( ?thesis \)
  by simp
next
  case (divides \( q \))
  with \( \langle c \neq 0 \rangle \) show \( ?thesis \)
  by (simp add: mult.left-commute \[of \ c\])
next
  case (remainder \( q \ r \))
  from remainder \( \langle 1 - 3 \rangle \) show \( ?thesis \)
  proof (rule div-eqI)
    have \( a * b = a * (b \ div c) + a * (b \ mod c) \)
    by (simp add: div-mult-mod-eq)
    also have \( \ldots = (a * (b \ div c) + q) * c \)
    using divides by (simp add: algebra-simps)
    finally have \( (a * b) \ div c = \ldots \ div c \)
    by simp
    with divides show \( ?thesis \)
    by simp
next
  case (remainder \( q \ r \))
  from remainder \( \langle 1 - 3 \rangle \) show \( ?thesis \)
  proof (rule div-eqI)
    have \( a * b = a * (b \ div c) + a * (b \ mod c) \)
    by (simp add: div-mult-mod-eq)
    also have \( \ldots = a * c * (b \ div c) + q * c + r \)
    using remainder by (simp add: algebra-simps)
    finally show \( (a * (b \ div c) + a * (b \ mod c) \ div c) * c + r = a * b \)
    using remainder \( \langle 5 - 7 \rangle \) by (simp add: algebra-simps)
  qed

lemma \( \text{div-mult1-eq} \):
\( (a * b) \ div c = a * (b \ div c) + a * (b \ mod c) \)
proof (cases \( a * (b \ mod c) \) c rule: divmod-cases)
  case (divides \( q \))
  have \( a * b = a * (b \ div c) + b \ mod c \)
  by (simp add: div-mult-mod-eq)
  also have \( \ldots = (a * (b \ div c) + q) * c \)
  using divides by (simp add: algebra-simps)
  finally have \( (a * b) \ div c = \ldots \ div c \)
  by simp
  with divides show \( ?thesis \)
  by simp
next
  case (remainder \( q \ r \))
  from remainder \( \langle 1 - 3 \rangle \) show \( ?thesis \)
  proof (rule div-eqI)
    have \( a * b = a * (b \ div c) + a * (b \ mod c) \)
    by (simp add: div-mult-mod-eq)
    also have \( \ldots = a * c * (b \ div c) + q * c + r \)
    using remainder by (simp add: algebra-simps)
    finally show \( (a * (b \ div c) + a * (b \ mod c) \ div c) * c + r = a * b \)
    using remainder \( \langle 5 - 7 \rangle \) by (simp add: algebra-simps)
  qed
next
case by0
then show ?thesis
by simp
qed

lemma div-add1-eq:
(a + b) div c = a div c + b div c + (a mod c + b mod c) div c
proof (cases a mod c + b mod c rule: divmod_cases)
case (divides q)
have a + b = (a div c * c + a mod c) + (b div c * c + b mod c)
using mod-mult-div-eq [of a c] mod-mult-div-eq [of b c] by (simp add: ac-simps)
also have ... = (a div c + b div c) * c + (a mod c + b mod c)
by (simp add: algebra-simps)
also have ... = (a div c + b div c + q) * c
using divides by (simp add: algebra-simps)
finally have (a + b) div c = (a div c + b div c + q) * c div c
by simp
with divides show ?thesis
by simp
next
case (remainder q r)
from remainder(1-3) show ?thesis
proof (rule div-eqI)
have (a div c + b div c + q) * c + r + (a mod c + b mod c) =
  (a div c * c + a mod c) + (b div c * c + b mod c) + q * c + r
by (simp add: algebra-simps)
also have ... = a + b + (a mod c + b mod c)
by (simp add: div-mult-mod-eq remainder) (simp add: ac-simps)
finally show (a div c + b div c + (a mod c + b mod c) div c) * c + r = a + b
using remainder by simp
qed
next
case by0
then show ?thesis
by simp
qed

lemma div-eq-0-iff:
a div b = 0 ↔ euclidean-size a < euclidean-size b ∨ b = 0 (is - ↔ ?P)
if division-segment a = division-segment b
proof
assume ?P
with that show a div b = 0
by (cases b = 0) (auto intro: div-eqI)
next
assume a div b = 0
then have a mod b = a
using div-mult-mod-eq [of a b] by simp
with mod-size-less [of b a] show ?P
  by auto
qed
end

class unique-euclidean-ring = euclidean-ring + unique-euclidean-semiring
begin
subclass euclidean-ring-cancel ..
end

54.4 Euclidean division on nat

instantiation nat :: normalization-semidom
begin

definition normalize-nat :: nat ⇒ nat
  where [simp]: normalize = (id :: nat ⇒ nat)

definition unit-factor-nat :: nat ⇒ nat
  where unit-factor n = (if n = 0 then 0 else 1 :: nat)

lemma unit-factor-simps [simp]:
  unit-factor 0 = (0 :: nat)
  unit-factor (Suc n) = 1
  by (simp-all add: unit-factor-nat-def)

definition divide-nat :: nat ⇒ nat ⇒ nat
  where m div n = (if n = 0 then 0 else Max {k::nat. k * n ≤ m})

instance
  by standard (auto simp add: divide-nat-def ac-simps unit-factor-nat-def intro: Max-eqI)
end

lemma coprime-Suc-0-left [simp]:
  coprime (Suc 0) n
  using coprime-1-left [of n] by simp

lemma coprime-Suc-0-right [simp]:
  coprime n (Suc 0)
  using coprime-1-right [of n] by simp

lemma coprime-common-divisor-nat: coprime a b ⇒ x dvd a ⇒ x dvd b ⇒ x = 1
  for a b :: nat
by (drule coprime-common-divisor [of - x]) simp-all

instantiation nat :: unique-euclidean-semiring
begin

definition euclidean-size-nat :: nat ⇒ nat
where [simp]: euclidean-size-nat = id

definition division-segment-nat :: nat ⇒ nat
where [simp]: division-segment-nat n = 1

definition modulo-nat :: nat ⇒ nat ⇒ nat
where m mod n = m − (m div n * (n::nat))

instance proof
  fix m n :: nat
  have ex: ∃ k. k * n ≤ l for l :: nat
    by (rule exI [of - 0]) simp
  have fin: finite { k. k * n ≤ l } if n > 0 for l
    proof –
      from that have { k. k * n ≤ l } ⊆ { k. k ≤ l }
      by (cases n) auto
      then show ?thesis
      by (rule finite-subset) simp
    qed
  have mult-div-unfold: n * (m div n) = Max { l. l ≤ m ∧ n dvd l }
    proof (cases n = 0)
      case True
      moreover have { l. l = 0 ∧ l ≤ m } = { 0::nat }
        by auto
      ultimately show ?thesis
      by simp
    next
      case False
      with ex [of m] fin have n * Max { k. k * n ≤ m } = Max (times n ' { k. k * n ≤ m })
        by (auto simp add: nat-mul-max-right intro: hom-Max-commute)
      also have times n ' { k. k * n ≤ m } = { l. l ≤ m ∧ n dvd l }
        by (auto simp add: ac-simps elim!: dvdE)
      finally show ?thesis
      using False by (simp add: divide-nat-def ac-simps)
    qed
  have less-eq: m div n * n ≤ m
    by (auto simp add: mult-div-unfold ac-simps intro: Max.boundedI)
  then show m div n * n + m mod n = m
    by (simp add: modulo-nat-def)
  assume n ≠ 0
  show euclidean-size (m mod n) < euclidean-size n
    proof –
have \( m < \text{Suc} (m \div n) \times n \)

proof (rule ccontr)
  assume \( \neg m < \text{Suc} (m \div n) \times n \)
  then have \( \text{Suc} (m \div n) \times n \leq m \)
    by (simp add: not-less)
  moreover from \( \langle n \neq 0 \rangle \) have \( \text{Max} \{ k. k \times n \leq m \} < \text{Suc} (m \div n) \)
    by (simp add: divide-nat-def)
  with \( \langle n \neq 0 \rangle \) ex fin have \( \bigwedge k. k \times n \leq m \implies k < \text{Suc} (m \div n) \)
    by auto
  ultimately have \( \text{Suc} (m \div n) < \text{Suc} (m \div n) \)
    by blast
  then show False
    by simp
  qed
with \( \langle n \neq 0 \rangle \) show \( \neg \text{thesis} \)
  by (simp add: modulo-nat-def)
qed

show \( \text{euclidean-size} m \leq \text{euclidean-size} (m \times n) \)
  using \( \langle n \neq 0 \rangle \) by (cases n) simp-all
fix \( q, r : \text{nat} \)
show \( (q \times n + r) \div n = q \) if \( \text{euclidean-size} r < \text{euclidean-size} n \)
proof -
  from that have \( r < n \)
    by simp
  have \( k \leq q \) if \( k \times n \leq q \times n + r \) for \( k \)
  proof (rule ccontr)
    assume \( \neg k \leq q \)
    then have \( q < k \)
      by simp
    then obtain \( l \) where \( k = \text{Suc} (q + l) \)
      by (auto simp add: less-iff-Suc-add)
    with \( \langle r < n \rangle \) that show False
      by (simp add: algebra-simps)
  qed
with \( \langle n \neq 0 \rangle \) ex fin show \( \neg \text{thesis} \)
  by (auto simp add: divide-nat-def Max-eq-iff)
qed simp-all

end

Tool support

ML
structure Cancel-Div-Mod-Nat = Cancel-Div-Mod
(
  val div-name = const-name (divide);
  val mod-name = const-name (modulo);
  val mk-binop = HOLogic.mk-binop;
  val dest-plus = HOLogic.dest_bin const-name (Groups.plus) HOLogic.natT;
val mk-sum = Arith-Data.mk-sum;

fun dest-sum tm =
  if HOLogic.is-zero tm then []
  else
    (case try HOLogic.dest-Suc tm of
     SOME t => HOLogic.Suc-zero :: dest-sum t
    | NONE =>>
      (case try dest-plus tm of
       SOME (t, u) => dest-sum t @ dest-sum u
      | NONE =>> [tm]));

val div-mod-eqs = map mk-meta-eq @
{thms cancel-div-mod-rules};

val prove-eq-sums = Arith-Data.prove-conv2 all-tac
  (Arith-Data.simp-all-tac @
{thms add-0-left add-0-right ac-simps})

lemma div-nat-eqI:
  m div n = q if n * q ≤ m and m < n * Suc q for m n q :: nat
by (rule div-eqI [of - m - n * q]) (use that in simp-all add: algebra-simps:)

lemma mod-nat-eqI:
  m mod n = r if r < n and r ≤ m and n dvd m - r for m n r :: nat
by (rule mod-eqI [of - - (m - r) div n]) (use that in simp-all add: algebra-simps:)

lemma div-mult-self-is-m [simp]:
  m * n div n = m if n > 0 for m n :: nat
using that by simp

lemma div-mult-self1-is-m [simp]:
  n * m div n = m if n > 0 for m n :: nat
using that by simp

lemma mod-less-divisor [simp]:
  m mod n < n if n > 0 for m n :: nat
using mod-size-less [of n m] that by simp

lemma mod-le-divisor [simp]:
  m mod n ≤ n if n > 0 for m n :: nat
using that by (auto simp add: le-less)

lemma div-times-less-eq-dividend [simp]:
  m div n * n ≤ m for m n :: nat
by (simp add: minus-mod-eq-div-mult [symmetric])
THEORY “Euclidean-Division”

lemma times-div-less-eq-dividend [simp]:
\[ n \times (m \div n) \leq m \text{ for } m, n :: \text{nat} \]
using div-times-less-eq-dividend [of m n]
by (simp add: ac-simps)

lemma dividend-less-div-times:
\[ m < n + (m \div n) \times n \text{ if } 0 < n \text{ for } m, n :: \text{nat} \]
proof -
from that have \( m \mod n < n \)
by simp
then show \(?thesis\)
by (simp add: minus-mod-eq-div-mult [symmetric])
qed

lemma dividend-less-times-div:
\[ m < n + n \times (m \div n) \text{ if } 0 < n \text{ for } m, n :: \text{nat} \]
using dividend-less-div-times [of n m] that
by (simp add: ac-simps)

lemma mod-Suc-le-divisor [simp]:
\[ m \mod Suc n \leq n \]
using mod-less-divisor [of Suc n m] by arith

lemma mod-less-eq-dividend [simp]:
\[ m \mod n \leq m \text{ for } m, n :: \text{nat} \]
proof (rule add-leD2)
from \( \langle n \leq m \rangle \) obtain \( q \) where \( m = n + q \)
by (auto simp add: le_iff_add)
then show \(?thesis\)
by (simp add: div-add-self1)
qed

lemma le-mod-geq:
\[ m \div n = Suc ((m - n) \div n) \text{ if } 0 < n \text{ and } n \leq m \text{ for } m, n :: \text{nat} \]
proof -
from \( \langle n \leq m \rangle \) obtain \( q \) where \( m = n + q \)
by (auto simp add: le_iff_add)
with \( \langle 0 < n \rangle \) show \(?thesis\)
by (simp add: div-add-self1)
qed

lemma le-mod-geq:
\[ m \mod n = (m - n) \mod n \text{ if } n \leq m \text{ for } m, n :: \text{nat} \]
proof -
from \( \langle n \leq m \rangle \) obtain \( q \) where \( m = n + q \)
by (auto simp add: le_iff_add)
then show \( \text{thesis} \)
  by simp
qed

lemma div-if:
  \( m \div n = (\text{if } m < n \lor n = 0 \text{ then } \text{Suc } ((m - n) \div n)) \)
  by (simp add: le_div_geq)

lemma mod-if:
  \( m \mod n = (\text{if } m < n \text{ then } m \text{ else } (m - n) \mod n) \) for \( m, n :: \text{nat} \)
  by (simp add: le_mod_geq)

lemma div-eq-0-iff:
  \( m \div n = 0 \longleftrightarrow m < n \lor n = 0 \) for \( m, n :: \text{nat} \)
  by (simp add: div_eq_0_iff)

lemma div-greater-zero-iff:
  \( m \div n > 0 \longleftrightarrow n \leq m \land n > 0 \) for \( m, n :: \text{nat} \)
  using div_eq_0_iff [of \( m, n \)]
  by auto

lemma mod-greater-zero-iff-not-dvd:
  \( m \mod n > 0 \longleftrightarrow \neg n \text{ dvd } m \) for \( m, n :: \text{nat} \)
  by (simp add: dvd_eq_mod_eq_0)

lemma div-Suc-0 [simp]:
  \( m \div \text{Suc } 0 = m \)
  using div-by-1 [of \( m \)]
  by simp

lemma mod-Suc-0 [simp]:
  \( m \mod \text{Suc } 0 = 0 \)
  using mod-by-1 [of \( m \)]
  by simp

lemma div2-Suc-Suc [simp]:
  \( \text{Suc } (\text{Suc } m) \div 2 = \text{Suc } (m \div 2) \)
  by (simp add: numeral-2_eq_2 le_div_geq)

lemma Suc-n-div-2-gt-zero [simp]:
  \( 0 < \text{Suc } n \div 2 \text{ if } n > 0 \) for \( n :: \text{nat} \)
  using that by (cases \( n \)) simp_all

lemma div-2-gt-zero [simp]:
  \( 0 < n \div 2 \text{ if } \text{Suc } 0 < n \) for \( n :: \text{nat} \)
  using that Suc-n_div-2-gt-zero [of \( n - 1 \)]
  by simp

lemma mod2-Suc-Suc [simp]:
  \( \text{Suc } (\text{Suc } m) \mod 2 = m \mod 2 \)
  by (simp add: numeral-2_eq_2 le_mod_geq)
lemma add-self-div-2 [simp]:
\[(m + m) \div 2 = m \text{ for } m :: \text{nat}\]
by (simp add: mult-2 [symmetric])

lemma add-self-mod-2 [simp]:
\[(m + m) \mod 2 = 0 \text{ for } m :: \text{nat}\]
by (simp add: mult-2 [symmetric])

lemma mod2-gr-0 [simp]:
\[0 < m \mod 2 \iff m \mod 2 = 1 \text{ for } m :: \text{nat}\]
proof –
  have \(m \mod 2 < 2\)
  by (rule mod-less-divisor) simp
  then have \(m \mod 2 = 0 \lor m \mod 2 = 1\)
  by arith
  then show \(?thesis\)
  by auto
qed

lemma mod-Suc-eq [mod-simps]:
\[\text{Suc } (m \mod n) \mod n = \text{Suc } m \mod n\]
proof –
  have \((m \mod n + 1) \mod n = (m + 1) \mod n\)
  by (simp only: mod-simps)
  then show \(?thesis\)
  by simp
qed

lemma mod-Suc-Suc-eq [mod-simps]:
\[\text{Suc } (\text{Suc } (m \mod n)) \mod n = \text{Suc } (\text{Suc } m) \mod n\]
proof –
  have \((m \mod n + 2) \mod n = (m + 2) \mod n\)
  by (simp only: mod-simps)
  then show \(?thesis\)
  by simp
qed

lemma Suc-mod-mult-self1 [simp]:
\[\text{Suc } (m + k \ast n) \mod n = \text{Suc } m \mod n\]
and Suc-mod-mult-self2 [simp]:
\[\text{Suc } (m + n \ast k) \mod n = \text{Suc } m \mod n\]
and Suc-mod-mult-self3 [simp]:
\[\text{Suc } (k \ast n + m) \mod n = \text{Suc } m \mod n\]
and Suc-mod-mult-self4 [simp]:
\[\text{Suc } (n \ast k + m) \mod n = \text{Suc } m \mod n\]
by (subst mod-Suc-eq [symmetric], simp add: mod-simps)

lemma Suc-0-mod-eq [simp]:
\[\text{Suc } 0 \mod n = \text{of-bool } (n \neq \text{Suc } 0)\]
by (cases n) simp-all

context
fixes \( m, n, q : \mathbb{N} \)

begin

private lemma eucl-rel-mult2:
\[ m \mod n + n \cdot (m \div n \mod q) < n \cdot q \]
if \( n > 0 \) and \( q > 0 \)

proof -
  from \( \langle n > 0 \rangle \) have \( m \mod n < n \)
  by (rule mod-less-divisor)
  from \( \langle q > 0 \rangle \) have \( m \div n \mod q < q \)
  by (rule mod-less-divisor)
then obtain \( s \) where \( q = \text{Suc} (m \div n \mod q + s) \)
  by (blast dest: less-imp-Suc-add)
moreover have \( m \mod n + n \cdot (m \div n \mod q) < n \cdot \text{Suc} (m \div n \mod q + s) \)
  using \( \langle m \mod n < n \rangle \) by (simp add: add-mult-distrib2)
ultimately show ?thesis
  by simp
qed

lemma div-mult2-eq:
\[ m \div (n \cdot q) = (m \div n) \div q \]
proof (cases \( n = 0 \vee q = 0 \))
  case True
  then show ?thesis
    by auto
next
  case False
  with eucl-rel-mult2 show ?thesis
    by (auto intro: div-eqI \[ of \ n \cdot (m \div n \mod q) \]
            simp add: algebra-simps add-mult-distrib2 \[ symmetric \])
qed

lemma mod-mult2-eq:
\[ m \mod (n \cdot q) = n \cdot (m \div n \mod q) + m \mod n \]
proof (cases \( n = 0 \vee q = 0 \))
  case True
  then show ?thesis
    by auto
next
  case False
  with eucl-rel-mult2 show ?thesis
    by (auto intro: mod-eqI \[ of \ - (m \div n) \ div q \]
            simp add: algebra-simps add-mult-distrib2 \[ symmetric \])
qed

end

lemma div-le-mono:
THEORY "Euclidean-Division"

\[ m \div k \leq n \div k \text{ if } m \leq n \text{ for } m, n, k :: \text{nat} \]

proof

from that obtain \( q \) where \( n = m + q \)
by (auto simp add: le_iff_add)
then show \(?thesis\)
by (simp add: div-add1-eq [of \( m \) \( q \) \( k \)])
qed

Antimonotonicity of \((\div)\) in second argument

lemma \text{div-le-mono2}:
\[ k \div n \leq k \div m \text{ if } 0 < m \text{ and } m \leq n \text{ for } m, n, k :: \text{nat} \]
using that proof (induct \( k \) arbitrary; m rule: less-induct)
case (less \( k \))
show \(?case\)
proof (cases \( n \leq k \))
case False
then show \(?thesis\)
by simp
next
case True
have \((k - n) \div n \leq (k - m) \div n\)
using less.prems
by (blast intro: div-le-mono diff-le-mono2)
also have \(\ldots \leq (k - m) \div m\)
using \((n \leq k) \div \text{less.prems less.hyps [of } k - m m\)\)
by simp
finally show \(?thesis\)
using \((n \leq k) \div \text{less.prems}\)
by (simp add: le-div-geq)
qed

lemma \text{div-le-dividend} [simp]:
\[ m \div n \leq m \text{ for } m, n :: \text{nat} \]
using \text{div-le-mono2} [of \( 1 \) \( m \)] by (cases \( n = 0 \)) simp-all

lemma \text{div-less-dividend} [simp]:
\[ m \div n < m \text{ if } 1 < n \text{ and } 0 < m \text{ for } m, n :: \text{nat} \]
using that proof (induct \( m \) rule: less-induct)
case (less \( m \))
show \(?case\)
proof (cases \( n < m \))
case False
with less show \(?thesis\)
by (cases \( n = m \)) simp-all
next
case True
then show \(?thesis\)
using less.hyps [of \( m - n \)] \text{less.prems}
THEORY "Euclidean-Division"

by (simp add: le-div-geq)
qed
qed

lemma div-eq-dividend-iff:
  \( m \div n = m \iff n = 1 \) if \( m > 0 \) for \( m n : \text{nat} \)
proof
  assume \( n = 1 \)
  then show \( m \div n = m \)
  by simp
next
  assume \( P; m \div n = m \)
  show \( n = 1 \)
proof (rule ccontr)
  have \( n \neq 0 \)
  by (rule ccontr) (use that \( P \) in auto)
  moreover assume \( n \neq 1 \)
  ultimately have \( n > 1 \)
  by simp
  with that have \( m \div n < m \)
  by simp
  with \( P \) show False
  by simp
qed
qed

lemma less-mult-imp-div-less:
  \( m \div n < i \) if \( m < i \times n \) for \( m n i : \text{nat} \)
proof
  from that have \( i \times n > 0 \)
  by (cases \( i \times n = 0 \)) simp-all
  then have \( i > 0 \) and \( n > 0 \)
  by simp-all
  have \( m \div n \times n \leq m \)
  by simp
  then have \( m \div n \times n < i \times n \)
  using that by (rule le-less-trans)
  with \( \langle n > 0 \rangle \) show ?thesis
  by simp
qed

A fact for the mutilated chess board

lemma mod-Suc:
  \( \text{Suc } m \mod n = (\text{if } \text{Suc } (m \mod n) = n \text{ then } 0 \text{ else } \text{Suc } (m \mod n)) \) (is - = ?rhs)
proof (cases \( n = 0 \))
  case True
  then show ?thesis
  by simp
next
case False

have Suc m mod n = Suc (m mod n) mod n
  by (simp add: mod-simps)
also have \ldots \ = ?rhs
  using False by (auto intro!: mod-nat-eqI intro: neq-le-trans simp add: Suc-le-eq)
finally show ?thesis .
qed

lemma Suc-times-mod-eq:
  Suc (m * n) mod m = 1 if Suc 0 < m
using that by (simp add: mod-Suc)

lemma Suc-times-numeral-mod-eq [simp]:
  Suc (numeral k * n) mod numeral k = 1 if numeral k \neq (1::nat)
by (rule Suc-times-mod-eq) (use that in simp)

lemma Suc-div-le-mono [simp]:
  m div n \leq Suc m div n
by (simp add: div-le-mono)

These lemmas collapse some needless occurrences of Suc: at least three Sucs,
since two and fewer are rewritten back to Suc again! We already have some
rules to simplify operands smaller than 3.

lemma div-Suc-eq-div-add3 [simp]:
  m div Suc (Suc (Suc n)) = m div (3 + n)
by (simp add: Suc3-eq-add-3)

lemma mod-Suc-eq-mod-add3 [simp]:
  m mod Suc (Suc (Suc n)) = m mod (3 + n)
by (simp add: Suc3-eq-add-3)

lemma Suc-div-eq-add3-div:
  Suc (Suc (Suc m)) div n = (3 + m) div n
by (simp add: Suc3-eq-add-3)

lemma Suc-mod-eq-add3-mod:
  Suc (Suc (Suc m)) mod n = (3 + m) mod n
by (simp add: Suc3-eq-add-3)

lemmas Suc-div-eq-add3-div-numeral [simp] =
  Suc-div-eq-add3-div [of - numeral v] for v

lemmas Suc-mod-eq-add3-mod-numeral [simp] =
  Suc-mod-eq-add3-mod [of - numeral v] for v

lemma (in field-char-0) of-nat-div:
  of-nat (m div n) = ((of-nat m - of-nat (m mod n)) / of-nat n)
proof –
  have of-nat (m div n) = ((of-nat (m div n * n + m mod n) - of-nat (m mod
THEORY "Euclidean-Division"

\( n \) / of-nat \( n \) :: 'a

unfolding of-nat-add by (cases \( n = 0 \)) simp-all
then show ?thesis
by simp
qed

An "induction" law for modulus arithmetic.

lemma mod-induct [consumes 3, case-names step]:
P \( m \) if P \( n \) and \( n < p \) and \( m < p \)
and step: \( \forall n. n < p \implies P n \implies P (\text{Suc } n \mod p) \)
using \( \langle \text{m < p} \rangle \) proof (induct \( m \))
case 0
show ?case
proof (rule ccontr)
assume \( \neg P \ 0 \)
from \( \langle n < p \rangle \) have \( \theta < p \)
by simp
from \( \langle n < p \rangle \) obtain \( m \) where \( \theta < m \) and \( p = n + m \)
by (blast dest: less-imp-add-positive)
with \( \langle P \ n \rangle \) have \( P \ (p - m) \)
by simp
moreover have \( \neg P \ (p - m) \)
using \( \langle \theta < m \rangle \) proof (induct \( m \))
case 0
then show ?case
by simp
next
case (Suc \( m \))
show ?case
proof
assume \( P: P \ (p - \text{Suc} \ m) \)
with \( \langle \neg P \ 0 \rangle \) have Suc \( m < p \)
by (auto intro: ccontr)
then have Suc \( (p - \text{Suc} \ m) = p - m \)
by arith
moreover from \( \langle \theta < p \rangle \) have \( p - \text{Suc} \ m < p \)
by arith
with \( P \) step have \( P \ ((\text{Suc} \ (p - \text{Suc} \ m)) \mod p) \)
by blast
ultimately show False
using \( \langle \neg P \ 0 \rangle \), Suc.hyps by (cases \( m = 0 \)) simp-all
qed
qed
ultimately show False
by blast
qed
next
case (Suc \( m \))
then have \( m < p \) and \( \text{mod: Suc} \ m \mod p = \text{Suc} \ m \)
by simp-all
from \(m < p\) have \(P m\)
by (rule Suc.hyps)
with \(m < p\) have \(P (\text{Suc } m \mod p)\)
by (rule step)
with \(\mod\) show \(?case\)
by simp
qed

lemma \text{split-div}:
\[
P (m \div n) \iff (n = 0 \rightarrow P 0) \land (n \neq 0 \rightarrow \\
(\forall i. j < n \rightarrow m = n \ast i + j \rightarrow P i))
\]
is \(?P = \(?Q\)) for \(m n :: \text{nat}\)
proof (cases \(n = 0\))
case True
then show \(?thesis\)
by simp
next
case False
then have \(n \ast q \leq m \land m < n \ast \text{Suc } q \rightarrow P q\)
by (auto intro: \text{div-nat-eqI \text{dividend-less-times-div}})
then show \(?thesis\)
by simp
qed

lemma \text{split-div}':
\[
P (m \div n) \iff n = 0 \land P 0 \lor (\exists q. (n \ast q \leq m \land m < n \ast \text{Suc } q) \land P q)
\]
proof (cases \(n = 0\))
case True
then show \(?thesis\)
by simp
next
case False
then have \(n \ast q \leq m \land m < n \ast \text{Suc } q \iff m \div n = q\) for \(q\)
by (auto intro: \text{div-nat-eqI dividend-less-times-div})
then show \(?thesis\)
by auto
qed

lemma \text{split-mod}:
\[
P (m \mod n) \iff (n = 0 \rightarrow P m) \land (n \neq 0 \rightarrow \\
\]
(∀ i j. j < n → m = n * i + j → P j))
(is ?P ↔ ?Q) for m n :: nat

proof (cases n = 0)
case True
then show ?thesis
  by simp
next
case False
show ?thesis
proof
  assume ?P
  with False show ?Q
    by auto
next
  assume ?Q
  with False have *: ∀ i j. j < n → m = n * i + j → P j
    by simp
  with False show ?P
  (auto intro: * [of - m div n])
qed
qed

54.5  Euclidean division on int

instantiation int :: normalization-semidom begin

definition normalize-int :: int ⇒ int
  where [simp]: normalize = (abs :: int ⇒ int)
definition unit-factor-int :: int ⇒ int
  where [simp]: unit-factor = (sgn :: int ⇒ int)
definition divide-int :: int ⇒ int ⇒ int
  where k div l = (if l = 0 then 0
    else if sgn k = sgn l
      then int (nat |k| div nat |l|)
    else − int (nat |k| div nat |l|) + of-bool (∼ l dvd k))

lemma divide-int-unfold:
  (sgn k * int m) div (sgn l * int n) =
    (if sgn l = 0 ∨ sgn k = 0 ∨ n = 0 then 0
      else if sgn k = sgn l
        then int (m div n)
      else − int (m div n + of-bool (∼ n dvd m)))
  (auto simp add: divide-int-def sgn-0-0 sgn-1-pos sgn-mult abs-mult
    nat-mult-distrib)

instance proof
fix k :: int show k div 0 = 0
by (simp add: divide-int-def)

next
fix k l :: int
assume l ≠ 0
obtain n m and s t where k: k = sgn s * int n and l: l = sgn t * int m
by (blast intro: int-sgnE elim: that)
then have k * l = sgn (s * t) * int (n * m)
by (simp add: ac-simps sgn-mult)
with k l (l ≠ 0) show k * l div l = k
by (simp only: divide-int-unfold)
qed (auto simp add: algebra-simps sgn-mult sgn-1-pos sgn-0-0)

end

lemma coprime-int-iff [simp]:
coprime (int m) (int n) ←→ coprime m n (is ?P ←→ ?Q)
proof
assume ?P
show ?Q
proof (rule coprimeI)
fix q
assume q dvd m q dvd n
then have int q dvd int m int q dvd int n
  by simp-all
with (?P) have is-unit (int q)
  by (rule coprime-common-divisor)
then show is-unit q
  by simp
qed

next
assume ?Q
show ?P
proof (rule coprimeI)
fix k
assume k dvd int m k dvd int n
then have nat |k| dvd m nat |k| dvd n
  by simp-all
with (?Q) have is-unit (nat |k|)
  by (rule coprime-common-divisor)
then show is-unit k
  by simp
qed

qed

lemma coprime-abs-left-iff [simp]:
coprime |k| l ←→ coprime k l for k l :: int
using coprime-normalize-left-iff [of k l] by simp
THEORY "Euclidean-Division"

lemma coprime-abs-right-iff [simp]:
  coprime k |l| ⟷ coprime k l for k l :: int
using coprime-abs-left-iff [of l k] by (simp add: ac-simps)

lemma coprime-nat-abs-left-iff [simp]:
coprime (nat |k|) n ⟷ coprime k (int n)
proof –
define m where m = nat |k|
then have |k| = int m
  by simp
moreover have coprime k (int n) ⟷ coprime |k| (int n)
  by simp
ultimately show ?thesis
  by simp
qed

lemma coprime-nat-abs-right-iff [simp]:
coprime n (nat |k|) ⟷ coprime (int n) k
using coprime-nat-abs-left-iff [of k n] by (simp add: ac-simps)

lemma coprime-common-divisor-int:
coprime a b =⇒ x dvd a =⇒ x dvd b =⇒ |x| = 1
  for a b :: int
  by (drule coprime-common-divisor [of - - x]) simp-all

instantiation int :: idom-modulo
begin

definition modulo-int :: int ⇒ int ⇒ int
  where k mod l = (if l = 0 then k
    else if sgn k = sgn l
      then sgn l * int (nat |k| mod nat |l|)
      else sgn l * ((|l| * of-bool (¬ l dvd k)) − int (nat |k| mod nat |l|)))

lemma modulo-int-unfold:
  (sgn k * int m) mod (sgn l * int n) =
  (if sgn l = 0 ∨ sgn k = 0 ∨ n = 0 then sgn k * int m
    else if sgn k = sgn l
      then sgn l * int (m mod n)
      else sgn l * (int (n * of-bool (¬ n dvd m)) − int (m mod n)))
  by (auto simp add: modulo-int-def sgn-0-0 sgn-1-pos sgn-mult abs-mult
    nat-mult-distrib)

instance proof
  fix k l :: int
  obtain n m and s t where k = sgn s * int n and l = sgn t * int m
    by (blast intro: int-sgnE elim: that)
  then show k dvd l * l + k mod l = k
THEORY “Euclidean-Division”

by (auto simp add: divide-int-unfold modulo-int-unfold algebra-simps dest!: sgn-not-eq-imp)
(simp-all add: of-nat-mult [symmetric] of-nat-add [symmetric]
distrib-left [symmetric] minus-mult-right
del: of-nat-mult minus-mult-right [symmetric])

qed

end

instantiation int :: unique-euclidean-ring

begin

definition euclidean-size-int :: int ⇒ nat
where euclidean-size-int = (nat ◦ abs :: int ⇒ nat)

definition division-segment-int :: int ⇒ int
where division-segment-int k = (if k ≥ 0 then 1 else −1)

lemma division-segment-eq-sgn:
division-segment k = sgn k if k ≠ 0 for k :: int
using that by (simp add: division-segment-int-def)

lemma abs-division-segment [simp]:
|division-segment k| = 1 for k :: int
by (simp add: division-segment-int-def)

lemma abs-mod-less:
|k mod l| < |l| if l ≠ 0 for k l :: int
proof –
obtain n m and s t where k = sgn s * int n and l = sgn t * int m
by (blast intro: int-sgnE elim: that)
with that show ?thesis
by (simp add: modulo-int-unfold sgn-0-0 sgn-1-pos sgn-1-neg
abs-mult mod-greater-zero-iff-not-dvd)

qed

lemma sgn-mod:
sgn (k mod l) = sgn l if l ≠ 0 ¬ l dvd k for k l :: int
proof –
obtain n m and s t where k = sgn s * int n and l = sgn t * int m
by (blast intro: int-sgnE elim: that)
with that show ?thesis
by (simp add: modulo-int-unfold sgn-0-0 sgn-1-pos sgn-1-neg
sgn-mult mod-eq-0-iff-dvd)

qed

instance proof
fix k l :: int
show division-segment (k mod l) = division-segment l if
l ≠ 0 and ¬ l dvd k
using that by (simp add: division-segment-eq-sgn dvd-eq-mod-eq-0 sgn-mod)

next
fix l q r :: int
obtain n m and s t
  where l: l = sgn s * int n and q: q = sgn t * int m
  by (blast intro: int-sgnE elim: that)
assume (l ≠ 0)
with l have s ≠ 0 and n > 0
  by (simp-all add: sgn-0-0)
assume division-segment r = division-segment l
moreover have r = sgn r * |r|
  by (simp add: sgn-mult-abs)
moreover define u where u = nat |r|
ultimately have r = sgn l * int u
  using division-segment-eq-sgn ⟨l ≠ 0⟩ by (cases r = 0) simp-all
with l ⟨n > 0⟩ have r: r = sgn n * int m
  by (simp add: sgn-mult)
show (q * l + r) div l = q
proof (cases q = 0 ∨ r = 0)
case True
  then show ?thesis
  proof
    assume q = 0
    then show ?thesis
      using l r ⟨u < n⟩ by (simp add: divide-int-unfold)
next
assume r = 0
from ⟨r = 0⟩ have *: q * l + r = sgn (t * s) * int (n * m)
  using q l by (simp add: ac-simps sgn-mult)
from ⟨s ≠ 0⟩ ⟨n > 0⟩ show ?thesis
  by (simp only: *, simp only: q l divide-int-unfold)
  (auto simp add: sgn-mult sgn-0-0 sgn-1-pos)
qed

next
case False
with q r have t ≠ 0 and m > 0 and s ≠ 0 and u > 0
  by (simp-all add: sgn-0-0)
moreover from ⟨0 < m⟩ ⟨u < n⟩ have u ≦ m * n
  using mult-le-less-imp-less [of 1 m u n] by simp
ultimately have *: q * l + r = sgn (s * t)
  * int (if t < 0 then m * n − u else m * n + u)
  using l q r
  by (simp add: sgn-mult algebra-simps of-nat-diff)
have (m * n − u) div n = m − 1 if u > 0
  using ⟨0 < m⟩ ⟨u < n⟩ that
by (auto intro: div-nat-eqI simp add: algebra-simps)
moreover have \( n \text{ dvd } m \ast n - u \leftrightarrow n \text{ dvd } u \)
using \( \langle u \leq m \ast n \rangle \text{ dvd-diffD1} \) [of \( n \ast m \ast u \)]
by auto
ultimately show \(? \text{thesis} \)
using \( \langle s \neq 0 \rangle \langle m > 0 \rangle \langle u > 0 \rangle \langle u < n \rangle \langle u \leq m \ast n \rangle \) 
(auto simp add: sgn-mult sgn-0-0 sgn-1-pos algebra-simps dest: dvd-imp-le)
qed

lemma pos-mod-bound [simp]:
\( k \mod l < l \) if \( l > 0 \) for \( k \ast l :: \text{int} \)
proof –
obtain \( m \) and \( s \) where \( k = sgn s \ast int m \)
by (rule int-sgnE)
moreover from that obtain \( n \) where \( l = sgn 1 \ast int n \)
by (cases \( l \)) simp-all
moreover from this that have \( n > 0 \)
by simp
ultimately show \(? \text{thesis} \)
by (simp only: modulo-int-unfold)
(simp add: mod-greater-zero-iff-not-dvd)
qed

lemma neg-mod-bound [simp]:
\( l < k \mod l \) if \( l < 0 \) for \( k \ast l :: \text{int} \)
proof –
obtain \( m \) and \( s \) where \( k = sgn s \ast int m \)
by (rule int-sgnE)
moreover from that obtain \( q \) where \( l = sgn (-1) \ast int (Suc q) \)
by (cases \( l \)) simp-all
moreover define \( n \) where \( n = Suc q \)
then have \( Suc q = n \)
by simp
ultimately show \(? \text{thesis} \)
by (simp only: modulo-int-unfold)
(simp add: mod-greater-zero-iff-not-dvd)
qed

lemma pos-mod-sign [simp]:
\( 0 \leq k \mod l \) if \( l > 0 \) for \( k \ast l :: \text{int} \)
proof –
obtain \( m \) and \( s \) where \( k = sgn s \ast int m \)
by (rule int-sgnE)
moreover from that obtain \( n \) where \( l = sgn 1 \ast int n \)
by (cases l) auto
moreover from this that have n > 0
by simp
ultimately show ?thesis
  by (simp only: modulo-int-unfold) simp
qed

lemma neg-mod-sign [simp]:
k mod l ≤ 0 if l < 0 for k l :: int
proof –
  obtain m and s where k = sgn s * int m
  by (rule int-sgnE)
moreover from that obtain q where l = sgn (- 1) * int (Suc q)
  by (cases l) simp-all
moreover define n where n = Suc q
then have Suc q = n
  by simp
ultimately show ?thesis
  by (simp only: modulo-int-unfold) simp
qed

54.6 Special case: euclidean rings containing the natural numbers

class unique-euclidean-semiring-with-nat = semidom + semiring-char-0 + unique-euclidean-semiring +
  assumes of-nat-div: of-nat (m div n) = of-nat m div of-nat n
  and division-segment-of-nat [simp]: division-segment (of-nat n) = 1
  and division-segment-euclidean-size [simp]: division-segment a * of-nat (euclidean-size a) = a
begin

lemma division-segment-eq-iff:
a = b if division-segment a = division-segment b
  and euclidean-size a = euclidean-size b
using that division-segment-euclidean-size [of a] by simp

lemma euclidean-size-of-nat [simp]:
euclidean-size (of-nat n) = n
proof –
  have division-segment (of-nat n) * of-nat (euclidean-size (of-nat n)) = of-nat n
    by (fact division-segment-euclidean-size)
then show ?thesis by simp
qed

lemma of-nat-euclidean-size:
of-nat (euclidean-size a) = a div division-segment a
proof –
  have of-nat (euclidean-size a) = division-segment a * of-nat (euclidean-size a)
div \text{division-segment} a
\text{by} \ (\text{subst} \ \text{nonzero-mult-cancel-left}) \ \text{simp-all}
\text{also have} \ldots = a \text{div} \text{division-segment} a
\text{by} \ \text{simp}
\text{finally show} \ \text{thesis} .
\text{qed}

\text{lemma} \ \text{division-segment-1} \ [\text{simp}]:
\text{division-segment} \ 1 = 1
\text{using} \ \text{division-segment-of-nat} \ [\text{of} \ 1] \ \text{by} \ \text{simp}

\text{lemma} \ \text{division-segment-numeral} \ [\text{simp}]:
\text{division-segment} \ (\text{numeral} \ k) = 1
\text{using} \ \text{division-segment-of-nat} \ [\text{of} \ \text{numeral} \ k] \ \text{by} \ \text{simp}

\text{lemma} \ \text{euclidean-size-1} \ [\text{simp}]:
\text{euclidean-size} \ 1 = 1
\text{using} \ \text{euclidean-size-of-nat} \ [\text{of} \ 1] \ \text{by} \ \text{simp}

\text{lemma} \ \text{euclidean-size-numeral} \ [\text{simp}]:
\text{euclidean-size} \ (\text{numeral} \ k) = \text{numeral} \ k
\text{using} \ \text{euclidean-size-of-nat} \ [\text{of} \ \text{numeral} \ k] \ \text{by} \ \text{simp}

\text{lemma} \ \text{of-nat-dvd-iff}:
of-nat \ m \ \text{dvd} \ of-nat \ n \ \iff \ m \ \text{dvd} \ n \ (\text{is} \ ?P \ \iff \ ?Q)
\text{proof} \ (\text{cases} \ m = 0)
\text{case} \ True
\text{then show} \ \text{thesis}
\text{by} \ \text{simp}
\text{next}
\text{case} \ False
\text{show} \ \text{thesis}
\text{proof}
\text{assume} \ ?Q
\text{then show} \ ?P
\text{by} \ \text{auto}
\text{next}
\text{assume} \ ?P
\text{with} \ False \ \text{have} \ of-nat \ n = \ of-nat \ n \ \text{div} \ of-nat \ m * \ of-nat \ m
\text{by} \ \text{simp}
\text{then have} \ of-nat \ n = \ of-nat \ (n \ \text{div} \ m * m)
\text{by} \ (\text{simp add: of-nat-div})
\text{then have} \ n = n \ \text{div} \ m * m
\text{by} \ (\text{simp only: of-nat-eq-iff})
\text{then have} \ n = m * (n \ \text{div} \ m)
\text{by} \ (\text{simp add: ac-simps})
\text{then show} \ ?Q ..
\text{qed}
\text{qed}
lemma of-nat-mod:
  of-nat (m mod n) = of-nat m mod of-nat n
proof −
  have of-nat m div of-nat n * of-nat n + of-nat m mod of-nat n = of-nat m
    by (simp add: div-mult-mod-eq)
  also have of-nat m = of-nat (m div n * n + m mod n)
    by simp
  finally show ?thesis
    by (simp only: of-nat-div of-nat-mult of-nat-add) simp
qed

lemma one-div-two-eq-zero [simp]:
  1 div 2 = 0
proof −
  from of-nat-div [symmetric] have of-nat 1 div of-nat 2 = of-nat 0
    by (simp only:) simp
  then show ?thesis
    by simp
qed

lemma one-mod-two-eq-one [simp]:
  1 mod 2 = 1
proof −
  from of-nat-mod [symmetric] have of-nat 1 mod of-nat 2 = of-nat 1
    by (simp only:) simp
  then show ?thesis
    by simp
qed

lemma one-mod-2-pow-eq [simp]:
  1 mod (2 ^ n) = of-bool (n > 0)
proof −
  have 1 mod (2 ^ n) = of-nat (1 mod (2 ^ n))
    using of-nat-mod [of 1 2 ^ n] by simp
  also have ... = of-bool (n > 0)
    by simp
  finally show ?thesis .
qed

lemma one-div-2-pow-eq [simp]:
  1 div (2 ^ n) = of-bool (n = 0)
using div-mult-mod-eq [of 1 2 ^ n] by auto

lemma div-mult2-eq':
  a div (of-nat m * of-nat n) = a div of-nat m div of-nat n
proof (cases a of-nat m * of-nat n rule: divmod-cases)
  case (divides q)
  then show ?thesis
using nonzero-mult-div-cancel-right [of of-nat m q * of-nat n]
by (simp add: ac-simps)

next

  case (remainder q r)
  then have division-segment r = 1
    using division-segment-of-nat [of m * n] by simp
  with division-segment-euclidean-size [of r]
  have of-nat (euclidean-size r) = r
    by simp
  have a mod (of-nat m * of-nat n) div (of-nat m * of-nat n) = 0
    by simp
  with remainder(6) have r div (of-nat m * of-nat n) = 0
    by simp
  with (of-nat (euclidean-size r) = r)
  have of-nat (euclidean-size r) div (of-nat m * of-nat n) = 0
    by simp
  then have of-nat (euclidean-size r div (m * n)) = 0
    by (simp add: of-nat-div)
  then have of-nat (euclidean-size r div m div n) = 0
    by (simp add: div-mult2-eq)
  with (of-nat (euclidean-size r) = r) have r div of-nat m div of-nat n = 0
    by (simp add: of-nat-div)
  with remainder(1)
  have q = (r div of-nat m + q * of-nat n) div of-nat m div of-nat n
    by simp
  with remainder(5) remainder(7) show ?thesis
    using div-plus-div-distrib-dvd-right [of of-nat m q * (of-nat m * of-nat n) r]
    by (simp add: ac-simps)

next

  case by0
  then show ?thesis
    by auto
qed


lemma mod-mult2-eq':
  a mod (of-nat m * of-nat n) = of-nat m * (a div of-nat m mod of-nat n) + a mod of-nat m
proof
  have a div (of-nat m * of-nat n) * (of-nat m * of-nat n) + a mod (of-nat m * of-nat n) = a div of-nat m div of-nat n * of-nat n * of-nat m + (a div of-nat m mod of-nat n * of-nat m + a mod of-nat m)
    by (simp add: combine-common-factor div-mult-mod-eq)
  moreover have a div of-nat m div of-nat n * of-nat n * of-nat m = of-nat n * of-nat m * (a div of-nat m div of-nat n)
    by (simp add: ac-simps)
  ultimately show ?thesis
    by (simp add: div-mult2-eq' mult-commute)
qed
lemma div-mult2-numeral-eq:
  \( a \div \text{numeral } k \div \text{numeral } l = a \div \text{numeral } (k \times l) \) (is \( ?A = ?B \))
proof
  have \( ?A = a \div \text{of-nat } (\text{numeral } k) \div \text{of-nat } (\text{numeral } l) \)
    by simp
  also have \( \ldots = a \div (\text{of-nat } (\text{numeral } k) \times \text{of-nat } (\text{numeral } l)) \)
    by (fact div-mult2-eq [symmetric])
  also have \( \ldots = ?B \)
    by simp
  finally show \( \text{thesis} \).
qed

lemma numeral-Bit0-div-2:
  \( \text{numeral } (\text{num.Bit0 } n) \div 2 = \text{numeral } n \)
proof
  have \( \text{numeral } (\text{num.Bit0 } n) = \text{numeral } n + \text{numeral } n \)
    by (simp only: numeral.simps)
  also have \( \ldots = \text{numeral } n \times 2 \)
    by (simp add: mult-2-right)
  finally have \( \text{numeral } (\text{num.Bit0 } n) \div 2 = \text{numeral } n \times 2 \div 2 \)
    by simp
  also have \( \ldots = \text{numeral } n \)
    by (rule nonzero-mult-div-cancel-right) simp
  finally show \( \text{thesis} \).
qed

lemma numeral-Bit1-div-2:
  \( \text{numeral } (\text{num.Bit1 } n) \div 2 = \text{numeral } n \)
proof
  have \( \text{numeral } (\text{num.Bit1 } n) = \text{numeral } n + \text{numeral } n + 1 \)
    by (simp only: numeral.simps)
  also have \( \ldots = \text{numeral } n \times 2 + 1 \)
    by (simp add: mult-2-right)
  finally have \( \text{numeral } (\text{num.Bit1 } n) \div 2 = (\text{numeral } n \times 2 + 1) \div 2 \)
    by simp
  also have \( \ldots = \text{numeral } n \times 2 \div 2 + 1 \div 2 \)
    using dvd-triv-right by (rule div-plus-die-distrib-dvd-left)
  also have \( \ldots = \text{numeral } n \div 2 \)
    by simp
  also have \( \ldots = \text{numeral } n \)
    by (rule nonzero-mult-div-cancel-right) simp
  finally show \( \text{thesis} \).
qed

lemma exp-mod-exp:
  \( 2^m \mod 2^n = \text{of-bool } (m < n) \times 2^m \)
proof
  have \( (\text{2::nat})^m \mod 2^n = \text{of-bool } (m < n) \times 2^m \) (is \( ?\text{lhs} = ?\text{rhs} \))
    by (auto simp add: not-less monoid-mult-class.power-dest!: le-Suc-ex)
then have \langle \text{of-nat} \ ?\text{lhs} = \text{of-nat} \ ?\text{rhs} \rangle 
  by simp
then show ?thesis 
  by (simp add: \text{of-nat-mod})
qed

lemma \textit{mask-mod-exp}:
\langle (2 \cdot n - 1) \mod 2 \cdot m = 2 \cdot \min m n - 1 \rangle
proof -
have \langle (2 \cdot n - 1) \mod 2 \cdot m = 2 \cdot \min m n - (1::nat) \rangle 
  (is \langle ?\text{lhs} = ?\text{rhs} \rangle)
proof (cases \langle n \leq m \rangle)
case True
  then show ?thesis 
    by (simp add: Suc-le-lessD min.absorb2)
next
case False
  then have \langle m < n \rangle 
    by simp
  then obtain q where n: \langle n = \text{Suc} q + m \rangle 
    by (auto dest: less-imp-Suc-add)
  then have \langle \min m n = m \rangle 
    by simp
  moreover have \langle (2::nat) \cdot m \leq 2 \cdot 2 \cdot q \cdot 2 \cdot m \rangle 
    using mult-le-mono1 [of 1 \langle 2 \cdot 2 \cdot q \rangle \langle 2 \cdot m \rangle] by simp
  with n have \langle 2 \cdot n - 1 = (2 \cdot \text{Suc} q - 1) \cdot 2 \cdot m + (2 \cdot m - (1::nat)) \rangle 
    by (simp add: monoid-mult-class.power-add algebra-simps)
  ultimately show ?thesis 
    by (simp only: euclidean-semiring-cancel-class.mod-mult-self3) simp
qed
then have \langle \text{of-nat} \ ?\text{lhs} = \text{of-nat} \ ?\text{rhs} \rangle 
  by simp
then show ?thesis 
  by (simp add: \text{of-nat-mod \ of-nat-diff})
qed

lemma \textit{of-bool-half-eq-0} [simp]:
\langle \text{of-bool} b \div 2 = 0 \rangle
by simp

end

class \textit{unique-euclidean-ring-with-nat} = \textit{ring} + \textit{unique-euclidean-semiring-with-nat}
instance nat :: \textit{unique-euclidean-semiring-with-nat}
  by standard (simp-all add: dvd-eq-mod-eq-0)
instance int :: \textit{unique-euclidean-ring-with-nat}
  by standard (simp-all add: dvd-eq-mod-eq-0 divide-int-def division-segment-int-def)
54.7 Code generation

code-identifier
  code-module Euclidean-Division \rightarrow (SML) Arith and (OCaml) Arith and (Haskell) Arith

end

55 Parity in rings and semirings

theory Parity
  imports Euclidean-Division
begin

55.1 Ring structures with parity and even/odd predicates

class semiring-parity = comm-semiring-1 + semiring-modulo +
  assumes even-iff-mod-2-eq-zero: 2 dvd a \leftrightarrow a mod 2 = 0
  and odd-iff-mod-2-eq-one: \neg 2 dvd a \leftrightarrow a mod 2 = 1
  and odd-one [simp]: \neg 2 dvd 1
begin

abbreviation even :: 'a \Rightarrow bool
  where even a \equiv 2 dvd a

abbreviation odd :: 'a \Rightarrow bool
  where odd a \equiv \neg 2 dvd a

lemma parity-cases [case-names even odd]:
  assumes even a = \Rightarrow a mod 2 = 0 = \Rightarrow P
  assumes odd a = \Rightarrow a mod 2 = 1 = \Rightarrow P
  shows P
  using assms by (cases even a)
  (simp-all add: even-iff-mod-2-eq-zero [symmetric] odd-iff-mod-2-eq-one [symmetric])

lemma odd-of-bool-self [simp]:
  (odd (of-bool p) \leftrightarrow p)
  by (cases p) simp-all

lemma not-mod-2-eq-0-eq-1 [simp]:
  a mod 2 \neq 0 \leftrightarrow a mod 2 = 1
  by (cases a rule: parity-cases) simp-all

lemma not-mod-2-eq-1-eq-0 [simp]:
  a mod 2 \neq 1 \leftrightarrow a mod 2 = 0
  by (cases a rule: parity-cases) simp-all

lemma evenE [elim?]:
  assumes even a
obtains $b$ where $a = 2 \times b$
using assms by (rule dvdE)

lemma oddE [elim?!]:
  assumes odd $a$
  obtains $b$ where $a = 2 \times b + 1$
proof –
  have $a = 2 \times (a \div 2) + a \mod 2$
    by (simp add: mult-div-mod-eq)
  with assms have $a = 2 \times (a \div 2) + 1$
    by (simp add: odd-iff-mod-2-eq-one)
  then show ?thesis ..
qed

lemma mod-2-eq-odd:
  $a \mod 2 = \text{of-bool (odd } a\text{)}$
by (auto elim: oddE simp add: even-iff-mod-2-eq-zero)

lemma of-bool-odd-eq-mod-2:
  $\text{of-bool (odd } a\text{)} = a \mod 2$
by (simp add: mod-2-eq-odd)

lemma even-mod-2-iff [simp]:
  $(\text{even } (a \mod 2) \longleftrightarrow \text{even } a)$
by (simp add: mod-2-eq-odd)

lemma mod2-eq-if:
  $a \mod 2 = (\text{if even } a \text{ then } 0 \text{ else } 1)$
by (simp add: mod-2-eq-odd)

lemma even-zero [simp]:
  even 0
by (fact dvd-0-right)

lemma odd-even-add:
  even $(a + b)$ if odd $a$ and odd $b$
proof –
  from that obtain $c \; d$ where $a = 2 \times c + 1$ and $b = 2 \times d + 1$
    by (blast elim: oddE)
  then have $a + b = 2 \times c + 2 \times d + (1 + 1)$
    by (simp only: ac-simps)
  also have ... $= 2 \times (c + d + 1)$
    by (simp add: algebra-simps)
  finally show ?thesis ..
qed

lemma even-add [simp]:
  even $(a + b) \longleftrightarrow (\text{even } a \longleftrightarrow \text{even } b)$
by (auto simp add: dvd-add-right-iff dvd-add-left-iff odd-even-add)
lemma odd-add [simp]:
  odd (a + b) ↔ ¬(odd a ↔ odd b)
by simp

lemma even-plus-one-iff [simp]:
even (a + 1) ↔ odd a
by (auto simp add: dvd-add-right-iff intro: odd-even-add)

lemma even-mul-iff [simp]:
even (a * b) ↔ even a ∨ even b (is ?P ↔ ?Q)
proof
  assume ?Q
  then show ?P
    by auto
next
  assume ?P
  show ?Q
    proof (rule ccontr)
      assume ¬(even a ∨ even b)
      then have odd a and odd b
        by auto
      then obtain r s where a = 2 * r + 1 and b = 2 * s + 1
        by (blast elim: oddE)
      then have a * b = (2 * r + 1) * (2 * s + 1)
        by simp
      also have ... = 2 * (2 * r * s + r + s) + 1
        by (simp add: algebra-simps)
      finally have odd (a * b)
        by simp
      with ⟨?P; show False
        by auto
      qed

lemma even-numeral [simp]: even (numeral (Num.Bit0 n))
proof
  have even (2 * numeral n)
    unfolding even-mul-iff by simp
  then have even (numeral n + numeral n)
    unfolding mult-2
  then show ?thesis
    unfolding numeral.simps
  qed

lemma odd-numeral [simp]: odd (numeral (Num.Bit1 n))
proof
  assume even (numeral (Num.Bit1 n))
  then have even (numeral n + numeral n + 1)
unfolding numeral.simps.
then have even \((2 \ast \text{numeral } n + 1)\)
unfolding mult-2.
then have \(2 \text{ dvd numeral } n \ast 2 + 1\)
by (simp add: ac-simps)
then have \(2 \text{ dvd 1}\)
using dvd-add-times-triv-left-iff [of 2 numeral n 1] by simp
then show False by simp
qed

lemma even-power [simp]: even \((a ^ n)\) ↔ even a ∧ n > 0
by (induct n) auto

lemma mask-eq-sum-exp:
\(2 ^ n - 1 = (\sum_{q. q < n} 2 ^ m)\)
proof –
have \*: \(\{q. q < \text{Suc } m\} = \text{insert } m \{q. q < m\}\); for m
by auto
have \((2 ^ n = (\sum_{m \in \{q. q < n\}} 2 ^ m) + 1)\)
by (induction n) (simp-all add: ac-simps mult-2 \*)
then have \((2 ^ n - 1 = (\sum_{m \in \{q. q < n\}} 2 ^ m) + 1 - 1)\)
by simp
then show \(?thesis\)
by simp
qed

lemma mask-eq-seq-sum:
\((2 ^ n - 1 = ((\lambda k. 1 + k \ast 2) ^ n) 0)\)
proof –
have \((2 ^ n = ((\lambda k. 1 + k \ast 2) ^ n) 0 + 1)\)
by (induction n) (simp-all add: ac-simps mult-2)
then show \(?thesis\)
by simp
qed
55.2 Special case: euclidean rings containing the natural numbers

context unique-euclidean-semiring-with-nat
begin

subclass semiring-parity
proof
  show \( 2 \) dvd \( a \) \( \iff \) \( a \mod 2 = 0 \) for \( a \)
    by (fact dvd-eq-mod-eq-0)
  show \( \neg 2 \) dvd \( a \) \( \iff \) \( a \mod 2 = 1 \) for \( a \)
    proof
      assume \( a \mod 2 = 1 \)
      then show \( \neg 2 \) dvd \( a \)
        by auto
    next
      assume \( \neg 2 \) dvd \( a \)
      have \( \text{eucl: euclidean-size (} a \mod 2 \text{) = 1} \)
        proof (rule order-antisym)
          show \( \text{euclidean-size (} a \mod 2 \text{)} \leq 1 \)
            using mod-size-less [of \( 2 \) \( a \)] by simp
          show \( 1 \leq \text{euclidean-size (} a \mod 2 \text{)} \)
            using \( \neg 2 \) dvd \( a \) by (simp add: Suc-le-eq dvd-eq-mod-eq-0)
        qed
      from \( \neg 2 \) dvd \( a \) have \( \neg \text{of-nat } 2 \) dvd \( \text{division-segment } a \ast \text{of-nat (euclidean-size } a \text{)} \)
        by simp
      then have \( \neg \text{of-nat } 2 \) dvd \( \text{of-nat (euclidean-size } a \text{)} \)
        by (auto simp only: dvd-mult-unit-iff is-unit-division-segment)
      then have \( \neg 2 \) dvd \( \text{euclidean-size } a \)
        using of-nat-dvd-iff [of \( 2 \)] by simp
      then have \( \text{euclidean-size } a \mod 2 = 1 \)
        by (simp add: semidom-modulo-class dvd-eq-mod-eq-0)
      then have \( \text{of-nat (euclidean-size } a \mod 2 \text{) = of-nat } 1 \)
        by simp
      then have \( \neg \text{of-nat (euclidean-size } a \text{) mod } 2 = 1 \)
        by (simp add: of-nat-mod)
      from \( \neg 2 \) dvd \( a \) \( \text{eucl} \)
      show \( a \mod 2 = 1 \)
        by (auto intro: division-segment-eq-iff simp add: division-segment-mod)
    qed
show \( \neg \text{is-unit } 2 \)
proof (rule notI)
  assume \( \text{is-unit } 2 \)
  then have \( \neg \text{of-nat } 2 \) dvd \( \text{of-nat } 1 \)
    by simp
then have is-unit (2 :: nat)
  by (simp only: of-nat-dvd-iff)
then show False
  by simp
qed

lemma even-of-nat [simp]:
even (of-nat a) ↔ even a
proof —
have even (of-nat a) ↔ of-nat 2 dvd of-nat a
  by simp
also have . . . ↔ even a
  by (simp only: of-nat-dvd-iff)
finally show ?thesis .
qed

lemma even-succ-div-two [simp]:
even a ⇒ (a + 1) div 2 = a div 2
by (cases a = 0) (auto elim!: evenE dest: mult-not-zero)

lemma odd-succ-div-two [simp]:
odd a ⇒ (a + 1) div 2 = a div 2 + 1
by (auto elim!: oddE simp add: add.assoc)

lemma even-two-times-div-two:
even a ⇒ 2 * (a div 2) = a
by (fact dvd-mult-div-cancel)

lemma odd-two-times-div-two-succ [simp]:
odd a ⇒ 2 * (a div 2) + 1 = a
using mult-div-mod-eq [of 2 a]
by (simp add: even-iff-mod-2-eq-zero)

lemma coprime-left-2-iff-odd [simp]:
coprime 2 a ↔ odd a
proof
  assume odd a
show coprime 2 a
proof (rule coprimeI)
  fix b
  assume b dvd 2 b dvd a
then have b dvd a mod 2
    by (auto intro: dvd-mod)
  with (odd a) show is-unit b
    by (simp add: mod-2-eq-odd)
  qed
next
  assume coprime 2 a

show odd a
proof (rule notI)
  assume even a
  then obtain b where a = 2 * b ..
  with (coprime 2 a) have coprime 2 (2 * b)
    by simp
  moreover have ~ coprime 2 (2 * b)
    by (rule not-coprimeI [of 2]) simp-all
  ultimately show False
    by blast
qed
qed

lemma coprime-right-2-iff-odd [simp]:
  coprime a 2 <-> odd a
  using coprime-left-2-iff-odd [of a] by (simp add: ac-simps)
end

context unique-euclidean-ring-with-nat
begin

subclass ring-parity ..

lemma minus-1-mod-2-eq [simp]:
  -1 mod 2 = 1
  by (simp add: mod-2-eq-odd)

lemma minus-1-div-2-eq [simp]:
  -1 div 2 = -1
proof-
  from div-mult-mod-eq [of - 1 2]
  have -1 div 2 * 2 = -1 * 2
    using add-implies-diff by fastforce
  then show ?thesis
    using mult-right-cancel [of 2 - 1 div 2 - 1] by simp
qed

end

55.3 Instance for nat

instance nat :: unique-euclidean-semiring-with-nat
  by standard (simp-all add: dvd-eq-mod-eq-0)

lemma even-Suc-Suc-iff [simp]:
  even (Suc (Suc n)) <-> even n
  using dvd-add-triv-right-iff [of 2 n] by simp
lemma even-Suc [simp]: even (Suc n) ⟷ odd n
using even-plus-one-iff [of n] by simp

lemma even-diff-nat [simp]:
   even (m − n) ⟷ m < n ∨ even (m + n) for m n :: nat
proof (cases n ≤ m)
case True
then have m − n + n * 2 = m + n by (simp add: mult-2-right)
moreover have even (m − n) ⟷ even (m − n + n * 2) by simp
ultimately have even (m − n) ⟷ even (m + n) by (simp only:)
then show ?thesis by auto
next
case False
then show ?thesis by simp
qed

lemma odd-pos:
   odd n =⇒ 0 < n for n :: nat
by (auto elim: oddE)

lemma Suc-double-not-eq-double:
   Suc (2 * m) ≠ 2 * n
proof
assume Suc (2 * m) = 2 * n
moreover have odd (Suc (2 * m)) and even (2 * n)
   by simp-all
ultimately show False by simp
qed

lemma double-not-eq-Suc-double:
   2 * m ≠ Suc (2 * n)
using Suc-double-not-eq-double [of n m] by simp

lemma odd-Suc-minus-one [simp]: odd n =⇒ Suc (n − Suc 0) = n
by (auto elim: oddE)

lemma even-Suc-div-two [simp]:
   even n =⇒ Suc n div 2 = n div 2
using even-succ-div-two [of n] by simp

lemma odd-Suc-div-two [simp]:
   odd n =⇒ Suc n div 2 = Suc (n div 2)
using odd-succ-div-two [of n] by simp

lemma odd-two-times-div-two-nat [simp]:
assumes odd n
shows 2 * (n div 2) = n − (1 :: nat)
proof −
from assms have 2 * (n div 2) + 1 = n
by (rule odd-two-times-div-two-succ)
then have \( \text{Suc} \left( 2 \times (n \text{ div } 2) \right) - 1 = n - 1 \)
  by simp
then show \(?thesis\)
  by simp
qed

lemma \(\text{not-mod2-eq-Suc-0-eq-0}\) [simp]:
\( n \text{ mod } 2 \neq \text{Suc} 0 \leftrightarrow n \text{ mod } 2 = 0 \)
using \(\text{not-mod-2-eq-1-eq-0}\) [of \(n\)] by simp

lemma \(\text{odd-card-imp-not-empty}\):
\(\{A \neq \{}\) if \(\langle\text{odd } (\text{card } A)\rangle\)
using that by auto

lemma \(\text{nat-induct2}\) [case-names 0 1 step]:
assumes \(P \ 0\ P \ 1\) and step: \(\\land\:\text{nat}. \ P \ n \Rightarrow P \ (n + 2)\)
shows \(P \ n\)
proof (induct \(n\) rule: less-induct)
case \(\text{less } n\)
show \(?case\)
proof (cases \(n < \text{Suc} (\text{Suc} \ 0)\))
case True
then show \(?thesis\)
  using \(\text{assms}\) by (auto simp: \(\text{less-Suc-eq}\))
next
case False
then obtain \(k\) where \(n = \text{Suc} (\text{Suc} \ k)\)
  by (force simp: \(\text{not-less nat-le-iff-add}\))
then have \(k<n\)
  by simp
with \(\text{less \ assms \ have } P \ (k+2)\)
  by blast
then show \(?thesis\)
  by (simp add: \(k\))
qed

lemmas \(\text{mask-eq-sum-exp-nat}\):
\(\begin{array}{l}
2 ^ \ n - \text{Suc} 0 = (\sum_{m \in \{q. \ q < n\}. \ 2 ^ \ m}) \\
\end{array}\)
using \(\text{mask-eq-sum-exp}\) [where \(?'a = \text{nat}\)] by simp

context semiring-parity
begin

lemma \(\text{even-sum-iff}\):
\(\langle\text{even } (\text{sum } f \ A) \leftrightarrow \text{even } (\text{card } \{a \in A. \ \text{odd } (f \ a)\}\rangle\) if \(\langle\text{finite } A\rangle\)
using that proof (induction \(A\))
case empty
then show \(?\text{case}\)
  by simp
next
  case (insert a A)
  moreover have \(\{ b \in \text{insert} \ a \ A. \ \text{odd} \ (f \ b) \} = \{ \text{if odd} \ (f \ a) \ \text{then} \ \{ a \} \ \text{else} \ \{ \} \}) \cup \{ b \in A. \ \text{odd} \ (f \ b) \} \)
    by auto
  ultimately show \(?\text{case}\)
    by simp
qed

lemma \(\text{even-prod-iff}\\):\)
\(\langle \text{even} \ (\text{prod} \ f \ A) \longleftrightarrow (\exists a \in A. \ \text{even} \ (f \ a)) \rangle \text{ if } (\text{finite} \ A)\)
using that by (induction \(A\)) simp-all

lemma \(\text{even-mask-iff} \ [\text{simp}]\\):\)
\(\langle \text{even} \ (2 ^ n - 1) \longleftrightarrow n = 0 \rangle \)
proof (cases \(\langle n = 0 \rangle\))
  case True
  then show \(?\text{thesis}\)
    by simp
next
  case False
  then have \(\{ a. \ a = 0 \land a < n \} = \{ 0 \}\)
    by auto
  then show \(?\text{thesis}\)
    by (auto simp add: mask-eq-sum-exp even-sum-iff)
qed

end

55.4 Parity and powers

context ring-1
begin

lemma \(\text{power-minus-even} \ [\text{simp}]:\)
\(\langle \text{even} \ n \Longrightarrow (\sim \ a) ^ n = a ^ n \rangle \)
by (auto elim: evenE)

lemma \(\text{power-minus-odd} \ [\text{simp}]:\)
\(\langle \text{odd} \ n \Longrightarrow (\sim \ a) ^ n = - (a ^ n) \rangle \)
by (auto elim: oddE)

lemma \(\text{uminus-power-if}\\):\)
\((\sim \ a) ^ n = (\text{if even} \ n \ \text{then} \ a ^ n \ \text{else} \ - (a ^ n))\)
by auto

lemma \(\text{neg-one-even-power} \ [\text{simp}]:\)
\(\langle \text{even} \ n \Longrightarrow (\sim 1) ^ n = 1 \rangle \)
by simp
lemma neg-one-odd-power [simp]: odd n ⟹ (−1) ^ n = −1
  by simp

lemma neg-one-power-add-eq-neg-one-power-diff: k ≤ n ⟹ (−1) ^ (n + k) =
  (−1) ^ (n − k)
  by (cases even (n + k)) auto

lemma minus-one-power-iff: (−1) ^ n = (if even n then 1 else −1)
  by (induct n) auto

end

context linordered-idom
begin

lemma zero-le-even-power: even n ⟹ 0 ≤ a ^ n
  by (auto elim: evenE)

lemma zero-le-odd-power: odd n ⟹ 0 ≤ a ^ n ⟷ 0 ≤ a
  by (auto simp add: power-even-eq zero-le-mult-iff elim: oddE)

lemma zero-le-power-eq: 0 ≤ a ^ n ⟷ even n ∨ odd n ∧ 0 ≤ a
  by (auto simp add: zero-le-even-power zero-le-odd-power)

lemma zero-less-power-eq: 0 < a ^ n ⟷ n = 0 ∨ even n ∧ a ≠ 0 ∨ odd n ∧ 0
  < a
  proof −
  have [simp]: 0 = a ^ n ⟷ a = 0 ∧ n > 0
    unfolding power-eq-0-iff [of a n, symmetric] by blast
  show ?thesis
    unfolding less-le zero-le-power-eq by auto
  qed

lemma power-less-zero-eq [simp]: a ^ n < 0 ⟷ odd n ∧ a < 0
  unfolding not-le [symmetric] zero-le-power-eq by auto

lemma power-zero-eq: a ^ n ≤ 0 ⟷ n ≥ 0 ∧ (odd n ∧ a ≤ 0 ∨ even n ∧ a
  = 0)
  unfolding not-less [symmetric] zero-less-power-eq by auto

lemma power-even-abs: even n ⟹ |a| ^ n = a ^ n
  using power-abs [of a n] by (simp add: zero-le-even-power)

lemma power-mono-even:
  assumes even n and |a| ≤ |b|
  shows a ^ n ≤ b ^ n
  proof −
  have 0 ≤ |a| by auto
  with |a| ≤ |b| have |a| ^ n ≤ |b| ^ n
by \((\text{rule power-mono})\)

with \((\text{even } n)\) show ?thesis

by \((\text{simp add: power-even-abs})\)

qed

\begin{lemma}
\textbf{power-mono-odd:}

assumes \(\text{odd } n\) and \(a \leq b\)

shows \(a^n \leq b^n\)

\begin{proof}
\begin{cases}
\text{case True} & \\
\text{case False}
\end{cases}

\begin{proof}
\begin{cases}
\text{cases } a < 0 & \\
\text{cases } a \geq 0
\end{cases}

\begin{proof}
\begin{cases}
\text{cases } b < 0 & \\
\text{cases } b \geq 0
\end{cases}

\begin{proof}
\begin{cases}
\text{cases } -b < 0 & \\
\text{cases } -b \geq 0
\end{cases}

\begin{proof}
\end{proof}

\end{proof}

\end{proof}

\end{proof}

\text{show } ?\text{thesis by simp}

qed

\end{proof}

\end{lemma}

Simplify, when the exponent is a numeral

\begin{lemma}
\textbf{zero-le-power-eq-numeral [simp]:}

\(0 \leq a^n \leftrightarrow \text{numeral } w \leq a\)

by \((\text{fact zero-le-power-eq})\)

\begin{lemma}
\textbf{zero-less-power-eq-numeral [simp]:}

\(0 < a^n \leftrightarrow \text{numeral } w < a\)

by \((\text{fact zero-less-power-eq})\)

\begin{lemma}
\textbf{power-le-zero-eq-numeral [simp]:}

\(a^n \leq 0 \leftrightarrow \text{numeral } w < 0\)

by \((\text{fact power-le-zero-eq})\)

\end{lemma}
**THEORY “Parity”**

lemma power-less-zero-eq-numeral [simp]:
\[ a \cdot \text{numeral} w < 0 \leftrightarrow \text{odd} (\text{numeral} w :: \text{nat}) \land a < 0 \]
by (fact power-less-zero-eq)

lemma power-even-abs-numeral [simp]:
\[ \text{even} (\text{numeral} w :: \text{nat}) \implies |a| \cdot \text{numeral} w = a \cdot \text{numeral} w \]
by (fact power-even-abs)

end

context unique-euclidean-semiring-with-nat
begin

lemma even-mask-div-iff':
\[ \langle \text{even} ((2 \cdot m - 1) \div 2 \cdot n) \leftrightarrow m \leq n \rangle \]
proof
- have \[ \langle \text{even} ((2 \cdot m - 1) \div 2 \cdot n) \leftrightarrow \text{even} ((2 \cdot m - \ Suc 0) \div 2 \cdot n) \rangle \]
  by (simp only: of-nat-div)
also have \[ \langle \ldots \leftrightarrow \text{even} ((2 \cdot m - \ Suc 0) \div 2 \cdot n) \rangle \]
  by simp
also have \[ \langle \ldots \leftrightarrow m \leq n \rangle \]
proof (cases \( m \leq n \))
  case True
  then have \[ \langle \text{odd} (\sum_{a \in \{q. \ q < m \}} (2 \cdot a \div 2 \cdot n) \cdot q) \rangle \]
    by (simp add: Suc-le-lessD)
  ultimately have \[ \langle \text{odd} (\sum_{a \in \{q. \ q < m \}} (2 \cdot a \div 2 \cdot n) \cdot q) \rangle \]
    by (subst euclidean-semiring-cancel-class.sum-div-partition) simp-all
  qed
finally show \[ \langle \text{optional} \rangle \]

end
55.5 Instance for int

lemma even-diff-iff:
  even \((k - l)\) \iff even \((k + l)\) for \(k, l : int\)
  by (fact even-diff)

lemma even-abs-add-iff:
  even \((|k| + l)\) \iff even \((k + l)\) for \(k, l : int\)
  by simp

lemma even-add-abs-iff:
  even \((k + |l|)\) \iff even \((k + l)\) for \(k, l : int\)
  by simp

lemma even-nat-iff:
  \(0 \leq k \Rightarrow even (nat k) \iff even k\)
  by (simp add: even-of-nat \[of nat k\], where ?'a = int, symmetric)

lemma zdiv-zmult2-eq:
  \(\langle a \div (b \ast c) = (a \div b) \div c \rangle\) if \(c \geq 0\) for \(a, b, c : int\)
  proof (cases \(b \geq 0\))
    case True
    with that show \(?\)thesis
    using div-mult2-eq' \[of a \langle nat b \rangle \langle nat c \rangle\] by simp
  next
    case False
    with that show \(?\)thesis
    using div-mult2-eq' \[of \langle - a \rangle \langle nat (- b) \rangle \langle nat c \rangle\] by simp
  qed

lemma zmod-zmult2-eq:
  \(\langle a \mod (b \ast c) = b \ast (a \div b \mod c) + a \mod b \rangle\) if \(c \geq 0\) for \(a, b, c : int\)
  proof (cases \(b \geq 0\))
    case True
    with that show \(?\)thesis
    using mod-mult2-eq' \[of a \langle nat b \rangle \langle nat c \rangle\] by simp
  next
    case False
    with that show \(?\)thesis
    using mod-mult2-eq' \[of \langle - a \rangle \langle nat (- b) \rangle \langle nat c \rangle\] by simp
  qed

55.6 Abstract bit structures

class semiring-bits = semiring-parity +
  assumes bits-induct [case-names stable rec]:
  \((\forall a. a \div 2 = a \Rightarrow P a)\)
\[ \implies (\forall a, b. \ P a \implies (\text{of-bool } b + 2 \times a) \div 2 = a \implies P (\text{of-bool } b + 2 \times a)) \]
\[ \implies P a \]

assumes bits-div-0 [simp]: \(0 \div a = 0\)
and bits-div-by-1 [simp]: \(a \div 1 = a\)
and bits-mod-div-trivial [simp]: \(a \mod b \div b = 0\)
and even-succ-div-2 [simp]: \(\text{even } a \implies (1 + a) \div 2 = a \div 2\)
and even-mask-div iff: \(\text{even } ((2 \times m - 1) \div 2 \times n) \iff 2 \times n = 0 \lor m \leq n\)
and exp-div-exp eq: \(2 \times m \div 2 \times n = \text{of-bool } (2 \times m \neq 0 \land m \geq n) \times 2 \times (m - n)\)
and div-exp eq: \(a \div 2 \times m \div 2 \times n = a \div 2 \times (m + n)\)
and mod-exp eq: \(a \mod 2 \times m \mod 2 \times n = a \mod 2 \times \text{min } m \times n\)
and mult-exp-mod-exp eq: \(m \leq n \implies (a \times 2 \times m) \mod (2 \times n) = (a \mod 2 \times (n - m)) \times 2 \times m\)
and div-exp-mod-exp eq: \(a \div 2 \times n \mod 2 \times m = a \mod (2 \times (n + m)) \div 2 \times n\)
and even-mult-exp-div-iff: \(\text{even } (a \times 2 \times m \div 2 \times n) \iff m > n \lor 2 \times n = 0 \lor (m \leq n \land \text{even } (a \div 2 \times (n - m)))\)

begin

lemma bits-div-by-0 [simp]:
\(a \div 0 = 0\)
by (metis add-cancel-right-right bits-mod-div-trivial mod-mult-div eq mult-not-zero)

lemma bits-1-div-2 [simp]:
\(1 \div 2 = 0\)
using even-succ-div-2 [of 0] by simp

lemma bits-1-div-exp [simp]:
\(1 \div 2 \times n = \text{of-bool } (n = 0)\)
using div-exp eq [of 1 1] by (cases n) simp-all

lemma even-succ-div-exp [simp]:
\((1 + a) \div 2 \times n = a \div 2 \times n\) if \(\text{even } a\) and \(n > 0\)
proof (cases n)
  case 0
  with that show ?thesis
  by simp
next
  case (Suc n)
  with \(\text{even } a\) have \((1 + a) \div 2 \times \text{Suc } n = a \div 2 \times \text{Suc } n\)
  proof (induction n)
    case 0
    then show ?case
    by simp
  next
    case (Suc n)
    then show ?case
    using div-exp eq [of - 1 \(\text{Suc } n\), symmetric]
le lemma even-succ-mod-exp [simp]:
⟨(1 + a) mod 2 ^ n = 1 + (a mod 2 ^ n)⟩ if (even a) and (n > 0)
using div-mult-mod-eq [of (1 + a) (2 ^ n)] that
apply simp
by (metis local.add.left-commute local.add-left-cancel local.div-mult-mod-eq)

lemma bits-mod-by-1 [simp]:
⟨a mod 1 = 0⟩
using div-mult-mod-eq [of a 1] by simp

lemma bits-mod-0 [simp]:
⟨0 mod a = 0⟩
using div-mult-mod-eq [of 0 a] by simp

lemma bits-one-mod-two-eq-one [simp]:
⟨1 mod 2 = 1⟩
by (simp add: mod2-eq-if)

definition bit :: 'a ⇒ nat ⇒ bool
where ⟨bit a n ←→ odd (a div 2 ^ n)⟩:

lemma bit-0 [simp]:
⟨bit a 0 ←→ odd a⟩
by (simp add: bit-def)

lemma bit-Suc:
⟨bit a (Suc n) ←→ bit (a div 2) n⟩
using div-exp-eq [of a 1 n] by (simp add: bit-def)

lemma bit-rec:
⟨bit a n ←→ (if n = 0 then odd a else bit (a div 2) (n - 1))⟩
by (cases n) (simp-all add: bit-Suc)

lemma bit-0-eq [simp]:
⟨bit 0 = bot⟩
by (simp add: fun-eq-iff bit-def)

context
fixes a
assumes stable: ⟨a div 2 = a⟩
begin

lemma bits-stable-imp-add-self:
THEORY "Parity"

⟨a + a mod 2 = 0⟩
proof –
  have ⟨a div 2 * 2 + a mod 2 = a⟩
    by (fact div-mul-mod-eq)
  then have ⟨a * 2 + a mod 2 = a⟩
    by (simp add: stable)
  then show ?thesis
    by (simp add: mult-2-right ac-simps)
qed

lemma stable-imp-bit-iff-odd:
  ⟨bit a n ←→ odd a⟩
by (induction n) (simp-all add: stable bit-Suc)
end

lemma bit-iff-idd-imp-stable:
  ⟨a div 2 = a⟩ if ⟨∀ n. bit a n ←→ odd a⟩
using that proof (induction a rule: bits-induct)
  case (stable a)
  then show ?case
    by simp
  next
  case (rec a b)
  from rec.plems [of 1] have [simp]: ⟨b = odd a⟩
    by (simp add: rec.hyps bit-Suc)
  from rec.hyps have hyp: ⟨(of-bool (odd a) + 2 * a) div 2 = a⟩
    by simp
  have ⟨bit a n ←→ odd a⟩ for n
    using rec.plems [of (Suc n)] by (simp add: hyp bit-Suc)
  then have ⟨a div 2 = a⟩
    by (rule rec.IH)
  then have ⟨of-bool (odd a) + 2 * a = 2 * (a div 2) + of-bool (odd a)⟩
    by (simp add: ac-simps)
  also have ⟨... = a⟩
    using mult-div-mod-eq [of 2 a]
    by (simp add: of-bool-odd-eq-mod-2)
  finally show ?case
    using ⟨a div 2 = a⟩ by (simp add: hyp)
qed

lemma exp-eq-0-imp-not-bit:
  ⟨¬ bit a n⟩ if ⟨2 ^ n = 0⟩
using that by (simp add: bit-def)

lemma bit-eqI:
  ⟨a = b⟩ if ⟨∀ n. 2 ^ n ≠ 0 ⇒ bit a n ←→ bit b n⟩
proof –
  have ⟨bit a n ←→ bit b n⟩ for n
proof (cases $2 ^ n = 0$)
  case True
  then show ?thesis
    by (simp add: exp-eq-0-imp-not-bit)
next
  case False
  then show ?thesis
    by (rule that)
qed
then show ?thesis proof
  (induction $a$ arbitrary; $b$ rule: bits-induct)
  case (stable $a$)
    from stable (of $0$) have **: even $b$ $\leftrightarrow$ even $a$
      by simp
    have $b \div 2 = b$
      proof
        (rule bit-iff-idd-imp-stable)
        fix $n$
        from stable have *: $\langle$bit $b$ $n$ $\leftrightarrow$ bit $a$ $n$\rangle
          by simp
        also have $\langle$bit $a$ $n$ $\leftrightarrow$ odd $a$\rangle
          using stable by (simp add: stable-imp-bit-iff-odd)
        finally show $\langle$bit $b$ $n$ $\leftrightarrow$ odd $b$\rangle
          by (simp add: **)
      qed
    from ** have $\langle$even $b$ $\leftrightarrow$ even $a$\rangle
      by simp
    then have $\langle$even $b$ $\leftrightarrow$ even $a$\rangle
      by simp
    then have $2 \ast (a + b) = b + b \mod 2 + (a + b)$
      by simp
    then have $2 \ast (a + (a + b) + b + b \mod 2 + a)$
      (simp add: ac-simps)
    with $\langle$even $b$ $\leftrightarrow$ even $a$\rangle show ?case
      by (simp add: bits-stable-imp-add-self)
next
  case (rec $a$ $p$)
    from rec.prems [of $0$] have [simp]: $\langle$p $\leftrightarrow$ odd $b$\rangle
      by simp
    from rec.hyps have $\langle$bit $a$ $n$ $\leftrightarrow$ bit $(b \div 2)$ $n\rangle$ for $n$
      using rec.prems [of $(\ Suc$ $n)$] by (simp add: bit-Suc)
    then have $\langle$a $\leftrightarrow$ $b$ $\div 2$\rangle
      by (rule rec.IH)
    then have $\langle$even $b$ $\leftrightarrow$ even $a$\rangle
      by simp
    then have $\langle$even $b$ $\leftrightarrow$ even $a$\rangle
      by simp
    also have $\langle$even $b$ $\leftrightarrow$ even $a$\rangle
      by simp
    finally show ?case
      by (simp add: mod2-eq-if)
lemma bit-eq-iff:
\( (a = b \iff (\forall n. \text{bit } a n \leftrightarrow \text{bit } b n) ) \)
by (auto intro: bit-eqI)

lemma bit-exp-iff:
\( (\text{bit } (2^n m) n \leftrightarrow 2^m \neq 0 \land m = n) \)
by (auto simp add: bit-def exp-div-exp-eq)

lemma bit-1-iff:
\( (\text{bit } 1 n \leftrightarrow 1 \neq 0 \land n = 0) \)
using bit-exp-iff[of 0 n] by simp

lemma bit-2-iff:
\( (\text{bit } 2 n \leftrightarrow 2 \neq 0 \land n = 1) \)
using bit-exp-iff[of 1 n] by auto

lemma even-bit-succ-iff:
\( (\text{bit } (1 + a) n \leftrightarrow \text{bit } a n \lor n = 0) \) if \( \langle \text{even } a \rangle \)
using that by (cases \( n = 0 \)) (simp-all add: bit-def)

lemma odd-bit-iff-bit-pred:
\( (\text{bit } a n \leftrightarrow \text{bit } (a - 1) n \lor n = 0) \) if \( \langle \text{odd } a \rangle \)
proof -
from \( \langle \text{odd } a \rangle \) obtain \( b \) where \( \langle a = 2 * b + 1 \rangle \) . .
moreover have \( (\text{bit } (2 * b) n \lor n = 0 \leftrightarrow \text{bit } (1 + 2 * b) n) \)
using even-bit-succ-iff by simp
ultimately show \( \text{thesis} \) by (simp add: ac-simps)
qed

lemma bit-double-iff:
\( (\text{bit } (2 * a) n \leftrightarrow \text{bit } a (n - 1) \land n \neq 0 \land 2^n \neq 0) \)
using even-mult-exp-div-exp-iff[of a 1 n]
by (cases \( n \), auto simp add: bit-def ac-simps)

lemma bit-eq-rec:
\( (a = b \leftrightarrow (\text{even } a \leftrightarrow \text{even } b) \land a \div 2 = b \div 2) \) (is \( \langle ?P = ?Q \rangle \))
proof
assume \( \langle ?P \rangle \)
then show \( \langle ?Q \rangle \)
  by simp
next
assume \( \langle ?Q \rangle \)
then have \( (\text{even } a \leftrightarrow \text{even } b) \) and \( (a \div 2 = b \div 2) \)
  by simp-all
show \( \langle ?P \rangle \)
proof (rule bit-eqI)
  fix \( n \)
  show \( (\text{bit } a n \leftrightarrow \text{bit } b n) \)
proof (cases n)
  case 0
  with even a ⟷ even b show ?thesis
    by simp
next
  case (Suc n)
  moreover from a div 2 = b div 2 have bit (a div 2) n = bit (b div 2) n
    by simp
  ultimately show ?thesis
    by (simp add: bit-Suc)
qed
qed

lemma bit-mask-iff:
  (bit (2 ^ m - 1) n ⟷ 2 ^ n ≠ 0 ∧ n < m)
  by (simp add: bit-def even-mask-div-iff not-le)
end

lemma nat-bit-induct [case-names zero even odd]:
  P n if zero: P 0
  and even: ∀n. P n ⟹ n > 0 ⟹ P (2 * n)
  and odd: ∀n. P n ⟹ P (Suc (2 * n))
proof (induction n rule: less-induct)
  case (less n)
  show P n
    proof (cases n = 0)
      case True with zero show ?thesis by simp
    next
      case False
      with less have hyp: P (n div 2) by simp
      show ?thesis
        proof (cases even n)
          case True
          then have n ≠ 1
            by auto
          with (n ≠ 0) have n div 2 > 0
            by simp
          with (even n) hyp even [of n div 2] show ?thesis
            by simp
        next
          case False
          with hyp odd [of n div 2] show ?thesis
            by simp
        qed
    qed
  qed
qed
instance nat :: semiring-bits
proof
  show \( \langle P \rangle \) if \( \text{stable} \)
    \( \langle \forall n \cdot n \div 2 = n \Longrightarrow P \rangle \)
    and rec: \( \langle \forall n b \cdot P n \Longrightarrow (\text{of-bool} b + 2 \ast n) \div 2 = n \Longrightarrow P (\text{of-bool} b + 2 * n) \rangle \)
      for \( P \) and \( n :: \text{nat} \)
proof (induction \( n \) rule: nat-bit-induct)
  case zero
  from \( \text{stable} \) \[0\] show \( ? \) case
    by simp
next
  case (even \( n \))
  with rec \[\text{of} \text{False}\] show \( ? \) case
    by simp
next
  case (odd \( n \))
  with rec \[\text{of} \text{True}\] show \( ? \) case
    by simp
qed

show \( \langle \langle q \mod 2 \ast m \mod 2 \ast n = q \mod 2 \ast \min m n \rangle \rangle \)
  for \( q \) \( \langle m \rangle \) \( \text{nat} \)
apply (auto simp add: \( \langle \langle m \rangle \leq \langle n \rangle \rangle \))
done

show \( \langle (q \ast 2 \ast m) \mod (2 \ast n) = (q \mod 2 \ast (n - m)) \ast 2 \ast m \rangle \) if \( \langle \langle m \rangle \leq \langle n \rangle \rangle \)
  for \( q \) \( \langle m \rangle \) \( \text{nat} \)
using that
apply (auto simp add: \( \langle \langle \text{div-mod-cancel} \rangle \rangle \) \( \langle \langle \text{div-mul2-eq} \rangle \rangle \) \( \langle \langle \text{mod-add-cancel} \rangle \rangle \) \( \langle \langle \text{split} \rangle \rangle \) \( \langle \langle \text{split-min} \rangle \rangle \))
done

show \( \langle \langle \text{even} ((2 \ast m - (1::\text{nat})) \div 2 \ast n) \leftrightarrow 2 \ast n = (0::\text{nat}) \vee m \leq n \rangle \rangle \)
  for \( m \) \( \langle n \rangle \) \( \text{nat} \)
using \( \langle \langle \text{even-mask-div-iff} \rangle \rangle \) \[\text{where} \; \langle \langle a = \text{nat} \rangle \rangle \; \langle \langle \text{of} \langle m \rangle \rangle \rangle \] by simp

show \( \langle \langle \text{even} (q \ast 2 \ast m \div 2 \ast n) \leftrightarrow n < m \vee \langle (2::\text{nat}) \ast n = 0 \rangle \vee m \leq n \rangle \ast n - m \rangle \rangle \)
  for \( m \) \( \langle n \rangle \) \( \langle r \rangle \) \( \text{nat} \)
apply (auto simp add: \( \langle \langle \text{not-less} \rangle \rangle \) \( \langle \langle \text{power-add} \rangle \rangle \) \( \langle \langle \text{ac-simps} \rangle \rangle \) \( \langle \langle \text{dest!} \rangle \rangle \) \( \langle \langle \text{le-Suc-ex} \rangle \rangle \))
apply (metis \( \langle \langle \text{full-types} \rangle \rangle \) dvd-mul dvd-mul-imp dvd-power-iff-le \( \langle \langle \text{not-less} \rangle \rangle \) \( \langle \langle \text{order-refl} \rangle \rangle \) dvd-power-Suc)
done
qed (auto simp add: \( \langle \langle \text{div-mul2-eq} \rangle \rangle \) \( \langle \langle \text{mod-mul2-eq} \rangle \rangle \) \( \langle \langle \text{power-add} \rangle \rangle \) \( \langle \langle \text{power-diff} \rangle \rangle \))

lemma int-bit-induct [case-names zero minus even odd]:
\( P \) \( \langle k \rangle \) if \( \langle \text{zero-int} \rangle \)
  \( \langle \langle P \rangle \rangle \)
and minus-int: \( \langle \langle - 1 \rangle \rangle \) \( \langle \langle \text{P} \rangle \rangle \)
and even-int: \( \langle \langle P \rangle \rangle \) \( \langle \langle k \neq 0 \rangle \rangle \) \( \langle \langle P \rangle \rangle \)
and odd-int: \( \langle \langle P \rangle \rangle \) \( \langle \langle k \neq - 1 \rangle \rangle \) \( \langle \langle P \rangle \rangle \)
for \( k :: \text{int} \)
proof (cases $k \geq 0$)
case True
define $n$ where $n = \text{nat } k$
with True have $k = \text{int } n$
  by simp
then show $P k$
proof (induction $n$ arbitrary; $k$ rule: nat-bit-induct)
case zero
  then show ?case
    by (simp add: zero-int)
next
case (even $n$)
  have $P (\text{int } n \cdot 2)$
    by (rule even-int) (use even in simp-all)
  with even show ?case
    by (simp add: ac-simps)
next
case (odd $n$)
  have $P (1 + (\text{int } n \cdot 2))$
    by (rule odd-int) (use odd in simp-all)
  with odd show ?case
    by (simp add: ac-simps)
qed
next
case False
define $n$ where $n = \text{nat } (- k - 1)$
with False have $k = - \text{int } n - 1$
  by simp
then show $P k$
proof (induction $n$ arbitrary; $k$ rule: nat-bit-induct)
case zero
  then show ?case
    by (simp add: minus-int)
next
case (even $n$)
  have $P (1 + (- \text{int } (\text{Suc } n) \cdot 2))$
    by (rule add-int) (use even in simp-all add: algebra-simps)
  also have $\ldots = - \text{int } (2 \cdot n) - 1$
    by (simp add: algebra-simps)
  finally show ?case
    using even by simp
next
case (odd $n$)
  have $P (- \text{int } (\text{Suc } n) \cdot 2)$
    by (rule even-int) (use odd in simp-all add: algebra-simps)
  also have $\ldots = - \text{int } (\text{Suc } (2 \cdot n)) - 1$
    by (simp add: algebra-simps)
  finally show ?case
    using odd by simp
instance $\text{int} :: \text{semiring-bits}$
proof
show $(P\ k) \text{ if stable: } (\forall k. \text{div } 2 = k \implies P\ k)$
and rec: $(\forall k b. \ P\ k \implies (\text{of-bool } b + 2 * k) \text{ div } 2 = k \implies P\ (\text{of-bool } b + 2 * k))$)
for $P$ and $k :: \text{int}$
proof (induction $k$ rule: int-bit-induct)
  case zero
  from stable [of 0] show ?case
  by simp
next
  case minus
  from stable [of $(-1)$] show ?case
  by simp
next
  case (even $k$)
  with rec [of $k$ False] show ?case
  by (simp add: ac-simps)
next
  case (odd $k$)
  with rec [of $k$ True] show ?case
  by (simp add: ac-simps)
qed
show $(2 :: \text{int}) ^ m \text{ div } 2 ^ n = \text{of-bool } ((2 :: \text{int}) ^ m \neq 0 \land n \leq m) * 2 ^ (m - n))$
for $m n :: \text{nat}$
proof (cases $(m < n)$)
  case True
  then have $(n = m + (n - m))$
  by simp
  then have $(2 :: \text{int}) ^ m \text{ div } 2 ^ n = (2 :: \text{int}) ^ m \text{ div } 2 ^ (m + (n - m))$
  by simp
  also have $(\ldots = (2 :: \text{int}) ^ m \text{ div } (2 ^ m * 2 ^ (n - m)))$
  by (simp add: power-add)
  also have $(\ldots = (2 :: \text{int}) ^ m \text{ div } 2 ^ (m + (n - m)))$
  by (simp add: zdiv-zmult2-eq)
  finally show ?thesis using $(m < n)$ by simp
next
  case False
  then show ?thesis
  by (simp add: power-diff)
qed
show $(k \text{ mod } 2 ^ m \text{ mod } 2 ^ n = k \text{ mod } 2 ^ \text{min } m n)$
for $m n :: \text{nat}$ and $k :: \text{int}$
using mod-exp-eq [of $(\text{nat } k)\ m\ n$]
apply (auto simp add: mod-mod-cancel zdiv-zmult2-eq power-add zmod-zmult2-eq)
THEORY "Parity" 1051

le_iff_add split: split-min-lin
  apply (auto simp add: less_iff_Suc_add mod_mod_cancel power_add)
  apply (simp only: flip mult.left_commute [of \( 2 ^ m \)])
  apply (subst zmod_zmult2_eq) apply simp-all
  done

show \( (k \cdot 2 ^ m) \mod (2 ^ n) = (k \mod 2 ^ (n - m)) \cdot 2 ^ m \)
  if \( m \leq n \) for \( m \) \( n \) :: nat and \( k :: int \)
  using that
  apply (auto simp add: power_add zmod_zmult2_eq le_iff_add split:
    split-min-lin)
  apply (simp add: ac_simps)
  done

show \( \text{even } ((2 ^ m - (1 :: int)) \div 2 ^ n) \iff 2 ^ n = (0 :: int) \lor m \leq n \)
  for \( m \) \( n \) :: nat
  using even_mask_div_iff [where ?'a = int, of \( m \) \( n \)] by simp
  apply (auto simp add: not_less power_add ac_simps dest: le_Suc_ex)
  done

qed (auto simp add: zdiv_zmult2_eq zmod_zmult2_eq power_add power_diff not_le)

class semiring_bit_shifts = semiring_bits +
fixes push_bit :: \( \langle \text{nat} \Rightarrow 'a \Rightarrow 'a \rangle \)
  assumes push_bit_eq_mult: \( \langle \text{push-bit } n \ a = a \cdot 2 ^ n \rangle \)
fixes drop_bit :: \( \langle \text{nat} \Rightarrow 'a \Rightarrow 'a \rangle \)
  assumes drop_bit_eq_div: \( \langle \text{drop-bit } n \ a = a \div 2 ^ n \rangle \)
begin

definition take_bit :: \( \langle \text{nat} \Rightarrow 'a \Rightarrow 'a \rangle \)
  where take_bit_eq_mod: \( \langle \text{take-bit } n \ a = a \mod 2 ^ n \rangle \)

Logically, \text{push-bit}, \text{drop-bit} and \text{take-bit} are just aliases; having them as separate operations makes proofs easier, otherwise proof automation would fiddle with concrete expressions \( (2 :: 'a)^n \) in a way obfuscating the basic algebraic relationships between those operations. Having \text{push-bit} and \text{drop-bit} as definitional class operations takes into account that specific instances of these can be implemented differently wrt. code generation.

lemma bit_iff_odd_drop_bit:
  \( \langle \text{bit } a \ n \longleftrightarrow \text{odd } (\text{drop-bit } n \ a) \rangle \)
  by (simp add: bit_def drop_bit_eq_div)

lemma even_drop_bit_iff_not_bit:
  \( \langle \text{even } (\text{drop-bit } n \ a) \longleftrightarrow \neg \text{bit } a \ n \rangle \)
  by (simp add: bit_iff_odd_drop_bit)

lemma div_push_bit_of_1_eq_drop_bit:
  \( \langle a \div \text{push-bit } n \ 1 = \text{drop-bit } n \ a \rangle \)
  by (simp add: push_bit_eq_mult drop_bit_eq_div)
lemma bits-ident:
push-bit n (drop-bit n a) + take-bit n a = a
using div-mul-mod-eq by (simp add: push-bit-eq-mult take-bit-eq-mod drop-bit-eq-div)

lemma push-bit-push-bit [simp]:
push-bit m (push-bit n a) = push-bit (m + n) a
by (simp add: push-bit-eq-mult power-add ac-simps)

lemma push-bit-0-id [simp]:
push-bit 0 = id
by (simp add: fun-eq-iff push-bit-eq-mult)

lemma push-bit-of-0 [simp]:
push-bit n 0 = 0
by (simp add: push-bit-eq-mult)

lemma push-bit-of-1:
push-bit n 1 = 2 \^ n
by (simp add: push-bit-eq-mult)

lemma push-bit-Suc [simp]:
push-bit (Suc n) a = push-bit n (a \ast 2)
by (simp add: push-bit-eq-mult ac-simps)

lemma push-bit-double:
push-bit n (a \ast 2) = push-bit n a \ast 2
by (simp add: push-bit-eq-mult ac-simps)

lemma push-bit-add:
push-bit n (a + b) = push-bit n a + push-bit n b
by (simp add: push-bit-eq-mult algebra-simps)

lemma take-bit-0 [simp]:
take-bit 0 a = 0
by (simp add: take-bit-eq-mod)

lemma take-bit-Suc:
\langle take-bit (Suc n) a = take-bit n (a div 2) \ast 2 + of-bool (odd a) \rangle
proof -
  have \langle take-bit (Suc n) (a div 2 \ast 2 + of-bool (odd a)) = take-bit n (a div 2) \ast 2 + of-bool (odd a) \rangle
  using even-succ-mod-exp [of \langle 2 \ast (a div 2) \rangle \langle Suc n \rangle]
  mult-exp-mod-exp-eq [of 1 \langle Suc n \rangle \langle a div 2 \rangle]
  by (auto simp add: take-bit-eq-mod ac-simps)
  then show \langle thesis \rangle
  using div-mul-mod-eq [of a 2] by (simp add: mod-2-eq-odd)
qed
lemma take-bit-rec:
\[ \text{take-bit } n \ a = \begin{cases} 0 & \text{if } n = 0 \ \text{then} \ 0 \ \text{else} \ \text{take-bit} \ (n - 1) \ (a \ \text{div} \ 2) * 2 + \text{of-bool} \ (\text{add} \ a) \end{cases} \]
by (cases \ n) (simp-all add: take-bit-Suc)

lemma take-bit-of-0 [simp]:
take-bit 0 a = 0
by (simp add: take-bit-eq-mod)

lemma take-bit-of-1 [simp]:
take-bit 1 a = of-bool (n > 0)
by (cases \ n) (simp-all add: take-bit-Suc)

lemma drop-bit-of-0 [simp]:
drop-bit 0 a = 0
by (simp add: drop-bit-eq-div)

lemma drop-bit-of-1 [simp]:
drop-bit 1 a = of-bool (n = 0)
by (simp add: drop-bit-eq-div)

lemma drop-bit-0 [simp]:
drop-bit 0 = \text{id}
by (simp add: fun-eq-iff drop-bit-eq-div)

lemma drop-bit-Suc:
drop-bit (Suc \ n) a = drop-bit n (a \ \text{div} \ 2)
using \ div-exp-eq \ [of \ a \ 1] \ by (simp add: drop-bit-eq-div)

lemma drop-bit-rec:
drop-bit n a = \begin{cases} \text{if } n = 0 \ \text{then} \ 0 \ \text{else} \ \text{drop-bit} \ (n - 1) \ (a \ \text{div} \ 2) \end{cases}
by (cases \ n) (simp-all add: drop-bit-Suc)

lemma drop-bit-half:
drop-bit n (a \ \text{div} \ 2) = drop-bit n \ a \ \text{div} \ 2
by (induction \ n \ \text{arbitrary:} \ a) \ (simp-all add: drop-bit-Suc)

lemma drop-bit-of-bool [simp]:
drop-bit \ (\text{of-bool} \ b) = \text{of-bool} \ (n = 0 \ \& \ b)
by (cases \ n) \ \text{simp-all}

lemma take-bit-eq-0-imp-dvd:
take-bit n a = 0 \ \Rightarrow \ 2 ^ n \ \text{dvd} \ a
by (simp add: take-bit-eq-mod mod-0-imp-dvd)

lemma even-take-bit-eq [simp]:
even \ (\text{take-bit} \ n \ a) \iff \ n = 0 \ \lor \ \text{even} \ a
by (simp add: take-bit-rec \ [of \ n \ a])
lemma take-bit-take-bit [simp]:
  take-bit m (take-bit n a) = take-bit (min m n) a
by (simp add: take-bit-eq-mod mod-exp-eq ac-simps)

lemma drop-bit-drop-bit [simp]:
  drop-bit m (drop-bit n a) = drop-bit (m + n) a
by (simp add: drop-bit-eq-div power-add div-exp-eq ac-simps)

lemma push-bit-take-bit:
  push-bit m (take-bit n a) = take-bit (m + n) (push-bit m a)
apply (simp add: push-bit-eq-mult take-bit-eq-mod power-add ac-simps)
using mult-exp-mod-exp-eq [of m (m + n) a] apply (simp add: ac-simps power-add)
done

lemma take-bit-push-bit:
  take-bit m (push-bit n a) = push-bit n (take-bit (m - n) a)
proof (cases m ≤ n)
case True
  then show ?thesis
    apply (simp add:)
    apply (simp-all add: push-bit-eq-mult take-bit-eq-mod)
    apply (auto dest: le-Suc-ex simp add: power-add ac-simps)
    using mult-exp-mod-exp-eq [of m (m + n) a] for n
    apply (simp add: ac-simps)
done
next
case False
  then show ?thesis
    using push-bit-take-bit [of n m - n a]
    by simp
qed

lemma take-bit-drop-bit:
  take-bit m (drop-bit n a) = drop-bit n (take-bit (m - n) a)
proof (cases m ≤ n)
case True
  using take-bit-drop-bit [of n m - n a] by simp
next
case False
  then obtain q where ⟨m = n + q⟩
    by (auto simp add: not-le dest: less-imp-Suc-add)
  then have ⟨drop-bit m (take-bit n a) = 0⟩
    using div-exp-eq [of ⟨a mod 2^n⟩ n q]
    by (simp add: take-bit-eq-mod drop-bit-eq-div)
with False show thesis 
  by simp 
qed 

lemma even-push-bit-iff [simp]:
  even (push-bit n a) ←→ n ≠ 0 ∨ even a:
  by (simp add: push-bit-eq-mult) auto 

lemma bit-push-bit-iff:
  bit (push-bit m a) n ←→ n ≥ m ∧ 2 ^ n ≠ 0 ∧ (n < m ∨ bit a (n - m))
  by (auto simp add: bit-def push-bit-eq-mult even-mult-exp-div-exp-iff) 

lemma bit-drop-bit-eq:
  bit (drop-bit n a) = bit a ◦ (+) n
  by (simp add: bit-def fun-eq-iff ac-simps flip: drop-bit-eq-div) 

lemma bit-take-bit-iff:
  bit (take-bit m a) n ←→ n < m ∧ bit a n
  by (simp add: bit-def drop-bit-take-bit not-le flip: drop-bit-eq-div) 

lemma stable-imp-drop-bit-eq:
  drop-bit n a = a
  if ⟨a div 2 = a⟩
  by (induction n) (simp-all add: that drop-bit-Suc) 

lemma stable-imp-take-bit-eq:
  take-bit n a = (if even a then 0 else 2 ^ n - 1)
  if ⟨a div 2 = a⟩
proof (rule bit-eqI)
  fix m
  assume ⟨2 ^ m ≠ 0⟩
  with that show ⟨bit (take-bit n a) m ←→ bit (if even a then 0 else 2 ^ n - 1) m⟩
  by (simp add: bit-take-bit-iff bit-mask-iff stable-imp-bit-iff-odd) 
qed 

end

instantiation nat :: semiring-bit-shifts 
begin 

definition push-bit-nat :: (nat ⇒ nat ⇒ nat)
  where (push-bit-nat n m = m * 2 ^ n) 

definition drop-bit-nat :: (nat ⇒ nat ⇒ nat)
  where (drop-bit-nat n m = m div 2 ^ n) 

instance proof
  show ⟨push-bit n m = m * 2 ^ n⟩ for n m :: nat
by (simp add: push-bit-nat-def)

show \( \langle \text{drop-bit } n \ m = m \div 2 \ ^{\ n} \rangle \) for \( n \ m :: \text{nat} \)
  by (simp add: drop-bit-nat-def)
qed

end

instantiation \( \text{int} :: \text{semiring-bit-shifts} \)
begin

definition push-bit-int :: \( \langle \text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \rangle \)
  where \( \langle \text{push-bit-int } n \ k = k \times 2 \ ^{\ n} \rangle \)

definition drop-bit-int :: \( \langle \text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \rangle \)
  where \( \langle \text{drop-bit-int } n \ k = k \div 2 \ ^{\ n} \rangle \)

instance proof
  show \( \langle \text{push-bit } n \ k = k \times 2 \ ^{\ n} \rangle \) for \( n :: \text{nat} \) and \( k :: \text{int} \)
    by (simp add: push-bit-int-def)
  show \( \langle \text{drop-bit } n \ k = k \div 2 \ ^{\ n} \rangle \) for \( n :: \text{nat} \) and \( k :: \text{int} \)
    by (simp add: drop-bit-int-def)
qed
end

lemma bit-push-bit-iff-nat:
\( \langle \text{bit } (\text{push-bit } m \ q) \ n \longleftrightarrow m \leq n \wedge \text{bit } q \ (n - m) \rangle \) for \( q :: \text{nat} \)
  by (auto simp add: bit-push-bit-iff)

lemma bit-push-bit-iff-int:
\( \langle \text{bit } (\text{push-bit } m \ k) \ n \longleftrightarrow m \leq n \wedge \text{bit } k \ (n - m) \rangle \) for \( k :: \text{int} \)
  by (auto simp add: bit-push-bit-iff)

class \( \text{unique-euclidean-semiring-with-bit-shifts} = \text{unique-euclidean-semiring-with-nat} + \text{semiring-bit-shifts} \)
begin

lemma take-bit-of-exp [simp]:
\( \langle \text{take-bit } m \ (2 \ ^{\ n}) = \text{of-bool } (n < m) \times 2 \ ^{\ n} \rangle \)
    by (simp add: take-bit-eq-mod exp-mod-exp)

lemma take-bit-of-2 [simp]:
\( \langle \text{take-bit } n \ 2 = \text{of-bool } (2 \leq n) \times 2 \rangle \)
    using take-bit-of-exp \[ of \ n \] by simp

lemma take-bit-of-mask:
\( \langle \text{take-bit } m \ (2 \ ^{\ n - 1}) = 2 \ ^{\ \text{min } m \ n - 1} \rangle \)
    by (simp add: take-bit-eq-mod mask-mod-exp)
lemma push-bit-eq-0-iff [simp]:
  push-bit n a = 0 ⟷ a = 0
by (simp add: push-bit-eq-mult)

lemma push-bit-numeral [simp]:
  push-bit (numeral l) (numeral k) = push-bit (pred-numeral l) (numeral (Nat.Bit0 k))
by (simp only: numeral-eq-Suc power-Suc numeral0 [of k] mult-2 [symmetric])
(simp add: ac-simps)

lemma push-bit-of-nat:
  push-bit n (of-nat m) = of-nat (push-bit n m)
by (simp add: push-bit-eq-mult Parity.push-bit-eq-mult)

lemma take-bit-add:
  take-bit n (take-bit n a + take-bit n b) = take-bit n (a + b)
by (simp add: take-bit-eq-mod mod-simps)

lemma take-bit-eq-0-iff:
  take-bit n a = 0 ⟷ 2 ^ n dvd a
by (simp add: take-bit-eq-mod dvd-0-iff-dvd)

lemma take-bit-of-1-eq-0-iff [simp]:
  take-bit n 1 = 0 ⟷ n = 0
by (simp add: take-bit-eq-mod)

lemma take-bit-numeral-bit0 [simp]:
  take-bit (numeral l) (numeral (Nat.Bit0 k)) = take-bit (pred-numeral l) (numeral k) * 2
by (simp only: numeral-eq-Suc power-Suc numeral0 [of k] mult-2 [symmetric] take-bit-Suc
ac-simps even-mult-iff nonzero-mult-div-cancel-right [OF numeral-neq-zero])
simp

lemma take-bit-numeral-bit1 [simp]:
  take-bit (numeral l) (numeral (Nat.Bit1 k)) = take-bit (pred-numeral l) (numeral k) * 2 + 1
by (simp only: numeral-eq-Suc power-Suc numeral1 [of k] mult-2 [symmetric] take-bit-Suc
ac-simps even-add even-mult-iff div-mult-self1 [OF numeral-neq-zero]) (simp add: ac-simps)

lemma take-bit-of-nat:
  take-bit n (of-nat m) = of-nat (take-bit n m)
by (simp add: take-bit-eq-mod Parity.take-bit-eq-mod of-nat-mod [of m 2 ^ n])

lemma drop-bit-numeral-bit0 [simp]:
  drop-bit (numeral l) (numeral (Nat.Bit0 k)) = drop-bit (pred-numeral l) (numeral k)
by (simp only: numeral-eq-Suc power-Suc numeral-Bit0 [of k] mult-2 [symmetric]
   drop-bit-Suc
   nonzero-mult-div-cancel-left [OF numeral-neq-zero])

lemma drop-bit-numeral-bit1 [simp]:
  drop-bit (numeral l) (numeral (Num.Bit1 k)) = drop-bit (pred-numeral l) (numeral k)
by (simp only: numeral-eq-Suc power-Suc numeral-Bit1 [of k] mult-2 [symmetric]
   drop-bit-Suc
   div-mult-self4 [OF numeral-neq-zero]) simp

lemma drop-bit-of-nat:
  drop-bit n (of-nat m) = of-nat (drop-bit n m)
by (simp add: drop-bit-eq-div Parity.drop-bit-eq-div [of m 2 ^ n])

lemma bit-of-nat-iff-bit [simp]:
  ⟨bit (of-nat m) n ⟷ bit m n⟩
proof –
  have ⟨even (m div 2 ^ n) ⟷ even (of-nat (m div 2 ^ n))⟩
    by simp
  also have ⟨of-nat (m div 2 ^ n) = of-nat m div of-nat (2 ^ n)⟩
    by (simp add: of-nat-div)
  finally show ?thesis
    by (simp add: bit-def semiring-bits-class.bit-def)
qed

lemma of-nat-push-bit:
  ⟨of-nat (push-bit m n) = push-bit m (of-nat n)⟩
by (simp add: push-bit-eq-mult semiring-bit-shifts-class.push-bit-eq-mult)

lemma of-nat-drop-bit:
  ⟨of-nat (drop-bit m n) = drop-bit m (of-nat n)⟩
by (simp add: drop-bit-eq-div semiring-bit-shifts-class.drop-bit-eq-div of-nat-div)

lemma of-nat-take-bit:
  ⟨of-nat (take-bit m n) = take-bit m (of-nat n)⟩
by (simp add: take-bit-eq-mod semiring-bit-shifts-class.take-bit-eq-mod of-nat-mod)

lemma bit-push-bit-iff-of-nat-iff:
  ⟨bit (push-bit m (of-nat r)) n ⟷ m ≤ n ∧ bit (of-nat r) (n - m)⟩
by (auto simp add: bit-push-bit-iff)
end

instance nat :: unique-euclidean-semiring-with-bit-shifts..

instance int :: unique-euclidean-semiring-with-bit-shifts..

lemma push-bit-of-Suc-0 [simp]:
THEORY “Parity”

push-bit n (Suc 0) = 2 ^ n
using push-bit-of-1 [where ?'a = nat] by simp

lemma take-bit-of-Suc-0 [simp]:
take-bit n (Suc 0) = of-bool (0 < n)
using take-bit-of-1 [where ?'a = nat] by simp

lemma drop-bit-of-Suc-0 [simp]:
drop-bit n (Suc 0) = of-bool (n = 0)
using drop-bit-of-1 [where ?'a = nat] by simp

lemma take-bit-eq-self:
⟨take-bit n m = m⟩ if ⟨m < 2 ^ n⟩ for n m :: nat
using that by (simp add: take-bit-eq-mod)

lemma push-bit-minus-one:
push-bit n (− 1 :: int) = − (2 ^ n)
by (simp add: push-bit-eq-mult)

lemma minus-1-div-exp-eq-int:
⟨− 1 div (2 :: int) ^ n = − 1⟩
by (induction n) (use div-exp-eq [symmetric, of ⟨− 1 :: int⟩ 1] in ⟨simp-all add: ac-simps⟩)

lemma drop-bit-minus-one [simp]:
⟨drop-bit n (− 1 :: int) = − 1⟩
by (simp add: drop-bit-eq-div minus-1-div-exp-eq-int)

lemma take-bit-minus:
take-bit n (− (take-bit n k)) = take-bit n (− k)
for k :: int
by (simp add: take-bit-eq-mod mod-minus-eq)

lemma take-bit-diff:
take-bit n (take-bit n k − take-bit n l) = take-bit n (k − l)
for k l :: int
by (simp add: take-bit-eq-mod mod-diff-eq)

lemma take-bit-nonnegative [simp]:
take-bit n k ≥ 0
for k :: int
by (simp add: take-bit-eq-mod)

lemma drop-bit-push-bit-int:
⟨drop-bit m (push-bit n k) = drop-bit (m − n) (push-bit (n − m) k)⟩ for k :: int
by (cases m ≤ n) (auto simp add: mult.left-commute [of (2 ^ n)] mult.commute [of - (2 ^ n)] mult.assoc
mult.commute [of k] drop-bit-eq-div push-bit-eq-mult not-le power-add dest!: le-Suc-ex less-imp-Suc-add)
56  More on quotient and remainder

theory Divides
imports Parity
begin

inductive eucl-rel-int :: int ⇒ int ⇒ int × int ⇒ bool
  where
  eucl-rel-int-by0: eucl-rel-int k 0 (0, k)
  | eucl-rel-int-dividesI: l ≠ 0 ⇒ k = q * l ⇒ eucl-rel-int k l (q, 0)
  | eucl-rel-int-remainderI: sgn r = sgn l ⇒ |r| < |l|
    ⇒ k = q * l + r ⇒ eucl-rel-int k l (q, r)

lemma eucl-rel-int-iff:
  eucl-rel-int k l (q, r) ←→ k = l * q + r ∧
  (if 0 < l then 0 ≤ r ∧ r < l else if l < 0 then l < r ∧ r ≤ 0 else q = 0)
  by (cases r = 0)
  (auto elim!: eucl-rel-int_cases intro: eucl-rel-int-by0 eucl-rel-int-dividesI eucl-rel-int-remainderI
  simp add: ac_simps sgn-1-pos sgn-1-neg)

lemma unique-quotient-lemma:
  assumes b * q' + r' ≤ b * q + r 0 ≤ r' r' < b r < b shows q' ≤ (q::int)
  proof –
    have r' + b * (q'−q) ≤ r
      using assms by (simp add: right-diff-distrib)
    moreover have 0 < b * (1 + q − q')
      using assms by (simp add: right-diff-distrib distrib-left)
    moreover have b * q' < b * (1 + q)
      using assms by (simp add: right-diff-distrib distrib-left)
    ultimately show ?thesis
      using assms by (simp add: mult-less-cancel-left)
  qed

lemma unique-quotient-lemma-neg:
  b * q' + r' ≤ b * q + r ⇒ r ≤ 0 ⇒ b < r ⇒ b < r' ⇒ q ≤ (q'::int)
  by (rule_tac b = −b and r = −r' and r' = −r in unique-quotient-lemma) auto

lemma unique-quotient:
  eucl-rel-int a b (q, r) ⇒ eucl-rel-int a b (q', r') ⇒ q = q'
  apply (rule order-antisym)
  apply (simp-all add: eucl-rel-int-iff linorder-neg-iff split: if-split-asm)
  apply (blast intro: order-eq-refl [THEN unique-quotient-lemma] order-eq-refl
  [THEN unique-quotient-lemma-neg] sym)+
  done
lemma unique-remainder:
  \( \text{eucl-rel-int } a \ b \ (q, r) \implies \text{eucl-rel-int } a \ b \ (q', r') \implies r = r' \)
apply (subgoal-tac q = "q")
apply (simp add: eucl-rel-int-iff)
done

lemma eucl-rel-int:
  \( \text{eucl-rel-int } k \ l \ (k \div l, k \mod l) \)
proof (cases k rule: int-cases3)
  case zero
  then show \(?thesis\)
    by (simp add: eucl-rel-int-iff divide-int-def modulo-int-def)
next
  case (pos n)
  then show \(?thesis\)
    using div-mult-mod-eq [of n]
    by (cases l rule: int-cases3)
      (auto simp del: of-nat-mult of-nat-add
        [symmetric] algebra-simps
eucl-rel-int-iff divide-int-def modulo-int-def)
next
  case (neg n)
  then show \(?thesis\)
    using div-mult-mod-eq [of n]
    by (cases l rule: int-cases3)
      (auto simp del: of-nat-mult of-nat-add
        [symmetric] algebra-simps
eucl-rel-int-iff divide-int-def modulo-int-def)
qed

lemma divmod-int-unique:
  assumes \( \text{eucl-rel-int } k \ l \ (q, r) \)
  shows \( \text{div-int-unique: } k \div l = q \text{ \ and } \text{mod-int-unique: } k \mod l = r \)
using assms eucl-rel-int [of k l]
using unique-quotient [of k l] unique-remainder [of k l]
by auto

lemma div-abs-eq-div-nat:
  \(|k| \div |l| = \text{int} \ (\text{nat } |k| \div \text{nat } |l|)\)
by (simp add: divide-int-def)

lemma mod-abs-eq-div-nat:
  \(|k| \mod |l| = \text{int} \ (\text{nat } |k| \mod \text{nat } |l|)\)
by (simp add: modulo-int-def)
lemma `zdiv-int`:
`int (a div b) = int a div int b`
by (simp add: divide-int-def)

lemma `zmod-int`:
`int (a mod b) = int a mod int b`
by (simp add: modulo-int-def)

lemma `div-sgn-abs-cancel`:
fixes `k l v :: int`
assumes `v ≠ 0`
shows `sgn v * |k| div (sgn v * |l|) = |k| div |l|`
proof –
from assms have `sgn v = -1 ∨ sgn v = 1`
  by (cases v ≥ 0) auto
then show ?thesis
  using assms unfolding divide-int-def [of `sgn v * |k| sgn v * |l|`]
  by (fastforce simp add: not-less div-abs-eq-div-nat)
qed

lemma `div-sgn-abs-cancel`:
fixes `k l v :: int`
assumes `sgn k = sgn l`
shows `k div l = |k| div |l|`
proof (cases `l = 0`)
case `True`
then show ?thesis
  by simp
next
case `False`
with assms have `(sgn k * |k|) div (sgn l * |l|) = |k| div |l|`
  using div-sgn-abs-cancel [of l k l] by simp
then show ?thesis
  by (simp add: sgn-mult-abs)
qed

lemma `div-sdv-dsgn-abs`:
fixes `k l :: int`
assumes `l dvd k`
shows `k div l = (sgn k * sgn l) * (|k| div |l|)`
proof (cases `k = 0 ∨ l = 0`)
case `True`
then show ?thesis
  by auto
next
case `False`
than have `k ≠ 0 and l ≠ 0`
  by auto
show ?thesis
proof (cases sgn l = sgn k)
  case True
  then show ?thesis
    by (simp add: div-eq-sgn-abs)
next
  case False
  with ⟨k ≠ 0⟩ ⟨l ≠ 0⟩
  have sgn l * sgn k = - 1
    by (simp add: sgn-if split: if-splits)
  with assms show ?thesis
    unfolding divide-int-def [of k l]
    by (auto simp add: zdiv-int ac-simps)
qed

lemma div-noneq-sgn-abs:
  fixes k l :: int
  assumes l ≠ 0
  assumes sgn k ≠ sgn l
  shows k div l = - (|k| div |l|) - of-bool (∼ l dvd k)
  using assms
  by (simp only: divide-int-def [of k l], auto simp add: not-less zdiv-int)

56.1.1 General Properties of div and mod

lemma div-pos-pos-trivial [simp]:
  k div l = 0 if k ≥ 0 and k < l for k l :: int
  using that
  by (simp add: unique-euclidean-semiring-class.div-eq-0-iff division-segment-int-def)

lemma mod-pos-pos-trivial [simp]:
  k mod l = k if k ≥ 0 and k < l for k l :: int
  using that
  by (simp add: mod-eq-self-iff-div-eq-0)

lemma div-neg-neg-trivial [simp]:
  k div l = 0 if k ≤ 0 and l < k for k l :: int
  using that (cases k = 0)
  by (simp add: unique-euclidean-semiring-class.div-eq-0-iff
              division-segment-int-def)

lemma mod-neg-neg-trivial [simp]:
  k mod l = k if k ≤ 0 and l < k for k l :: int
  using that
  by (simp add: mod-eq-self-iff-div-eq-0)

lemma div-pos-neg-trivial:
  k div l = - 1 if 0 < k and k + l ≤ 0 for k l :: int
  apply (rule div-int-unique [of - - k + l])
  apply (use that in (auto simp add: eucl-rel-int-iff))
  done

lemma mod-pos-neg-trivial:
THEORY "Divides"

$k \mod l = k + l$ if $0 < k$ and $k + l \leq 0$ for $k, l :: \text{int}$

apply (rule mod-int-unique [of - - 1])
apply (use that in (auto simp add: eucl-rel-int-iff))
done

There is neither div-neg-pos-trivial nor mod-neg-pos-trivial because $(0 :: 'a) \div \text{l} = (0 :: 'a)$ would supersede it.

56.1.2 Laws for div and mod with Unary Minus

lemma zdiv-zminus1-eq-if:
  \[ b \neq (0 :: \text{int}) \implies (-a) \div b = (if a \mod b = 0 \text{ then } - (a \div b) \text{ else } - (a \div b) - 1) \]
by (blast intro: eucl-rel-int [THEN zminus1-lemma, THEN div-int-unique])

lemma zmod-zminus1-eq-if:
  \[ (-a :: \text{int}) \mod b = (if a \mod b = 0 \text{ then } 0 \text{ else } b - (a \mod b)) \]
proof (cases b = 0)
case False
  then show \?thesis
  by (blast intro: eucl-rel-int [THEN zminus1-lemma, THEN mod-int-unique])
qed auto

lemma zmod-zminus1-not-zero:
  fixes k l :: \text{int}
  shows \[ k \mod l \neq 0 \implies k \mod l \neq 0 \]
  by (simp add: mod-eq-0-iff-dvd)

lemma zmod-zminus2-not-zero:
  fixes k l :: \text{int}
  shows \[ k \mod - l \neq 0 \implies k \mod l \neq 0 \]
  by (simp add: mod-eq-0-iff-dvd)

lemma zdiv-zminus2-eq-if:
  \[ b \neq (0 :: \text{int}) \implies a \div (-b) = \]
  \[ (if a \mod b = 0 \text{ then } - (a \div b) \text{ else } - (a \div b) - 1) \]
  by (auto simp add: zdiv-zminus1-eq-if div-minus-right)

lemma zmod-zminus2-eq-if:
  \[ a \mod (-b :: \text{int}) = (if a \mod b = 0 \text{ then } 0 \text{ else } (a \mod b) - b) \]
  by (auto simp add: zmod-zminus1-eq-if mod-minus-right)
56.1.3 Monotonicity in the First Argument (Dividend)

lemma zdiv-mono1:
  fixes b::int
  assumes a ≤ a' 0 < b shows a div b ≤ a' div b
  proof (rule unique-quotient-lemma)
    show b * (a div b) + a mod b ≤ b * (a' div b) + a' mod b
    using assms(1) by auto
  qed (use assms in auto)

lemma zdiv-mono1-neg:
  fixes b::int
  assumes a ≤ a' b < 0 shows a' div b ≤ a div b
  proof (rule unique-quotient-lemma-neg)
    show b * (a div b) + a mod b ≤ b * (a' div b) + a' mod b
    using assms(1) by auto
  qed (use assms in auto)

56.1.4 Monotonicity in the Second Argument (Divisor)

lemma q-pos-lemma:
  fixes q::int
  assumes 0 ≤ b'q' + r' r' < b' 0 < b'
  shows 0 ≤ q'
  proof -
    have 0 < b'* (q' + 1)
      using assms by (simp add: distrib-left)
    with assms show ?thesis
      by (simp add: zero-less-mult-iff)
  qed

lemma zdiv-mono2-lemma:
  fixes q::int
  assumes eq: b*q + r = b'*q' + r' and le: 0 ≤ b'*q' + r' and r' < b' 0 ≤ r 0 < b' b' ≤ b
  shows q ≤ q'
  proof -
    have 0 ≤ q'
      using q-pos-lemma le (r' < b' 0 < b') by blast
    moreover have b*q = r' - r + b'*q'
      using eq by linarith
    ultimately have b*q < b* (q' + 1)
      using mult-right-mono assms unfolding distrib-left by fastforce
    with assms show ?thesis
      by (simp add: mult-less-cancel-left-pos)
  qed

lemma zdiv-mono2:
  fixes a::int
  assumes 0 ≤ a 0 < b' b' ≤ b shows a div b ≤ a div b
proof (rule zdiv-mono2-lemma)
  have b ≠ 0
    using assms by linarith
  show b * (a div b) + a mod b = b' * (a div b') + a mod b'
    by simp
qed (use assms in auto)

lemma zdiv-mono2-neg-lemma:
  fixes q'::int
  assumes b*q + r = b'*q' + r' b'*q' + r' < 0 r < b 0 ≤ r' 0 < b' b' ≤ b
  shows q' ≤ q
proof
  have b'*q' < 0
    using assms by linarith
  with assms have q' ≤ 0
    by (simp add: mult-less-0-iff)
  have b*q' ≤ b'*q'
    by (simp add: (q' ≤ 0) assms (6) mult-right-mono-neg)
  then have b*q' < b' (q + 1)
    using assms by (simp add: distrib-left)
  then show ?thesis
    using assms by (simp add: mult-less-cancel-left)
qed

lemma zdiv-mono2-neg:
  fixes a::int
  assumes a < 0 0 < b' b' ≤ b
  shows a div b' ≤ a div b
proof (rule zdiv-mono2-neg-lemma)
  have b ≠ 0
    using assms by linarith
  show b * (a div b) + a mod b = b' * (a div b') + a mod b'
    by simp
qed (use assms in auto)

lemma div-pos-geq:
  fixes k l :: int
  assumes 0 < l and l ≤ k
  shows k div l = (k - l) div l + 1
proof
  have k = (k - l) + l ..
  then obtain j where k: k = j + l ..
  with assms show ?thesis by (simp add: div-add-self2)
qed

lemma mod-pos-geq:
  fixes k l :: int
  assumes 0 < l and l ≤ k
  shows k mod l = (k - l) mod l
proof
have \( k = (k - l) + l \) by simp
then obtain \( j \) where \( k = j + l \)
with assms show \(?thesis\) by simp
qed

56.1.5 Splitting Rules for \texttt{div} and \texttt{mod}

The proofs of the two lemmas below are essentially identical

lemma \texttt{split-pos-lemma}:
\[
\text{\(0 < k \implies P(n \div k :: \text{int})(n \mod k) = (\forall i \ j. \ 0 \leq j \land j < k \land n = k \times i + j \implies P \ i \ j)\)}
\]
by auto

lemma \texttt{split-neg-lemma}:
\[
\text{\(k < 0 \implies P(n \div k :: \text{int})(n \mod k) = (\forall i \ j. \ k < j \land j \leq 0 \land n = k \times i + j \implies P \ i \ j)\)}
\]
by auto

lemma \texttt{split-zdiv}:
\[
P(n \div k :: \text{int}) =
(\neg k = 0 \implies P 0) \land
(0 < k \implies (\forall i \ j. \ 0 \leq j \land j < k \land n = k \times i + j \implies P \ i \ j)) \land
(k < 0 \implies (\forall i \ j. \ k < j \land j \leq 0 \land n = k \times i + j \implies P \ i \ j))
\]
proof (cases \( k = 0 \))
  case False
  then show \(?thesis\)
  unfolding linorder-neq_iff
  by (auto simp add: split-pos-lemma [of concl: \(\lambda x \ y. \ P \ x\)] split-neg-lemma [of concl: \(\lambda x \ y. \ P \ x\)])
qed auto

lemma \texttt{split-zmod}:
\[
P(n \mod k :: \text{int}) =
(\neg k = 0 \implies P n) \land
(0 < k \implies (\forall i \ j. \ 0 \leq j \land j < k \land n = k \times i + j \implies P \ j)) \land
(k < 0 \implies (\forall i \ j. \ k < j \land j \leq 0 \land n = k \times i + j \implies P \ j))
\]
proof (cases \( k = 0 \))
  case False
  then show \(?thesis\)
  unfolding linorder-neq_iff
  by (auto simp add: split-pos-lemma [of concl: \(\lambda x \ y. \ P \ y\)] split-neg-lemma [of concl: \(\lambda x \ y. \ P \ y\)])
qed auto

Enable (lin)arith to deal with (\texttt{div}) and (\texttt{mod}) when these are applied to some constant that is of the form \texttt{numeral} \( k \):

declare \texttt{split-zdiv} [of - - numeral \( k \), arith-split] for \( k \)
declare \texttt{split-zmod} [of - - numeral \( k \), arith-split] for \( k \)
56.1.6 Computing \textit{div} and \textit{mod} with shifting

\textbf{lemma} \textit{pos-eucl-rel-int-mult-2}:
\begin{itemize}
  \item assumes $0 \leq b$
  \item assumes \textit{eucl-rel-int} $a \ b \ (q, \ r)$
  \item shows \textit{eucl-rel-int} $(1 + 2 \ast a) \ (2 \ast b) \ (q, \ 1 + 2 \ast r)$
\end{itemize}
\textit{using} \textit{assms unfolding eucl-rel-int-iff} \textbf{by} \textit{auto}

\textbf{lemma} \textit{neg-eucl-rel-int-mult-2}:
\begin{itemize}
  \item assumes $b \leq 0$
  \item assumes \textit{eucl-rel-int} $(a + 1) \ b \ (q, \ r)$
  \item shows \textit{eucl-rel-int} $(1 + 2 \ast a) \ (2 \ast b) \ (q, \ 2 \ast r - 1)$
\end{itemize}
\textit{using} \textit{assms unfolding eucl-rel-int-iff} \textbf{by} \textit{auto}

\textbf{computing div by shifting}
\textbf{lemma} \textit{pos-zdiv-mult-2}:
\begin{itemize}
  \item \textit{fixes} $a \ b :: \textit{int}$
  \item assumes $0 \leq a$
  \item shows $(1 + 2 \ast b) \ \text{div} \ (2 \ast a) = b \ \text{div} \ a$
\end{itemize}
\textit{using} \textit{pos-eucl-rel-int-mult-2} \textbf{[OF - eucl-rel-int]}
\textbf{by} \textit{(rule div-int-unique)}

\textbf{lemma} \textit{neg-zdiv-mult-2}:
\begin{itemize}
  \item \textit{fixes} $A : a \leq (0::\textit{int})$
  \item shows $(1 + 2 \ast b) \ \text{div} \ (2 \ast a) = (b+1) \ \text{div} \ a$
\end{itemize}
\textit{using} \textit{neg-eucl-rel-int-mult-2} \textbf{[OF \ A eucl-rel-int]}
\textbf{by} \textit{(rule div-int-unique)}

\textbf{lemma} \textit{zdiv-numeral-Bit0} \textbf{[simp]}:
\begin{itemize}
  \item \textit{numeral} $(\textit{Num.Bit0} \ v) \ \text{div} \ \textit{numeral} \ (\textit{Num.Bit0} \ w) =$
  \item \textit{unfolding} \textit{numeral.simps}
  \item \textit{unfolding} \textit{mult-2} \textbf{[symmetric]}
\end{itemize}
\textbf{by} \textit{(rule div-mult-mult1, simp)}

\textbf{lemma} \textit{zdiv-numeral-Bit1} \textbf{[simp]}:
\begin{itemize}
  \item \textit{numeral} $(\textit{Num.Bit1} \ v) \ \text{div} \ \textit{numeral} \ (\textit{Num.Bit0} \ w) =$
  \item \textit{unfolding} \textit{numeral.simps}
  \item \textit{unfolding} \textit{mult-2} \textbf{[symmetric]} \textit{add.commute} \textbf{[of - 1]}
\end{itemize}
\textbf{by} \textit{(rule pos-zdiv-mult-2, simp)}

\textbf{lemma} \textit{pos-zmod-mult-2}:
\begin{itemize}
  \item \textit{fixes} $a \ b :: \textit{int}$
  \item assumes $0 \leq a$
  \item shows $(1 + 2 \ast b) \ \text{mod} \ (2 \ast a) = 1 + 2 \ast (b \ \text{mod} \ a)$
\end{itemize}
\textit{using} \textit{pos-eucl-rel-int-mult-2} \textbf{[OF \ assms eucl-rel-int]}
\textbf{by} \textit{(rule mod-int-unique)}

\textbf{lemma} \textit{neg-zmod-mult-2}:
\begin{itemize}
  \item \textit{fixes} $a \ b :: \textit{int}$
  \item assumes $a \leq 0$
  \item shows $(1 + 2 \ast b) \ \text{mod} \ (2 \ast a) = 2 \ast ((b + 1) \ \text{mod} \ a) - 1$
\end{itemize}
\textit{using} \textit{neg-eucl-rel-int-mult-2} \textbf{[OF \ assms eucl-rel-int]}
\textbf{by} \textit{(rule mod-int-unique)}
lemma zmod-numeral-Bit0 [simp]:
numeral (Num.Bit0 v) mod numeral (Num.Bit0 w) =
(2::int) * (numeral v mod numeral w)
unfolding numeral-Bit0 [of v] numeral-Bit0 [of w]
unfolding mult-2 [symmetric] by (rule mod-mul-mult1)

lemma zmod-numeral-Bit1 [simp]:
numeral (Num.Bit1 v) mod numeral (Num.Bit0 w) =
2 * (numeral v mod numeral w) + (1::int)
unfolding numeral-Bit1 [of v] numeral-Bit0 [of w]
unfolding mult-2 [symmetric] add.commute [of - 1]
by (rule pos-zmod-mult-2, simp)

lemma zdiv-eq-0-iff:
i div k = 0 ←→ k = 0 ∨ 0 ≤ i ∧ i < k ∨ i ≤ 0 ∧ k < i (is ?L = ?R)
for i k :: int
proof
assume ?L
moreover have ?L → ?R
by (rule split-zdiv [THEN iffD2]) simp
ultimately show ?R
by blast
next
assume ?R then show ?L
by auto
qed

lemma zmod-trival-iff:
fixes i k :: int
shows i mod k = i ←→ k = 0 ∨ 0 ≤ i ∧ i < k ∨ i ≤ 0 ∧ k < i
proof
have i mod k = i ←→ i div k = 0
by safe (insert div-mult-mod-eq [of i k], auto)
with zdiv-eq-0-iff
show ?thesis
by simp
qed

56.1.7 Quotients of Signs

lemma div-eq-minus1: 0 < b → -1 div b = -1 for b :: int
by (simp add: divide-int-def)

lemma zmod-minus1: 0 < b → -1 mod b = b - 1 for b :: int
by (auto simp add: modulo-int-def)

lemma div-neg-pos-less0:
fixes a::int

assumes $a < 0$ 0 $< b$
shows $a \div b < 0$

proof –
have $a \div b \leq -1 \div b$
  using zdiv-mono1 assms by auto
also have $... \leq -1$
  by (simp add: assms(2) div-eq-minus1)
finally show $?thesis$
  by force

qed

lemma $\text{div-nonneg-neg-le0}$:  
| ($0 :: \text{int}$) $\leq a$; $b < 0$ |
| $\implies a \div b \leq 0$
by (drule zdiv-mono1-neg, auto)

lemma $\text{div-nonpos-pos-le0}$:  
| ($a :: \text{int}$) $\leq 0$; $b > 0$ |
| $\implies a \div b \leq 0$
by (drule zdiv-mono1, auto)

Now for some equivalences of the form $a \div b \geq 0 \iff \ldots$ conditional upon the sign of $a$ or $b$. There are many more. They should all be simp rules unless that causes too much search.

lemma $\text{pos-imp-zdiv-nonneg-iff}$:  
| fixes $a :: \text{int}$
| assumes $0 < b$
| shows $(0 \leq a \div b) = (0 \leq a)$
proof
  show $0 \leq a \div b \implies 0 \leq a$
    using assms
    by (simp add: linorder-not-less [symmetric]) (blast intro: div-neg-pos-less0)
next
  assume $0 \leq a$
  then have $0 \div b \leq a \div b$
    using zdiv-mono1 assms by blast
  then show $0 \leq a \div b$
    by auto

qed

lemma $\text{pos-imp-zdiv-pos-iff}$:  
| $0 < k \implies 0 < (i :: \text{int}) \div k \leftarrow k \leq i$
| using pos-imp-zdiv-nonneg-iff [of $k i$] zdiv-eq-0-iff [of $i k$] by arith

lemma $\text{neg-imp-zdiv-nonneg-iff}$:  
| fixes $a :: \text{int}$
| assumes $b < 0$
| shows $(0 \leq a \div b) = (a \leq 0)$
| using assms by (simp add: div-minus-minus [of $a$, symmetric] pos-imp-zdiv-nonneg-iff
del: div-minus-minus)
THEORY "Divides"

lemma pos-imp-zdiv-neg-iff: \((0::\text{int}) < b \Longrightarrow (a \text{ div } b < 0) = (a < 0)\)
  by (simp add: linorder-not-le [symmetric] pos-imp-zdiv-nonneg-iff)

lemma neg-imp-zdiv-neg-iff: \(b < (0::\text{int}) \Longrightarrow (a \text{ div } b < 0) = (0 < a)\)
  by (simp add: linorder-not-le [symmetric] neg-imp-zdiv-nonneg-iff)

lemma nonneg1-imp-zdiv-pos-iff:
  fixes a :: int
  assumes 0 \leq a
  shows a \text{ div } b > 0 \iff a \geq b \land b > 0
proof
  have 0 < a \text{ div } b \Longrightarrow b \leq a
    using div-pos-pos-trivial[of a b] assms by arith
  moreover have 0 < a \text{ div } b \Longrightarrow b > 0
    using assms div-nonneg-neg-le0[of a b] by (cases b=0; force)
  moreover have b \leq a \land 0 < b \Longrightarrow 0 < a \text{ div } b
    using int-one-le-iff-zero-less[of a \text{ div } b] zdiv-mono1[of b a b] by simp
  ultimately show ?thesis
    by blast
qed

lemma zmod-le-nonneg-dividend: \((m::\text{int}) \geq 0 \Longrightarrow m \mod k \leq m\)
  by (rule split-zmod[THEN iffD2]) (fastforce dest: q-pos-lemma intro: split-mult-pos-le)

56.1.8 Further properties

lemma div-int-pos-iff:
  \(k \text{ div } l \geq 0 \iff k = 0 \lor l = 0 \lor k \geq 0 \land l \geq 0\)
  for k l :: int
proof (cases k = 0 \lor l = 0)
  case False
  then show ?thesis
    apply (auto simp add: pos-imp-zdiv-nonneg-iff neg-imp-zdiv-nonneg-iff)
    by (meson neg-imp-zdiv-neg-iff not-le not-less-iff-gr-or-eq)
  qed auto

lemma mod-int-pos-iff:
  \(k \mod l \geq 0 \iff l \text{ dvd } k \lor l = 0 \lor k \geq 0 \lor l > 0\)
  for k l :: int
proof (cases l > 0)
  case False
  then show ?thesis
    by (simp add: dvd-eq-mod-eq-0) (use neg-mod-sign[of l k] in (auto simp add: le-less not-less))
  qed auto

Simplify expressions in which div and mod combine numerical constants
lemma int-div-pos-eq: \[(a::int) = b \cdot q + r; \ 0 \leq r; \ r < b] \implies a \div b = q\)
by (rule div-int-unique [of a b q r], simp add: eucl-rel-int-iff)

lemma int-div-neg-eq: \[(a::int) = b \cdot q + r; \ r \leq 0; \ b < r] \implies a \div b = q\)
by (rule div-int-unique [of a b q r], simp add: eucl-rel-int-iff)

lemma int-mod-pos-eq: \[(a::int) = b \cdot q + r; \ 0 \leq r; \ r < b] \implies a \mod b = r\)
by (rule mod-int-unique [of a b q r], simp add: eucl-rel-int-iff)

lemma int-mod-neg-eq: \[(a::int) = b \cdot q + r; \ r \leq 0; \ b < r] \implies a \mod b = r\)
by (rule mod-int-unique [of a b q r], simp add: eucl-rel-int-iff)

lemma abs-div: \(y::int\) dvd \(x\) \implies |\(x \div y\)| = |\(x\)| \(\div \)|\(y\)|
unfolding dvd-def by (cases y=0) (auto simp add: abs-mult)

Suggested by Matthias Daum

lemma int-power-div-base:
fixes \(k :: int\)
assumes \(0 < m \ 0 < k\)
shows \(k \cdot m \div k = (k::int) \cdot (m - Suc 0)\)
proof
–
have eq: \(k \cdot m = k \cdot ((m - Suc 0) + Suc 0)\)
by (simp add: assms)
show ?thesis

text{ using assms by (simp only: power-add eq) auto }

qed

Distributive laws for function nat.

lemma nat-div-distrib:
assumes \(0 \leq x\)
shows nat \((x \div y)\) = nat \(x \div nat y\)
proof (cases y 0::int rule: linorder-cases)
case less
with assms show ?thesis

text{ using div-nonneg-neg-le0 by auto }
next
case greater
then show ?thesis

text{ by (simp add: nat-eq iff pos-imp-zdiv-nonneg-iff zdiv-int) }
qed auto

lemma nat-mod-distrib:
assumes \(0 \leq x \ 0 \leq y\)
shows nat \((x \mod y)\) = nat \(x \mod nat y\)
proof (cases y 0)

qed
case False
with assms show ?thesis
  by (simp add: nat-eq-iff zmod-int)
qed auto

Suggested by Matthias Daum

lemma int-div-less-self:
  fixes x :: int
  assumes 0 < x 1 < k
  shows  x div k < x
proof –
  have nat x div nat k < nat x
    by (simp add: assms)
with assms show ?thesis
  by (simp add: nat-div-distrib [symmetric])
qed

lemma mod-eq-dvd-iff-nat:
  m mod q = n mod q ←→ q dvd m − n if m ≥ n for m n q :: nat
proof –
  have int m mod int q = int n mod int q ←→ int q dvd int m − int n
    by (simp add: mod-eq-dvd-iff)
with that have int (m mod q) = int (n mod q) ←→ int q dvd int (m − n)
  by (simp only: of-nat-mod of-nat-diff)
then show ?thesis
  by simp
qed

lemma mod-eq-nat1E:
  fixes m n q :: nat
  assumes m mod q = n mod q and m ≥ n
  obtains s where m = n + q * s
proof –
  from assms have q dvd m − n
    by (simp add: mod-eq-dvd-iff-nat)
then obtain s where m − n = q * s ..
  with (m ≥ n) have m = n + q * s
    by simp
with that show thesis .
qed

lemma mod-eq-nat2E:
  fixes m n q :: nat
  assumes m mod q = n mod q and n ≥ m
  obtains s where n = m + q * s
using assms mod-eq-nat1E [of n q m] by (auto simp add: ac-simps)

lemma nat-mod-eq-lemma:
  assumes (x::nat) mod n = y mod n and y ≤ x
shows $\exists q. \ x = y + n \cdot q$
using 

lemma nat-mod-eq-iff: $(\cdot x::nat) \ mod n = y \ mod n \iff (\exists q1 \ q2. \ x + n \cdot q1 = y + n \cdot q2)$
(is ?lhs = ?rhs)

proof

assume $H: x \ mod n = y \ mod n$
{assume $xy: x \leq y$
  from $H$ have th: $y \ mod n = x \ mod n$ by simp
  from nat-mod-eq-lemma[OF th $xy$] have ?rhs
  apply clarify apply (rule-tac x = q in exI) by (rule exI[where $x=0$, simp])}
moreover
{assume $xy: y \leq x$
  from nat-mod-eq-lemma[OF $H$ $xy$] have ?rhs
  apply clarify apply (rule-tac x = $0$ in exI, simp)}
ultimately show ?rhs using linear[of $x \ y$] by blast
next
assume ?rhs then obtain $q1 \ q2$ where $q12: x + n \cdot q1 = y + n \cdot q2$ by blast
hence $(x + n \cdot q1) \ mod n = (y + n \cdot q2) \ mod n$ by simp
thus ?lhs by simp

qed

56.2 Numeral division with a pragmatic type class

The following type class contains everything necessary to formulate a division algorithm in ring structures with numerals, restricted to its positive segments. This is its primary motivation, and it could surely be formulated using a more fine-grained, more algebraic and less technical class hierarchy.

class unique-euclidean-semiring-numeral = unique-euclidean-semiring-with-nat + linordered-semidom +

assumes

div-less: $0 \leq a \Longrightarrow a < b \Longrightarrow a \ div b = 0$
and mod-less: $0 \leq a \Longrightarrow a < b \Longrightarrow a \ mod b = a$
and div-positive: $0 < b \Longrightarrow b \leq a \Longrightarrow a \ div b > 0$
and mod-less-eq-dividend: $0 \leq a \Longrightarrow a \ mod b \leq a$
and pos-mod-bound: $0 < b \Longrightarrow a \ mod b < b$
and pos-mod-sign: $0 < b \Longrightarrow 0 \leq a \ mod b$
and div-mult2-eq: $0 \leq c \Longrightarrow a \ mod (b \cdot c) = b \cdot (a \ div b \ mod c) + a \ mod b$
and div-mult2-eq: $0 \leq c \Longrightarrow a \ div (b \cdot c) = a \ div b \ div c$

assumes
divmod-def: $\text{divmod } m \ n = (\text{numeral } m \ div \ \text{numeral } n, \ \text{numeral } m \ mod \ \text{numeral } n)$

and
divmod-step-def: $\text{divmod-step } l \ qr = (\text{let } (q, r) = qr$
in if $r \geq \text{numeral } l$ then $(2 \ast q + 1, \ r - \text{numeral } l)$
else $(2 \ast q, \ r)$

These are conceptually definitions but force generated code to be monomorphic
wrt. particular instances of this class which yields a significant speedup.

begin

lemma divmod-digit-1:
assumes \(0 \leq a, 0 < b\) and \(b \leq a \mod (2 \cdot b)\)
shows \(2 \cdot (a \div (2 \cdot b)) + 1 = a \div b\) (is \(?P\))
and \(a \mod (2 \cdot b) - b = a \mod b\) (is \(?Q\))
proof –
from asssms mod-less-eq-dividend [of a 2 \cdot b] have \(b \leq a\)
by (auto intro: trans)
with \(0 < b\) have \(0 < a \div b\) by (auto intro: div-positive)
then have [simp]: \(1 \leq a \div b\) by (simp add: discrete)
with \(0 < b\) have mod-less: \(a \mod b < b\) by (simp add: pos-mod-bound)
define \(w\) where \(w = a \div b \mod 2\)
then have w-exhaust: \(w = 0 \lor w = 1\) by auto
have mod-w: \(a \mod (2 \cdot b) = a \mod b + b \cdot w\)
  by (simp add: w-def mod-mult2-eq ac-simps)
moreover have \(b \leq a \mod b + b\)
proof –
from \(0 < b\) pos-mod-sign have \(0 \leq a \mod b\) by blast
then have \(0 + b \leq a \mod b + b\) by (rule add-right-mono)
then show \(?\)thesis by simp
qed

lemma divmod-digit-0:
assumes \(0 < b\) and \(a \mod (2 \cdot b) < b\)
shows \(2 \cdot (a \div (2 \cdot b)) = a \div b\) (is \(?P\))
and \(a \mod (2 \cdot b) = a \mod b\) (is \(?Q\))
proof –
define \(w\) where \(w = a \div b \mod 2\)
then have w-exhaust: \(w = 0 \lor w = 1\) by auto
have mod-w: \(a \mod (2 \cdot b) = a \mod b + b \cdot w\)
  by (simp add: w-def mod-mult2-eq ac-simps)
moreover have \(b \leq a \mod b + b\)
proof –
from \(0 < b\) have \(0 \leq a \mod b\) by blast
then have \(0 + b \leq a \mod b + b\) by (rule add-right-mono)
then show \(?\)thesis by simp
qed

moreover note asssms w-exhaust
ultimately have \(w = 0\) by auto
with mod-w have mod: \(a \mod (2 \cdot b) = a \mod b\) by simp
have \(2 \cdot (a \div (2 \cdot b)) = a \div b - w\)
  by (simp add: w-def div-mult2-eq minus-mod-eq-mult-div ac-simps)
with \(\langle w = 0 \rangle\) have \(\langle w = 0 \rangle\) \(\langle \div (2 \cdot b) = a \div b\) by simp
then show \(?P\) and \(?Q\)
by (simp-all add: div mod)

qed

lemma mod-double-modulus:
  assumes \( m > 0 \) \( x \geq 0 \)
  shows \( x \mod (2 \ast m) = x \mod m \lor x \mod (2 \ast m) = x \mod m + m \)
proof (cases \( x \mod (2 \ast m) < m \))
  case True
  thus \( ?thesis \) using assms using divmod-digit-0[of \( m \) \( x \)] by auto
next
  case False
  hence \( \ast \): \( x \mod (2 \ast m) - m = x \mod m \)
  using assms by (intro divmod-digit-1) auto
  hence \( x \mod (2 \ast m) = x \mod m + m \)
  by (subst \( * \) [symmetric], subst le-add-diff-inverse2) (use False in auto)
  thus \( ?thesis \) by simp

qed

lemma fst-divmod:
  \( \text{fst } (\text{divmod } m \ n) = \text{numeral } m \ \text{div } \text{numeral } n \)
by (simp add: divmod-def)

lemma snd-divmod:
  \( \text{snd } (\text{divmod } m \ n) = \text{numeral } m \ \text{mod } \text{numeral } n \)
by (simp add: divmod-def)

This is a formulation of one step (referring to one digit position) in school-method division: compare the dividend at the current digit position with the remainder from previous division steps and evaluate accordingly.

lemma divmod-step-eq [simp]:
  \( \text{divmod-step } l \ (q, r) = (\text{if numeral } l \leq r \)
  \( \text{then } (2 \ast q + 1, r - \text{numeral } l) \ \text{else } (2 \ast q, r) \) \)
by (simp add: divmod-step-def)

This is a formulation of school-method division. If the divisor is smaller than the dividend, terminate. If not, shift the dividend to the right until termination occurs and then reiterate single division steps in the opposite direction.

lemma divmod-divmod-step:
  \( \text{divmod } m \ n = (\text{if } m < n \ \text{then } (0, \text{numeral } m) \)
  \( \text{else } \text{divmod-step } n \ (\text{divmod } m \ (\text{Num.Bit0 } n)) \) \)
proof (cases \( m < n \))
  case True then have \( \text{numeral } m < \text{numeral } n \) by simp
  then show \( ?thesis \) by (simp add: prod-eq-iff div-less mod-less fst-divmod snd-divmod)
next
  case False
  have \( \text{divmod } m \ n = \)
THEORY “Divides”

\[
\text{divmod-step } n \text{ (numeral } m \text{ div } (2 \times \text{numeral } n), \\
\text{numeral } m \text{ mod } (2 \times \text{numeral } n))
\]

proof (cases \text{numeral } n = \text{numeral } m \text{ mod } (2 \times \text{numeral } n))

\[
\begin{align*}
\text{case True} \\
&\text{with divmod-step-eq} \\
&\text{have divmod-step } n \text{ (numeral } m \text{ div } (2 \times \text{numeral } n), \\
\text{numeral } m \text{ mod } (2 \times \text{numeral } n)) = \\
&\quad (2 \times (\text{numeral } m \text{ div } (2 \times \text{numeral } n)) + 1, \\
\text{numeral } m \text{ mod } (2 \times \text{numeral } n) - \text{numeral } n)
\end{align*}
\]

by simp

moreover from True divmod-digit-1 [of \text{numeral } m \text{ numeral } n]

have \(2 \times (\text{numeral } m \text{ div } (2 \times \text{numeral } n)) + 1 = \text{numeral } m \text{ div numeral } n\)

and numeral \text{mod } (2 \times \text{numeral } n) - \text{numeral } n = \text{numeral } m \text{ mod numeral } n

by simp-all

ultimately show \(?\text{thesis}\) by (simp only: divmod-def)

next

\[
\begin{align*}
\text{case False then have *: numeral } m \text{ mod } (2 \times \text{numeral } n) < \text{numeral } n \\
&\text{by (simp add: not-le)} \\
&\text{with divmod-step-eq} \\
&\text{have divmod-step } n \text{ (numeral } m \text{ div } (2 \times \text{numeral } n), \\
\text{numeral } m \text{ mod } (2 \times \text{numeral } n)) = \\
&\quad (2 \times (\text{numeral } m \text{ div } (2 \times \text{numeral } n)), \\
\text{numeral } m \text{ mod } (2 \times \text{numeral } n))
\end{align*}
\]

by auto

moreover from \* divmod-digit-0 [of \text{numeral } n \text{ numeral } m]

have \(2 \times (\text{numeral } m \text{ div } (2 \times \text{numeral } n)) = \text{numeral } m \text{ div numeral } n\)

and numeral \text{mod } (2 \times \text{numeral } n) = \text{numeral } m \text{ mod numeral } n

by (simp-all only: zero-less-numeral)

ultimately show \(?\text{thesis}\) by (simp only: divmod-def)

qed

then have \(\text{divmod } m \text{ n} = \\
\text{divmod-step } n \text{ (numeral } m \text{ div numeral } (\text{Num.Bit0 } n), \\
\text{numeral } m \text{ mod numeral } (\text{Num.Bit0 } n))\)

by (simp only: numeral.simps distrib mult-1)

then have \(\text{divmod } m \text{ n} = \text{divmod-step } n \text{ (divmod } m \text{ (Num.Bit0 } n))\)

by (simp add: divmod-def)

with False show \(?\text{thesis}\) by simp

qed

The division rewrite proper – first, trivial results involving \(1\)

\[
\text{lemma divmod-trivial [simp]:} \\
\text{divmod } \text{Num.One Num.One } = (\text{numeral } \text{Num.One}, \text{0}) \\
\text{divmod } (\text{Num.Bit0 } m) \text{ Num.One } = (\text{numeral } (\text{Num.Bit0 } m), \text{0}) \\
\text{divmod } (\text{Num.Bit1 } m) \text{ Num.One } = (\text{numeral } (\text{Num.Bit1 } m), \text{0}) \\
\text{divmod } \text{num.} \text{One } (\text{num.Bit0 } n) = (\text{0}, \text{Numeral1}) \\
\text{divmod } \text{num.} \text{One } (\text{num.Bit1 } n) = (\text{0}, \text{Numeral1})
\]

using \text{divmod-divmod-step [of Num.One]} by (simp-all add: divmod-def)

Division by an even number is a right-shift
lemma divmod-cancel [simp]:
  divmod (Num.Bit0 m) (Num.Bit0 n) = (case divmod m n of (q, r) => (q, 2 * r)) (is ?P)
  divmod (Num.Bit1 m) (Num.Bit0 n) = (case divmod m n of (q, r) => (q, 2 * r + 1)) (is ?Q)

proof -
  have *: \(\forall q. \text{numeral (Num.Bit0 q)} = 2 * \text{numeral q}\)
  by (simp-all only: numeral-mult numeral.simps distrib)
  simp-all
  have 1 div 2 = 0 1 mod 2 = 1 by (auto intro: div-less mod-less)
  then show ?P and ?Q
  by (simp-all add: fst-divmod snd-divmod prod-eq-iff split-def)
qed

The really hard work

lemma divmod-steps [simp]:
  divmod (num.Bit0 m) (num.Bit1 n) = 
  (if m <= n then (0, numeral (num.Bit0 m))
    else divmod-step (num.Bit1 n)
         (divmod (num.Bit0 m))
         (num.Bit0 (num.Bit1 n))))
  divmod (num.Bit1 m) (num.Bit1 n) = 
  (if m < n then (0, numeral (num.Bit1 m))
    else divmod-step (num.Bit1 n)
         (divmod (num.Bit1 m))
         (num.Bit0 (num.Bit1 n))))
by (simp-all add: divmod-divmod-step)

lemmas divmod-algorithm-code = divmod-step-eq divmod-trivial divmod-cancel divmod-steps

Special case: divisibility

definition divides-aux :: 'a × 'a ⇒ bool
where
  divides-aux qr ↔ snd qr = 0

lemma divides-aux-eq [simp]:
  divides-aux (q, r) ↔ r = 0
  by (simp add: divides-aux-def)

lemma dvd-numeral-simp [simp]:
  numeral m dvd numeral n ↔ divides-aux (divmod n m)
  by (simp add: divmod-def mod-eq-0-iff-dvd)

Generic computation of quotient and remainder

lemma numeral-div-numeral [simp]:
  numeral k div numeral l = fst (divmod k l)
by (simp add: fst-divmod)

lemma numeral-mod-numeral [simp]:
    numeral k mod numeral l = snd (divmod k l)
by (simp add: snd-divmod)

lemma one-div-numeral [simp]:
    1 div numeral n = fst (divmod Num.One n)
by (simp add: fst-divmod)

lemma one-mod-numeral [simp]:
    1 mod numeral n = snd (divmod Num.One n)
by (simp add: snd-divmod)

Computing congruences modulo \(2^q\)

lemma cong-exp-iff-simps:
    numeral n mod numeral Num.One = 0
      ⬛️ True
    numeral (Num.Bit0 n) mod numeral (Num.Bit0 q) = 0
      ⬛️ numeral n mod numeral q = 0
    numeral (Num.Bit1 n) mod numeral (Num.Bit0 q) = 0
      ⬛️ False
    numeral m mod numeral Num.One = (numeral n mod numeral Num.One)
      ⬛️ True
    numeral Num.One mod numeral (Num.Bit0 q) = (numeral Num.One mod numeral (Num.Bit0 q))
      ⬛️ True
    numeral Num.One mod numeral (Num.Bit0 q) = (numeral (Num.Bit0 n) mod numeral (Num.Bit0 q))
      ⬛️ False
    numeral Num.One mod numeral (Num.Bit0 q) = (numeral (Num.Bit1 n) mod numeral (Num.Bit0 q))
      ⬛️ (numeral n mod numeral q = 0)
    numeral (Num.Bit0 m) mod numeral (Num.Bit0 q) = (numeral Num.One mod numeral (Num.Bit0 q))
      ⬛️ False
    numeral (Num.Bit0 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit0 n) mod numeral (Num.Bit0 q))
      ⬛️ numeral m mod numeral q = (numeral n mod numeral q)
    numeral (Num.Bit0 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit1 n) mod numeral (Num.Bit0 q))
      ⬛️ False
    numeral (Num.Bit1 m) mod numeral (Num.Bit0 q) = (numeral Num.One mod numeral (Num.Bit0 q))
      ⬛️ (numeral m mod numeral q = 0)
    numeral (Num.Bit1 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit0 n) mod numeral (Num.Bit0 q))
      ⬛️ False
    numeral (Num.Bit1 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit1 n) mod numeral (Num.Bit0 q))
      ⬛️ False
    numeral (Num.Bit1 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit1 n) mod numeral (Num.Bit0 q))
      ⬛️ False
    numeral (Num.Bit1 m) mod numeral (Num.Bit0 q) = (numeral (Num.Bit1 n) mod numeral (Num.Bit0 q))
      ⬛️ False
mod numeral (Num.Bit0 q)
  \[\mapsto\text{numeral } m \mod \text{numeral } q = \text{numeral } n \mod \text{numeral } q\]
by (auto simp add: case_prod_beta dest: arg_cong [of _ _ even])
end

hide-fact (open) div-less mod-less mod-less-eq-dividend mod-mult2-eq div-mult2-eq

instantiation nat :: unique_euclidean_semiring_numeral
begin

definition divmod_nat :: num ⇒ num ⇒ nat × nat
where
  divmod'_nat_def: divmod_nat m n = (numeral m div numeral n, numeral m mod numeral n)

definition divmod_step_nat :: num ⇒ nat × nat ⇒ nat × nat
where
  divmod_step_nat l qr = (let (q, r) = qr
  in if r ≥ numeral l then (2 * q + 1, r - numeral l)
    else (2 * q, r))

instance by standard
(auto simp add: divmod'_nat_def divmod_step_nat_def div_greater_zero_iff div_mult2_eq
mod_mult2_eq)
end

declare divmod_algorithm_code [where ?a = nat, code]

lemma Suc-0-div-numeral [simp]:
  fixes k l :: num
  shows Suc 0 div numeral k = fst (divmod Num.One k)
  by (simp_all add: fst_divmod)

lemma Suc-0-mod-numeral [simp]:
  fixes k l :: num
  shows Suc 0 mod numeral k = snd (divmod Num.One k)
  by (simp_all add: snd_divmod)

instantiation int :: unique_euclidean_semiring_numeral
begin

definition divmod_int :: num ⇒ num ⇒ int × int
where
  divmod_int m n = (numeral m div numeral n, numeral m mod numeral n)

definition divmod_step_int :: num ⇒ int × int ⇒ int × int
where
THEORY "Divides"

divmod-step-int l qr = (let (q, r) = qr
  in if r ≥ numeral l then (2 * q + 1, r - numeral l)
  else (2 * q, r))

instance
  by standard (auto intro: zmod-le-nonneg-dividend simp add: divmod-int-def divmod-step-int-def
  pos-imp-zdiv-pos-iff zmod-zmult2-eq zdiv-zmult2-eq)
end

declare divmod-algorithm-code [where ?'a = int, code]

context
begin

qualified definition adjust-div :: int × int ⇒ int
  where
  adjust-div qr = (let (q, r) = qr in q + of-bool (r ≠ 0))

qualified lemma adjust-div-eq [simp, code]:
  adjust-div (q, r) = q + of-bool (r ≠ 0)
  by (simp add: adjust-div-def)

qualified definition adjust-mod :: int ⇒ int ⇒ int
  where
    simp]: adjust-mod l r = (if r = 0 then 0 else l - r)

lemma minus-numeral-div-numeral [simp]:
  - numeral m div numeral n = -(adjust-div (divmod m n) :: int)
proof
  have int (fst (divmod m n)) = fst (divmod m n)
    by (simp only: fst-divmod divide-int-def) auto
  then show ?thesis
    by (auto simp add: split-def Let-def adjust-div-def divides-aux-def divide-int-def) qed

lemma minus-numeral-mod-numeral [simp]:
  - numeral m mod numeral n = adjust-mod (numeral n) (snd (divmod m n) :: int)
proof (cases snd (divmod m n) = (0::int))
  case True
  then show ?thesis
    by (simp add: mod-eq-0-iff-dvd divides-aux-def)
next
  case False
  then have int (snd (divmod m n)) = snd (divmod m n) if snd (divmod m n) ≠ (0::int)
    by (simp only: snd-divmod modulo-int-def) auto
  then show ?thesis
by (simp add: divides-aux-def adjust-div-def) (simp add: divides-aux-def modulo-int-def)

qed

lemma numeral-div-minus-numeral [simp]:
numeral m div - numeral n = - (adjust-div (divmod m n) :: int)
proof -
  have int (fst (divmod m n)) = fst (divmod m n)
    by (simp only: fst-divmod divide-int-def) auto
  then show ?thesis
    by (auto simp add: split-def Let-def adjust-div-def divides-aux-def divide-int-def)
qed

lemma numeral-mod-minus-numeral [simp]:
numeral m mod - numeral n = - adjust-mod (numeral n) (snd (divmod m n) :: int)
proof (cases snd (divmod m n) = (0::int))
  case True
  then show ?thesis
    by (simp add: mod-eq-0-iff-dvd divides-aux-def)
next
  case False
  then have int (snd (divmod m n)) = snd (divmod m n) if snd (divmod m n) ≠ (0::int)
    by (simp only: snd-divmod modulo-int-def) auto
  then show ?thesis
    by (simp add: divides-aux-def adjust-div-def) (simp add: divides-aux-def modulo-int-def)
qed

lemma minus-one-div-numeral [simp]:
- 1 div numeral n = - (adjust-div (divmod Num.One n) :: int)
using minus-numeral-div-numeral [of Num.One n] by simp

lemma minus-one-mod-numeral [simp]:
- 1 mod numeral n = adjust-mod (numeral n) (snd (divmod Num.One n) :: int)
using minus-numeral-mod-numeral [of Num.One n] by simp

lemma one-div-minus-numeral [simp]:
  1 div - numeral n = - (adjust-div (divmod Num.One n) :: int)
using numeral-div-minus-numeral [of Num.One n] by simp

lemma one-mod-minus-numeral [simp]:
  1 mod - numeral n = - adjust-mod (numeral n) (snd (divmod Num.One n) :: int)
using numeral-mod-minus-numeral [of Num.One n] by simp

end
56.2.1 Dedicated simproc for calculation

There is space for improvement here: the calculation itself could be carried out outside the logic, and a generic simproc (simplifier setup) for generic calculation would be helpful.

```
lemma div-positive-int:
  k div l > 0 if k ≥ l and l > 0 for k l :: int
using that div-positive [of l k] by blast
```

```
56.2.1 Dedicated simproc for calculation

There is space for improvement here: the calculation itself could be carried out outside the logic, and a generic simproc (simplifier setup) for generic calculation would be helpful.

```
```
in fn phi =>
let
  val simps = Morphism.fact phi (@[\{ thms div-0 mod-0 div-by-0 mod-by-0 div-by-1
                   mod-by-1
                   one-div-numeral one-mod-numeral minus-one-div-numeral minus-one-mod-numeral
                   one-div-minus-numeral one-mod-minus-numeral
                   numeral-div-numeral numeral-mod-numeral minus-numeral-div-numeral minus-numeral-mod-numeral
                   numeral-div-minus-numeral numeral-mod-minus-numeral
                   div-minus-minus mod-minus-minus Divides.adjust-div-eq of-bool-eq one-neq-zero
                   numeral-neq-zero neg-equal-0-iff-equal arith-simps arith-special divmod-trivial
                   divmod-cancel divmod-steps divmod-step-eq fst-conv snd-conv numeral-One
                   case-prod-beta rel-simps Divides.adjust-mod-def div-minus1-right mod-minus1-right
                   minus-minus numeral-times-numeral mult-zero-right mult-1-right
                   @ (@{lemma 0 = 0 <-> True by simp}}));
fun prepare-simpset ctxt = HOL-ss |
Simplifier.simpset-map ctxt (Simplifier.add-cong if-cong #> fold Simplifier.add-simp simps)
in fn ctxt => successful-rewrite (Simplifier.put-simpset (prepare-simpset ctxt) ctxt) end end
⟩

56.2.2 Code generation
definition divmod-nat :: nat ⇒ nat ⇒ nat × nat
where
divmod-nat m n = (m div n, m mod n)

lemma fst-divmod-nat [simp]:
  fst (divmod-nat m n) = m div n
  by (simp add: divmod-nat-def)

lemma snd-divmod-nat [simp]:
  snd (divmod-nat m n) = m mod n
  by (simp add: divmod-nat-def)

lemma divmod-nat-if [code]:
  Divides.divmod-nat m n = (if n = 0 ∨ m < n then (0, m) else
  let (q, r) = Divides.divmod-nat (m - n) n in (Suc q, r))
  by (simp add: prod-eq-iff case-prod-beta not-less le-div-geq le-mod-geq)

lemma [code]:
  m div n = fst (divmod-nat m n)
  m mod n = snd (divmod-nat m n)
  by simp-all

lemma [code]:
  fixes k :: int
  shows
    k div 0 = 0
THEORY "Numeral-Simprocs"

k mod 0 = k
0 div k = 0
0 mod k = 0
k div Int.Pos Num.One = k
k mod Int.Pos Num.One = 0
k div Int.Neg Num.One = -k
k mod Int.Neg Num.One = 0
Int.Pos m div Int.Pos n = (fst (divmod m n) :: int)
Int.Pos m mod Int.Pos n = (snd (divmod m n) :: int)
Int.Neg m div Int.Pos n = - (Divides.adjust-div (divmod m n) :: int)
Int.Neg m mod Int.Pos n = Divides.adjust-mod (Int.Pos n) (snd (divmod m n) :: int)
Int.Pos m div Int.Neg n = - (Divides.adjust-div (divmod m n) :: int)
Int.Pos m mod Int.Neg n = - Divides.adjust-mod (Int.Pos n) (snd (divmod m n) :: int)
Int.Neg m div Int.Neg n = (fst (divmod m n) :: int)
Int.Neg m mod Int.Neg n = - (snd (divmod m n) :: int)
by simp

code-identifier
code-module Divides -> (SML) Arith and (OCaml) Arith and (Haskell) Arith

56.3 Lemmas of doubtful value

lemma div-geq: m div n = Suc ((m div n) div n) if 0 < n and ¬ m < n for m n :: nat
  by (rule le-div-geq) (use that in simp-all add: not-less:)

lemma mod-geq: m mod n = (m div n) mod n if ¬ m < n for m n :: nat
  by (rule le-mod-geq) (use that in simp add: not-less:)

lemma mod-eq-0D: ∃ q. m = d * q if m mod d = 0 for m d :: nat
  using that by (auto simp add: mod-eq-0-iff-dvd)

lemma pos-mod-conj: 0 < b ==> 0 ≤ a mod b ∧ a mod b < b for a b :: int
  by simp

lemma neg-mod-conj: b < 0 ==> a mod b ≤ 0 ∧ b < a mod b for a b :: int
  by simp

lemma zmod-eq-0D: m mod d = 0 <-> (∃ q. m = d * q) for m d :: int
  by (auto simp add: mod-eq-0-iff-dvd)

lemma zmod-eq-0D [dest!]: ∃ q. m = d * q if m mod d = 0 for m d :: int
  using that by auto

end
Combination and Cancellation Simprocs for Numerical Expressions

theory Numeral-Simprocs
imports Divides
begin

ML-file ⟨~/src/Provers/Arith/assoc-fold.ML⟩
ML-file ⟨~/src/Provers/Arith/cancel-numerals.ML⟩
ML-file ⟨~/src/Provers/Arith/combine-numerals.ML⟩
ML-file ⟨~/src/Provers/Arith/cancel-numeral-factor.ML⟩
ML-file ⟨~/src/Provers/Arith/extract-common-term.ML⟩

lemmas semiring-norm =
  Let-def arith-simps diff-nat-numeral rel-simps
  if-False if-True
  add-0 add-Suc add-numeral-left
  add-neg-numeral-left mult-numeral-left
  numeral-One [symmetric] uminus-numeral-One [symmetric] Suc-eq-plus1
  eq-numeral-iff-iszero not-iszero-Numeral1

declare split-div [of - - numeral k, arith-split] for k
declare split-mod [of - - numeral k, arith-split] for k

For combine-numerals

lemma left-add-mult-distrib: i*u + (j*u + k) = (i+j)*u + (k::nat)
by (simp add: add-mult-distrib)

For cancel-numerals

lemma nat-diff-add-eq1:
  j <= (i::nat) ==>(i*u + m) - (j*u + n)) = (((i-j)*u + m) - n)
by (simp split: nat-diff-split add: add-mult-distrib)

lemma nat-diff-add-eq2:
  i <= (j::nat) ==>((i*u + m) - (j*u + n)) = (m - (j-i)*u + n)
by (simp split: nat-diff-split add: add-mult-distrib)

lemma nat-eq-add-iff1:
  j <= (i::nat) ==>(i*u + m = j*u + n) = ((i-j)*u + m = n)
by (auto split: nat-diff-split simp add: add-mult-distrib)

lemma nat-eq-add-iff2:
  i <= (j::nat) ==>((i*u + m = j*u + n) = (m = (j-i)*u + n)
by (auto split: nat-diff-split simp add: add-mult-distrib)

lemma nat-less-add-iff1:
  j <= (i::nat) ==>(i*u + m < j*u + n) = ((i-j)*u + m < n)
by (auto split: nat-diff-split simp add: add-mult-distrib)
lemma nat-less-add-iff2:
  \( i < (j :: \text{nat}) \iff (i \times u + m < j \times u + n) = (m < (j - i) \times u + n) \)
by (auto split: nat-diff-split simp add: add-mult-distrib)

lemma nat-le-add-iff1:
  \( j \leq (i :: \text{nat}) \iff (i \times u + m \leq j \times u + n) = ((j - i) \times u + m \leq n) \)
by (auto split: nat-diff-split simp add: add-mult-distrib)

lemma nat-le-add-iff2:
  \( i < (j :: \text{nat}) \iff (i \times u + m < j \times u + n) = (m \leq (j - i) \times u + n) \)
by (auto split: nat-diff-split simp add: add-mult-distrib)

For cancel-numeral-factors

lemma nat-mult-le-cancel1: \( (0 :: \text{nat}) < k \iff (k \times m \leq k \times n) = (m \leq n) \)
by auto

lemma nat-mult-less-cancel1: \( (0 :: \text{nat}) < k \iff (k \times m < k \times n) = (m < n) \)
by auto

lemma nat-mult-eq-cancel1: \( (0 :: \text{nat}) < k \iff (k \times m = k \times n) = (m = n) \)
by auto

lemma nat-mult-div-cancel1: \( (0 :: \text{nat}) < k \iff (k \times m) \div (k \times n) = (m \div n) \)
by auto

lemma nat-mult-dvd-cancel-disj [simp]:
  \((k \times m) \quad \text{dvd} \quad (k \times n) = (k = 0 \lor m \quad \text{dvd} \quad (n :: \text{nat}))\)
by (auto simp: dvd-eq-mod-eq-0 mod-mult-mult1)

lemma nat-mult-dvd-cancel1: \( 0 < k \implies (k \times m) \quad \text{dvd} \quad (k \times n :: \text{nat}) = (m \quad \text{dvd} \quad n) \)
by (auto)

For cancel-factor

lemmas nat-mult-le-cancel-disj = mult-le-cancel1

lemmas nat-mult-less-cancel-disj = mult-less-cancel1

lemma nat-mult-eq-cancel-disj:
  fixes k n :: nat
  shows \( k \times m = k \times n \iff k = 0 \lor m = n \)
by auto

lemma nat-mult-div-cancel-disj [simp]:
  fixes k n :: nat
  shows \( (k \times m) \quad \text{div} \quad (k \times n) = (\text{if } k = 0 \text{ then } 0 \text{ else } m \quad \text{div} \quad n) \)
by (fact div-mult-mult1-if)

lemma numeral-times-minus-swap:
  fixes x :: 'a :: comm_ring_1 shows \( \text{numeral } w \times -x = x \times - \text{numeral } w \)
by (simp add: mult.commute)

ML-file (Tools/numeral-simprocs.ML)

simproc-setup semiring-assoc-fold
((a::comm-semiring-1-cancel) * b) =
(fn phi => Numeral-Simprocs.assoc-fold)

simproc-setup int-combine-numerals
((i::comm-ring-1) + j | (i::comm-ring-1) - j) =
(fn phi => Numeral-Simprocs.combine-numerals)

simproc-setup field-combine-numerals
((i::{field,ring-char-0}) + j | (i::{field,ring-char-0}) - j) =
(fn phi => Numeral-Simprocs.field-combine-numerals)

simproc-setup inteq-cancel-numerals
((l::comm-ring-1) + m = n | (l::comm-ring-1) - m = n | (l::comm-ring-1) * m = n | - (l::comm-ring-1) = m | (l::comm-ring-1) = - m) =
(fn phi => Numeral-Simprocs.eq-cancel-numerals)

simproc-setup intless-cancel-numerals
((l::linordered-idom) + m < n | (l::linordered-idom) < m + n | (l::linordered-idom) - m < n | (l::linordered-idom) < m - n | (l::linordered-idom) * m < n | - (l::linordered-idom) < m | (l::linordered-idom) < - m) =
(fn phi => Numeral-Simprocs.less-cancel-numerals)

simproc-setup intle-cancel-numerals
((l::linordered-idom) + m ≤ n | (l::linordered-idom) ≤ m + n | (l::linordered-idom) - m ≤ n | (l::linordered-idom) ≤ m - n | (l::linordered-idom) * m ≤ n | - (l::linordered-idom) ≤ m | (l::linordered-idom) ≤ - m) =
THEORY "Numeral-Simprocs"

(fn phi => Numeral-Simprocs.le-cancel-numerals)

simproc-setup ring-eq-cancel-numeral-factor
((l::'a::{idom,ring-char-0}) * m = n
 |(l::'a::{idom,ring-char-0}) = m * n) =
(fn phi => Numeral-Simprocs.eq-cancel-numeral-factor)

simproc-setup ring-less-cancel-numeral-factor
((l::'a::linordered-idom) * m < n
 |(l::'a::linordered-idom) < m * n) =
(fn phi => Numeral-Simprocs.less-cancel-numeral-factor)

simproc-setup ring-le-cancel-numeral-factor
((l::'a::linordered-idom) * m <\= n
 |(l::'a::linordered-idom) <\= m * n) =
(fn phi => Numeral-Simprocs.le-cancel-numeral-factor)

simproc-setup int-div-cancel-numeral-factors
(((l::'a::{euclidean-semiring-cancel,comm-ring-1,ring-char-0}) * m) div n
 |(l::'a::{euclidean-semiring-cancel,comm-ring-1,ring-char-0}) div (m * n)) =
(fn phi => Numeral-Simprocs.div-cancel-numeral-factor)

simproc-setup divide-cancel-numeral-factor
(((l::'a::{field,ring-char-0}) * m) / n
 |((numeral v)::'a::{field,ring-char-0}) / (numeral w)) =
(fn phi => Numeral-Simprocs.divide-cancel-numeral-factor)

simproc-setup ring-le-cancel-factor
((l::'a::idom) * m = n | (l::'a::idom) = m * n) =
(fn phi => Numeral-Simprocs.eq-cancel-factor)

simproc-setup linordered-ring-le-cancel-factor
((l::'a::linordered-idom) * m <\= n
 |(l::'a::linordered-idom) <\= m * n) =
(fn phi => Numeral-Simprocs.le-cancel-factor)

simproc-setup linordered-ring-less-cancel-factor
((l::'a::linordered-idom) * m < n
 |(l::'a::linordered-idom) < m * n) =
(fn phi => Numeral-Simprocs.less-cancel-factor)

simproc-setup int-div-cancel-factor
(((l::'a::euclidean-semiring-cancel) * m) div n
 |(l::'a::euclidean-semiring-cancel) div (m * n)) =
(fn phi => Numeral-Simprocs.divide-cancel-factor)

simproc-setup int-mod-cancel-factor
THEORY "Numeral-Simprocs"

\[ ((\ell::'a::euclidean-semiring-cancel) \ast m) \mod n \]
\[ ((\ell::'a::euclidean-semiring-cancel) \mod (m \ast n)) = \langle \text{fn phi} => \text{Numeral-Simprocs.mod-cancel-factor} \rangle \]

\textbf{simproc-setup dvd-cancel-factor}
\[ ((\ell::'a::idom) \ast m) \dvd n \]
\[ ((\ell::'a::idom) \dvd (m \ast n)) = \langle \text{fn phi} => \text{Numeral-Simprocs.dvd-cancel-factor} \rangle \]

\textbf{simproc-setup divide-cancel-factor}
\[ ((\ell::'a::field) \ast m) / n \]
\[ ((\ell::'a::field) / (m \ast n)) = \langle \text{fn phi} => \text{Numeral-Simprocs.divide-cancel-factor} \rangle \]

\textbf{ML-file} ⟨Tools/nat-numeral-simprocs.ML⟩

\textbf{simproc-setup nat-combine-numerals}
\[ ((\ell::nat) + j | \text{Suc}(i + j)) = \langle \text{fn phi} => \text{Nat-Numeral-Simprocs.combine-numerals} \rangle \]

\textbf{simproc-setup nateq-cancel-numerals}
\[ ((\ell::nat) + m = n | (\ell::nat) = m + n | \text{Suc} m = n | m = \text{Suc} n) = \langle \text{fn phi} => \text{Nat-Numeral-Simprocs.eq-cancel-numerals} \rangle \]

\textbf{simproc-setup natless-cancel-numerals}
\[ ((\ell::nat) + m < n | (\ell::nat) < m + n | \text{Suc} m < n | m < \text{Suc} n) = \langle \text{fn phi} => \text{Nat-Numeral-Simprocs.less-cancel-numerals} \rangle \]

\textbf{simproc-setup natle-cancel-numerals}
\[ ((\ell::nat) + m \leq n | (\ell::nat) \leq m + n | \text{Suc} m \leq n | m \leq \text{Suc} n) = \langle \text{fn phi} => \text{Nat-Numeral-Simprocs.le-cancel-numerals} \rangle \]

\textbf{simproc-setup natdiff-cancel-numerals}
\[ ((\ell::nat) + m) - n | (\ell::nat) - (m + n) | \text{Suc} m - n | m - \text{Suc} n) = \langle \text{fn phi} => \text{Nat-Numeral-Simprocs.diff-cancel-numerals} \rangle \]

\textbf{simproc-setup nat-eq-cancel-numeral-factor}
\[ ((\ell::nat) \ast m = n | (\ell::nat) = m \ast n) = \langle \text{fn phi} => \text{Nat-Numeral-Simprocs.eq-cancel-numeral-factor} \rangle \]

\textbf{simproc-setup nat-less-cancel-numeral-factor}
THEORY "Numeral-Simprocs"

\((\langle l \text{:: nat} \rangle \ast m < n \mid (l \text{:: nat}) < m \ast n) = \) \\
\(\langle \text{fn phi => Nat-Numeral-Simprocs.less-cancel-numeral-factor} \rangle \)

\textbf{simproc-setup nat-le-cancel-numeral-factor} \\
\(\langle (\langle l \text{:: nat} \rangle \ast m <= n \mid (l \text{:: nat}) <= m \ast n) = \) \\
\(\langle \text{fn phi => Nat-Numeral-Simprocs.le-cancel-numeral-factor} \rangle \)

\textbf{simproc-setup nat-dve-cancel-numeral-factor} \\
\(\langle (\langle l \text{:: nat} \rangle \ast m) \text{ div } n \mid (l \text{:: nat}) \text{ div } (m \ast n) = \) \\
\(\langle \text{fn phi => Nat-Numeral-Simprocs.dve-cancel-numeral-factor} \rangle \)

\textbf{simproc-setup nat-eq-cancel-factor} \\
\(\langle (\langle l \text{:: nat} \rangle \ast m = n \mid (l \text{:: nat}) = m \ast n) = \) \\
\(\langle \text{fn phi => Nat-Numeral-Simprocs.eq-cancel-factor} \rangle \)

\textbf{simproc-setup nat-less-cancel-factor} \\
\(\langle (\langle l \text{:: nat} \rangle \ast m < n \mid (l \text{:: nat}) < m \ast n) = \) \\
\(\langle \text{fn phi => Nat-Numeral-Simprocs.less-cancel-factor} \rangle \)

\textbf{declaration} \\
\(K \text{ Lin-Arith.add-simprocs} \) \\
\(\textbf{simproc} (\text{semiring-assoc-fold}), \) \\
\(\textbf{simproc} (\text{int-combine-numerals}), \) \\
\(\textbf{simproc} (\text{int-cancel-numerals}), \) \\
\(\textbf{simproc} (\text{intless-cancel-numerals}), \) \\
\(\textbf{simproc} (\text{intle-cancel-numerals}), \) \\
\(\textbf{simproc} (\text{field-combine-numerals}), \) \\
\(\textbf{simproc} (\text{nat-combine-numerals}), \) \\
\(\textbf{simproc} (\text{nateq-cancel-numerals}), \) \\
\(\textbf{simproc} (\text{natless-cancel-numerals}), \) \\
\(\textbf{simproc} (\text{natle-cancel-numerals}), \) \\
\(\textbf{simproc} (\text{natdiff-cancel-numerals}), \) \\
\(\textbf{simproc} (\text{Nat-Simprocs.field-divide-cancel-numeral-factor})) \)
58 Semiring normalization

theory Semiring-Normalization
imports Numeral-Simprocs
begin

Prelude

class comm-semiring-1-cancel-crossproduct = comm-semiring-1-cancel +
  assumes crossproduct-eq: w * y + x * z = w * z + x * y ←→ w = x ∨ y = z
begin

lemma crossproduct-noteq:
  a ≠ b ∧ c ≠ d ←→ a * c + b * d ≠ a * d + b * c
  by (simp add: crossproduct-eq)

lemma add-scale-eq-noteq:
  r ≠ 0 ⇒ a = b ∧ c ≠ d ⇒ a + r * c ≠ b + r * d
proof (rule notI)
  assume nz: r ≠ 0 and cnd: a = b ∧ c≠d
  and eq: a + (r * c) = b + (r * d)
  have (0 * d) + (r * c) = (0 * c) + (r * d)
    using add-left-imp-eq eq mult-zero-left by (simp add: cnd)
  then show False using crossproduct-eq [of 0 d] nz cnd by simp
qed

lemma add-0-iff:
  b = b + a ←→ a = 0
  using add-left-imp-eq [of b a 0] by auto

end

subclass (in idom) comm-semiring-1-cancel-crossproduct
proof
  fix w x y z
  show w * y + x * z = w * z + x * y ←→ w = x ∨ y = z
  proof
    assume w * y + x * z = w * z + x * y
    then have w * y + x * z - w * z - x * y = 0 by (simp add: algebra-simps)
    then have w * (y - z) - x * (y - z) = 0 by (simp add: algebra-simps)
    then have (y - z) * (w - x) = 0 by (simp add: algebra-simps)
    then have y - z = 0 ∨ w - x = 0 by (rule divisors-zero)
    then show w = x ∨ y = z by auto
  qed (auto simp add: ac-simps)
qed
instance nat :: comm-semiring-1-cancel-crossproduct
proof
  fix w x y z :: nat
  have aux: \( y < z \implies w \cdot y + x \cdot z = w \cdot z + x \cdot y \implies w = x \)
  proof
    fix y z :: nat
    assume y < z then have \( \exists k. z = y + k \land k \neq 0 \) by (intro exI [of z - y])
    then obtain k where z = y + k and k \( \neq 0 \) by blast
    assume w \( \cdot y + x \cdot z = w \cdot z + x \cdot y \) then have \( (w \cdot y + x \cdot y) + x \cdot k = (w \cdot y + x \cdot y) + w \cdot k \) by (simp add: algebra-simps)
    then have x \( \cdot k = w \cdot k \) by simp
    then show w = x using \( k \neq 0 \) by simp
  qed
  show w \( \cdot y + x \cdot z = w \cdot z + x \cdot y \) \( \iff \) w = x \( \lor \) y = z
  by (auto simp add: neq_iff dest!: aux)
  qed

Semiring normalization proper

ML-file ⟨Tools/semiring-normalizer.ML⟩

context comm-semiring-1 begin

lemma semiring-normalization-rules [no-atp]:
  \( (a \cdot m) + (b \cdot m) = (a + b) \cdot m \)
  \( (a \cdot m) + m = (a + 1) \cdot m \)
  \( m + (a \cdot m) = (a + 1) \cdot m \)
  \( m + m = (1 + 1) \cdot m \)
  \( 0 + a = a \)
  \( a + 0 = a \)
  \( a \cdot b = b \cdot a \)
  \( (a + b) \cdot c = (a \cdot c) + (b \cdot c) \)
  \( 0 \cdot a = 0 \)
  \( a \cdot 0 = 0 \)
  \( 1 \cdot a = a \)
  \( a \cdot 1 = a \)
  \( (lx \cdot ly) \cdot (rx \cdot ry) = (lx \cdot rx) \cdot (ly \cdot ry) \)
  \( (lx \cdot ly) \cdot (rx \cdot ry) = lx \cdot (ly \cdot (rx \cdot ry)) \)
  \( (lx \cdot ly) \cdot (rx \cdot ry) = rx \cdot ((lx \cdot ly) \cdot ry) \)
  \( (lx \cdot ly) \cdot rx = (lx \cdot rx) \cdot ly \)
  \( (lx \cdot ly) \cdot rx = lx \cdot (ly \cdot rx) \)
  \( lx \cdot (rx \cdot ry) = (lx \cdot rx) \cdot ry \)
  \( lx \cdot (rx \cdot ry) = rx \cdot (lx \cdot ry) \)
  \( (a + b) + (c + d) = (a + c) + (b + d) \)
  \( (a + b) + c = a + (b + c) \)
  \( a + (c + d) = c + (a + d) \)
  \( (a + b) + c = (a + c) + b \)
THEORY "Semiring-Normalization"

\[ a + c = c + a \]
\[ a + (c + d) = (a + c) + d \]
\[ (x \cdot p) \cdot (x \cdot q) = x \cdot (p \cdot q) \]
\[ x \cdot (x \cdot q) = x \cdot (\text{Suc} q) \]
\[ (x \cdot q) \cdot x = x \cdot (\text{Suc} q) \]
\[ x \cdot x = x^2 \]
\[ (x \cdot y) \cdot q = (x \cdot q) \cdot (y \cdot q) \]
\[ x \cdot p \cdot q = x \cdot (p \cdot q) \]
\[ x \cdot 0 = 1 \]
\[ x \cdot 1 = x \]
\[ x \cdot (y + z) = (x \cdot y) + (x \cdot z) \]
\[ x \cdot (\text{Suc} q) = x \cdot (x \cdot q) \]
\[ x \cdot (2 \cdot n) = (x \cdot n) \cdot (x \cdot n) \]

by (simp-all add: algebra-simps power-add power2-eq-square power-mult-distrib power-mult del: one-add-one)

local-setup ⟨
Semiring-Normalizer.declare @\{thm comm-semiring-1-axioms\}
\{semiring = ([\textbf{term} (x + y), \textbf{term} (x \cdot y), \textbf{term} (x \cdot n), \textbf{term} (0), \textbf{term} (1)],
@\{thms semiring-normalization-rules\}),
ring = ([], []),
field = ([], []),
idom = [],
ideal = []\}
⟩

end

context comm-ring-1
begin

lemma ring-normalization-rules [no-atp]:
\[ -x = (-1) \cdot x \]
\[ x - y = x + (-y) \]
by simp-all

local-setup ⟨
Semiring-Normalizer.declare @\{thm comm-ring-1-axioms\}
\{semiring = ([\textbf{term} (x + y), \textbf{term} (x \cdot y), \textbf{term} (x \cdot n), \textbf{term} (0), \textbf{term} (1)],
@\{thms semiring-normalization-rules\}),
ring = ([\textbf{term} (x - y), \textbf{term} (-x)], @\{thms ring-normalization-rules\}),
field = ([], []),
idom = [],
ideal = []\}
⟩

end

context comm-semiring-1-cancel-crossproduct
begin

local-setup ⟨
Semiring-Normalizer.declare @\{thm comm-semiring-1-cancel-crossproduct-axioms\}
\{semiring = ([\\text{term\,} x + y, \text{term\,} x * y, \text{term\,} x ^ n, \text{term\,} 0, \text{term\,} 1],
  @\{thms semiring-normalization-rules\}),
  ring = ([], []),
  field = ([], []),
  idom = @\{thms crossproduct-noteq add-scale-eq-noteq\},
  ideal = []\}
⟩

end

code-identifier
code-module Semiring-Normalization -> (SML) Arith and (OCaml) Arith and
(Haskell) Arith
end
59  Groebner bases

theory Groebner-Basis
imports Semiring-Normalization Parity
begin

59.1  Groebner Bases

lemmas bool-simps = simp-thms(1-34) — FIXME move to HOL.HOL.

lemma nff-simps: — FIXME shadows fact binding in HOL.HOL.

(P \land Q) = (P \lor \neg Q) (\neg(P \lor Q)) = (\neg P \land \neg Q)
(P \rightarrow Q) = (\neg P \lor Q)
(P = Q) = ((P \land Q) \lor (\neg P \land \neg Q)) (\neg \neg(P)) = P
by blast+

lemma dnf:
(P \land (Q \lor R)) = ((P \land Q) \lor (P \land R))
((Q \lor R) \land P) = ((Q \land P) \lor (R \land P))
(P \land Q) = (Q \land P)
(P \lor Q) = (Q \lor P)
by blast+

lemmas weak-dnf-simps = dnf bool-simps

lemma PFalse:
    P \equiv \text{False} \implies \neg P
    \neg P \implies (P \equiv \text{False})
by auto

named-theorems algebra pre-simplification rules for algebraic methods
ML-file ⟨Tools/groebner.ML⟩

method-setup algebra = ⟨:
let
  fan keyword k = Scan.lift (Args.$$ k -- Args.colon) >> K ()
val addN = add
val delN = del
val any-keyword = keyword addN || keyword delN
val thms = Scan.repeats (Scan.unless any-keyword Attrib.multi-thm);
in
  Scan.optional (keyword addN |-- thms) [] --
  Scan.optional (keyword delN |-- thms) [] >>
  (fn (add-ths, del-ths) => fn ctxt => SIMPLE-METHOD1 (Groebner.algebra-tac add-ths del-ths ctxt))
end

solve polynomial equations over (semi)rings and ideal membership problems using
Groebner bases

declare dvd-def[algebra]
THEORY “Set-Interval”

declare mod-eq-0-iff-dvd[algebra]
declare mod-div-trivial[algebra]
declare mod-mod-trivial[algebra]
declare div-by-0[algebra]
declare mod-by-0[algebra]
declare mult-div-mod-eq[algebra]
declare div-minus-minus[algebra]
declare mod-minus-minus[algebra]
declare div-minus-right[algebra]
declare mod-minus-right[algebra]
declare div-0[algebra]
declare mod-0[algebra]
declare mod-by-1[algebra]
declare div-by-1[algebra]
declare mod-minus1-right[algebra]
declare div-minus1-right[algebra]
declare mod-mult-self2-is-0[algebra]
declare mod-mult-self1-is-0[algebra]
declare zmod-eq-0-iff[algebra]
declare dvd-0-left-iff[algebra]
declare z dvd1-eq[algebra]
declare mod-eq-dvd-iff[algebra]
declare nat-mod-eq-iff[algebra]

context semiring-parity
begin

declare even-mult-iff [algebra]
declare even-power [algebra]
end

context ring-parity
begin

declare even-minus [algebra]
end

declare even-Suc [algebra]
declare even-diff-nat [algebra]
end

60 Set intervals

theory Set-Interval
imports Divides
begin
lemma card-2-iff: \( \text{card } S = 2 \iff (\exists x \ y. \ S = \{x, y\} \land x \neq y) \) 
  by \( \text{(auto simp: card-Suc-eq numeral-eq-Suc)} \)

lemma card-2-iff': \( \text{card } S = 2 \iff (\exists x \in S. \exists y \in S. \ x \neq y \land (\forall z \in S. \ z = x \lor z = y)) \) 
  by \( \text{(auto simp: card-Suc-eq numeral-eq-Suc)} \)

context ord 
begin

definition lessThan :: 'a => 'a set ((1{..<})) where 
\{..<n\} == \{ x. x < n \}

definition atMost :: 'a => 'a set ((1{..}) where 
\{..n\} == \{ x. x \leq n \}

definition greaterThan :: 'a => 'a set ((1{<..}) where 
\{l..<\} == \{ x. l < x \}

definition atLeast :: 'a => 'a set ((1{..}) where 
\{l..\} == \{ x. l \leq x \}

definition greaterThanLessThan :: 'a => 'a => 'a set ((1{<..<}) where 
\{l<..<u\} == \{ l<.. \} \text{ Int } \{..<u\}

definition atLeastLessThan :: 'a => 'a => 'a set ((1{..<}) where 
\{l..<u\} == \{ l..< \} \text{ Int } \{..<u\}

definition greaterThanAtMost :: 'a => 'a => 'a set ((1{<..}) where 
\{l<..u\} == \{ l<.. \} \text{ Int } \{..u\}

definition atLeastAtMost :: 'a => 'a => 'a set ((1{..}) where 
\{l..u\} == \{ l.. \} \text{ Int } \{..u\}

end

A note of warning when using \{..<n\} on type nat: it is equivalent to \{0..<n\} 
but some lemmas involving \{m..<n\} may not exist in \{..<n\}-form as well.
syntax (ASCII)
-UNION-le :: 'a => 'a => 'b set => 'b set  ((\text{UN} \cdot \leq) / -) [0, 0, 10] 10
-UNION-less :: 'a => 'a => 'b set => 'b set  ((\text{UN} \cdot <) / -) [0, 0, 10] 10
-INTER-le :: 'a => 'a => 'b set => 'b set  ((\text{INT} \cdot \leq) / -) [0, 0, 10] 10
-INTER-less :: 'a => 'a => 'b set => 'b set  ((\text{INT} \cdot <) / -) [0, 0, 10] 10

syntax (latex output)
-UNION-le :: 'a => 'a => 'b set => 'b set  ((\text{UN} \bigcup (\text{unbreakable}) \cdot \leq) / -) [0, 0, 10] 10
-UNION-less :: 'a => 'a => 'b set => 'b set  ((\text{UN} \bigcup (\text{unbreakable}) \cdot <) / -) [0, 0, 10] 10
-INTER-le :: 'a => 'a => 'b set => 'b set  ((\text{INT} \bigcap (\text{unbreakable}) \cdot \leq) / -) [0, 0, 10] 10
-INTER-less :: 'a => 'a => 'b set => 'b set  ((\text{INT} \bigcap (\text{unbreakable}) \cdot <) / -) [0, 0, 10] 10

translations
\bigcup i \leq n. A = \bigcup i \in \{..n\}. A
\bigcup i < n. A = \bigcup i \in \{..<n\}. A
\bigcap i \leq n. A = \bigcap i \in \{..n\}. A
\bigcap i < n. A = \bigcap i \in \{..<n\}. A

60.1 Various equivalences

lemma (in ord) lessThan-iff [iff]: (i < lessThan k) = (i<k)
by (simp add: lessThan-def)

lemma Compl-lessThan [simp]:
  !!k: 'a::linorder. -lessThan k = atLeast k
by (auto simp add: lessThan-def atLeast-def)

lemma single-Diff-lessThan [simp]: !!k: 'a::preorder. \{k\} - lessThan k = \{k\}
by auto

lemma (in ord) greaterThan-iff [iff]: (i \in greaterThan k) = (k<i)
by (simp add: greaterThan-def)

lemma Compl-greaterThan [simp]:
  !!k: 'a::linorder. -greaterThan k = atMost k
by (auto simp add: greaterThan-def atMost-def)

lemma Compl-atMost [simp]: !!k: 'a::linorder. -atMost k = greaterThan k
apply (subst Compl-greaterThan [symmetric])
apply (rule double-complement)
done

lemma (in ord) atLeast-iff [iff]: (i ∈ atLeast k) = (k<=>i)
  by (simp add: atLeast-def)

lemma atLeast-empty-triv [simp]: {{}..} = UNIV
  by auto

lemma atMost-empty-triv [simp]: {..<} = UNIV
  by auto

lemma atMost-subset-iff [iff]:
  (atMost x ⊆ atMost y) = (y ≤ (x::'a::preorder))
  by (blast intro: order-trans)

lemma atMost-eq-iff [iff]:
  (atMost x = atMost y) = (x = (y::'a::order))
  by (blast intro: order-antisym order-trans)

lemma greaterThan-subset-iff [iff]:
  (greaterThan x ⊆ greaterThan y) = (y ≤ (x::'a::linorder))
  unfolding greaterThan-def by (auto simp: linorder-not-less [symmetric])

lemma greaterThan-eq-iff [iff]:
  (greaterThan x = greaterThan y) = (x = (y::'a::linorder))
  by (auto simp: elim!: equalityE)

lemma atMost-subset-iff [iff]: (atMost x ⊆ atMost y) = (x ≤ (y::'a::preorder))
  by (blast intro: order-trans)
THEORY "Set-Interval"

lemma atMost-eq-iff [iff]: (atMost x = atMost y) = (x = (y::'a::order))
by (blast intro: order-antisym order-trans)

lemma lessThan-subset-iff [iff]:
(lessThan x ⊆ lessThan y) = (x ≤ (y::'a::linorder))
unfolding lessThan-def by (auto simp: linorder-not-less [symmetric])

lemma lessThan-eq-iff [iff]:
(lessThan x = lessThan y) = (x = (y::'a::linorder))
by (auto simp: elim!: equalityE)

lemma lessThan-strict-subset-iff:
fixes m n :: 'a::linorder
shows {..<m} <..< n ←→ m < n
by (metis leD lessThan-subset-iff linorder-linear not-less-iff-gr-or-eq psubset-eq)

lemma (in linorder) Ici-subset-Ioi-iff: {a ..} ⊆ {b <..<} ←→ b < a
by auto

lemma (in linorder) Iic-subset-Iio-iff: {..<a} ⊆ {..<b} ←→ a < b
by auto

lemma (in preorder) Ioi-le-Ico: {a <..<} ⊆ {a ..}
by (auto intro: less-imp-le)

60.3 Two-sided intervals

context ord
begin

lemma greaterThanLessThan-eq: {..<a <..< b} = {a ..} ∩ {..<b}
by auto

lemma (in order) atLeastLessThan-eq-atLeastAtMost-diff:

\{a..<b\} = \{a..b\} - \{b\}
by (auto simp add: atLeastLessThan-def atLeastAtMost-def)

lemma (in order) greaterThanAtMost-eq-atLeastAtMost-diff:
\{a..<b\} = \{a..b\} - \{a\}
by (auto simp add: greaterThanAtMost-def atLeastAtMost-def)
end

60.3.1 Emptyness, singletons, subset

context preorder
begin

lemma atLeastMost-empty-iff[simp]:
\{a..b\} = {} \iff \neg a \leq b
by auto (blast intro: order-trans)

lemma atLeastMost-empty-iff2[simp]:
\{\} = \{a..b\} \iff \neg a \leq b
by auto (blast intro: order-trans)

lemma atLeastLessThan-empty-iff[simp]:
\{a..<b\} = {} \iff \neg a < b
by auto (blast intro: le-less-trans)

lemma atLeastLessThan-empty-iff2[simp]:
\{\} = \{a..<b\} \iff \neg a < b
by auto (blast intro: le-less-trans)

lemma greaterThanAtMost-empty-iff[simp]:
\{k..<l\} = {} \iff \neg k < l
by auto (blast intro: less-le-trans)

lemma greaterThanAtMost-empty-iff2[simp]:
\{\} = \{k..<l\} \iff \neg k < l
by auto (blast intro: less-le-trans)

lemma atLeastMost-subset-iff[simp]:
\{a..b\} \subseteq \{c..d\} \iff \neg a \leq b \lor c \leq a \land b \leq d
unfolding atLeastAtMost-def atLeast-def atMost-def
by (blast intro: order-trans)

lemma atLeastMost-psubset-iff:
\{a..b\} \subset \{c..d\} \iff
\((\neg a \leq b) \lor c \leq a \land b \leq d \land (c < a \lor b < d)) \land c \leq d
by (simp add: psubset-eq set-eq-iff less-le-not-le)(blast intro: order-trans)

lemma atLeastAtMost-subseteq-atLeastLessThan-iff:
\{a..b\} \subseteq \{c..< d\} \iff (a \leq b \longrightarrow c \leq a \land b < d)
by auto (blast intro: local.order-trans local.le-less-trans elim: )+
lemma Icc-subset-Ici-iff [simp]:
\{l..h\} ⊆ \{l'..\} = (∃ l≤h \lor l≥l')
by (auto simp: subset-eq intro: order-trans)

lemma Icc-subset-Iic-iff [simp]:
\{l..h\} ⊆ \{..h'\} = (∃ l≤h \lor h≤h')
by (auto simp: subset-eq intro: order-trans)

lemma not-Ici-eq-empty [simp]: \{l..\} ≠ \{
by (auto simp: set-eq-iff)

lemma not-Iic-eq-empty [simp]: \{..h\} ≠ \{
by (auto simp: set-eq-iff)


lemmas not-empty-eq-Iic-eq-empty [simp] = not-Iic-eq-empty [symmetric]

end

context order

begin

lemma atLeastAtMost-empty [simp]:
b < a =⇒ \{a..b\} = \{
by (auto simp: atLeastAtMost-def atMost-def atLeast-def)

lemma atLeastLessThan-empty [simp]:
b ≤ a =⇒ \{a..<b\} = \{
by (auto simp: atLeastLessThan-def)

lemma greaterThanAtMost-empty [simp]: l ≤ k =⇒ \{k..<l\} = \{
by (auto simp: greaterThanAtMost-empty_def greaterThan-def atMost-def)

lemma greaterThanLessThan-empty [simp]: l ≤ k =⇒ \{k..<l\} = \{
by (auto simp: greaterThanLessThan-empty_def greaterThan-def lessThan-def)

lemma atLeastAtMost-singleton [simp]: \{a..a\} = \{
by (auto simp add: atLeastAtMost-def atMost-def atLeast-def)

lemma atLeastAtMost-singleton': a = b =⇒ \{a .. b\} = \{
by simp

lemma Icc-eq-Icc [simp]:
\{l..h\} = \{l'..h'\} = (l = l' \land h = h' \lor \neg l \leq h \land \neg l' \leq h')
by (simp add: order-class.eq_iff) (auto intro: order-trans)

lemma atLeastAtMost-singleton-iff [simp]:
\{a .. b\} = \{c\} =⇒ a = b \land b = c

proof
assume \{a..b\} = \{c\}
hence \(\neg (\neg a \leq b)\) unfolding atLeastMost-empty-iff[symmetric] by simp
with \{a..b\} = \{c\} have \(c \leq a \land b \leq c\) by auto
with * show \(a = b \land b = c\) by auto
qed simp

end

context no-top
begin

lemma not-UNIV-le-Icc[simp]: \(\neg UNIV \subseteq \{l..h\}\)
using gt-ex[of h]
by (auto simp: subset-eq less-le-not-le)

lemma not-UNIV-le-Iic[simp]: \(\neg UNIV \subseteq \{..h\}\)
using gt-ex[of h]
by (auto simp: subset-eq less-le-not-le)

lemma not-Ici-le-Icc[simp]: \(\neg \{l..\} \subseteq \{l'..h'\}\)
using gt-ex[of h']
by (auto simp: subset-eq less-le
(blast dest:antisym-conv intro: order-trans))

lemma not-Ici-le-Iic[simp]: \(\neg \{l..\} \subseteq \{..h'\}\)
using gt-ex[of h']
by (auto simp: subset-eq less-le
(antisym-conv intro: order-trans))

end

context no-bot
begin

lemma not-UNIV-le-Ici[simp]: \(\neg UNIV \subseteq \{l..\}\)
using lt-ex[of l]
by (auto simp: subset-eq less-le-not-le)

lemma not-Ici-le-Icc[simp]: \(\neg \{..h\} \subseteq \{l'..h'\}\)
using lt-ex[of l']
by (auto simp: subset-eq less-le
(antisym-conv intro: order-trans))

lemma not-Ici-le-Ici[simp]: \(\neg \{..h\} \subseteq \{l'..\}\)
using lt-ex[of l']
by (auto simp: subset-eq less-le
(antisym-conv intro: order-trans))

end

context no-top
begin
lemma not-UNIV-eq-Icc[simp]: \( \neg \text{UNIV} = \{l'..h'\} \)
using gt-ex[of h'] by (auto simp: set-eq-iff less-le-not-le)

lemmas not-Icc-eq-UNIV[simp] = not-UNIV-eq-Icc[symmetric]

lemma not-UNIV-eq-Iic[simp]: \( \neg \text{UNIV} = \{..h'\} \)
using gt-ex[of h'] by (auto simp: set-eq-iff less-le-not-le)

lemmas not-Iic-eq-UNIV[simp] = not-UNIV-eq-Iic[symmetric]

lemma not-Icc-eq-Iic[simp]: \( \neg \{l..h\} = \{l'..\} \)
unfolding atLeastAtMost-def using not-Ici-le-Iic[of l'] by blast

lemmas not-Ici-eq-Icc[simp] = not-Icc-eq-Iic[symmetric]

lemma not-Ici-eq-Ici[simp]: \( \neg \{l..\} = \{l'..\} \)
using not-Ici-le-Iic[of l' h] by blast

lemmas not-Iic-eq-Ici[simp] = not-Ici-eq-Iic[symmetric]

end

context no-bot
begin

lemma not-UNIV-eq-Ici[simp]: \( \neg \text{UNIV} = \{l'..\} \)
using lt-ex[of a] by (auto simp: set-eq-iff less-le-not-le)

lemmas not-Ici-eq-UNIV[simp] = not-UNIV-eq-Ici[symmetric]

lemma not-Icc-eq-Iic[simp]: \( \neg \{l..h\} = \{..h'\} \)
unfolding atLeastAtMost-def using not-Ici-le-Iic[of h'] by blast

lemmas not-Iic-eq-Icc[simp] = not-Icc-eq-Iic[symmetric]

end

context dense-linorder
begin

lemma greaterThanLessThan-empty-iff[simp]:
\( \{a..<<b\} = \{} \leftrightarrow b \leq a \)
using dense[of a b] by (cases a < b) auto

lemma greaterThanLessThan-empty-iff2[simp]:
\( \{} = \{a..<<b\} \leftrightarrow b \leq a \)
using dense[of a b] by (cases a < b) auto
lemma atLeastLessThan-subseteq-atLeastAtMost-iff:
{a ..< b} ⊆ {c .. d} ←→ (a < b → c ≤ a ∧ b ≤ d)
using dense[of max a d b]
by (force simp: subset-eq Ball-def not-less[symmetric])

lemma greaterThanAtMost-subseteq-atLeastAtMost-iff:
{a <.. b} ⊆ {c .. d} ←→ (a < b → c ≤ a ∧ b ≤ d)
using dense[of a min c b]
by (force simp: subset-eq Ball-def not-less[symmetric])

lemma greaterThanLessThan-subseteq-atLeastAtMost-iff:
{a <..< b} ⊆ {c ..< d} ←→ (a < b −→ c ≤ a ∧ b ≤ d)
using dense[of a min c b] dense[of max a d b]
by (force simp: subset-eq Ball-def not-less[symmetric])

lemma greaterThanLessThan-subseteq-greaterThanLessThan:
{a <..< b} ⊆ {c ..< d} ←→ (a < b −→ a ≥ c ∧ b ≤ d)
using dense[of a min c b] dense[of max a d b]
by (force simp: subset-eq Ball-def not-less[symmetric])

lemma greaterThanLessThan-subseteq-greaterThanAtMost:
{a <..< b} ⊆ {c <..< d} ←→ (a < b −→ c ≤ a ∧ b ≤ d)
using dense[of a min c b] dense[of max a d b]
by (force simp: subset-eq Ball-def not-less[symmetric])

end

context no-top
begin

lemma greaterThan-non-empty[simp]: {x <..} ≠ {}
using gt-ex[of x] by auto

end

context no-bot
begin
lemma lessThan-non-empty [simp]: \{..<x\} \neq \{\}
using lt-ex[of x] by auto

end

lemma (in linorder) atLeastLessThan-subset-iff:
\{a..<b\} \subseteq \{c..<d\} \implies b \leq a \lor c \leq a \land b \leq d
apply (auto simp: subset-eq Ball-def not-le)
apply (erule_tac x=a in spec)
apply (erule_tac x=d in allE)
apply (auto simp: )
done

lemma atLeastLessThan-inj:
fixes a b c d :: 'a::linorder
assumes eq: \{a..<b\} = \{c..<d\} and a < b c < d
shows a = c b = d
using assms by (metis atLeastLessThan-subset-iff eq less-le-not-le antisym-conv2 subset-refl)

lemma atLeastLessThan-eq-iff:
fixes a b c d :: 'a::linorder
assumes a < b c < d
shows \{a..<b\} = \{c..<d\} \longleftrightarrow a = c \land b = d
using atLeastLessThan-inj assms by auto

lemma (in order) Ioc-subset-iff:
\{a..<b\} \subseteq \{c..<d\} \longleftrightarrow (b \leq a \lor c \leq a \land b \leq d)
by (auto simp: subset-eq Ball-def)

lemma (in linorder) Ioc-inj:
\{a..<b\} = \{c..<d\} \longleftrightarrow (b \leq a \land c \leq a \land b \leq d)
by (metis eq-iff greaterThanAtMost-empty-iff2 greaterThanAtMost-iff le-cases not-le)

lemma (in bounded-lattice) atLeastAtMost-eq-UNIV-iff:
\{x..\} = UNIV \longleftrightarrow x = bot
by (auto simp: Leq-atLeastAtMost intro: le-bot)

lemma (in order-bot) atLeast-eq-empty-iff:
\{..\} = UNIV \longleftrightarrow x = bot
by (auto simp: set-eq intro: le-bot)

lemma (in order-top) atMost-eq-UNIV-iff:
\{..x\} = UNIV \longleftrightarrow x = top
by (auto simp: set-eq intro: top-le)

lemma (in bounded-lattice) atLeastAtMost-eq-UNIV-iff:
\{x..y\} = UNIV \longleftrightarrow (x = bot \land y = top)
by (auto simp: set-eq intro: top-le)

lemma atMost-eq-empty-iff:
\{..<n::'a::linorder, order-bot\} = \{} \longleftrightarrow n = bot
THEORY "Set-Interval"

by (auto simp: set-eq-iff not-less le-bot)

lemma lessThan-empty-iff: {..< n::nat} = { } ⟷ n = 0
by (simp add: Iio-eq-empty-iff bot-nat-def)

lemma mono-image-least:
assumes f-mono: mono f and f-img: f ' {m ..< n} = {m' ..< n'} m < n
shows f m = m'
proof
from f-img have {m' ..< n'} ≠ { }
  by (metis atLeastLessThan-empty-iff image-is-empty)
with f-img have m' ∈ f ' {m ..< n} by auto
then obtain k where f k = m' m ≤ k by auto
moreover have m' ≤ f m using f-img by auto
ultimately show f m = m'
  using f-mono by (auto elim: monoE[where x=m and y=k])
qed

60.4 Infinite intervals

context dense-linorder
begin

lemma infinite-Ioo:
assumes a < b
shows ¬ finite {a..<b}
proof
assumefin: finite {a..<b}
moreover have ne: {a..<b} ≠ { }
  using (a < b) by auto
ultimately have a < Max {a ..< b} Max {a ..< b} < b
  using Max-in[of {a ..< b}] by auto
then obtain x where Max {a ..< b} < x x < b
  using dense[of Max {a ..< b} b] by auto
then have x ∈ {a ..< b}
  using (a < Max {a ..< b}) by auto
then have x ≤ Max {a ..< b}
  using fin by auto
with ⟨Max {a ..< b} < x⟩ show False by auto
qed

lemma infinite-Icc: a < b ⟹ ¬ finite {a .. b}
using greaterThanLessThan-subseteq-atLeastAtMost-iff[of a b a b] infinite-Ioo[of a b]
by (auto dest: finite-subset)

lemma infinite-Ico: a < b ⟹ ¬ finite {a ..< b}
using greaterThanLessThan-subseteq-atLeastLessThan-iff[of a b a b] infinite-Ioo[of a b]
THEORY "Set-Interval"

by (auto dest: finite-subset)

lemma infinite-Ioc: a < b ⟷ ¬ finite {a..< b}
using greaterThanLessThan-subseteq-greaterThanAtMost-iff[of a b a b] infinite-Ioo[of a b]
by (auto dest: finite-subset)

lemma infinite-Ioo-iff [simp]: infinite {a..<b} ⟷ a < b
using not-less-iff-gr-or-eq by (fastforce simp: infinite-Ioo)

lemma infinite-Icc-iff [simp]: infinite {a..b} ⟷ a < b
using not-less-iff-gr-or-eq by (fastforce simp: infinite-Icc)

lemma infinite-Ico-iff [simp]: infinite {a..<b} ⟷ a < b
using not-less-iff-gr-or-eq by (fastforce simp: infinite-Ico)

lemma infinite-Ioc-iff [simp]: infinite {a..<b} ⟷ a < b
using not-less-iff-gr-or-eq by (fastforce simp: infinite-Ioc)

end

lemma infinite-Iio: ¬ finite {..< a :: 'a :: {no-bot, linorder}}
proof
assume finite {..< a}
then have *: ∀x. x < a ⟷ Min {..< a} ≤ x
  by auto
obtain x where x < a
  using lt-ex by auto
obtain y where y < Min {..< a}
  using lt-ex by auto
also have Min {..< a} ≤ x
  using (x < a) by fact
also note (x < a)
finally have Min {..< a} ≤ y
  by fact
with ⟨y < Min {..< a}⟩ show False by auto
qed

lemma infinite-Iic: ¬ finite {.. a :: 'a :: {no-bot, linorder}}
using infinite-Iio[of a] finite-subset[of {..< a} {.. a}]
by (auto simp: subset-eq less-imp-le)

lemma infinite-Ioi: ¬ finite {a :: 'a :: {no-top, linorder} <..}
proof
assume finite {a <..}
then have *: ∀x. a < x ⟷ x ≤ Max {a <..}
  by auto
obtain \( y \) where \( \text{Max} \{a <..\} < y \)
using \texttt{gt-ex} by \texttt{auto}

obtain \( x \) where \( x: a < x \)
using \texttt{gt-ex} by \texttt{auto}
also from \( x \) have \( x \leq \text{Max} \{a <..\} \)
by \texttt{fact}
also note \( (\text{Max} \{a <..\} < y) \)
finally have \( y \leq \text{Max} \{a <..\} \)
by \texttt{fact}
with \( (\text{Max} \{a <..\} < y) \) show False by \texttt{auto}

\[
\text{lemma} \hspace{1em} \text{infinite-Ici} : \neg \hspace{1em} \text{finite} \{a :: 'a :: \{\text{no-top, linorder} \} ..\}
\hspace{1em} \text{using} \hspace{1em} \text{infinite-Ioi[of a] finite-subset[of \{a <..\} \{a ..\}]} \\
\hspace{1em} \text{by} \hspace{1em} (\text{auto simp: subset-eq less-imp-le})
\]

\textbf{60.4.1 Intersection}

\textbf{context} \texttt{linorder}
\begin{verbatim}
begin

\textbf{lemma} \texttt{Int-atLeastAtMost[simp]:} \{a..b\} Int \{c..d\} = \{max a c .. min b d\}
\hspace{1em} by \texttt{auto}

\textbf{lemma} \texttt{Int-atLeastAtMostR1[simp]:} \{..b\} Int \{c..d\} = \{c .. min b d\}
\hspace{1em} by \texttt{auto}

\textbf{lemma} \texttt{Int-atLeastAtMostR2[simp]:} \{a..\} Int \{c..d\} = \{max a c .. d\}
\hspace{1em} by \texttt{auto}

\textbf{lemma} \texttt{Int-atLeastAtMostL1[simp]:} \{a..b\} Int \{..d\} = \{a .. min b d\}
\hspace{1em} by \texttt{auto}

\textbf{lemma} \texttt{Int-atLeastAtMostL2[simp]:} \{a..b\} Int \{..\} = \{max a c .. b\}
\hspace{1em} by \texttt{auto}

\textbf{lemma} \texttt{Int-atLeastLessThan[simp]:} \{a..<b\} Int \{c..<d\} = \{max a c ..< min b d\}
\hspace{1em} by \texttt{auto}

\textbf{lemma} \texttt{Int-greaterThanAtMost[simp]:} \{a..<b\} Int \{c..<d\} = \{max a c <.. min b d\}
\hspace{1em} by \texttt{auto}

\textbf{lemma} \texttt{Int-greaterThanLessThan[simp]:} \{a..<b\} Int \{c..<d\} = \{max a c <..< min b d\}
\hspace{1em} by \texttt{auto}

\textbf{lemma} \texttt{Int-atMost[simp]:} \{..a\} \cap \{..b\} = \{.. min a b\}
\end{verbatim}
THEORY "Set-Interval"

by (auto simp: min-def)

lemma Ioc-disjoint: \{a<..<b\} ∩ \{c<..<d\} = {} ←→ b ≤ a ∨ d ≤ c ∨ b ≤ c ∨ d ≤ a
  by auto
end

context complete-lattice

begin

lemma shows Sup-atLeast[simp]: Sup \{x ..\} = top
  and Sup-greaterThanAtLeast[simp]: x < top ⇒ Sup \{x ..\} = top
  and Sup-atMost[simp]: Sup \{.. y\} = y
  and Sup-atLeastAtMost[simp]: x ≤ y ⇒ Sup \{x .. y\} = y
  and Sup-greaterThanAtMost[simp]: x < y ⇒ Sup \{x .. y\} = y
by (auto intro!: Sup-eqI)

lemma shows Inf-atMost[simp]: Inf \{.. x\} = bot
  and Inf-atMostLessThan[simp]: top < x ⇒ Inf \{..< x\} = bot
  and Inf-atLeast[simp]: Inf \{x ..\} = x
  and Inf-atLeastAtMost[simp]: x ≤ y ⇒ Inf \{x .. y\} = x
  and Inf-atLeastLessThan[simp]: x < y ⇒ Inf \{x ..< y\} = x
by (auto intro!: Inf-eqI Sup-eqI intro!: dense-le dense-le-bounded dense-ge dense-ge-bounded)

end

lemma fixes x y :: 'a :: {complete-lattice, dense-linearorder}
  shows Sup-lessThan[simp]: Sup \{..< y\} = y
    and Sup-atLeastLessThan[simp]: x < y ⇒ Sup \{x ..< y\} = y
    and Sup-greaterThanLessThan[simp]: x < y ⇒ Sup \{x <..< y\} = y
    and Inf-greaterThan[simp]: Inf \{x ..\} = x
    and Inf-greaterThanAtMost[simp]: x < y ⇒ Inf \{x .. y\} = x
    and Inf-greaterThanLessThan[simp]: x < y ⇒ Inf \{x <..< y\} = x
by (auto intro!: Inf-eqI Sup-eqI intro: dense-le dense-le-bounded dense-ge dense-ge-bounded)

60.5 Intervals of natural numbers

60.5.1 The Constant lessThan

lemma lessThan-0 [simp]: lessThan (0::nat) = {}
by (simp add: lessThan-def)

lemma lessThan-Suc: lessThan (Suc k) = insert k (lessThan k)
by (simp add: lessThan-def less-Suc-eq, blast)

The following proof is convenient in induction proofs where new elements
get indices at the beginning. So it is used to transform \{..<Suc n\} to 0 and \{..<n\}.

\begin{proof}

\begin{lemma}
  \textit{zero-notin-Suc-image} \quad \simp: \; 0 \notin Suc ' A
  \begin{proof}
    \textit{auto}
  \end{proof}
\end{lemma}

\begin{lemma}
  \textit{lessThan-Suc-eq-insert-0} \quad \{..<Suc n\} = insert 0 (Suc ' \{..<n\})
  \begin{proof}
    \textit{(auto simp: image-iff less-eq-0-disj)}
  \end{proof}
\end{lemma}

\begin{lemma}
  \textit{lessThan-Suc-atMost} \quad lessThan (Suc k) = atMost k
  \begin{proof}
    \textit{(simp add: lessThan-def atMost-def less-Suc-eq-le)}
  \end{proof}
\end{lemma}

\begin{lemma}
  \textit{atMost-Suc-eq-insert-0} \quad \{.. Suc n\} = insert 0 (Suc ' \{.. n\})
  \begin{proof}
    \textit{unfolding \textit{lessThan-Suc-atMost}[symmetric] \textit{lessThan-Suc-eq-insert-0}[of Suc n] ..}
  \end{proof}
\end{lemma}

\begin{lemma}
  \textit{UN-lessThan-UNIV}: \(\bigcup m::\text{nat}. \text{lessThan m} = UNIV\)
  \begin{proof}
    \textit{by blast}
  \end{proof}
\end{lemma}

\subsection{The Constant \textit{greaterThan}}

\begin{lemma}
  \textit{greaterThan-0} \quad \textit{greaterThan 0 = range Suc}
  \begin{proof}
    \textit{unfolding \textit{greaterThan-def}}
    \begin{proof}
      \textit{by \textit{blast dest: gr0-conv-Suc \[THEN iffD1\]}}
    \end{proof}
  \end{proof}
\end{lemma}

\begin{lemma}
  \textit{greaterThan-Suc} \quad \textit{greaterThan (Suc k) = greaterThan k - \{Suc k\}}
  \begin{proof}
    \textit{unfolding \textit{greaterThan-def}}
    \begin{proof}
      \textit{by \textit{auto elim: linorder-neqE}}
    \end{proof}
  \end{proof}
\end{lemma}

\begin{lemma}
  \textit{INT-greaterThan-UNIV}: \(\bigcap m::\text{nat}. \text{greaterThan m} = \{\}\)
  \begin{proof}
    \textit{by blast}
  \end{proof}
\end{lemma}

\subsection{The Constant \textit{atLeast}}

\begin{lemma}
  \textit{atLeast-0} \quad \textit{atLeast (0::nat) = UNIV}
  \begin{proof}
    \textit{(unfold atLeast-def UNIV-def, simp)}
  \end{proof}
\end{lemma}

\begin{lemma}
  \textit{atLeast-Suc} \quad \textit{atLeast (Suc k) = atLeast k - \{k\}}
  \begin{proof}
    \textit{unfolding \textit{atLeast-def by \textit{auto simp: order-le-less Suc-le-eq}}}
  \end{proof}
\end{lemma}

\begin{lemma}
  \textit{atLeast-Suc-greaterThan} \quad \textit{atLeast (Suc k) = greaterThan k}
  \begin{proof}
    \textit{by \textit{(auto simp add: greaterThan-def atLeast-def less-Suc-eq-le)}}
  \end{proof}
\end{lemma}

\begin{lemma}
  \textit{UN-atLeast-UNIV}: \(\bigcup m::\text{nat}. \text{atLeast m} = UNIV\)
  \begin{proof}
    \textit{by blast}
  \end{proof}
\end{lemma}

\subsection{The Constant \textit{atMost}}

\begin{lemma}
  \textit{atMost-0} \quad \textit{atMost (0::nat) = \{0\}}
  \begin{proof}
    \textit{(simp add: atMost-def)}
  \end{proof}
\end{lemma}
**THEORY “Set-Interval”**

** lemma atMost-Suc: atMost (Suc k) = insert (Suc k) (atMost k) **

unfolding atMost-def by (auto simp add: less-Suc-eq order-le-less)

** lemma UN-atMost-UNIV: (∪ m::nat. atMost m) = UNIV **

by blast

60.5.5 The Constant atLeastLessThan

The orientation of the following 2 rules is tricky. The lhs is defined in terms of the rhs. Hence the chosen orientation makes sense in this theory — the reverse orientation complicates proofs (eg nontermination). But outside, when the definition of the lhs is rarely used, the opposite orientation seems preferable because it reduces a specific concept to a more general one.

** lemma atLeast0LessThan [code-abbrev]: {0::nat..<n} = {..<n} **

by (simp add:lessThan-def atLeastLessThan-def)

** lemma atLeast0AtMost [code-abbrev]: {0..n::nat} = {..n} **

by (simp add:atMost-def atLeastAtMost-def)

** lemma lessThan-atLeast0: [..<n] = {0::nat..<n} **

by (simp add: atLeast0LessThan)

** lemma atMost-atLeast0: {..n} = {0::nat..n} **

by (simp add: atLeast0AtMost)

** lemma atLeastLessThan0: {m..<0::nat} = {} **

by (simp add: atLeastLessThan-def)

** lemma atLeast0-lessThan-Suc: {0..<Suc n} = insert n {0..<n} **

by (simp add: atLeast0LessThan lessThan-Suc)

** lemma atLeast0-lessThan-Suc-eq-insert-0: {0..<Suc n} = insert 0 (Suc ‘ {0..<n}) **

by (simp add: atLeast0LessThan lessThan-Suc-eq-insert-0)

60.5.6 The Constant atLeastAtMost

** lemma Icc-eq-insert-lb-nat: m ≤ n ⇒ {m..n} = insert m {Suc m..n} **

by auto

** lemma atLeast0-atMost-Suc: **

{0..Suc n} = insert (Suc n) {0..n} **

by (simp add: atLeast0AtMost atMost-Suc)

** lemma atLeast0-atMost-Suc-eq-insert-0: **

{0..Suc n} = insert 0 (Suc ‘ {0..n}) **

by (simp add: atLeast0AtMost atMost-Suc-eq-insert-0)
60.5.7 Intervals of \textit{nats} with \texttt{Suc}

Not a simp rule because the RHS is too messy.

\texttt{lemma atLeastLessThanSuc:}
\begin{verbatim}
{m..<Suc n} = (if m ≤ n then insert n {m..<n} else { })
\end{verbatim}
\texttt{by (auto simp add: atLeastLessThan-def)}

\texttt{lemma atLeastLessThan-singleton [simp]: {m..<Suc m} = {m}}
\texttt{by (auto simp add: atLeastLessThan-def)}

\texttt{lemma atLeastLessThanSuc-atLeastAtMost:}
\begin{verbatim}
{l..< Suc u} = {l..< u}
\end{verbatim}
\texttt{by (simp add: lessThan-Suc-atMost atLeastAtMost-def atLeastLessThan-def)}

\texttt{lemma atLeastSucAtMost-greaterThanAtMost:}
\begin{verbatim}
{Suc l..< u} = {l..<..< u}
\end{verbatim}
\texttt{by (simp add: atLeast-Suc-greaterThan atLeastAtMost-def greaterThanAtMost-def)}

\texttt{lemma atLeastSucLessThan-greaterThanLessThan:}
\begin{verbatim}
{Suc l..< u} = {l..<..< u}
\end{verbatim}
\texttt{by (simp add: atLeast-Suc-greaterThan atLeastLessThan-def greaterThanLessThan-def)}

\texttt{lemma atLeastAtMostSuc-conv: m ≤ Suc n =⇒ {m..<Suc n} = insert (Suc n) {m..<n}}
\texttt{by (auto simp add: atLeastAtMost-def)}

\texttt{lemma atLeastAtMost-insertL: m ≤ n =⇒ insert m {Suc m..<n} = {m..<n}}
\texttt{by auto}

The analogous result is useful on \texttt{int}:

\texttt{lemma atLeastAtMostPlus1-int-conv:}
\begin{verbatim}
m ≤ 1+n =⇒ {m..<1+n} = insert (Suc n) {m..<n::int}
\end{verbatim}
\texttt{by (auto intro: set-eqI)}

\texttt{lemma atLeastLessThan-add-Un: i ≤ j =⇒ {i..<j+k} = {i..<j} ∪ {j..<j+k::nat}}
\texttt{by (induct k) (simp-all add: atLeastLessThanSuc)}

60.5.8 Intervals and numerals

\texttt{lemma lessThan-nat-numeral: — Evaluation for specific numerals}
\begin{verbatim}
lessThan (numeral k :: nat) = insert (pred-numeral k) (lessThan (pred-numeral k))
\end{verbatim}
\texttt{by (simp add: numeral-eq-Suc lessThan-Suc)}

\texttt{lemma atMost-nat-numeral: — Evaluation for specific numerals}
\begin{verbatim}
atMost (numeral k :: nat) = insert (numeral k) (atMost (pred-numeral k))
\end{verbatim}
\texttt{by (simp add: numeral-eq-Suc atMost-Suc)}

\texttt{lemma atLeastLessThan-nat-numeral: — Evaluation for specific numerals}
\begin{verbatim}
atLeastLessThan m (numeral k :: nat) =
\end{verbatim}
(if $m \leq (\text{pred-numeral } k)$ then insert (pred-numeral $k$) (atLeastLessThan $m$
(pred-numeral $k$))
else {})
by (simp add: numeral-eq-Suc atLeastLessThanSuc)

60.5.9 Image

context linordered-semidom

begin

lemma image-add-atLeast[simp]: plus $k$ \{i..\} = \{k + i..\}
proof
  have $n = k + (n - k)$ if $i + k \leq n$ for $n$
  proof
    have $n = (n - (k + i)) + (k + i)$ using that
    then show $n = k + (n - k)$
      by (metis add-commute le-add-diff-inverse)
  qed
  then show \(?thesis
    by (fastforce simp: add-le-imp-le-diff add.commute)
qed

lemma image-add-atLeastAtMost [simp]:
  plus $k$ \{i..j\} = \{i + k..j + k\} (is \(?A = \(?B
proof
  show \(?A \subseteq \(?B
    by (auto simp add: ac-simps)
  next
  show \(?B \subseteq \(?A
    proof
      fix $n$
      assume $n \in \(?B$
      then have $i \leq n - k$
        by (simp add: add-le-imp-le-diff)
      have $n = n - k + k$
      proof
        from \(<n \in \(?B: have $n = n - (i + k) + (i + k)$
          by simp
        also have \(\ldots = n - k - i + i + k$
          by (simp add: algebra-simps)
        also have \(\ldots = n - k + k$
          using $i \leq n - k$ by simp
        finally show \(?thesis
      qed
    moreover have $n - k \in \{i..\}$
      using \(<n \in \(?B:
        by (auto simp: add-le-imp-le-diff add-le-add-imp-diff-le)
  ultimately show $n \in \(?A

END
lemma \textit{image-add-atLeastAtMost}' [simp]:
\[(\lambda n. n + k) \cdot \{i..j\} = \{i + k..j + k\}\]
by (simp add: \textit{add.commute} \([of - k]\))

lemma \textit{image-add-atLeastLessThan} [simp]:
\[\text{plus } k \cdot \{i..<j\} = \{i + k..<j + k\}\]
by (simp add: image-set-diff atLeastLessThan-eq-atLeastAtMost-diff ac-simps)

lemma \textit{image-add-greaterThanAtMost}' [simp]:
\[(\lambda n. n + k) \cdot \{a..<b\} = \{-x..<c + b\}\]
by (simp add: image-set-diff greaterThanAtMost-eq-atLeastAtMost-diff ac-simps)

end

context ordered-ab-group-add
begin

lemma fixes \(x::'a\)
shows \textit{image-uminus-greaterThan} [simp]: \textit{uminus} \(\cdot \{x..<\} = \{-x..<\}\)
and \textit{image-uminus-atLeast} [simp]: \textit{uminus} \(\cdot \{..x\} = \{-x..\}\)
proof safe
  fix \(y\) assume \(y < -x\)
hence \(*\): \(x < -y\) using neg-less-iff-less[of \(-y\) \(x\)] by simp
  have \(-(-y) \in \textit{uminus} \cdot \{x..<\}\)
  by (rule \textit{imageI}) (simp add: \(*\))
  thus \(y \in \textit{uminus} \cdot \{x..<\}\) by simp
next
  fix \(y\) assume \(y \leq -x\)
  have \(-(-y) \in \textit{uminus} \cdot \{..x\}\)
  by (rule \textit{imageI}) (insert \(y \leq -x\)[THEN le-imp-neg-le], simp)
  thus \(y \in \textit{uminus} \cdot \{..x\}\) by simp
qed simp-all

lemma fixes \(x::'a\)
shows \textit{image-uminus-lessThan} [simp]: \textit{uminus} \(\cdot \{..<x\} = \{-x..<\}\)
and \textit{image-uminus-atMost} [simp]: \textit{uminus} \(\cdot \{..x\} = \{-x..\}\)
proof
  have \textit{uminus} \(\cdot \{..<x\} = \textit{uminus} \cdot \{\neg x..<\}\)
  and \textit{uminus} \(\cdot \{..x\} = \textit{uminus} \cdot \{\neg x..\}\) by simp-all
  thus \textit{uminus} \(\cdot \{..<x\} = \{-x..<\}\) and \textit{uminus} \(\cdot \{..x\} = \{-x..\}\)
by (simp-all add: image-image
del: image-uminus-greaterThan image-uminus-atLeast)

qed

lemma
fixes x :: 'a
shows image-uminus-atLeastAtMost [simp]: uminus ' {x..y} = {-y..-x}
and image-uminus-greaterThanAtMost [simp]: uminus ' {x..<y} = {-y..<-x}
and image-uminus-greaterThanLessThan [simp]: uminus ' {x..<y} = {-y..<-x}
by (simp-all add: atLeastAtMost-def greaterThanAtMost-def atLeastLessThan-def
  greaterThanLessThan-def image-Int [OF inj-uminus] Int-commute)

lemma image-add-atMost [simp]: (+) c ' {..a} = {..c + a}
by (auto intro!: image-eqI [where x=x - c for x] simp: algebra-simps)

end

lemma image-Suc-atLeastAtMost [simp]:
Suc ' {i..j} = {Suc i..Suc j}
using image-add-atLeastAtMost [of 1 i j]
by (simp only: plus-1-eq-Suc simp)

lemma image-Suc-atLeastLessThan [simp]:
Suc ' {i..<j} = {Suc i..<Suc j}
using image-add-atLeastLessThan [of 1 i j]
by (simp only: plus-1-eq-Suc simp)

corollary image-Suc-atMost: 
Suc ' {..n} = {1..Suc n}
by (simp add: atMost-atLeast0 atLeastLessThanSuc-atLeastAtMost)

corollary image-Suc-lessThan: 
Suc ' {..<n} = {1..n}
by (simp add: lessThan-atLeast0 atLeastLessThanSuc-atLeastAtMost)

lemma image-diff-atLeastAtMost [simp]:
fixes d::'a::linordered-idom shows ((-) d ' {a..b}) = {d-b..d-a}
apply auto
apply (rule-tac x=d-x in rev-image-eqI, auto)
done

lemma image-diff-atLeastLessThan [simp]:
fixes a b ::'a::linordered-idom
shows (-) c ' {a..<b} = {c - b..<c - a}
proof - 
  have (-) c ' {a..<b} = (+) c ' uminus ' {a..<b}
  unfolding image-image by simp
  also have ... = {c - b..<c - a} by simp
finally show \( \text{thesis by simp} \)

qed

lemma image-minus-const-greaterThanAtMost\{simp\}:
fixes a b c ::'a::linordered-idom
shows \((-) c \cdot \{a<..b\} = \{c - b..<c - a\}\)
proof -
  have \((-) c \cdot \{a<..b\} = (+) c \cdot uminus \cdot \{a<..b\}\)
    unfolding image-image by simp
  also have \(\ldots = \{c - b..<c - a\}\ by simp\)
finally show \(\text{thesis by simp}\)
qed

lemma image-minus-const-atLeast\{simp\}:
fixes a c ::'a::linordered-idom
shows \((-) c \cdot \{a..\} = \{..c - a\}\)
proof -
  have \((-) c \cdot \{a..\} = (+) c \cdot uminus \cdot \{a ..\}\)
    unfolding image-image by simp
  also have \(\ldots = \{..c - a\}\ by simp\)
finally show \(\text{thesis by simp}\)
qed

lemma image-minus-const-AtMost\{simp\}:
fixes b c ::'a::linordered-idom
shows \((-) c \cdot \{..b\} = \{c - b..\}\)
proof -
  have \((-) c \cdot \{..b\} = (+) c \cdot uminus \cdot \{..b\}\)
    unfolding image-image by simp
  also have \(\ldots = \{c - b..\}\ by simp\)
finally show \(\text{thesis by simp}\)
qed

lemma image-minus-const-atLeastAtMost' \{simp\}:
(\(\lambda t. t - d\)' \cdot \{a..b\} = \{a - d..b - d\}\) for d ::'a::linordered-idom
by (metis (no-types, lifting) diff-conv-add-uminus image-add-atLeastAtMost' image-cong)

context linordered-field
begin

lemma image-mult-atLeastAtMost \{simp\}:
\((\ast) d \cdot \{a..b\}\) = \(d*a..d*b\) if \(d>0\)
using that
by (auto simp: field-simps mult-le-cancel-right intro: rev-image-eql [where \(x=x/d\) for \(x\)])

lemma image-divide-atLeastAtMost \{simp\}:
\((\lambda c. c / d)' \cdot \{a..b\}\) = \(a/d..b/d\) if \(d>0\)
proof -
from that have inverse $d > 0$
  by simp
with image-mult-atLeastAtMost [of inverse $d$ $a$ $b$]
have $(*)$ (inverse $d$) $'$ $\{a..b\}$ = $\{\text{inverse } d \times a..\text{inverse } d \times b\}$
  by blast
moreover have $(*)$ (inverse $d$) = $(\lambda c. c / d)$
  by (simp add: fun-eq-iff field-simps)
ultimately show ?thesis
  by simp
qed

lemma image-mult-atLeastAtMost-if':
$$(\lambda x. x \times c) \to \{x..y\} =$$
$$(\text{if } x \leq y \text{ then if } c > 0 \text{ then } \{x \times c .. y \times c\} \text{ else } \{y \times c .. x \times c\} \text{ else } \{\})$$
using image-mult-atLeastAtMost-if [of $c$ $x$ $y$] by (auto simp add: ac-simps)
lemma image-affinity-atLeastAtMost:
((\lambda x. m * x + c) ' {a..b}) = (if \{a..b\}={} then {} 
else if 0 ≤ m then \{m * a + c .. m * b + c\} 
else \{m * b + c .. m * a + c\})

proof -
have \*: (\lambda x. m * x + c) = ((\lambda x. x + c) ∘ (\*) m)
  by (simp add: fun-eq-iff)
show \?thesis by (simp only: \* image-comp [symmetric] image-mult-atLeastAtMost-if)
(auto simp add: mult-le-cancel-left)
qed

lemma image-affinity-atLeastAtMost-diff:
((\lambda x. m * x − c) ' {a..b}) = (if \{a..b\}={} then {} 
else if 0 ≤ m then \{m * a − c .. m * b − c\} 
else \{m * b − c .. m * a − c\})
using image-affinity-atLeastAtMost [of m − c a b]
by simp

lemma image-affinity-atLeastAtMost-div:
((\lambda x. x/m + c) ' {a..b}) = (if \{a..b\}={} then {} 
else if 0 ≤ m then \{a/m + c .. b/m + c\} 
else \{b/m + c .. a/m + c\})
using image-affinity-atLeastAtMost [of inverse m c a b]
by (simp add: field-class.field-divide-inverse algebra-simps inverse-eq-divide)

lemma image-affinity-atLeastAtMost-div-diff:
((\lambda x. x/m − c) ' {a..b}) = (if \{a..b\}={} then {} 
else if 0 ≤ m then \{a/m − c .. b/m − c\} 
else \{b/m − c .. a/m − c\})
using image-affinity-atLeastAtMost-div-diff [of inverse m c a b]
by (simp add: field-class.field-divide-inverse algebra-simps inverse-eq-divide)
end

lemma atLeast1-lessThan-eq-remove0:
\{Suc 0..<n\} = \{..<n\} − \{0\}
by auto

lemma atLeast1-atMost-eq-remove0:
\{Suc 0..n\} = \{..n\} − \{0\}
by auto

lemma image-add-int-atLeastLessThan:
(\lambda x. x + (\{..\} : int)) ' \{0..<a-l\} = \{1..<u\}
apply (auto simp add: image-def)
apply (rule-tac x = x − l in bexI)
apply auto
done
lemma image-minus-const-atLeastLessThan-nat:
  fixes c :: nat
  shows \((\lambda i. i - c) \cdot \{ x ..< y \} =
    (if c < y then \{ x - c ..< y - c \} else if x < y then \{ 0 \} else \{ \}))
  (is _ = ?right)
proof safe
  fix a assume a: a \in ?right
  show a \in (\lambda i. i - c) \cdot \{ x ..< y \}
  proof cases
    assume c < y with a
    show ?thesis
      by (auto intro!: image-eqI[of - - a + c])
  next
    assume \neg c < y with a
    show ?thesis
      by (auto intro!: image-eqI[of - x] split: if-split-asm)
  qed
qed auto

lemma image-int-atLeastLessThan:
  int \cdot \{ a..<b \} = \{ int a..<int b \}
  by (auto intro!: image-eqI [where x = nat x for x])

lemma image-int-atLeastAtMost:
  int \cdot \{ a..b \} = \{ int a..int b \}
  by (auto intro!: image-eqI [where x = nat x for x])

60.5.10 Finiteness

lemma finite-lessThan [iff]: fixes k :: nat shows finite \{..<k\}
  by (induct k) (simp-all add: lessThan-Suc)

lemma finite-atMost [iff]: fixes k :: nat shows finite \{..k\}
  by (induct k) (simp-all add: atMost-Suc)

lemma finite-greaterThanLessThan [iff]:
  fixes l :: nat shows finite \{l..<..<u\}
  by (simp add: greaterThanLessThan-def)

lemma finite-atLeastLessThan [iff]:
  fixes l :: nat shows finite \{l..<..<u\}
  by (simp add: atLeastLessThan-def)

lemma finite-greaterThanAtMost [iff]:
  fixes l :: nat shows finite \{l..<..<u\}
  by (simp add: greaterThanAtMost-def)

lemma finite-atLeastAtMost [iff]:
  fixes l :: nat shows finite \{l..<..<u\}
  by (simp add: atLeastAtMost-def)
A bounded set of natural numbers is finite.

**lemma** bounded-nat-set-is-finite:  
(∀ i ∈ ℕ. i < (n::nat)) ⇒ finite N

by (rule finite-subset [OF finite-lessThan]) auto

A set of natural numbers is finite iff it is bounded.

**lemma** finite-nat-set-iff-bounded:

finite N = (∃ m. ∀ n ∈ ℕ. n < m)

**proof**

assume f: ?F show ?B

using Max_ge [OF ⟨?F⟩], simplified less-Suc_eq_le [symmetric] by blast

next

assume ?B show ?F using ⟨?B⟩ by (blast intro: bounded-nat-set-is-finite)

qed

**lemma** finite-nat-set-iff-bounded-le:

finite N = (∃ m. ∀ n ∈ ℕ. n ≤ m)

unfolding finite-nat-set-iff-bounded by (blast dest: less-imp-le-nat le-imp-less-Suc)

**lemma** finite-less-ub:

x::nat =⇒ nat. (!n. n ≤ f n) =⇒ finite {n. f n ≤ u}

by (rule-tac B = {..u} in finite-subset, auto intro: order-trans)

**lemma** bounded-Max-nat:

fixes P :: nat ⇒ bool

assumes x: P x and M: ∀ x. P x \implies x ≤ M

obtains m where P m \implies ∀ x. P x \implies x ≤ m

**proof**

have finite {x. P x}

using M finite-nat-set-iff-bounded-le by auto

then have Max {x. P x} ∈ {x. P x}

using Max-in x by auto

then show ?thesis

by (simp add: finite {x. P x}; that)

qed

Any subset of an interval of natural numbers the size of the subset is exactly that interval.

**lemma** subset-card-intvl-is-intvl:

assumes A ⊆ {k..<k + card A}

shows A = {k..<k + card A}

**proof** (cases finite A)

case True

from this and assms show ?thesis

**proof** (induct A rule: finite-linorder-max-induct)

case empty thus ?case by auto

next

case (insert b A)

hence *: b ∉ A by auto
with `insert` have \( A \leq \{k..<k + \text{card } A \} \) and \( b = k + \text{card } A \)
by `fastforce`
with `insert *` show `?case` by `auto`
qed
next
  case `False`
  with `assms` show `?thesis` by `simp`
qed

60.5.11 Proving Inclusions and Equalities between Unions

lemma `UN-le-eq-Un0`:
\[
(\bigcup_{i \leq n:\text{nat}}. M\ i) = (\bigcup_{i \in \{1..n\}. M\ i}) \cup M\ 0 \quad (\text{is } ?A = ?B)
\]
proof
  show `?A \subseteq ?B`
  proof
    fix `x` assume `x \in ?A`
    then obtain `i` where `i \leq n \ x \in M\ i` by `auto`
    show `x \in ?B`
      proof (cases `i`)
        case `0` with `i` show `?thesis` by `simp`
      next
        case `(Suc \ j)` with `i` show `?thesis` by `auto`
      qed
    qed
  qed
next
  show `?B \subseteq ?A` by `fastforce`
qed

lemma `UN-le-add-shift`:
\[
(\bigcup_{i \leq n:\text{nat}}. M\ (i+k)) = (\bigcup_{i \in \{k..n+k\}. M\ i}) \quad (\text{is } ?A = ?B)
\]
proof
  show `?A \subseteq ?B` by `fastforce`
next
  show `?B \subseteq ?A`
  proof
    fix `x` assume `x \in ?B`
    then obtain `i` where `i \in \{k..n+k\} \ x \in M\ (i)` by `auto`
    hence `i-k \leq n \wedge x \in M\ ((i-k)+k)` by `auto`
    thus `x \in ?A` by `blast`
  qed
qed

lemma `UN-le-add-shift-strict`:
\[
(\bigcup_{i < n:\text{nat}}. M\ (i+k)) = (\bigcup_{i \in \{k..<n+k\}. M\ i}) \quad (\text{is } ?A = ?B)
\]
proof
  show `?B \subseteq ?A`
  proof
    fix `x` assume `x \in ?B`
then obtain \( i \) where \( i \in \{ k..<n+k \} \) \( x \in M(i) \) by auto
then have \( i - k < n \land x \in M((i-k) + k) \) by auto
then show \( x \in \mathit{A} \) using UN-le-add-shift by blast
qed

qed (fastforce)

lemma UN-UN-finite-eq: \((\bigcup n::nat. \bigcup i\in\{0..<n\}. A i) = (\bigcup n. A n)\) by (auto simp add: atLeast0LessThan)

lemma UN-finite-subset:
\((\forall n::nat. \bigcup i\in\{0..<n\}. A i) \subseteq C) \implies (\bigcup n. A n) \subseteq C\)
by (subst UN-UN-finite-eq [symmetric]) blast

lemma UN-finite2-subset:
\((\forall n::nat. (\bigcup i\in\{0..<n\}. A i) \subseteq (\bigcup i\in\{0..<n+k\}. B i))\)
shows \((\bigcup n. A n) \subseteq (\bigcup n. B n)\)
proof (rule UN-finite-subset, rule)
fix \( n \) and \( a \)
from assms have \((\bigcup i\in\{0..<n\}. A i) \subseteq (\bigcup i\in\{0..<n+k\}. B i)\).
moreover assume \( a \in (\bigcup i\in\{0..<n\}. A i)\)
ultimately have \( a \in (\bigcup i\in\{0..<n+k\}. B i)\) by blast
then show \( a \in (\bigcup i. B i) \) by (auto simp add: UN-UN-finite-eq)
qed

lemma UN-finite2-eq:
\((\forall n::nat. (\bigcup i\in\{0..<n\}. A i) = (\bigcup i\in\{0..<n+k\}. B i)) \implies (\bigcup n. A n) = (\bigcup n. B n)\)
apply (rule subset-antisym [OF UN-finite-subset UN-finite2-subset])
apply auto
apply (force simp add: atLeastLessThan-add-Un [of 0])
done

60.5.12  Cardinality

lemma card-lessThan [simp]: card \{..<u\} = u
by (induct u, simp-all add: lessThan-Suc)

lemma card-atMost [simp]: card \{..u\} = Suc u
by (simp add: lessThan-Suc-atMost [THEN sym])

lemma card-atLeastLessThan [simp]: card \{l..<u\} = u - l
proof
  have \{l..<u\} = (\lambda x. x + l) \cdot \{..<u-l\}
  apply (auto simp add: image-def atLeastLessThan-def lessThan-def)
  apply (rule_tac x = x - l in exI)
  apply arith
done
then have card \{l..<u\} = card \{..<u-l\}
by (simp add: card-image inj-on-def)
then show thesis
  by simp
qed

lemma card-atLeastAtMost [simp]: card \{l..u\} = Suc u - l
  by (subst atLeastLessThanSuc-atLeastAtMost [THEN sym], simp)

lemma card-greaterThanAtMost [simp]: card \{l..<u\} = u - l
  by (subst atLeastSucAtMost-greaterThanAtMost [THEN sym], simp)

lemma card-greaterThanLessThan [simp]: card \{l..<u\} = u - Suc l
  by (subst atLeastSucLessThan-greaterThanLessThan [THEN sym], simp)

lemma subset-eq-atLeast0-lessThan-finite:
  fixes n :: nat
  assumes N ⊆\{0..<n\}
  shows finite N
  using assms finite-atLeastLessThan by (rule finite-subset)

lemma subset-eq-atLeast0-atMost-finite:
  fixes n :: nat
  assumes N ⊆\{0..n\}
  shows finite N
  using assms finite-atLeastAtMost by (rule finite-subset)

lemma ex-bij-betw-nat-finite:
  finite M ⇒ ∃h. bij-betw h \{0..<\text{card } M\} M
  apply (erule finite-imp-nat-seg-image-inj-on)
  apply (auto simp: atLeast0LessThan [symmetric] lessThan-def [symmetric] card-image bij-betw-def)
  done

lemma ex-bij-betw-finite-nat:
  finite M ⇒ ∃h. bij-betw h \{0..<\text{card } M\} M
  by (blast dest: ex-bij-betw-nat-finite bij-betw-inv)

lemma finite-same-card-bij:
  finite A ⇒ finite B ⇒ card A = card B ⇒ ∃h. bij-betw h A B
  apply (erule ex-bij-betw-finite-nat)
  apply (auto simp: ex-bij-betw-nat-finite ex-bij-betw-nat-finite)
  done

lemma ex-bij-betw-nat-finite-1:
  finite M ⇒ ∃h. bij-betw h \{1..\text{card } M\} M
  by (rule finite-same-card-bij) auto

lemma bij-betw-iff-card:
  assumes finite A finite B
shows $(\exists f. \text{bij-betw} f A B) \iff (\text{card } A = \text{card } B)$

proof

assume card $A = \text{card } B$
moreover obtain $f$ where $\text{bij-betw} f A [0..<\text{card } A]$ using assms ex-bij-betw-finite-nat by blast
moreover obtain $g$ where $\text{bij-betw} g [0..<\text{card } B] B$ using assms ex-bij-betw-nat-finite by blast
ultimately have $\text{bij-betw} (g \circ f) A B$ by (auto simp: bij-betw-trans)
thus $(\exists f. \text{bij-betw} f A B)$ by blast
qed (auto simp: bij-betw-same-card)

lemma subset-eq-atLeast0-lessThan-card:
fixes $n :: \text{nat}$
assumes $N \subseteq \{0..<n\}$
shows $\text{card } N \leq n$
proof
from assms finite-lessThan have $\text{card } N \leq \text{card } \{0..<n\}$ using card-mono by blast
then show $\text{thesis}$ by simp
qed

Relational version of card-inj-on-le:

lemma card-le-if-inj-on-rel:
assumes $\text{finite } B$
$\forall a. a \in A \implies \exists b. b \in B \land r a b$
$\forall a1 a2 b. [a1 \in A; a2 \in A; b \in B; r a1 b; r a2 b] \implies a1 = a2$
shows $\text{card } A \leq \text{card } B$
proof
let $?P = \lambda a b. b \in B \land r a b$
let $?f = \lambda a. \text{SOME } b. ?P a b$
have $1: ?f ' A \subseteq B$ by (auto intro: some12-ex[OF assms(2)])
have inj-on $?f A$
proof (auto simp: inj-on-def)
fix a1 a2 assume assms: $a1 \in A$ $a2 \in A$ $?f a1 = ?f a2$
have $0: ?f a1 \in B$ using $1 \langle a1 \in A \rangle$ by blast
have $1: r a1 (\langle ?f a1 \rangle$ using some1-ex[OF assms(2)[OF $a1 \in A$]] by blast
have $2: r a2 (\langle ?f a1 \rangle$ using some1-ex[OF assms(2)[OF $a2 \in A$]] assms(3) by auto
show $a1 = a2$ using assms(3)[OF assms(1,2) 0 1 2].
qed
with $1$ show $\text{thesis}$ using card-inj-on-le[OF $?f A B$] assms(1) by simp
qed

60.6 Intervals of integers

text科创

lemma atLeastLessThanPlusOne-atLeastAtMost-int: $\{l..<u+1\} = \{l..(u::int)\}$
by (auto simp add: atLeastAtMost-def atLeastLessThan-def)
THEORY "Set-Interval"

lemma atLeastPlusOneAtMost-greaterThanAtMost-int: \{1..u\} = \{l<..(u::int)\}
  by (auto simp add: atLeastAtMost-def greaterThanAtMost-def)

lemma atLeastPlusOneLessThan-greaterThanLessThan-int:
  \{l+1..<u\} = \{l..<u::int\}
  by (auto simp add: atLeastLessThan-def greaterThanLessThan-def)

60.6.1 Finiteness

lemma image-atLeastZeroLessThan-int: 0 ≤ u ==> 
  \{(0::int)..<u\} = int ' {..<nat u}
  unfolding image-def lessThan-def
  apply auto
  apply (rule_tac x = nat x in exI)
  apply (auto simp add: zless-nat-eq-int-zless [THEN sym])
  done

lemma finite-atLeastZeroLessThan-int: finite \{(0::int)..<u\}
  proof (cases 0 ≤ u)
    case True
    then show thesis
      by (auto simp: image-atLeastZeroLessThan-int inj-on-def)
  qed auto

lemma finite-atLeastLessThan-int [iff]: finite \{l..<u::int\}
  by (simp only: image-add-int-atLeastLessThan symmetric, 
                  finite-atLeastZeroLessThan-int)

lemma finite-atLeastAtMost-int [iff]: finite \{l..(u::int)\}
  by (subst atLeastLessThanPlusOne-atLeastAtMost-int [THEN sym], simp)

lemma finite-greaterThanAtMost-int [iff]: finite \{l<..<(u::int)\}
  by (subst atLeastPlusOneAtMost-greaterThanAtMost-int [THEN sym], simp)

lemma finite-greaterThanLessThan-int [iff]: finite \{l<..<u::int\}
  by (subst atLeastPlusOneLessThan-greaterThanLessThan-int [THEN sym], simp)

60.6.2 Cardinality

lemma card-atLeastZeroLessThan-int: card \{(0::int)..<u\} = nat u
  proof (cases 0 ≤ u)
    case True
    then show thesis
      by (auto simp: image-atLeastZeroLessThan-int card-image inj-on-def)
  qed auto

lemma card-atLeastLessThan-int [simp]: card \{l..<u\} = nat (u - l)
  proof
    have card \{l..<u\} = card \{0..<u-l\}
      apply (subst image-add-int-atLeastLessThan [symmetric])
  qed
apply (rule card-image)
apply (simp add: inj-on-def)
done
then show ?thesis
  by (simp add: card-atLeastZeroLessThan-int)
qed

lemma card-atLeastAtMost-int [simp]:
card {l..u} = nat (u − l + 1)
apply (subst atLeastLessThanPlusOne-atLeastAtMost-int [THEN sym])
apply (auto simp add: algebra-simps)
done

lemma card-greaterThanAtMost-int [simp]:
card {l..<u} = nat (u − l)
by (subst atLeastPlusOneAtMost-greaterThanAtMost-int [THEN sym], simp)

lemma card-greaterThanLessThan-int [simp]:
card {l..<u} = nat (u − (l + 1))
apply (simp add: card-eq-0-iff)
done

lemma finite-M-bounded-by-nat: finite {k. P k ∧ k < (i::nat)}
proof −
  have {k. P k ∧ k < Suc i} ⊆ {..<i} by auto
  with finite-lessThan[of i] show ?thesis by (simp add: finite-subset)
qed

lemma card-less:
  assumes zero-in-M: 0 ∈ M
  shows card {k ∈ M. k < Suc i} ≠ 0
proof −
  from zero-in-M have {k ∈ M. k < Suc i} ≠ {} by auto
qed

lemma card-less-Suc2:
  assumes 0 /∈ M shows card {k. Suc k ∈ M ∧ k < i} = card {k ∈ M. k < Suc i}
proof −
  have *: [j ∈ M; j < Suc i] ⇒ j − Suc 0 < i ∧ Suc (j − Suc 0) ∈ M ∧ Suc 0 ≤ j for j
    by (cases j) (use assms in auto)
  show ?thesis
  proof (rule card-bij-eq)
    show inj-on Suc {k. Suc k ∈ M ∧ k < i}
      by force
    show inj-on (λx. x − Suc 0) {k ∈ M. k < Suc i}
      by (rule inj-on-diff-nat) (use * in blast)
    qed
  qed

lemma card-less-Suc:
assumes $\emptyset \in M$

shows $\text{Suc} (| k. \text{Suc} k \in M \land k < i |) = | k \in M. k < \text{Suc} i |

proof –

have $\text{Suc} (| k. \text{Suc} k \in M \land k < i |) = \text{Suc} (| k. \text{Suc} k \in M - \{\emptyset\} \land k < i |)$
  by simp

also have $\ldots = \text{Suc} (| k \in M - \{\emptyset\}. k < \text{Suc} i |)$
  apply (subst card-less-Suc2)
  using assms by auto

also have $\ldots = \text{card} (| k \in M \land k < \text{Suc} i | - \{\emptyset\})$
  by (force intro: arg-cong [where $f=\text{card}!$])

also have $\ldots = \text{card} (\text{insert} 0 (| k \in M \land k < \text{Suc} i | - \{\emptyset\}))$
  by (simp add: card-insert)

also have $\ldots = \text{card} | k \in M. k < \text{Suc} i |
  using assms
  by (force simp add: intro: arg-cong [where $f=\text{card}!$])

finally show $?thesis.$

qed

60.7 Lemmas useful with the summation operator sum

For examples, see Algebra/poly/UnivPoly2.thy

60.7.1 Disjoint Unions

Singletons and open intervals

lemma isl-disj-un-singleton:
  $\{l:<a::lorder\} \cup \{l<..\} = \{l..\}$
  $\{..<u\} \cup \{u::\text{linorder}\} = \{..u\}$
  $(l':a::lorder) < u \Longrightarrow \{l\} \cup \{l..<u\} = \{l..<u\}$
  $(l':a::lorder) < u \Longrightarrow \{l..<u\} \cup \{u\} = \{l..<u\}$
  $(l':a::lorder) \leq u \Longrightarrow \{l\} \cup \{l..u\} = \{l..u\}$
  $(l':a::lorder) \leq u \Longrightarrow \{l..<u\} \cup \{u\} = \{l..<u\}$
  by auto

One- and two-sided intervals

lemma isl-disj-un-one:
  $(l':a::lorder) < u \Longrightarrow \{..l\} \cup \{l..<u\} = \{..<u\}$
  $(l':a::lorder) \leq u \Longrightarrow \{..<l\} \cup \{l..<u\} = \{..<u\}$
  $(l':a::lorder) \leq u \Longrightarrow \{..<l\} \cup \{l..u\} = \{..<u\}$
  $(l':a::lorder) \leq u \Longrightarrow \{l..<u\} \cup \{u..<\} = \{l..<\}$
  $(l':a::lorder) < u \Longrightarrow \{l..<u\} \cup \{u..<\} = \{l..<\}$
  $(l':a::lorder) \leq u \Longrightarrow \{l..<u\} \cup \{u..<\} = \{l..<\}$
  $(l':a::lorder) \leq u \Longrightarrow \{l..<u\} \cup \{u..<\} = \{l..<\}$
  by auto

Two- and two-sided intervals
THEORY "Set-Interval"

lemma ivl-disj-un-two:
  \[ (l::'a::linorder) < m; m \leq u \implies \{l..<m\} \cup \{m..<u\} = \{l..<u\} \]
  \[ (l::'a::linorder) \leq m; m < u \implies \{l..<m\} \cup \{m..<u\} = \{l..<u\} \]
  \[ (l::'a::linorder) \leq m; m \leq u \implies \{l..m\} \cup \{m..<u\} = \{l..u\} \]
  \[ (l::'a::linorder) < m; m < u \implies \{l..<m\} \cup \{m..<u\} = \{l..<u\} \]
  \[ (l::'a::linorder) \leq m; m < u \implies \{l..m\} \cup \{m..<u\} = \{l..<u\} \]
  \[ (l::'a::linorder) < m; m \leq u \implies \{l..<m\} \cup \{m..<u\} = \{l..<u\} \]
  \[ (l::'a::linorder) \leq m; m \leq u \implies \{l..m\} \cup \{m..<u\} = \{l..<u\} \]
by auto

lemma ivl-disj-un-two-touch:
  \[ (l::'a::linorder) < m; m < u \implies \{l..<m\} \cup \{m..<u\} = \{l..<u\} \]
  \[ (l::'a::linorder) \leq m; m < u \implies \{l..<m\} \cup \{m..<u\} = \{l..<u\} \]
  \[ (l::'a::linorder) < m; m \leq u \implies \{l..<m\} \cup \{m..<u\} = \{l..<u\} \]
  \[ (l::'a::linorder) \leq m; m \leq u \implies \{l..m\} \cup \{m..<u\} = \{l..<u\} \]
by auto

lemmas ivl-disj-un = ivl-disj-un-singleton ivl-disj-un-two ivl-disj-un-one ivl-disj-un-two-touch

60.7.2 Disjoint Intersections

One- and two-sided intervals

lemma ivl-disj-int-one:
  \{..l::'a::order\} \ Inter \ \{l..<u\} = \{\} \]
  \{..l\} \ Inter \ \{l..<u\} = \{\}
  \{..l\} \ Inter \ \{l..u\} = \{\}
  \{l..<u\} \ Inter \ \{u..<\} = \{\}
  \{l..<u\} \ Inter \ \{u..\} = \{\}
  \{l..<u\} \ Inter \ \{u..\} = \{\}
by auto

Two- and two-sided intervals

lemma ivl-disj-int-two:
  \{l::'a::order..<m\} \ Inter \ \{m..<u\} = \{\}
  \{l..<m\} \ Inter \ \{m..<u\} = \{\}
  \{l..<m\} \ Inter \ \{m..<u\} = \{\}
  \{l..<m\} \ Inter \ \{m..<u\} = \{\}
  \{l..<m\} \ Inter \ \{m..<u\} = \{\}
  \{l..<m\} \ Inter \ \{m..<u\} = \{\}
by auto

lemmas ivl-disj-int = ivl-disj-int-one ivl-disj-int-two
60.7.3 Some Differences

**lemma** `ivl-diff [simp]`:
\[ i \leq n \Rightarrow \{i..<m\} - \{i..<n\} = \{n..<(m::'a::linorder)\} \]
**by** `(auto)`

**lemma** `(in linorder) `lessThan-minus-lessThan ` [simp]`:
\[ \{..< n\} - \{..< m\} = \{m ..< n\} \]
**by** `auto`

**lemma** `(in linorder) `atLeastAtMost-diff-ends`:
\[ \{a..b\} - \{a, b\} = \{a..<b\} \]
**by** `auto`

60.7.4 Some Subset Conditions

**lemma** `ivl-subset [simp]`: \([\{i..<j\} \subseteq \{m..<n\}] = (j \leq i \lor m \leq i \land j \leq (n::'a::linorder))\]
**using** `linorder-class.le-less-linear [of i n]`
**apply** `(auto simp: linorder-not-le)`
**apply** `(force intro: leI)+`
**done**

**lemma** `obtain-subset-with-card-n`:
**assumes** \( n \leq \text{card } S \)
**obtains** \( T \) where \( T \subseteq S \text{ card } T = n \text{ finite } T \)
**proof** –
**obtain** \( n' \) where \( \text{card } S = n + n' \)
**by** `(metis assms le-add-diff-inverse)`
**with** `that` show `thesis`
**proof** `(induct n' arbitrary: S)`
**case** `0`
**then show** `?case`
**by** `(cases finite S) auto`
**next**
**case** `Suc`
**then show** `?case`
**by** `(simp add: card-Suc-eq) (metis subset-insertI2)`
**qed**
**qed**

60.8 Generic big monoid operation over intervals

**context** `semiring-char-0`
**begin**

**lemma** `inj-on-of-nat [simp]`:
\( \text{inj-on of-nat } N \)
**by** `rule simp`

**lemma** `bij-betw-af-nat [simp]`: `
bij-betw of-nat \( N \leftrightarrow \text{of-nat} \cdot N = A \)
by (simp add: bij-betw-def)

end

context comm-monoid-set
begin

lemma atLeastLessThan-reindex:
\( F \circ g \{ h \cdot m..<n \} = F \circ (g \circ h) \{ m..<n \} \)
if bij-betw h \{ m..<n \} \{ h \cdot m..<n \} for m n :: nat
proof
from that have inj-on h \{ m..<n \} and h \cdot \{ m..<n \} = \{ h \cdot m..<n \}
by (simp-all add: bij-betw-def)
then show ?thesis
using reindex [of h \{ m..<n \} g] by simp

qed

lemma atLeastAtMost-reindex:
\( F \circ g \{ h \cdot m..n \} = F \circ (g \circ h) \{ m..n \} \)
if bij-betw h \{ m..n \} \{ h \cdot m..n \} for m n :: nat
proof
from that have inj-on h \{ m..n \} and h \cdot \{ m..n \} = \{ h \cdot m..n \}
by (simp-all add: bij-betw-def)
then show ?thesis
using reindex [of h \{ m..n \} g] by simp

qed

lemma atLeastLessThan-shift-bounds:
\( F \circ g \{ m+k..<n+k \} = F \circ (g \circ \text{plus} k) \{ m..<n \} \)
for m n k :: nat
using atLeastLessThan-reindex [of plus k m n g]
by (simp add: ac-simps)

lemma atLeastAtMost-shift-bounds:
\( F \circ g \{ m+k..n+k \} = F \circ (g \circ \text{plus} k) \{ m..n \} \)
for m n k :: nat
using atLeastAtMost-reindex [of plus k m n g]
by (simp add: ac-simps)

lemma atLeast-Suc-lessThan-Suc-shift:
\( F \circ g \{ \text{Suc} \cdot m..<\text{Suc} n \} = F \circ (g \circ \text{Suc}) \{ m..<n \} \)
using atLeastLessThan-shift-bounds [of - - 1]
by (simp add: plus-1-eq-Suc)

lemma atLeast-Suc-atMost-Suc-shift:
\( F \circ g \{ \text{Suc} \cdot m..\text{Suc} n \} = F \circ (g \circ \text{Suc}) \{ m..n \} \)
using atLeastAtMost-shift-bounds [of - - 1]
by (simp add: plus-1-eq-Suc)
lemma atLeast-int-lessThan-int-shift:
  \( F \ g \{ \text{int \ } m..<\text{int \ } n \} = F \ (g \circ \text{int}) \ \{m..<n\} \)
  by (rule atLeastLessThan-reindex)
  (simp add: image-int-atLeastLessThan)

lemma atLeast-int-atMost-int-shift:
  \( F \ g \{ \text{int \ } m..\text{int \ } n \} = F \ (g \circ \text{int}) \ \{m..n\} \)
  by (rule atLeastAtMost-reindex)
  (simp add: image-int-atLeastAtMost)

lemma atLeast0-lessThan-Suc:
  \( F \ g \{ 0..<\text{Suc \ } n \} = F g \{ 0..<\text{Suc \ } n \} \star g \ n \)
  by (simp add: atLeast0-lessThan-Suc ac-simps)

lemma atLeast0-atMost-Suc:
  \( F \ g \{ 0..\text{Suc \ } n \} = F g \{ 0..\text{Suc \ } n \} \star g \ (\text{Suc \ } n) \)
  by (simp add: atLeast0-atMost-Suc ac-simps)

lemma atLeast0-lessThan-Suc-shift:
  \( F \ g \{ 0..<\text{Suc \ } n \} = g \ 0 \star F \ (g \circ \text{Suc}) \ \{0..<\text{Suc \ } n \} \)
  by (simp add: atLeast0-lessThan-Suc-eq-insert-0 atLeast-Suc-lessThan-Suc-shift)

lemma atLeast0-atMost-Suc-shift:
  \( F \ g \{ 0..<\text{Suc \ } n \} = g \ 0 \star F \ (g \circ \text{Suc}) \ \{0..<\text{Suc \ } n \} \)
  by (simp add: atLeast0-atMost-Suc-eq-insert-0 atLeast-Suc-atMost-Suc-shift)

lemma atLeast-Suc-lessThan:
  \( F \ g \{ m..<\text{n} \} = g m \star F \ g \ \{\text{Suc \ } m..<\text{n} \} \) if \( m < n \)
proof –
  from that have \( \{m..<\text{n} \} = \text{insert \ } m \ \{\text{Suc \ } m..<\text{n} \} \)
  by auto
  then show \( \text{thesis} \) by simp
qed

lemma atLeast-Suc-atMost:
  \( F \ g \{ m..\text{n} \} = g m \star F \ g \ \{\text{Suc \ } m..\text{n} \} \) if \( m \leq n \)
proof –
  from that have \( \{m..\text{n} \} = \text{insert \ } m \ \{\text{Suc \ } m..\text{n} \} \)
  by auto
  then show \( \text{thesis} \) by simp
qed

lemma interval-cong:
  \( a = c \Rightarrow b = d \Rightarrow (\forall x. c \leq x \Rightarrow x < d \Rightarrow g x = h x) \)
  \( \Rightarrow F \ g \{a..<b\} = F h \{c..<d\} \)
  by (rule cong) simp-all

lemma atLeastLessThan-shift-0:
THEORY “Set-Interval”

lemma atLeastAtMost-shift-0:
  fixes m n p :: nat
  shows F g \{m..<n\} = F (g o plus m) \{0..<n - m\}
  using atLeastLessThan-shift-bounds \[of g 0 m n - m\]
  by (cases m \leq n) simp-all

lemma atLeastAtMost-shift-0-
  fixes m n p :: nat
  assumes m \leq n
  shows F g \{m..n\} = F (g o plus m) \{0..n - m\}
  using assms atLeastLessThan-rev-at-least-Suc-atMost
  by (cases m \leq n) simp-all

lemma atLeastLessThan-concat:
  fixes m n p :: nat
  shows m \leq n \Rightarrow n \leq p \Rightarrow F g \{m..<n\} * F g \{n..<p\} = F g \{m..<p\}
  by (simp add: union-disjoint [symmetric] ivl-disj-un)

lemma atLeastLessThan-at leastSuc-atMost-
  F g \{n..<m\} = F (\lambda i. g (m + n - Suc i)) \{n..<m\}
  by (rule reindex-bij-witness \[where i=\lambda i. m + n - Suc i and j=\lambda i. m + n - Suc i], auto)

lemma atLeastAtMost-rev:
  fixes n m :: nat
  shows F g \{n..m\} = F (\lambda i. g (m + n - i)) \{n..m\}
  by (rule reindex-bij-witness \[where i=\lambda i. m + n - i and j=\lambda i. m + n - i\])
  auto

lemma atLeastLessThan-rev-at leastSuc-atMost-
  F g \{n..<m\} = F (\lambda i. g (m + n - i)) \{Suc n..m\}
  unfolding atLeastLessThan-rev \[of g n m\]
  by (cases m) (simp-all add: atLeast-Suc-atMost-Suc-shift atLeastLessThanSuc-atLeastAtMost)

end

60.9 Summation indexed over intervals

syntax (ASCII)
  -from-to-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\text{SUM} - = -..-/ -) [0,0,0,10] 10)
  -from-upto-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\text{SUM} - = -..<-/ -) [0,0,0,10] 10)
  -upto-sum :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\text{SUM} -..<-/ -) [0,0,0,10] 10)
  -upto-sum :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b ((\text{SUM} -<=-/ -) [0,0,0,10] 10)

syntax (latex-sum output)
  -from-to-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b
  ((\text{\$\sum_{a}^{b}\$}) [0,0,0,10] 10)
  -from-upto-sum :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b
  ((\text{\$\sum_{a}^{b-1}\$}) [0,0,0,10] 10)
  -upto-sum :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b
  ((\text{\$\sum_{a}^{b}\$}) [0,0,10] 10)
proof

-zero-middle

begin

context comm-monoid-set

unsimplified premise

This congruence rule should be used for sums over intervals as the standard

\begin{align*}
\sum_{x} & \in \{a..b\}. e \\
\sum_{x} & \in \{a..<b\}. e \\
\sum_{x} & \in \{..b\}. e \\
\sum_{x} & \in \{..<b\}. e
\end{align*}

Note that for uniformity on nat it is better to use \(\sum x = 0..<n. e\) rather than \(\sum x<n. e\): sum may not provide all lemmas available for \(\{m..<n\}\) also in the special form for \(\{..<n\}\).

This congruence rule should be used for sums over intervals as the standard

\(\text{sum.cong}\) does not work well with the simplifier who adds the unsimplified premise \(x \in B\) to the context.

context comm-monoid-set

begin

lemma zero-middle:

assumes \(1 \leq p \land k \leq p\)

shows \(F ((\lambda j. \text{if } j < k \text{ then } g \ j \text{ else if } j = k \text{ then } 1 \text{ else } h \ (j - \text{Suc } 0))) \{..p\} = F ((\lambda j. \text{if } j < k \text{ then } g \ j \text{ else } h \ j) \{..p - \text{Suc } 0\} \ (\text{is \ ?lhs = \ ?rhs})

proof

have \([\text{simp}]: \{..p - \text{Suc } 0\} \cap \{j. j < k\} = \{..<k\} \{..p - \text{Suc } 0\} \cap \{j. j < k\} = \{k..p - \text{Suc } 0\}\)
using assms by auto
have ?lhs = \( F \ g \{..<k\} \ast \ F \ (\lambda j. \text{if } j = k \text{ then } 1 \text{ else } h \ (j - \text{Suc } 0)) \ \{k..p\}\)
  using union-disjoint [of \{..<k\} \{k..p\}] assms
  by (simp add: ivl-disj-int-one ivl-disj-un-one)
also have \(\ldots = F \ g \{..<k\} \ast \ F \ h \ \{k..p - \text{Suc } 0\}\)
  using reindex [of Suc \{k..p - \text{Suc } 0\}] assms by simp
also have \(\ldots = ?rhs\)
  by (simp add: If-cases)
finally show ?thesis.
qed

lemma atMost-Suc [simp]:
  \( F \ g \{..\text{Suc } n\} = F \ g \{..n\} \ast g \ (\text{Suc } n)\)
  by (simp add: atMost-Suc ac-simps)

lemma lessThan-Suc [simp]:
  \( F \ g \{..<\text{Suc } n\} = F \ g \{..<n\} \ast g \ n\)
  by (simp add: lessThan-Suc ac-simps)

lemma cl-ivl-Suc [simp]:
  \( F \ g \{m..\text{Suc } n\} = (\text{if } \text{Suc } n < m \text{ then } 1 \text{ else } F \ g \{m..<n\} \ast g(\text{Suc } n))\)
  by (auto simp: ac-simps atLeastAtMostSuc-conv)

lemma op-ivl-Suc [simp]:
  \( F \ g \{m..<\text{Suc } n\} = (\text{if } n < m \text{ then } 1 \text{ else } F \ g \{m..<n\} \ast g(n))\)
  by (auto simp: ac-simps atLeastLessThanSuc)

lemma head:
  fixes \(n::\text{nat}\)
  assumes mn: \(m \leq n\)
  shows \( F \ g \{m..n\} = g \ m \ast F \ g \{m..<n\}\) (is \(?lhs = ?rhs\))
proof
  from mn
  have \( \{m..n\} = \{m\} \cup \{m..<n\} \)
    by (auto intro: ivl-disj-un-singleton)
  hence \(?lhs = F \ g \{(m\} \cup \{m..<n\}\)\)
    by (simp add: atLeast0LessThan)
  also have \(\ldots = ?rhs\) by simp
  finally show ?thesis.
qed

lemma ub-add-nat:
  assumes \(m::\text{nat}\) \(\leq n + 1\)
  shows \( F \ g \{m..n + p\} = F \ g \{m..n\} \ast F \ g \{n + 1..n + p\}\)
proof
  have \( \{m..n+p\} = \{m..n\} \cup \{n+1..n+p\}\) using \(m \leq n + 1\) by auto
  thus ?thesis by (auto simp: ivl-disj-int union-disjoint atLeastSucAtMost-greaterThanAtMost)
qed

lemma nat-group:
  fixes k::nat shows F (\( \lambda m. F \) g \{m * k ..< m * k + k\}) \{..<n\} = F g \{..< n * k\}
proof (cases k)
  case (Suc l)
  then have k > 0
  by auto
  then show \(?thesis\)
  by (induct n) (simp-all add: atLeastLessThan-concat add.commute atLeast0LessThan[symmetric])
qed auto

lemma triangle-reindex-eq:
  fixes n :: nat
  shows F (\( \lambda (i,j). g \) i j \{\( i, j \). \( i+j < n\)\}) \{\( ..<n\)\} = F (\( \lambda k. F \) (\( \lambda i. g \) i (k - i)) \{\( ..\)k\})
  apply (simp add: Sigma)
  apply (rule reindex-bij-witness[where j=\( \lambda (i, \( j \)) (i+j, i)\) and \( i=\lambda (k, i). (i, k - i)\)])
  apply auto
  done

lemma triangle-reindex-eq:
  fixes n :: nat
  shows F (\( \lambda (i,j). g \) i j \{\( i, j \). \( i+j \leq n\)\}) \{\( ..\)n\} = F (\( \lambda k. F \) (\( \lambda i. g \) i (k - i)) \{\( ..\)k\})
  using triangle-reindex [of g Suc n]
  by (simp only: Nat.less-Suc-eq-le lessThan-Suc-atMost)

lemma nat-diff-reindex: F (\( \lambda i. g \) (n - Suc i)) \{..<n\} = F g \{..<n\}
  by (rule reindex-bij-witness[where \( i=\lambda i. n - Suc i\) and \( j=\lambda i. n - Suc i\)]) auto

lemma shift-bounds-nat-ivl:
  F g \{m+k..<n+k\} = F (\( \lambda i. g(i + k)\))\{m..<n+:nat\}
  by (induct n, auto simp: atLeastLessThanSuc)

lemma shift-bounds-cl-nat-ivl:
  F g \{m+k..n+k\} = F (\( \lambda i. g(i + k)\))\{m..n+:nat\}
  by (rule reindex-bij-witness[where \( i=\lambda i. i + k\) and \( j=\lambda i. i - k\)]) auto

corollary shift-bounds-cl-Suc-ivl:
  F g \{Suc m..Suc n\} = F (\( \lambda i. g(Suc i)\))\{m..n\}
  by (simp add: shift-bounds-cl-nat-ivl[where \( k=Suc 0\), simplified])

corollary Suc-reindex-ivl: m \leq n \implies F g \{m..n\} \ast g (Suc n) = g m * F (\( \lambda i. g(Suc i)\))\{m..n\}
  by (simp add: assoc atLeast-Suc-atMost flip: shift-bounds-cl-Suc-ivl)
corollary shift-bounds-Suc-ivl:
  \( F \cdot g \{ \text{Suc } m.. \text{Suc } n \} = F (\lambda i. g(\text{Suc } i))\{m..n\} \)
by (simp add: shift-bounds-nat-ivl[where k=Suc 0, simplified])

lemma atMost-Suc-shift:
  shows \( F \cdot g \{.. \text{Suc } n \} = g 0 * F (\lambda i. g (\text{Suc } i)) \{..n\} \)
proof (induct n)
  case 0 show \( \text{Suc } 0 \) by (rule atMost-Suc)
next
  case (Suc n) note IH = this
  have \( F \cdot g \{.. \text{Suc } (\text{Suc } n) \} = F \cdot g \{.. \text{Suc } n \} * g (\text{Suc } (\text{Suc } n)) \)
  by (rule atMost-Suc)
  also have \( F \cdot g \{.. \text{Suc } n \} = g 0 * F (\lambda i. g (\text{Suc } i)) \{..n\} \)
  by (rule IH)
  also have \( g 0 * F (\lambda i. g (\text{Suc } i)) \{..n\} * g (\text{Suc } (\text{Suc } n)) \)
  by (rule assoc)
  also have \( F (\lambda i. g (\text{Suc } i)) \{..n\} * g (\text{Suc } (\text{Suc } n)) = F (\lambda i. g (\text{Suc } i)) \{.. \text{Suc } n \} \)
  by (rule atMost-Suc [symmetric])
  finally show \( \text{Suc } n \) by (rule atMost-Suc)
qed

lemma lessThan-Suc-shift:
  \( F \cdot g \{..\text{Suc } n \} = g 0 * F (\lambda i. g (\text{Suc } i)) \{..n\} \)
by (induction n) (simp-all add: ac-simps)

lemma atMost-shift:
  \( F \cdot g \{..n\} = g 0 * F (\lambda i. g (\text{Suc } i)) \{..n\} \)
by (metis atLeast0AtMost atLeast0LessThan atLeastLessThanSuc-atLeastAtMost)
  atLeastSucAtMost-greaterThanAtMost le0 head shift-bounds-Suc-ivl)

lemma last-plus:
  fixes n::nat shows \( m \leq n \implies F \cdot g \{m..n\} = g n * F \cdot g \{m..n\} \)
by (cases n) (auto simp: atLeastLessThanSuc-atLeastAtMost commute)

lemma nested-swap:
  \( F (\lambda i. F (\lambda j. a i j) \{0..i\}) \{0..n\} = F (\lambda j. F (\lambda i. a i j) \{\text{Suc } j..n\}) \{0..n\} \)
by (induction n) (auto simp: distrib)

lemma nested-swap':
  \( F (\lambda i. F (\lambda j. a i j) \{..i\}) \{n\} = F (\lambda j. F (\lambda i. a i j) \{\text{Suc } j..n\}) \{..n\} \)
by (induction n) (auto simp: distrib)

lemma atLeast1-atMost-eq:
  \( F \cdot g \{\text{Suc } 0..n\} = F (\lambda k. g (\text{Suc } k)) \{..n\} \)
proof -
  have \( F \cdot g \{\text{Suc } 0..n\} = F \cdot g (\text{Suc } \cdot \{..n\}) \)
THEORY “Set-Interval”

by (simp add: image-Suc-lessThan)
also have \dots = F (\lambda k. g (Suc k)) \{..<n\}
  by (simp add: reindex)
finally show \?thesis .
qed

lemma atLeastLessThan-Suc: \(a \leq b \implies F g \{a..<Suc b\} = F g \{a..<b\} \ast g b\)
  by (simp add: atLeastLessThanSuc commute)

lemma nat-ivl-Suc':
  assumes \(m \leq Suc n\)
  shows \(F g \{m..Suc n\} = g (Suc n) \ast F g \{m..n\}\)
proof
  from assms have \(\{m..Suc n\} = insert (Suc n) \{m..n\}\) by auto
  also have \(F g \dots = g (Suc n) \ast F g \{m..n\}\) by simp
  finally show \?thesis .
qed

lemma in-pairs: \(F g \{2* m..Suc(2*n)\} = F (\lambda i. g(2*i) \ast g(Suc(2*i))) \{m..n\}\)
proof (induction n)
  case 0
  show \?case by (cases m=0) auto
next
  case (Suc n)
  then show \?case by (auto simp: assoc split: if-split-asm)
qed

lemma in-pairs-0: \(F g \{..Suc(2*n)\} = F (\lambda i. g(2*i) \ast g(Suc(2*i))) \{..n\}\)
using in-pairs [of - 0 n] by (simp add: atLeast0AtMost)

end

lemma sum-natinterval-diff:
  fixes f :: 'a::nat
  shows \(\sum (\lambda k. f k - f(k+1)) \{(m::nat).. n\} = \) (if \(m \leq n\) then \(f m - f(n + 1)\) else 0)
by (induct n, auto simp: algebra-simps not-le le-Suc-eq)

lemma sum-diff-nat-ivl:
  fixes f :: 'a::ab-group-add
  shows \[ m \leq n; n \leq p \] \implies \(\sum f \{m..<p\} - \sum f \{m..<n\} = \sum f \{n..<p\}\)
using sum.atLeastLessThan-concat [of m n p f,symmetric]
by (simp add: ac-simps)

lemma sum-diff-distrib: \(\forall x. Q x \leq P x \implies (\sum x<n. P x) - (\sum x<n. Q x) = (\sum x<n. P x - Q x :: nat)\)
by (subst sum-subtractf-nat) auto
context unique-euclidean-semiring-with-bit-shifts
begin

lemma take-bit-sum:
    take-bit n a = (∑ k = 0..<n. push-bit k (of-bool (bit a k)))
for n :: nat
proof (induction n arbitrary: a)
case 0
  then show ?case by simp
next
case (Suc n)
  have (∑ k = 0..<Suc n. push-bit k (of-bool (bit a k))) =
    of-bool (odd a) + (∑ k = Suc 0..<Suc n. push-bit k (of-bool (bit a k)))
    by (simp add: sum.atLeast-Suc-lessThan ac-simps)
  also have (∑ k = Suc 0..<Suc n. push-bit k (of-bool (bit a k)))
    = (∑ k = 0..<n. push-bit k (of-bool (bit (a div 2) k))) * 2
    by (simp only: sum.atLeast-Suc-lessThan-Suc-shift (simp add: sum-distrib-right
    push-bit-double drop-bit-Suc bit-Suc)
finally show ?case
  using Suc [of a div 2] by (simp add: ac-simps take-bit-Suc)
qed

60.9.1 Shifting bounds

context comm-monoid-add
begin

context
  fixes f :: nat ⇒ 'a
  assumes f 0 = 0
begin

lemma sum-shift-lb-Suc0-0-upt:
  sum f {Suc 0..<Suc k} = sum f {0..<k}
proof (cases k)
case 0
  then show ?thesis by simp
next
case (Suc k)
  moreover have {0..<Suc k} = insert 0 {Suc 0..<Suc k}
    by auto
  ultimately show ?thesis
    using f 0 = 0 by simp
qed

lemma sum-shift-lb-Suc0-0: sum f {Suc 0..k} = sum f {0..k}
proof (cases k)
  case 0
  with ⟨f 0 = 0⟩ show ?thesis
  by simp
next
  case (Suc k)
  moreover have {0..Suc k} = insert 0 {Suc 0..Suc k}
  by auto
  ultimately show ?thesis
  using ⟨f 0 = 0⟩ by simp
qed

end

end

lemma sum-Suc-diff:
  fixes f :: nat ⇒ 'a::ab-group-add
  assumes m ≤ Suc n
  shows (∑ i = m..n. f(Suc i) − f i) = f (Suc n) − f m
using assms by (induct n) (auto simp: le-Suc-eq)

lemma sum-Suc-diff':
  fixes f :: nat ⇒ 'a::ab-group-add
  assumes m ≤ n
  shows (∑ i = m..<n. f (Suc i) − f i) = f n − f m
using assms by (induct n) (auto simp: le-Suc-eq)

60.9.2 Telescoping

lemma sum-telescope:
  fixes f :: nat ⇒ 'a::ab-group-add
  shows (∑ i. f i − f (Suc i)) {.. i} = f 0 − f (Suc i)
  by (induct i) simp-all

lemma sum-telescope'::
  assumes m ≤ n
  shows (∑ k∈{Suc m..n}. f k − f (k − 1)) = f n − (f m :: 'a :: ab-group-add)
  by (rule dec-induct[OF assms]) (simp-all add: algebra-simps)

lemma sum-lessThan-telescope:
  (∑ n<m. f (Suc n) − f n :: 'a :: ab-group-add) = f m − f 0
  by (induction m) (simp-all add: algebra-simps)

lemma sum-lessThan-telescope'::
  (∑ n<m. f n − f (Suc n) :: 'a :: ab-group-add) = f 0 − f m
by (induction m) (simp-all add: algebra-simps)

60.9.3 The formula for geometric sums

lemma sum-power2: \( \sum_{i=0..<k} (2::nat)^i = 2^k - 1 \)
by (induction k) (auto simp: mult-2)

lemma geometric-sum:
  assumes \( x \neq 1 \)
  shows \( \sum_{i<n} x ^ i = (x ^ n - 1) / (x - 1 :: 'a :: field) \)
proof
  from assms obtain y where \( y = x - 1 \) and \( y \neq 0 \) by simp-all
  moreover have \( \sum_{i<n} (y + 1) ^ i = ((y + 1) ^ n - 1) / y \)
  by (induct n) (simp-all add: field-simps \( y \neq 0 \))
  ultimately show ?thesis by simp
qed

lemma diff-power-eq-sum:
  fixes \( y :: 'a :: \{ comm-ring, monoid-mult \} \)
  shows \( x ^ (Suc n) - y ^ (Suc n) = (x - y) * \sum_{p<Suc n} (x ^ p) * y ^ (n - p) \)
proof (induct n)
  case (Suc n)
  have \( x ^ Suc (Suc n) - y ^ Suc (Suc n) = x * (x ^ Suc n) - y * (y ^ Suc n) \)
  by simp
  also have \( ... = y * (x ^ Suc n - y ^ Suc n) + (x - y) * (x ^ Suc n) \)
  by (simp add: algebra-simps)
  also have \( ... = y * ((x - y) * \sum_{p<Suc n} (x ^ p) * y ^ (n - p)) + (x - y) * (x ^ Suc n) \)
  by (simp only: Suc)
  also have \( ... = (x - y) * (y * \sum_{p<Suc n} (x ^ p) * y ^ (n - p)) + (x - y) * (x ^ Suc n) \)
  by (simp only: mult.left-commute)
  also have \( ... = (x - y) * \sum_{p<Suc n} (x ^ p) * y ^ (Suc n - p) \)
  by (simp add: field-simps Suc-diff-le sum-distrib-right sum-distrib-left)
  finally show ?case .
qed simp

corollary power-diff-sumr2: — COMPLEX-POLYFUN in HOL Light
  fixes \( x :: 'a :: \{ comm-ring, monoid-mult \} \)
  shows \( x ^ n - y ^ n = (x - y) * \sum_{i<n} (n - Suc i) * x ^ i \)
using diff-power-eq-sum[of \( x ^ n - 1 \) \( y \)]
by (cases \( n = 0 \)) (simp-all add: field-simps)

lemma power-diff-1-eq:
  fixes \( x :: 'a :: \{ comm-ring, monoid-mult \} \)
  shows \( n \neq 0 \implies x ^ n - 1 = (x - 1) * \sum_{i<n} (x ^ i) \)
using diff-power-eq-sum [of \( x ^ 1 - 1 \) \( 0 \)]
lemma one-diff-power-eq:\[ \text{fixes } x :: 'a :: \{ \text{comm-ring}, \text{monoid-mult} \} \]
shows \[ n \neq 0 \Rightarrow 1 - x^n = (1 - x) \times (\sum_{i < n} x^i) \]
using diff-power-eq-sum [of 1 - x]
by (cases n) auto

lemma one-diff-power-eq:\[ \text{fixes } x :: 'a :: \{ \text{comm-ring}, \text{monoid-mult} \} \]
shows \[ n \neq 0 \Rightarrow 1 - x^n = (1 - x) \times (\sum_{i < n} x^i) \]
by (metis one-diff-power-eq[of n x] sum.nat-diff-reindex)

lemma sum-gp-basic:\[ \text{fixes } x :: 'a :: \{ \text{comm-ring}, \text{monoid-mult} \} \]
shows \[ (1 - x) \times (\sum_{i \leq n} x^i) = 1 - x^{Suc n} \]
by (simp only: one-diff-power-eq[of Suc n x] lessThan-Suc-atMost)

lemma sum-power-shift:\[ \text{fixes } x :: 'a :: \{ \text{comm-ring}, \text{monoid-mult} \} \]
assumes \[ m \leq n \]
shows \[ (\sum_{i = m}^n x^i) = x^m \times (\sum_{i < n-m} x^i) \]
proof -
  have \[ (\sum_{i = m}^n x^i) = x^m \times (\sum_{i = m}^n x^{i-m}) \]
    by (simp add: sum-distrib-left power-add [symmetric])
  also have \[ (\sum_{i = m}^n x^{i-m}) = (\sum_{i < n-m} x^i) \]
    using \( m \leq n \) by (intro sum.reindex-bij-witness[where j=\lambda i. i - m and i=\lambda i. i + m]) auto
  finally show \( \text{thesis} \).
qed

lemma sum-gp-multiplied:\[ \text{fixes } x :: 'a :: \{ \text{comm-ring}, \text{monoid-mult} \} \]
assumes \[ m \leq n \]
shows \[ (1 - x) \times (\sum_{i = m}^n x^i) = x^m - x^{Suc n} \]
proof -
  have \[ (1 - x) \times (\sum_{i = m}^n x^i) = x^m \times (1 - x) \times (\sum_{i < n-m} x^i) \]
    by (metis mult.assoc mult.commute assms sum-power-shift)
  also have \[ ... = x^m \times (1 - x^{Suc(n-m)}) \]
    by (metis mult.assoc sum-gp-basic)
  also have \[ ... = x^m - x^{Suc n} \]
    using assms
    by (simp add: algebra-simps) (metis le-add-diff-inverse power-add)
  finally show \( \text{thesis} \).
qed

lemma sum-gp:\[ \text{fixes } x :: 'a :: \{ \text{comm-ring}, \text{division-ring} \} \]
shows \[ (\sum_{i = m}^n x^i) = \]
THEORY "Set-Interval"

(if n < m then 0
  else if x = 1 then of-nat((n + 1) − m)
  else (x ^ m − x ^ Suc n) / (1 − x))
using sum-gp-multiplied[of m n x] apply auto
by (metis eq-iff-diff-eq-0 mult.commute nonzero-divide-eq-eq)

60.9.4 Geometric progressions

lemma sum-gp0:
  fixes x :: 'a::{comm-ring,division-ring}
  shows (∑ i≤n. x ^ i) = (if x = 1 then of-nat n + 1 else (1 − x ^ Suc n) / (1 − x))
  using sum-gp-basic[of x n]
by (simp add: mult.commute field-split-simps)

lemma sum-power-add:
  fixes x :: 'a::{comm-ring,monoid-mult}
  shows (∑ i∈I. x ^ (m + i)) = x ^ m * (∑ i∈I. x ^ i)
by (simp add: sum-distrib-left power-add)

lemma sum-gp-offset:
  fixes x :: 'a::{comm-ring,division-ring}
  shows (∑ i=m..m+n. x ^ i) =
    (if x = 1 then of-nat n + 1 else x ^ m * (1 − x ^ Suc n) / (1 − x))
  using sum-gp[of x m m+n]
by (auto simp: algebra-simps field-split-simps)

lemma sum-gp-strict:
  fixes x :: 'a::{comm-ring,division-ring}
  shows (∑ i<n. x ^ i) = (if x = 1 then of-nat n else (1 − x ^ n) / (1 − x))
by (induct n) (auto simp: algebra-simps field-split-simps)

60.9.5 The formulae for arithmetic sums

context comm-semiring-1
begin

lemma double-gauss-sum:
  2 * (∑ i = 0..n. of-nat i) = of-nat n * (of-nat n + 1)
by (induct n) (simp-all add: sum.atLeast0-atMost-Suc algebra-simps left-add-twice)

lemma double-gauss-sum-from-Suc-0:
  2 * (∑ i = Suc 0..n. of-nat i) = of-nat n * (of-nat n + 1)
proof –
  have sum.of-nat {Suc 0..n} = sum.of-nat (insert 0 {Suc 0..n})
    by simp
  also have . . . = sum.of-nat {0..n}
    by (cases n) (simp-all add: atLeast0-atMost-Suc-eq-insert-0)
finally show ?thesis
  by (simp add: double-gauss-sum)
qed

lemma double-arith-series:
\[ 2 \times \left( \sum_{i=0}^{n} a + \text{of-nat } i \times d \right) = (\text{of-nat } n + 1) \times (2 \times a + \text{of-nat } n \times d) \]
proof
- have \( \left( \sum_{i=0}^{n} a + \text{of-nat } i \times d \right) = \left( \sum_{i=0}^{n} a \right) + \left( \sum_{i=0}^{n} \text{of-nat } i \times d \right) \)
  by (rule sum.distrib)
also have \( \ldots = (\text{of-nat } (\text{Suc } n) \times a + d \times (\sum_{i=0}^{n} \text{of-nat } i)) \)
  by (simp add: sum-distrib-left algebra-simps)
finally show \?thesis
  by (simp add: algebra-simps double-gauss-sum left-add-twice)
qed

context unique-euclidean-semiring-with-nat
begin

lemma gauss-sum:
\( \left( \sum_{i=0}^{n} \text{of-nat } i \right) = \text{of-nat } n \times (\text{of-nat } n + 1) \div 2 \)
using double-gauss-sum [of n, symmetric] by simp

lemma gauss-sum-from-Suc-0:
\( \left( \sum_{i=\text{Suc } \mathbf{0}}^{n} \text{of-nat } i \right) = \text{of-nat } n \times (\text{of-nat } n + 1) \div 2 \)
using double-gauss-sum-from-Suc-0 [of n, symmetric] by simp

lemma arith-series:
\( \left( \sum_{i=0}^{n} a + \text{of-nat } i \times d \right) = (\text{of-nat } n + 1) \times (2 \times a + \text{of-nat } n \times d) \div 2 \)
using double-arith-series [of a d n, symmetric] by simp

end

lemma gauss-sum-nat:
\( \sum \{0..n\} = (n \times \text{Suc } n) \div 2 \)
using gauss-sum [of n, where ?'a = nat] by simp

lemma arith-series-nat:
\( \left( \sum_{i=0}^{n} a + \text{of-nat } i \times d \right) = \text{Suc } n \times (2 \times a + n \times d) \div 2 \)
using arith-series [of a d n] by simp

lemma Sum-Icc-int:
\( \sum \{m..n\} = (n \times (n + 1) - m \times (m - 1)) \div 2 \)
if \( m \leq n \) for \( m n :: \text{int} \)
using that proof (induct i \equiv nat (n - m) arbitrary: m n)
case \( \emptyset \)
then have \( m = n \)
  by arith
then show \?case
by (simp add: algebra-simps mult-2 [symmetric])

next
  case (Suc i)
  have 0: \(i = \text{nat}((n-1) - m)\) \(m \leq n-1\) using Suc(2,3) by arith+
  have \(\sum \{m..n\} = \sum \{m..I+(n-1)\}\) by simp
  also have \(\ldots = \sum \{m..n-I\} + n\) using \(m \leq n\):
    by (subst atLeastAtMostPlus1-int-conv) simp-all
  also have \(\ldots = ((n-I)*(n-I+1) - m*(m-I))\) \(\text{div}~2 + n\)
    by (simp add: Suc(2,3)[OF 0])
  also have \(\ldots = ((n-I)*(n-I+1) - m*(m-I) + 2*n)\) \(\text{div}~2\) by simp
  also have \(\ldots = (n*(n+1) - m*(m-I))\) \(\text{div}~2\)
    by (simp add: algebra-simps mult-2-right)
  finally show \(?case\).

qed

lemma Sum-Icc-nat:
\(\sum \{m..n\} = (n * (n + 1) - m * (m - 1))\) \(\text{div}~2\) for \(m n::\text{nat}\)

proof (cases \(m \leq n\))
  case True
  then have*: \(m * (m - 1) \leq n * (n + 1)\)
    by (meson diff-le-self order-trans le-add1 mult-le-mono)
  have \(\text{int} \\{\sum \{m..n\}\} = (\sum \{\text{int} m..\text{int} n\}\)
    by (simp add: sum.atLeast-atMost-int-shift)
  also have \(\ldots = \text{int} (n * (n + 1) - \text{int} m * (\text{int} m - 1))\) \(\text{div}~2\)
    using \(m \leq n\) by (simp add: Sum-Icc-int)
  also have \(\ldots = \text{int} ((n + 1) * (n - 1) - m * (m - 1))\) \(\text{div}~2\)
    using le-square * by (simp add: algebra-simps of-nat-div of-nat-diff)
  finally show ?thesis
    by (simp only: of-nat-eq-iff)
next
  case False
  then show ?thesis
    by (auto dest: less-imp-Suc-add simp add: not-le algebra-simps)

qed

lemma Sum-Ico-nat:
\(\sum \{m..<n\} = (n * (n - 1) - m * (m - 1))\) \(\text{div}~2\) for \(m n::\text{nat}\)

by (cases \(n\)) (simp-all add: atLeastLessThanSuc-atLeastAtMost Sum-Icc-nat)

60.9.6 Division remainder

lemma range-mod:
  fixes \(n::\text{nat}\)
  assumes \(n > 0\)
  shows \(\text{range}~(\lambda m. m \text{ mod} n) = \{0..<n\}\) \(\text{is} \ ?A = \ ?B\)

proof (rule set-eqI)
  fix \(m\)
  show \(m \in \ ?A \longleftrightarrow m \in \ ?B\)

proof
assume \( m \in \mathcal{A} \)

with assms show \( m \in \mathcal{B} \)

by auto

next

assume \( m \in \mathcal{B} \)

moreover have \( m \mod n \in \mathcal{A} \)

by (rule rangeI)

ultimately show \( m \in \mathcal{A} \)

by simp

qed

60.10 Products indexed over intervals

syntax (ASCII)

\[-\text{from-to-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\text{PROD} - = -\ldots/-) \ [0,0,0,10] \ 10)\]

\[-\text{from-upto-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\text{PROD} - = -\ldots/-) \ [0,0,0,10] \ 10)\]

\[-\text{upt-to-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\text{PROD} - = -\ldots/-) \ [0,0,0,10] \ 10)\]

syntax (latex-prod output)

\[-\text{from-to-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\exists \Pi_{- = -} [0,0,0,10] \ 10)\]

\[-\text{from-upto-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\exists \Pi_{- = -} [0,0,0,10] \ 10)\]

\[-\text{upt-to-prod} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\exists \Pi_{- = -} [0,0,0,10] \ 10)\]

translations

\[\prod x=a..b. \ t \ = \ \text{CONST \ prod} (\lambda x. \ t) \ {a..b}\]

\[\prod x=a..<b. \ t \ = \ \text{CONST \ prod} (\lambda x. \ t) \ {a..<b}\]

\[\prod i\leq n. \ t \ = \ \text{CONST \ prod} (\lambda i. \ t) \ {..n}\]

\[\prod i<n. \ t \ = \ \text{CONST \ prod} (\lambda i. \ t) \ {..<n}\]

lemma prod-int-plus-eq: prod int \{i..i+j\} = \prod \{int i..int (i+j)\}

by (induct j) (auto simp add: atLeastAtMostSuc-conv atLeastAtMostPlus1-int-conv)

lemma prod-int-eq: prod int \{i..j\} = \prod \{int i..int j\}

proof (cases \( i \leq j \))

case True
then show \( \text{thesis} \)
  by (metis le-iff-add prod-int-plus-eq)
next
  case False
  then show \( \text{thesis} \)
    by auto
qed

60.11 Efficient folding over intervals

function fold-atLeastAtMost-nat where
  \[
  \text{fold-atLeastAtMost-nat } f \ a (\text{nat}) \text{ acc } = \\
  \begin{cases}
    \text{acc} & \text{if } a > b \\
    \text{fold-atLeastAtMost-nat } f (a+1) b (f \ a \text{ acc}) & \text{else}
  \end{cases}
  \]
by pat-completeness auto
termination by (relation measure \((\lambda (-,a,b,-). \text{Suc } b - a))\) auto

lemma fold-atLeastAtMost-nat:
  assumes comp-fun-commute \( f \)
  shows fold-atLeastAtMost-nat \( f \ a b \text{ acc } = \) Finite-Set.fold \( f \text{ acc } \{a..b\} \)
using assms
proof (induction \( f \ a b \text{ acc } \) rule: fold-atLeastAtMost-nat.induct, goal-cases)
  case (1 \( f \ a b \text{ acc } \))
  interpret comp-fun-commute \( f \) by fact
  show \( ?\text{thesis} \)
    proof (cases \( a > b \))
      case True
      thus \( ?\text{thesis} \)
        by (subst fold-atLeastAtMost-nat.simps) auto
    next
      case False
      with i show \( ?\text{thesis} \)
        by (subst fold-atLeastAtMost-nat.simps)
          (auto simp: atLeastAtMost-insertL[symmetric] fold-fun-left-comm)
  qed
qed

lemma sum-atLeastAtMost-code:
  sum \( f \ \{a..b\} = \) fold-atLeastAtMost-nat \( \lambda a \text{ acc. } f \ a + \text{ acc a b 0} \)
proof
  have \( \text{comp-fun-commute } (\lambda a. (\text{+}) (f \ a)) \)
    by unfold-locales (auto simp: o-def add-ac)
  thus \( ?\text{thesis} \)
    by (simp add: sum.eq-fold fold-atLeastAtMost-nat o-def)
qed

lemma prod-atLeastAtMost-code:
  prod \( f \ \{a..b\} = \) fold-atLeastAtMost-nat \( \lambda a \text{ acc. } f \ a \ast \text{ acc a b 1} \)
proof
  have \( \text{comp-fun-commute } (\lambda a. (*) (f \ a)) \)
    by unfold-locales (auto simp: o-def mult-ac)
61 Decision Procedure for Presburger Arithmetic

theory Presburger
imports Groebner-Basis Set-Interval
keywords try0 :: diag
begin

ML-file (Tools/Qelim/qelim.ML)
ML-file (Tools/Qelim/cooper-procedure.ML)

61.1 The $-\infty$ and $+\infty$ Properties

lemma minf:
$\exists (z :: 'a::linorder). \forall x < z. \ P x = P' x; \ \exists z. \forall x < z. \ Q x = Q' x$
\[ \implies \exists z. \forall x < z. \ (P x \land Q x) = (P' x \land Q' x) \]
$\exists (z :: 'a::linorder). \forall x < z. \ P x = P' x; \ \exists z. \forall x < z. \ Q x = Q' x$
\[ \implies \exists z. \forall x < z. \ (P x \lor Q x) = (P' x \lor Q' x) \]
$\exists (z :: 'a::linorder). \forall x < z. \ (x = t) = False$
$\exists (z :: 'a::linorder). \forall x < z. \ (x \neq t) = True$
$\exists (z :: 'a::linorder). \forall x < z. \ (x < t) = True$
$\exists (z :: 'a::linorder). \forall x < z. \ (x > t) = False$
$\exists (z :: 'a::linorder). \forall x < z. \ (x \geq t) = False$
$\exists z. \forall (x :: 'b::linorder, plus, Rings.ded) < z. \ (d dvd x + s) = (d dvd x + s)$
$\exists z. \forall (x :: 'b::linorder, plus, Rings.ded) > z. \ (\neg d dvd x + s) = (\neg d dvd x + s)$
$\exists z. \forall x < z. \ F = F$
by (erule exE, erule exE, rule-tac x=\min z za in exI, simp)+, (rule-tac x=t in exI, fastforce)+ simp-all

lemma pinf:
$\exists (z :: 'a::linorder). \forall x > z. \ P x = P' x; \ \exists z. \forall x > z. \ Q x = Q' x$
\[ \implies \exists z. \forall x > z. \ (P x \land Q x) = (P' x \land Q' x) \]
$\exists (z :: 'a::linorder). \forall x > z. \ P x = P' x; \ \exists z. \forall x > z. \ Q x = Q' x$
\[ \implies \exists z. \forall x > z. \ (P x \lor Q x) = (P' x \lor Q' x) \]
$\exists (z :: 'a::linorder). \forall x > z. \ (x = t) = False$
$\exists (z :: 'a::linorder). \forall x > z. \ (x \neq t) = True$
$\exists (z :: 'a::linorder). \forall x > z. \ (x < t) = False$
$\exists (z :: 'a::linorder). \forall x > z. \ (x \leq t) = True$
$\exists (z :: 'a::linorder). \forall x > z. \ (x > t) = True$
$\exists (z :: 'a::linorder). \forall x > z. \ (x \geq t) = True$
$\exists z. \forall (x :: 'b::linorder, plus, Rings.ded) > z. \ (d dvd x + s) = (d dvd x + s)$
THEORY "Presburger"

1150

\[ \exists z. \forall (x, y, z): \{ \text{linorder, plus, Rings, dvd} \} \rightarrow (\neg d \text{ dvd } x + s) = (\neg d \text{ dvd } x + s) \]

\[ \exists z. \forall x > z. F = F \]

by (\text{erule exE, erule exE, rule-tac } x = \text{max } z \text{ za in } \text{exI, simp} +, \text{ (rule-tac } x = t \text{ in exI, fastforce} +) \text{ simp-all}]

\begin{itemize}
  \item \text{lemmas inf-period:}
    \[ \forall x k. P x = P (x - k + D); \forall x k. Q x = Q (x - k + D) \]
    \[ \Rightarrow \forall x k. (P x \land Q x) = (P (x - k + D) \land Q (x - k + D)) \]
    \[ \forall x k. P x = P (x - k + D); \forall x k. Q x = Q (x - k + D) \]
    \[ \Rightarrow \forall x k. (P x \lor Q x) = (P (x - k + D) \lor Q (x - k + D)) \]
    \[ (d: \text{a: [comm-ring, Rings, dvd]}) \text{ dvd } D \Rightarrow \forall x k. (d \text{ dvd } x + t) = (d \text{ dvd } (x - k + D) + t) \]
    \[ (d: \text{a: [comm-ring, Rings, dvd]}) \text{ dvd } D \Rightarrow \forall x k. (\neg d \text{ dvd } x + t) = (\neg d \text{ dvd } (x - k + D) + t) \]
    \[ \forall x k. F = F \]
    \text{apply} (\text{auto elim!}: \text{dvdE simp add: algebra-simps})
  \item \text{unfolding mult.assoc [symmetric] distrib-right [symmetric] left-diff-distrib [symmetric]}
  \item \text{unfolding dvd-def mult.associate [of d]}
  \end{itemize}

by auto

61.2 The A and B sets

\begin{itemize}
  \item \text{lemmas set:}
    \[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in B. x \neq b + j) \rightarrow P x \rightarrow P(x - D) \]
    \[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in B. x \neq b + j) \rightarrow Q x \rightarrow Q(x - D) \]
    \[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in B. x \neq b + j) \rightarrow (P x \land Q x) \rightarrow (P(x - D) \land Q(x - D)) \]
    \[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in B. x \neq b + j) \rightarrow (P x \lor Q x) \rightarrow (P(x - D) \lor Q(x - D)) \]
    \[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in B. x \neq b + j) \rightarrow (P x \Rightarrow Q x) \rightarrow (P(x - D) \Rightarrow Q(x - D)) \]
    \[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in B. x \neq b + j) \rightarrow (x = t) \rightarrow (x - D = t) \]
    \[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in B. x \neq b + j) \rightarrow (x = t) \rightarrow (x - D = t) \]
    \[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in B. x \neq b + j) \rightarrow (x < t) \rightarrow (x - D < t) \]
    \[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in B. x \neq b + j) \rightarrow (x \leq t) \rightarrow (x - D \leq t) \]
    \[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in B. x \neq b + j) \rightarrow (x > t) \rightarrow (x - D > t) \]
    \[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in B. x \neq b + j) \rightarrow (x \geq t) \rightarrow (x - D \geq t) \]
    \[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in B. x \neq b + j) \rightarrow (d \text{ dvd } x + t) \rightarrow (d \text{ dvd } (x - D) + t) \]
    \[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in B. x \neq b + j) \rightarrow (\neg d \text{ dvd } x + t) \rightarrow (\neg d \text{ dvd } (x - D) + t) \]
    \[ \forall x. (\forall j \in \{1 \ldots D\}. \forall b \in B. x \neq b + j) \rightarrow F \rightarrow F \]
\end{itemize}

\text{proof} (\text{blast, blast})
assumedp: D > 0 and tB: t − 1 ∈ B

show (∀x.(∀ j∈{1 .. D}. ∀ b∈B. x ≠ b + j) → (x = t) → (x − D = t))

apply (rule allI, rule implI, rule ballE[where x=1], erule ballE[where x=t−1])

apply algebra using dp tB by simp-all

next

assumedp: D > 0 and tB: t ∈ B

show (∀x.(∀ j∈{1 .. D}. ∀ b∈B. x ≠ b + j) → (x ≠ t) → (x − D ≠ t))

apply (rule allI, rule implI, rule ballE[where x=D], erule ballE[where x=t])

apply algebra

using dp tB by simp-all

next

assumedp: D > 0 thus (∀x.(∀ j∈{1 .. D}. ∀ b∈B. x ≠ b + j) → (x < t) → (x − D < t)) by arith

next

assumedp: D > 0 thus (forallx.∀ j∈{1 .. D}. ∀ b∈B. x ≠ b + j) → (x ≤ t) → (x − D ≤ t) by arith

next

assumedp: D > 0 and tB:t−1∈B

{fix x assume nob: ∀ j∈{1 .. D}. ∀ b∈B. x ≠ b + j and g: x > t and ng: ¬ (x − D) > t

hence x − t ≤ D and 1 ≤ x − t by simp+

hence ∃ j ∈ {1 .. D}. x − t = j by auto

hence ∃ j ∈ {1 .. D}. x = t + j by (simp add: algebra-simps)

with nob tB have False by simp}

thus ∀x.(∀ j∈{1 .. D}. ∀ b∈B. x ≠ b + j) → (x > t) → (x − D > t) by blast

next

assumedp: D > 0 and tB:t−1∈B

{fix x assume nob: ∀ j∈{1 .. D}. ∀ b∈B. x ≠ b + j and g: x ≥ t and ng: ¬ (x − D) ≥ t

hence x − (t − 1) ≤ D and 1 ≤ x − (t − 1) by simp+

hence ∃ j ∈ {1 .. D}. x − (t − 1) = j by auto

hence ∃ j ∈ {1 .. D}. x = (t − 1) + j by (simp add: algebra-simps)

with nob tB have False by simp}

thus ∀x.(∀ j∈{1 .. D}. ∀ b∈B. x ≠ b + j) → (x ≥ t) → (x − D ≥ t) by blast

next

assumed: d dvd D

{fix x assume H: d dvd x + t with d have d dvd (x − D) + t by algebra}

thus (forallx::int),(∀ j∈{1 .. D}. ∀ b∈B. x ≠ b + j) → (d dvd x+t) → (d dvd (x − D) + t) by simp

next

assumed: d dvd D

{fix x assume H: ¬(d dvd x + t) with d have ¬ d dvd (x − D) + t

by (clarsimp simp add: dvd-def, erule_tac x= ka + k in allE, simp add: algebra-simps)}

thus (forallx::int),(∀ j∈{1 .. D}. ∀ b∈B. x ≠ b + j) → (¬d dvd x+t) → (¬d dvd (x − D) + t) by auto

qed blast
lemma aset:
\[
\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow P x \rightarrow P(x + D);
\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow Q x \rightarrow Q(x + D) \]
\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow (P x \land Q x) \rightarrow (P(x + D) \land Q(x + D))
\]
\[
\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow P x \rightarrow P(x + D);
\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow Q x \rightarrow Q(x + D) \]
\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow (P x \lor Q x) \rightarrow (P(x + D) \lor Q(x + D))
\]
\[D > 0; t + 1 \in A \Rightarrow (\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow (x = t) \rightarrow (x + D = t))\]
\[D > 0; t \in A \Rightarrow (\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow (x = t) \rightarrow (x + D \neq t))\]
\[D > 0; t + 1 \in A \Rightarrow (\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow (x < t) \rightarrow (x + D < t))\]
\[D > 0 \Rightarrow (\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow (x > t) \rightarrow (x + D > t))\]
\[D > 0 \Rightarrow (\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow (x \geq t) \rightarrow (x + D \geq t))\]
\[d \text{ dvd } D \Rightarrow (\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow (d \text{ dvd } x + t) \rightarrow (d \text{ dvd } (x + D) + t))\]
\[\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow (x = t) \rightarrow (x + D = t)\)
\]
\[\text{proof (blast, blast)}\]
\[\text{assume dp: } D > 0 \text{ and } tA: t + 1 \in A\]
\[\text{show } (\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow (x = t) \rightarrow (x + D = t))\]
\[\text{apply (rule allI, rule impI, erule ballE[where } x = 1], erule ballE[where } x = t + 1])\]
\[\text{using dp } tA \text{ by simp-all}\]
\[\text{next}\]
\[\text{assume dp: } D > 0 \text{ and } tA: t \in A\]
\[\text{show } (\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow (x = t) \rightarrow (x + D \neq t))\]
\[\text{apply (rule allI, rule impI, erule ballE[where } x = D], erule ballE[where } x = t])\]
\[\text{using dp } tA \text{ by simp-all}\]
\[\text{next}\]
\[\text{assume dp: } D > 0 \text{ thus } (\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow (x > t) \rightarrow (x + D > t)) \text{ by arith}\]
\[\text{next}\]
\[\text{assume dp: } D > 0 \text{ thus } (\forall x. (\forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j) \rightarrow (x \geq t) \rightarrow (x + D \geq t)) \text{ by arith}\]
\[\text{next}\]
\[\text{assume dp: } D > 0 \text{ and } tA: t \in A\]
\[\{ \text{fix } x \text{ assume nob: } \forall j \in \{1 \cdots D\}, \forall b \in A. x \neq b - j \text{ and } g: x < t \text{ and } ng: \neg (x + D) < t\}\]
\[\text{hence } t - x \leq D \text{ and } 1 \leq t - x \text{ by simp+}\]
\[\text{hence } \exists j \in \{1 \cdots D\}. t - x = j \text{ by auto}\]
hence $\exists j \in \{1 \ldots D\}, \ x = t - j$ by (auto simp add: algebra-simps)
with nob $tA$ have False by simp
thus $\forall x.(\forall j \in \{1 \ldots D\}. \forall b \in A. \ x \neq b - j) \rightarrow (x < t) \rightarrow (x + D < t)$ by blast
next
assume $dp: D > 0$ and $tA:t + 1 \in A$
{fix $x$ assume nob: $\forall j \in \{1 \ldots D\}. \forall b \in A. \ x \neq b - j$ and $g: x \leq t$ and $ng: \neg(x + D) \leq t$}
hence $(t + 1) - x \leq D$ and $I \leq (t + 1) - x$ by (simp-all add: algebra-simps)
hence $\exists j \in \{1 \ldots D\}. \ (t + 1) - x = j$ by auto
hence $\exists j \in \{1 \ldots D\}. \ x = (t + 1) - j$ by (auto simp add: algebra-simps)
with nob $tA$ have False by simp
thus $\forall x.(\forall j \in \{1 \ldots D\}. \forall b \in A. \ x \neq b - j) \rightarrow (x \leq t) \rightarrow (x + D \leq t)$ by blast
next
assume $d: d \ dvd D$
{fix $x$ assume $H: d \ dvd x + t$ with $d$ have $d \ dvd (x + D) + t$}
by (clarsimp simp add: dvd_def,rule-tac $x = ka + k$ in $exI$,simp add: algebra-simps)
thus $\forall (x::int),(\forall j \in \{1 \ldots D\}. \forall b \in A. \ x \neq b - j) \rightarrow (d \ dvd x + t) \rightarrow (d \ dvd (x + D) + t)$ by simp
next
assume $d: d \ dvd D$
{fix $x$ assume $H: \neg(d \ dvd x + t)$ with $d$ have $\neg d \ dvd (x + D) + t$}
by (clarsimp simp add: dvd_def,erule-tac $x = ka + k$ in $allE$,simp add: algebra-simps)
thus $\forall (x::int),(\forall j \in \{1 \ldots D\}. \forall b \in A. \ x \neq b - j) \rightarrow (\neg d \ dvd x + t) \rightarrow (\neg d \ dvd (x + D) + t)$ by auto
qed blast

61.3 Cooper’s Theorem $-\infty$ and $+\infty$ Version

61.3.1 First some trivial facts about periodic sets or predicates

lemma periodic-finite-ex:
assumes $dpos: (0::int) < d$ and $modd: \forall x k. \ x = P(x - k \cdot d)$
shows $(\exists x. \ P x = (\exists j \in \{1..d\}. \ P j)$
(is $?LHS$ = $?RHS$)
proof
assume $?LHS$
then obtain $x$ where $P: P x$ ..
have $x \ mod \ d = x - (x \ div \ d) \cdot d$ by(simp add:mult-div-mod-eq symmetric ac-simps eq-diff-eq)
hence $P \ mod: P x = P(x \ mod \ d)$ using modd by simp
show $?RHS$
proof (cases)
assume $x \ mod \ d = 0$
hence $P \ 0$ using $P \ mod$ by simp
moreover have $P \ 0 = P(0 - (-1) \cdot d)$ using modd by blast
ultimately have $P \ d$ by simp
moreover have $d \in \{1..d\}$ using $dpos$ by simp
ultimately show $?RHS$ ..
next
  assume not0: \text{x mod d \neq 0}
  have \text{P(x mod d) using dpos P Pmod by simp}
  moreover have \text{x mod d \in \{1..d\}}
  proof
    from \text{dpos have 0 \leq x mod d by(rule pos-mod-sign)}
    moreover from \text{dpos have x mod d < d by(rule pos-mod-bound)}
    ultimately show \text{?thesis using not0 by simp}
  qed
  qed
qed

auto

61.3.2 The \(-\infty\) Version

lemma \text{decr-lemma}: \text{0 < (d::int) \implies x - (|x - z| + 1) \ast d < z}
by \text{(induct rule: int-gr-induct) (simp-all add: int-distrib)}

lemma \text{incr-lemma}: \text{0 < (d::int) \implies z < x + (|x - z| + 1) \ast d}
by \text{(induct rule: int-gr-induct) (simp-all add: int-distrib)}

lemma \text{decr-mult-lemma}:
  assumes dpos: \text{0 < d and minus: \forall x. P x \implies P(x - d)} and knneg: \text{0 <= k}
  shows \text{\forall x. P x \implies P(x - k \ast d)}
using knneg
proof \text{(induct rule:int-gr-induct)}
  case base thus \text{?case by simp}
next
  case \text{(step i)}
  \{ fix \text{x}
    have \text{P x \implies P (x - i \ast d) using step.hyps by blast}
    also have \ldots \implies P(x - (i + 1) \ast d) using minus[THEN spec, of x - i \ast d]
      by \text{(simp add: algebra-simps)}
    ultimately have \text{P x \implies P(x - (i + 1) \ast d) by blast}\}
  thus \text{?case ..}
qed

lemma \text{minusinfinity}:
  assumes dpos: \text{0 < d and}
  \text{P1eqP1: \forall x k. P1 x = P1(x - k \ast d) and eP1eqP1: \exists z::\text{int}. \forall x. x < z \implies (P x = P1 x)}
  shows \text{\exists x. P1 x \implies (\exists x. P x)}
proof
  assume eP1: \text{\exists x. P1 x ..}
  then obtain \text{x where P1: P1 x ..}
  from eP1 obtain \text{z where P1eqP1: \forall x. x < z \implies (P x = P1 x) ..}
  let \text{w = x - (|x - z| + 1) \ast d}
  from dpos have \text{\exists w < z by(rule decr-lemma)}
have $P1 \, x = P1 \, ?w$ using $P1eqP1$ by blast
also have $\ldots = P(?w)$ using $w \, P1eqP$ by blast
finally have $P \, ?w$ using $P1$ by blast
thus $\exists x. \, P \, x$ ..
qed

lemma $cpmi$:
assumes $dp: \, 0 < D$ and $p1: \exists z. \, \forall x < z. \, P \, x = P' \, x$
and $\neg b: \forall x. (\forall j \in \{1..D\}, \forall (b::int) \in B. \, x \neq b + j) \rightarrow P \, (x) \rightarrow P \, (x - D)$
and $pd: \forall \, x, \, k. \, P' \, x = P' \, (x - k + D)$
shows $(\exists x. \, P \, x) = ((\exists j \in \{1..D\} \cdot P' \, j) \lor (\exists j \in \{1..D\}, \exists b \in B. \, P \, (b + j)))$
(is $?L = (?R1 \lor ?R2)$)
proof
{assume $?R2$ hence $?L$ by blast}
moreover
{assume $H$: $?R1$ hence $?L$ using $\text{minusinfinity}[OF \, dp \, pd \, p1]$ $\text{periodic-finite-ex}[OF \, dp \, pd]$ by $\text{simp}$}
moreover
{fix $x$
  assume $P: \, P \, x$ and $H: \, \neg \, ?R2$
  {fix $y$
   assume $\neg (\exists j \in \{1..D\}, \exists b \in B. \, P \, (b + j))$ and $P: \, P \, y$
   hence $\neg (\exists j :: \text{int} \in \{1..D\}, \exists (b :: \text{int}) \in B. \, y = b + j)$ by auto
   with $\neg b \, P$ have $P \, (y - D)$ by auto
  }
  hence $\forall \, x. \, \neg (\exists j :: \text{int} \in \{1..D\}, \exists (b :: \text{int}) \in B. \, P \, (b + j)) \rightarrow P \, (x) \rightarrow P \, (x - D)$ by blast
  with $H \, P$ have $\text{th}: \, \forall \, x. \, P \, x \rightarrow P \, (x - D)$ by auto
  from $p1$ obtain $z$ where $z: \forall x. \, x < z \rightarrow (P \, x = P' \, x)$ by blast
  let $?y = x - ((|x - z| + 1) \cdot D$
  have $z p: 0 < (\, (|x - z| + 1) \, D$ by auto
  from $dp$ have $yz: \, ?y < z$ using $\text{decr-lemma}[OF \, dp]$ by auto
  from $z[\text{rule-format}, \, OF \, yz] \, \text{decr-malt-lemma}[OF \, dp \, th \, zp, \, \text{rule-format}, \, OF \, P]$
  have $\text{th2}: \, P' \, ?y$ by auto
  with $\text{periodic-finite-ex}[OF \, dp \, pd]$ have $?R1$ by blast
}
ultimately show $\forall \, x. \, P \, x$ ..
qed

61.3.3 The $+\infty$ Version

lemma $plusinfinity$:
assumes $dp: \, \text{pos}(0 :: \text{int}) < d$ and
$P1eqP1: \forall x. \, P' \, x = P' \, (x - k + d)$ and $eP1eqP1: \exists z. \, \forall \, x > z. \, P \, x = P' \, x$
shows $(\exists x. \, P' \, x) \rightarrow (\exists x. \, P \, x)$
proof
assume $eP1: \exists x. \, P' \, x$
then obtain $x$ where $P1: \, P' \, x$ ..
from $eP1eqP1$ obtain $z$ where $P1eqP: \, \forall \, x > z. \, P \, x = P' \, x$ ..
let $?w' = x + ((|x - z| + 1) \cdot d$
let $?w = x - ((|x - z| + 1) \cdot d$
have \( \exists x. P x \) ..

qed

lemma incr-mult-lemma:
assumes dpos: \( (0::\text{int}) < d \) and plus: \( \forall x::\text{int}. P x \rightarrow P(x + d) \) and knneg: 0 \(<= k \)
shows \( \forall x. P x \rightarrow P(x + k*d) \)
using knneg
proof (induct rule:int-ge-induct)
case base thus ?case by simp
next
case (step i)
fix x
have \( P x \rightarrow P (x + i * d) \) using step.hyps by blast
also have \( \ldots \rightarrow P(x + (i + 1) * d) \) using plus[THEN spec, of \( x + i * d \)]
  by (simp add:int-distrib ac-simps)
ultimately have \( P x \rightarrow P(x + (i + 1) * d) \) by blast
thus ?case ..
qed

lemma cppl: 0 < D and p1: \( \exists z. \forall x > z. P x = P' x \)
and nb: \( \forall x. (\forall j \in \{1..D\}. \forall (b::\text{int}) \in A. x \neq b - j) \rightarrow P (x) \rightarrow P (x + D) \)
and pd: \( \forall x k. P' x = P' (x-k*D) \)
shows \( (\exists x. P x) = ((\exists j \in \{1..D\} . P' j) \lor (\exists j \in \{1..D\}. \exists b \in A. P' (b - j))) \) (is \( ?L = (?R1 \lor ?R2) \))
proof-
{assume \( ?R2 \) hence \(?L \) by blast}
moreover
{assume H: \(?R1 \) hence \(?L \) using plusinfint[OF dp pd p1] periodic-finite-ex[OF dp pd] by simp}
moreover
{fix x
assume P: \( P x \) and H: \( \neg ?R2 \)
{fix y assume \( \neg(\exists j \in \{1..D\}. \exists b \in A. P (b - j)) \) and P: \( P y \)
  hence \( \neg(\exists (j::\text{int}) \in \{1..D\}. \exists (b::\text{int}) \in A. y = b - j) \) by auto
  with nb P have \( P (y + D) \) by auto }
  hence \( \forall x. \neg(\exists (j::\text{int}) \in \{1..D\}. \exists (b::\text{int}) \in A. P (b - j)) \rightarrow P (x) \rightarrow P (x + D) \) by blast
  with H P have th: \( \forall x. P x \rightarrow P (x + D) \) by auto
  from p1 obtain z where \( z: \forall x. x > z \rightarrow (P x = P' x) \) by blast
  let \(?y = x + ((|x - z| + 1)\times D \)
  have \( z p: 0 <= (|x - z| + 1) \) by arith
  from dp have \( yz: \forall y > z \) using incr-lemma[OF dp] by simp

THEORY "Presburger"

from z [rule-format, OF yz] incr-mult-lemma [OF dp th zp, rule-format, OF P]

have th2: P' ?y by auto
with periodic-finite-ex [OF dp pd]

have ?R1 by blast

ultimately show ?thesis by blast

qed

lemma simp-from-to: {i..j::int} = (if j < i then {} else insert i {i+1..j})
apply (simp add: atLeastAtMost-def atLeast-def atMost-def)
done

theorem unity-coeff-ex: (∃x::{semiring-0,Rings}.dvd). P (l * x) \equiv (∃x. l dvd (x + 0) \land P x)
apply (rule eq-reflection [symmetric])
defer
apply (erule exE)
apply (rule_tac x = l * x in exI)
apply (simp add: dvd-def)
apply (rule_tac x = x in exI, simp)
apply (erule conjE)
apply simp
apply (erule dvdE)
apply (rule_tac x = k in exI)
apply simp
done

lemma zdvd-mono:
fixes k m t :: int
assumes k ≠ 0
shows m dvd t \equiv k * m dvd k * t
using assms by simp

lemma uminus-dvd-conv:
fixes d t :: int
shows d dvd t \equiv - d dvd t and d dvd t \equiv d dvd - t
by simp-all

Theorems for transforming predicates on nat to predicates on int

lemma zdiff-int-split: P (int (x - y)) =
((y ≤ x \rightarrow P (int x - int y)) \land (x < y \rightarrow P 0))
by (cases y ≤ x) (simp-all add: of-nat-diff)

Specific instances of congruence rules, to prevent simplifier from looping.

theorem imp-le-cong:
[x = x'; 0 ≤ x' \implies P = P'] \implies (0 ≤ (x::int) \implies P) = (0 ≤ x' \implies P)
THEORY “Presburger”

by simp

theorem conj-le-cong:

\[ x = x' \land 0 \leq x' \implies P = P' \] \[ (0 \leq (x::int) \land P) = (0 \leq x' \land P') \]

by (simp cong: conj-cong)

ML-file ⟨Tools/Qelim/cooper.ML⟩

method-setup presburger = :

let

fan keyword k = Scan.lift (Args.$$k -- Args.colon) >> K ()
fan simple-keyword k = Scan.lift (Args.$$k) >> K ()
val addN = add
val delN = del
val elimN = elim
val any-keyword = keyword addN || keyword delN || simple-keyword elimN
val thms = Scan.repeats (Scan.unless any-keyword Attrib.multi-thm)
in

Scan.optional (simple-keyword elimN >> K false) true ---
Scan.optional (keyword addN |-- thms) [] ---
Scan.optional (keyword delN |-- thms) [] >>
(fn ((elim, add-ths), del-ths) => fn ctxt =>>
  SIMPLE-METHOD′ (Cooper.tac elim-ths del-ths ctxt))
end

Cooper’s algorithm for Presburger arithmetic

declare mod-eq-0-iff-dvd [presburger]
declare mod-by-Suc-0 [presburger]
declare mod-0 [presburger]
declare mod-by-1 [presburger]
declare mod-self [presburger]
declare div-by-0 [presburger]
declare mod-by-0 [presburger]
declare mod-div-trivial [presburger]
declare mult-div-mod-eq [presburger]
declare div-mult-mod-eq [presburger]
declare mod-mult-self1 [presburger]
declare mod-mult-self2 [presburger]
declare mod2-Suc-Suc [presburger]
declare not-mod-2-eq-0-eq-1 [presburger]
declare nat-zero-less-power-iff [presburger]

lemma [presburger, algebra]: m mod 2 = (1::nat) \iff \neg 2 dvd m by presburger
lemma [presburger, algebra]: m mod 2 = Suc 0 \iff \neg 2 dvd m by presburger
lemma [presburger, algebra]: m mod (Suc (Suc 0)) = (1::nat) \iff \neg 2 dvd m by presburger
lemma [presburger, algebra]: m mod (Suc (Suc 0)) = Suc 0 \iff \neg 2 dvd m by presburger
lemma [presburger, algebra]: m mod 2 = (1::int) \iff \neg 2 dvd m by presburger
context semiring-parity
begin

declare even-mult-iff [presburger]

declare even-power [presburger]

lemma [presburger]:
even (a + b) ⟷ even a ∧ even b ∨ odd a ∧ odd b
by auto

end

context ring-parity
begin

declare even-minus [presburger]

end

context linordered-idom
begin

declare zero-le-power-eq [presburger]

declare zero-less-power-eq [presburger]

declare power-less-zero-eq [presburger]

declare power-le-zero-eq [presburger]

end

declare even-Suc [presburger]

lemma [presburger]:
Suc n div Suc (Suc 0) = n div Suc (Suc 0) ⟷ even n
by presburger

declare even-diff-nat [presburger]

lemma [presburger]:
fixes k :: int
shows (k + 1) div 2 = k div 2 ⟷ even k
by presburger

lemma [presburger]:
fixes k :: int
shows \((k + 1) \div 2 = k \div 2 + 1 \leftrightarrow \text{odd } k\)
by presburger

lemma [presburger]:
\[
even n \leftrightarrow even (\text{int } n)
\]
by simp

61.4 Nice facts about division by 4::'a

lemma even-even-mod-4-iff:
\[
even (n::nat) \leftrightarrow even (n \mod 4)
\]
by presburger

lemma odd-mod-4-div-2:
\[
n \mod 4 = (3::nat) \Longrightarrow \text{odd } ((n - \text{Suc } 0) \div 2)
\]
by presburger

lemma even-mod-4-div-2:
\[
n \mod 4 = \text{Suc } 0 \Longrightarrow even ((n - \text{Suc } 0) \div 2)
\]
by presburger

61.5 Try0
ML-file ⟨Tools/try0.ML⟩
end

62 Bindings to Satisfiability Modulo Theories (SMT) solvers based on SMT-LIB 2

theory SMT
imports Divides
keywords smt-status :: diag
begin

62.1 A skolemization tactic and proof method

lemma choices:
\[
\begin{align*}
\& Q. \forall x. \exists y y a. Q x y y a \Longrightarrow \exists f fa. \forall x. Q x (f x) (fa x) \\
\& Q. \forall x. \exists y y a yb. Q x y y a yb \Longrightarrow \exists f fa fb. \forall x. Q x (f x) (fa x) (fb x) \\
\& Q. \forall x. \exists y y a yb yc. Q x y y a yb yc \Longrightarrow \exists f fa fb fc. \forall x. Q x (f x) (fa x) (fb x) (fc x) \\
\& Q. \forall x. \exists y y a yb yc yd. Q x y y a yb yc yd \Longrightarrow \exists f fa fb fc fd. \forall x. Q x (f x) (fa x) (fb x) (fc x) (fd x) \\
\& Q. \forall x. \exists y y a yb yc ye. Q x y y a yb yc ye \Longrightarrow \exists f fa fb fc fd fe. \forall x. Q x (f x) (fa x) (fb x) (fc x) (fd x) (fe x) \\
\& Q. \forall x. \exists y y a yb yc ye yf. Q x y y a yb yc ye yf \Longrightarrow \exists f fa fb fc fd fe ff. \forall x. Q x (f x) (fa x) (fb x) (fc x) (fd x) (fe x) (ff x) \\
\& Q. \forall x. \exists y y a yb yc ye yf yy. Q x y y a yb yc ye yf yy \Longrightarrow
\end{align*}
\]
THEORY “SMT” 1161

\[ \exists f \ fa \ fb \ fc \ fd \ fe \ ff \ fg \ \forall x. \ Q \ x \ (f \ x) \ (fa \ x) \ (fb \ x) \ (fc \ x) \ (fd \ x) \ (fe \ x) \ (ff \ x) \ (fg \ x) \]

by metis

lemma bchoices:
\[ \forall Q. \ \forall x \in S. \ \exists y \ ya \ yc. \ Q \ x \ ya \ yc \ \Rightarrow \ \exists f \ fa \ fb \ \forall x \in S. \ Q \ x \ (f \ x) \ (fa \ x) \ (fb \ x) \]

by metis

ML
\[
\text{fun moura-tac ctx =}
\text{Atomize-Elim.atomize-elim-tac ctx THEN}^
\text{SELECT-GOAL (Cla simp.auto-tac (ctx addSIs @\{thms choice choices bchoice bchoices\}) THEN)}^
\text{ALLGOALS (Metis-Tacticmetis-tac (take 1 ATP-Proof-Reconstruct.partial-type-encs) ATP-Proof-Reconstruct.default-metis-lam-trans ctx [] ORELSE' blast-tac ctx)}}
\]

method-setup moura = (Scan.succeed (SIMPLE-METHOD' o moura-tac))
\>
solve skolemization goals, especially those arising from Z3 proofs

hide-fact (open) choices bchoices

62.2 Triggers for quantifier instantiation

Some SMT solvers support patterns as a quantifier instantiation heuristics. Patterns may either be positive terms (tagged by "pat") triggering quantifier instantiations – when the solver finds a term matching a positive pattern, it instantiates the corresponding quantifier accordingly – or negative terms (tagged by "nopat") inhibiting quantifier instantiations. A list of patterns of the same kind is called a multipattern, and all patterns in a multipattern are considered conjunctively for quantifier instantiation. A list of multipatterns is called a trigger, and their multipatterns act disjunctively during quantifier instantiation. Each multipattern should mention at least all quantified variables of the preceding quantifier block.
THEORY "SMT"

typedef 'a symb-list

consts
  Symb-Nil :: 'a symb-list
  Symb-Cons :: 'a ⇒ 'a symb-list ⇒ 'a symb-list

typedecl pattern

consts
  pat :: 'a ⇒ pattern
  nopat :: 'a ⇒ pattern

definition trigger :: pattern symb-list symb-list ⇒ bool ⇒ bool where
  trigger - P = P

62.3 Higher-order encoding

Application is made explicit for constants occurring with varying numbers of arguments. This is achieved by the introduction of the following constant.

definition fun-app :: 'a ⇒ 'a where fun-app f = f

Some solvers support a theory of arrays which can be used to encode higher-order functions. The following set of lemmas specifies the properties of such (extensional) arrays.

lemmas array-rules = ext fun-upd-apply fun-upd-same fun-upd-other fun-upd-upd fun-app-def

62.4 Normalization

lemma case-bool-if[abs-def]: case-bool x y P = (if P then x else y)
  by simp

lemmas Ex1-def-raw = Ex1-def[abs-def]
lemmas Ball-def-raw = Ball-def[abs-def]
lemmas Bex-def-raw = Bex-def[abs-def]
lemmas abs-if-raw = abs-if[abs-def]
lemmas min-def-raw = min-def[abs-def]
lemmas max-def-raw = max-def[abs-def]

lemma nat-zero-as-int:
  0 = nat 0
  by simp

lemma nat-one-as-int:
  1 = nat 1
  by simp

lemma nat-numeral-as-int: numeral = (λi. nat (numeral i)) by simp
lemma nat-less-as-int: 
\( < \) = (\( \lambda ab. \int a < \int b \) ) by simp

lemma nat-leq-as-int:
\( \leq \) = (\( \lambda ab. \int a \leq \int b \) ) by simp

lemma Suc-as-int:
\( \text{Suc} \) = (\( \lambda a. \text{nat}(\int a + 1) \) ) by (rule ext) simp

lemma nat-plus-as-int:
\( + \) = (\( \lambda ab. \text{nat}(\int a + \int b) \) ) by (rule ext)+ simp

lemma nat-minus-as-int:
\( - \) = (\( \lambda ab. \text{nat}(\int a - \int b) \) ) by (rule ext)+ simp

lemma nat-times-as-int:
\( \ast \) = (\( \lambda ab. \text{nat}(\int a \ast \int b) \) ) by simp add: nat-mult-distrib

lemma nat-div-as-int:
\( \text{div} \) = (\( \lambda ab. \text{nat}(\int a \div \int b) \) ) by simp add: nat-div-distrib

lemma nat-mod-as-int:
\( \text{mod} \) = (\( \lambda ab. \text{nat}(\int a \mod \int b) \) ) by simp add: nat-mod-distrib

lemma int-Suc:
\( \int \text{Suc} \) = \( \int n \) + 1 by simp

lemma int-plus:
\( \int (n + m) \) = \( \int n \) + \( \int m \) by (rule of-nat-add)

lemma int-minus:
\( \int (n - m) \) = \( \int (\text{nat}(\int n - \int m)) \) by auto

lemma int-int-comparison:
fixes a b :: nat
shows (a = b) = (\( \int a = \int b \) )
and (a < b) = (\( \int a < \int b \) )
and (a ≤ b) = (\( \int a \leq \int b \) )
by simp-all

lemma int-ops:
fixes a b :: nat
shows \( \int 0 = 0 \) 
and \( \int 1 = 1 \)
and \( \int (\text{numeral} n) = \text{numeral} n \)
and \( \int (\text{Suc} a) = \int a + 1 \)
and \( \int (a + b) = \int a + \int b \)
and \( \int (a - b) = (\text{if } \int a < \int b \text{ then } 0 \text{ else } \int a - \int b) \)
and \( \int (a \ast b) = \int a \ast \int b \)
and \( \int (a \text{ div} b) = \int a \text{ div} \int b \)
and \( \int (a \text{ mod} b) = \int a \text{ mod} \int b \)
by (auto intro: zdiv-int zmod-int)

lemma int-if:
fixes a b :: nat
shows \( \int (\text{if } P \text{ then } a \text{ else } b) = (\text{if } P \text{ then } \int a \text{ else } \int b) \)
by simp

62.5 Integer division and modulo for Z3

The following Z3-inspired definitions are overspecified for the case where \( l = 0 \). This Schönheitsfehler is corrected in the \text{div-as-z3div} and \text{mod-as-z3mod} theorems.

definition z3div :: int ⇒ int ⇒ int where
\( z3div k l = (\text{if } l \geq 0 \text{ then } k \text{ div } l \text{ else } - (k \text{ div } -l)) \)

definition z3mod :: int ⇒ int ⇒ int where
\( z3mod k l = k \text{ mod } (\text{if } l \geq 0 \text{ then } l \text{ else } -l) \)
lemma \textit{div-as-z3div}:
\[
\forall k\ l.\ k\ \text{div}\ l \equiv (\text{if } l \leq 0\ \text{then } 0\ \text{else if } l > 0\ \text{then } z3\text{div } k\ l\ \text{else } z3\text{div } (-k)\ (-l))
\]
by (simp add: z3div-def)

lemma \textit{mod-as-z3mod}:
\[
\forall k\ l.\ k\ \text{mod}\ l \equiv (\text{if } l \leq 0\ \text{then } k\ \text{else if } l > 0\ \text{then } z3\text{mod } k\ l\ \text{else } -z3\text{mod } (-k)\ (-l))
\]
by (simp add: z3mod-def)

62.6 Extra theorems for veriT reconstruction

lemma \textit{verit-sko-forall}:
\[
\langle \forall x.\ P\ x \iff P\ (\text{SOME } x.\ \neg P\ x)\rangle
\]
using someI[of \(\lambda x.\ \neg P\ x\)]
by auto

lemma \textit{verit-sko-forall}':
\[
\langle P\ (\text{SOME } x.\ \neg P\ x) = A\Rightarrow (\forall x.\ P\ x) = A\rangle
\]
by (subst verit-sko-forall)

lemma \textit{verit-sko-forall-indirect}:
\[
\langle x = (\text{SOME } x.\ \neg P\ x) \Rightarrow (\forall x.\ P\ x) \iff P\ x\rangle
\]
using someI[of \(\lambda x.\ \neg P\ x\)]
by auto

lemma \textit{verit-sko-ex}:
\[
\langle \exists x.\ P\ x \iff P\ (\text{SOME } x.\ P\ x)\rangle
\]
using someI[of \(\lambda x.\ P\ x\)]
by auto

lemma \textit{verit-sko-ex}':
\[
\langle P\ (\text{SOME } x.\ P\ x) = A\Rightarrow (\exists x.\ P\ x) = A\rangle
\]
by (subst verit-sko-ex)

lemma \textit{verit-sko-ex-indirect}:
\[
\langle x = (\text{SOME } x.\ P\ x) \Rightarrow (\exists x.\ P\ x) \iff P\ x\rangle
\]
using someI[of \(\lambda x.\ P\ x\)]
by auto

lemma \textit{verit-Pure-trans}:
\[
\langle P \equiv Q \iff Q \equiv P\rangle
\]
by auto

lemma \textit{verit-if-cong}:
assumes \(b \equiv c\)
and \(c \iff x \equiv w\)
and \(\neg c \iff y \equiv v\)
shows \(\langle \text{if } b\ \text{then } x\ \text{else } y\rangle \equiv (\text{if } c\ \text{then } u\ \text{else } v)\)
using assms if-cong[of \(b\ c\ u\)] by auto

lemma \textit{verit-if-weak-cong}:
\(b \equiv c \Rightarrow (\text{if } b\ \text{then } x\ \text{else } y) \equiv (\text{if } c\ \text{then } x\ \text{else } y)\)
by auto
lemma verit-ite-intro-simp:
\[
\langle (\text{if } c \text{ then } (a :: 'a) = (\text{if } c \text{ then } P \text{ else } Q') \text{ else } Q) = (\text{if } c \text{ then } a = P \text{ else } Q) \rangle
\]
\[
\langle (\text{if } c \text{ then } R \text{ else } b = (\text{if } c \text{ then } R' \text{ else } Q') = (\text{if } c \text{ then } a' = a' \text{ else } b' = b') \rangle
\]
by (auto split: if-splits)

lemma verit-or-neg:
\[
\langle (A = \Rightarrow B) = \Rightarrow B \lor \neg A \rangle
\]
\[
\langle (\neg A = \Rightarrow B) = \Rightarrow B \lor A \rangle
\]
by auto

lemma verit-subst-bool:
\[
\langle P = \Rightarrow f \text{ True} = \Rightarrow f P \rangle
\]
by auto

lemma verit-and-pos:
\[
\langle a \Rightarrow \neg b \lor A \Rightarrow \neg(a \land b) \lor A \rangle
\]
\[
\langle a \Rightarrow A \Rightarrow \neg a \lor A \rangle
\]
\[
\langle \neg a \Rightarrow A \Rightarrow a \lor A \rangle
\]
by blast +

lemma verit-la-generic:
\[
(a::int) \leq x \lor a = x \lor a \geq x
\]
by linarith

lemma verit-tmp-bfun-elim:
\[
\langle (\text{if } b \text{ then } P \text{ True else } P \text{ False}) = P b \rangle
\]
by (cases b)

lemma verit-eq-true-simplify:
\[
\langle P = \text{True} \equiv P \rangle
\]
by auto

lemma verit-and-neg:
\[
\langle B \lor B' = \Rightarrow (A \land B) \lor \neg A \lor B' \rangle
\]
\[
\langle B \lor B' = \Rightarrow (\neg A \land B) \lor A \lor B' \rangle
\]
by auto

lemma verit-forall-inst:
\[
\langle A \leftarrow B = \Rightarrow \neg A \lor B \rangle
\]
\[
\langle \neg A \leftarrow B = \Rightarrow A \lor B \rangle
\]
\[
\langle A \leftarrow B = \Rightarrow \neg B \lor A \rangle
\]
\[
\langle A \leftarrow \neg B = \Rightarrow B \lor A \rangle
\]
\[
\langle A \leftarrow B = \Rightarrow \neg A \lor B \rangle
\]
\[
\langle \neg A \rightarrow B = \Rightarrow A \lor B \rangle
\]
by blast +

lemma verit-eq-transitive:
\[
\langle A = B = \Rightarrow B = C = \Rightarrow A = C \rangle
\]
\[ (A = B \implies C = B \implies A = C) \]
\[ (B = A \implies B = C \implies A = C) \]
\[ (B = A \implies C = B \implies A = C) \]
by auto

### 62.7 Setup

- ML-file ⟨Tools/SMT/smt-util.ML⟩
- ML-file ⟨Tools/SMT/smt-failure.ML⟩
- ML-file ⟨Tools/SMT/smt-config.ML⟩
- ML-file ⟨Tools/SMT/smt-builtin.ML⟩
- ML-file ⟨Tools/SMT/smt-datatypes.ML⟩
- ML-file ⟨Tools/SMT/smt-normalize.ML⟩
- ML-file ⟨Tools/SMT/smt-translate.ML⟩
- ML-file ⟨Tools/SMT/smtlib.ML⟩
- ML-file ⟨Tools/SMT/smtlib-interface.ML⟩
- ML-file ⟨Tools/SMT/smtlib-proof.ML⟩
- ML-file ⟨Tools/SMT/z3-proof.ML⟩
- ML-file ⟨Tools/SMT/z3-isar.ML⟩
- ML-file ⟨Tools/SMT/smt-solver.ML⟩
- ML-file ⟨Tools/SMT/cvc4-interface.ML⟩
- ML-file ⟨Tools/SMT/cvc4-proof-parse.ML⟩
- ML-file ⟨Tools/SMT/verit-proof.ML⟩
- ML-file ⟨Tools/SMT/verit-isar.ML⟩
- ML-file ⟨Tools/SMT/verit-proof-parse.ML⟩
- ML-file ⟨Tools/SMT/conj-disj-perm.ML⟩
- ML-file ⟨Tools/SMT/smt-replay-methods.ML⟩
- ML-file ⟨Tools/SMT/smt-replay.ML⟩
- ML-file ⟨Tools/SMT/z3-interface.ML⟩
- ML-file ⟨Tools/SMT/z3-replay-rules.ML⟩
- ML-file ⟨Tools/SMT/z3-replay-methods.ML⟩
- ML-file ⟨Tools/SMT/z3-replay.ML⟩
- ML-file ⟨Tools/SMT/verit-replay-methods.ML⟩
- ML-file ⟨Tools/SMT/verit-replay.ML⟩
- ML-file ⟨Tools/SMT/smt-systems.ML⟩

method-setup smt = \( \langle \text{Scan.} \text{optional} \text{Attrib.thms} \parallel \rangle \) >>
\( (\text{fn thms => fn ctxt =>} \) METHOD (\text{fn facts => HEADGOAL \{SMT-Solver.smt-tac ctxt (thms @ facts)\}}))
\( \rangle \) apply an SMT solver to the current goal

### 62.8 Configuration

The current configuration can be printed by the command \textit{smt-status}, which shows the values of most options.
62.9 General configuration options

The option `smt-solver` can be used to change the target SMT solver. The possible values can be obtained from the `smt-status` command.

```
declare [[smt-solver = z3]]
```

Since SMT solvers are potentially nonterminating, there is a timeout (given in seconds) to restrict their runtime.

```
declare [[smt-timeout = 20]]
```

SMT solvers apply randomized heuristics. In case a problem is not solvable by an SMT solver, changing the following option might help.

```
declare [[smt-random-seed = 1]]
```

In general, the binding to SMT solvers runs as an oracle, i.e., the SMT solvers are fully trusted without additional checks. The following option can cause the SMT solver to run in proof-producing mode, giving a checkable certificate. This is currently only implemented for Z3.

```
declare [[smt-oracle = false]]
```

Each SMT solver provides several commandline options to tweak its behaviour. They can be passed to the solver by setting the following options.

```
declare [[cvc3-options = ]]
declare [[cvc4-options = --full-saturate-quant --inst-when=full-last-call --inst-no-entail --term-db-mode=relevant --multi-trigger-linear]]
declare [[verit-options = --index-fresh-sorts]]
declare [[z3-options = ]]
```

The SMT method provides an inference mechanism to detect simple triggers in quantified formulas, which might increase the number of problems solvable by SMT solvers (note: triggers guide quantifier instantiations in the SMT solver). To turn it on, set the following option.

```
declare [[smt-infer-triggers = false]]
```

Enable the following option to use built-in support for datatypes, codatatypes, and records in CVC4. Currently, this is implemented only in oracle mode.

```
declare [[cvc4-extensions = false]]
```

Enable the following option to use built-in support for div/mod, datatypes, and records in Z3. Currently, this is implemented only in oracle mode.

```
declare [[z3-extensions = false]]
```

62.10 Certificates

By setting the option `smt-certificates` to the name of a file, all following applications of an SMT solver are cached in that file. Any further application
of the same SMT solver (using the very same configuration) re-uses the cached certificate instead of invoking the solver. An empty string disables caching certificates.

The filename should be given as an explicit path. It is good practice to use the name of the current theory (with ending ".certs" instead of ".thy") as the certificates file. Certificate files should be used at most once in a certain theory context, to avoid race conditions with other concurrent accesses.

\begin{verbatim}
declare [smt-certificates = ]
\end{verbatim}

The option \texttt{smt-read-only-certificates} controls whether only stored certificates should be used or invocation of an SMT solver is allowed. When set to \texttt{true}, no SMT solver will ever be invoked and only the existing certificates found in the configured cache are used; when set to \texttt{false} and there is no cached certificate for some proposition, then the configured SMT solver is invoked.

\begin{verbatim}
declare [smt-read-only-certificates = false]
\end{verbatim}

\subsection{Tracing}

The SMT method, when applied, traces important information. To make it entirely silent, set the following option to \texttt{false}.

\begin{verbatim}
declare [smt-verbose = true]
\end{verbatim}

For tracing the generated problem file given to the SMT solver as well as the returned result of the solver, the option \texttt{smt-trace} should be set to \texttt{true}.

\begin{verbatim}
declare [smt-trace = false]
\end{verbatim}

\subsection{Schematic rules for Z3 proof reconstruction}

Several \texttt{prof} rules of Z3 are not very well documented. There are two lemma groups which can turn failing Z3 proof reconstruction attempts into succeeding ones: the facts in \texttt{z3-rule} are tried prior to any implemented reconstruction procedure for all uncertain Z3 proof rules; the facts in \texttt{z3-simp} are only fed to invocations of the simplifier when reconstructing theory-specific proof steps.

\begin{verbatim}
lemmas [z3-rule] =
  refl eq-commute conj-commute disj-commute simp-thms nnf-simps
  ring-distrib field-simps times-divide-eq-right times-divide-eq-left
  if-True if-False not-not
  NO-MATCH-def
\end{verbatim}

\begin{verbatim}
lemma [z3-rule]:
  \((P \land Q) = (\neg (\neg P \lor \neg Q))\)
  \((P \land Q) = (\neg (\neg Q \lor \neg P))\)
\end{verbatim}
\[ \neg P \land Q = (\neg (P \lor \neg Q)) \]
\[ \neg P \land Q = (\neg (\neg Q \lor P)) \]
\[ P \land \neg Q = (\neg (P \lor Q)) \]
\[ (P \land \neg Q) = (\neg (Q \lor P)) \]
\[ \neg P \land \neg Q = (\neg (P \lor Q)) \]
\[ \neg P \land \neg Q = (\neg (Q \lor P)) \]
\[ \text{by auto} \]

\text{lemma \[z3\]-rule]:
\[ (P \rightarrow Q) = (Q \lor \neg P) \]
\[ (\neg P \rightarrow Q) = (P \lor Q) \]
\[ (\neg P \rightarrow Q) = (Q \lor P) \]
\[ (\True \rightarrow P) = P \]
\[ (P \rightarrow \True) = \True \]
\[ (\False \rightarrow P) = \True \]
\[ (P \rightarrow P) = \True \]
\[ (\neg (A \leftrightarrow \neg B)) \leftrightarrow (A \leftrightarrow B) \]
\[ \text{by auto} \]

\text{lemma \[z3\]-rule}:
\[ ((P = Q) \rightarrow R) = (R \lor (Q = (\neg P))) \]
\[ \text{by auto} \]

\text{lemma \[z3\]-rule}:
\[ (\neg \True) = \False \]
\[ (\neg \False) = \True \]
\[ (x = x) = \True \]
\[ (P = \True) = P \]
\[ (\True = P) = P \]
\[ (P = \False) = (\neg P) \]
\[ (\False = P) = (\neg P) \]
\[ (\neg P) = P = \False \]
\[ (P = (\neg P)) = \False \]
\[ ((\neg P) = (\neg Q)) = (P = Q) \]
\[ \neg (P = (\neg Q)) = (P = Q) \]
\[ \neg ((\neg P) = Q) = (P = Q) \]
\[ (P \neq Q) = (Q = (\neg P)) \]
\[ (P = Q) = ((\neg P \lor Q) \land (P \lor Q)) \]
\[ (P \neq Q) = ((\neg P \lor \neg Q) \land (P \lor Q)) \]
\[ \text{by auto} \]

\text{lemma \[z3\]-rule}:
\[ (\text{if } P \text{ then } P \text{ else } \neg P) = \True \]
\[ (\text{if } \neg P \text{ then } P \text{ else } P) = \True \]
\[ (\text{if } P \text{ then } \True \text{ else False}) = P \]
\[ (\text{if } P \text{ then } \False \text{ else } \True) = (\neg P) \]
\[ (\text{if } P \text{ then } Q \text{ else } \True) = ((\neg P) \lor Q) \]
\[ (\text{if } P \text{ then } Q \text{ else } \True) = (Q \lor (\neg P)) \]
\[ (\text{if } P \text{ then } Q \text{ else } \neg Q) = (P = Q) \]
THEORY “SMT”

(if \( P \) then \( Q \) else \( \neg Q \)) = (\( Q = P \))
(if \( P \) then \( \neg Q \) else \( Q \)) = (\( P = (\neg Q) \))
(if \( P \) then \( \neg Q \) else \( Q \)) = ((\neg Q) = P)
(if \( \neg P \) then \( x \) else \( y \)) = (if \( P \) then \( y \) else \( x \))
(if \( P \) then \( (\neg Q) \) else \( P \)) = (\( P \land (\neg Q) \) then \( y \) else \( x \))
(if \( P \) then \( (\neg Q) \) else \( P \)) = (\( (\neg Q) \land P \) then \( y \) else \( x \))
(if \( P \) then \( (\neg Q) \) else \( P \)) = (\( (\neg Q) \land P \) then \( y \) else \( x \))
(if \( P \) then \( (\neg Q) \) else \( P \)) = (\( (\neg Q) \land P \) then \( y \) else \( x \))
(if \( P \) then \( (\neg Q) \) else \( P \)) = (\( (\neg Q) \land P \) then \( y \) else \( x \))

by auto

lemma \([z3\text{-rule}]:\)
0 + (\( x::\text{int} \)) = \( x \)
\( x + 0 = x \)
\( x + x = 2 \ast x \)
\( 0 \ast x = 0 \)
\( 1 \ast x = x \)
\( x + y = y + x \)
by (auto simp add: mult-2)

lemma \([z3\text{-rule}]:\)
\( P = Q \lor P \lor Q \)
\( P = Q \lor \neg P \lor \neg Q \)
(\( \neg P \)) = \( Q \lor \neg P \lor \neg Q \)
(\( \neg P \)) = \( Q \lor \neg P \lor \neg Q \)
\( P = (\neg Q) \lor \neg P \lor \neg Q \)
\( \neg P \neq Q \lor P \lor \neg Q \)
\( P \neq Q \lor \neg P \lor Q \)
(\( \neg P \)) = \( Q \lor \neg P \lor \neg Q \)
\( P \lor \neg Q \lor \neg Q \)
\( P \lor \neg Q \lor (\neg P) \neq Q \)
\( P \lor \neg Q \lor (\neg P) \neq Q \)
\( P \lor \neg Q \lor (\neg P) \neq Q \)
\( P \lor \neg Q \lor (\neg P) \neq Q \)
\( P \lor \neg Q \lor (\neg P) \neq Q \)
\( P \lor \neg Q \lor (\neg P) \neq Q \)
\( \neg P \lor \neg Q \lor (\neg P) \neq Q \)
\( \neg P \lor \neg Q \lor (\neg P) \neq Q \)
\( \neg P \lor \neg Q \lor (\neg P) \neq Q \)
\( \neg P \lor \neg Q \lor (\neg P) \neq Q \)
\( \neg P \lor \neg Q \lor (\neg P) \neq Q \)
\( \neg P \lor \neg Q \lor (\neg P) \neq Q \)
\( (P \text{ then } x \text{ else } y) = (P \lor Q \lor \neg P \lor \neg Q) \)
\( P \lor (\text{if } P \text{ then } x \text{ else } y) = y \)
\( \neg P \lor x = (\text{if } P \text{ then } x \text{ else } y) \)
\( \neg P \lor (\text{if } P \text{ then } x \text{ else } y) = x \)
\( P \lor R \lor (\text{if } P \text{ then } Q \text{ else } R) \)
\( \neg P \lor Q \lor (\text{if } P \text{ then } Q \text{ else } R) \)
\( \neg P \lor Q \lor (\text{if } P \text{ then } Q \text{ else } R) \)
\( \neg P \lor (\text{if } P \text{ then } Q \text{ else } R) \lor \neg P \lor Q \)
\( \neg P \lor (\text{if } P \text{ then } Q \text{ else } R) \lor P \lor R \)
\( \neg P \lor (\text{if } P \text{ then } Q \text{ else } R) \lor \neg Q \lor \neg Q \)
THEORY "Sledgehammer"

(if P then Q else R) ∨ P ∨ ¬ R
(if P then ¬ Q else R) ∨ ¬ P ∨ Q
(if P then Q else ¬ R) ∨ P ∨ R
by auto

hide-type (open) symb-list pattern
hide-const (open) Symb-Nil Symb-Cons trigger pat nopat fun-app z3div z3mod

end

63 Sledgehammer: Isabelle–ATP Linkup

theory Sledgehammer
imports Presburger SMT
keywords
  sledgehammer :: diag and
  sledgehammer-params :: thy-decl
begin

lemma size-ne-size-imp-ne: size x ≠ size y ⇒ x ≠ y
  by (erule contrapos-nn) (rule arg-cong)

ML-file ⟨Tools/Sledgehammer/async-manager-legacy.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-util.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-fact.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-proof-methods.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-isar-annotate.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-isar-proof.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-isar-preplay.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-isar-compress.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-isar-minimize.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-isar.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-prover.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-prover-atp.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-prover-smt.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-prover-minimize.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-mepo.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-mash.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer.ML⟩
ML-file ⟨Tools/Sledgehammer/sledgehammer-commands.ML⟩

end

64 Numeric types for code generation onto target language numerals only

theory Code-Numeral
imports Divides Lifting
begin

64.1 Type of target language integers

typedef integer = UNIV :: int set
  morphisms int-of-integer integer-of-int ..

setup-lifting type-definition-integer

lemma integer-eq-iff:
  k = l ←→ int-of-integer k = int-of-integer l
  by transfer rule

lemma integer-eqI:
  int-of-integer k = int-of-integer l =⇒ k = l
  using integer-eq-iff [of k l] by simp

lemma int-of-integer-int-of-int [simp]:
  int-of-integer (integer-of-int k) = k
  by transfer rule

lemma integer-of-int-int-of-integer [simp]:
  integer-of-int (int-of-integer k) = k
  by transfer rule

instantiation integer :: ring-1
begin

lift-definition zero-integer :: integer
  is 0 :: int
  .

declare zero-integer.rep-eq [simp]

lift-definition one-integer :: integer
  is 1 :: int
  .

declare one-integer.rep-eq [simp]

lift-definition plus-integer :: integer ⇒ integer ⇒ integer
  is plus :: int ⇒ int ⇒ int
  .

declare plus-integer.rep-eq [simp]

lift-definition uminus-integer :: integer ⇒ integer
  is uminus :: int ⇒ int
  .
declare uminus-integer.rep-eq [simp]

lift-definition minus-integer :: integer ⇒ integer ⇒ integer
  is minus :: int ⇒ int ⇒ int
  .

declare minus-integer.rep-eq [simp]

lift-definition times-integer :: integer ⇒ integer ⇒ integer
  is times :: int ⇒ int ⇒ int
  .

declare times-integer.rep-eq [simp]

instance proof
qed (transfer, simp add: algebra-simps)+

end

instance integer :: Rings.dvd ..

countext
  includes lifting-syntax
  notes transfer-rule-numeral [transfer-rule]
begin

lemma [transfer-rule]:
  (pcr-integer ===> pcr-integer ===> (↔)) (dvd) (dvd)
  by (unfold dvd-def) transfer-prover

lemma [transfer-rule]:
  ((↔) ===> pcr-integer) of-bool of-bool
  by (unfold of-bool-def) transfer-prover

lemma [transfer-rule]:
  ((=) ===> pcr-integer) int of-nat
  by (rule transfer-rule-of-nat) transfer-prover+

lemma [transfer-rule]:
  ((=) ===> pcr-integer) (λk. k) of-int
  proof –
    have ((=) ===> pcr-integer) of-int of-int
      by (rule transfer-rule-of-int) transfer-prover+
    then show ?thesis by (simp add: id-def)
    qed

lemma [transfer-rule]:
  ((=) ===> pcr-integer) numeral numeral
THEORY "Code-Numeral"

by transfer-prover

lemma [transfer-rule]:
  \((=) \Longrightarrow (\_)) \Longrightarrow pcr-integer\) Num.sub Num.sub
by (unfold Num.sub-def) transfer-prover

lemma [transfer-rule]:
  \((pcr-integer \Longrightarrow (=)) \Longrightarrow pcr-integer\) (\^\_\_\_) (\^\_\_\_)
by (unfold power-def) transfer-prover

end

lemma int-of-integer-of-nat [simp]:
  int-of-integer (of-nat n) = of-nat n
by transfer rule

lift-definition integer-of-nat :: nat \Rightarrow integer
  is of-nat :: nat \Rightarrow int

.

lemma integer-of-nat-eq-of-nat [code]:
  integer-of-nat = of-nat
by transfer rule

lemma int-of-integer-integer-of-nat [simp]:
  int-of-integer (integer-of-nat n) = of-nat n
by transfer rule

lift-definition nat-of-integer :: integer \Rightarrow nat
  is Int.nat
.

lemma nat-of-integer-of-nat [simp]:
  nat-of-integer (of-nat n) = n
by transfer simp

lemma int-of-integer-of-int [simp]:
  int-of-integer (of-int k) = k
by transfer simp

lemma nat-of-integer-integer-of-nat [simp]:
  nat-of-integer (integer-of-nat n) = n
by transfer simp

lemma integer-of-int-eq-of-int [simp, code-abbrev]:
  integer-of-int = of-int
by transfer (simp add: fun-eq-iff)

lemma of-int-integer-of [simp]:
of-int (int-of-integer k) = (k :: integer)
by transfer rule

lemma int-of-integer-numeral [simp]:
  int-of-integer (numeral k) = numeral k
by transfer rule

lemma int-of-integer-sub [simp]:
  int-of-integer (Num.sub k l) = Num.sub k l
by transfer rule

definition integer-of-num :: num ⇒ integer
  where [simp]:
  integer-of-num = numeral

lemma integer-of-num [code]:
  integer-of-num Num.One = 1
  integer-of-num (Num.Bit0 n) = (let k = integer-of-num n in k + k)
  integer-of-num (Num.Bit1 n) = (let k = integer-of-num n in k + k + 1)
by (simp-all only: integer-of-num-def numeral.simps Let-def)

lemma integer-of-num-triv:
  integer-of-num Num.One = 1
  integer-of-num (Num.Bit0 Num.One) = 2
by simp-all

instantiation integer :: {linordered-idom, equal}
begin

lift-definition abs-integer :: integer ⇒ integer
  is abs :: int ⇒ int
  .

declare abs-integer.rep-eq [simp]

lift-definition sgn-integer :: integer ⇒ integer
  is sgn :: int ⇒ int
  .

declare sgn-integer.rep-eq [simp]

lift-definition less-eq-integer :: integer ⇒ integer ⇒ bool
  is less-eq :: int ⇒ int ⇒ bool
  .

lemma integer-less-eq-iff:
  k ≤ l ↔ int-of-integer k ≤ int-of-integer l
by (fact less-eq-integer.rep-eq)

lift-definition less-integer :: integer ⇒ integer ⇒ bool
is less :: int ⇒ int ⇒ bool .

lemma integer-less-iff:
  k < l ⟷ int-of-integer k < int-of-integer l
by (fact less-integer.rep-eq)

lift-definition equal-integer :: integer ⇒ integer ⇒ bool
is HOL.equal :: int ⇒ int ⇒ bool .

instance
  by standard (transfer, simp add: algebra-simps equal-less-not-le [symmetric]
mult-strict-right-mono linear)+

end

context
  includes lifting-syntax
begin

lemma [transfer-rule]:
  ⟨(per-integer ===> per-integer ===> per-integer) min min⟩
by (unfold min-def) transfer-prover

lemma [transfer-rule]:
  ⟨(per-integer ===> per-integer ===> per-integer) max max⟩
by (unfold max-def) transfer-prover

end

lemma int-of-integer-min [simp]:
  int-of-integer (min k l) = min (int-of-integer k) (int-of-integer l)
by transfer rule

lemma int-of-integer-max [simp]:
  int-of-integer (max k l) = max (int-of-integer k) (int-of-integer l)
by transfer rule

lemma nat-of-integer-non-positive [simp]:
  k ≤ 0 ⇒ nat-of-integer k = 0
by transfer simp

lemma of-nat-of-integer [simp]:
  of-nat (nat-of-integer k) = max 0 k
by transfer auto

instantiation integer :: unique-euclidean-ring
begin
lift-definition \texttt{divide-integer :: integer \Rightarrow integer} \\
\texttt{is divide :: int \Rightarrow int \Rightarrow int} .

\texttt{declare divide-integer.rep-eq [simp]}

lift-definition \texttt{modulo-integer :: integer \Rightarrow integer} \\
\texttt{is modulo :: int \Rightarrow int \Rightarrow int} .

\texttt{declare modulo-integer.rep-eq [simp]}

lift-definition \texttt{euclidean-size-integer :: integer \Rightarrow nat} \\
\texttt{is euclidean-size :: int \Rightarrow nat} .

\texttt{declare euclidean-size-integer.rep-eq [simp]}

lift-definition \texttt{division-segment-integer :: integer \Rightarrow integer} \\
\texttt{is division-segment :: int \Rightarrow int} .

\texttt{declare division-segment-integer.rep-eq [simp]}

instance \texttt{by (standard; transfer)} \\
\texttt{(use mult-le-mono2 [of 1] in (auto simp add: sgn-mult-abs abs-mult sgn-mult abs-mod-less sgn-mod nat-mult-distrib}} \\
\texttt{division-segment-mult division-segment-mod intro: div-eqI))}

\texttt{end}

\texttt{lemma [code]:}
\texttt{euclidean-size = nat-of-integer \circ abs}
\texttt{by (simp add: fun-eq-iff nat-of-integer.rep-eq)}

\texttt{lemma [code]:}
\texttt{division-segment \ (k :: integer) = (if k \geq 0 then 1 else \ - 1)}
\texttt{by transfer (simp add: division-segment-int-def)}

\texttt{instance integer :: unique-euclidean-ring-with-nat}
\texttt{by (standard; transfer) (simp-all add: of-nat-div division-segment-int-def)}

\texttt{instantiation integer :: semiring-bit-shifts}
\texttt{begin}

lift-definition \texttt{push-bit-integer :: (nat \Rightarrow integer \Rightarrow integer)} \\
\texttt{is (push-bit)} .
lift-definition drop-bit-integer :: (nat ⇒ integer ⇒ integer)
is (drop-bit) .

instance by (standard; transfer)
(fact bit-eq-rec bits-induct push-bit-eq-mult drop-bit-eq-div
bits-div-0 bits-div-by-1 bits-mod-div-trivial even-succ-div-2
exp-div-exp-eq div-exp-eq mod-exp-eq mult-exp-mod-exp-eq
div-exp-mod-exp-eq even-mask-div-iff even-mult-exp-div-exp-iff)+

end

context
  includes lifting-syntax
begin

lemma [transfer-rule]:
⟨ (pcr-integer ===> (=) ===> (↔)) bit bit ⟩
by (unfold bit-def) transfer-prover

lemma [transfer-rule]:
⟨ (pcr-integer ===> (=) ===> (↔)) take-bit take-bit ⟩
by (unfold take-bit-eq-mod) transfer-prover

end

instance integer :: unique-euclidean-semiring-with-bit-shifts ..

lemma [code]:
⟨ push-bit n k = k * 2 ^ n ⟩
⟨ drop-bit n k = k div 2 ^ n ⟩ for k :: integer
by (fact push-bit-eq-mult drop-bit-eq-div)+

instantiation integer :: unique-euclidean-semiring-numeral
begin

definition divmod-integer :: num ⇒ num ⇒ integer × integer
where
  divmod-integer'·def: divmod-integer m n = (numeral m div numeral n, numeral m mod numeral n)

definition divmod-step-integer :: num ⇒ integer × integer ⇒ integer × integer
where
  divmod-step-integer l qr = (let (q, r) = qr
    in if r ≥ numeral l then (2 * q + l, r - numeral l)
    else (2 * q, r))

instance proof
  show divmod m n = (numeral m div numeral n :: integer, numeral m mod numeral n)
n)  
for m n by (fact divmod-integer'-def)

show divmod-step l qr = (let (q, r) = qr
  in if r ≥ numeral l then (2 * q + 1, r − numeral l)
  else (2 * q, r)) for l and qr :: integer × integer
by (fact divmod-step-integer-def)

qed (transfer, fact le-add-diff-inverse2
unique-euclidean-semiring-numeral-class.div-less
unique-euclidean-semiring-numeral-class.mod-less
unique-euclidean-semiring-numeral-class.div-positive
unique-euclidean-semiring-numeral-class.mod-less-eq-dividend
unique-euclidean-semiring-numeral-class.pos-mod-bound
unique-euclidean-semiring-numeral-class.pos-mod-sign
unique-euclidean-semiring-numeral-class.mod-mult2-eq
unique-euclidean-semiring-numeral-class.div-mult2-eq
unique-euclidean-semiring-numeral-class.discrete)+

end

declare divmod-algorithm-code [where ?a = integer,
  folded integer-of-num-def, unfolded integer-of-num-triv,
  code]

lemma integer-of-nat-0: integer-of-nat 0 = 0
by transfer simp

lemma integer-of-nat-1: integer-of-nat 1 = 1
by transfer simp

lemma integer-of-nat-numeral:
  integer-of-nat (numeral n) = numeral n
by transfer simp

64.2 Code theorems for target language integers

Constructors

definition Pos :: num ⇒ integer
where
  [simp, code-post]: Pos = numeral

context
  includes lifting-syntax
begin

lemma [transfer-rule]:
  (((=) ===> per-integer) numeral Pos;
by simp transfer-prover
end

lemma Pos-fold [code-unfold]:
  numeral Num.One = Pos Num.One
  numeral (Num.Bit0 k) = Pos (Num.Bit0 k)
  numeral (Num.Bit1 k) = Pos (Num.Bit1 k)
by simp-all

definition Neg :: num ⇒ integer
where
  [simp, code-abbrev]: Neg n = − Pos n

code-datatype 0::integer Pos Neg

A further pair of constructors for generated computations
context
begin

qualified definition positive :: num ⇒ integer
where [simp]: positive = numeral

qualified definition negative :: num ⇒ integer
where [simp]: negative = uminus ◦ numeral

lemma [code-computation-unfold]:
  numeral = positive
  Pos = positive
  Neg = negative
by (simp-all add: fun-eq-iff)
end

Auxiliary operations

lift-definition dup :: integer ⇒ integer
  is λk::int. k + k
.

lemma dup-code [code]:
dup 0 = 0
THEORY "Code-Numeral"

\begin{align*}
dup (\text{Pos } n) &= \text{Pos } (\text{Num.Bit0 } n) \\
dup (\text{Neg } n) &= \text{Neg } (\text{Num.Bit0 } n) \\
\text{by} & \ (\text{transfer}, \ \text{simp only: numeral-Bit0 minus-add-distrib}) \\
\end{align*}

\textbf{lift-definition} \quad \text{sub} :: \text{num} \Rightarrow \text{num} \Rightarrow \text{integer} \\
\quad \text{is } \lambda m \ n. \ \text{numeral } m - \text{numeral } n :: \text{int} \\

\textbf{lemma} \quad \text{sub-code [code]}: \\
\quad \text{sub } \text{Num.One Num.One } = 0 \\
\quad \text{sub } (\text{Num.Bit0 } m) \ \text{Num.One } = \text{Pos } (\text{Num.BitM } m) \\
\quad \text{sub } (\text{Num.Bit1 } m) \ \text{Num.One } = \text{Pos } (\text{Num.Bit0 } m) \\
\quad \text{sub } \text{Num.One } (\text{Num.Bit0 } n) = \text{Neg } (\text{Num.BitM } n) \\
\quad \text{sub } \text{Num.One } (\text{Num.Bit1 } n) = \text{Neg } (\text{Num.Bit0 } n) \\
\quad \text{sub } (\text{Num.Bit0 } m) (\text{Num.Bit0 } n) = \text{dup } (\text{sub } m \ n) \\
\quad \text{sub } (\text{Num.Bit1 } m) (\text{Num.Bit1 } n) = \text{dup } (\text{sub } m \ n) \\
\quad \text{sub } (\text{Num.Bit1 } m) (\text{Num.Bit0 } n) = \text{dup } (\text{sub } m \ n) + 1 \\
\quad \text{sub } (\text{Num.Bit0 } m) (\text{Num.Bit1 } n) = \text{dup } (\text{sub } m \ n) - 1 \\
\quad \text{by} & \ (\text{transfer}, \ \text{simp add: dbl-def dbl-inc-def dbl-dec-def}) \\

\textbf{Implementations} \\
\textbf{lemma} \quad \text{one-integer-code [code, code-unfold]}: \\
\quad 1 = \text{Pos Num.One} \\
\quad \text{by simp} \\

\textbf{lemma} \quad \text{plus-integer-code [code]}: \\
\quad k + 0 = (k::integer) \\
\quad 0 + l = (l::integer) \\
\quad \text{Pos } m + \text{Pos } n = \text{Pos } (m + n) \\
\quad \text{Pos } m + \text{Neg } n = \text{sub } m \ n \\
\quad \text{Neg } m + \text{Pos } n = \text{sub } n \ m \\
\quad \text{Neg } m + \text{Neg } n = \text{Neg } (m + n) \\
\quad \text{by} & \ (\text{transfer}, \ \text{simp}) \\

\textbf{lemma} \quad \text{uminus-integer-code [code]}: \\
\quad \text{uminus } 0 = (0::integer) \\
\quad \text{uminus } (\text{Pos } m) = \text{Neg } m \\
\quad \text{uminus } (\text{Neg } m) = \text{Pos } m \\
\quad \text{by simp-all} \\

\textbf{lemma} \quad \text{minus-integer-code [code]}: \\
\quad k - 0 = (k::integer) \\
\quad 0 - l = \text{uminus } (l::integer) \\
\quad \text{Pos } m - \text{Pos } n = \text{sub } m \ n \\
\quad \text{Pos } m - \text{Neg } n = \text{Pos } (m + n) \\
\quad \text{Neg } m - \text{Pos } n = \text{Neg } (m + n) \\
\quad \text{Neg } m - \text{Neg } n = \text{sub } n \ m \\
\quad \text{by} & \ (\text{transfer}, \ \text{simp})
lemma abs-integer-code [code]:
|k| = (if (k::integer) < 0 then −k else k)
by simp

lemma sgn-integer-code [code]:
sign k = (if k = 0 then 0 else if (k::integer) < 0 then −1 else 1)
by simp

lemma times-integer-code [code]:
k * 0 = (0::integer)
0 * l = (0::integer)
Pos m * Pos n = Pos (m * n)
Pos m * Neg n = Neg (m * n)
Neg m * Pos n = Neg (m * n)
Neg m * Neg n = Pos (m * n)
by simp-all

definition divmod-integer :: integer ⇒ integer ⇒ integer × integer
where
divmod-integer k l = (k div l, k mod l)

lemma fst-divmod-integer [simp]:
fst (divmod-integer k l) = k div l
by (simp add: divmod-integer-def)

lemma snd-divmod-integer [simp]:
snd (divmod-integer k l) = k mod l
by (simp add: divmod-integer-def)

definition divmod-abs :: integer ⇒ integer ⇒ integer × integer
where
divmod-abs k l = (|k| div |l|, |k| mod |l|)

lemma fst-divmod-abs [simp]:
fst (divmod-abs k l) = |k| div |l|
by (simp add: divmod-abs-def)

lemma snd-divmod-abs [simp]:
snd (divmod-abs k l) = |k| mod |l|
by (simp add: divmod-abs-def)

lemma divmod-abs-code [code]:
divmod-abs (Pos k) (Pos l) = divmod k l
divmod-abs (Neg k) (Neg l) = divmod k l
divmod-abs (Neg k) (Pos l) = divmod k l
divmod-abs (Pos k) (Neg l) = divmod k l
divmod-abs j 0 = (0, |j|)
divmod-abs 0 j = (0, 0)
by (simp-all add: prod-eq-iff)
lemma divmod-integer-eq-cases:
\begin{align*}
\text{divmod-integer } k \l &= \\
& \quad (\text{if } k = 0 \text{ then } (0, 0) \text{ else if } l = 0 \text{ then } (0, k) \text{ else} \\
& \quad \text{apsnd } \circ \text{ times } \circ \text{ sgn } \ l \text{ (if sgn } k = \text{ sgn } l \\
& \quad \text{then divmod-abs } k \ l \\
& \quad \text{else let } (r, s) = \text{divmod-abs } k \ l \text{ in} \\
& \quad \text{if } s = 0 \text{ then } (- r, 0) \text{ else } (- r - 1, |l| - s)) \\
\end{align*}
proof –
\begin{align*}
\text{have } \ast : \text{ sgn } k = \text{ sgn } l \longleftrightarrow k = 0 \wedge l = 0 \vee 0 < l \vee 0 < k \wedge l < 0 \wedge k < 0 \\
\text{for } k \ l :: \text{ int} \\
& \quad \text{by } (\text{auto simp add: sgn-if}) \\
\text{have } \ast \ast : - k = l \ast q \longleftrightarrow k = -(l \ast q) \text{ for } k \ l q :: \text{ int} \\
& \quad \text{by } \text{auto} \\
\text{show } \ast \ast \ast \\
& \quad \text{by } (\text{simp add: divmod-integer-def divmod-abs-def}) \\
& \quad (\text{transfer, auto simp add: } \ast \ast \ast \text{ not-less zdiv-zminus1-eq-if zmod-zminus1-eq-if} \\
& \quad \text{div-minus-right mod-minus-right}) \\
\text{qed}
\end{align*}

lemma divmod-integer-code [code]:
\begin{align*}
\text{divmod-integer } k \ l &= \\
& \quad (\text{if } k = 0 \text{ then } (0, 0) \\
& \quad \text{else if } l > 0 \text{ then} \\
& \quad \quad (\text{if } k > 0 \text{ then Code-Numeral.divmod-abs } k \ l \\
& \quad \quad \text{else case Code-Numeral.divmod-abs } k \ l \text{ of } (r, s) \Rightarrow \\
& \quad \quad \text{if } s = 0 \text{ then } (- r, 0) \text{ else } (- r - 1, l - s)) \\
& \quad \text{else if } l = 0 \text{ then } (0, k) \\
& \quad \text{else apsnd uminus} \\
& \quad \quad (\text{if } k < 0 \text{ then Code-Numeral.divmod-abs } k \ l \\
& \quad \quad \text{else case Code-Numeral.divmod-abs } k \ l \text{ of } (r, s) \Rightarrow \\
& \quad \quad \text{if } s = 0 \text{ then } (- r, 0) \text{ else } (- r - 1, - l - s)) \\
& \quad \text{by } (\text{cases } l 0 :: \text{ integer rule: linorder-cases}) \\
& \quad \text{(auto split: prod.split simp add: divmod-integer-eq-cases}) \\
\end{align*}

lemma div-integer-code [code]:
k div l = fst (divmod-integer k l) 
by simp

lemma mod-integer-code [code]:
k mod l = snd (divmod-integer k l) 
by simp

definition bit-cut-integer :: integer => integer x bool
where bit-cut-integer k = (k div 2, odd k)

lemma bit-cut-integer-code [code]:
bit-cut-integer k = (if k = 0 then (0, False) \\
& \quad \text{else let } (r, s) = \text{Code-Numeral.divmod-abs } k \ 2
in (if \( k > 0 \) then \( r \) else \( -r - s \), \( s = 1 \))

**proof**

- **have** bit-cut-integer \( k = (let (r, s) = \text{divmod-integer} k 2 \text{ in } (r, s = 1)) \)
  - **by** (simp add: divmod-integer-def bit-cut-integer-def odd-iff-mod-2-eq-one)
- **then show** ?thesis
  - **by** (simp add: divmod-integer-code) (auto simp add: split-def)

**qed**

**lemma** equal-integer-code [code]:

HOL.equal 0 \((0::integer)\) \(\Longleftrightarrow\) True
HOL.equal 0 \((\text{Pos } l)\) \(\Longleftrightarrow\) False
HOL.equal 0 \((\text{Neg } l)\) \(\Longleftrightarrow\) False
HOL.equal \((\text{Pos } k)\) 0 \(\Longleftrightarrow\) False
HOL.equal \((\text{Pos } k)\) \((\text{Pos } l)\) \(\Longleftrightarrow\) HOL.equal \(k\) \(l\)
HOL.equal \((\text{Pos } k)\) \((\text{Neg } l)\) \(\Longleftrightarrow\) False
HOL.equal \((\text{Neg } k)\) 0 \(\Longleftrightarrow\) False
HOL.equal \((\text{Neg } k)\) \((\text{Pos } l)\) \(\Longleftrightarrow\) False
HOL.equal \((\text{Neg } k)\) \((\text{Neg } l)\) \(\Longleftrightarrow\) HOL.equal \(k\) \(l\)
**by** (simp-all add: equal)

**lemma** equal-integer-refl [code nbe]:

HOL.equal \((k::integer)\) \(k\) \(\Longleftrightarrow\) True
**by** (fact equal-refl)

**lemma** less-eq-integer-code [code]:

\(0 \leq (0::integer)\) \(\Longleftrightarrow\) True
\(0 \leq \text{Pos } l\) \(\Longleftrightarrow\) True
\(0 \leq \text{Neg } l\) \(\Longleftrightarrow\) False
\(\text{Pos } k \leq 0\) \(\Longleftrightarrow\) False
\(\text{Pos } k \leq \text{Pos } l\) \(\Longleftrightarrow\) \(k \leq l\)
\(\text{Pos } k \leq \text{Neg } l\) \(\Longleftrightarrow\) False
\(\text{Neg } k \leq 0\) \(\Longleftrightarrow\) True
\(\text{Neg } k \leq \text{Pos } l\) \(\Longleftrightarrow\) True
\(\text{Neg } k \leq \text{Neg } l\) \(\Longleftrightarrow\) \(l \leq k\)
**by** simp-all

**lemma** less-integer-code [code]:

\(0 < (0::integer)\) \(\Longleftrightarrow\) False
\(0 < \text{Pos } l\) \(\Longleftrightarrow\) True
\(0 < \text{Neg } l\) \(\Longleftrightarrow\) False
\(\text{Pos } k < 0\) \(\Longleftrightarrow\) False
\(\text{Pos } k < \text{Pos } l\) \(\Longleftrightarrow\) \(k < l\)
\(\text{Pos } k < \text{Neg } l\) \(\Longleftrightarrow\) False
\(\text{Neg } k < 0\) \(\Longleftrightarrow\) True
\(\text{Neg } k < \text{Pos } l\) \(\Longleftrightarrow\) True
\(\text{Neg } k < \text{Neg } l\) \(\Longleftrightarrow\) \(l < k\)
**by** simp-all

**lift-definition** num-of-integer :: integer \(\Rightarrow\) num
THEORY “Code-Numeral”

is num-of-nat ◦ nat
.

lemma num-of-integer-code [code]:
num-of-integer k = (if k ≤ 1 then Num.One
else let
  (l, j) = divmod-integer k 2;
  l' = num-of-integer l;
  l'' = l' + l'
in if j = 0 then l'' else l'' + Num.One)
proof –

assume int-of-integer k mod 2 = 1
then have nat (int-of-integer k mod 2) = nat 1 by simp
moreover assume *: 1 < int-of-integer k
ultimately have **: nat (int-of-integer k) mod 2 = 1 by (simp add: nat-mod-distrib)

have num-of-nat (nat (int-of-integer k)) =
  num-of-nat (2 * (nat (int-of-integer k) div 2) + nat (int-of-integer k) mod 2)
by simp

then have num-of-nat (nat (int-of-integer k)) =
  num-of-nat (nat (int-of-integer k) div 2 + nat (int-of-integer k) div 2 + nat
  (int-of-integer k) mod 2)
by (simp add: mult-2)

with ** have num-of-nat (nat (int-of-integer k)) =
  num-of-nat (nat (int-of-integer k) div 2 + nat (int-of-integer k) div 2 + 1)
by simp

} aux = this

note aux = this

show ?thesis
by (auto simp add: num-of-integer-def nat-of-integer-def Let-def case-prod-beta
  nat-le integer-eq-iff less-eq-integer-def
  nat-mul-distrib nat-div-distrib num-of-nat-One num-of-nat-plus-distrib
  mult-2 [where 'a=nat] aux add-One)

qed

lemma nat-of-integer-code [code]:
nat-of-integer k = (if k ≤ 0 then 0
else let
  (l, j) = divmod-integer k 2;
  l' = nat-of-integer l;
  l'' = l' + l'
in if j = 0 then l'' else l'' + 1)
proof –

obtain j where k: k = integer-of-int j

proof
  show k = integer-of-int (int-of-integer k) by simp

qed

have *: nat j mod 2 = nat-of-integer (of-int j mod 2) if j ≥ 0
  using that by transfer (simp add: nat-mod-distrib)
from k show thesis
  by (auto simp add: split-def Let-def nat-of-integer-def nat-div-distrib mult-2
    symmetric
    minus-mod-eq-mult-div [symmetric] *)
qed

lemma int-of-integer-code [code]:
int-of-integer k = (if k < 0 then - (int-of-integer (- k))
  else if k = 0 then 0
  else let
    (l, j) = divmod-integer k 2;
    l' = 2 * int-of-integer l
  in if j = 0 then l' else l' + 1)
by (auto simp add: split-def Let-def integer-eq-iff minus-mod-eq-mult-div [symmetric])

lemma integer-of-int-code [code]:
integer-of-int k = (if k < 0 then - (integer-of-int (- k))
  else if k = 0 then 0
  else let
    l = 2 * integer-of-int (k div 2);
    j = k mod 2
  in if j = 0 then l else l + 1)
by (auto simp add: split-def Let-def integer-eq-iff minus-mod-eq-mult-div [symmetric])

hide-const (open) Pos Neg sub dup divmod-abs

64.3 Serializer setup for target language integers

code-reserved Eval int Integer abs

code-printing
  type-constructor integer ->
    (SML) IntInf.int
and (OCaml) Z.t
and (Haskell) Integer
and (Scala) BigInt
and (Eval) int
| class-instance integer :: equal ->
  (Haskell) -

code-printing
  constant 0::integer ->
    (SML) !(0/::IntInf.int)
and (OCaml) Z.zero
and (Haskell) !(0/::Integer)
and (Scala) BigInt(0)

setup i
fold (fn target =>
THEORY "Code-Numeral"

        Numeral.add-code const-name (Code-Numeral.Pos) 1 Code-Printer.literal-numeral
        target
        #=> Numeral.add-code const-name (Code-Numeral.Neg) (~) Code-Printer.literal-numeral
        target
        [SML, OCaml, Haskell, Scala]
      )

      code-printing
      constant plus :: integer ⇒ - ⇒ - →
        (SML) IntInf.+ ((-), (-))
        and (OCaml) Z.add
        and (Haskell) infixl 6 +
        and (Scala) infixl 7 +
        and (Eval) infixl 8 +
      | constant aminus :: integer ⇒ - →
        (SML) IntInf.~
        and (OCaml) Z.neg
        and (Haskell) negate
        and (Scala) !(- -)
        and (Eval) ~/ -
      | constant minus :: integer ⇒ - →
        (SML) IntInf. - ((-), (-))
        and (OCaml) Z.sub
        and (Haskell) infixl 6 -
        and (Scala) infixl 7 -
        and (Eval) infixl 8 -
      | constant Code-Numeral.dup →
        (SML) IntInf.*/(2, (-))
        and (OCaml) Z.shift1-left/ -/ 1
        and (Haskell) !(2 * -)
        and (Scala) !(2 * -)
        and (Eval) !(2 * -)
      | constant Code-Numeral.sub →
        (SML) !(raise/ Fail/ sub)
        and (OCaml) failwith/ sub
        and (Haskell) error/ sub
        and (Scala) !sys.error(sub)
      | constant times :: integer ⇒ - ⇒ - →
        (SML) IntInf.* ((-), (-))
        and (OCaml) Z.mul
        and (Haskell) infixl 7 *
        and (Scala) infixl 8 *
        and (Eval) infixl 9 *
      | constant Code-Numeral.divmod-abs →
        (SML) IntInf.divMod/ (IntInf.abs -)/ IntInf.abs -
        and (OCaml) (!fun k l →/ if Z.equal Z.zero l then/ (Z.zero, l) else/ Z.div'-rem/
        (Z.abs k)/ (Z.abs l))
        and (Haskell) divMod/ (abs -)/ (abs -)
        and (Scala) !((k: BigInt) => (l: BigInt) =>/ if (l == 0)/ (BigInt(0), k)
else/ (k.abs /%' l.abs)
    and (Eval) Integer.div'-mod/ (abs -)/ (abs -)

| constant HOL.equal :: integer ⇒ bool ⇒ (SML) ((- :: IntInf.int) = -)
    and (OCaml) Z.equal
    and (Haskell) infix 4 ==
    and (Scala) infixl 5 ==
    and (Eval) infixl 6 ==

| constant less-eq :: integer ⇒ bool ⇒ (SML) IntInf.<= ((-), (-))
    and (OCaml) Z.leq
    and (Haskell) infix 4 <=
    and (Scala) infixl 4 <=
    and (Eval) infixl 6 <=

| constant less :: integer ⇒ bool ⇒ (SML) IntInf.< ((-), (-))
    and (OCaml) Z.lt
    and (Haskell) infix 4 <
    and (Scala) infixl 4 <
    and (Eval) infixl 6 <

| constant abs :: integer ⇒ (SML) IntInf.abs
    and (OCaml) Z.abs
    and (Haskell) Prelude.abs
    and (Scala) -.abs
    and (Eval) abs

**64.4 Type of target language naturals**

typedef natural = UNIV :: nat set

morphisms nat-of-natural natural-of-nat ..

setup-lifting type-definition-natural

lemma natural-eq-iff [termination-simp]:
    m = n ⇔ nat-of-natural m = nat-of-natural n
    by transfer rule

lemma natural-eqI:
    nat-of-natural m = nat-of-natural n ⇒ m = n
    using natural-eq-iff [of m n] by simp

lemma nat-of-natural-of-nat-inverse [simp]:
    nat-of-natural (natural-of-nat n) = n
    by transfer rule
lemma natural-of-nat-of-natural-inverse [simp]:
    natural-of-nat (nat-of-natural n) = n
    by transfer rule

instantiation natural :: {comm-monoid-diff, semiring-1}
begin

lift-definition zero-natural :: natural
    is 0 :: nat
    .

declare zero-natural.rep-eq [simp]

lift-definition one-natural :: natural
    is 1 :: nat
    .

declare one-natural.rep-eq [simp]

lift-definition plus-natural :: natural ⇒ natural ⇒ natural
    is plus :: nat ⇒ nat ⇒ nat
    .

declare plus-natural.rep-eq [simp]

lift-definition minus-natural :: natural ⇒ natural ⇒ natural
    is minus :: nat ⇒ nat ⇒ nat
    .

declare minus-natural.rep-eq [simp]

lift-definition times-natural :: natural ⇒ natural ⇒ natural
    is times :: nat ⇒ nat ⇒ nat
    .

declare times-natural.rep-eq [simp]

instance proof
qed (transfer, simp add: algebra-simps)+
end

instance natural :: Rings.dvd ..

context
    includes lifting-syntax
begin
lemma [transfer-rule]:
\((\text{pcr-natural} \iff \text{pcr-natural} \iff (\leftrightarrow))\) (dvd) (dvd)
by (unfold dvd-def) transfer-prover

lemma [transfer-rule]:
\((\leftrightarrow \implies \text{pcr-natural})\) of-bool of-bool
by (unfold of-bool-def) transfer-prover

lemma [transfer-rule]:
\((\lambda n. n) \text{ of-nat}\)
proof –
have rel-fun HOL.eq pcr-natural (of-nat :: nat \Rightarrow nat) (of-nat :: nat \Rightarrow natural)
by (unfold of-nat-def) transfer-prover
then show ?thesis by (simp add: id-def)
qed

lemma [transfer-rule]:
\((\lambda n. \text{of-nat} (\text{numeral n}))\)
proof –
have \((\lambda n. \text{of-nat} (\text{numeral n}))\)
by (transfer-prover)
then show ?thesis by simp
qed

lemma [transfer-rule]:
\((\text{pcr-natural} \iff \text{pcr-natural} \iff \text{power})\) (dvd)
by (unfold power-def) transfer-prover

end

lemma nat-of-natural-of-nat [simp]:
nat-of-natural (of-nat n) = n
by transfer rule

lemma natural-of-nat-of-nat [simp, code-abbrev]:
natural-of-nat = of-nat
by transfer rule

lemma of-nat-of-natural [simp]:
of-nat (nat-of-natural n) = n
by transfer rule

lemma nat-of-natural-numeral [simp]:
nat-of-natural (numeral k) = numeral k
by transfer rule

instantiation natural :: {linordered-semiring, equal} begin
lift-definition less-eq-natural :: natural ⇒ natural ⇒ bool
  is less-eq :: nat ⇒ nat ⇒ bool
.

declare less-eq-natural.rep-eq [termination-simp]

lift-definition less-natural :: natural ⇒ natural ⇒ bool
  is less :: nat ⇒ nat ⇒ bool
.

declare less-natural.rep-eq [termination-simp]

lift-definition equal-natural :: natural ⇒ natural ⇒ bool
  is HOL.equal :: nat ⇒ nat ⇒ bool
.

instance proof
qed (transfer, simp add: algebra-simps equal less-le-not-le [symmetric] linear)+
end

context
  includes lifting-syntax
begin

lemma [transfer-rule]:
  ⟨(pcr-natural ===⟩ pcr-natural ===⟩ pcr-natural min min⟩
by (unfold min-def) transfer-prover

lemma [transfer-rule]:
  ⟨(pcr-natural ===⟩ pcr-natural ===⟩ pcr-natural max max⟩
by (unfold max-def) transfer-prover
end

lemma nat-of-natural-min [simp]:
  nat-of-natural (min k l) = min (nat-of-natural k) (nat-of-natural l)
by transfer rule

lemma nat-of-natural-max [simp]:
  nat-of-natural (max k l) = max (nat-of-natural k) (nat-of-natural l)
by transfer rule

instantiation natural :: unique-euclidean-semiring
begin

lift-definition divide-natural :: natural ⇒ natural ⇒ natural
  is divide :: nat ⇒ nat ⇒ nat
.

```
THEORY "Code-Numeral"

declare divide-natural.rep-eq [simp]

lift-definition modulo-natural :: natural ⇒ natural ⇒ natural
  is modulo :: nat ⇒ nat ⇒ nat

declare modulo-natural.rep-eq [simp]

lift-definition euclidean-size-natural :: natural ⇒ nat
  is euclidean-size :: nat ⇒ nat

declare euclidean-size-natural.rep-eq [simp]

lift-definition division-segment-natural :: natural ⇒ natural
  is division-segment :: nat ⇒ nat

declare division-segment-natural.rep-eq [simp]

instance
  by (standard; transfer)
    (auto simp add: algebra-simps unit-factor-nat-def gr0-conv-Suc)

end

lemma [code]:
  euclidean-size = nat-of-natural
by (simp add: fun-eq-iff)

lemma [code]:
  division-segment (n::natural) = 1
by (simp add: natural-eq-iff)

instance natural :: linordered-semidom
  by (standard; transfer) simp-all

instance natural :: unique-euclidean-semiring-with-nat
  by (standard; transfer) simp-all

instantiation natural :: semiring-bit-shifts
begin

lift-definition push-bit-natural :: (nat ⇒ natural ⇒ natural)
  is (push-bit) .

lift-definition drop-bit-natural :: (nat ⇒ natural ⇒ natural)
  is (drop-bit) .
instance by (standard; transfer)
  (fact bit-eq-rec bits-induct push-bit-eq-mult drop-bit-eq-div
   bits-div-0 bits-div-by-1 bits-mod-div-trivial even-succ-div-2
   exp-div-exp-eq div-exp-eq mod-exp-eq mult-exp-mod-exp-eq div-exp-mod-exp-eq
   even-mask-div-iff even-mult-exp-div-exp-iff)+
end

context
  includes lifting-syntax
begin

lemma [transfer-rule]:
  ⟨(pcr-natural ===> (=) ===> (↔)) bit bit⟩
  by (unfold bit-def) transfer-prover

lemma [transfer-rule]:
  ⟨((=) ===> pcr-natural ===> pcr-natural) take-bit take-bit⟩
  by (unfold take-bit-eq-mod) transfer-prover

end

instance natural :: unique-euclidean-semiring-with-bit-shifts..

lemma [code]:
  ⟨(push-bit n m = m * 2 ^ n)⟩
  ⟨(drop-bit n m = m div 2 ^ n) for m :: natural⟩
  by (fact push-bit-eq-mult drop-bit-eq-div)+

lift-definition natural-of-integer :: integer ⇒ natural
  is nat :: int ⇒ nat
  .

lift-definition integer-of-natural :: natural ⇒ integer
  is of-nat :: nat ⇒ int
  .

lemma natural-of-integer-of-natural [simp]:
  natural-of-integer (integer-of-natural n) = n
  by transfer simp

lemma integer-of-natural-of-integer [simp]:
  integer-of-natural (natural-of-integer k) = max 0 k
  by transfer auto

lemma int-of-integer-of-natural [simp]:
  int-of-integer (integer-of-natural n) = of-nat (nat-of-natural n)
  by transfer rule
lemma integer-of-natural-of-nat [simp]:
integer-of-natural (of-nat n) = of-nat n
by transfer rule

lemma [measure-function]:
is-measure nat-of-natural
by (rule is-measure-trivial)

64.5 Inductive representation of target language naturals

lift-definition Suc :: natural ⇒ natural
  is NatSuc.

declare Suc.rep-eq [simp]

old-rep-datatype 0::natural Suc
  by (transfer, fact nat.induct nat.inject nat.distinct)+

lemma natural-cases [case-names nat, cases type: natural]:
  fixes m :: natural
  assumes \( \land n. m = of-nat n \Rightarrow P \)
  shows P
  using assms by transfer blast

instantiation natural :: size
begin

definition size-nat where [simp, code]: size-nat = nat-of-natural
instance ..
end

lemma natural-decr [termination-simp]:
  \( n \neq 0 \Rightarrow \) nat-of-natural n − NatSuc 0 < nat-of-natural n
by transfer simp

lemma natural-zero-minus-one: (0::natural) − 1 = 0
by (rule zero-diff)

lemma Suc-natural-minus-one: Suc n − 1 = n
by transfer simp

hide-const (open) Suc

64.6 Code refinement for target language naturals

lift-definition Nat :: integer ⇒ natural
lemma [code-post]:
Nat 0 = 0
Nat 1 = 1
Nat (numeral k) = numeral k
by (transfer, simp)+

lemma [code abstype]:
Nat (integer-of-natural n) = n
by transfer simp

lemma [code]:
natural-of-nat n = natural-of-integer (integer-of-nat n)
by transfer simp

lemma [code abstract]:
integer-of-natural (natural-of-integer k) = max 0 k
by simp

lemma [code abbrev]:
natural-of-integer (Code-Numeral.Pos k) = numeral k
by transfer simp

lemma [code abstract]:
integer-of-natural 0 = 0
by transfer simp

lemma [code abstract]:
integer-of-natural 1 = 1
by transfer simp

lemma [code abstract]:
integer-of-natural (Code-Numeral.Suc n) = integer-of-natural n + 1
by transfer simp

lemma [code]:
nat-of-natural = nat-of-integer o integer-of-natural
by transfer (simp add: fun-eq-iff)

lemma [code, code-unfold]:
case-natural f g n = (if n = 0 then f else g (n - 1))
by (cases n rule: natural.exhaust) (simp-all, simp add: Suc-def)

declare natural.rec [code del]

lemma [code abstract]:
integer-of-natural (m + n) = integer-of-natural m + integer-of-natural n
by transfer simp

lemma [code abstract]:
  integer-of-natural (m - n) = max 0 (integer-of-natural m - integer-of-natural n)
  by transfer simp

lemma [code abstract]:
  integer-of-natural (m * n) = integer-of-natural m * integer-of-natural n
  by transfer simp

lemma [code abstract]:
  integer-of-natural (m div n) = integer-of-natural m div integer-of-natural n
  by transfer (simp add: zdiv-int)

lemma [code abstract]:
  integer-of-natural (m mod n) = integer-of-natural m mod integer-of-natural n
  by transfer (simp add: zmod-int)

lemma [code]:
  HOL.equal m n ↔ HOL.equal (integer-of-natural m) (integer-of-natural n)
  by transfer (simp add: equal)

lemma [code nbe]: HOL.equal n ::natural n ↔ True
  by (rule equal-class.equal-refl)

lemma [code]: m ≤ n ↔ integer-of-natural m ≤ integer-of-natural n
  by transfer simp

lemma [code]: m < n ↔ integer-of-natural m < integer-of-natural n
  by transfer simp

hide-const (open) Nat

lifting-update integer.lifting
lifting-forget integer.lifting

lifting-update natural.lifting
lifting-forget natural.lifting

code-reflect Code-Numeral
datatypes natural
functions Code-Numeral.Suc 0 :: natural 1 :: natural
  plus :: natural ⇒ - minus :: natural ⇒ -
  times :: natural ⇒ - divide :: natural ⇒ -
  modulo :: natural ⇒ -
  integer-of-natural natural-of-integer

end
65 Setup for Lifting/Transfer for the set type

theory Lifting-Set
imports Lifting
begin

65.1 Relator and predecator properties

lemma rel-setD1: \[ \rel-set R A B; x \in A \implies \exists y \in B. R x y \]
and \( \rel-setD2: \rel-set R A B; y \in B \implies \exists x \in A. R x y \)
by (simp-all add: rel-set-def)

lemma rel-set-conversep \(\simp\): rel-set A \{-1\} \{-1\} = (rel-set A)^{-1-1}
unfolding rel-set-def by auto

lemma rel-set-eq \(\relator-eq\): rel-set (=) = (=)
unfolding rel-set-def fun-eq-iff by auto

lemma rel-set-mono \(\relator-mono\):
assumes A \subseteq B
shows rel-set A \subseteq rel-set B
using assms unfolding rel-set-def by blast

lemma rel-set-OO \(\relator-distr\): rel-set R OO rel-set S = rel-set (R OO S)
apply (rule sym)
apply (intro ext)
subgoal for X Z
  apply (rule iffI)
  apply (rule relcomppI [where b\{y. (\exists x\in X. R x y) \land (\exists z\in Z. S y z)\}])
  apply (simp add: rel-set-def, fast)+
  done
done

lemma Domainp-set \(\relator-domain\):
  Domainp (rel-set T) = (\lambda A. Ball A (Domainp T))
unfolding rel-set-def Domainp-iff[abs-def]
apply (intro ext)
apply (rule iffI)
apply blast
subgoal for A by (rule exI [where x\{y. \exists x\in A. T x y\}]) fast
  done

lemma left-total-rel-set \(\transfer-rule\):
  left-total A \implies left-total (rel-set A)
unfolding left-total-def rel-set-def
apply safe
subgoal for X by (rule exI [where x\{y. \exists x\in X. A x y\}]) fast
  done

lemma left-unique-rel-set \(\transfer-rule\):

THEORY "Lifting-Set"

left-unique A \implies left-unique (rel-set A)

unfolding left-unique-def rel-set-def by fast

lemma right-total-rel-set [transfer-rule]:
right-total A \implies right-total (rel-set A)
using left-total-rel-set[of A^{-1}^{-1}] by simp

lemma right-unique-rel-set [transfer-rule]:
right-unique A \implies right-unique (rel-set A)
unfolding right-unique-def rel-set-def by fast

lemma bi-total-rel-set [transfer-rule]:
bi-total A \implies bi-total (rel-set A)
by (simp add: bi-total-alt-def left-total-rel-set right-total-rel-set)

lemma set-relator-eq-onp [relator-eq-onp]:
rel-set (eq-onp P) = eq-onp (\lambda A. Ball A P)
unfolding fun-eq-iff rel-set-def eq-onp-def Ball-def by fast

lemma bi-unique-rel-set-lemma:
assumes bi-unique R and rel-set R X Y
obtains f where Y = image f X and inj-on f X and \forall x\in X. R x (f x)
proof
define f where f x = (THE y. R x y) for x
{ fix x assume x \in X 
  with (rel-set R X Y) (bi-unique R) have R x (f x)
  by (simp add: bi-unique-def rel-set-def f-def) (metis theI)
  with assms (x \in X)
  have R x (f x) \forall x'\in X. R x' (f x) \implies x = x' \forall y\in Y. R x y \implies y = f x f x
  \in Y
  by (fastforce simp add: bi-unique-def rel-set-def)+ } 
note * = this
moreover
{ fix y assume y \in Y 
  with (rel-set R X Y) *(3) (y \in Y) have \exists x\in X. y = f x
  by (fastforce simp: rel-set-def) } 
ultimately show \forall x\in X. R x (f x) Y = image f X inj-on f X
  by (auto simp: inj-on-def image-iff)
qed

65.2 Quotient theorem for the Lifting package

lemma Quotient-set[quot-map]:
assumes Quotient R Abs Rep T
shows Quotient (rel-set R) (image Abs) (image Rep) (rel-set T)
using assms unfolding Quotient-alt-def4
apply (simp add: rel-set-OO[symmetric])
apply (simp add: rel-set-def)
apply fast
done

65.3 Transfer rules for the Transfer package

65.3.1 Unconditional transfer rules
context includes lifting-syntax
begin

lemma empty-transfer [transfer-rule]: (rel-set A) {} {}
  unfolding rel-set-def by simp

lemma insert-transfer [transfer-rule]:
  (A ==> rel-set A ==> rel-set A) insert insert
  unfolding rel-fun-def rel-set-def by auto

lemma union-transfer [transfer-rule]:
  (rel-set A ==> rel-set A ==> rel-set A) union union
  unfolding rel-fun-def rel-set-def by auto

lemma Union-transfer [transfer-rule]:
  (rel-set (rel-set A) ==> rel-set A) Union Union
  unfolding rel-fun-def rel-set-def by simp fast

lemma image-transfer [transfer-rule]:
  ((A ==> B) ==> rel-set A ==> rel-set B) image image
  unfolding rel-fun-def rel-set-def by simp fast

lemma UNION-transfer [transfer-rule]: — TODO deletion candidate
  (rel-set A ==> (A ==> rel-set B) ==> rel-set B) (\A f. \(A f. \bigcup (f^\prime A)))
  (\A f. \bigcup (f^\prime A))
  by transfer-prover

lemma Ball-transfer [transfer-rule]:
  (rel-set A ==> (A ==> (=)) ==> (=)) Ball Ball
  unfolding rel-set-def rel-fun-def by fast

lemma Bex-transfer [transfer-rule]:
  (rel-set A ==> (A ==> (=)) ==> (=)) Bex Bex
  unfolding rel-set-def rel-fun-def by fast

lemma Pow-transfer [transfer-rule]:
  (rel-set A ==> rel-set (rel-set A)) Pow Pow
  apply (rule rel-funI)
  apply (rule rel-setI)
subgoal for $X Y X'$

apply (rule rev-bexI [where $x = \{ y \in Y . \exists x \in X'. \ A \ x \ y \}])
apply clarsimp
apply (simp add: rel-set-def)
apply fast
done

subgoal for $X Y Y'$

apply (rule rev-bexI [where $x = \{ x \in X . \exists y \in Y'. \ A \ x \ y \}])
apply clarsimp
apply (simp add: rel-set-def)
apply fast
done

done

lemma rel-set-transfer [transfer-rule]:
\[(A === B ===> (=)) ===> \text{rel-set} A ===> \text{rel-set} B ===> (=) \text{rel-set} \]
unfolding rel-fun-def rel-set-def by fast

lemma bind-transfer [transfer-rule]:
\[(\text{rel-set} A ===> (A ===> \text{rel-set} B) ===> \text{rel-set} B) \text{ Set.bind Set.bind} \]
unfolding bind-UNION [abs-def] by transfer-prover

lemma INF-parametric [transfer-rule]: — TODO deletion candidate
\[(\text{rel-set} A ===> (A ===> \text{HOL.eq}) ===> \text{HOL.eq}) (\lambda A f. \text{Inf} (f' A)) (\lambda A f. \text{Inf} (f' A)) \]
by transfer-prover

lemma SUP-parametric [transfer-rule]: — TODO deletion candidate
\[(\text{rel-set} R ===> (R ===> \text{HOL.eq}) ===> \text{HOL.eq}) (\lambda A f. \text{Sup} (f' A)) (\lambda A f. \text{Sup} (f' A)) \]
by transfer-prover

65.3.2 Rules requiring bi-unique, bi-total or right-total relations

lemma member-transfer [transfer-rule]:
assumes bi-unique $A$
shows $(A ===> \text{rel-set} A ===> (=)) (\in) (\in)$
using assms unfolding rel-fun-def rel-set-def bi-unique-def by fast

lemma right-total-Collect-transfer [transfer-rule]:
assumes right-total $A$
shows $(A ===> (=)) ===> \text{rel-set} A (\lambda P. \text{Collect} (\lambda x. P x \land \text{Domainp} A x)) \text{ Collect}$
using assms unfolding right-total-def rel-set-def rel-fun-def Domainp-iff by fast

lemma Collect-transfer [transfer-rule]:
assumes bi-total $A$
shows $(A ===> (=)) ===> \text{rel-set} A) \text{ Collect Collect}$
using assms unfolding rel-fun-def rel-set-def bi-total-def by fast

lemma inter-transfer [transfer-rule]:
  assumes bi-unique A
  shows (rel-set A ===> rel-set A ===> rel-set A) inter inter
  using assms unfolding rel-fun-def rel-set-def bi-unique-def by fast

lemma Diff-transfer [transfer-rule]:
  assumes bi-unique A
  shows (rel-set A ===> rel-set A ===> rel-set A) (-) (-)
  using assms unfolding rel-fun-def rel-set-def bi-unique-def
  unfolding Ball-def Bex-def Diff-eq
  by (safe, simp, metis, simp, metis)

lemma subset-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (rel-set A ===> rel-set A ===> (=)) (⊆) (⊆)
  unfolding subset-eq [abs-def] by transfer-prover

context
  includes lifting-syntax
begin

lemma strict-subset-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (rel-set A ===> rel-set A ===> (=)) (⊂) (⊂)
  unfolding subset-not-subset-eq by transfer-prover
end

declare right-total-UNIV-transfer[transfer-rule]

lemma UNIV-transfer [transfer-rule]:
  assumes bi-total A
  shows (rel-set A) UNIV UNIV
  using assms unfolding rel-set-def bi-total-def by simp

lemma right-total-Compl-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A and [transfer-rule]: right-total A
  shows (rel-set A ===> rel-set A) (λS. uminus S ∩ Collect (Domainp A)) uminus
  unfolding Compl-eq [abs-def]
  by (subst Collect-conj-eq [symmetric]) transfer-prover

lemma Compl-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A and [transfer-rule]: bi-total A
  shows (rel-set A ===> rel-set A) uminus uminus
  unfolding Compl-eq [abs-def] by transfer-prover
lemma right-total-Inter-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A and [transfer-rule]: right-total A
  shows ( RELSET (RELSET A) === > RELSET A) (LAMBDA S. INTER S \cap COLLECT (DOMAINP A))

Inter
  unfolding Inter-eq[abs-def]
  by (subst COLLECT-COLLECT[ symmetric]) transfer-prover

lemma Inter-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A and [transfer-rule]: bi-total A
  shows ( RELSET (RELSET A) === > RELSET A) INTER Inter

unfolding Inter-eq[abs-def] by transfer-prover

lemma filter-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows ((A === > (=)) === > RELSET A === > RELSET A) SETFILTER SETFILTER

unfolding SETFILTER[abs-def] relfun-def rel-set-def by blast

lemma finite-transfer [transfer-rule]:
  bi-unique A \implies (RELSET A === > (=)) finite finite

by (rule relfunI, erule (1) bi-unique-rel-set-lemma)
  (auto dest: finite-imageD)

lemma card-transfer [transfer-rule]:
  bi-unique A \implies (RELSET A === > (=)) card card

by (rule relfunI, erule (1) bi-unique-rel-set-lemma)
  (simp add: card-image)

context
  includes lifting-syntax

begin

lemma vimage-right-total-transfer[transfer-rule]:
  assumes [transfer-rule]: bi-unique B right-total A
  shows ((A === > B) === > RELSET B === > RELSET A) (LAMBDAX. F \vdash X \cap COLLECT (DOMAINP A)) VIMAGE

proof
  let \$vimage = (LAMBDAX. \{X. F x \in B \land DOMAINP A x\})

  have ((A === > B) === > RELSET B === > RELSET A) \$vimage vimage
    unfolding vimage-def
    by transfer-prover

  also have \$vimage = (LAMBDAX. F \vdash X \cap COLLECT (DOMAINP A))
    by auto

  finally show \$thesis.

qed

end

lemma vimage-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-total A bi-unique B
THEORY "List"

shows \((A \rightarrow B) \rightarrow \text{rel-set } A \rightarrow \text{rel-set } B)\) vimage vimage
unfolding vimage-def[abs-def] by transfer-prover

lemma Image-parametric [transfer-rule]:
assumes bi-unique A
shows \((\text{rel-set } (\text{rel-prod } A \times B)) \rightarrow \text{rel-set } A \rightarrow \text{rel-set } B)\) (""") ("")
by (intro rel-funI rel-setI)
(force dest: rel-setD1 bi-uniqueDr[OF assms], force dest: rel-setD2 bi-uniqueDr[OF assms])

lemma inj-on-transfer[transfer-rule]:
\((A \rightarrow B) \rightarrow \text{rel-set } A \rightarrow \text{rel-set } B)\) inj-on inj-on
if [transfer-rule]: bi-unique A bi-unique B
unfolding inj-on-def
by transfer-prover

end

lemma (in comm-monoid-set) F-parametric [transfer-rule]:
fixes A :: 'b => 'c => bool
assumes bi-unique A
shows \((\text{rel-fun } (\text{rel-fun } A (\equiv)) \text{ (rel-fun } (\text{rel-set } A) (\equiv)) F F\)
proof (rule rel-funI)+
fix f :: 'b => 'a and g S T
assume rel-fun A (=) f g rel-set A S T
with (bi-unique A) obtain i where bij-betw i S T \(\forall x. x \in S \rightarrow f x = g (i x)\)
by (auto elim: bi-unique-rel-set-lemma simp: rel-fun-def bij-betw-def)
then show F f S = F g T
by (simp add: reindex-bij-betw)
qed

lemmas sum-parametric = sum.F-parametric
lemmas prod-parametric = prod.F-parametric

lemma rel-set-UNION:
assumes [transfer-rule]: \text{rel-set } Q A B \text{ rel-fun } Q \text{ (rel-set } R) f g
shows \text{rel-set } R (\bigcup (f \cdot A)) (\bigcup (g \cdot B))
by transfer-prover

end

66 The datatype of finite lists

theory List
imports Sledgehammer Code-Numeral Lifting-Set
begin

datatype (set: 'a) list =
Nil ([])
THEORY "List"
by (induct xs) auto

**Definition** coset :: 'a list ⇒ 'a set where

[simp]: coset xs = − set xs

**Primrec** append :: 'a list ⇒ 'a list ⇒ 'a list (infixr @ 65) where

append-Nil: [] @ ys = ys |
append-Cons: (x#xs) @ ys = x @ xs @ ys

**Primrec** rev :: 'a list ⇒ 'a list where

rev [] = [] |
rev (x # xs) = rev xs @ [x]

**Primrec** filter :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a list where

filter P [] = [] |
filter P (x # xs) = (if P x then x # xs else filter P xs)

Special input syntax for filter:

**Syntax** (ASCII)

- filter :: [pttrn, 'a list, bool] ⇒ 'a list ((1[-<-.-/ -]))

**Syntax**

- filter :: [pttrn, 'a list, bool] ⇒ 'a list ((1[-<-.-/ -]))

**Translations**

[x<xs . P] → CONST filter (λx. P) xs

**Primrec** fold :: ('a ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b ⇒ 'b where

fold-Nil: fold f [] = id |
fold-Cons: fold f (x # xs) = fold f xs o f x

**Primrec** foldr :: ('a ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b ⇒ 'b where

foldr-Nil: foldr f [] = id |
foldr-Cons: foldr f (x # xs) = f x o foldr f xs

**Primrec** foldl :: ('b ⇒ 'a ⇒ 'b) ⇒ 'a list ⇒ 'b ⇒ 'b where

foldl-Nil: foldl f a [] = a |
foldl-Cons: foldl f a (x # xs) = foldl f (f a x) xs

**Primrec** concat :: 'a list list ⇒ 'a list where

concat [] = [] |
concat (x # xs) = x @ concat xs

**Primrec** drop :: nat ⇒ 'a list ⇒ 'a list where

drop-Nil: drop n [] = [] |
drop-Cons: drop n (x # xs) = (case n of 0 ⇒ x # xs | Suc m ⇒ drop m xs)

— Warning: simpset does not contain this definition, but separate theorems for n = 0 and n = Suc k

**Primrec** take :: nat ⇒ 'a list ⇒ 'a list where

take-Nil: take n [] = [] |
take-Cons: take n (x # xs) = (case n of 0 ⇒ [] | Suc m ⇒ x # take m xs)
— Warning: simpset does not contain this definition, but separate theorems for
n = 0 and n = Suc k

primrec (nonexhaustive) nth :: 'a list ⇒ nat ⇒ 'a (infixl ! 100) where
nth-Cons: (x # xs)!n = (case n of 0 ⇒ x | Suc k ⇒ xs!k)
— Warning: simpset does not contain this definition, but separate theorems for
n = 0 and n = Suc k

primrec list-update :: 'a list ⇒ nat ⇒ 'a ⇒ 'a list where
list-update [] i v = []
list-update (x # xs) i v = (case i of 0 ⇒ v | Suc j ⇒ x # list-update xs j v)
— Warning: simpset does not contain this definition, but separate theorems for
xs = [] and xs = z # zs

abbreviation map2 :: ('a ⇒ 'b ⇒ 'c) ⇒ 'a list ⇒ 'b list ⇒ 'c list where
map2 f xs ys ≡ map (λ(x,y). f x y) (zip xs ys)

primrec product :: 'a list ⇒ 'b list ⇒ ('a × 'b) list where
product [] - = []
product (x#xs) ys = map (Pair x) ys @ product xs ys
hide-const (open) product
primrec product-lists :: 'a list list ⇒ 'a list list where
product-lists [] = [] |
product-lists (xs # xss) = concat (map (λx. map (Cons) (product-lists xss)) xs)

primrec upt :: nat ⇒ nat ⇒ nat list where
upt-0: [i..<0] = [] |
upt-Suc: [i..<(Suc j)] = (if i ≤ j then [i..<j] @ [j] else [])
definition insert :: 'a ⇒ 'a list ⇒ 'a list where
insert x xs = (if x ∈ set xs then xs else x # xs)
definition union :: 'a list ⇒ 'a list ⇒ 'a list where
union = fold insert
hide-const (open) insert union
hide-fact (open) insert-def union-def

primrec find :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a option where
find - [] = None |
find P (x#xs) = (if P x then Some x else find P xs)

In the context of multisets, count-list is equivalent to count ∘ mset and it
it advisable to use the latter.

primrec count-list :: 'a list ⇒ 'a ⇒ nat where
count-list [] y = 0 |
count-list (x#xs) y = (if x=y then count-list xs y + 1 else count-list xs y)
definition extract :: ('a ⇒ bool) ⇒ 'a list ⇒ ('a list * 'a * 'a list) option
where extract P xs =
  (case dropWhile (Not ∘ P) xs of
    [] ⇒ None |
    y#ys ⇒ Some(takeWhile (Not ∘ P) xs, y, ys))
hide-const (open) extract

primrec those :: 'a option list ⇒ 'a list option where
those [] = Some [] |
those (x # xs) = (case x of
  None ⇒ None |
  Some y ⇒ map-option (Cons) y (those xs))

primrec remove1 :: 'a ⇒ 'a list ⇒ 'a list where
remove1 x [] = [] |
remove1 x (y # xs) = (if x = y then xs else y # remove1 x xs)

primrec removeAll :: 'a ⇒ 'a list ⇒ 'a list where
removeAll x [] = [] |
removeAll x (y # xs) = (if x = y then removeAll x xs else y # removeAll x xs)

primrec distinct :: 'a list ⇒ bool where
distinct [] ←→ True |
distinct (x # xs) ←→ x ∉ set xs ∧ distinct xs

primrec remdups :: 'a list ⇒ 'a list where
remdups [] = [] |
remdups (x # xs) = (if x ∈ set xs then remdups xs else x # remdups xs)

fun remdups-adj :: 'a list ⇒ 'a list where
remdups-adj [] = [] |
remdups-adj [x] = [x] |
remdups-adj (x # y # xs) = (if x = y then remdups-adj (x # xs) else x # remdups-adj (y # xs))

primrec replicate :: nat ⇒ 'a ⇒ 'a list where
replicate-0: replicate 0 x = [] |
replicate-Suc: replicate (Suc n) x = x # replicate n x

Function size is overloaded for all datatypes. Users may refer to the list version as length.

abbreviation length :: 'a list ⇒ nat where
length ≡ size

definition enumerate :: nat ⇒ 'a list ⇒ (nat × 'a) list where
enumerate-eq-zip: enumerate n xs = zip [n..<n + length xs] xs

primrec rotate1 :: 'a list ⇒ 'a list where
rotate1 [] = [] |
rotate1 (x # xs) = xs @ [x]

definition rotate :: nat ⇒ 'a list ⇒ 'a list where
rotate n = rotate1 ✡ n

definition nths :: 'a list =⇒ nat set =⇒ 'a list where
nths xs A = map fst (filter (λp. snd p ∈ A) (zip xs [0..<size xs]))

primrec subseqs :: 'a list ⇒ 'a list list where
subseqs [] = [[]] |
subseqs (x#xs) = (let xss = subseqs xs in map (Cons x) xss @ xss)

primrec n-lists :: nat ⇒ 'a list ⇒ 'a list list where
n-lists 0 xs = [[]] |
n-lists (Suc n) xs = concat (map (λys. map (λy # ys) xs) (n-lists n xs))

hide-const (open) n-lists
function splice :: 'a list ⇒ 'a list ⇒ 'a list where
splice [] ys = ys |
splice (x#xs) ys = x # splice ys xs
by pat-completeness auto

termination
by (relation measure (\(\lambda (xs, ys). \text{size} \hspace{1pt} xs + \text{size} \hspace{1pt} ys\))) auto

function shuffles where
shuffles [] ys = \{ys\} |
shuffles xs [] = \{xs\} |
shuffles (x # xs) (y # ys) = (#) x ' shuffles xs (y # ys) \cup (#) y ' shuffles (x # xs) ys
by pat-completeness simp-all
termination by lexicographic-order

Use only if you cannot use Min instead:

fun min-list :: 'a::ord list ⇒ 'a where
min-list (x # xs) = (case xs of [] ⇒ x | - ⇒ min x (min-list xs))

Returns first minimum:

fun arg-min-list :: ('a ⇒ ('b::linorder)) ⇒ 'a list ⇒ 'a where
arg-min-list f [] = x |
arg-min-list f (x#y#zs) = (let m = arg-min-list f (y#zs) in if f x ≤ f m then x else m)

Figure 1 shows characteristic examples that should give an intuitive understanding of the above functions.

The following simple sort functions are intended for proofs, not for efficient implementations.

A sorted predicate w.r.t. a relation:

fun sorted-wrt :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ bool where
sorted-wrt P [] = True |
sorted-wrt P (x # ys) = ((\forall y \in \text{set} \hspace{1pt} ys. \hspace{1pt} P \hspace{1pt} x \hspace{1pt} y) \land \hspace{1pt} \text{sorted-wrt} \hspace{1pt} P \hspace{1pt} ys)

A class-based sorted predicate:

context linorder
begin

fun sorted :: 'a list ⇒ bool where
sorted [] = True |
sorted (x # ys) = ((\forall y \in \text{set} \hspace{1pt} ys. \hspace{1pt} x \leq y) \land \hspace{1pt} \text{sorted} \hspace{1pt} ys)

fun strict-sorted :: 'a list ⇒ bool where
strict-sorted [] = True |
strict-sorted (x # ys) = ((\forall y \in \text{List.set} \hspace{1pt} ys. \hspace{1pt} x < y) \land \hspace{1pt} \text{strict-sorted} \hspace{1pt} ys)
[a, b] @ [c, d] = [a, b, c, d]
length [a, b, c] = 3
set [a, b, c] = {a, b, c}
map f [a, b, c] = [f a, f b, f c]
rev [a, b, c] = [c, b, a]
hd [a, b, c, d] = a
tl [a, b, c, d] = [b, c, d]
last [a, b, c, d] = d
butlast [a, b, c, d] = [a, b, c]
filter (λn::nat. n<2) [0,2,1] = [0,1]
concat [[a, b], [c, d, e], [], [f]] = [a, b, c, d, e, f]
fold f [a, b, c] x = f c (f b (f a x))
foldr f [a, b, c] x = f a (f b (f c x))
foldl f x [a, b, c] = f (f x a) b c
zip [a, b, c] [x, y, z] = [(a, x), (b, y), (c, z)]
zip [a, b] [x, y, z] = [(a, x), (b, y)]
enumerate 3 [a, b, c] = [(3, a), (4, b), (5, c)]
List.product [a, b] [c, d] = [(a, c), (a, d), (b, c), (b, d)]
product-lists [a, b] [c, d] = [(a, c, d), [a, c, e], [b, c, d], [b, c, e]]
splice [a, b, c] [x, y, z] = [a, x, b, y, c, z]
splice [a, b, c, d] [x, y] = [a, x, b, y, c, d]
shuffles [a, b] [c, d] = {[a, b, c, d], [a, c, d, b], [a, c, b, d], [c, a, b, d], [c, a, d, b], [c, d, a, b]}
take 2 [a, b, c, d] = [a, b]
take 6 [a, b, c, d] = [a, b, c, d]
drop 2 [a, b, c, d] = [c, d]
drop 6 [a, b, c, d] = []
takeWhile (λn. n < 3) [1, 2, 3, 0] = [1, 2]
dropWhile (λn. n < 3) [1, 2, 3, 0] = [3, 0]
distinct [2, 0, 1]
remdups [2, 0, 2, 2, 1, 2] = [0, 1, 2]
remdups-adj [2, 0, 2, 3, 1, 1, 2, 1] = [2, 3, 1, 2, 1]
List.insert 2 [0, 1, 2] = [0, 1, 2]
List.insert 3 [0, 1, 2] = [3, 0, 1, 2]
List.union [2, 3, 4] [0, 1, 2] = [4, 3, 0, 1, 2]
find (λx. x < 0) [0, 0] = None
find (λx. x < 0) [0, 0, 0, 2] = Some 1
count-list [0, 1, 0, 2] 0 = 2
List.extract (λx. x < 0) [0, 0] = None
List.extract (λx. x < 0) [0, 1, 0, 2] = Some ([0], [1], [0, 2])
removeI 2 [2, 0, 2, 1, 2] = [0, 2, 1, 2]
removeAll 2 [2, 0, 2, 1, 2] = [0, 1, 2]
[a, b, c, d] ! 2 = c
[a, b, c, d][x := a] = [a, b, x, d]
ntns [a, b, c, d, e] {0, 2, 3} = [a, c, d]
subsqs [a, b] = [[a, b], [a], [b], []]
List.n-lists 2 [a, b, c] = [[a, a], [b, a], [c, a], [a, b], [b, b], [c, b], [a, c], [b, c], [c, c]]
rotateI [a, b, c, d] = [b, c, d, a]
rotate 3 [a, b, c, d] = [d, a, b, c]
replicate 4 a = [a, a, a, a]
[2..5] = [2, 3, 4]
min-list [3, 1, -2] = -2
arg-min-list (λx. i * i) [3, -1, 1, -2] = -1
lemma sorted-sorted-wrt: sorted = sorted-wrt (≤)
proof (rule ext)
  fix xs show sorted xs = sorted-wrt (≤) xs
  by(induction xs rule: sorted.induct) auto
qed

lemma strict-sorted-sorted-wrt: strict-sorted = sorted-wrt (<)
proof (rule ext)
  fix xs show strict-sorted xs = sorted-wrt (<) xs
  by(induction xs rule: strict-sorted.induct) auto
qed

primrec insort-key :: (′b ⇒ ′a) ⇒ ′b list ⇒ ′b list
where
  insort-key f x [] = [x] |
  insort-key f x (y#ys) =
  (if f x ≤ f y then (x#y)#ys else y#(insort-key f x ys))

definition sort-key :: (′b ⇒ ′a) ⇒ ′b list ⇒ ′b list
where
  sort-key f xs = foldr (insort-key f) xs []

definition insort-insert-key :: (′b ⇒ ′a) ⇒ ′b list ⇒ ′b list
where
  insort-insert-key f x xs =
  (if f x ∈ f ' set xs then xs else insort-key f x xs)

abbreviation sort ≡ sort-key (λx. x)
abbreviation insort ≡ insort-key (λx. x)
abbreviation insort-insert ≡ insort-insert-key (λx. x)

definition stable-sort-key :: (′b ⇒ ′a) ⇒ ′b list ⇒ ′b list ⇒ bool
where
  stable-sort-key sk =
  (∀ f xs k. filter (λy. f y = k) (sk f xs) = filter (λy. f y = k) xs)
end

66.1.1 List comprehension

Input syntax for Haskell-like list comprehension notation. Typical example: [(x,y). x ← xs, y ← ys, x ≠ y], the list of all pairs of distinct elements from xs and ys. The syntax is as in Haskell, except that | becomes a dot (like in Isabelle’s set comprehension): [e. x ← xs, ...] rather than [e| x ← xs, ...]. The qualifiers after the dot are
generators p ← xs, where p is a pattern and xs an expression of list type, or
guards b, where b is a boolean expression.
Just like in Haskell, list comprehension is just a shorthand. To avoid misunderstandings, the translation into desugared form is not reversed upon output. Note that the translation of \([e, \ x \leftarrow \ xs]\) is optimized to \(\text{map } (\lambda x. \ e) \ xs\).

It is easy to write short list comprehensions which stand for complex expressions. During proofs, they may become unreadable (and mangled). In such cases it can be advisable to introduce separate definitions for the list comprehensions in question.

**nonterminal** \(\text{lc-qual and lc-quals}\)

**syntax**

- \(\text{-listcompr} :: 'a \Rightarrow \text{lc-qual} \Rightarrow \text{lc-quals} \Rightarrow 'a \text{ list} ([- . --])\)
- \(\text{-lc-gen} :: 'a \Rightarrow 'a \text{ list} \Rightarrow \text{lc-qual} (- \leftarrow -)\)
- \(\text{-lc-test} :: \text{bool} \Rightarrow \text{lc-qual} (-)\)
- \(\text{-lc-end} :: \text{lc-quals} ([])\)
- \(\text{-lc-quals} :: \text{lc-qual} \Rightarrow \text{lc-quals} \Rightarrow \text{lc-quals} (, - -)\)

**syntax (ASCII)**

- \(\text{-lc-gen} :: 'a \Rightarrow 'a \text{ list} \Rightarrow \text{lc-qual} (< - -)\)

**parse-translation**

\[
\begin{align*}
\text{fun single} \ x &= \text{ConsC} \ \& \ x \ \& \ \text{NilC}; \\
\text{fun pat-tr} \ \text{ctxt} \ p \ e \ \text{opti} &= (* \% x. \ \text{case} \ x \ \text{of} \ p \Rightarrow e \mid - \Rightarrow [] * ) \\
& \quad \text{let} \ (\ast \ \text{FIXME proper name context}!! \ *) \\
& \quad \quad \text{val} \ x = \\
& \quad \quad \quad \text{Free} \ (\text{singleton} \ (\text{Name.\ variant-list} \ (\text{fold} \ \text{Term.\ add-free-names} \ [p, \ e] []))) \ x, \\
& \quad \quad \quad \text{dummyT}); \\
& \quad \quad \text{val} \ e = \text{if opti then single} \ e \ \text{else} \ e; \\
& \quad \quad \text{val} \ \text{case1} = \text{Syntax.\ const} \ \text{syntax-const} \ (-\text{case1}) \ \& \ p \ \& \ e; \\
& \quad \quad \text{val} \ \text{case2} = \\
& \quad \quad \quad \text{Syntax.\ const} \ \text{syntax-const} \ (-\text{case1}) \ \& \ \text{dummyC} \ \& \ \text{NilC}; \\
& \quad \quad \text{val} \ \text{cs} = \text{Syntax.\ const} \ \text{syntax-const} \ (-\text{case2}) \ \& \ \text{case1} \ \& \ \text{case2}; \\
& \quad \quad \text{in} \ \text{Syntax-Trans.\ abs-tr} \ [x, \ \text{Case-Translation.\ case-tr \ false} \ \text{ctxt} \ [x, \ cs]] \ \text{end}; \\
\end{align*}
\]

\[
\begin{align*}
\text{fun pair-pat-tr} \ (x \ \text{as Free} \ -) \ e &= \text{Syntax-Trans.\ abs-tr} \ [x, \ e] \\
& \mid \ \text{pair-pat-tr} \ (- \ \& \ p1 \ \& \ p2) \ e = \\
& \quad \text{Syntax.\ const} \ \text{const-syntx} \ \text{case-prod} \ \& \ \text{pair-pat-tr} \ p1 \ \text{pair-pat-tr} \ p2 \ e \\
\end{align*}
\]
fun pair-pat ctxt (Const (const-syntax:Pair,-) $ s $ t) =  
  pair-pat ctxt s andalso pair-pat ctxt t  
| pair-pat ctxt (Free (s,-)) =  
  let  
    val thy = Proof-Context.theory-of ctxt;  
    val s' = Proof-Context.intern-const ctxt s;  
    in not (Sign.declared-const thy s') end  
| pair-pat - t = (t = dummyC);  

fun abs-tr ctxt p e opti =  
  let val p = Term-Position.strip-positions p  
  in if pair-pat ctxt p  
    then (pair-pat-tr p e, true)  
    else (pat-tr ctxt p e opti, false)  
  end  

fun lc-tr ctxt [e, Const (syntax-const (-lc-test), -) $ b, qs] =  
  let val res =  
    (case qs of  
      Const (syntax-const (-lc-end), -) => single e  
    | Const (syntax-const (-lc-quals), -) $ q $ qs => lc-tr ctxt [e, q, qs]);  
  in ifC $ b $ res $ NilC end  
| lc-tr ctxt [e, Const (syntax-const (-lc-gen), -) $ p $ es,  
  Const (syntax-const (-lc-end), -)] =  
  (case abs-tr ctxt p e true of  
    (f, true) => mapC $ f $ es  
  | (f, false) => concatC $ (mapC $ f $ es))  
| lc-tr ctxt [e, Const (syntax-const (-lc-quals), -) $ p $ es,  
  Const (syntax-const (-lc-gen), -) $ q $ qs] =  
  let val e' = lc-tr ctxt [e, q, qs];  
  in concatC $ (mapC $ (fst (abs-tr ctxt p e' false)) $ es) end;  

in [(syntax-const (-listcompr), lc-tr)] end  

ML-val (  
let  
  val read = Syntax.read-term context o Syntax.implode-input;  
  fun check s1 s2 =  
    read s1 aconv read s2 orelse  
    error (Check failed:  
      quote (#1 (Input.source-content s1))  
      Position.here-list [Input.pos-of s1,  
      Input.pos-of s2]);
check \([(x,y,z), b] \text{ if } b \text{ then } [(x, y, z)] \text{ else } []\);
check \([(x,y,z), (x,-y)\leftarrow xs]\): map \((\lambda(x,-y)). (x, y, z))\) xs;
check \([(c e y, (x,-)\leftarrow xs, y\leftarrow ys)\): concat \((\lambda(x,-). \text{ map } (\lambda y. e x y)\) ys)\) xs);
check \([(x,y,z), x<a, x>b]\): if \(x < a\) then if \(b < x\) then \([(x, y, z)]\) else [] else [];
check \([(x,y,z), x\leftarrow xs, x>b]\): (concat \((\lambda x. \text{ if } b < x \text{ then } [(x, y, z)]\) else [])\) xs);
check \([(x,y,z), x<a, x\leftarrow xs]\): if \(x < a\) then map \((\lambda x. (x, y, z))\) xs else []; 
check \([(x,y), \text{ Cons True } x \leftarrow xs]\)
(concat \((\lambda x. \text{ case } xa \text{ of } [] \Rightarrow []\text{ | } \text{ True } # x \Rightarrow [(x, y)]\text{ | } \text{ False } # x \Rightarrow []\) xs);
check \([(x,y,z), \text{ Cons } x \leftarrow xs]\)
(concat \((\lambda xa. \text{ case } xa \text{ of } [] \Rightarrow []\text{ | } [x] \Rightarrow [(x, y, z)]\text{ | } x \# aa \# \text{ lista} \Rightarrow []\) xs);
check \([(x,y,z), x<a, x>b, x=d]\)
(if \(x < a\) then if \(b < x\) then if \(x = d\) then \([(x, y, z)]\) else [] else [] else []);
check \([(x,y,z), x<a, x>b, y\leftarrow ys]\)
(if \(x < a\) then if \(b < x\) then map \((\lambda y. (x, y, z))\) ys else [] else []);
check \([(x,y,z), x<a, (\_\_\_\_\_\_\_.)\leftarrow xs, y\leftarrow b]\)
(if \(x < a\) then concat \((\lambda(x,-). \text{ if } b < y \text{ then } [(x, y, z)]\) else [])\) xs) else [];
check \([(x,y,z), x<a, x\leftarrow xs, y\leftarrow ys]\)
(if \(x < a\) then concat \((\lambda x. \text{ map } (\lambda y. (x, y, z))\) ys)\) xs) else [];
check \([(x,y,z), x\leftarrow xs, x>b, y\leftarrow a]\)
(concat \((\lambda x. \text{ if } b < x\text{ then if } y < a \text{ then } [(x, y, z)]\) else []\) xs); 
check \([(x,y,z), x\leftarrow xs, x>b, y\leftarrow ys]\)
(concat \((\lambda x. \text{ if } b < x \text{ then map } (\lambda y. (x, y, z))\) ys\) else []\) xs);
check \([(x,y,z), x\leftarrow xs, (\_\_\_\_\_\_.)\leftarrow ys, y\leftarrow x]\)
(concat \((\lambda x. \text{ concat } (\lambda(y,-). \text{ if } x < y \text{ then } [(x, y, z)]\) else []\) ys)\) xs);
check \([(x,y,z), x\leftarrow xs, y\leftarrow ys, z\leftarrow zs]\)
(concat \((\lambda xa. \text{ concat } (\lambda y. \text{ map } (\lambda z. (x, y, z))\) zs)\) ys)\) xs);
end;

ML
{*
Simproc for rewriting list comprehensions applied to List.set to set comprehension.*
}
signature LIST-TO-SET-COMPREHENSION =
sig
  val simproc : Proof.context -> cterm -> thm option
end

structure List-to-Set-Comprehension : LIST-TO-SET-COMPREHENSION =
struct
{*
 conversion *
}
fun all-exists-conv cv ctxt ct =  
  (case Thm.term-of ct of  
    Const (const-name ⟨Ex⟩, -) $ Abs - =>  
      Conv.arg-conv (Conv.abs-conv (all-exists-conv cv o #2) ctxt) ct  
    | - => cv ctxt ct)

fun all-but-last-exists-conv cv ctxt ct =  
  (case Thm.term-of ct of  
    Const (const-name ⟨Ex⟩, -) $ Abs (-, -, Const (const-name ⟨Ex⟩, -) $ -) =>  
      Conv.arg-conv (Conv.abs-conv (all-but-last-exists-conv cv o #2) ctxt) ct  
    | - => cv ctxt ct)

fun Collect-conv cv ctxt ct =  
  (case Thm.term-of ct of  
    Const (const-name ⟨Collect⟩, -) $ Abs - => Conv.arg-conv (Conv.abs-conv cv ctxt) ct  
    | - => raise CTERM (Collect-conv, [ct]))

fun rewr-conv' th = Conv.rewr-conv (mk-meta-eq th)

fun conjunct-assoc-conv ct =  
  Conv.try-conv  
  (rewr-conv' @ {thm conj-assoc} then-conv HOLogic.conj-conv Conv.all-conv conjunct-assoc-conv) ct

fun right-hand-set-comprehension-conv conv ctxt =  
  HOLogic.Trueprop-conv (HOLogic.eq-conv Conv.all-conv  
    (Collect-conv (all-exists-conv conv o #2) ctxt))  

(* term abstraction of list comprehension patterns *)

datatype termlets = If | Case of typ * int

local

val set-Nil-I = @[lemma set [] = {x. False} by simp add: empty-def [symmetric]]
val set-singleton = @[lemma set [a] = {x. x = a} by simp]
val inst-Collect-mem-eq = @[lemma set A = {x. x ∈ set A} by simp]
val del-refl-eq = @[lemma (t = t ∧ P) ≡ P by simp]

fun mk-set T = Const (const-name ⟨set⟩, HOLogic.listT T T --› HOLogic.mk-setT T)
fun dest-set (Const (const-name ⟨set⟩, -) $ xs) = xs

fun dest-singleton-list (Const (const-name ⟨Cons⟩, -) $ t $ (Const (const-name ⟨Nil⟩, -))) = t  
  | dest-singleton-list t = raise TERM (dest-singleton-list, [t])
(*We check that one case returns a singleton list and all other cases return [], and return the index of the one singleton list case.*)

fun possible-index-of-singleton-case cases =
  let
    fun check (i, case-t) s =
      let
        val case = strip-abs-body case-t
          in
          case (Const (const-name, Nil), -) => s
          | _ => (case s of SOME NONE => SOME (SOME i) | _ => NONE)
        end
      in
        fold-index check cases (SOME NONE) | the-default NONE
      end

(*returns condition continuing term option*)

fun dest-if (Const (const-name, If, -) $ cond $ then-t $ Const (const-name, Nil, -)) =
  SOME (cond, then-t)
| dest-if - = NONE

(*returns (case-expr type index chosen-case constr-name) option*)

fun dest-case ctxt case-term =
  let
    val (case-const, args) = strip-comb case-term
    in
      case try dest-Const case-const of
        SOME (c, T) =>
          (case Ctr-Sugar.ctr-sugar-of-case ctxt c of
            SOME {ctrs, ...} =>
              (case possible-index-of-singleton-case (fst (split-last args)) of
                SOME i =>
                  let
                    val constr-names = map (fst o dest-Const) ctrs
                    val (Ts, -) = strip-type T
                    val T' = List.last Ts
                    in
                      SOME (List.last args, T', i, nth args i, nth constr-names i)
                    end
                  end
                | NONE => NONE)
            | NONE => NONE) end
    end

fun tac ctxt [] =
  resolve-tac ctxt [set-singleton] 1 ORELSE
  resolve-tac ctxt [inst-Collect-mem-eq] 1
| tac ctxt (If :: cont) =
  Splitter.split-tac ctxt @{thms if-split} 1
  THEN resolve-tac ctxt @{thms conjI} 1
  THEN resolve-tac ctxt @{thms impI} 1
  THEN Subgoal.FOCUS (fn {prems, context = cntxt, ...} =>
    CONVERSION (right-hand-set-comprehension-conv (K
      (HOLogic.conj_conv $ Conv.rewr_conv (List.last prems RS @{thm Eq-TrueI}))))
  )
THEORY “List”

Conv.all-conv
  then-conv
  rewr-conv' @{lemma (True ∧ P) = P by simp}) ctxt' 1) ctxt 1
  THEN tac ctxt cont
  THEN resolve-tac ctxt @{thsms impl} 1
  THEN Subgoal.FOCUS (fn {prems, context = ctxt', ...} =>
    CONVERSION (right-hand-set-comprehension-conv (K
      (HOLogic.conj-conv (Conv.rewr-conv (List.last prems RS @{thm Eq-False}))
        Conv.all-conv
        then-conv rewr-conv' @{lemma (False ∧ P) = False by simp})
      ctxt' )
    ) 1)
    THEN resolve-tac ctxt
    THEN tac ctxt (Case (T, i) :: cont) =
      let
        val SOME {injects, distincts, case-thms, split, ...} =
          Ctr-Sugar.ctr-sugar-of ctxt (fst (dest_Type T))
        in (* do case distinction *)
          Splitter.split-tac ctxt [split] 1
          THEN EVERY (map-index (fn (i', -) =>
            (if i' < length case-thms − 1 then resolve-tac ctxt @{thsms conjI} 1 else
              all-tac)
            )
          )
          THEN REPEAT-DETERM (resolve-tac ctxt @{thsms allI} 1)
          THEN resolve-tac ctxt @{thsms impl} 1
          THEN (if i' = i then
            (* continue recursively *)
            Subgoal.FOCUS (fn {prems, context = ctxt', ...} =>
              CONVERSION (Thm.eta-conversion then-conv right-hand-set-comprehension-conv
                (K
                  ((HOLogic.conj-conv
                    (HOLogic.eq-conv Conv.all-conv (rewr-conv' (List.last prems)))
                  )
                )
              )
            )
          )
          THEN (Conv.try-conv (Conv.rewr-conv (map mk-meta-eq injects))))
          Conv.all-conv
          then-conv (Conv.try-conv (Conv.rewr-conv del-refl-eq))
          then-conv conjunct-assoc-conv ctxt' )
          then-conv
          (HOLogic.Trueprop-conv
            (HOLogic.eq-conv Conv.all-conv (Collect-conv (fn (-, ctxt'') =>
              Conv.repeat-conv
              (all-but-last-exists-conv
                (K (rewr-conv'
                  @[lemma (∃x. x = t ∧ P x) = P t by simp})) ctxt''))
              )))
          then-conv
          ) )
  1) ctxt 1
  THEN tac ctxt cont
  else
    Subgoal.FOCUS (fn {prems, context = ctxt', ...} =>
      CONVERSION
      (right-hand-set-comprehension-conv (K
(HOLogic.conj-conv
  (HOLogic.eq-conv Conv.all-conv
    (rewr-conv' (List.last prems))) then-conv
    (Conv.rewrs-conv (map (fn th => th RS @{thm Eq-FalseI})))
  distincts)))
Conv.all-conv then-conv
  (rewr-conv' @{lemma (False ∧ P) = False by simp}) ctxt'
then-conv
HOLogic.Trueprop-conv
  (Collect-conv (fn (-, ctxt') =>
    Conv.repeat-conv
    (Conv.bottom-conv
      (K (rewr-conv' @{lemma (∃x. P) = P by simp}) ctxt''))
    ctxt')) 1) ctxt 1
THEN resolve-tac ctxt [set-Nil-I 1]) case-thms)
end

fun simproc ctxt redex =
  let
    fun make-inner-eqs bound-vs Tis eqs t =
      (case dest-case ctxt t of
        SOME (x, T, i, cont, constr-name) =>
          let
            val (vs, body) = strip-abs (Envir.eta-long (map snd bound-vs) cont)
            val x' = incr-boundvars (length vs) x
            val eqs' = map (incr-boundvars (length vs)) eqs
            val constr-t =
              list-comb
              (Const (constr-name, map snd vs ----> T), map Bound (((length vs) - 1) downto 0))
            val constr-eq = Const (const-name'HOL.eq, T ----> T ----> typ<bool>
              (Const constr-t $ x')
            in
              make-inner-eqs (rev vs @ bound-vs) (Case (T, i) :: Tis) (constr-eq :: eqs') body
            end
          | NONE =>
            (case dest-if t of
              SOME (condition, cont) => make-inner-eqs bound-vs (If :: Tis) (condition :: eqs) cont
            | NONE =>
              if null eqs then NONE (*no rewriting, nothing to be done*)
              else
                let
                  val Type (type-name list, [rT]) = fastype-of1 (map snd bound-vs, t)
                  val pat-eq =
                in
                  make-inner-eqs (rev vs @ bound-vs) (Case (T, i) :: Tis) (constr-eq :: eqs') body
                end
            end)
        | NONE =>
          if null eqs then NONE (*no rewriting, nothing to be done*)
          else
            let
              val Type (type-name list, [rT]) = fastype-of1 (map snd bound-vs, t)
              val pat-eq =
            in
              make-inner-eqs (rev vs @ bound-vs) (Case (T, i) :: Tis) (constr-eq :: eqs') body
            end
          end
      end
    in
      make-inner-eqs bound-vs Tis eqs redex
    end
in
fun simproc ctxt redex =
  let
    fun make-inner-eqs bound-vs Tis eqs t =
      (case dest-case ctxt t of
        SOME (x, T, i, cont, constr-name) =>
          let
            val (vs, body) = strip-abs (Envir.eta-long (map snd bound-vs) cont)
            val x' = incr-boundvars (length vs) x
            val eqs' = map (incr-boundvars (length vs)) eqs
            val constr-t =
              list-comb
              (Const (constr-name, map snd vs ----> T), map Bound (((length vs) - 1) downto 0))
            val constr-eq = Const (const-name'HOL.eq, T ----> T ----> typ<bool>
              (Const constr-t $ x')
            in
              make-inner-eqs (rev vs @ bound-vs) (Case (T, i) :: Tis) (constr-eq :: eqs') body
            end
          | NONE =>
            (case dest-if t of
              SOME (condition, cont) => make-inner-eqs bound-vs (If :: Tis) (condition :: eqs) cont
            | NONE =>
              if null eqs then NONE (*no rewriting, nothing to be done*)
              else
                let
                  val Type (type-name list, [rT]) = fastype-of1 (map snd bound-vs, t)
                  val pat-eq =
                in
                  make-inner-eqs (rev vs @ bound-vs) (Case (T, i) :: Tis) (constr-eq :: eqs') body
                end
            end)
        | NONE =>
          if null eqs then NONE (*no rewriting, nothing to be done*)
          else
            let
              val Type (type-name list, [rT]) = fastype-of1 (map snd bound-vs, t)
              val pat-eq =
            in
              make-inner-eqs (rev vs @ bound-vs) (Case (T, i) :: Tis) (constr-eq :: eqs') body
            end
          end
      end
    in
      make-inner-eqs bound-vs Tis eqs redex
    end
in
THEORY "List"

```plaintext
(case try dest-singleton-list t of
  SOME t' =>
    Const (const-name: HOL.eq., rT --> rT --> typ: bool) $ Bound (length bound-vs) $ t'
  | NONE =>
    Const (const-name: Set.member., rT --> HOLogic.mk-setT $ Bound (length bound-vs))
  | SOME _ = Const (const-name: bool, rT --> rT --> typ: bool) $ Bound (length bound-vs) $ t

val reverse-bounds = curry subst-bounds (((map Bound ((length bound-vs - 1) downto 0)) @ [Bound (length bound-vs)]))
val eqs' = map reverse-bounds eqs
val pat-eq' = reverse-bounds pat-eq
val inner-t =
  fold (fn (-., T) => fn t => HOLogic.exists-const T $ absdummy T)
  (rev bound-vs) 
  (fold (curry HOLogic.mk-conj) eqs' pat-eq')
val lhs = Thm.term-of redex
val rhs = HOLogic.mk-Collect (x, rT, inner-t)
val rewrite-rule-t = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs))
in SOME
  ((Goal.prove ctxt [] [] rewrite-rule-t
    (fn {context = ctxt', ...} => tac ctxt' (rev Tis))) RS @{thm eq-reflection})
end)
```

```
simproc-setup list-to-set-comprehension (set xs) =
  (K List-to-Set-Comprehension.simproc)
```

```
code-datatype set coset
hide-const (open) coset
```

66.1.2 [] and (#)

lemma not-Cons-self [simp]:
  xs \not= x # xs
by (induct xs) auto

lemma not-Cons-self2 [simp]: x \not= xs
by (rule not-Cons-self [symmetric])
```
lemma neq-Nil-conv: \((xs \neq [])\) = \((\exists y. \, xs = y \# ys)\)
by (induct xs) auto

lemma tl-Nil: \(tl \, xs = [] \leftrightarrow xs = [] \lor (\exists x. \, xs = [x])\)
by (cases xs) auto

lemma Nil-tl: \([\,] = tl \, xs \leftrightarrow \, xs = [\,] \lor (\exists x. \, xs = [x])\)
by (cases xs) auto

lemma length-induct:
\((\forall xs. \, \forall ys. \, length \, ys < length \, xs \rightarrow P \, ys \implies P \, xs) \implies P \, xs\)
by (fact measure-induct)

lemma induct-list012:
\([(\,]); \forall x. \, P \, [x]; \forall x \, y \, zs. \, P \, zs; P \, (y \# zs) ] \implies P \, (x \# y \# zs)] \implies P \, xs
by induction-schema (pat-completeness, lexicographic-order)

lemma list-nonempty-induct [consumes 1, case-names single cons]:
\([\,] \neq []; \forall x. \, P \, [x]; \forall x \, xs. \, xs \neq [] \implies P \, xs \implies P \, (x \# xs)] \implies P \, xs
by (induction xs rule: induct-list012) auto

lemma inj-split-Cons: inj-on \((\lambda (xs, n). \, n\#xs)\) X
by (auto intro!: inj-onI)

lemma inj-on-Cons1 [simp]: inj-on \((\#)\) A
by (simp add: inj-on-def)

66.1.3  length

Needs to come before @ because of theorem append-eq-append-conv.

lemma length-append [simp]: length \((xs @ ys)\) = length \(xs\) + length \(ys\)
by (induct xs) auto

lemma length-map [simp]: length \((map f \, xs)\) = length \(xs\)
by (induct xs) auto

lemma length-rev [simp]: length \((rev \, xs)\) = length \(xs\)
by (induct xs) auto

lemma length-tl [simp]: length \((tl \, xs)\) = length \(xs\) \(-\) 1
by (cases xs) auto

lemma length-0-conv [iff]: \((\text{length } xs = 0) = (xs = [])\)
by (induct xs) auto

lemma length-greater-0-conv [iff]: \((0 < \text{length } xs) = (xs \neq [])\)
by (induct xs) auto
theory "List"

lemma length-pos-if-in-set: \( x \in \text{set} \; xs \Rightarrow \text{length} \; xs > 0 \)
by auto

lemma length-Suc-conv:
\( (\text{length} \; xs = \text{Suc} \; n) = (\exists \; ys. \; xs = y \# \; ys \land \text{length} \; ys = n) \)
by (induct xs) auto

lemma Suc-length-conv:
\( (\text{Suc} \; n = \text{length} \; xs) = (\exists \; y \; ys. \; xs = y \# \; ys \land \text{length} \; ys = n) \)
by (induct xs; simp; blast)

lemma impossible-Cons:
\( \text{length} \; xs \leq \text{length} \; ys \Rightarrow xs = x \# ys = \text{False} \)
by (induct xs) auto

lemma list-induct2 [consumes 1, case-names Nil Cons]:
\( \text{length} \; xs = \text{length} \; ys \Rightarrow P \[\] \[\] \Rightarrow \\
(\forall \; xs \; y \; ys. \; \text{length} \; xs = \text{length} \; ys \Rightarrow P \; xs \; ys \Rightarrow P \; (x\#xs) \; (y\#ys)) \Rightarrow P \; zs \; ys \)
proof (induct xs arbitrary: ys)
  case (Cons x xs ys) then show \(?case\) by (cases ys) simp-all
qed simp

lemma list-induct3 [consumes 2, case-names Nil Cons]:
\( \text{length} \; xs = \text{length} \; ys \Rightarrow \text{length} \; ys = \text{length} \; zs \Rightarrow P \[\] \[\] \Rightarrow \\
(\forall \; xs \; y \; ys \; z \; zs. \; \text{length} \; xs = \text{length} \; ys \Rightarrow P \; xs \; ys \; zs \Rightarrow P \; (x\#xs) \; (y\#ys) \; (z\#zs)) \Rightarrow P \; zs \; ys \; zs \)
proof (induct xs arbitrary: ys zs)
  case Nil then show \(?case\) by simp
next
  case (Cons x xs ys zs) then show \(?case\) by (cases ys, simp-all)
    (cases zs, simp-all)
qed

lemma list-induct4 [consumes 3, case-names Nil Cons]:
\( \text{length} \; xs = \text{length} \; ys = \text{length} \; zs = \text{length} \; ws \Rightarrow P \[\] \[\] \[\] \Rightarrow \\
(\forall \; xs \; y \; ys \; z \; zs \; w \; ws. \; \text{length} \; xs = \text{length} \; ys \Rightarrow \text{length} \; ys = \text{length} \; zs \Rightarrow \text{length} \; zs = \text{length} \; ws \Rightarrow P \; ys \; zs \; ws \Rightarrow P \; (x\#xs) \; (y\#ys) \; (z\#zs) \; (w\#ws)) \Rightarrow P \; xs \; ys \; zs \; ws \)
proof (induct xs arbitrary: ys zs ws)
  case Nil then show \(?case\) by simp
next
  case (Cons x xs ys zs ws) then show \(?case\) by ((cases ys, simp-all), (cases zs,simp-all)) (cases ws, simp-all)
qed
lemma list-induct2':
\[
\begin{align*}
&[ P [] []; \\
&\forall x. P (x#xs) []; \\
&\forall y. P [] (y#ys); \\
&\forall x \; y \; xs \; ys. P xs ys \implies P (x#xs) (y#ys) ] \\
\implies P xs ys \\
\end{align*}
\]
by (induct xs arbitrary: ys) (case-tac x, auto)+

lemma list-all2-iff:
\[
\text{list-all2 } P \; xs \; ys \iff \text{length } xs = \text{length } ys \land (\forall (x, y) \in \text{set (zip } xs \; ys). \; P \; x \; y)
\]
by (induct xs ys rule: list-induct2') auto

lemma neq-if-length-neq:
\[
\text{length } xs \neq \text{length } ys \implies (xs = ys) = False
\]
by (rule Eq-FalseI) auto

simproc-setup list-neq ((xs::'a list) = ys) = (
(*
Reduces xs=ys to False if xs and ys cannot be of the same length.
This is the case if the atomic sublists of one are a submultiset
of those of the other list and there are fewer Cons's in one than the other.
*)

let

fun len (Const(const-name⟨Nil⟩)) acc = acc
| len (Const(const-name⟨Cons⟩)) ($) ($) ts, n) = len xs (ts, n+1)
| len (Const(const-name⟨append⟩)) ($) ($) xs, ys) acc = len xs (len ys acc)
| len (Const(const-name⟨rev⟩)) ($) xs) acc = len xs acc
| len (Const(const-name⟨map⟩)) ($) ($) xs) acc = len xs acc
| len t (ts, n) = (t:ts, n);

val ss = simpset-of context;

fun list-neq ctxt ct =
let
val (Const(\_,eqT) $ lhs $ rhs) = Thm.term-of ct;
val (ls,m) = len lhs (\[],0) and (rs,n) = len rhs (\[],0);
fun prove-neq() =
let
val Type(\_<,listT::_) = eqT;
val size = HOLogic.size-const listT;
val eq-len = HOLogic.mk-eq (size $ lhs, size $ rhs);
val neq-len = HOLogic.mk-Trueprop (HOLogic.Not $ eq-len);
val thm = Goal.prove ctxt [] [] neq-len
  (K (simp-tac (put-simpset ss ctxt) 1));
in SOME (thm RS @{thm neq-if-length-neq}) end
in
if m < n andalso submultiset (op aconv) (ls,rs) orelse
THEORY "List"

\[ n < m \text{ andalso submultiset } (op \text{ aconv}) (rs,ls) \]
then prove-neq() else NONE
end;
in K list-neq end

66.1.4  \texttt{@} – append

\textbf{global-interpretation} append: monoid append Nil

\textbf{proof}
\begin{itemize}
  \item fix \( x\)s \( y\)s \( z\)s :: 'a list
  \item show \( (x\)s \texttt{@} \( y\)s \texttt{@} \( z\)s = \( x\)s \texttt{@} (\( y\)s \texttt{@} \( z\)s)
    \begin{itemize}
      \item by (induct \( z\)s) simp-all
    \end{itemize}
  \item show \( x\)s \texttt{@} [] = \( x\)s
    \begin{itemize}
      \item by (induct \( x\)s) simp-all
    \end{itemize}
\end{itemize}
\textbf{qed simp}

\textbf{lemma} append-assoc [simp]: \( (x\)s \texttt{@} \( y\)s \texttt{@} \( z\)s = \( x\)s \texttt{@} (\( y\)s \texttt{@} \( z\)s)
\begin{itemize}
  \item by (fact append.assoc)
\end{itemize}

\textbf{lemma} append-Nil2: \( x\)s \texttt{@} [] = \( x\)s
\begin{itemize}
  \item by (fact append.right-neutral)
\end{itemize}

\textbf{lemma} append-is-Nil-conv [iff]: \( (x\)s \texttt{@} \( y\)s = [] \) = \( (x\)s = [] \land \( y\)s = []\)
\begin{itemize}
  \item by (induct \( x\)s) auto
\end{itemize}

\textbf{lemma} Nil-is-append-conv [iff]: \( ([] = x\)s \texttt{@} \( y\)s) = \( (x\)s = [] \land \( y\)s = []\)
\begin{itemize}
  \item by (induct \( x\)s) auto
\end{itemize}

\textbf{lemma} append-self-conv [iff]: \( (x\)s = x\)s \texttt{@} \( y\)s = \( y\)s
\begin{itemize}
  \item by (induct \( x\)s) auto
\end{itemize}

\textbf{lemma} self-append-conv [iff]: \( (x\)s = x\)s \texttt{@} \( y\)s = \( y\)s
\begin{itemize}
  \item by (induct \( x\)s) auto
\end{itemize}

\textbf{lemma} append-eq-append-conv [simp]:
\begin{itemize}
  \item length \( x\)s = length \( y\)s \lor length \( u\)s = length \( v\)s
  \item ==> (\( x\)s@\( u\)s = \( y\)s@\( v\)s) = (\( x\)s=\( y\)s \land \( u\)s=\( v\)s)
  \begin{itemize}
    \item by (induct \( x\)s arbitrary: \( y\)s; case-tac \( y\)s; force)
  \end{itemize}
\end{itemize}

\textbf{lemma} append-eq-append-conv2: \( (x\)s \texttt{@} \( y\)s = \( z\)s \texttt{@} \( t\)s) =
\begin{itemize}
  \item (\( \exists \)us. \( x\)s = \( z\)s \texttt{@} \( u\)s \land \( u\)s @ \( y\)s = \( t\)s \lor \( z\)s @ \( u\)s = \( z\)s \land \( y\)s = \( u\)s \texttt{@} \( t\)s)
\end{itemize}
\textbf{proof}
\begin{itemize}
  \item (induct \( x\)s arbitrary: \( y\)s \texttt{@} \( t\)s)
  \item case \texttt{(Cons x zs)}
  \item then show \texttt{?case}
    \begin{itemize}
      \item by (cases \( z\)s) auto
    \end{itemize}
  \item qed fastforce
\end{itemize}

\textbf{lemma} same-append-eq [iff, induct-simp]: \( (x\)s \texttt{@} \( y\)s = x\)s \texttt{@} \( z\)s = \( y\)s = \( z\)s
\begin{itemize}
  \item by (induct \( x\)s) simp-all
\end{itemize}

\textbf{proof}
\begin{itemize}
  \item by (cases \( z\)s) auto
\end{itemize}
\textbf{qed fastforce}
by simp

lemma append1-eq-conv [iff]: \( (xs @ [x] = ys @ [y]) = (xs = ys \land x = y) \)
by simp

lemma append-same-eq [iff, induct-simp]: \( (ys @ xs = zs @ xs) = (ys = zs) \)
by simp

lemma append-self-conv2 [iff]: \( (xs @ ys = ys) = (xs = []) \)
using append-same-eq [of - -] by auto

lemma self-append-conv2 [iff]: \( (ys = xs @ ys) = (xs = []) \)
using append-same-eq [of []] by auto

lemma hd-Cons-tl: \( xs \neq [] \implies hd xs = tl xs \)
by (fact list.collapse)

lemma hd-append: \( hd (xs @ ys) = (if xs = [] then hd ys else hd xs) \)
by (induct xs) auto

lemma hd-append2 [simp]: \( xs \neq [] \implies hd (xs @ ys) = hd xs \)
by (simp add: hd-append split: list.split)

lemma tl-append: \( tl (xs @ ys) = (case xs of [] \Rightarrow tl ys | z#zs \Rightarrow zs @ ys) \)
by (simp split: list.split)

lemma tl-append2 [simp]: \( xs \neq [] \implies tl (xs @ ys) = tl xs @ ys \)
by (simp add: tl-append split: list.split)

lemma Cons-eq-append-conv: \( x#xs = ys@zs = \)
\( (ys = [] \land x#xs = zs \lor (\exists ys'. x#ys' = ys \land xs = ys'@zs)) \)
by (cases ys) auto

lemma append-eq-Cons-conv: \( (ys@zs = x#xs) = \)
\( (ys = [] \land zs = x#xs \lor (\exists ys'. ys = x#ys' \land ys'@zs = xs)) \)
by (cases ys) auto

lemma longest-common-prefix:
\( \exists ps xs' ys'. xs = ps @ xs' \land ys = ps @ ys' \land (xs' = [] \lor ys' = [] \lor hd xs' \neq hd ys') \)
by (induct xs ys rule: list-induct2')
(blast, blast, blast,
metis (no-types, hide-lams) append-Cons append-Nil list.sel(1))

Trivial rules for solving @-equations automatically.

lemma eq-Nil-appendI: \( xs = ys \implies xs = [] @ ys \)
by simp
**THEORY “List”**

**lemma** Cons-eq-appendI:
\[ | x \# xs1 = ys; xs = xs1 @ zs | \implies x \# xs = ys @ zs \]
by (drule sym) simp

**lemma** append-eq-appendI:
\[ | xs @ xs1 = zs; ys = xs1 @ us | \implies xs @ ys = zs @ us \]
by (drule sym) simp

Simplification procedure for all list equalities. Currently only tries to rearrange @ to see if - both lists end in a singleton list, - or both lists end in the same list.

**simpproc-setup** list-eq ((xs::'a list) = ys) = |
  let
  fun last (cons as Const (const-name ⟨Cons⟩, -) $ - $ xs) =
    (case xs of Const (const-name ⟨Nil⟩, -) => cons | - => last xs)
  | last (Const (const-name ⟨append⟩, -) $ - $ ys) = last ys
  | last t = t;

  fun list1 (Const (const-name ⟨Cons⟩, -) $ - $ Const (const-name ⟨Nil⟩, fastype-of xs)) =
    true
  | list1 - = false;

  fun butlast ((cons as Const (const-name ⟨Cons⟩, -) $ x) $ xs) =
    (case xs of Const (const-name ⟨Nil⟩, -) => xs | - => cons $ butlast xs)
  | butlast ((app as Const (const-name ⟨append⟩, -) $ xs) $ ys) = app $ butlast ys
  | butlast xs = Const (const-name ⟨Nil⟩, fastype-of xs);

  val rearr-ss =
    simpset-of (put-simpset HOL-basic-ss context
      addsimps @{thm append-assoc}, @{thm append-Nil}, @{thm append-Cons});

  fun list-eq ctxt (F as (eq as Const (cdot, eqT)) $ lhs $ rhs) =
    let
    val lastl = last lhs and lastr = last rhs;
    fun rearr conv =
      let
      val lhsi = butlast lhs and rhsi = butlast rhs;
      val Type (_,listT::-) = eqT
      val appT = [listT, listT] ---> listT
      val app = Const (const-name ⟨append⟩, appT)
      val F2 = eq $ (app$lns$lastl) $ (app$rhss$lastr)
      val eq = HOLogic.mk_Trueprop (HOLogic.mk_eq (F,F2));
      val thm = Goal.prove ctxt [] [] eq
        (K (simp_tac (put-simpset rearr-ss ctxt) 1));
      in SOME ((conv RS (thm RS trans)) RS eq-reflection) end;
      in
      if list1 lastl andalso list1 lastr then rearr @{thm append1-eq-conv}
      else if lastl aconv lastr then rearr @{thm append-same-eq}
else NONE
  end;
  in fn => fn ctxt => fn ct => list-eq ctxt (Thm.term-of ct) end

66.1.5  map

lemma hd-map: xs ≠ [] => hd (map f xs) = f (hd xs)
by (cases xs) simp-all

lemma map-tl: map f (tl xs) = tl (map f xs)
by (cases xs) simp-all

lemma map-ext: (\x. x ∈ set xs => f x = g x) ==> map f xs = map g xs
by (induct xs) simp-all

lemma map-ident [simp]: map (\x. x) = (\x. x)
by (rule ext, induct-tac xs) auto

lemma map-append [simp]: map f (xs @ ys) = map f xs @ map f ys
by (induct xs) auto

lemma map-map [simp]: map f (map g xs) = map (f o g) xs
by (induct xs) auto

lemma map-comp-map [simp]: ((map f) o (map g)) = map(f o g)
by (rule ext) simp

lemma rev-map: rev (map f xs) = map f (rev xs)
by (induct xs) auto

lemma map-eq-cone [simp]: (map f xs = map g xs) = (\x ∈ set xs. f x = g x)
by (induct xs) auto

lemma map-cong [fundef-cong]:
x = y =⇒ (\x ∈ set ys => f x = g x) =⇒ map f xs = map g ys
by simp

lemma map-is-Nil-conv [iff]: (map f xs = []) = (xs = [])
by (cases xs) auto

lemma Nil-is-map-conv [iff]: ([] = map f xs) = (xs = [])
by (cases xs) auto

lemma map-eq-Cons-conv:
  (map f xs = y#ys) = (\exists zs. xs = z#zs ∧ f z = y ∧ map f zs = ys)
by (cases xs) auto

lemma Cons-eq-map-conv:
\[(x \# xs = \text{map } f \ ys) = (\exists z \ zs. \ ys = z \# zs \land x = f z \land xs = \text{map } f zs)\]

by (cases \(ys\) auto)

\textbf{lemmas} \(\text{map-eq-Cons-D} = \text{map-eq-Cons-cone} [\text{THEN } \text{iffD1}]\)

\textbf{lemmas} \(\text{Cons-eq-map-D} = \text{Cons-eq-map-cone} [\text{THEN } \text{iffD1}]\)

\textbf{declare} \(\text{map-eq-Cons-D} [\text{dest}]\) \(\text{Cons-eq-map-D} [\text{dest}]\)

\textbf{lemma} \(\text{ex-map-conv}\):
\[(\exists xs. \ ys = \text{map } f \ xs) = (\forall y \in \text{set } \ys. \exists x. \ y = f x)\]
by (induct \(ys\), auto simp add: \(\text{Cons-eq-map-conv}\))

\textbf{lemma} \(\text{map-eq-imp-length-eq}\):
assumes \(\text{map } f \ xs = \text{map } g \ ys\)
shows \(\text{length } xs = \text{length } ys\)
using \(\text{assms}\)
proof (induct \(ys\) arbitrary: \(xs\))
  case Nil then show \(?case\) by simp
next
  case (Cons \(y\) \(ys\)) then obtain \(z \ zs\) where \(xs: \ xs = z \# zs\) by auto
  from \(\text{Cons } \text{xs}\) have \(\text{map } f zs = \text{map } g \ ys\) by simp
  with \(\text{Cons}\) have \(\text{length } zs = \text{length } \ys\) by blast
  with \(\text{xs}\) show \(?case\) by simp
qed

\textbf{lemma} \(\text{map-inj-on}\):
assumes \(\text{map: } \text{map } f \ xs = \text{map } f \ ys\) and \(\text{inj: } \text{inj-on } f \ (\text{set } \text{xs } \cup \text{set } \ys)\)
shows \(\text{xs} = \text{ys}\)
using \(\text{map-eq-imp-length-eq [OF } \text{map} \text{ assms}\)
proof (induct rule: list-induct2)
  case (Cons \(x\) \(xs\) \(y\) \(ys\))
  then show \(?case\)
    by (auto intro: sym)
qed auto

\textbf{lemma} \(\text{inj-on-map-eq-map}\):
\(\text{inj-on } f \ (\text{set } \text{xs } \cup \text{set } \ys) \implies (\text{map } f \ xs = \text{map } f \ ys) = (\text{xs} = \text{ys})\)
by (blast dest: map-inj-on)

\textbf{lemma} \(\text{map-injective}\):
\(\text{map } f \ xs = \text{map } f \ ys \implies \text{inj } f \implies \text{xs} = \text{ys}\)
by (induct \(ys\) arbitrary: \(xs\) (auto dest!: injD)

\textbf{lemma} \(\text{inj-map-eq-map[simp]}: \text{inj } f \implies (\text{map } f \ xs = \text{map } f \ ys) = (\text{xs} = \text{ys})\)
by (blast dest: map-injective)

\textbf{lemma} \(\text{inj-mapI: } \text{inj } f \implies \text{inj } (\text{map } f)\)
by (iprover dest: map-injective injD intro: injI)

\textbf{lemma} \(\text{inj-mapD: } \text{inj } (\text{map } f) \implies \text{inj } f\)
by (metis (no-types, hide-lams) injI list.inject list.simps(9) the-inv-f-f)

lemma inj-map[iff]: inj (map f) = inj f
by (blast dest: inj-mapD intro: inj-mapI)

lemma inj-on-mapI: inj-on f (Union(set ' A)) ==> inj-on (map f) A
by (blast intro: inj-onI dest: inj-onD map-inj-on)

lemma map-idI: (\x. \x \in set xs \implies f x = x) ==> map f xs = xs
by (induct xs, auto)

lemma map-fun-upd [simp]: y \notin set xs \implies map (f(y:=v)) xs = map f xs
by (induct xs) auto

lemma map-fst-zip[simp]:
  length xs = length ys \implies map fst (zip xs ys) = xs
by (induct rule: list-induct2, simp-all)

lemma map-snd-zip[simp]:
  length xs = length ys \implies map snd (zip xs ys) = ys
by (induct rule: list-induct2, simp-all)

lemma map2-map-map: map2 h (map f xs) (map g xs) = map (\x. h (f x) (g x)) xs
by (induction xs) (auto)

functor map: map
by (simp-all add: id-def)

declare map.id [simp]

66.1.6  rev

lemma rev-append [simp]: rev (xs @ ys) = rev ys @ rev xs
by (induct xs) auto

lemma rev-rev-ident [simp]: rev (rev xs) = xs
by (induct xs) auto

lemma rev-swap: (rev xs = ys) = (xs = rev ys)
by auto
lemma rev-is-Nil-conv [iff]: (rev xs = []) = (xs = [])
by (induct xs) auto

lemma Nil-is-rev-conv [iff]: ([] = rev xs) = (xs = [])
by (induct xs) auto

lemma rev-singleton-conv [simp]: (rev xs = [x]) = (xs = [x])
by (cases xs) auto

lemma singleton-rev-conv [simp]: ([x] = rev xs) = (xs = [x])
by (cases xs) auto

lemma rev-is-rev-conv [iff]: (rev xs = rev ys) = (xs = ys)
proof (induct xs arbitrary: ys)
  case Nil
  then show ?case by force
next
  case Cons
  then show ?case by (cases ys) auto
qed

lemma inj-on-rev [iff]: inj-on rev A
by (simp add: inj-on-def)

lemma rev-induct [case-names Nil snoc]:
  [| P [] |] !!x xs. P xs ===> P (xs @ [x]) [] ===> P xs
apply(simplesubst rev-rev-ident[symmetric])
apply(rule_tac list = rev xs in list.induct, simp-all)
done

lemma rev-exhaust [case-names Nil snoc]:
  (xs = [] ==> P) ==> (!xs y. xs = ys @ [y] ==> P) ==> P
by (induct xs rule: rev-induct) auto

lemmas rev-cases = rev-exhaust

lemma rev-nonempty-induct [consumes 1, case-names single snoc]:
assumes xs ≠ []
and single: !!x. P [x]
and snoc': !!xs. xs ≠ [] ==> P xs ==> P (xs@[x])
shows P xs
using (xs ≠ []): proof (induct xs rule: rev-induct)
case (snoc x xs) then show ?case
  proof (cases xs)
    case Nil thus ?thesis by (simp add: single)
  next
    case Cons with snoc show ?thesis by (fastforce intro!: snoc')
qed
qed simp

lemma rev-eq-Cons-iff [iff]: (rev xs = y#ys) = (xs = rev ys @ [y])
by (rule rev-cases[of xs]) auto

66.1.7 set
declare list.set[code-post] — pretty output

lemma finite-set [iff]: finite (set xs)
by (induct xs) auto

lemma set-append [simp]: set (xs @ ys) = (set xs ∪ set ys)
by (induct xs) auto

lemma hd-in-set [simp]: xs ≠ [] =⇒ hd xs ∈ set xs
by (cases xs) auto

lemma set-subset-Cons: set xs ⊆ set (x # xs)
by auto

lemma set-Consd: y ∈ set (x # xs) =⇒ y=x ∧ y ∈ set xs
by auto

lemma set-empty [iff]: (set xs = {}) = (xs = [])
by (induct xs) auto

lemma set-empty2 [iff]: ({} = set xs) = (xs = [])
by (induct xs) auto

lemma set-rev [simp]: set (rev xs) = set xs
by (induct xs) auto

lemma set-map [simp]: set (map f xs) = f(set xs)
by (induct xs) auto

lemma set-filter [simp]: set (filter P xs) = {x. x ∈ set xs ∧ P x}
by (induct xs) auto

lemma set-upt [simp]: set[i..<j] = {i..<j}
by (induct j) auto

lemma split-list: x ∈ set xs =⇒ ∃ ys zs. xs = ys @ x # zs
proof (induct xs)
case Nil thus ?case by simp
next
case Cons thus ?case by (auto intro: Cons-eq-appendI)
qed
lemma in-set-conv-decomp: $x \in \text{set } xs \iff (\exists y s z s. x = y @ x \# z s)$
  by (auto elim: split-list)

lemma split-list-first: $x \in \text{set } xs \implies (\exists y s z s. x = y @ x \# z s \land x \notin \text{set } y s)$
proof (induct xs)
  case Nil thus ?case by simp
next
case (Cons a xs)
  show ?case
  proof cases
    assume $x = a$ thus ?case using Cons by fastforce
  next
    assume $x \neq a$ thus ?case using Cons by (fastforce intro: Cons-eq-appendI)
  qed
qed

lemma in-set-conv-decomp-first:
  $(x \in \text{set } xs) = (\exists y s z s. x = y @ x \# z s \land x \notin \text{set } y s)$
by (auto dest!: split-list-first)

lemma split-list-last: $x \in \text{set } xs \implies (\exists y s z s. x = y @ x \# z s \land x \notin \text{set } z s)$
proof (induct xs rule: rev-induct)
  case Nil thus ?case by simp
next
case (snoc a xs)
  show ?case
  proof cases
    assume $x = a$ thus ?case using snoc by (auto intro!: exI)
  next
    assume $x \neq a$ thus ?case using snoc by fastforce
  qed
qed

lemma in-set-conv-decomp-last:
  $(x \in \text{set } xs) = (\exists y s z s. x = y @ x \# z s \land x \notin \text{set } z s)$
by (auto dest!: split-list-last)

lemma split-list-prop: $\exists x \in \text{set } xs. P x \implies (\exists y s z s. x = y @ x \# z s \land P x)$
proof (induct xs)
  case Nil thus ?case by simp
next
case Cons thus ?case
  by (simp add:Bex-def)(metis append-Cons append.simps(1))
qed

lemma split-list-propE:
  assumes $\exists x \in \text{set } xs. P x$
  obtains $y s z s$ where $x = y @ x \# z s$ and $P x$
using split-list-prop [OF assms] by blast

lemma split-list-first-prop:
\[ \exists x \in \text{set} \; \text{xs}. \; P \; x \implies \exists y s \; x s = y s @ x \# z s \wedge P \; x \wedge (\forall y \in \text{set} \; y s. \neg P \; y) \]
proof (induct \( z s \))
  case Nil thus \?case by simp
next
  case (Cons x xs)
  show \?case
  proof cases
    assume \( P \; x \)
    hence \( x \# x s = [ ] @ x \# x s \wedge P \; x \wedge (\forall y \in \text{set} \; [ ]. \neg P \; y) \) by simp
    thus \?thesis by fast
  next
    assume \( \neg P \; x \)
    hence \( \exists x \in \text{set} \; x s. \; P \; x \) using Cons(2) by simp
    thus \?thesis using \( \neg P \; x \) Cons(1) by (metis append-Cons set-ConsD)
  qed
qed

lemma split-list-first-propE:
assumes \( \exists x \in \text{set} \; x s. \; P \; x \)
obtains \( y s \; x \; z s \) where \( x s = y s @ x \# z s \wedge P \; x \wedge \forall y \in \text{set} \; y s. \neg P \; y \)
using split-list-first-prop [OF assms] by blast

lemma split-list-first-prop-iff:
\[ (\exists x \in \text{set} \; x s. \; P \; x) \iff (\exists y s \; x s. \; x s = y s @ x \# z s \wedge P \; x \wedge (\forall y \in \text{set} \; y s. \neg P \; y)) \]
by (rule, erule split-list-first-prop) auto

lemma split-list-last-prop:
\[ \exists x \in \text{set} \; x s. \; P \; x \implies \exists y s \; x s = y s @ x \# z s \wedge P \; x \wedge (\forall z \in \text{set} \; z s. \neg P \; z) \]
proof (induct \( x s \) rule:rev-induct)
  case Nil thus \?case by simp
next
  case (snoc x xs)
  show \?case
  proof cases
    assume \( P \; x \) thus \?thesis by (auto intro!: exI)
  next
    assume \( \neg P \; x \)
    hence \( \exists x \in \text{set} \; x s. \; P \; x \) using snoc(2) by simp
    thus \?thesis using \( \neg P \; x \) snoc(1) by fastforce
  qed
qed

lemma split-list-last-propE:
assumes $\exists x \in \text{set } xs. \ P x$

obtains $ys \ x \ zs$ where $xs = ys @ x \# zs$ and $P x$ and $\forall z \in \text{set } zs. \neg \ P z$

using split-list-last-prop [OF assms] by blast

lemma split-list-last-prop-iff:
$\exists x \in \text{set } xs. \ P x \iff \exists ys \ x \ zs. \ xs = ys @ x \# zs \land P x \land (\forall z \in \text{set } zs. \neg \ P z)$
by rule (erule split-list-last-prop, auto)

lemma finite-list:
finite A $\implies \exists xs. \text{set } xs = A$
by (erule finite-induct) (auto simp add: list.set (2)[symmetric] simp del: list.set(2))

lemma card-length:
$\text{card } (\text{set } xs) \leq \text{length } xs$
by (induct xs) (auto)

lemma set-minus-filter-out:
$\text{set } xs - \{y\} = \text{set } (\text{filter } (\lambda x. \neg (x = y)) \ xs)$
by (induct xs) auto

lemma append-Cons-eq-iff:
$[ x \notin \text{set } xs; x \notin \text{set } ys ] \implies
xs @ x \# ys = xs' @ x \# ys' \iff (xs = xs' \land ys = ys')$
by(auto simp: append-eq-Cons-conv Cons-eq-append-conv append-eq-append-conv2)

66.1.8 filter

lemma filter-append [simp]: $\text{filter } P (xs @ ys) = \text{filter } P xs @ \text{filter } P ys$
by (induct xs) auto

lemma rev-filter: $\text{rev } (\text{filter } P xs) = \text{filter } P (\text{rev } xs)$
by (induct xs) simp-all

lemma filter-filter [simp]: $\text{filter } P (\text{filter } Q xs) = \text{filter } (\lambda x. Q x \land P x) \ xs$
by (induct xs) auto

lemma length-filter-le [simp]: $\text{length } (\text{filter } P xs) \leq \text{length } xs$
by (induct xs) (auto simp add: le-SucI)

lemma sum-length-filter-compl:
$\text{length}(\text{filter } P xs) + \text{length}(\text{filter } (\lambda x. \neg P x) \ xs) = \text{length } xs$
by(induct xs) simp-all

lemma filter-True [simp]: $\forall x \in \text{set } xs. \ P x \Longrightarrow \text{filter } P xs = xs$
by (induct xs) auto

lemma filter-False [simp]: $\forall x \in \text{set } xs. \neg P x \Longrightarrow \text{filter } P xs = []$
by (induct xs) auto
lemma filter-empty-conv: (filter P xs = []) = (∀ x ∈ set xs. ¬ P x)
by (induct xs) simp-all

lemma filter-id-conv: (filter P xs = xs) = (∀ x ∈ set xs. P x)
proof (induct xs)
  case (Cons x xs)
  then show ?case
    using length-filter-le
    by (simp add: impossible-Cons)
qed auto

lemma filter-map: filter P (map f xs) = map f (filter (P ∘ f) xs)
by (induct xs) simp-all

lemma length-filter-map[simp]: length (filter P (map f xs)) = length (filter (P ∘ f) xs)
by (simp add: filter-map)

lemma filter-is-subset [simp]: set (filter P xs) ≤ set xs
by auto

lemma length-filter-less:
  [ x ∈ set xs; ¬ P x ] ⇒ length(filter P xs) < length xs
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons x xs)
  let ?S = { i. i < length xs ∧ p(xs!i) }
  have fin: finite ?S by (fast intro: bounded-nat-set-is-finite)
  show ?case (is ?l = card ?S')
  proof (cases)
    assume p x
    hence eq: ?S' = insert 0 (Suc ?S)
    by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
    show ?case (is ?l = card ?S')
    proof (cases)
      assume p x
      hence (cases)
        by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
      show ?case (is ?l = card ?S')
      proof (cases)
        assume p x
        hence eq: ?S' = insert 0 (Suc ?S)
        by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
        show ?case (is ?l = card ?S')
        proof (cases)
          assume p x
          hence eq: ?S' = insert 0 (Suc ?S)
          by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
          show ?case (is ?l = card ?S')
          proof (cases)
            assume p x
            hence eq: ?S' = insert 0 (Suc ?S)
            by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
            show ?case (is ?l = card ?S')
            proof (cases)
              assume p x
              hence eq: ?S' = insert 0 (Suc ?S)
              by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
              show ?case (is ?l = card ?S')
              proof (cases)
                assume p x
                hence eq: ?S' = insert 0 (Suc ?S)
                by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                show ?case (is ?l = card ?S')
                proof (cases)
                  assume p x
                  hence eq: ?S' = insert 0 (Suc ?S)
                  by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                  show ?case (is ?l = card ?S')
                  proof (cases)
                    assume p x
                    hence eq: ?S' = insert 0 (Suc ?S)
                    by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                    show ?case (is ?l = card ?S')
                    proof (cases)
                      assume p x
                      hence eq: ?S' = insert 0 (Suc ?S)
                      by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                      show ?case (is ?l = card ?S')
                      proof (cases)
                        assume p x
                        hence eq: ?S' = insert 0 (Suc ?S)
                        by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                        show ?case (is ?l = card ?S')
                        proof (cases)
                          assume p x
                          hence eq: ?S' = insert 0 (Suc ?S)
                          by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                          show ?case (is ?l = card ?S')
                          proof (cases)
                            assume p x
                            hence eq: ?S' = insert 0 (Suc ?S)
                            by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                            show ?case (is ?l = card ?S')
                            proof (cases)
                              assume p x
                              hence eq: ?S' = insert 0 (Suc ?S)
                              by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                              show ?case (is ?l = card ?S')
                              proof (cases)
                                assume p x
                                hence eq: ?S' = insert 0 (Suc ?S)
                                by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                show ?case (is ?l = card ?S')
                                proof (cases)
                                  assume p x
                                  hence eq: ?S' = insert 0 (Suc ?S)
                                  by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                  show ?case (is ?l = card ?S')
                                  proof (cases)
                                    assume p x
                                    hence eq: ?S' = insert 0 (Suc ?S)
                                    by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                    show ?case (is ?l = card ?S')
                                    proof (cases)
                                      assume p x
                                      hence eq: ?S' = insert 0 (Suc ?S)
                                      by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                      show ?case (is ?l = card ?S')
                                      proof (cases)
                                        assume p x
                                        hence eq: ?S' = insert 0 (Suc ?S)
                                        by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                        show ?case (is ?l = card ?S')
                                        proof (cases)
                                          assume p x
                                          hence eq: ?S' = insert 0 (Suc ?S)
                                          by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                          show ?case (is ?l = card ?S')
                                          proof (cases)
                                            assume p x
                                            hence eq: ?S' = insert 0 (Suc ?S)
                                            by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                            show ?case (is ?l = card ?S')
                                            proof (cases)
                                              assume p x
                                              hence eq: ?S' = insert 0 (Suc ?S)
                                              by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                              show ?case (is ?l = card ?S')
                                              proof (cases)
                                                assume p x
                                                hence eq: ?S' = insert 0 (Suc ?S)
                                                by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                show ?case (is ?l = card ?S')
                                                proof (cases)
                                                  assume p x
                                                  hence eq: ?S' = insert 0 (Suc ?S)
                                                  by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                  show ?case (is ?l = card ?S')
                                                  proof (cases)
                                                    assume p x
                                                    hence eq: ?S' = insert 0 (Suc ?S)
                                                    by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                    show ?case (is ?l = card ?S')
                                                    proof (cases)
                                                      assume p x
                                                      hence eq: ?S' = insert 0 (Suc ?S)
                                                      by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                      show ?case (is ?l = card ?S')
                                                      proof (cases)
                                                        assume p x
                                                        hence eq: ?S' = insert 0 (Suc ?S)
                                                        by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                        show ?case (is ?l = card ?S')
                                                        proof (cases)
                                                          assume p x
                                                          hence eq: ?S' = insert 0 (Suc ?S)
                                                          by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                          show ?case (is ?l = card ?S')
                                                          proof (cases)
                                                            assume p x
                                                            hence eq: ?S' = insert 0 (Suc ?S)
                                                            by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                            show ?case (is ?l = card ?S')
                                                            proof (cases)
                                                              assume p x
                                                              hence eq: ?S' = insert 0 (Suc ?S)
                                                              by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                              show ?case (is ?l = card ?S')
                                                              proof (cases)
                                                                assume p x
                                                                hence eq: ?S' = insert 0 (Suc ?S)
                                                                by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                                show ?case (is ?l = card ?S')
                                                                proof (cases)
                                                                  assume p x
                                                                  hence eq: ?S' = insert 0 (Suc ?S)
                                                                  by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                                  show ?case (is ?l = card ?S')
                                                                  proof (cases)
                                                                    assume p x
                                                                    hence eq: ?S' = insert 0 (Suc ?S)
                                                                    by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                                    show ?case (is ?l = card ?S')
                                                                    proof (cases)
                                                                      assume p x
                                                                      hence eq: ?S' = insert 0 (Suc ?S)
                                                                      by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                                      show ?case (is ?l = card ?S')
                                                                      proof (cases)
                                                                        assume p x
                                                                        hence eq: ?S' = insert 0 (Suc ?S)
                                                                        by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                                        show ?case (is ?l = card ?S')
                                                                        proof (cases)
                                                                          assume p x
                                                                          hence eq: ?S' = insert 0 (Suc ?S)
                                                                          by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                                          show ?case (is ?l = card ?S')
                                                                          proof (cases)
                                                                            assume p x
                                                                            hence eq: ?S' = insert 0 (Suc ?S)
                                                                            by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                                            show ?case (is ?l = card ?S')
                                                                            proof (cases)
                                                                              assume p x
                                                                              hence eq: ?S' = insert 0 (Suc ?S)
                                                                              by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                                              show ?case (is ?l = card ?S')
                                                                              proof (cases)
                                                                                assume p x
                                                                                hence eq: ?S' = insert 0 (Suc ?S)
                                                                                by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                                                show ?case (is ?l = card ?S')
                                                                                proof (cases)
                                                                                  assume p x
                                                                                  hence eq: ?S' = insert 0 (Suc ?S)
                                                                                  by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
                                                                                  show ?case (is ?l = card ?S')
                                                                                  proof (cases)
ITU
finally show \(?\text{thesis} \).

next

assume \(\neg p x\)

hence \(\text{eq} : \?S' = \text{Suc } \?S\)

by (auto simp add: image-def split:nat.split elim:lessE)

have \(\text{length } (\text{filter } p (x \# xs)) = \text{card } \?S\)

using \(\text{Cons } (\neg p x)\) by simp

also have \(\ldots = \text{card}(\text{Suc } ?S)\) using \(\text{fin}\)

by (simp add: card-image)

also have \(\ldots = \text{card } \?S'\) using \(eq\) \(\text{fin}\)

by (simp add: card-insert-if)

finally show \(?\text{thesis}\).

qed

qed

lemma \(\text{Cons-eq-filterD}\):
\(x\#xs = \text{filter } P ys \implies \exists us vs. ys = us @ x \# vs \land (\forall u \in \text{set } us. \neg P u) \land P x \land xs = \text{filter } P vs\)

(is \(- \implies \exists us vs. \?P ys us vs)\)

proof (induct \(ys\))

case Nil

thus \(?\text{case}\) by simp

next

case (Cons \(y ys\))

show \(?\text{case } (\text{is } \exists x. \?Q x)\)

proof cases

assume \(Py: P y\)

show \(?\text{thesis}\)

proof cases

assume \(x = y\)

with \(Py\) Cons.prems have \(?Q []\) by simp

then show \(?\text{thesis } ..\)

next

assume \(x \neq y\)

with \(Py\) Cons.prems show \(?\text{thesis } by simp\)

qed

next

assume \(\neg P y\)

with Cons obtain us vs where \(?P (y\#ys) (y\#us)\) vs by fastforce

then have \(?Q (y\#us)\) by simp

then show \(?\text{thesis } ..\)

qed

lemma \(\text{filter-eq-ConsD}\):
\(\text{filter } P ys = x\#xs \implies \exists us vs. ys = us @ x \# vs \land (\forall u \in \text{set } us. \neg P u) \land P x \land xs = \text{filter } P vs\)

by (rule Cons-eq-filterD) simp

lemma \(\text{filter-eq-Cons-iff}\):
(filter \(P\) \(ys = x \# xs\)) =
(\(\exists us vs. ys = us \oplus x \# vs \land (\forall u \in \text{set} us. \neg P u) \land P x \land xs = \text{filter} \ P \ vs\))
by(auto dest:filter-eq-ConsD)

lemma Cons-eq-filter-iff:
\(x \# xs = \text{filter} \ P \ ys\) =
(\(\exists us vs. ys = us \oplus x \# vs \land (\forall u \in \text{set} us. \neg P u) \land P x \land xs = \text{filter} \ P \ vs\))
by(auto dest:Cons-eq-filterD)

lemma inj-on-filter-key-eq:
assumes \(\text{inj-on} f (\text{insert} y (\text{set} \ xs))\)
shows \(\text{filter} (\lambda x. f \ y = f x) \ xs = \text{filter} (\text{HOL.eq} \ y) \ xs\)
using assms by (induct \(xs\)) auto

lemma filter-cong[$\text{fundef-cong}$]:
\(xs = ys \Longrightarrow (\forall x. x \in \text{set} \ ys \Longrightarrow P \ x = Q \ x) \Longrightarrow \text{filter} \ P \ xs = \text{filter} \ Q \ ys\)
by (induct \(ys\) arbitrary: \(xs\)) auto

66.1.9 List partitioning

primrec partition :: ('a \Rightarrow bool) \Rightarrow 'a list \Rightarrow 'a list \times 'a list where
partition \(P\) [] = ([], [])
partition \(P\) (x # xs) =
(let (yes, no) = partition \(P\) \(xs\))
in if \(P\) \(x\) then (x # yes, no) else (yes, x # no))

lemma partition-filter1: \(\text{fst} (\text{partition} \ P \ xs) = \text{filter} \ P \ xs\)
by (induct \(xs\)) (auto simp add: Let-def split-def)

lemma partition-filter2: \(\text{snd} (\text{partition} \ P \ xs) = \text{filter} (\text{Not} \circ P) \ xs\)
by (induct \(xs\)) (auto simp add: Let-def split-def)

lemma partition-P:
assumes \(\text{partition} \ P \ xs = (yes, no)\)
shows \((\forall p \in \text{set} yes. P p) \land (\forall p \in \text{set} no. \neg P p)\)
proof –
from assms have \(yes = \text{fst} (\text{partition} \ P \ xs)\) and \(no = \text{snd} (\text{partition} \ P \ xs)\)
by simp-all
then show \(\text{thesis}\) by (simp-all add: partition-filter1 partition-filter2)
qed

lemma partition-set:
assumes \(\text{partition} \ P \ xs = (yes, no)\)
shows \(\text{set} yes \cup \text{set} no = \text{set} xs\)
proof –
from assms have \(yes = \text{fst} (\text{partition} \ P \ xs)\) and \(no = \text{snd} (\text{partition} \ P \ xs)\)
by simp-all
then show \(\text{thesis}\) by (auto simp add: partition-filter1 partition-filter2)
qed
lemma partition-filter-conv[simp]:
partition f xs = (filter f xs, filter (Not o f) xs)
unfolding partition-filter2[symmetric]
unfolding partition-filter1[symmetric] by simp

declare partition.simps[simp del]

66.1.10  concat

lemma concat-append [simp]: concat (xs @ ys) = concat xs @ concat ys
  by (induct xs) auto

lemma concat-eq-Nil-conv [simp]: (concat xs = []) = (\forall x \in set xss. x = [])
  by (induct xss) auto

lemma Nil-eq-concat-conv [simp]: (concat xss = []) = (\forall x \in set xss. x = [])
  by (induct xss) auto

lemma set-concat [simp]: set (concat xs) = (\cup x \in set xs. set x)
  by (induct xs) auto

lemma concat-map-singleton [simp]: concat(map (% x. [f x]) xs) = map f xs
  by (induct xs) auto

lemma map-concat: map f (concat xs) = concat (map (map f) xs)
  by (induct xs) auto

lemma filter-concat: filter p (concat xs) = concat (map (filter p) xs)
  by (induct xs) auto

lemma rev-concat: rev (concat xs) = concat (map rev (rev xs))
  by (induct xs) auto

lemma concat-eq-concat-iff: \forall (x, y) \in set (zip xs ys). length x = length y ==> length xs = length ys ==> (concat xs = concat ys) = (xs = ys)
proof (induct xs arbitrary: ys)
case (Cons x xs ys)
thus ?case by (cases ys) auto
qed (auto)

lemma concat-injective: concat xs = concat ys ==> length xs = length ys ==> \forall (x, y) \in set (zip xs ys). length x = length y ==> xs = ys
  by (simp add: concat-eq-concat-iff)

lemma concat-eq-appendD:
  assumes concat xss = ys @ zs xss \neq []
  shows \exists xss1 xs xss2. xss = xss1 @ (xs @ xss') \# xss2 \land ys = concat xss1 @ xs \land zs = xss' @ concat xss2
using assms

proof (induction xss arbitrary: ys)
  case (Cons xs xss)
  from Cons.prems consider
    us where xs @ us = ys concat xss = us @ zs |
    us where xs = ys @ us @ concat xss = zs
  by (auto simp add: append-eq-append-conv2)
then show ?case
proof cases
  case 1
  then show ?thesis using Cons.IH [OF 1 (2)]
    by (cases xss) (auto intro: exI [where x = []], metis append.assoc append-Cons concat.simps(2))
qed (auto intro: exI [where x = []])

qed simp

lemma concat-eq-append-conv:
  concat xss = ys @ zs ←→
  (if xss = [] then ys = [] ∧ zs = []
  else ∃ xss1 xs xs' xss2. xss = xss1 @ (xs @ xs') # xss2 ∧ ys = concat xss1 @ xs ∧ zs = xs' @ concat xss2)
  by (auto dest: concat-eq-appendD)

66.1.11 (1)

lemma nth-Cons-0 [simp, code]: (x # xs)!0 = x
  by auto

lemma nth-Cons-Suc [simp, code]: (x # xs)!(Suc n) = xs!n
  by auto

declare nth.simps [simp del]

lemma nth-Cons-pos [simp]: 0 < n =⇒ (x#xs) ! n = xs ! (n - 1)
  by (auto simp: Nat.gr0_conv_Suc)

lemma nth-append:
  (xs @ ys)!n = (if n < length xs then xs!n else ys!(n - length xs))
proof (induct xs arbitrary: n)
  case (Cons x xs)
  then show ?case
    using less-Suc-eq-0-disj by auto
qed simp

lemma nth-append-length [simp]: (xs @ x ≠ ys) ! length xs = x
  by (induct xs) auto

lemma nth-append-length-plus [simp]: (xs @ ys) ! (length xs + n) = ys ! n
  by (induct xs) auto
lemma nth-map [simp]: \( n < \text{length } xs \implies (\text{map } f \; xs)!n = f(xs!n) \)
proof (induct xs arbitrary: n)
  case (Cons x xs)
  then show ?case
    using less-Suc-eq-0-disj by auto
qed simp

lemma nth-tl: \( n < \text{length } (\text{tl } xs) \implies \text{tl } xs ! n = xs ! (Suc \; n) \)
by (induction xs) auto

lemma hd-conv-nth: \( xs \neq [] \implies \text{hd } xs = xs!0 \)
by (cases xs) simp-all

lemma list-eq-iff-nth-eq:
\((xs = ys) = (\text{length } xs = \text{length } ys \land (\forall i < \text{length } xs. xs!i = ys!i))\)
proof (induct xs arbitrary: ys)
  case (Cons x xs ys)
  show ?case
    proof (cases ys)
      case (Cons y ys)
      then show ?thesis
        using Cons.hyps by fastforce
    qed
  qed simp
  qed force

lemma set-conv-nth: \( \text{set } xs = \{xs!i \mid i. \; i < \text{length } xs\} \)
proof (induct xs)
  case (Cons x xs)
  have insert x \{xs ! i \mid i. \; i < \text{length } xs\} = \{(x # xs)!i \mid i. \; i < Suc (\text{length } xs)\}
  \( (\text{is } ?L=\; ?R) \)
  proof
    show ?L \subseteq ?R
      by force
    show ?R \subseteq ?L
      using less-Suc-eq-0-disj by auto
  qed
  with Cons show ?case
    by simp
  qed simp

lemma in-set-conv-nth: \( (x \in \text{set } xs) = (\exists i < \text{length } xs. xs!i = x) \)
by (auto simp: set-conv-nth)

lemma nth-equal-first-eq:
assumes \( x \notin \text{set } xs \)
assumes \( n \leq \text{length } xs \)
sshows \( (x # xs)!n = x \implies n = 0 \) (is \( \text{?lhs} \implies \text{?rhs} \))
proof
  assume ?lhs
  show ?rhs
proof (rule ccontr)
  assume n ≠ 0
  then have n > 0 by simp
  with (?lhs) have xs ! (n - 1) = x by simp
  moreover from (n > 0; n ≤ length xs) have n - 1 < length xs by simp
  ultimately have ∃ i < length xs. xs ! i = x by auto
  with (x ι set xs) in-set-conv-nth [of x xs] show False by simp
qed
next
  assume ?rhs then show ?lhs by simp
qed

lemma nth-non-equal-first-eq:
  assumes x ≠ y
  shows (x # xs) ! n = y ↔ xs ! (n - 1) = y ∧ n > 0 (is ?lhs ↔ ?rhs)
proof
  assume ?lhs with assms have n > 0 by (cases n) simp-all
  with (?lhs) show ?rhs by simp
next
  assume ?rhs then show ?lhs by simp
qed

lemma list-ball-nth: [n < length xs; ∀ x ∈ set xs. P x] ⇒ P(xs!n)
  by (auto simp add: set-conv-nth)

lemma nth-mem [simp]: n < length xs ⇒ xs!n ∈ set xs
  by (auto simp add: set-conv-nth)

lemma all-nth-imp-all-set:
  [∀ i < length xs. P(xs!i); x ∈ set xs] ⇒ P x
  by (auto simp add: set-conv-nth)

lemma all-set-conv-all-nth:
  (∀ x ∈ set xs. P x) = (∀ i. i < length xs → P (xs ! i))
  by (auto simp add: set-conv-nth)

lemma rev-nth:
  n < size xs ⇒ rev xs ! n = xs ! (length xs - Suc n)
proof (induct xs arbitrary: n)
  case Nil thus ?case by simp
next
  case (Cons x xs)
  hence n: n < Suc (length xs) by simp
  moreover
  { assume n < length xs
    with n obtain n' where n': length xs - n = Suc n'}
by (cases length \(xs\) = \(n\), auto)

moreover

from \(n'\) have \(\text{length }xs - \text{Suc }n = n'\) by simp

ultimately

have \(xs! (\text{length }xs - \text{Suc }n) = (x \neq \text{xs})\) \((\text{length }xs - n)\) by simp

ultimately

show \(?\text{case}\) by (clarsimp simp add: \text{Cons nth-append})

qed

lemma Skolem-list-nth:

\[
(\forall i < k. \exists x. P i x) = (\exists xs. \text{size }xs = k \land (\forall i < k. P i (xs!i)))
\]

(is - = (\exists xs. \?P k xs))

proof (induct \(k\))

next

next (Suc \(k\))

show \(?\text{case}\) (is \(?L = \?R\) is - = (\exists xs. \?P' xs))

proof

assume \(?R\) thus \(?L\) using Suc by auto

next

assume \(?L\)

with Suc obtain \(x\) \(xs\) where \(?P k xs \land P k x\) by (metis less-Suc-eq)

hence \(?P' (xs@[x])\) by (simp add: \text{nth-append less-Suc-eq})

thus \(?R\) ..

qed

qed

66.1.12  list-update

lemma length-list-update [simp]: \(\text{length}(xs[i:=x]) = \text{length }xs\)

by (induct \(xs\) arbitrary: \(i\)) (auto split: \text{nat.split})

lemma nth-list-update:

\(i < \text{length }xs\Longrightarrow (xs[i:=x])!j = (\text{if } i = j \text{ then } x \text{ else } xs!j)\)

by (induct \(xs\) arbitrary: \(i\) \(j\)) (auto simp add: \text{nth-Cons split: \text{nat.split}})

lemma nth-list-update-eq [simp]: \(i < \text{length }xs \Longrightarrow (xs[i:=x])!i = x\)

by (simp add: \text{nth-list-update})

lemma nth-list-update-neq [simp]: \(i \neq j \Longrightarrow xs[i:=x]!j = xs!j\)

by (induct \(xs\) arbitrary: \(i\) \(j\)) (auto simp add: \text{nth-Cons split: \text{nat.split}})

lemma list-update-id [simp]: \(xs[i := xs!i] = xs\)

by (induct \(xs\) arbitrary: \(i\)) (simp-all split: \text{nat.splits})

lemma list-update-beyond [simp]: \(\text{length }xs \leq i \Longrightarrow xs[i:=x] = xs\)

proof (induct \(xs\) arbitrary: \(i\))

  case (\text{Cons }xs \(i\))
then show \(?\)case
  by (metis leD length-list-update list-eq-iff-nth-eq nth-list-update-neq)
qed simp

lemma list-update-nonempty[simp]: \(xs[k:=x] = [] \iff xs=[]\)
  by (simp only: length-0-conv[symmetric] length-list-update)

lemma list-update-same-conv:
  \(i < \text{size} \; xs \implies (xs \at i := x) = (xs \at i := x)\)
  by (induct xs arbitrary: i) (auto split: nat.split)

lemma list-update-append1:
  \(i < \text{size} \; xs \implies (xs @ ys) \at i := x = (xs \at i := x) @ ys\)
  by (induct xs arbitrary: i) (auto split: nat.split)

lemma list-update-append:
  \(\text{(xs @ ys) \at n := x} = (\text{if } n < \text{length} \; xs \text{ then } xs \at n := x \at n := x \text{ else } xs @ \text{(ys \at n := x := x)})\)
  by (induct xs arbitrary: n) (auto split: nat.splits)

lemma list-update-length [simp]:
  \((xs \at x 
ot= ys)[\text{length} \; xs := y] = (xs \at y \not= ys)\)
  by (induct xs, auto)

lemma map-update: \(\text{map} \; f \; (xs[k := y]) = (\text{map} \; f \; xs)[k := f \; y]\)
  by (induct xs arbitrary: k) (auto split: nat.splits)

lemma rev-update:
  \(k < \text{length} \; xs \implies \text{rev} \; (xs[k := y]) = (\text{rev} \; xs)[\text{length} \; xs - k - 1 := y]\)
  by (induct xs arbitrary: k) (auto simp: list-update-append split: nat.splits)

lemma update-zip:
  \((\text{zip} \; xs \; ys)[i := xy] = \text{zip} \; (xs[i := \text{fst} \; xy]) \; (ys[i := \text{snd} \; xy])\)
  by (induct ys arbitrary: i xy xs) (auto, case-tac xs, auto split: nat.split)

lemma set-update-subset-insert: \(\text{set} \; (xs[i := x]) \subseteq \text{insert} \; x \; (\text{set} \; xs)\)
  by (induct xs arbitrary: i) (auto split: nat.split)

lemma set-update-subsetI: \(\text{set} \; xs \subseteq A; \; x \in A \implies \text{set} \; (xs[i := x]) \subseteq A\)
  by (blast dest!: set-update-subset-insert [THEN subsetD])

lemma set-update-memI:
  \(n < \text{length} \; xs \implies x \in \text{set} \; (xs[n := x])\)
  by (induct xs arbitrary: n) (auto split: nat.splits)

lemma list-update-overwrite[simp]:
  \(xs[i := x, i := y] = xs[i := y]\)
  by (induct xs arbitrary: i) (simp-all split: nat.split)

lemma list-update-swap:
THEORY "List"

\[ i \neq i' \implies xs [i := x, i' := x'] = xs [i' := x', i := x] \]
by (induct xs arbitrary: i i') (simp-all split: nat.split)

lemma list-update-code [code]:
\[
\begin{align*}
[x[i := y] = &\ []] \\
(x # xs)[0 := y] = &\ y # xs \\
(x # xs)[\text{Suc } i := y] = &\ x # xs[i := y]
\end{align*}
\]
by simp-all

66.1.13 last and butlast

lemma last-snoc [simp]: last \((xs @ [x])\) = x
by (induct xs) auto

lemma butlast-snoc [simp]: butlast \((xs @ [x])\) = xs
by (induct xs) auto

lemma last-ConsL: \(xs = [] \implies \text{last}(x#xs) = x\)
by simp

lemma last-ConsR: \(xs \neq [] \implies \text{last}(x#xs) = \text{last} xs\)
by simp

lemma last-append: \(\text{last}(xs @ ys) = (\text{if } ys = [] \text{ then } \text{last} xs \text{ else } \text{last} ys)\)
by (induct xs) (auto)

lemma last-appendL[simp]: \(ys = [] \implies \text{last}(xs @ ys) = \text{last} xs\)
by (simp add: last-append)

lemma last-appendR[simp]: \(ys \neq [] \implies \text{last}(xs @ ys) = \text{last} ys\)
by (simp add: last-append)

lemma last-tl: \(xs = [] \lor \text{tl} \; xs \neq [] \implies \text{last} (\text{tl} \; xs) = \text{last} xs\)
by (induct xs) simp-all

lemma butlast-tl: \(\text{butlast} (\text{tl} \; xs) = \text{tl} (\text{butlast} \; xs)\)
by (induct xs) simp-all

lemma hd-rev: \(xs \neq [] \implies \text{hd}(\text{rev} \; xs) = \text{last} xs\)
by (rule rev-exhaust[of xs]) simp-all

lemma last-rev: \(xs \neq [] \implies \text{last}(\text{rev} \; xs) = \text{hd} \; xs\)
by (cases xs) simp-all

lemma last-in-set[simp]: \(as \neq [] \implies \text{last} \; as \in \text{set} \; as\)
by (induct as) auto

lemma length-butlast [simp]: \(\text{length} \; (\text{butlast} \; xs) = \text{length} \; xs - 1\)
by (induct xs rule: rev-induct) auto
lemma butlast-append:
\[ \text{butlast}\ (xs \@\ ys) = (\text{if } ys = [] \text{ then butlast } xs \text{ else } xs \@\ \text{butlast } ys) \]
by (induct xs arbitrary: ys) auto

lemma append-butlast-last-id [simp]:
\[ xs \neq [] \Rightarrow \text{butlast } xs \@\ [\text{last } xs] = xs \]
by (induct xs) auto

lemma in-set-butlastD:
\[ x \in \text{set}\ (\text{butlast } xs) \Rightarrow x \in \text{set } xs \]
by (induct xs) (auto split: if-split-asm)

lemma in-set-butlast-appendI:
\[ x \in \text{set}\ (\text{butlast } xs) \lor x \in \text{set}\ (\text{butlast } ys) \Rightarrow x \in \text{set}\ (\text{butlast } (xs \@\ ys)) \]
by (auto dest: in-set-butlastD simp add: butlast-append)

lemma last-drop[simp]:
\[ n < \text{length } xs \Rightarrow \text{last } (\text{drop } n\ xs) = \text{last } xs \]
by (induct xs arbitrary: n)(auto split:nat.split)

lemma nth-butlast:
assumes \( n < \text{length } (\text{butlast } xs) \) shows \( \text{butlast } xs [\_\ldots\_ ! n = xs [\_\ldots\_ ! n \]
proof (cases xs)
  case (Cons y ys)
  moreover from \( \text{assms} \) have \( \text{butlast } xs [\_\ldots\_ ! n = (\text{butlast } xs [\_\ldots\_ ! (\text{last } xs)]) [\_\ldots\_ ! n \]
  by (simp add: nth-append)
  ultimately show \( ?\text{thesis} \) using append-butlast-last-id by simp
qed simp

lemma last-conv-nth:
\[ xs \neq [] \Rightarrow \text{last } xs = xs ![\text{length } xs - 1] \]
by (induct xs)(auto simp:neq-Nil-conv)

lemma butlast-conv-take:
\( \text{butlast } xs = \text{take } (\text{length } xs - 1)\ xs \)
by (induction \( xs \) rule: induct-list012) simp-all

lemma last-list-update:
\[ xs \neq [] \Rightarrow \text{last } (xs [k:=x]) = (\text{if } k = \text{size } xs - 1 \text{ then } x \text{ else last } xs) \]
by (auto simp: last-conv-nth)

lemma butlast-list-update:
\( \text{butlast}(xs[k:=x]) = \)
\( (\text{if } k = \text{size } xs - 1 \text{ then butlast } xs \text{ else } (\text{butlast } xs)[k:=x]) \)
by (cases \( xs \) rule:rev-cases)(auto simp: list-update-append split: nat.splits)

lemma last-map:
\[ xs \neq [] \Rightarrow \text{last } (\text{map } f\ xs) = f\ (\text{last } xs) \]
by (cases \( xs \) rule: rev-cases) simp-all

lemma map-butlast:
\( \text{map } f\ (\text{butlast } xs) = \text{butlast } (\text{map } f\ xs) \)
by (induct \( xs \)) simp-all
lemma snoc-eq-iff-butlast:
\[
\text{xs} @ [x] = \text{ys} \iff (\text{ys} \neq [] \land \text{butlast} \text{ ys} = \text{xs} \land \text{last} \text{ ys} = x)
\]
by fastforce

corollary longest-common-suffix:
\[
\exists \text{ss} \text{ xs'} \text{ ys'}. \text{xs} = \text{xs'} \@ \text{ss} \land \text{ys} = \text{ys'} \@ \text{ss} \\
\land (\text{xs'} = [] \lor \text{ys'} = [] \lor \text{last} \text{ xs'} \neq \text{last} \text{ ys'})
\]
using longest-common-prefix[of rev \text{xs} rev \text{ys}]
unfolding rev-swap rev-append by (metis last-rev rev-is-Nil-conv)

lemma butlast-rev [simp]: \text{butlast} (rev \text{xs}) = rev (tl \text{xs})
by (cases \text{xs}) simp-all

66.1.14 take and drop

lemma take-0: take 0 \text{xs} = []
by (induct \text{xs}) auto

lemma drop-0: drop 0 \text{xs} = \text{xs}
by (induct \text{xs}) auto

lemma take0[simp]: take 0 = (\lambda \text{xs}. [])
by (rule ext) (rule take-0)

lemma drop0[simp]: drop 0 = (\lambda \text{x}. \text{x})
by (rule ext) (rule drop-0)

lemma take-Suc-Cons [simp]: take (Suc \text{n}) (x \# \text{xs}) = x \# take \text{n} \text{xs}
by simp

lemma drop-Suc-Cons [simp]: drop (Suc \text{n}) (x \# \text{xs}) = drop \text{n} \text{xs}
by simp

declare take-Cons [simp del] and drop-Cons [simp del]

lemma take-Suc: \text{xs} \neq [] \implies take (Suc \text{n}) \text{xs} = hd \text{xs} \# take \text{n} (tl \text{xs})
by (clarsimp simp add: neq-Nil-conv)

lemma drop-Suc: drop (Suc \text{n}) \text{xs} = drop \text{n} (tl \text{xs})
by (cases \text{xs}, simp-all)

lemma hd-take[simp]: j > 0 \implies hd (take j \text{xs}) = hd \text{xs}
by (metis gr0-conv-Suc list.sel(1) take.simps(1) take-Suc)

lemma take-tl: take \text{n} (tl \text{xs}) = tl (take (Suc \text{n}) \text{xs})
by (induct \text{xs} arbitrary: \text{n}) simp-all

lemma drop-tl: drop \text{n} (tl \text{xs}) = tl(drop \text{n} \text{xs})
by (induct \text{xs} arbitrary: \text{n}, simp-all add:drop-Cons drop-Suc split:nat.split)
lemma tl-take: \( tl (\text{take } n \; xs) = \text{take } (n - 1) \; (tl \; xs) \)
by (cases \( n \), simp, cases \( xs \), auto)

lemma tl-drop: \( tl (\text{drop } n \; xs) = \text{drop } n \; (tl \; xs) \)
by (simp only: drop-tl)

lemma nth-via-drop: \( \text{drop } n \; xs = y \# ys \implies xs!n = y \)
by (induct \( xs \) arbitrary: \( n \), simp)(auto simp: drop-Cons nth-Cons split: nat.splits)

lemma take-Suc-conv-app-nth:
\( i < \text{length } xs \implies \text{take } (\text{Suc } i) \; xs = \text{take } i \; xs \; @ \; [xs!i] \)
proof (induct \( xs \) arbitrary: \( i \))
  case Nil
  then show ?case by simp
next
  case Cons
  then show ?case by (cases \( i \)) auto
qed

lemma Cons-nth-drop-Suc:
\( i < \text{length } xs \implies (xs!i) \# (\text{drop } (\text{Suc } i) \; xs) = \text{drop } i \; xs \)
proof (induct \( xs \) arbitrary: \( i \))
  case Nil
  then show ?case by simp
next
  case Cons
  then show ?case by (cases \( i \)) auto
qed

lemma length-take [simp]: \( \text{length } (\text{take } n \; xs) = \text{min } (\text{length } xs) \; n \)
by (induct \( n \) arbitrary: \( xs \))(auto, case-tac \( xs \), auto)

lemma length-drop [simp]: \( \text{length } (\text{drop } n \; xs) = (\text{length } xs - n) \)
by (induct \( n \) arbitrary: \( xs \))(auto, case-tac \( xs \), auto)

lemma take-all [simp]: \( \text{length } xs \leq n \implies \text{take } n \; xs = xs \)
by (induct \( n \) arbitrary: \( xs \))(auto, case-tac \( xs \), auto)

lemma drop-all [simp]: \( \text{length } xs \leq n \implies \text{drop } n \; xs = [] \)
by (induct \( n \) arbitrary: \( xs \))(auto, case-tac \( xs \), auto)

lemma take-append [simp]:
\( \text{take } n \; (xs \; @ \; ys) = (\text{take } n \; xs \; @ \; \text{take } (n - \text{length } xs) \; ys) \)
by (induct \( n \) arbitrary: \( xs \))(auto, case-tac \( xs \), auto)

lemma drop-append [simp]:
\( \text{drop } n \; (xs \; @ \; ys) = \text{drop } n \; xs \; @ \; \text{drop } (n - \text{length } xs) \; ys \)
by (induct \( n \) arbitrary: \( xs \))(auto, case-tac \( xs \), auto)
lemma take-take [simp]: take n (take m xs) = take (min n m) xs
proof (induct m arbitrary: xs n)
  case 0
  then show ?case by simp
next
  case Suc
  then show ?case by (cases xs; cases n) simp-all
qed

lemma drop-drop [simp]: drop n (drop m xs) = drop (n + m) xs
proof (induct m arbitrary: xs)
  case 0
  then show ?case by simp
next
  case Suc
  then show ?case by (cases xs) simp-all
qed

lemma take-drop: take n (drop m xs) = drop m (take (n + m) xs)
proof (induct m arbitrary: xs n)
  case 0
  then show ?case by simp
next
  case Suc
  then show ?case by (cases xs; cases n) simp-all
qed

lemma drop-take: drop n (take m xs) = take (m - n) (drop n xs)
  by (induct xs arbitrary: m n) (auto simp: take-Cons drop-Cons split: nat.split)

lemma append-take-drop-id [simp]: take n xs @ drop n xs = xs
proof (induct n arbitrary: xs)
  case 0
  then show ?case by simp
next
  case Suc
  then show ?case by (cases xs) simp-all
qed

lemma take-eq-nil [simp]: (take n xs = []) = (n = 0 ∨ xs = [])
  by (induct xs arbitrary: n) (auto simp: take-Cons split: nat.split)

lemma drop-eq-nil [simp]: (drop n xs = []) = (length xs ≤ n)
  by (induct xs arbitrary: n) (auto simp: drop-Cons split: nat.split)

lemma take-map: take n (map f xs) = map f (take n xs)
proof (induct n arbitrary: xs)
  case 0
then show \( \text{?case by simp} \)
next
case Suc
then show \( \text{?case by (cases xs) simp-all} \)
qed

lemma \text{drop-map}: \text{drop n (map f xs) = map f (drop n xs)}
proof (induct n arbitrary: xs)
  case 0
  then show \( \text{?case by simp} \)
next
case Suc
then show \( \text{?case by (cases xs) simp-all} \)
qed

lemma \text{rev-take}: \text{rev (take i xs) = drop (length xs - i) (rev xs)}
proof (induct xs arbitrary: i)
  case Nil
  then show \( \text{?case by simp} \)
next
case Cons
then show \( \text{?case by (cases i) auto} \)
qed

lemma \text{rev-drop}: \text{rev (drop i xs) = take (length xs - i) (rev xs)}
proof (induct xs arbitrary: i)
  case Nil
  then show \( \text{?case by simp} \)
next
case Cons
then show \( \text{?case by (cases i) auto} \)
qed

lemma \text{drop-rev}: \text{drop n (rev xs) = rev (take (length xs - n) xs)}
by (cases length xs < n) (auto simp: rev-take)

lemma \text{take-rev}: \text{take n (rev xs) = rev (drop (length xs - n) xs)}
by (cases length xs < n) (auto simp: rev-drop)

lemma \text{nth-take [simp]}: \( i < n \implies (take n xs)!i = xs!i \)
proof (induct xs arbitrary: i n)
  case Nil
  then show \( \text{?case by simp} \)
next
case Cons
then show \( \text{?case by (cases n; cases i) simp-all} \)
qed

lemma \text{nth-drop [simp]}:
\( n \leq \text{length } xs \implies (\text{drop } n \ xs)!i = xs!(n + i) \)

**proof** (induct \( n \) arbitrary: \( xs \))

  - case \( \emptyset \)
    
    then show \( \text{?case by simp} \)
  
  next

  - case \( \text{Suc} \)
    
    then show \( \text{?case by (cases } xs) \text{ simp-all} \)

  qed

**lemma** butlast-take:

\( n \leq \text{length } xs \implies \text{butlast (take } n \ xs) = \text{take (n - 1) } xs \)

by (simp add: butlast-conv-take min.absorb1 min.absorb2)

**lemma** butlast-drop: \( \text{butlast (drop } n \ xs) = \text{drop } n \ (\text{butlast } xs) \)

by (simp add: butlast-conv-take drop-take ac-simps)

**lemma** take-butlast: \( n < \text{length } xs \implies \text{take } n \ (\text{butlast } xs) = \text{take } n \ xs \)

by (simp add: butlast-conv-take min.absorb1)

**lemma** hd-drop-conv-nth: \( n < \text{length } xs \implies \text{hd (drop } n \ xs) = xs!n \)

by (simp add: hd-conv-nth)

**lemma** set-take-subset-set-take:

\( m \leq n \implies \text{set (take } m \ xs) \subseteq \text{set (take } n \ xs) \)

**proof** (induct \( xs \) arbitrary: \( m \ n \))

  - case \( \text{Cons } x \ xs \ m \ n \)
    
    then show \( \text{?case by (cases } n) \text{ (auto simp: take-Cons)} \)

  qed simp

**lemma** set-take-subset: \( \text{set (take } n \ xs) \subseteq \text{set } xs \)

by (induct \( xs \) arbitrary: \( n \)) (auto simp: take-Cons split:nat.split)

**lemma** set-drop-subset: \( \text{set (drop } n \ xs) \subseteq \text{set } xs \)

by (induct \( xs \) arbitrary: \( n \)) (auto simp: drop-Cons split:nat.split)

**lemma** set-drop-subset-set-drop:

\( m \geq n \implies \text{set (drop } m \ xs) \subseteq \text{set (drop } n \ xs) \)

**proof** (induct \( xs \) arbitrary: \( m \ n \))

  - case \( \text{Cons } x \ xs \ m \ n \)
    
    then show \( \text{?case by (clarsimp simp: drop-Cons split: nat.split) (metis set-drop-subset subset_iff)} \)

  qed simp

**lemma** in-set-takeD: \( x \in \text{set (take } n \ xs) \implies x \in \text{set } xs \)

using set-take-subset by fast
lemma in-set-dropD: x ∈ set(drop n xs) ⟹ x ∈ set xs
   using set-drop-subset by fast

lemma append-eq-conj:
  (xs @ ys = zs) = (xs = take (length xs) zs ∧ ys = drop (length xs) zs)
proof (induct xs arbitrary: zs)
  case (Cons x xs zs) then show ?case
    by (cases zs, auto)
qed

lemma map-eq-append-conv:
  map f xs = ys @ zs ←→ (∃ us vs. xs = us @ vs ∧ ys = map f us ∧ zs = map f vs)
proof
  have map f xs ≠ ys @ zs ∧ map f xs ≠ ys @ zs ∨ map f xs ≠ ys @ zs ∨ map f xs ≠ ys @ zs ∧
    (∃ bs bsa. xs = bs @ bsa ∧ ys = map f bs ∧ zs = map f bsa)
    by (metis append-eq-conj append-take-drop-id drop-map take-map)
  then show ?thesis
    using map-append by blast
qed

lemma append-eq-append-conv-if:
  (xs1 @ xs2 = ys1 @ ys2) =
    (if size xs1 ≤ size ys1
      then xs1 = take (size xs1) ys1 ∧ xs2 = drop (size xs1) ys1 @ ys2
      else take (size ys1) xs1 = ys1 ∧ drop (size ys1) xs1 @ xs2 = ys2)
proof (induct xs1 arbitrary: ys1)
  case (Cons a xs1 ys1) then show ?case
    by (cases ys1, auto)
qed

lemma take-add: take (i+j) xs = take i xs @ take j (drop i xs)
proof (induct xs arbitrary: i)
  case (Cons x xs i) then show ?case
    by (cases i, auto)
qed

lemma take-hd-drop:
  n < length xs ⟹ take n xs @ [hd (drop n xs)] = take (Suc n) xs
by (induct xs arbitrary: n) (simp-all add:drop-Cons split:nat.split)

lemma id-take-nth-drop:
  i < length xs ⟹ xs = take i xs @ xs!i # drop (Suc i) xs
proof –
  assume si: i < length xs
hence xs = take (Suc i) xs @ drop (Suc i) xs by auto
moreover
from si have take (Suc i) xs = take i xs @ [xs!i]
  using take-Suc-conv-app-nth by blast
ultimately show ?thesis by auto
qed

lemma take-update-cancel[simp]: n ≤ m ⇒ take n (xs[m := y]) = take n xs
by (simp add: list-eq-iff-nth-eq)

lemma drop-update-cancel[simp]: n < m ⇒ drop m (xs[n := x]) = drop m xs
by (simp add: list-eq-iff-nth-eq)

lemma upd-conv-take-nth-drop:
i < length xs ⇒ xs[i:=a] = take i xs @ a # drop (Suc i) xs
proof –
  assume i: i < length xs
have xs[i:=a] = (take i xs @ xs[i # drop (Suc i) xs][i:=a]
  by (rule arg-cong[OF id-take-nth-drop[OF i]])
also have .. = take i xs @ a # drop (Suc i) xs
  using i by (simp add: list-update-append)
finally show ?thesis.
qed

lemma take-update-swap: take m (xs[n := x]) = (take m xs)[n := x]
proof (cases n ≥ length xs)
case False
  then show ?thesis
  by (simp add: upd-conv-take-nth-drop take-Cons drop-take min-def diff-Suc split: nat.split)
qed auto

lemma drop-update-swap:
assumes m ≤ n shows drop m (xs[n := x]) = (drop m xs)[n−m := x]
proof (cases n ≥ length xs)
case False
  with assms show ?thesis
  by (simp add: upd-conv-take-nth-drop drop-take)
qed auto

lemma nth-image: l ≤ size xs ⇒ nth xs \{0..<l\} = set(take l xs)
by (auto simp: set-conv-nth image-def) (metis Suc-le-eq nth-take order-trans)

66.1.15 takeWhile and dropWhile

lemma length-takeWhile-le: length (takeWhile P xs) ≤ length xs
by (induct xs) auto
lemma takeWhile-dropWhile-id [simp]: takeWhile P xs @ dropWhile P xs = xs
by (induct xs) auto

lemma takeWhile-append1 [simp]:
[x ∈ set xs; ¬P(x)] ⟹ takeWhile P (xs @ ys) = takeWhile P xs
by (induct xs) auto

lemma takeWhile-append2 [simp]:
(∀x. x ∈ set xs ⟹ P x) ⟹ takeWhile P (xs @ ys) = xs @ takeWhile P ys
by (induct xs) auto

lemma takeWhile-tail: ¬ P x ⟹ takeWhile P (xs @ (x#)) = takeWhile P xs
by (induct xs) auto

lemma takeWhile-nth: j < length (takeWhile P xs) ⟹ takeWhile P xs ! j = xs ! j
by (metis nth-append takeWhile-dropWhile-id)

lemma dropWhile-nth: j < length (dropWhile P xs) ⟹ dropWhile P xs ! j = xs ! (j + length (takeWhile P xs))
by (metis add.commute nth-append-length-plus takeWhile-dropWhile-id)

lemma length-dropWhile-le: length (dropWhile P xs) ≤ length xs
by (induct xs) auto

lemma dropWhile-append1 [simp]:
[x ∈ set xs; ¬P(x)] ⟹ dropWhile P (xs @ ys) = (dropWhile P xs)@ys
by (induct xs) auto

lemma dropWhile-append2 [simp]:
(∀x. x ∈ set xs ⟹ P(x)) ⟹ dropWhile P (xs @ ys) = dropWhile P ys
by (induct xs) auto

lemma dropWhile-append3:
¬ P y ⟹ dropWhile P (xs @ y # ys) = dropWhile P xs @ y # ys
by (induct xs) auto

lemma dropWhile-last:
xs ∈ set xs ⟹ ¬ P x ⟹ last (dropWhile P xs) = last xs
by (auto simp add: dropWhile-append3 in-set-conv-decomp)

lemma set-dropWhileD: xs ∈ set (dropWhile P xs) ⟹ xs ∈ set xs
by (induct xs) (auto split: if-split-asm)

lemma set-takeWhileD: xs ∈ set (takeWhile P xs) ⟹ xs ∈ set xs ∧ P x
by (induct xs) (auto split: if-split-asm)

lemma takeWhile-eq-all-conv[simp]:
THEORY "List"

(takeWhile P xs = xs) = (∀ x ∈ set xs. P x)
by (induct xs, auto)

lemma dropWhile-eq-Nil-conv [simp]:
(dropWhile P xs = []) = (∀ x ∈ set xs. P x)
by (induct xs, auto)

lemma dropWhile-eq-Cons-conv:
(dropWhile P xs = y#ys) = (xs = takeWhile P xs @ y @ ys ∧ ¬ P y)
by (induct xs, auto)

lemma distinct-takeWhile [simp]:
distinct xs = distinct (takeWhile P xs)
by (induct xs) (auto dest: set-takeWhileD)

lemma distinct-dropWhile [simp]:
distinct xs = distinct (dropWhile P xs)
by (induct xs) auto

lemma takeWhile-map: takeWhile P (map f xs) = map f (takeWhile (P o f) xs)
by (induct xs) auto

lemma dropWhile-map: dropWhile P (map f xs) = map f (dropWhile (P o f) xs)
by (induct xs) auto

lemma takeWhile-eq-take: takeWhile P xs = take (length (takeWhile P xs)) xs
by (induct xs) auto

lemma dropWhile-eq-drop: dropWhile P xs = drop (length (takeWhile P xs)) xs
by (induct xs) auto

lemma hd-dropWhile: dropWhile P xs ≠ [] → ¬ P (hd (dropWhile P xs))
by (induct xs) auto

lemma takeWhile-eq-filter:
assumes ( dropWhile P xs ) → ¬ P x
shows takeWhile P xs = filter P xs
proof –
have A: filter P xs = filter P (takeWhile P xs @ dropWhile P xs)
    by simp
    have B: filter P (dropWhile P xs) = []
        unfolding filter-empty-conv using assms by blast
    have filter P xs = takeWhile P xs
        unfolding A filter-append B
        by (auto simp add: filter-id-conv dest: set-takeWhileD)
    thus ?thesis ..
    qed

lemma takeWhile-eq-take-P-nth:
[ [ ∃ i. i < n ; i < length xs ] → P (xs ! i) ; n < length xs → ¬ P (xs ! n) ] →
takeWhile \( P \, x \)s = take \( n \) \( x \)s

**proof**

- **case** Nil
- **thus** case by simp

**next**

- **case** (Cons \( x \) \( x \)s)
- **show** case
  **proof**
  **(cases** \( n \)**)
  **case** 0
  - with Cons **show** thesis by simp
  **next**
  **case** \( \)simp; (Suc \( n \)′)
  - have \( P \, x \) using Cons.prems(1)[of 0] by simp
  **moreover** have takeWhile \( P \, x \)s = take \( n \)′ \( x \)s
    **proof**
    **(rule** Cons.hyps)
    - fix \( i \)
      - assume \( i < n \)′ \( i < \) length \( x \)s
        - thus \( (x \, ! \, i) \) using Cons.prems(1)[of Suc \( i \)] by simp
    **next**
    - assume \( n \)′ < length \( x \)s
      - thus \( \)P \( (x \, ! \, n \)′) using Cons by auto
    **qed**
    **ultimately** show thesis by simp
  **qed**

**qed**

**lemma** nth-length-takeWhile:

\[
\text{length (takeWhile } P \text{ xs) < length xs} \implies \neg \, P \, (x \, ! \, (\text{length (takeWhile } P \text{ xs)}))
\]

**by** (induct \( x \)s) auto

**lemma** length-takeWhile-less-P-nth:

- **assumes** all: \( \land \, i < j \implies P \, (x \, ! \, i) \) and \( j \leq \text{length \( x \)s} \)
- **shows** \( j \leq \text{length (takeWhile } P \text{ xs)} \)

**proof**

**(rule** classical)

- assume \( \neg \)thesis
- hence \( \text{length (takeWhile } P \text{ xs) < length xs using assns by simp} \)
- thus ?thesis using all (¬ ?thesis) nth-length-takeWhile[of \( P \, x \)s] by auto
  **qed**

**lemma** takeWhile-neq-rev: [distinct \( x \)s; \( x \in \text{set \( x \)s}\] \implies 

\[
\text{takeWhile } (\lambda y. y \neq x) \text{ (rev } x \text{s)} = \text{rev (tl (dropWhile } (\lambda y. y \neq x) \text{ x)s))}
\]

**by**(induct \( x \)s) (auto simp: takeWhile-tail[where l=[]])

**lemma** dropWhile-neq-rev: [distinct \( x \)s; \( x \in \text{set \( x \)s}\] \implies 

\[
\text{dropWhile } (\lambda y. y \neq x) \text{ (rev } x \text{s)} = x \# \text{rev (takeWhile } (\lambda y. y \neq x) \text{ x)s)}
\]

**proof**

**(induct \( x \)s)

- **case** (Cons \( a \) \( x \)s)
- **then** show case
  **by**(auto, subst dropWhile-append2, auto)
qed simp

lemma takeWhile-not-last:
distinct xs \implies takeWhile (\lambda y. y \neq \text{last} \: \text{xs}) \: \text{xs} = \text{butlast} \: \text{xs}
by (induction \: \text{xs} \: \text{rule: induct-list012}) \: \text{auto}

lemma takeWhile-cong [fundef-cong]:
[l = k; \forall x. \: x \in \text{set} \: l \implies P \: x = \: Q \: x] 
\implies \text{takeWhile} \: P \: l = \text{takeWhile} \: Q \: k
by (induct k \: \text{arbitrary:} \: l) \: (simp-all)

lemma dropWhile-cong [fundef-cong]:
[l = k; \forall x. \: x \in \text{set} \: l \implies P \: x = \: Q \: x] 
\implies \text{dropWhile} \: P \: l = \text{dropWhile} \: Q \: k
by (induct k \: \text{arbitrary:} \: l, simp-all)

lemma takeWhile-idem [simp]:
\text{takeWhile} \: P \: (\text{takeWhile} \: P \: \text{xs}) = \text{takeWhile} \: P \: \text{xs}
by (induct \: \text{xs}) \: \text{auto}

lemma dropWhile-idem [simp]:
\text{dropWhile} \: P \: (\text{dropWhile} \: P \: \text{xs}) = \text{dropWhile} \: P \: \text{xs}
by (induct \: \text{xs}) \: \text{auto}

66.1.16  zip

lemma zip-Nil [simp]: zip [] \: \text{ys} = []
by (induct \: \text{ys}) \: \text{auto}

lemma zip-Cons-Cons [simp]: zip (x \# \text{xs}) \: (y \# \text{ys}) = (x, y) \# \text{zip} \: \text{xs} \: \text{ys}
by simp

declare zip-Cons [simp del]

lemma [code]:
zip [] \: \text{ys} = []
zip \: \text{xs} \: [] = []
zip (x \# \text{xs}) \: (y \# \text{ys}) = (x, y) \# \text{zip} \: \text{xs} \: \text{ys}
by (fact zip-Nil zip.simps(1) zip-Cons-Cons)+

lemma zip-Cons1:
zip (x\#xs) \: \text{ys} = (\text{case} \: \text{ys} \: \text{of} \: [] \Rightarrow [] \mid y\#\text{ys} \Rightarrow (x, y)\#\text{zip} \: \text{xs} \: \text{ys})
by (auto split:list.split)

lemma length-zip [simp]:
length (\text{zip} \: \text{xs} \: \text{ys}) = \text{min} \: (\text{length} \: \text{xs}) \: (\text{length} \: \text{ys})
by (induct \: \text{xs} \: \text{ys} \: \text{rule: list-induct2'}) \: \text{auto}

lemma zip-obtain-same-length:
assumes \( \forall zs \ ws \ n. \ \text{length} \ zs = \text{length} \ ws \ \Rightarrow \ n = \min (\text{length} \ zs) (\text{length} \ ys) \)
\[ \quad \Rightarrow \ zs = \text{take} \ n \ xs \ \Rightarrow \ ws = \text{take} \ n \ ys \ \Rightarrow \ P (\text{zip} \ zs \ ws) \]
shows \( P (\text{zip} \ xs \ ys) \)
proof –
  let \( \exists n = \min (\text{length} \ xs) (\text{length} \ ys) \)
  have \( P (\text{zip} (\text{take} \ ?n \ xs) (\text{take} \ ?n \ ys)) \)
    by (rule \( \text{assms} \)) \( \text{simp-all} \)
  moreover have \( \text{zip} \ zs \ ys = \text{zip} (\text{take} \ ?n \ xs) (\text{take} \ ?n \ ys) \)
    proof (induct \( \text{xs} \) arbitrary: \( \text{ys} \))
      case \( \text{Nil} \) then show \( ?\text{thesis} \) by simp
    next
    case \( \text{Cons} \ x \ xs \) then show \( ?\text{thesis} \) using \( \text{cases} \) \( \text{ys} \) \( \text{simp-all} \)
    qed
  ultimately show \( ?\text{thesis} \) by simp
qed

lemma \( \text{zip-append1} \):
  \( \text{zip} (\text{xs} \ @ \ \text{ys}) \ zs = \text{zip} \ \text{xs} (\text{take} (\text{length} \ \text{xs}) \ zs) \ \text{zips} \ \text{ys} (\text{drop} (\text{length} \ \text{xs}) \ zs) \)
  by (induct \( \text{xs} \) \( \text{zs} \) rule: \( \text{list\-induct2} \) ) \( \text{auto} \)

lemma \( \text{zip-append2} \):
  \( \text{zip} \ \text{xs} (\text{ys} \ @ \ \text{zs}) = \text{zip} (\text{take} (\text{length} \ \text{ys}) \ \text{xs}) \ \text{ys} \ \text{zips} \ \text{drop} (\text{length} \ \text{ys}) \ \text{xs} \ \text{zs} \)
  by (induct \( \text{xs} \) \( \text{ys} \) rule: \( \text{list\-induct2} \) ) \( \text{auto} \)

lemma \( \text{zip-append} [\text{simp}] \):
  \[ \| \text{length} \ \text{xs} = \text{length} \ \text{us} \ | \ | \Rightarrow \]
  \( \text{zip} (\text{xs} \ @ \ \text{ys}) (\text{us} \ @ \ \text{vs}) = \text{zip} \ \text{xs} \ \text{us} \ \text{zips} \ \text{ys} \ \text{vs} \)
  by (simp add: \( \text{zip\-append1} \) )

lemma \( \text{zip-rev} \):
  \( \text{length} \ \text{xs} = \text{length} \ \text{ys} \ | | \Rightarrow \text{zip} (\text{rev} \ \text{xs}) (\text{rev} \ \text{ys}) = \text{rev} (\text{zip} \ \text{xs} \ \text{ys}) \)
  by (induct rule: \( \text{list\-induct2} \), \( \text{simp-all} \) )

lemma \( \text{zip-map-map} \):
  \( \text{zip} (\text{map} \ f \ \text{xs}) (\text{map} \ g \ \text{ys}) = \text{map} (\lambda (x, y). (f x, g y)) (\text{zip} \ \text{xs} \ \text{ys}) \)
proof (induct \( \text{xs} \) arbitrary: \( \text{ys} \))
  case \( \text{Cons} \ x \ xs \) note \( \text{Cons-x-xs} = \text{Cons.hyps} \)
  show \( ?\text{thesis} \)
    proof (cases \( \text{ys} \))
      case \( \text{Cons} \ y \ ys' \)
      show \( ?\text{thesis} \) unfolding \( \text{Cons} \) using \( \text{Cons-x-xs} \) \( \text{by simp} \)
      qed simp
    qed simp
  qed simp

lemma \( \text{zip-map1} \):
  \( \text{zip} (\text{map} \ f \ \text{xs}) \ \text{ys} = \text{map} (\lambda(x, y). (f x, y)) (\text{zip} \ \text{xs} \ \text{ys}) \)
  using \( \text{zip-map-map} [\text{of} \ f \ \text{xs} \ \lambda x. \ x \ \text{ys}] \) \( \text{by simp} \)
lemma zip-map2:
zip xs (map f ys) = map (λ(x, y). (x, f y)) (zip xs ys)
using zip-map-map[of λx. x xs f ys] by simp

lemma map-zip-map:
map f (zip (map g xs) ys) = map (%(x, y). f (g x, y)) (zip xs ys)
by (auto simp: zip-map1)

lemma map-zip-map2:
map f (zip xs (map g ys)) = map (%(x, y). f (x, g y)) (zip xs ys)
by (auto simp: zip-map2)

Courtesy of Andreas Lochbihler:
lemma zip-same-conv-map:
zip xs xs = map (λx. (x, x)) xs
by (induct xs) auto

lemma nth-zip:
[i | i < length xs; i < length ys] ==> (zip xs ys)!i = (xs!i, ys!i)
proof (induct ys arbitrary: i xs)
  case (Cons y ys)
  then show ?case by (cases xs, simp-all add: nth.simps split: nat.split)
qed auto

lemma set-zip:
set (zip xs ys) = { (xs!i, ys!i) | i. i < min (length xs) (length ys) }
by (simp add: set-conv-nth cong: rev-conj-cong)

lemma zip-same:
((a, b) ∈ set (zip xs xs)) = (a ∈ set xs ∧ a = b)
by (induct xs) auto

lemma zip-update:
zip (xs[i:=x]) (ys[i:=y]) = (zip xs ys)[i:=(x,y)]
by (simp add: update-zip)

lemma zip-replicate:
zip (replicate i x) (replicate j y) = replicate (min i j) (x, y)
proof (induct i arbitrary: j)
  case (Suc i)
  then show ?case by (cases j, auto)
qed auto

lemma zip-replicate1:
zip (replicate n x) ys = map (Pair x) (take n ys)
by (induction ys arbitrary: n)(case-tac [2] n, simp-all)

lemma take-zip:
take n (zip xs ys) = zip (take n xs) (take n ys)
proof (induct n arbitrary: xs ys)
case 0
then show \(?case\) by simp
next
case Suc
then show \(?case\) by (cases \(xs\); cases \(ys\)) simp-all
qed

lemma \(\text{drop-zip}\) \(\text{drop } n \ (\text{zip } xs \ ys) = \text{zip } (\text{drop } n \ xs) \ (\text{drop } n \ ys)\)
proof (induct \(n\) arbitrary: \(xs\) \(ys\))
case 0
then show \(?case\) by simp
next
case Suc
then show \(?case\) by (cases \(xs\); cases \(ys\)) simp-all
qed

lemma \(\text{zip-takeWhile-fst}\): \(\text{zip } (\text{takeWhile } P \ xs) \ ys = \text{takeWhile } (P \circ \text{fst}) \ (\text{zip } xs \ ys)\)
proof (induct \(xs\) arbitrary: \(ys\))
case Nil
then show \(?case\) by simp
next
case Cons
then show \(?case\) by (cases \(ys\)) auto
qed

lemma \(\text{zip-takeWhile-snd}\): \(\text{zip } xs \ (\text{takeWhile } P \ ys) = \text{takeWhile } (P \circ \text{snd}) \ (\text{zip } xs \ ys)\)
proof (induct \(xs\) arbitrary: \(ys\))
case Nil
then show \(?case\) by simp
next
case Cons
then show \(?case\) by (cases \(ys\)) auto
qed

lemma \(\text{set-zip-leftD}\): \((x,y) \in \text{set } (\text{zip } xs \ ys) \Rightarrow x \in \text{set } xs\)
by (induct \(xs\) \(ys\) rule: list-induct2') auto

lemma \(\text{set-zip-rightD}\): \((x,y) \in \text{set } (\text{zip } xs \ ys) \Rightarrow y \in \text{set } ys\)
by (induct \(xs\) \(ys\) rule: list-induct2') auto

lemma \(\text{in-set-zipE}\):
\((x,y) \in \text{set}(\text{zip } xs \ ys) \Rightarrow (\{ x \in \text{set } xs; y \in \text{set } ys \} \Rightarrow R) \Rightarrow R\)
by (blast dest: set-zip-leftD set-zip-rightD)

lemma \(\text{zip-map-fst-snd}\): \(\text{zip } (\text{map } \text{fst} \ zs) \ (\text{map } \text{snd} \ zs) = zs\)
by (induct \(zs\)) simp-all

lemma \(\text{zip-eq-conv}\):
length xs = length ys → zip xs ys = zs ←→ map fst zs = xs ∧ map snd zs = ys
by (auto simp add: zip-map-fst-snd)

lemma in-set-zip:
  p ∈ set (zip xs ys) ←→ (∃ n. xs ! n = fst p ∧ ys ! n = snd p ∧ n < length xs ∧ n < length ys)
by (cases p) (auto simp add: set-zip)

lemma in-set-impl-in-set-zip1:
assumes length xs = length ys
assumes x ∈ set xs
obtains y where (x, y) ∈ set (zip xs ys)
proof –
  from assms have x ∈ set (map fst (zip xs ys)) by simp
  from this that show ?thesis by fastforce
qed

lemma in-set-impl-in-set-zip2:
assumes length xs = length ys
assumes y ∈ set ys
obtains x where (x, y) ∈ set (zip xs ys)
proof –
  from assms have y ∈ set (map snd (zip xs ys)) by simp
  from this that show ?thesis by fastforce
qed

lemma zip-eq-Nil-iff:
zip xs ys = [] ←→ xs = [] ∨ ys = []
by (cases xs; cases ys) simp-all

lemma zip-eq-ConsE:
assumes zip xs ys = xy # xys
obtains x xs' y ys' where xs = x # xs'
  and ys = y # ys' and xy = (x, y)
  and xys = zip xs' ys'
proof –
  from assms have xs ≠ [] and ys ≠ []
    using zip-eq-Nil-iff [of xs ys] by simp-all
  then obtain x xs' y ys' where xs: xs = x # xs'
    and ys: ys = y # ys'
    by (cases xs; cases ys) auto
  with assms have xy = (x, y) and xys = zip xs' ys'
    by simp-all
  with xs ys show ?thesis ..
qed

lemma semilattice-map2:
semilattice (map2 (•)) if semilattice (•)
  for f (infixl • 70)
proof
  from that interpret semilattice \( f \).
  show \(?\)thesis
proof
  show \( \text{map} 2 \, (\ast) \, (\text{map} 2 \, (\ast) \, x s \, y s) \, z s = \text{map} 2 \, (\ast) \, x s \, (\text{map} 2 \, (\ast) \, y s \, z s) \)
    for \( x s \, y s \, z s :: 'a \text{ list} \)
  proof (induction \( \text{zip} \, x s \, (\text{zip} \, y s \, z s) \) arbitrary: \( x s \, y s \, z s \))
    case Nil
    from Nil [symmetric] show \(?\)case
      by (auto simp add: \( \text{zip-eq-Nil-iff} \))
  next
    case (Cons \( x y z \, x y z s \))
    from Cons . hyps (2) [symmetric] show \(?\)case
      by (rule \( \text{zip-eq-ConsE} \)) (erule \( \text{zip-eq-ConsE} \), auto intro: Cons . hyps (1) simp add: \( \text{ac-simps} \))
  qed
  show \( \text{map} 2 \, (\ast) \, x s \, y s = \text{map} 2 \, (\ast) \, y s \, x s \)
    for \( x s \, y s :: 'a \text{ list} \)
  proof (induction \( \text{zip} \, x s \, y s \) arbitrary: \( x s \, y s \))
    case Nil
    then show \(?\)case
      by (auto simp add: \( \text{zip-eq-Nil-iff} \) dest: sym)
  next
    case (Cons \( x y x y s \))
    from Cons . hyps (2) [symmetric] show \(?\)case
      by (rule \( \text{zip-eq-ConsE} \)) (auto intro: Cons . hyps (1) simp add: \( \text{ac-simps} \))
  qed
  show \( \text{map} 2 \, (\ast) \, x s \, x s = x s \)
    for \( x s :: 'a \text{ list} \)
  by (induction \( x s \) simp-all)
  qed
  qed

lemma pair-list-eqI:
  assumes \( \text{map} \, \text{fst} \, x s = \text{map} \, \text{fst} \, y s \) and \( \text{map} \, \text{snd} \, x s = \text{map} \, \text{snd} \, y s \)
  shows \( x s = y s \)
proof
  from assms (1) have \( \text{length} \, x s = \text{length} \, y s \) by (rule \( \text{map-eq-imp-length-eq} \))
  from this assms show \(?\)thesis
    by (induct \( x s \, y s \) rule: list-induct2) (simp-all add: \( \text{prod-eqI} \))
  qed

lemma hd-zip:
  \( \langle \text{hd} \, (\text{zip} \, x s \, y s) \rangle = (\text{hd} \, x s, \text{hd} \, y s) \) if \( x s \neq [] \) and \( y s \neq [] \)
  using that by (cases \( x s \); cases \( y s \)) simp-all

lemma last-zip:
  \( \langle \text{last} \, (\text{zip} \, x s \, y s) \rangle = (\text{last} \, x s, \text{last} \, y s) \) if \( x s \neq [] \) and \( y s \neq [] \)
  and \( \langle \text{length} \, x s = \text{length} \, y s \rangle \)
using that by (cases xs rule: rev-cases; cases ys rule: rev-cases) simp-all

66.1.17 list-all2

lemma list-all2-lengthD [intro?):
list-all2 P xs ys ===> \( \text{length } xs = \text{length } ys \)
by (simp add: list-all2-iff)

lemma list-all2-Nil [iff, code]: list-all2 P [] ys = (ys = [])
by (simp add: list-all2-iff)

lemma list-all2-Nil2 [iff, code]: list-all2 P xs [] = (xs = [])
by (simp add: list-all2-iff)

lemma list-all2-Cons [iff, code]:
list-all2 P (x # xs) (y # ys) = (P x y ∧ list-all2 P xs ys)
by (auto simp add: list-all2-iff)

lemma list-all2-Cons1:
list-all2 P (x # xs) ys = (∃ zs. ys = z # zs ∧ P x z ∧ list-all2 P zs ys)
by (cases ys) auto

lemma list-all2-Cons2:
list-all2 P xs (y # ys) = (∃ zs. xs = z # zs ∧ P z y ∧ list-all2 P zs ys)
by (cases xs) auto

lemma list-all2-induct [consumes 1, case-names Nil Cons, induct set: list-all2]:
assumes P: list-all2 P xs ys
assumes Nil: R [] []
assumes Cons: \( \forall x s y. \)
\( [P x y; list-all2 P xs ys; R xs ys] \implies R (x # xs) (y # ys) \)
shows R xs ys
using P
by (induct xs arbitrary: ys) (auto simp add: list-all2-Cons1 Nil Cons)

lemma list-all2-rev [iff]:
list-all2 P (rev xs) (rev ys) = list-all2 P xs ys
by (simp add: list-all2-iff zip-rev cong: conj-cong)

lemma list-all2-rev1:
list-all2 P (rev xs) ys = list-all2 P xs (rev ys)
by (subst list-all2-rev [symmetric]) simp

lemma list-all2-append1:
list-all2 P (xs @ ys) zs =
(∃ us vs. zs = us @ vs ∧ length us = length xs ∧ length vs = length ys ∧
list-all2 P xs us ∧ list-all2 P ys vs) is ?lhs = ?rhs
proof
assume ?lhs
then show ?rhs
  apply (rule-tac x = take (length xs) zs in exI)
  apply (rule-tac x = drop (length xs) zs in exI)
  apply (force split: nat-diff-split simp add: list-all2-iff zip-append1)
  done
next
assume ?rhs
then show ?lhs
  by (auto simp add: list-all2-iff)
qed

lemma list-all2-append2:
  list-all2 P xs (ys @ zs) =
  (∃ us vs. xs = us @ vs ∧ length us = length ys ∧ length vs = length zs ∧
  list-all2 P us ys ∧ list-all2 P vs zs) (is ?lhs = ?rhs)
proof
  assume ?lhs
  then show ?rhs
    apply (rule-tac x = take (length ys) xs in exI)
    apply (rule-tac x = drop (length ys) xs in exI)
    apply (force split: nat-diff-split simp add: list-all2-iff zip-append2)
    done
next
assume ?rhs
then show ?lhs
  by (auto simp add: list-all2-iff)
qed

lemma list-all2-append:
  length xs = length ys \implies
  list-all2 P (xs@us) (ys@vs) = (list-all2 P xs ys ∧ list-all2 P us vs)
by (induct rule: list-induct2, simp-all)

lemma list-all2-appendI [intro?, trans]:
  [ list-all2 P a b; list-all2 P c d ] \implies list-all2 P (a@c) (b@d)
by (simp add: list-all2-append list-all2-lengthD)

lemma list-all2-conv-all-nth:
  list-all2 P xs ys =
  (length xs = length ys ∧ (∀ i < length xs. P (xs!i) (ys!i)))
by (force simp add: list-all2-iff set-zip)

lemma list-all2-trans:
  assumes tr: !!a b c. P1 a b ==> P2 b c ==> P3 a c
  shows !!bs cs. list-all2 P1 as bs ==> list-all2 P2 bs cs ==> list-all2 P3 as cs
    (is !!bs cs. PROP ?Q as bs cs)
proof (induct as)
  fix x xs bs assume l1: !!bs cs. PROP ?Q xs bs cs
show \(!cs.\) PROP ?Q (x \# xs) bs cs
proof (induct bs)
  fix y ys cs assume I2: \(!cs.\) PROP ?Q (x \# xs) ys cs
  show PROP ?Q (x \# xs) (y \# ys) cs
    by (induct cs) (auto intro: tr H1 I2)
qed simp

lemma list-all2-all-nthI [intro?]:
  length a = length b \implies (\forall n. n < length a \implies P (a!n) (b!n)) \implies list-all2 P a b
  by (simp add: list-all2-conv-all-nth)

lemma list-all2I:
\forall x \in set (zip a b). case-prod P x \implies length a = length b \implies list-all2 P a b
  by (simp add: list-all2-iff)

lemma list-all2-nthD:
  [[ list-all2 P xs ys; p < size xs ]] \implies P (xs!p) (ys!p)
by (simp add: list-all2-conv-all-nth)

lemma list-all2-nthD2:
  [[ list-all2 P xs ys; p < size ys ]] \implies P (xs!p) (ys!p)
  by (frule list-all2-lengthD) (auto intro: list-all2-nthD)

lemma list-all2-map1:
  list-all2 P (map f as) bs = list-all2 (\lambda x y. P (f x) y) as bs
  by (simp add: list-all2-conv-all-nth)

lemma list-all2-map2:
  list-all2 P as (map f bs) = list-all2 (\lambda x y. P x (f y)) as bs
  by (auto simp add: list-all2-conv-all-nth)

lemma list-all2-refl [intro?):
  (\forall x. P x x) \implies list-all2 P xs xs
  by (simp add: list-all2-conv-all-nth)

lemma list-all2-update-cong:
  [[ list-all2 P xs ys; P x y ]] \implies list-all2 P (xs[i:=x]) (ys[i:=y])
  by (cases i < length ys) (auto simp add: list-all2-conv-all-nth nth-list-update)

lemma list-all2-takeI [simp,intro?):
  list-all2 P xs ys \implies list-all2 P (take n xs) (take n ys)
proof (induct xs arbitrary: n ys)
  case (Cons x xs)
  then show ?case
    by (cases n) (auto simp: list-all2-Cons1)
qed auto

lemma list-all2-dropI [simp,intro?):
list-all2 P xs ys \implies list-all2 P (\text{drop } n \text{ } xs) (\text{drop } n \text{ } ys)

\textbf{proof (induct } xs \text{ arbitrary: } n \text{ } ys)\)

\textbf{case (Cons } x \text{ } xs)\)

\textbf{then show } \text{?case} \text{ by (cases } n \text{ ) (auto simp: list-all2-Cons1)}

\textbf{qed auto}

\textbf{lemma list-all2-mono [intro?]:}

\text{list-all2 } P \text{ } xs \text{ } ys \implies (\forall \text{ } xs \text{ } ys. \text{ } P \text{ } xs \text{ } ys \implies Q \text{ } xs \text{ } ys) \implies \text{list-all2 } Q \text{ } xs \text{ } ys

\textbf{by (rule list.rel-mono-strong)}

\textbf{lemma list-all2-eq:}

\text{xs } = \text{ys } \longleftrightarrow \text{list-all2 } (=) \text{ } xs \text{ } ys

\textbf{by (auto simp add: list-all2-eq list-all2-conv-all-nth cong: conj-cong)}

\textbf{lemma list-all2-same:}

\text{list-all2 } P \text{ } xs \text{ } xs \longleftrightarrow (\forall \text{ } x \in \text{set } xs. \text{ } P \text{ } x \text{ } x)

\textbf{by (auto simp add: list-all2-conv-all-nth set-conv-nth)}

\textbf{lemma zip-assoc:}

\text{zip } xs \text{ } (\text{zip } ys \text{ } zs) = \text{map } (\lambda (x, \text{ } y). \text{ } (x, \text{ } y, z)) \text{ (zip } xs \text{ } ys \text{ ) } zs

\textbf{by (rule list-all2-all-nthI[where } P=\text{=}\text{, unfolded list.rel-eq]) simp-all}

\textbf{lemma zip-commute:}

\text{zip } xs \text{ } ys = \text{map } (\lambda(x, \text{ } y). \text{ } (y, \text{ } x)) \text{ (zip } ys \text{ } xs \text{ )}

\textbf{by (rule list-all2-all-nthI[where } P=\text{=}\text{, unfolded list.rel-eq]) simp-all}

\textbf{lemma zip-left-commute:}

\text{zip } xs \text{ } (\text{zip } ys \text{ } zs) = \text{map } (\lambda(y, \text{ } (x, \text{ } z)). \text{ } (x, \text{ } y, z)) \text{ (zip } ys \text{ } (zip } xs \text{ ) } zs

\textbf{by (rule list-all2-all-nthI[where } P=\text{=}\text{, unfolded list.rel-eq]) simp-all}

\textbf{lemma zip-replicate2:}

\text{zip } xs \text{ } (\text{replicate } n \text{ } y) = \text{map } (\lambda x. \text{ } (x, \text{ } y)) \text{ (take } n \text{ } xs)

\textbf{by (subst zip-commute)(simp add: zip-replicate1)}

\textbf{66.1.18 List.product and product-lists}

\textbf{lemma product-concat-map:}

\text{List.product } xs \text{ } ys = \text{concat } (\text{map } (\lambda x. \text{ map } (\lambda y. \text{ } (x, y))) \text{ } ys) \text{ } xs

\textbf{by (induction } xs \text{ ) (simp)+}

\textbf{lemma set-product[simp]:}

\text{set } (\text{List.product } xs \text{ } ys) = \text{set } xs \times \text{set } ys

\textbf{by (induct } xs \text{ ) auto}

\textbf{lemma length-product [simp]:}

\text{length } (\text{List.product } xs \text{ } ys) = \text{length } xs \times \text{length } ys

\textbf{by (induct } xs \text{ ) simp-all}
lemma product-nth:
  assumes n < length xs * length ys
  shows List.product xs ys ! n = (xs ! (n div length ys), ys ! (n mod length ys))
  using assms proof (induct xs arbitrary: n)
    case Nil then show ?case by simp
  next
    case (Cons x xs n)
    then have length ys > 0 by auto
    with Cons show ?case by (auto simp add: nth-append not-less le-mod-geq le-div-geq)
  qed

lemma in-set-product-lists-length:
  xs ∈ set (product-lists xss) ⇒ length xs = length xss
  by (induct xss arbitrary: xs) auto

lemma product-lists-set:
  set (product-lists xss) = {xs. list-all2 (λx ys. x ∈ set ys) xs xss}
  (is ?L = Collect ?R)
  proof (intro equalityI subsetI, unfold mem-Collect-eq)
    fix xs assume xs ∈ ?L
    then have length xs = length xss by (rule in-set-product-lists-length)
    from this ⟨xs ∈ ?L⟩ show ?R xs by (induct xs xss rule: list-induct2) auto
  next
    fix xs assume ?R xs
    then show xs ∈ ?L by induct auto
  qed

66.1.19 fold with natural argument order

lemma fold-simps [code]: — eta-expanded variant for generated code – enables
tail-recursion optimisation in Scala
  fold f [] s = s
  fold f (x # xs) s = fold f xs (f x s)
  by simp-all

lemma fold-remove1-split:
  [ ∀ y. x ∈ set xs ⇒ y ∈ set xs ⇒ f x o f y = f y o f x;
    x ∈ set xs ]
  ⇒ fold f xs = fold f (remove1 x xs) o f x
  by (induct xs) (auto simp add: comp-assoc)

lemma fold-cong [fundef-cong]:
  a = b ⇒ xs = ys ⇒ (∀x. x ∈ set xs ⇒ f x = g x)
  ⇒ fold f xs a = fold g ys b
  by (induct ys arbitrary: a b xs) simp-all

lemma fold-id: (∀x. x ∈ set xs ⇒ f x = id) ⇒ fold f xs = id
  by (induct xs) simp-all
lemma fold-commute:
\( (\forall x. x \in \text{set} \text{xs} \implies h \circ g x = f x \circ h) \implies h \circ \text{fold} g \text{xs} = \text{fold} f \text{xs} \circ h \)  
by (induct \text{xs}) (simp-all add: fun-eq-iff)

lemma fold-commute-apply:
assumes \( \forall x. x \in \text{set} \text{xs} \implies h \circ g x = f x \circ h \)
shows \( h \circ \text{fold} g \text{xs} = \text{fold} f \text{xs} \circ h \)
proof –  
from assms have \( h \circ \text{fold} g \text{xs} = \text{fold} f \text{xs} \circ h \) by (rule fold-commute)
then show \( ?\)thesis by (simp add: fun-eq-iff)
qed

lemma fold-invariant:
\[ \begin{align*}
\forall x. x \in \text{set} \text{xs} \implies Q x; \\
P s; \quad \forall x s. Q x \implies P s \implies P (f x s) \end{align*} \]
\( \implies P (\text{fold} f \text{xs} s) \)  
by (induct \text{xs} arbitrary: \( s \)) simp-all

lemma fold-append [simp]: \( \text{fold} f (\text{xs} @ \text{ys}) = \text{fold} f \text{ys} \circ \text{fold} f \text{xs} \)
by (induct \text{xs}) simp-all

lemma fold-map [code-unfold]: \( \text{fold} g (\text{map} f \text{xs}) = \text{fold} (g \circ f) \text{xs} \)
by (induct \text{xs}) simp-all

lemma fold-filter:
\( \text{fold} f (\text{filter} P \text{xs}) = \text{fold} (\lambda x. \text{if} P x \text{ then} f x \text{ else} \text{id}) \text{xs} \)
by (induct \text{xs}) simp-all

lemma fold-rev:
\( \forall x y. x \in \text{set} \text{xs} \implies y \in \text{set} \text{xs} \implies f y \circ f x = f x \circ f y \)
\( \implies \text{fold} f (\text{rev} \text{xs}) = \text{fold} f \text{xs} \)  
by (induct \text{xs}) (simp-all add: fold-commute-apply fun-eq-iff)

lemma fold-Cons-rev: \( \text{fold} \ \text{Cons} \text{xs} = \text{append} (\text{rev} \text{xs}) \)
by (induct \text{xs}) simp-all

lemma rev-conv-fold [code]: \( \text{rev} \text{xs} = \text{fold} \ \text{Cons} \text{xs} [] \)
by (simp add: fold-Cons-rev)

lemma fold-append-concat-rev: \( \text{fold} \ \text{append} \text{xss} = \text{append} (\text{concat} (\text{rev} \text{xss})) \)
by (induct \text{xss}) simp-all

Finite-Set.fold and fold

lemma (in comp-fun-commute) fold-set-fold-remdups:
Finite-Set.fold \( f y \) (set \text{xs}) = \( \text{fold} f \) (remdups \text{xs}) \( y \)  
by (rule sym, induct \text{xs} arbitrary: \( y \)) (simp-all add: fold-fun-left-comm insert-absorb)

lemma (in comp-fun-idem) fold-set-fold:
Finite-Set.fold \( f y \) (set \text{xs}) = \( \text{fold} f \text{xs} \) \( y \)
by (rule sym, induct xs arbitrary: y) (simp-all add: fold-fun-left-comm)

lemma union-set-fold [code]: set xs ∪ A = fold Set.insert xs A
proof –
  interpret comp-fun-idem Set.insert
  by (fact comp-fun-idem-insert)
  show ??thesis by (simp add: union-fold-insert fold-set-fold)
qed

lemma union-coset-filter [code]:
  List.coset xs ∪ A = List.coset (List.filter (λx. x ∉ A) xs)
by auto

lemma minus-set-fold [code]: A − set xs = fold Set.remove xs A
proof –
  interpret comp-fun-idem Set.remove
  by (fact comp-fun-idem-remove)
  show ??thesis by (simp add: minus-fold-remove [of -A] fold-set-fold)
qed

lemma minus-coset-filter [code]:
  A − List.coset xs = set (List.filter (λx. x ∈ A) xs)
by auto

lemma inter-set-filter [code]:
  A ∩ set xs = set (List.filter (λx. x ∈ A) xs)
by auto

lemma inter-coset-fold [code]:
  A ∩ List.coset xs = fold Set.remove xs A
by (simp add: Diff-eq [symmetric] minus-set-fold)

lemma (in semilattice-set) set-eq-fold [code]:
  F (set (x ≠ xs)) = fold f xs x
proof –
  interpret comp-fun-idem f
  by standard (simp-all add: fun-eq-iff left-commute)
  show ??thesis by (simp add: eq-fold fold-set-fold)
qed

lemma (in complete-lattice) Inf-set-fold:
  Inf (set xs) = fold inf xs top
proof –
  interpret comp-fun-idem inf :: 'a ⇒ 'a ⇒ 'a
  by (fact comp-fun-idem-inf)
  show ??thesis by (simp add: Inf-fold-inf fold-set-fold inf-commute)
qed
declare Inf-set-fold [where 'a = 'a set, code]

lemma (in complete-lattice) Sup-set-fold:
  Sup (set xs) = fold sup xs bot
proof -
  interpret comp-fun-idem sup :: 'a => 'a => 'a
  by (fact comp-fun-idem-sup)
  show ?thesis by (simp add: Sup-fold-sup fold-set-fold sup-commute)
qed

declare Sup-set-fold [where 'a = 'a set, code]

lemma (in complete-lattice) INF-set-fold:
  Inf (f ' set xs) = fold (inf ◦ f) xs top
using Inf-set-fold[of map f xs] by (simp add: fold-map)

lemma (in complete-lattice) SUP-set-fold:
  INF (f ' set xs) = fold (sup ◦ f) xs bot
using Sup-set-fold[of map f xs] by (simp add: fold-map)

66.1.20 Fold variants: foldr and foldl

Correspondence

lemma foldr-conv-fold [code-abbrev]: foldr f xs = fold f (rev xs)
by (induct xs) simp-all

lemma foldl-conv-fold: foldl f s xs = fold (λx s. f s x) xs s
by (induct xs arbitrary: s) simp-all

lemma foldr-conv-foldl: — The “Third Duality Theorem” in Bird & Wadler:
  foldr f xs a = foldl (λx y. f y x) a (rev xs)
by (simp add: foldr-conv-fold foldl-conv-fold)

lemma foldl-conv-foldr:
  foldl f a xs = foldr (λx y. f y x) (rev xs) a
by (simp add: foldr-conv-fold foldl-conv-fold)

lemma foldr-fold:
  (∀x y. x ∈ set xs =⇒ y ∈ set xs =⇒ f y o f x = f x o f y)
  =⇒ foldr f xs = fold f xs
unfolding foldr-conv-fold by (rule fold-rev)

lemma foldr-cong [fundef-cong]:
  a = b =⇒ l = k =⇒ (∀x a. x ∈ set l =⇒ f x a = g x a) =⇒ foldr f l a = foldr g k b
by (auto simp add: foldr-conv-fold intro!: foldl-cong)

lemma foldl-cong [fundef-cong]:
  a = b =⇒ l = k =⇒ (∀x a. x ∈ set l =⇒ f a x = g a x) =⇒ foldl f a l = foldl
\begin{verbatim}

g b k
  by (auto simp add: foldl-conv-fold intro: fold-cong)

lemma foldr-append [simp]: foldr f (xs @ ys) a = foldr f xs (foldr f ys a)
  by (simp add: foldr-conv-fold)

lemma foldl-append [simp]: foldl f a (xs @ ys) = foldl f (foldl f a xs) ys
  by (simp add: foldl-conv-fold)

lemma foldr-map [code-unfold]: foldr g (map f xs) a = foldr (g ◦ f) xs a
  by (simp add: foldr-conv-fold fold-map rev-map)

lemma foldr-filter: foldr f (filter P xs) = foldr (λx. if P x then f x else id) xs
  by (simp add: foldr-conv-fold rev-filter fold-filter)

lemma foldl-map [code-unfold]: foldl g a (map f xs) = foldl (λa x. g a (f x)) a xs
  by (simp add: foldl-conv-fold fold-map comp-def)

lemma concat-conv-foldr: concat xss = foldr append xss []
  by (simp add: fold-append-concat-rev foldr-conv-fold)

66.1.21 upt

lemma upt-rec [code]: [i..<j] = (if i<j then i#[Suc i..<j] else [])
  — simp does not terminate!
  by (induct j) auto

lemmas upt-rec-numeral [simp] = upt-rec [of numeral m numeral n] for m n

lemma upt-conv Nil [simp]: j ≤ i ==> [i..<j] = []
  by (subst upt-rec) simp

lemma upt-eq-nil-conv [simp]: ([i..<j] = []) = (j = 0 ∨ j ≤ i)
  by (induct j) simp-all

lemma upt-eq-Cons-conv:
  ([i..<j] = x#xs) = (i < j ∧ i = x ∧ [i+1..<j] = xs)
proof (induct j arbitrary: x xs)
  case (Suc j)
  then show ?case
  by (simp add: upt-rec)
qed simp

lemma upt-Suc-append: i ≤ j ==> [i..<(Suc j)] = [i..<j]@[j]
  — Only needed if upt-Suc is deleted from the simpset.
  by simp
\end{verbatim}
lemma upt-conv-Cons: \( i < j \Rightarrow [i..<j] = i \# [Suc i..<j] \)
by (simp add: upt-rec)

lemma upt-conv-Cons-Cons: — no precondition
\( m \# n \# ns = [m..<q] \Leftarrow\Rightarrow n \# ns = [Suc m..<q] \)
proof (cases \( m < q \))
  case False then show \(?thesis\) by simp
next
  case True then show \(?thesis\) by (simp add: upt-conv-Cons)
qed

lemma upt-add-eq-append: \( i < j \Rightarrow [i..<j]+k = [i..<j]\@[j..<j+k] \)
— LOOPS as a simprule, since \( j \leq j \).
by (induct \( k \)) auto

lemma length-upt [simp]: \( \text{length} [i..<j] = j - i \)
by (induct \( j \)) (auto simp add: Suc-diff-le)

lemma nth-upt [simp]: \( i + k < j \Rightarrow [i..<j] ! k = i + k \)
by (induct \( j \)) (auto simp add: less-Suc-eq nth-append split: nat-diff-split)

lemma hd-upt [simp]: \( i < j \Rightarrow \text{hd} [i..<j] = i \)
by (simp add: upt-conv-Cons)

lemma tl-upt [simp]: \( \text{tl} [m..<n] = [Suc m..<n] \)
by (simp add: upt-rec)

lemma last-upt [simp]: \( i < j \Rightarrow \text{last} [i..<j] = j - 1 \)
by (cases \( j \)) (auto simp: upt-Suc-append)

lemma take-upt [simp]: \( i + m \leq n \Rightarrow \text{take} m [i..<n] = [i..<i+m] \)
proof (induct \( m \) arbitrary: \( i \))
  case Suc \( m \)
  then show \(?case\) by (subst take-Suc-conv-app-nth) auto
qed simp

lemma drop-upt [simp]: \( \text{drop} m [i..<j] = [i+m..<j] \)
by (induct \( j \)) auto

lemma map-Suc-upt: \( \text{map} \ Suc [m..<n] = [Suc m..<n] \)
by (induct \( n \)) auto

lemma map-add-upt: \( \text{map} (\lambda i. i + n) [0..<m] = [n..<m + n] \)
by (induct \( m \)) simp-all

lemma nth-map-upt: \( i < n-m \Rightarrow (\text{map} f [m..<n]) ! i = f(m+i) \)
proof (induct \( n \ m \) arbitrary: \( i \) rule: diff-induct)
case (3 x y)
then show ?case
  by (metis add.commute length-upt less-diff-conv nth-map nth-upt)
qed auto

lemma map-decr-upt: map (λn. n - Suc 0) [Suc m..<Suc n] = [m..<n]
by (induct n) simp-all

lemma map-upt-Suc: map f [0..<Suc n] = f 0 # map (λi. f (Suc i)) [0..<n]
by (induct n arbitrary: f) auto

lemma nth-take-lemma:
  k ≤ length xs =⇒ k ≤ length ys =⇒ ((∀i. i < k =⇒ xs!i = ys!i) =⇒ take k xs = take k ys
proof (induct k arbitrary: xs ys)
  case (Suc k)
  then show ?case
  apply (simp add: less-Suc-eq-0-disj)
  by (simp add: Suc.prems(3) take-Suc-conv-app-nth)
qed simp

lemma nth-equalityI:
  [length xs = length ys; (∀i. i < length xs =⇒ xs!i = ys!i)] =⇒ xs = ys
by (rule nth-take-lemma [OF le-refl eq-imp-le]) simp-all

lemma map-nth:
  map (λi. xs!i) [0..<length xs] = xs
by (rule nth-equalityI, auto)

lemma list-all2-antisym:
  [ (∀x y. [P x y; Q y x] =⇒ x = y); list-all2 P xs ys; list-all2 Q ys xs ]
  =⇒ xs = ys
by (simp add: list-all2-conv-all-nth nth-equalityI)

lemma take-equalityI: (∀i. take i xs = take i ys) =⇒ xs = ys
— The famous take-lemma.
by (metis length-take min.commute order-refl take-all)

lemma take-Cons':
  take n (x # xs) = (if n = 0 then [] else x # take (n - 1) xs)
by (cases n) simp-all

lemma drop-Cons':
  drop n (x # xs) = (if n = 0 then x # xs else drop (n - 1) xs)
by (cases n) simp-all

lemma nth-Cons': (x # xs)!n = (if n = 0 then x else xs!((n - 1)))
by (cases n) simp-all
lemma take-Cons-numeral [simp]:
  \(\text{take (numeral } v\) (x # xs) = x # \text{take (numeral } v - 1\) xs
by (simp add: take-Cons')

lemma drop-Cons-numeral [simp]:
  \(\text{drop (numeral } v\) (x # xs) = \text{drop (numeral } v - 1\) xs
by (simp add: drop-Cons')

lemma nth-Cons-numeral [simp]:
  \((x # xs)!\text{numeral } v = xs!\text{numeral } v - 1\)
by (simp add: nth-Cons')

66.1.22 upto: interval-list on int

function upto :: int ⇒ int ⇒ int list ((I[−/−])) where
  upto i j = (if i ≤ j then i # [i+1..j] else [])
by auto

termination
by (relation measure (%(i::int, j). nat (j - i + 1))) auto

declare upto.simps[simp del]

lemmas upto-rec-numeral [simp] =
  upto.simps[of numeral m numeral n]
  upto.simps[of numeral m - numeral n]
  upto.simps[of - numeral m numeral n]
  upto.simps[of - numeral m - numeral n] for m n

lemma upto-empty[simp]: j < i ⇒ [i..j] = []
by (simp add: upto.simps)

lemma upto-single[simp]: [i..i] = [i]
by (simp add: upto.simps)

lemma upto-Nil[simp]: [i..j] = [] ⇐ j < i
by (simp add: upto.simps)

lemma upto-Nil2[simp]: [] = [i..j] ⇐ j < i
by (simp add: upto.simps)

lemma upto-rec1: i ≤ j ⇒ [i..j] = i#[i+1..j]
by (simp add: upto.simps)

lemma upto-rec2: i ≤ j ⇒ [i..j] = [i..j - 1]@[j]
proof (induct nat (j - i) arbitrary: i j)
  case 0 thus ?case by (simp add: upto.simps)
next
case (Suc n)
hence n = nat (j - (i + 1)) i < j by linarith+
from this(2) Suc.hyps(1)[OF this(1)] Suc(2,3) upto-rec1 show ?case by simp
qed

lemma length upto[simp]: length [i..j] = nat(j - i + 1)
by(induction i j rule: upto.induct) (auto simp: upto.simps)

lemma set upto[simp]: set[i..j] = {i..j}
proof(induct i j rule: upto.induct)
  case (1 i j)
  from this show ?case unfolding upto.simps[of i j] by auto
qed

lemma nth upto[simp]: i + int k ≤ j =⇒ [i..j] ! k = i + int k
apply(induction i j arbitrary: k rule: upto.induct)
apply(subst upto-rec1)
apply(auto simp add: nth-Cons')
done

lemma upto-split1:
i ≤ j =⇒ j ≤ k =⇒ [i..k] = [i..j-1] @ [j..k]
proof (induction j rule: int-ge-induct)
  case base thus ?case by (simp add: upto-rec1)
next
  case step thus ?case using upto-rec1 upto-rec2 by simp
qed

lemma upto-split2:
i ≤ j =⇒ j ≤ k =⇒ [i..k] = [i..j] @ [j+1..k]
using upto-rec1 upto-rec2 upto-split1 by auto

lemma upto-split3: [ i ≤ j; j ≤ k ] =⇒ [i..k] = [i..j-1] @ j # [j+1..k]
using upto-rec1 upto-split1 by auto

Tail recursive version for code generation:
definition upto-aux :: int ⇒ int ⇒ int list ⇒ int list where
upto-aux i j js = [i..j] @ js

lemma upto-aux-rec [code]:
  upto-aux i j js = (if j<i then js else upto-aux i (j - 1) (j#js))
by (simp add: upto-aux-def upto-rec2)

lemma upto-code[code]: [i..j] = upto-aux i j []
by(simp add: upto-aux-def)

66.1.23 distinct and remdups and remdups-adj
lemma distinct-tl: distinct xs =⇒ distinct (tl xs)
by (cases xs) simp-all
lemma distinct-append [simp]:
  \( \text{distinct}(xs @ ys) = (\text{distinct} \; xs \land \text{distinct} \; ys \land \text{set} \; xs \cap \text{set} \; ys = \{\}) \)
by (induct xs) auto

lemma distinct-rev [simp]: \( \text{distinct}(\text{rev} \; xs) = \text{distinct} \; xs \)
by (induct xs) auto

lemma set-remdups [simp]: \( \text{set} \; (\text{remdups} \; xs) = \text{set} \; xs \)
by (induct xs) (auto simp add: insert-absorb)

lemma distinct-remdups [iff]: \( \text{distinct} \; (\text{remdups} \; xs) \)
by (induct xs) auto

lemma distinct-remdups-id: \( \text{distinct} \; xs \implies \text{remdups} \; xs = xs \)
by (induct xs) auto

lemma remdups-id-iff-distinct [simp]: \( \text{remdups} \; xs = xs \iff \text{distinct} \; xs \)
by (metis distinct-remdups finite-list set-remdups)

lemma remdups-eq-nil-iff [simp]: \( \text{remdups} \; x = [] \iff x = [] \)
by (induct x, auto)

lemma remdups-eq-nil-right-iff [simp]: \( [] = \text{remdups} \; x \iff x = [] \)
by (induct x, auto)

lemma length-remdups-leq [iff]: \( \text{length} \; (\text{remdups} \; xs) \leq \text{length} \; xs \)
by (induct xs) auto

lemma length-remdups-eq [iff]:
  \( \text{length} \; (\text{remdups} \; xs) = \text{length} \; xs \implies \text{remdups} \; xs = xs \)
proof (induct xs)
  case (Cons a xs)
  then show \(?case\)
    by simp (metis Suc-n-not-le-n impossible-Cons length-remdups-leq)
qed auto

lemma remdups-filter: \( \text{remdups}(\text{filter} \; P \; xs) = \text{filter} \; P \; (\text{remdups} \; xs) \)
by (induct xs) auto

lemma distinct-map:
  \( \text{distinct}(\text{map} \; f \; xs) = (\text{distinct} \; xs \land \text{inj-on} \; f \; (\text{set} \; xs)) \)
by (induct xs) auto

lemma distinct-map-filter:
  \( \text{distinct} \; (\text{map} \; f \; xs) \implies \text{distinct} \; (\text{map} \; f \; (\text{filter} \; P \; xs)) \)
by (induct xs) auto

lemma distinct-filter [simp]: distinct xs ==> distinct (filter P xs)
by (induct xs) auto

lemma distinct-upt [simp]: distinct[i..<j]
by (induct j) auto

lemma distinct-upto [simp]: distinct[i..j]
apply(induct j rule:upto.induct)
apply(subst upto.simps)
apply(simp)
done

lemma distinct-take [simp]: distinct xs ==> distinct (take i xs)
proof (induct xs arbitrary: i)
case (Cons a xs)
  then show ?case
    by (metis Cons.prems append-take-drop-id distinct-append)
qed auto

lemma distinct-drop [simp]: distinct xs ==> distinct (drop i xs)
proof (induct xs arbitrary: i)
case (Cons a xs)
  then show ?case
    by (metis Cons.prems append-take-drop-id distinct-append)
qed auto

lemma distinct-list-update:
  assumes d: distinct xs and a: a \notin set xs - {xs!i}
shows distinct (xs[i:=a])
proof (cases i < length xs)
case True
  with a have anot: a \notin set (take i xs @ xs ! i # drop (Suc i) xs) - {xs!i}
    by simp (metis in-set-dropD in-set-takeD)
  show ?thesis
proof (cases a = xs!i)
case True
  with d show ?thesis
    by auto
next
case False
  then show ?thesis
    using d anot (i < length xs)
  apply (simp add: upd-cone-take-nth-drop)
    by (metis disjoint-insert(1) distinct-append id-take-nth-drop set-simps(2))
qed
next
case False with d show ?thesis by auto
theory "List"

lemma distinct-concat:
\[
\text{distinct } \; \text{xs; } \forall \; \text{ys}. \; \text{ys} \in \text{set xs} \implies \text{distinct } \; \text{ys; } \forall \; \text{zs}. \; [\; \text{ys} \in \text{set xs} \; \land \; \text{zs} \in \text{set xs} \; \land \; \text{ys} \neq \; \text{zs} \; ] \implies \text{set } \text{ys} \cap \text{set } \text{zs} = \{\}
\]
\]
by (induct xs) auto

It is best to avoid this indexed version of distinct, but sometimes it is useful.

lemma distinct-conv-nth:
\[
\text{distinct } \; \text{xs} = (\forall \; i < \text{size } \text{xs}. \; \forall \; j < \text{size } \text{xs}. \; i \neq j \implies \text{xs}!i \neq \text{xs}!j)
\]
proof (induct xs)
case (Cons x xs)
show ?case
apply (auto simp add: Cons nth-Cons split: nat.split-asm)
apply (metis Suc-less-eq2 in-set-conv-nth less-not-refl zero-less-Suc)+
done
qed auto

lemma nth-eq-iff-index-eq:
\[
[\; \text{distinct } \; \text{xs}; \; i < \text{length } \text{xs}; \; j < \text{length } \text{xs}; \; i = j \implies \text{xs}!i = \text{xs}!j] = (i = j)
\]
by (auto simp: distinct-conv-nth)

lemma distinct-Ex1:
\[
\text{distinct } \; \text{xs} \implies \exists \; i. \; i < \text{size } \text{xs} \land \text{xs}!i = x
\]
by (auto simp: in-set-conv-nth nth-eq-iff-index-eq)

lemma inj-on-nth:
\[
\exists \; i. \; i < \text{size } \text{xs} \implies \text{inj-on } (\text{nth } \text{xs}) \; \text{I}
\]
by (rule inj-onI) (simp add: nth-eq-iff-index-eq)

lemma bij-betw-nth:
assumes distinct xs A = \{..<\text{length } \text{xs}\} B = \text{set } \text{xs}
shows bij_betw (\text{(!)} \; \text{xs}) \; A \; B
using assms unfolding bij_betw_def
by (auto intro!: inj-on-nth simp: set-conv-nth)

lemma set-update-distinct:
\[
[\; \text{distinct } \; \text{xs}; \; n < \text{length } \text{xs} \; ] \implies \text{set} (\text{xs}[n := x]) = \text{insert} \; x \; (\text{set } \text{xs} - \{x\})
\]
by (auto simp: set-eq-iff in-set-conv-nth nth-list-update nth-eq-iff-index-eq)

lemma distinct-swap:
\[
[\; i < \text{size } \text{xs}; \; j < \text{size } \text{xs} \; ] \implies \text{distinct}(\text{xs}[i := \text{xs}!j, \; j := \text{xs}!i]) = \text{distinct } \; \text{xs}
\]
apply (simp add: distinct-conv-nth nth-list-update)
apply safe
apply metis+
done

lemma set-swap[simp]:
THEORY "List"

[\ i < \text{size} \; xs; \ j < \text{size} \; xs \implies \text{set}(xs[i := xs!j, \ j := xs!i]) = set xs\]
bysimp add: set-conv-nth nth-list-update) metis

lemma distinct-card: distinct \; xs \implies \text{card (set \; xs)} = \text{size} \; xs
by (induct \; xs) auto

lemma card-distinct: \text{card (set \; xs)} = \text{size} \; xs \implies \text{distinct} \; xs
proof (induct \; xs)
case (Cons \; x \; xs)
show \; ?case
proof (cases \; x \in \text{set} \; xs)
case False with Cons show \; ?thesis by simp
next
case True with Cons assms have \text{card (set \; xs)} = Suc (length \; xs)
by (simp add, card-insert-if split: if-split-asm)
moreover have \text{card (set \; xs)} \leq \text{length} \; xs by (rule \; card-length)
ultimately have False by simp
thus \; ?thesis..
qed
qed simp

lemma distinct-length-filter: distinct \; xs \implies \text{length (filter \; P \; xs)} = \text{card} (\{x. \; P \; x\} \text{ Int set} \; xs)
by (induct \; xs) (auto)

lemma not-distinct-decomp: \neg \text{distinct} \; ws \implies \exists \; xs \; ys \; zs \; y. \; ws = xs \# y \# ys \# \; zs
proof (induct \; n \equiv \text{length} \; ws \; \text{arbitrary:ws})
case (Suc \; n \; ws)
then show \; ?case
proof
obtain \; ws \; zs \; y \; ys where decomp: \; as = (xs \# y \# ys) \text{ @ y @ zs}
using not-distinct-decomp[of ws \; y \; y \; zs]
by auto
show \; ?case
proof (cases \; distinct \; (xs \# y \# ys))
case True
with \; decomp have \; dec \; as \; (xs \# y \# ys) \; y \; zs by (simp add: dec-def)

proof
assume \; ?L then show \; ?R
proof (induct \; length \; as \; \text{arbitrary: as \; rule: less-induct})
case less
obtain \; xs \; ys \; zs \; y \; where \; \text{decomp: as = (xs @ y @ y) @ y @ zs}
using not-distinct-decomp[of less \; assms]
by auto
show \; ?case
proof (cases \; distinct \; (xs @ y @ y))
case True
with \; decomp have \; dec \; as \; (xs @ y @ y) \; y \; zs by (simp add: dec-def)
then show ?thesis by blast
next
case False
with less decomp obtain xs' y' ys' where dec (xs @ y # ys) xs' y' ys'
by atomize-elim auto
with decomp have dec as xs' y' (ys' @ y # zs) by (simp add: dec-def)
then show ?thesis by blast
qed
qed
qed (auto simp: dec-def)

lemma distinct-product:
distinct xs ⇒ distinct ys ⇒ distinct (List.product xs ys)
by (induct xs) (auto intro: inj-onI simp add: distinct-map)

lemma distinct-product-lists:
assumes ∀xs ∈ set xss. distinct xs
shows distinct (product-lists xss)
using assms proof (induction xss)
case (Cons xs xss) note * = this
then show ?case
proof (cases product-lists xss)
case Nil then show ?thesis by (induct xs) simp-all
next
case (Cons ps pss) with * show ?thesis
by (auto intro!: inj-onI distinct-concat simp add: distinct-map)
qed
qed simp

lemma length-remdups-concat:
length (remdups (concat xss)) = card (⋃xs∈set xss. set xs)
by (simp add: distinct-card [symmetric])

lemma remdups-append2:
remdups (xs @ remdups ys) = remdups (xs @ ys)
by(induction xs) auto

lemma length-remdups-card-conv: length(remdups xs) = card(set xs)
proof −
  have xs: concat[xs] = xs by simp
  from length-remdups-concat[of [xs]] show ?thesis unfolding xs by simp
qed

lemma remdups-remdups: remdups (remdups xs) = remdups xs
by (induct xs) simp-all

lemma distinct-butlast:
assumes distinct xs
shows distinct (butlast xs)
proof \((\text{cases } \text{xs} = [])\)
\begin{align*}
\text{case } \text{False} & \quad \text{from } \text{xs} \neq []; \text{ obtain } \text{ys} \text{ y where } \text{xs} = \text{ys} @ [y] \text{ by } (\text{cases } \text{xs} \text{ rule: rev-cases}) \\
& \quad \text{auto} \\
& \quad \text{with } (\text{distinct } \text{xs}) \text{ show } \text{?thesis by simp} \tag*{qed (auto)}
\end{align*}

\textbf{lemma} \text{remdups-map-remdups:} \\
\text{remdups } (\text{map } f (\text{remdups } \text{xs})) = \text{remdups } (\text{map } f \text{ xs}) \\
\text{by } (\text{induct } \text{xs}) \text{ simp-all}

\textbf{lemma} \text{distinct-zipI1:} \\
\text{assumes } \text{distinct } \text{xs} \\
\text{shows } \text{distinct } (\text{zip } \text{xs} \text{ ys}) \\
\text{proof } (\text{rule zip-obtain-same-length}) \\
\text{fix } \text{xs}': \text{ 'a list and } \text{ys}' :: \text{ 'b list and } n \\
\text{assume } \text{length } \text{xs}' = \text{length } \text{ys}' \\
\text{assume } \text{xs}' = \text{take } n \text{ xs} \\
\text{with } \text{assms } \text{have } \text{distinct } \text{xs}' \text{ by simp} \\
\text{with } (\text{length } \text{xs}' = \text{length } \text{ys'} \text{) show } \text{distinct } (\text{zip } \text{xs}' \text{ ys'}) \\
\text{by } (\text{induct } \text{xs}' \text{ ys'} \text{ rule: list-induct2}) (\text{auto elim: in-set-zipE}) \tag*{qed}

\textbf{lemma} \text{distinct-zipI2:} \\
\text{assumes } \text{distinct } \text{ys} \\
\text{shows } \text{distinct } (\text{zip } \text{xs} \text{ ys}) \\
\text{proof } (\text{rule zip-obtain-same-length}) \\
\text{fix } \text{xs}' :: \text{ 'b list and } \text{ys}' :: \text{ 'a list and } n \\
\text{assume } \text{length } \text{xs}' = \text{length } \text{ys}' \\
\text{assume } \text{ys}' = \text{take } n \text{ ys} \\
\text{with } \text{assms } \text{have } \text{distinct } \text{ys}' \text{ by simp} \\
\text{with } (\text{length } \text{xs}' = \text{length } \text{ys'}) \text{ show } \text{distinct } (\text{zip } \text{xs}' \text{ ys'}) \\
\text{by } (\text{induct } \text{xs}' \text{ ys'} \text{ rule: list-induct2}) (\text{auto elim: in-set-zipE}) \tag*{qed}

\textbf{lemma} \text{set-take-disj-set-drop-if-distinct:} \\
\text{distinct } \text{vs} \implies i \leq j \implies \text{set } (\text{take } i \text{ vs}) \cap \text{ set } (\text{drop } j \text{ vs}) = \{\} \\
\text{by } (\text{auto simp: in-set-conv-nth distinct-conv-nth})

\textbf{lemma} \text{distinct-singleton: } \text{distinct } [x] \text{ by simp}

\textbf{lemma} \text{distinct-length-2-or-more:} \\
\text{distinct } (a \# b \# \text{xs}) \iff (a \neq b \land \text{distinct } (a \# \text{xs}) \land \text{distinct } (b \# \text{xs})) \\
\text{by force}

\textbf{lemma} \text{remdups-adj-altdef: } (\text{remdups-adj } \text{xs} = \text{ys}) \iff \\
(\exists f :: \text{nat } => \text{nat. mono } f \land f : \{0 ..< \text{size } \text{xs}\} = \{0 ..< \text{size } \text{ys}\}
proof
  assume \(?L\)
  then show \(?\exists f. \ (?p f xs ys)\)
proof (induct xs arbitrary: ys rule: remdups-adj.induct)
  case (1 ys)
  thus \(?\case by (intro exI[of - id])\) (auto simp: mono-def)
next
  case (2 x ys)
  thus \(?\case by (intro exI[of - id])\) (auto simp: mono-def)
next
  case (3 x1 x2 xs ys)
  let \(?xs = x1 \# x2 \# xs\)
  let \(?cond = x1 = x2\)
  define zs where \(zs = \text{reemdups-adj} (x2 \# xs)\)
  from \(3(1-2)[of zs]\)
  obtain \(p\) where \(p f (x2 \# xs)\) unfolding zs-def by (cases \(?cond\)) auto
  then have \(f0: f 0 = 0\)
  by (intro mono-image-least[where \(f=f\)] blast+)
  from \(p\) have mono: mono \(?f\) unfolding mono-def by auto
  show \(?case unfolding ys\)
proof (intro exI[of - \(f\)] conjI allI impI)
  show mono \(?f\) by fact
next
  fix \(i\) assume \(i: i < \text{length} \?xs\)
  with \(p\) show \(?\?xs ! i = \?=x1 zs ! (\?f i)\) using zs0 by auto
next
  fix \(i\) assume \(i: i + 1 < \text{length} \?xs\)
  with \(p\) show \((\?\?xs ! i = \?=x1 zs ! (\?f (i + 1)))\) = \((\?f i = \?=f (i + 1))\)
  by (cases \(i\)) (auto simp: f0)
next
  have \(id: \{0..<\text{length} (\?=x1 zs)\} = \text{insert} \?0 (\?=\text{Suc} \?i \{0..<\text{length} zs\})\)
  unfolding zsne by (cases \(?cond\)) auto
  \{ fix \(i\) assume \(i: i < \text{Suc} (\text{length} zs)\)
  hence \(\text{Suc} \?i \in \{0..<\text{Suc} (\text{Suc} (\text{length} zs))\} \cap \text{Collect} (\langle<\ 0)\) by auto
  from imageI[OF this, of \(\lambdai. \text{Suc} (f (\?i - \text{Suc} 0))\)]
  have \(?\text{Suc} (f \?i) \in (\lambdai. \text{Suc} (f (\?i - \text{Suc} 0))) \cdot (\{0..<\text{Suc} (\text{Suc} (\text{length} zs) \langle<\ 0)\})\) by auto
qed
xs))) ∩ Collect ((<) 0)) by auto

then show ?f ⊢ \{0 ..< length ?xs\} = \{0 ..< length (?x1 zs)\}

unfolding id f-xs-zs[symmetric] by auto

qed

next

assume ∃ f. ?p f xs ys
then show ?L
proof

(induct xs arbitrary; ys rule: remdups-adj.induct)

case 1 then show ?case by auto

next

case (2 x) then obtain f where f-img: f ⊢ \{0 ..< size [x]\} = \{0 ..< size ys\}
and f-nth: ∀ i. i < size [x] \implies [x]!i = ys!f i
by blast

have length ys = card (f ⊢ \{0 ..< size [x]\})
using f-img by auto
then have *: length ys = f by auto
then have f 0 = 0 using f-img by auto
with * show ?case using f-nth by (cases ys) auto

next

case (3 x1 x2 xs)
from 3.prems obtain f where f-mono: mono f
and f-img: f ⊢ \{0 ..< length (x1 # x2 # xs)\} = \{0 ..< length ys\}
and f-nth:

∀ i. i < length (x1 # x2 # xs) \implies (x1 # x2 # xs)!i = ys!f i
∀ i. i + 1 < length (x1 # x2 # xs) \implies
((x1 # x2 # xs)!i = (x1 # x2 # xs)!(i + 1)) = (f i = f (i + 1))
by blast

show ?case
proof cases

assume x1 = x2

let ?f' = f o Suc

have remdups-adj (x1 # xs) = ys
proof (intro 3.hyps exI conjI implI allI)

show mono ?f'
using f-mono by (simp add: mono-iff-le-Suc)

next

have ?f' ⊢ \{0 ..< length (x1 # xs)\} = f ⊢ \{Suc 0 ..< length (x1 # x2 # xs)\}

apply safe
apply fastforce
subgoal for ... x by (cases x) auto
done
also have ... = f ⊢ \{0 ..< length (x1 # x2 # xs)\}
proof
  have f 0 = f (Suc 0) using ⟨x1 = x2⟩ f-nth[of 0] by simp
  then show ?thesis
    apply safe
    apply fastforce
    subgoal for ... x by (cases x) auto
  done
qed
also have ... = {0 ..< length ys} by fact
finally show ?thesis using ⟨x1 = x2⟩ by simp
next
  assume x1 ≠ x2
  have 2 ≤ length ys proof
    have 2 = card {f 0, f 1} using ⟨x1 ≠ x2⟩ f-nth[of 0] by auto
    also have ... ≤ card {f i | 0 ..< length (x1 # x2 # xs)}
      by (rule card_mono) auto
    finally have Suc 0 ≠ f i for i using ⟨x1 ≠ x2⟩ by auto
  then show False using f-img ⟨2 ≤ length ys⟩ by auto
qed

have f 0 = 0 using f-mono f-img by (rule mono-image-least) simp

have f (Suc 0) = Suc 0 proof (rule ccontr)
  assume f (Suc 0) ≠ Suc 0
  then have Suc 0 < f (Suc 0) using ⟨x1 ≠ x2⟩ f-nth[of 0] ⟨0 = 0⟩ by auto
  then have \( \forall i. \ Suc 0 < f (Suc i) \)
    using f-mono
    by (meson Suc-le-mono le0 less_le_trans monoD)
  then have Suc 0 ≠ f i for i using ⟨0 = 0⟩
    by (cases i) fastforce+
  then have Suc 0 ∉ f i | 0 ..< length (x1 # x2 # xs) by auto
  then show False using f-img ⟨2 ≤ length ys⟩ by auto
qed

obtain ys' where ys = x1 # x2 # ys'
  using ⟨2 ≤ length ys; f-nth[of 0]; f-nth[of 1]⟩
  apply (cases ys)
  apply (auto simp: ⟨f 0 = 0⟩; if (Suc 0) = Suc 0;)
  done

define f' where f' x = f (Suc x) - 1 for x

{ fix i
  have Suc 0 ≤ f (Suc 0) using f-nth[of 0] ⟨x1 ≠ x2; f 0 = 0⟩ by auto
also have \( \ldots \leq f (\text{Suc } i) \) using \textit{f-mono} by (rule \textit{monoD}) arith
finally have \( \text{Suc } 0 \leq f (\text{Suc } i) \).
\}
ote{\textit{Suc0-le-f-Suc} = this

\{ fix \ i \ have \ f (\text{Suc } i) = \text{Suc } (f' i)
using \textit{Suc0-le-f-Suc}[\text{of } i] \text{ by (auto simp: } f'\text{-def) }
\}
ote{\textit{f-Suc} = this

have \textit{remdups-adj} \( (x2 \ # \ xs) = (x2 \ # \ ys') \)
proof (intro \textit{3.hyps} \textit{ezI} \textit{conjI} \textit{implI} allI)
show \textit{mono} \( f' \)
using \textit{Suc0-le-f-Suc} \textit{f-mono} \text{ by (auto simp: } f'\text{-def mono-iff-le-Suc le-diff-iff)
next
have \( f' \ : \{ 0\.< \text{ length } (x2 \ # \ xs)\} = (\lambda x. f x - 1) \ : \{ 0\.< \text{ length } (x1 \ # \ x2 \ #\ xs)\} \)
by (auto simp: \( f'\text{-def} \quad \text{if } 0 = 0 : \{ f (\text{Suc } 0) = \text{Suc } 0 \text{ image-def Bex-def less-Suc-eq-0-disj) }
\)
also have \( \ldots = (\lambda x. x - 1) \ : \{ 0\.< \text{ length } (x1 \ # \ x2 \ #\ xs)\} \)
by (auto simp: \textit{image-comp})
also have \( \ldots = (\lambda x. x - 1) \ : \{ 0\.< \text{ length } ys \} \)
by (simp only: \textit{f-mono})
also have \( \ldots = \{ 0\.< \text{ length } (x2 \ # \ ys')\} \)
using \( (\textit{ys} = \_\text{ by (fastforce intro: rev-image-eql) )} \)
finally show \( f' \ : \{ 0\.< \text{ length } (x2 \ # \ xs)\} = \{ 0\.< \text{ length } (x2 \ # \ ys')\} \).
qed (insert \( \textit{f-nth}[\text{of } \text{Suc } i \text{ for } i] \ (x1 \neq x2) \), auto simp \textit{add: f-Suc }\( \textit{ys} = \_\text{) by simp }
qed
qed

\lemma{\textit{hd-remdups-adj}[\textit{simp}]\text{: } \textit{hd} (\textit{remdups-adj} \textit{xs}) = \textit{hd} \textit{xs} \text{ by (induction } \textit{xs} \text{ rule: } \textit{remdups-adj.induct) simp-all}}

\lemma{\textit{remdups-adj-Cons} \text{: } \textit{remdups-adj} \ (x \ # \ xs) = (\text{case } \textit{remdups-adj} \textit{xs} \text{ of } [] \Rightarrow \{ x \} \ | \ y \ # \ xs \Rightarrow if x = y \text{ then } y \ # \ xs \text{ else } x \ # \ y \ # \ xs) \text{ by (induct } \textit{xs} \text{ arbitrary: } \textit{x) (auto split: list.splits) }}

\lemma{\textit{remdups-adj-append-two} \text{: } \textit{remdups-adj} \ (xs \ @ \ [x,y]) = \textit{remdups-adj} \ (xs \ @ \ [x]) \ @ \ (if x = y \text{ then } [] \text{ else } [y]) \text{ by (induct } \textit{xs} \text{ rule: } \textit{remdups-adj.induct, simp-all) }}

\lemma{\textit{remdups-adj-adjacent} \text{: } Suc \ i < \text{ length } (\textit{remdups-adj} \textit{xs}) \Rightarrow \text{remdups-adj} \textit{xs} \ ! \ i \neq \text{remdups-adj} \textit{xs} \ ! \ Suc \ i \text{ proof (induction } \textit{xs} \text{ arbitrary: } i \text{ rule: } \textit{remdups-adj.induct) case } (3 \ x \ y \ i) \text{ thus } \text{?case by (cases } i, \text{ cases } x = y \text{) (simp, auto simp: hd-conv-nth[symmetric]) qed simp-all }}
lemma remdups-adj-rev [simp]: remdups-adj (rev xs) = rev (remdups-adj xs)
by (induct xs rule: remdups-adj.induct, simp-all add: remdups-adj-append-two)

lemma remdups-adj-length [simp]: length (remdups-adj xs) \leq length xs
by (induct xs rule: remdups-adj.induct, auto)

lemma remdups-adj-length-get [simp]: xs \neq [] \implies length (remdups-adj xs) \geq Suc 0
by (induct xs rule: remdups-adj.induct, simp-all)

lemma remdups-adj-Nil-iff [simp]: remdups-adj xs = [] \iff xs = []
by (induct xs rule: remdups-adj.induct, simp-all)

lemma remdups-adj-set [simp]: set (remdups-adj xs) = set xs
by (induct xs rule: remdups-adj.induct, simp-all)

lemma remdups-adj-Cons-alt [simp]: x \# tl (remdups-adj (x \# xs)) = remdups-adj (x \# xs)
by (induct xs rule: remdups-adj.induct, auto)

lemma remdups-adj-distinct: distinct xs \implies remdups-adj xs = xs
by (induct xs rule: remdups-adj.induct, simp-all)

lemma remdups-adj-append:
  remdups-adj (xs_1 \@ x \# xs_2) = remdups-adj (xs_1 \@ [x]) \@ tl (remdups-adj (x \# xs_2))
by (induct xs_1 rule: remdups-adj.induct, simp-all)

lemma remdups-adj-singleton:
  remdups-adj xs = [x] \implies xs = replicate (length xs) x
by (induct xs rule: remdups-adj.induct, auto split: if-split-asm)

lemma remdups-adj-map-injective:
  assumes inj f
shows remdups-adj (map f xs) = map f (remdups-adj xs)
by (induct xs rule: remdups-adj.induct) (auto simp add: injD [OF assms])

lemma remdups-adj-replicate:
  remdups-adj (replicate n x) = (if n = 0 then [] else [x])
by (induction n) (auto simp: remdups-adj-Cons)

lemma remdups-upt [simp]: remdups [m..<n] = [m..<n]
proof (cases m \leq n)
  case False then show ?thesis by simp
next
case True then obtain q where n = m + q
  by (auto simp add: le_iff_add)
moreover have remdups [m..<m + q] = [m..<m + q]
  by (induct q) simp-all

THEORY "List"
ultimately show \textit{thesis} by \texttt{simp}

\textbf{66.1.24} \textit{insert}

\begin{description}
\item[lemma \texttt{in-set-insert} [simp]:] \( x \in \text{set} \; \texttt{xs} \implies \texttt{List.insert} \; x \; \texttt{xs} = \texttt{xs} \) by \texttt{(simp add: List.insert-def)}
\item[lemma \texttt{not-in-set-insert} [simp]:] \( x \notin \text{set} \; \texttt{xs} \implies \texttt{List.insert} \; x \; \texttt{xs} = x \# \texttt{xs} \) by \texttt{(simp add: List.insert-def)}
\item[lemma \texttt{insert-Nil} [simp]:] \( \texttt{List-insert} \; x \; [] = [x] \) by \texttt{simp}
\item[lemma \texttt{set-insert} [simp]:] \( \text{set} (\texttt{List-insert} \; x \; \texttt{xs}) = \texttt{insert} \; x \; (\text{set} \; \texttt{xs}) \) by \texttt{(auto simp add: List.insert-def)}
\item[lemma \texttt{distinct-insert} [simp]:] \( \text{distinct} (\texttt{List-insert} \; x \; \texttt{xs}) = \text{distinct} \; \texttt{xs} \) by \texttt{(simp add: List.insert-def)}
\item[lemma \texttt{insert-remdups}:] \( \texttt{List-insert} \; x \; (\texttt{remdups} \; \texttt{xs}) = \texttt{remdups} \; (\texttt{List-insert} \; x \; \texttt{xs}) \) by \texttt{(simp add: List.insert-def)}
\end{description}

\textbf{66.1.25} \textit{List.union}

This is all one should need to know about union:

\begin{description}
\item[lemma \texttt{set-union} [simp]:] \( \text{set} (\texttt{List.union} \; \texttt{xs} \; \texttt{ys}) = \text{set} \; \texttt{xs} \cup \text{set} \; \texttt{ys} \) by \texttt{(induct} \; \texttt{xs} \; \texttt{arbitrary: \texttt{ys}} \; \texttt{simp-all)}
\item[lemma \texttt{distinct-union} [simp]:] \( \text{distinct}(\texttt{List.union} \; \texttt{xs} \; \texttt{ys}) = \text{distinct} \; \texttt{ys} \) by \texttt{(induct} \; \texttt{xs} \; \texttt{arbitrary: \texttt{ys}} \; \texttt{simp-all)}
\end{description}

\textbf{66.1.26} \textit{find}

\begin{description}
\item[lemma \texttt{find-None-iff}]: \( \texttt{List.find} \; P \; \texttt{xs} = \texttt{None} \iff \neg (\exists \; x. \; x \in \text{set} \; \texttt{xs} \wedge P \; x) \) \texttt{proof} \texttt{(induction} \; \texttt{xs} \texttt{)}
\item[case \texttt{Nil} \texttt{thus} \texttt{?case} \texttt{by} \texttt{simp}]
\item[case \texttt{(Cons} \; x \; \texttt{xs} \texttt{)} \texttt{thus} \texttt{?case} \texttt{by} \texttt{(fastforce} \texttt{split: if-splits)}]
\texttt{qed}
\item[lemma \texttt{find-Some-iff}]: \( \texttt{List.find} \; P \; \texttt{xs} = \texttt{Some} \; x \iff (\exists \; i<\texttt{length} \; \texttt{xs}. \; P \; (\texttt{xs}!i) \wedge x = \texttt{xs}!i \wedge (\forall \; j<i. \; \neg P \; (\texttt{xs}!j))) \)\end{description}
proof (induction xs)
  case Nil thus ?case by simp
next
  case (Cons x xs) thus ?case
    apply (auto simp; nth-Cons' split: if_splits)
    using diff-Suc-1 [unfolded One-nat-def] less-Suc-eq-0-disj by fastforce
qed

lemma find-cong [fundef-cong]:
  assumes xs = ys and \( \forall x. x \in \text{set} \ ys \implies P x = Q x \)
  shows List.find P xs = List.find Q ys
proof (cases List.find P xs)
  case None thus ?thesis by (metis find-None-iff assms)
next
  case (Some x)
  hence List.find Q ys = Some x using assms
    by (auto simp add: find-Some-iff)
  thus ?thesis using Some by auto
qed

lemma find-dropWhile:
  List.find P xs = (case dropWhile (Not o P) xs
                  of [] \Rightarrow None
                    | x # y \Rightarrow Some x)
  by (induct xs) simp-all

66.1.27 count-list

lemma count-notin [simp]: \( x \notin \text{set} \ xs \implies \text{count-list} \ xs \ x = 0 \)
  by (induction xs) auto

lemma count-le-length: \( \text{count-list} \ xs \ x \leq \text{length} \ xs \)
  by (induction xs) auto

lemma sum-count-set:
  set xs \subseteq X \implies \text{finite} \ X \implies \text{sum} (\text{count-list} \ xs) \ X = \text{length} \ xs
proof (induction xs arbitrary: X)
  case (Cons x xs)
  then show ?case
    apply (auto simp: sum.If-cases sum.remove)
    by (metis (no-types) Cons.IH Cons.prems(2) diff-eq sum.remove)
qed simp

66.1.28 List.extract

lemma extract-None iff: List.extract P xs = None \iff (\exists x \in \text{set} \ xs. P x)
by (auto simp: extract-def dropWhile-eq-Cons-conv split: list.split)
  (metis in-set-conv-decomp)

lemma extract-SomeE:
THEORY “List”

\textbf{List}\_extract \( P \) \( xs \) = Some \((ys, y, zs) \) \( \implies \)
\( xs = ys \# ys \# zs \land P y \land \neg (\exists y \in set ys. \ P y) \)
\textit{by} (auto simp: extract-def dropWhile-eq-Cons-conv split: list.splits)

\textbf{lemma} extract-Some-iff:
\begin{align*}
\text{List}\_extract \ P \xs = \text{Some} \ ((\ys, y, \zs)) \iff &\ \xs = \ys \# y \# \zs \land P y \land \neg (\exists y \in set \ys. \ P y) \\
\text{by} &\ (auto \ simp: \ extract-def \ dropWhile-eq-Cons-conv \ split: \ list.splits)
\end{align*}

\textbf{lemma} extract-Nil-code[code]: \( \text{List}\_extract \ P \ []; \) \( = \text{None} \)
\textit{by} (simp add: extract-def)

\textbf{lemma} extract-Cons-code[code]:
\begin{align*}
\text{List}\_extract \ P \ (x \# xs) = &\ \begin{cases} 
\text{if} \ P \ x \ \text{then} \ \text{Some} \ ([], x, xs) \ \text{else} \ \\
\text{case} \ \text{List}\_extract \ P \ xs \ \text{of} \\
\text{None} \Rightarrow &\ \text{None} \ | \\
\text{Some} \ ((\ys, y, \zs)) \Rightarrow \text{Some} \ (x \# \ys, y, \zs) \\
\end{cases} \\
\text{by} (auto simp add: extract-def comp-def split: list.splits) \\
\text{metis} \ \text{dropWhile-eq-Nil-conv} \ \text{list}\_distinct(1)
\end{align*}

\textbf{66.1.29 remove1}

\textbf{lemma} remove1-append:
\begin{align*}
\text{remove1} \ x \ (xs \ @ \ ys) = &\ \begin{cases} 
\text{if} \ x \in set xs \ \text{then} \ \text{remove1} \ x \ xs \ @ \ ys \ \text{else} \ xs \ @ \ \text{remove1} \ x \ ys \\
\end{cases} \\
\textit{by} (induct xs) \ \text{auto}
\end{align*}

\textbf{lemma} remove1-commute: \( \text{remove1} \ x \ (\text{remove1} \ y \ zs) = \text{remove1} \ y \ (\text{remove1} \ x \ zs) \)
\textit{by} (induct zs) \ \text{auto}

\textbf{lemma} in-set-remove1[simp]:
\begin{align*}
a \neq b \implies a \in set(\text{remove1} \ b \ xs) = &\ (a \in set \ xs) \\
\textit{by} (induct \ xs) \ \text{auto}
\end{align*}

\textbf{lemma} set-remove1-subset: \( set(\text{remove1} \ x \ xs) \subseteq set \ xs \)
\textit{by} (induct \ xs) \ \text{auto}

\textbf{lemma} set-remove1-eq[simp]: \( \text{distinct} \ xs \implies set(\text{remove1} \ x \ xs) = set \ xs - \{x\} \)
\textit{by} (induct \ xs) \ \text{auto}

\textbf{lemma} length-remove1:
\begin{align*}
\text{length}(\text{remove1} \ x \ xs) = &\ \begin{cases} 
\text{if} \ x \in set \ xs \ \text{then} \ \text{length} \ xs - 1 \ \text{else} \ \text{length} \ xs \\
\end{cases} \\
\textit{by} (induct \ xs) \ \text{auto dest!:length-pos-if-in-set}
\end{align*}

\textbf{lemma} remove1-filter-not[simp]:
\begin{align*}
\neg P \ x \implies \text{remove1} \ x \ (\text{filter} \ P \ xs) = \text{filter} \ P \ xs \\
\textit{by}(induct \ xs) \ \text{auto}
\end{align*}

\textbf{lemma} filter-remove1:
filter $Q$ (remove1 $x$ $xs$) = remove1 $x$ (filter $Q$ $xs$)
by (induct $xs$) auto

lemma notin-set-remove1[simp]: $x \notin set$ $xs$ $\implies$ $x \notin set$(remove1 $y$ $xs$)
by(insert set-remove1-subset) fast

lemma distinct-remove1[simp]: distinct $xs$ $\implies$ distinct(remove1 $x$ $xs$)
by (induct $xs$) simp-all

lemma remove1-remdups:
  distinct $xs$ $\implies$ remove1 $x$ (remdups $xs$) = remdups (remove1 $x$ $xs$)
by (induct $xs$) simp-all

lemma remove1-idem: $x \notin set$ $xs$ $\implies$ remove1 $x$ $xs$ = $xs$
by (induct $xs$) simp-all

66.1.30  removeAll

lemma removeAll-filter-not-eq:
removeAll $x$ $xs$ = filter ($\lambda y. x \neq y$)
proof
  fix $xs$
  show removeAll $x$ $xs$ = filter ($\lambda y. x \neq y$) $xs$
    by (induct $xs$) auto
qed

lemma removeAll-append[simp]:
removeAll $x$ ($xs$ @ $ys$) = removeAll $x$ $xs$ @ removeAll $x$ $ys$
by (induct $xs$) auto

lemma set-removeAll[simp]: set(removeAll $x$ $xs$) = set $xs$ - {$x$}
by (induct $xs$) auto

lemma removeAll-id[simp]: $x \notin set$ $xs$ $\implies$ removeAll $x$ $xs$ = $xs$
by (induct $xs$) auto

lemma removeAll-filter-not[simp]:
~ $P$ $x$ $\implies$ removeAll $x$ (filter $P$ $xs$) = filter $P$ $xs$
by(induct $xs$) auto

lemma distinct-removeAll:
  distinct $xs$ $\implies$ distinct (removeAll $x$ $xs$)
by (simp add: removeAll-filter-not-eq)

lemma distinct-remove1-removeAll:
  distinct $xs$ $\implies$ remove1 $x$ $xs$ = removeAll $x$ $xs$
by (induct $xs$) simp-all
lemma map-removeAll-inj-on: inj-on f (insert x (set xs)) \implies \map f (removeAll x xs) = removeAll (f x) (map f xs)  
by (induct xs) (simp-all add: inj-on-def)

lemma map-removeAll-inj: inj f \implies 
map f (removeAll x xs) = removeAll (f x) (map f xs)  
by (rule map-removeAll-inj-on, erule subset-inj-on, rule subset-UNIV)

lemma length-removeAll-less-eq [simp]: 
length (removeAll x xs) \leq length xs  
by (simp add: removeAll-filter-not-eq)

lemma length-removeAll-less [termination-simp]:  
x \in set xs \implies length (removeAll x xs) < length xs  
by (auto dest: length-filter-less simp add: removeAll-filter-not-eq)

66.1.31 replicate  

lemma length-replicate [simp]: length (replicate n x) = n  
by (induct n) auto

lemma replicate-eqI: 
assumes length xs = n and \And y \in set xs \implies y = x  
shows xs = replicate n x  
using assms  
proof (induct xs arbitrary: n)  
case Nil then show ?case by simp  
next  
case (Cons x xs) then show ?case by (cases n) simp-all  
qed

lemma Ex-list-of-length: \exists xs. length xs = n  
by (rule exI[of - replicate n undefined]) simp

lemma map-replicate [simp]: map f (replicate n x) = replicate n (f x)  
by (induct n) auto

lemma map-replicate-const:  
map (\lambda x. k) lst = replicate (length lst) k  
by (induct lst) auto

lemma replicate-app-Cons-same:  
(replicate n x) @ (x # xs) = x # replicate n x @ xs  
by (induct n) auto

lemma rev-replicate [simp]: rev (replicate n x) = replicate n x  
by (induct n) (auto simp: replicate-app-Cons-same)
lemma replicate-add: replicate (n + m) x = replicate n x @ replicate m x
by (induct n) auto

Courtesy of Matthias Daum:
lemma append-replicate-commute:
  replicate n x @ replicate k x = replicate k x @ replicate n x
by (metis add commute replicate-add)

Courtesy of Andreas Lochbihler:
lemma filter-replicate:
  filter P (replicate n x) = (if P x then replicate n x else [])
by (induct n) auto

lemma hd-replicate [simp]: n ≠ 0 ==> hd (replicate n x) = x
by (induct n) auto

lemma tl-replicate [simp]: tl (replicate n x) = replicate (n - 1) x
by (induct n) auto

lemma last-replicate [simp]: n ≠ 0 ==> last (replicate n x) = x
by (atomize (full), induct n) auto

lemma nth-replicate [simp]: i < n ==> (replicate n x)!i = x
by (induct n arbitrary: i)(auto simp: nth-Cons split: nat.split)

Courtesy of Matthias Daum (2 lemmas):
lemma take-replicate[simp]: take i (replicate k x) = replicate (min i k) x
proof (cases k ≤ i)
  case True
  then show ?thesis
  by (simp add: min-def)
next
  case False
  then have replicate k x = replicate i x @ replicate (k - i) x
  by (simp add: replicate-add [symmetric])
  then show ?thesis
  by (simp add: min-def)
qed

lemma drop-replicate[simp]: drop i (replicate k x) = replicate (k-i) x
proof (induct k arbitrary: i)
  case (Suc k)
  then show ?case
  by (simp add: drop-Cons')
qed simp

lemma set-replicate-Suc: set (replicate (Suc n) x) = {x}
by (induct n) auto
lemma set-replicate [simp]: \( n \neq 0 \implies \text{set } (\text{replicate } n \ x) = \{x\} \)
by (fast dest!: not0_implies_Suc intro!: set-replicate-Suc)

lemma set-replicate-conv-if: \( \text{set } (\text{replicate } n \ x) = (\text{if } n = 0 \text{ then } \{\} \text{ else } \{x\}) \)
by auto

lemma in-set-replicate [simp]: \( (x \in \text{set } (\text{replicate } n \ y)) = (x = y \land n \neq 0) \)
by (simp add: set-replicate-conv-if)

lemma Ball-set-replicate [simp]:
\( (\forall x \in \text{set } (\text{replicate } n \ a). \ P x) = (P a \lor n=0) \)
by (simp add: set-replicate-conv-if)

lemma Bex-set-replicate [simp]:
\( (\exists x \in \text{set } (\text{replicate } n \ a). \ P x) = (P a \land n \neq 0) \)
by (simp add: set-replicate-conv-if)

lemma replicate-append-same:
\( \text{replicate } i \ x \ @ \ [x] = x \# \text{replicate } i \ x \)
by (induct i) simp-all

lemma map-replicate-trivial:
\( \text{map } (\lambda i. \ x) [0..<i] = \text{replicate } i \ x \)
by (induct i) (simp-all add: replicate-append-same)

lemma concat-replicate-trivial [simp]:
\( \text{concat } (\text{replicate } i \ []) = [] \)
by (induct i) (auto simp add: map-replicate-const)

lemma replicate-empty [simp]: \( (\text{replicate } n \ x = []) \longleftrightarrow n=0 \)
by (induct n) auto

lemma empty-replicate [simp]: \( ([]) = \text{replicate } n \ x \longleftrightarrow n=0 \)
by (induct n) auto

lemma replicate-eq-replicate [simp]:
\( (\text{replicate } m \ x = \text{replicate } n \ y) \longleftrightarrow (m=n \land (m\neq0 \longrightarrow x\equiv y)) \)
proof (induct m arbitrary: n)
case (Suc m n)
then show \(?case\)
by (induct n) auto
qed simp

lemma replicate-length-filter:
\( \text{replicate } (\text{length } (\text{filter } (\lambda y. x = y) \ xs)) \ x = \text{filter } (\lambda y. x = y) \ xs \)
by (induct xs) auto

lemma comm-append-are-replicate:
\([x s \neq []; y s \neq []; x s @ y s = y s @ x s]\)
THEORY "List"

1292

\[ \Rightarrow \exists m n \, zs . \, \text{concat} (\text{replicate} m \, zs) = xs \land \text{concat} (\text{replicate} n \, zs) = ys \]

proof (induction length \((xs @ ys)\) arbitrary; \(xs \, ys\) rule: less-induct)

\begin{enumerate}
  \item case less

  define \(xs' \, ys'\) where \(xs' = (\text{if} (\text{length} \, xs \leq \text{length} \, ys) \text{ then } xs \text{ else } ys)\)

  and \(ys' = (\text{if} (\text{length} \, xs \leq \text{length} \, ys) \text{ then } ys \text{ else } xs)\)

  then have prems': \(\text{length} \, xs' \leq \text{length} \, ys'\)

  and \(xs' @ ys' = ys' @ xs'\)

  and \(\text{len}: \text{length} \,(xs @ ys) = \text{length} \,(xs' @ ys')\)

  using less by (auto intro: less.hyps)

  from prems'

  obtain \(ws\) where \(ys' = xs' @ ws\)

  by (auto simp: append-eq-append-conv2)

  have \(\exists m \, n \, zs . \, \text{concat} (\text{replicate} m \, zs) = xs' \land \text{concat} (\text{replicate} n \, zs) = ys'\)

  proof (cases \(ws = []\))

  \begin{enumerate}
    \item case True

      then have \(\text{concat} (\text{replicate} 1 \, xs') = xs'\)

      and \(\text{concat} (\text{replicate} 1 \, ys') = ys'\)

      using \(ys' = xs' @ ws\) by auto

      then show ?thesis by blast

  next

  \item case False

    from \(ys' = xs' @ ws\) and \(xs' @ ys' = ys' @ xs'\)

    have \(xs' @ ws = ws @ xs'\) by simp

    then have \(\exists m \, n \, zs . \, \text{concat} (\text{replicate} m \, zs) = xs' \land \text{concat} (\text{replicate} n \, zs) = ws\)

    using False and \(xs' \not= []\) and \(ys' = xs' @ ws\) and \(\text{len}\)

    by (intro less.hyps) auto

    then obtain \(m \, n \, zs\) where \(*: \text{concat} (\text{replicate} m \, zs) = xs'\)

    and \(\text{concat} (\text{replicate} n \, zs) = ws\) by blast

    then have \(\text{concat} (\text{replicate} (m + n) \, zs) = ys'\)

    using \(ys' = xs' @ ws\)

    by (simp add: replicate-add)

    with * show ?thesis by blast

  qed

  then show ?case

  using \(xs'\)-def \(ys'\)-def by meson

  qed

lemma comm-append-is-replicate:

  fixes \(xs \, ys\) :: \('a\ list\)

  assumes \(xs \not= []\) \(ys \not= []\)

  assumes \(xs @ ys = ys @ xs\)

  shows \(\exists n \, zs . \, n > 1 \land \text{concat} (\text{replicate} n \, zs) = xs @ ys\)

proof –
obtain \( m \) \( n \) \( zs \) where \( \text{concat} (\text{replicate} \ m \ zs) = xs \) and \( \text{concat} (\text{replicate} \ n \ zs) = ys \) using \( \text{comm-append-are-replicate[of xs ys, OF assms]} \) by \text{blast} 
then have \( m + n > 1 \) and \( \text{concat} (\text{replicate} \ (m+n) \ zs) = xs @ ys \) using \( \text{comm-append-are-replicate[of xs ys, OF assms]} \) by \text{blast} 
then show \( \text{thesis} \) by \text{blast} 
qed

\textbf{lemma Cons-replicate-eq:} 
\( x \# xs = \text{replicate} \ n \ y \longleftrightarrow x = y \land n > 0 \land xs = \text{replicate} \ (n - 1) \ x \) 
by (induct \( n \)) \text{auto}

\textbf{lemma replicate-length-same:} 
\( (\forall y \in \text{set} \ xs. y = x) \implies \text{replicate} \ (\text{length} \ xs) \ x = xs \) 
by (induct \( xs \)) simp-all

\textbf{lemma foldr-replicate \[simp\]:} 
\( \text{foldr} f (\text{replicate} \ n \ x) = f \ x \ ^^ n \) 
by (induct \( n \)) (simp-all)

\textbf{lemma fold-replicate \[simp\]:} 
\( \text{fold} f (\text{replicate} \ n \ x) = f \ x \ ^^ n \) 
by (subst foldr-fold [symmetric]) simp-all

66.1.32 \textbf{enumerate} 

\textbf{lemma enumerate-simps \[simp, code\]:} 
\( \text{enumerate} \ n \ [] = [] \) 
\( \text{enumerate} \ n \ (x \# xs) = (n, x) \# \text{enumerate} \ (\text{Suc} \ n) \ xs \) 
by (simp-all add: enumerate-eq-zip upt-rec)

\textbf{lemma length.enumerate \[simp\]:} 
\( \text{length} (\text{enumerate} \ n \ xs) = \text{length} \ xs \) 
by (simp add: enumerate-eq-zip)

\textbf{lemma map-fst-enumerate \[simp\]:} 
\( \text{map} \ \text{fst} \ (\text{enumerate} \ n \ xs) = [n..<n + \text{length} \ xs] \) 
by (simp add: enumerate-eq-zip)

\textbf{lemma map-snd-enumerate \[simp\]:} 
\( \text{map} \ \text{snd} \ (\text{enumerate} \ n \ xs) = xs \) 
by (simp add: enumerate-eq-zip)

\textbf{lemma in-set enumerated-eq:} 
\( p \in \text{set} \ (\text{enumerate} \ n \ xs) \longleftrightarrow n \leq \text{fst} \ p \land \text{fst} \ p < \text{length} \ xs + n \land \text{nth} \ xs \ (\text{fst} \ p - n) = \text{snd} \ p \) 
\textbf{proof} – 
\{ fix \( m \)
assumption \( n \leq m \)

moreover assume \( m < \text{length } xs + n \)

ultimately have \([n..<n + \text{length } xs]! (m - n) = m \land \)
\( xs ! (m - n) = xs ! (m - n) \land m - n < \text{length } xs \) by auto

then have \( \exists q. [n..<n + \text{length } xs]! q = m \land \)
\( xs ! q = xs ! (m - n) \land q < \text{length } xs .. \)

} then show \( \text{thesis} \) by (cases \( p \)) (auto simp add: enumerate-eq-zip in-set-zip)

qed

lemma nth-enumerate-eq: \( m < \text{length } xs \Rightarrow \text{enumerate } n \; xs \; ! \; m = (n + m, xs \; ! \; m) \)

by (simp add: enumerate-eq-zip)

lemma enumerate-replicate-eq:
\( \text{enumerate } n \; (\text{replicate } m \; a) = \text{map } (\lambda q. \; (q, a)) \; [n..<n + m] \)

by (rule pair-list-eqI)

(simp-all add: enumerate-eq-zip comp-def map-replicate-const)

lemma enumerate-Suc-eq:
\( \text{enumerate } (\text{Suc } n) \; xs = \text{map } (\text{apfst } \text{Suc}) \; (\text{enumerate } n \; xs) \)

by (rule pair-list-eqI)

(simp-all add: not-le, simp del: map-map add map-Suc-upt map-map [symmetric])

lemma distinct-enumerate [simp]:
\( \text{distinct } (\text{enumerate } n \; xs) \)

by (simp add: enumerate-eq-zip distinct-zipI1)

lemma enumerate-append-eq:
\( \text{enumerate } n \; (\; xs \; @ \; ys) = \text{enumerate } n \; xs \; @ \; \text{enumerate } (n + \text{length } xs) \; ys \)

by (simp add: enumerate-eq-zip add assoc zip-append2)

lemma enumerate-map-upt:
\( \text{enumerate } n \; (\text{map } f \; [n..<m]) = \text{map } (\lambda k. \; (k, f \; k)) \; [n..<m] \)

by (cases \( n \leq m \)) (simp-all add: zip-map2 zip-same-conv-map enumerate-eq-zip)

66.1.33 rotate1 and rotate

lemma rotate0[simp]: \( \text{rotate } 0 = \text{id} \)

by (simp add: rotate-def)

lemma rotate-Suc[simp]: \( \text{rotate } (\text{Suc } n) \; xs = \text{rotate1 } (\text{rotate } n \; xs) \)

by (simp add: rotate-def)

lemma rotate-add:
\( \text{rotate } (m+n) = \text{rotate } m \; \circ \; \text{rotate } n \)

by (simp add: rotate-def funpow-add)

lemma rotate-rotate: \( \text{rotate } m \; (\text{rotate } n \; xs) = \text{rotate } (m+n) \; xs \)

by (simp add: rotate-add)
lemma rotate1-map: \( \text{rotate1} (\text{map} \ f \ xs) = \text{map} \ f \ (\text{rotate1} \ xs) \)
by\((\text{cases} \ xs) \ \text{simp-all}\)

lemma rotate1-rotate-swap: \( \text{rotate1} \ (\text{rotate} \ n \ xs) = \text{rotate} \ n \ (\text{rotate1} \ xs) \)
by\((\text{simp add:rotate-def funpow-swap1})\)

lemma rotate1-length01[simp]: \( \text{length} \ xs \leq 1 \implies \text{rotate1} \ xs = xs \)
by\((\text{cases} \ xs) \ \text{simp-all}\)

lemma rotate-length01[simp]: \( \text{length} \ xs \leq 1 \implies \text{rotate} \ n \ xs = xs \)
by\((\text{induct} \ n) \ (\text{simp-all add:rotate-def})\)

lemma rotate1-hd-tl: \( xs \neq [] \implies \text{rotate1} \ xs = \text{tl} \ xs @ [\text{hd} \ xs] \)
by\((\text{cases} \ xs) \ \text{simp-all}\)

lemma rotate-drop-take: 
\( \text{rotate} \ n \ xs = \text{drop} \ (n \mod \text{length} \ xs) \ xs \@ \text{take} \ (n \mod \text{length} \ xs) \ xs \)
proof \((\text{induct} \ n)\)
\(\text{case} \ (\text{Suc} \ n)\)
\(\text{show} \ ?\text{case}\)
proof \((\text{cases} \ xs = [])\)
\(\text{case} \ False\)
\(\text{then show} \ ?\text{thesis}\)
proof \((\text{cases} \ n \mod \text{length} \ xs = 0)\)
\(\text{case} \ True\)
\(\text{then show} \ ?\text{thesis}\)
\(\text{apply} \ (\text{simp add: mod-Suc})\)
\(\text{by} \ (\text{simp add: False Suc.hyps drop-Suc rotate1-hd-tl take-Suc})\)
next
\(\text{case} \ False\)
\(\text{with} \ (xs \neq []): \text{Suc}\)
\(\text{show} \ ?\text{thesis}\)
\(\text{by} \ (\text{simp add: rotate-def mod-Suc rotate1-hd-tl drop-Suc symmetric[drop-tl symmetric] take-hd-drop linorder-not-le})\)
qed
qed simp

lemma rotate-conv-mod: \( \text{rotate} \ n \ xs = \text{rotate} \ (n \mod \text{length} \ xs) \ xs \)
by\((\text{simp add:rotate-drop-take})\)

lemma rotate-id[simp]: \( n \mod \text{length} \ xs = 0 \implies \text{rotate} \ n \ xs = xs \)
by\((\text{simp add:rotate-drop-take})\)

lemma length-rotate1[simp]: \( \text{length}(\text{rotate1} \ xs) = \text{length} \ xs \)
by\((\text{cases} \ xs) \ \text{simp-all}\)

lemma length-rotate[simp]: \( \text{length}(\text{rotate} \ n \ xs) = \text{length} \ xs \)
by (induct n arbitrary: xs) (simp-all add:rotate-def)

lemma distinct1-rotate[simp]: distinct(rotate1 xs) = distinct xs
by (cases xs) auto

lemma distinct-rotate[simp]: distinct(rotate n xs) = distinct xs
by (induct n) (simp-all add:rotate-def)

lemma rotate-map: rotate n (map f xs) = map f (rotate n xs)
by (simp add:rotate-drop-take take-map drop-map)

lemma set-rotate1[simp]: set(rotate1 xs) = set xs
by (cases xs) auto

lemma set-rotate[simp]: set(rotate n xs) = set xs
by (induct n) (simp-all add:rotate-def)

lemma rotate1-is-Nil-conv[simp]: (rotate1 xs = []) = (xs = [])
by (cases xs) auto

lemma rotate-is-Nil-conv[simp]: (rotate n xs = []) = (xs = [])
by (induct n) (simp-all add:rotate-def)

lemma rotate-rev:
  rotate n (rev xs) = rev(rotate (length xs - (n mod length xs)) xs)
proof (cases length xs = 0 ∨ n mod length xs = 0)
  case False
  then show ?thesis
  by (simp add:rotate-drop-take rev-drop rev-take)
qed force

lemma hd-rotate-conv-nth:
  assumes xs ≠ []
  shows hd(rotate n xs) = xs!(n mod length xs)
proof
  have n mod length xs < length xs
    using assms by simp
  then show ?thesis
  by (metis drop-eq-Nil hd-append2 hd-drop-conv-nth leD rotate-drop-take)
qed

lemma rotate-append: rotate (length l) (l @ q) = q @ l
by (induct l arbitrary: q) (auto simp add:rotate1-rotate-swap)

66.1.34 nths — a generalization of (!) to sets

lemma nths-empty [simp]: nths xs {} = []
by (auto simp add: nths-def)

lemma nths-nil [simp]: nths [] A = []
lemma nths-all: \( \forall i < \text{length } xs. \ i \in I \implies \text{nths } xs \ I = xs \)
apply (simp add: nths-def)
apply (subst filter-True)
apply (clarsimp simp: in-set-zip subset_iff)
apply simp
done

lemma length-nths:
length (nths xs I) = \{ i. i < \text{length } xs \land i \in I \}
by (simp add: nths-def length-filter-conv-card cong: conj-cong)

lemma nths-shift-lemma-Suc:
map fst (filter (\(\lambda p. P(\text{Suc}(\text{snd } p)))\) (zip xs is)) =
map fst (filter (\(\lambda p. P(\text{snd } p))\) (zip xs (map Suc is)))
proof (induct xs arbitrary: is)
case (Cons x xs is)
show ?case
by (cases is) (auto simp add: Cons
hyps)
qed simp

lemma nths-shift-lemma:
map fst (filter (\(\lambda p. \text{snd } p + i \in A\)) (zip xs [0..<\text{length } xs])) =
map fst (filter (\(\lambda p. \text{snd } p + i \in A\)) (zip xs [0..<\text{length } xs]))
by (induct xs rule: rev-induct) (simp-all add: commute)

lemma nths-append:
nths (l @ l') A = (if 0 \in A then [x] else []) @ nths l \{ j. \ j + \text{length } l \in A \}
unfolding nths-def
proof (induct l' rule: rev-induct)
case (snoc x xs)
then show ?case
by (simp add: upt-add-eq-append[of 0] nths-shift-lemma add.commute)
qed auto

lemma nths-Cons:
nths (x # l) A = (if 0 \in A then [x] else []) @ nths l \{ j. \ Suc j \in A \}
proof (induct l rule: rev-induct)
case (snoc x xs)
then show ?case
by (simp flip: append-Cons add: nths-append)
qed (auto simp: nths-def)

lemma nths-map: nths (map f xs) I = map f (nths xs I)
by (induction xs arbitrary: I) (simp-all add: nths-Cons)

lemma set-nths: \{ \text{set} (nths xs I) = \{ xs!i| i < \text{size } xs \land i \in I \}
by (induct xs arbitrary: I) (auto simp: nths-Cons nth-Cons split:nat.split dest!:
lemma set-nths-subset: set(nths xs I) ⊆ set xs
by (auto simp add: set-nths)

lemma notin-set-nthsI [simp]: x ∉ set xs ⟹ x ∉ set(nths xs I)
by (auto simp add: set-nths)

lemma in-set-nthsD: x ∈ set(nths xs I) ⟹ x ∈ set xs
by (auto simp add: set-nths)

lemma nths-singleton [simp]: nths [x] A = (if 0 ∈ A then [x] else [])
by (simp add: nths-Cons)

lemma distinct-nthsI [simp]: distinct xs ⟹ distinct (nths xs I)
by (induct xs arbitrary: I) (auto simp: nths-Cons)

lemma nths-upt-eq-take [simp]: nths l {..<n} = take n l
by (induct l rule: rev-induct) (simp-all split: nat-diff-split add: nths-append)

lemma nths-nths: nths (nths xs A) B = nths xs {i ∈ A. ∃j ∈ B. card {i' ∈ A. i' < i} = j}
apply (induction xs arbitrary: A B)
apply (auto simp add: nths-Cons card-less-Suc card-less-Suc2)
done

lemma drop-eq-nths: drop n xs = nths xs {i. i ≥ n}
apply (induction xs arbitrary: n)
apply (auto simp add: nths-Cons nths-all-drop-Cons' intro: arg-cong2 [where f=nths, OF refl])
done

lemma nths-drop: nths (drop n xs) I = nths xs ((+) n · I)
by (force simp: drop-eq-nths nths-nths simp flip: atLeastLessThan-iff
intro: arg-cong2 [where f=nths, OF refl])

lemma filter-eq-nths: filter P xs = nths xs {i. i < length xs ∧ P(xs!i)}
by (induction xs) (auto simp: nths-Cons)

lemma filter-in-nths:
distinct xs ⟹ filter (∀x. x ∈ set (nths xs s)) xs = nths xs s
proof (induct xs arbitrary: s)
case Nil thus ?case by simp
next
case (Cons a xs)
then have ∀x. x ∈ set xs ⟹ x ≠ a by auto
with Cons show ?case by (simp add: nths-Cons cong: filter-cong)
qed
subseqs and List.n-lists

lemma length-subseqs: length (subseqs xs) = 2 ^ length xs
  by (induct xs) (simp-all add: Let-def)

lemma subseqs-powset: set ' set (subseqs xs) = Pow (set xs)
  proof
    have aux: \x A. set ' Cons x ' A = insert x ' set ' A
      by (auto simp add: image-def)
    have set (map set (subseqs xs)) = Pow (set xs)
      by (induct xs) (simp-all add: aux Let-def Pow-insert Un-commute comp-def del: map-map)
    then show ?thesis by simp
  qed

lemma distinct-set-subseqs:
  assumes distinct xs
  shows distinct (map set (subseqs xs))
  proof (rule card-distinct)
    have finite (set xs) ..
    then have card (Pow (set xs)) = 2 ^ card (set xs)
      by (rule card-Pow)
    with assms distinct-card [of xs] have card (Pow (set xs)) = 2 ^ length xs
      by simp
    then show card (set (map set (subseqs xs))) = length (map set (subseqs xs))
      by (simp add: subseqs-powset length-subseqs)
  qed

lemma n-lists-Nil [simp]: List.n-lists n [] = (if n = 0 then [[]] else [])
  by (induct n) simp-all

lemma length-n-lists-elem: ys \in set (List.n-lists n xs) \implies length ys = n
  by (induct n arbitrary: ys) auto

lemma set-n-lists: set (List.n-lists n xs) = \{ys. length ys = n \land set ys \subseteq set xs\}
  proof (rule set-eqI)
    fix ys :: 'a list
    show ys \in set (List.n-lists n xs) \iff ys \in \{ys. length ys = n \land set ys \subseteq set xs\}
    proof
      have ys \in set (List.n-lists n xs) \implies length ys = n
        by (induct n arbitrary: ys) auto
      moreover have \forall x. ys \in set (List.n-lists n xs) \implies x \in set ys \implies x \in set xs
        by (induct n arbitrary: ys) auto
      moreover have set ys \subseteq set xs \implies ys \in set (List.n-lists (length ys) xs)
        by (induct ys) auto
      ultimately show ?thesis by auto
    qed
  qed

lemma subseqs-refl: xs \in set (subseqs xs)
by (induct xs) (simp-all add: Let-def)

lemma subset-subseqs: \( X \subseteq \text{set} \, \text{xs} \implies X \in \text{set} \, \text{subseqs} \, \text{xs} \)
unfolding subseqs-powset by simp

lemma Cons-in-subseqsD: \( y \# \, \text{ys} \in \text{set} \, \text{subseqs} \, \text{xs} \implies \text{ys} \in \text{set} \, \text{subseqs} \, \text{xs} \)
by (induct xs) (auto simp: Let-def)

lemma subseqs-distinctD: \( \text{ys} \in \text{set} \, \text{subseqs} \, \text{xs}; \, \text{distinct} \, \text{xs} \) = \( \Rightarrow \) \( \text{distinct} \, \text{ys} \)
proof (induct xs arbitrary: \( \text{ys} \))
  then show \( ?\text{case} \)
  by (auto simp: Let-def)
qed simp

66.1.36 splice

lemma splice-Nil2 simp: \( \text{splice} \, \text{xs} \, [] = \text{xs} \)
by (cases \( \text{xs} \)) simp-all

lemma length-splice simp: \( \text{length} \, \text{splice} \, \text{xs} \, \text{ys} = \text{length} \, \text{xs} + \text{length} \, \text{ys} \)
by (induct \( \text{xs} \, \text{ys} \) rule: splice.induct) auto

lemma split-Nil-iff simp: \( \text{splice} \, \text{xs} \, \text{ys} = [] \Leftrightarrow \text{xs} = [] \wedge \text{ys} = [] \)
by (induct \( \text{xs} \, \text{ys} \) rule: splice.induct) auto

lemma splice-replicate simp: \( \text{splice} \, \text{replicate} \, \text{m} \, \text{x} \, \text{replicate} \, \text{n} \, \text{x} = \text{replicate} \, \text{(m+n)} \, \text{x} \)
apply (induction \( \text{replicate} \, \text{m} \, \text{x} \, \text{replicate} \, \text{n} \, \text{x} \) arbitrary: \( \text{m} \, \text{n} \) rule: splice.induct)
apply (auto simp add: Cons-replicate-eq dest: gr0-implies-Suc)
done

66.1.37 shuffles

lemma shuffles-commutes: \( \text{shuffles} \, \text{xs} \, \text{ys} = \text{shuffles} \, \text{ys} \, \text{xs} \)
by (induction \( \text{xs} \, \text{ys} \) rule: shuffles.induct) (simp-all add: Un-commute)

lemma Nil-in-shuffles simp: \( [] \in \text{shuffles} \, \text{xs} \, \text{ys} \leftrightarrow \text{xs} = [] \wedge \text{ys} = [] \)
by (induct \( \text{xs} \, \text{ys} \) rule: shuffles.induct) auto

lemma shufflesE:
\( \text{zs} \in \text{shuffles} \, \text{xs} \, \text{ys} \implies \)
\( (\text{zs} = \text{xs} \implies \text{ys} = [] \implies \text{P}) \implies \)
\( (\text{zs} = \text{ys} \implies \text{xs} = [] \implies \text{P}) \implies \)
\( (\forall x \, x \# x' \, \text{zs} = x = x' \implies \text{zs} = z \# z' \implies x = z \implies z' \in \text{shuffles} \, x' \text{ys} \implies \text{P}) \implies \)
\( (\forall y \, y \# y' \, \text{zs} = y = y' \implies \text{zs} = z \# z' \implies y = z \implies z' \in \text{shuffles} \, x \text{ys}' \implies \text{P}) \implies \text{P} \)
by (induct \( \text{xs} \, \text{ys} \) rule: shuffles.induct) auto
lemma \textit{Cons-in-shuffles-iff}:
\[
\begin{aligned}
z \neq [] & \land \text{hd} \, xs = z \land zs \in \text{shuffles} \, (\text{tl} \, xs) \, ys \\
yz \neq [] & \land \text{hd} \, ys = z \land zs \in \text{shuffles} \, xs \, (\text{tl} \, ys)
\end{aligned}
\]
by (induct \, xs \, ys \, rule: \text{shuffles.induct}) auto

lemma \textit{splice-in-shuffles} [simp, intro]: splice \, xs \, ys \in \text{shuffles} \, xs \, ys
by (induction \, xs \, ys \, rule: \text{splice.induct}) (simp-all add: \text{Cons-in-shuffles-iff} \, \text{shuffles-commutes})

lemma \textit{Nil-in-shufflesI}: xs = [] \implies ys = [] \implies [] \in \text{shuffles} \, xs \, ys
by simp

lemma \textit{Cons-in-shuffles-leftI}: zs \in \text{shuffles} \, xs \, ys \implies z \# zs \in \text{shuffles} \, (z \# xs) \, ys
by (cases ys) auto

lemma \textit{Cons-in-shuffles-rightI}: zs \in \text{shuffles} \, xs \, ys \implies z \# zs \in \text{shuffles} \, xs \, (z \# ys)
by (cases xs) auto

lemma \textit{finite-shuffles} [simp, intro]: finite (\text{shuffles} \, xs \, ys)
by (induction \, xs \, ys \, rule: \text{shuffles.induct}) simp-all

lemma \textit{length-shuffles}: zs \in \text{shuffles} \, xs \, ys \implies \text{length} \, zs = \text{length} \, xs + \text{length} \, ys
by (induction \, xs \, ys \, arbitrary: zs \, rule: \text{shuffles.induct}) auto

lemma \textit{set-shuffles}: zs \in \text{shuffles} \, xs \, ys \implies \text{set} \, zs = \text{set} \, xs \cup \text{set} \, ys
by (induction \, xs \, ys \, arbitrary: zs \, rule: \text{shuffles.induct}) auto

lemma \textit{distinct-disjoint-shuffles}:
assumes distinct zs distinct ys set xs \cap set ys = \{\} zs \in \text{shuffles} \, xs \, ys
shows distinct zs
using assms
proof (induction \, xs \, ys \, arbitrary: zs \, rule: \text{shuffles.induct})
case (3 \, x \, xs \, y \, ys)
show ?case
proof (cases zs)
case (\text{Cons} \, z \, zs')
with 3.prems and 3.IH[of zs'] show thesis by (force dest: \text{set-shuffles})
qed simp-all
qed simp-all

lemma \textit{Cons-shuffles-subset1} (\#) \, x \, \text{\shortmid} \, \text{shuffles} \, xs \, ys \subseteq \text{shuffles} \, (x \, \# \, xs) \, ys
by (cases ys) auto

lemma \textit{Cons-shuffles-subset2} (\#) \, y \, \text{\shortmid} \, \text{shuffles} \, xs \, ys \subseteq \text{shuffles} \, xs \, (y \, \# \, ys)
by (cases zs) auto

lemma \textit{filter-shuffles}:
filter \( P \cdot \) shuffles \( xs \) \( ys \) = shuffles \((\text{filter } P \cdot xs)\) \((\text{filter } P \cdot ys)\)

proof

have \(*\): \( z = \text{filter } P \cdot A \) = \((\text{if } P \cdot x \text{ then } x \cdot \text{filter } P \cdot A \text{ else } \text{filter } P \cdot A)\)

proof

by \((\text{auto simp: image-image})\)

show \(?thesis\)

by \((\text{induction } xs \ ys \text{ rule: shuffles.induct})\)

\((\text{simp-all split: if-splits add: image-Un \{*\} Un-absorb1 Un-absorb2 \}}\)

\Cons-shuffles-subset1 Cons-shuffles-subset2\)

qed

lemma \text{filter-shuffles-disjoint1}:

assumes \( \{s \cap y \} \in \text{shuffles } xs \ ys \)

shows \( \text{filter } (\lambda x. \ x \in s) \) \( z = \text{xs} \) \( (\text{if } P \cdot x \text{ then } x \cdot \text{filter } P \cdot A \text{ else } \text{filter } P \cdot A)\)

using \(\text{assms}\)

proof

from \(\text{assms}\) have \(\text{filter } P \cdot z \in \text{filter } P \cdot \text{shuffles } xs \ ys\) by \(\text{blast}\)

also have \(\text{filter } P \cdot \text{shuffles } xs \ ys = \text{shuffles } (\text{filter } P \cdot \text{xs})\) \(\text{filter } P \cdot \text{ys})\)

by \((\text{rule filter-shuffles})\)

also have \(\text{filter } P \cdot \text{xs} = \text{xs}\) by \((\text{rule filter-True})\) \(\text{simp-all}\)

also have \(\text{filter } P \cdot \text{ys} = \text{[]}\) by \((\text{rule filter-False})\) \(\text{insert \text{assms}(1)}\), \(\text{auto}\)

also have \(\text{shuffles } xs \text{ } \text{[]} = \{xs\}\) by \(\text{simp}\)

finally show \(\text{filter } P \cdot z = \text{xs}\) by \(\text{simp}\)

next

from \(\text{assms}\) have \(\text{filter } P \cdot z \in \text{filter } P \cdot \text{shuffles } xs \ ys\) by \(\text{blast}\)

also have \(\text{filter } P \cdot \text{shuffles } xs \ ys = \text{shuffles } (\text{filter } P \cdot \text{xs})\) \(\text{filter } P \cdot \text{ys})\)

by \((\text{rule filter-shuffles})\)

also have \(\text{filter } P \cdot \text{ys} = \text{ys}\) by \((\text{rule filter-True})\) \(\text{insert assms}(1), \text{auto}\)

also have \(\text{filter } P \cdot \text{xs} = \text{[]}\) by \((\text{rule filter-False})\) \(\text{insert assms}(1), \text{auto}\)

also have \(\text{shuffles } \text{[]} \text{ } \text{ys} = \{\text{ys}\}\) by \(\text{simp}\)

finally show \(\text{filter } P \cdot z = \text{ys}\) by \(\text{simp}\)

qed

lemma \text{filter-shuffles-disjoint2}:

assumes \( \{s \cap y \} \in \text{shuffles } xs \ ys\)

shows \( \text{filter } (\lambda x. \ x \in s) \) \( z = \text{xs} \) \( (\text{if } P \cdot x \text{ then } x \cdot \text{filter } P \cdot A \text{ else } \text{filter } P \cdot A)\)

using \(\text{filter-shuffles-disjoint1}[\text{of } ys \ xs \ zs\text{ assms}\]

\(\text{simp-all add: shuffles-commutes Int-commute}\)

lemma \text{partition-in-shuffles}:

\(xs \in \text{shuffles } (\text{filter } P \cdot \text{xs})\) \(\text{(filter } (\lambda x. \neg P \cdot x) \text{ } \text{xs})\)

proof

\((\text{induction } xs)\)

\((\text{case } (\text{Cons } x \ xs)\)

\((\text{show } \text{?case})\)

\((\text{proof } (\text{cases } P \ x)\)

\((\text{case } \text{True})\)

\((\text{hence } x \# xs \in (\#) x \cdot \text{shuffles } (\text{filter } P \cdot \text{xs}) \text{ (filter } (\lambda x. \neg P \cdot x) \text{ } \text{xs})\)

\((\text{by } (\text{intro image1 Cons.IH}))\)

\)}
also have ... ⊆ shuffles (filter P (x # xs)) (filter (λx. ¬P x) (x # xs))
  by (simp add: True Cons-shuffles-subset1)
finally show thesis .
next
  case False
  hence x # xs ∈ (#) x ' shuffles (filter P xs) (filter (λx. ¬P x) xs)
  by (intro imageI Cons.IH)
also have ... ⊆ shuffles (filter P (x # xs)) (filter (λx. ¬P x) (x # xs))
  by (simp add: False Cons-shuffles-subset2)
finally show thesis .
qed

lemma inv-image-partition:
  assumes "∀ x. x ∈ set xs ⇒ P x ∧ y ∈ set ys ⇒ ¬P y"
  shows "partition P − {((xs, ys))} = shuffles xs ys"
proof (intro equalityI subsetI)
  fix zs assume zs: zs ∈ shuffles xs ys
  hence [simp]: set zs = set xs ∪ set ys by (rule set-shuffles)
  from assms have filter P zs = filter (λx. x ∈ set xs) zs
  by (intro filter-cong refl; force)+
  moreover from assms have set xs ∩ set ys = {} by auto
  ultimately show zs ∈ partition P − {((xs, ys))} using zs
  by (simp add: o-def filter-shuffles-disjoint1 filter-shuffles-disjoint2)
next
  fix zs assume zs ∈ partition P − {((xs, ys))}
  thus zs ∈ shuffles xs ys using partition-in-shuffles[of zs] by (auto simp: o-def)
qed

66.1.38 Transpose

function transpose where
  transpose [] = [] |
  transpose ([] # xss) = transpose xss |
  transpose ((x # xs) # xss) =
  (x # [h. (h#t) ← xss]) # transpose (xs # [t. (h#t) ← xss])
by pat-completeness auto

lemma transpose-aux-filter-head:
  concat (map (case-list []) (λh t. [h])) xss) =
  map (λxs. hd xs) (filter (λys. ys ≠ []) xss)
  by (induct xss) (auto split: list.split)

lemma transpose-aux-filter-tail:
  concat (map (case-list []) (λh t. [t])) xss) =
  map (λxs. tl xs) (filter (λys. ys ≠ []) xss)
  by (induct xss) (auto split: list.split)
**lemma** transpose-aux-max:

\[
\text{max} (\text{Suc} (\text{length} \, \text{xs})) (\text{foldr} (\lambda \text{xs}. \text{max} (\text{length} \, \text{xs})) \, \text{xs} \, 0) = \\
\text{Suc} (\text{max} (\text{length} \, \text{xs})) (\text{foldr} (\lambda \text{xs}. \text{max} (\text{length} \, \text{xs} - \text{Suc} \, 0)) (\text{filter} (\lambda \text{ys}. \text{ys} \neq [])) \, \text{xss}) \, 0)
\]

(is max - ?foldB = Suc (max - ?foldA))

**proof** (cases (filter (\lambda \text{ys}. \text{ys} \neq []) \, \text{xss} = []))

**case** True

hence foldr (\lambda \text{xs}. \text{max} (\text{length} \, \text{xs})) \, \text{xss} \, 0 = 0

**proof** (induct xss)

**case** (Cons x xs)

then have x = [] by (cases x) auto

with Cons show ?case by auto

**qed** simp

thus ?thesis using True by simp

next

**case** False

have foldA: \text{foldA} = \text{foldr} (\lambda \text{x}. \text{max} (\text{length} \, \text{x})) (\text{filter} (\lambda \text{ys}. \text{ys} \neq [])) \, \text{xss} \, 0 - 1

by (induct xss) auto

have foldB: \text{foldB} = \text{foldr} (\lambda \text{x}. \text{max} (\text{length} \, \text{x})) (\text{filter} (\lambda \text{ys}. \text{ys} \neq [])) \, \text{xss} \, 0

by (induct xss) auto

have 0 < ?foldB

**proof**

from False

obtain z zs where zs: (\text{filter} (\lambda \text{ys}. \text{ys} \neq []) \, \text{xss}) = z\#zs by (auto simp: neq-Nil-conv)

hence z \in \text{set} (\text{filter} (\lambda \text{ys}. \text{ys} \neq []) \, \text{xss}) by auto

hence z \neq [] by auto

thus ?thesis unfolding foldB zs

by (auto simp: max-def intro: less-le-trans)

**qed**

thus ?thesis unfolding foldA foldB max-Suc-Suc symmetric by simp

**qed**

termination transpose

by (relation measure (\lambda \text{xs}. \text{foldr} (\lambda \text{xs}. \text{max} (\text{length} \, \text{xs})) \, \text{xs} \, 0 + \text{length} \, \text{xs}))

(auto simp: transpose-aux-filter-tail foldr-map comp-def transpose-aux-max less-Suc-eq-le)

**lemma** transpose-empty: (transpose \text{xs} = []) \leftrightarrow (\forall x \in \text{set} \, \text{xs}. \, x = [])

by (induct rule: transpose.induct) simp-all

**lemma** length-transpose:

fixes \text{xs} :: 'a list list

shows length (transpose \text{xs}) = foldr (\lambda \text{xs}. \text{max} (\text{length} \, \text{xs})) \, \text{xs} \, 0
lemma nth-transpose:
  fixes xs :: 'a list list
  assumes i < length (transpose xs)
  shows transpose xs ! i = map (λxs. xs ! i) (filter (λys. i < length ys) xs)
  using assms proof (induct arbitrary: i rule: transpose.induct)
    case (3 x xs xss)
    define XS where XS = (x # xs) # xss
    hence simp: XS ≠ [] by auto
    thus ?case proof (cases i)
      case 0 thus ?thesis by (simp add: transpose-aux-filter-head hd-conv-nth)
      next case (Suc j)
      have ∗: ∀xs. xs ≠ # map tl xss = map tl ((x#xs)#xss) by simp
      have ∗∗: ∀xs. (x#xs) ≠ filter (λys. ys ≠ []) xss = filter (λys. ys ≠ [])
        ((x#xs)#xss) by simp
      { fix x have Suc j < length x ⟷ x ≠ [] ∧ j < length x = Suc 0
        by (cases x) simp-all
      } note ∗∗∗ = this
      have j-less: j < length (transpose (xs ≠ concat (map (case-list [] (λh t. [t])) xss)))
        using 3.prems by (simp add: transpose-aux-filter-tail length-transpose Suc)
      show ?thesis unfolding transpose.simps i = Suc j; nth-Cons-Suc 3.hyps[OF j-less]
        apply (auto simp: transpose-aux-filter-tail filter-map comp-def length-transpose
          ∗ ∗ ∗ ∗ ∗ XS-def[symmetric])
        by (simp add: nth-tl)
      qed
    qed simp-all

lemma transpose-map-map:
  transpose (map (map f) xs) = map (map f) (transpose xs)
  proof (rule nth-equalityI)
    have simp: length (transpose (map (map f) xs)) = length (transpose xs)
      by (simp add: length-transpose foldr-map comp-def)
    show length (transpose (map (map f) xs)) = length (map (map f) (transpose xs)) by simp
      fix i assume i < length (transpose (map (map f) xs))
      thus transpose (map (map f) xs) ! i = map (map f) (transpose xs) ! i
        by (simp add: nth-transpose filter-map comp-def)
66.1.39  min and arg-min

lemma min-list-Min: \( \text{xs} \neq [] \implies \text{min-list} \trm{xs} = \trm{Min} (\text{set} \trm{xs}) \)

by (induction \trm{xs} rule: \text{induct012})(auto)

lemma f-arg-min-list-f: \( \text{xs} \neq [] \implies f(\text{arg-min-list} f \trm{xs}) = \trm{Min} (f' (\text{set} \trm{xs})) \)

by (induction \trm{f} \trm{xs} rule: \text{arg-min-list.induct}) (auto simp: min-def intro!: antisym)

lemma arg-min-list-in: \( \text{xs} \neq [] \implies \text{arg-min-list} f \trm{xs} \in \text{set} \trm{xs} \)

by (induction \trm{xs} rule: \text{induct-list012}) (auto simp: Let-def)

66.1.40  (In)finiteness

lemma finite-maxlen:
  \( \text{finite} (M::\trm{a list set}) \implies \exists n. \forall s \in M. \text{size} s < n \)

proof (induct rule: finite.induct)
  case emptyI show ?case by simp
next
  case (insertI M \text{xs}) then obtain \( n \) where \( \forall s \in M. \text{length} s < n \) by blast
  hence \( \forall s \in \text{insert} \trm{xs} M. \text{size} s < \text{max} n (\text{size} \trm{xs}) + 1 \) by auto
  thus ?case ..
qed

lemma lists-length-Suc-eq:
  \( \{ \text{xs}. \text{set} \trm{xs} \subseteq A \land \text{length} \trm{xs} = \text{Suc} n \} = \)
  \( (\lambda(x, n). n\#\trm{xs}) \cdot (\{ \text{xs}. \text{set} \trm{xs} \subseteq A \land \text{length} \trm{xs} = n \} \times A) \)

by (auto simp: length-Suc-conv)

lemma finite-lists-length-eq:
  \( \text{finite} \{ \text{xs}. \text{set} \trm{xs} \subseteq A \land \text{length} \trm{xs} = n \} \)

and card-lists-length-eq: card \( \{ \text{xs}. \text{set} \trm{xs} \subseteq A \land \text{length} \trm{xs} = n \} = (\text{card} A) ^ n \)

using \( \text{finite} A \)

by (induct n)
  (auto simp: card-image inj-split-Cons lists-length-Suc-ev cong: conj-cong)

lemma finite-lists-length-le:
  \( \text{assumes finite} A \text{ shows finite} \{ \text{xs}. \text{set} \trm{xs} \subseteq A \land \text{length} \trm{xs} \leq n \} \)

(is finite \( \trm{S} \))

proof
  have \( \trm{S} = (\bigcup n \in \{0..n\}. \{ \text{xs}. \text{set} \trm{xs} \subseteq A \land \text{length} \trm{xs} = n \}) \) by auto
  thus ?thesis by (auto intro!: finite-lists-length-ev[OF \( \text{finite} A \)] simp only:)
qed

lemma card-lists-length-le:
  \( \text{assumes finite} A \text{ shows card} \{ \text{xs}. \text{set} \trm{xs} \subseteq A \land \text{length} \trm{xs} \leq n \} = (\sum i \leq n. \text{card} A \cdot i) \)

proof
  have \( (\sum i \leq n. \text{card} A \cdot i) = \text{card} (\bigcup i \leq n. \{ \text{xs}. \text{set} \trm{xs} \subseteq A \land \text{length} \trm{xs} = i \}) \)
using (finite A)
by (subst card-UN-disjoint)
(auto simp add: card-lists-length-eq finite-lists-length-eq)
also have \((\bigcup i \leq n. \{\text{xs. set xs} \subseteq A \land \text{length xs} = i\}) = \{\text{xs. set xs} \subseteq A \land \text{length xs} \leq n\}\)
  by auto
finally show \(?\text{thesis}\) by simp
qed

lemma finite-lists-distinct-length-eq [intro]:
assumes finite A
shows finite \(\{\text{xs. length xs} = n \land \text{distinct xs} \land \text{set xs} \subseteq A\}\) (is finite \(?S\))
proof 
  have finite \(\{\text{xs. set xs} \subseteq A \land \text{length xs} = n\}\)
    using (finite A) by (rule finite-lists-length-eq)
  moreover have \(?S\) \(\subseteq\) \(\{\text{xs. set xs} \subseteq A \land \text{length xs} = n\}\) by auto
  ultimately show \(?\text{thesis}\) using finite-subset by auto
qed

lemma card-lists-distinct-length-eq:
assumes finite A k \(\leq\) card A
shows card \(\{\text{xs. length xs} = k \land \text{distinct xs} \land \text{set xs} \subseteq A\}\) = \(\prod\{\text{card A} - k + 1 \ldots \text{card A}\}\)
using assms
proof (induct k)
  case 0
  then have \(\{\text{xs. length xs} = 0 \land \text{distinct xs} \land \text{set xs} \subseteq A\}\) = \(\{\}\) by auto
  then show \(?\text{case}\) by simp
next
  case (Suc k)
  let \(?k\text{-list} = \lambda k \text{ xs}. \text{length xs} = k \land \text{distinct xs} \land \text{set xs} \subseteq A\)
  have inj-Cons: \(\forall A. \text{inj-on} (\lambda (\text{xs, n}). n \neq \text{xs}) A\) by (rule inj-onI) auto
  from Suc have \(k \leq \text{card A}\) by simp
  moreover note (finite A)
  moreover have \(\{\text{xs. \?k-list k xs}\}\)
    by (rule finite-subset) (use finite-lists-length-eq[OF \(\text{finite A}; \text{of k}\) in auto]
  moreover have \(\forall i. j. i \neq j \rightarrow \{i\} \times (A - set i) \cap \{j\} \times (A - set j) = \{\}\)
    by auto
  moreover have \(\forall i. i \in \{\text{xs. \?k-list k xs}\} \implies \text{card} (A - set i) = \text{card} A - k\)
    by (simp add: card-Diff-subset distinct-card)
  moreover have \(\{\text{xs. \?k-list (Suc k) xs}\}\) =
    \(\{\lambda (\text{xs, n}). n \neq \text{xs}\} \cup ((\lambda \text{xs. \{xs\} \times (A - set xs)}) \cup \{\text{xs. \?k-list k xs}\}\)
    by (auto simp: length-Suc-conv)
  moreover have Suc (card A - Suc k) = card A - k using Suc.prems by simp
  then have \(\text{card} A - k \ast \prod\{\text{Suc (card A - k)}, \text{card A}\}\) = \(\prod\{\text{Suc (card A - Suc k)}, \text{card A}\}\)
    by (subst prod.insert[symmetric]) (simp add: atLeastAtMost-insertL)+
  ultimately show \(?\text{case}\)
THEORY "List"

by (simp add: card-image inj-Cons card-UN-disjoint Suc.hyps algebra-simps)

qed

lemma card-lists-distinct-length-eq':
  assumes k < card A
  shows card \{xs. length xs = k ∧ distinct xs ∧ set xs ⊆ A\} = \prod\{card A - k + 1 .. card A\}
proof -
  from k < card A have finite A and k ≤ card A using card-infinite by force+
  from this show ?thesis by (rule card-lists-distinct-length-eq)

qed

lemma infinite-UNIV-listI: ¬ finite(UNIV::'a list set)
  by (metis UNIV-I finite-maxlen length-replicate less-irrefl)

lemma same-length-different:
  assumes xs ≠ ys and length xs = length ys
  shows ∃pre x xs' y ys'. x≠y ∧ xs = pre @ [x] @ xs' ∧ ys = pre @ [y] @ ys'
using assms
proof (induction xs arbitrary: ys)
  case Nil
  then show ?case by auto
next
  case (Cons x xs)
  then obtain z zs where ys: ys = Cons z zs
    by (metis length-Suc-conv)
  show ?case
  proof (cases x=z)
    case True
    then have xs ≠ zs length xs = length zs
      using Cons.prems ys by auto
    then obtain pre u xs' v ys' where u≠v and xs: xs = pre @ [u] @ xs' and zs:
      zs = pre @ [v] @ ys'
      using Cons.IH by meson
    then have x ≠ xs = (z#pre) @ [u] @ xs' ∧ ys = (z#pre) @ [v] @ ys'
      by (simp add: True ys)
    with u≠v show ?thesis
      by blast
  next
    case False
    then have x ≠ xs = [] @ [x] @ xs ∧ ys = [] @ [z] @ zs
      by (simp add: ys)
    then show ?thesis
      using False by blast
  qed

qed
66.2 Sorting

66.2.1 sorted-wrt

Sometimes the second equation in the definition of sorted-wrt is too aggressive because it relates each list element to all its successors. Then this equation should be removed and sorted-wrt2-simps should be added instead.

lemma sorted-wrt1: sorted-wrt P [x] = True
by(simp)

lemma sorted-wrt2: transp P \implies\ sorted-wrt P (x # y # zs) = (P x y \land\ sorted-wrt P (y # zs))
proof (induction zs arbitrary; x y)
  case (Cons z zs)
  then show \_case
    by simp (meson transpD)+
qed auto

lemmas sorted-wrt2-simps = sorted-wrt1 sorted-wrt2

lemma sorted-wrt-true [simp]:
  sorted-wrt (λ- -. True) xs
by (induction xs) simp-all

lemma sorted-wrt-append:
  sorted-wrt P (xs @ ys) \iff\ sorted-wrt P xs \land\ sorted-wrt P ys \land\ (\forall x \in\ set xs. \forall y \in\ set ys. P x y)
by (induction xs) auto

lemma sorted-wrt-map:
  sorted-wrt R (map f xs) = sorted-wrt (λx y. R (f x) (f y)) xs
by (induction xs) simp-all

lemma
  assumes sorted-wrt f xs
  shows sorted-wrt-take: sorted-wrt f (take n xs)
  and sorted-wrt-drop: sorted-wrt f (drop n xs)
proof
  from assms have sorted-wrt f (take n xs @ drop n xs) by simp
  thus sorted-wrt f (take n xs) and sorted-wrt f (drop n xs)
    unfolding sorted-wrt-append by simp-all
qed

lemma sorted-wrt-filter:
  sorted-wrt f xs \implies\ sorted-wrt f (filter P xs)
by (induction xs) auto

lemma sorted-wrt-rev:
  sorted-wrt P (rev xs) = sorted-wrt (λx y. P y x) xs
by (induction xs) (auto simp add: sorted-wrt-append)

lemma sorted-wrt-mono-rel:
\((\forall x y. [ x \in \text{set} \; xs; \; y \in \text{set} \; xs; \; P \; x \; y ] \implies Q \; x \; y) \implies \text{sorted-wrt} \; P \; xs \implies \text{sorted-wrt} \; Q \; xs)\)
by(induction xs)(auto)

lemma sorted-wrt01: length xs \leq 1 \implies \text{sorted-wrt} \; P \; xs
by(auto simp: le-Suc-eq length-Suc-conv)

lemma sorted-wrt-iff-nth-less:
\((\text{sorted-wrt} \; P \; xs) = (\forall i \; j. \; i < j \implies j < \text{length} \; xs \implies P \; (xs \; ! \; i) \; (xs \; ! \; j))\)
by (induction xs) (auto simp: in-set-conv-nth Ball-def nth-Cons split: nat.split)

lemma sorted-wrt-nth-less:
\([ \text{sorted-wrt} \; P \; xs; \; i < j; \; j < \text{length} \; xs ] \implies P \; (xs \; ! \; i) \; (xs \; ! \; j)\)
by(auto simp: sorted-wrt-iff-nth-less)

lemma sorted-wrt-upt[simp]: \text{sorted-wrt} \; (\lt) \; [m..n]
by(induction n) (auto simp: sorted-wrt-append)

lemma sorted-wrt-upto[simp]: \text{sorted-wrt} \; (\lt) \; [i..j]
proof(induct i j rule: upto.induct)
  case (1 i j)
  from this show ?case by simp
next
  case snoc
  thus ?case by simp
qed

Each element is greater or equal to its index:

lemma sorted-wrt-less-idx:
\text{sorted-wrt} \; (\lt) \; ns \implies i < \text{length} \; ns \implies i \leq \text{ns}!i
proof (induction ns arbitrary: i rule: rev-induct)
  case Nil thus ?case by simp
next
  case snoc
  thus ?case by auto
  by (auto simp: nth-append sorted-wrt-append)
  (metis less-antisym not-less nth-mem)
qed

66.2.2 \textit{sorted}

context linorder
begin

Sometimes the second equation in the definition of \textit{sorted} is too aggressive because it relates each list element to all its successors. Then this equation should be removed and \textit{sorted2-simps} should be added instead. Executable code is one such use case.
lemma sorted1: sorted [x] = True
by simp

lemma sorted2: sorted (x ≠ y ≠ zs) = (x ≤ y ∧ sorted (y ≠ zs))
by (induction zs) auto

lemmas sorted2-simps = sorted1 sorted2

lemmas [code] = sorted.simps(1) sorted2-simps

lemma sorted-append:
    sorted (xs @ ys) = (sorted xs ∧ sorted ys ∧ (∀ x ∈ set xs. ∀ y ∈ set ys. x ≤ y))
by (simp add: sorted-sorted-wrt sorted-wrt-append)

lemma sorted-map:
    sorted (map f xs) = sorted-wrt (λ x y. f x ≤ f y) xs
by (simp add: sorted-sorted-wrt sorted-wrt-map)

lemma sorted01: length xs ≤ 1 ⇒ sorted xs
by (simp add: sorted-sorted-wrt sorted-wrt01)

lemma sorted-tl:
    sorted xs ⇒ sorted (tl xs)
by (cases xs) (simp-all)

lemma sorted-iff-nth-mono-less:
    sorted xs = (∀ i j. i < j −→ j < length xs −→ xs ! i ≤ xs ! j)
by (simp add: sorted-sorted-wrt sorted-iff-nth-less)

lemma sorted-iff-nth-mono:
    sorted xs = (∀ i j. i ≤ j −→ j < length xs −→ xs ! i ≤ xs ! j)
by (auto simp: sorted-iff-nth-mono-less nat-less-le)

lemma sorted-nth-mono:
    sorted xs ⇒ i ≤ j ⇒ j < length xs ⇒ xs ! i ≤ xs ! j
by (auto simp: sorted-iff-nth-mono)

lemma sorted-rev-nth-mono:
    sorted (rev xs) ⇒ i ≤ j ⇒ j < length xs ⇒ xs ! i ≤ xs ! j
using sorted-iff-nth-mono[ of rev xs length xs − j − 1 length xs − i − 1]
    rev-nth[of length xs − i − 1 xs] rev-nth[of length xs − j − 1 xs]
by auto

lemma sorted-map-remove1:
    sorted (map f xs) ⇒ sorted (map f (remove1 x xs))
by (induct xs) (auto)

lemma sorted-remove1: sorted xs ⇒ sorted (remove1 a xs)
using sorted-map-remove1 [of λx. x] by simp
lemma sorted-butlast:
  assumes \( xs \neq [] \) and \( \text{sorted} \; xs \)
  shows \( \text{sorted} \; (\text{butlast} \; xs) \)
proof −
  from \( xs \neq [] \) obtain \( ys \; y \) where \( xs = ys \; @ \; [y] \)
    by (cases \; xs \; rule: \; rev-cases) auto
  with \( \text{sorted} \; xs \) show \( \text{thesis} \)
    by (simp add: \; \text{sorted-append})
qed

lemma sorted-replicate \( [\text{simp}] \): \( \text{sorted} \; (\text{replicate} \; n \; x) \)
by (induction \; n) (auto)

lemma sorted-remdups[\text{simp}]:
  \( \text{sorted} \; xs \implies \text{sorted} \; (\text{remdups} \; xs) \)
by (induct \; xs) (auto)

lemma sorted-remdups-adj[\text{simp}]:
  \( \text{sorted} \; xs \implies \text{sorted} \; (\text{remdups-adj} \; xs) \)
by (induct \; xs \; rule: \; remdups-adj.induct, \; simp-all \; split: \; if-split-asn)

lemma sorted-nths: \( \text{sorted} \; xs \implies \text{sorted} \; (\text{nths} \; xs \; I) \)
by (induction \; xs \; arbitrary: \; I) (auto \; simp: \; \text{nths-Cons})

lemma sorted-distinct-set-unique:
  assumes \( \text{sorted} \; xs \; \text{distinct} \; xs \; \text{sorted} \; ys \; \text{distinct} \; ys \; \text{set} \; xs = \; \text{set} \; ys \)
  shows \( xs = ys \)
proof −
  from \; assms have \( 1: \; \text{length} \; xs = \; \text{length} \; ys \)
    by (auto dest!: \; \text{distinct-card})
  from \; assms show \( \text{thesis} \)
    proof (induct \; rule: \; list-induct2[\text{OF} \; I])
      case \( 1 \) show \( \text{case} \) by simp
    next
      case \( 2 \) thus \( \text{case} \) by simp (metis \; \text{Diff-insert-absorb} \; \text{antisym} \; \text{insertE} \; \text{insert-iff})
    qed
  qed

lemma map-sorted-distinct-set-unique:
  assumes \( \text{inj-on} \; f \; (\text{set} \; xs \; \cup \; \text{set} \; ys) \)
  assumes \( \text{sorted} \; (\text{map} \; f \; xs) \; \text{distinct} \; (\text{map} \; f \; xs) \)
    \( \text{sorted} \; (\text{map} \; f \; ys) \; \text{distinct} \; (\text{map} \; f \; ys) \)
  assumes \( \text{set} \; xs = \; \text{set} \; ys \)
  shows \( xs = ys \)
proof −
  from \; assms have \( \text{map} \; f \; xs = \; \text{map} \; f \; ys \)
    by (simp \; add: \; \text{sorted-distinct-set-unique})
  with \( \text{inj-on} \; f \; (\text{set} \; xs \; \cup \; \text{set} \; ys) \)
    show \( xs = ys \)
    by (blast \; intro: \; \text{map-inj-on})
  qed
lemma
assumes sorted xs
shows sorted-take: sorted (take n xs)
and sorted-drop: sorted (drop n xs)
proof
  from assms have sorted (take n xs @ drop n xs) by simp
then show sorted (take n xs) and sorted (drop n xs)
    unfolding sorted-append by simp-all
qed

lemma sorted-dropWhile: sorted xs ==\> sorted (dropWhile P xs)
  by (auto dest: sorted-drop simp add: dropWhile-eq-drop)

lemma sorted-takeWhile: sorted xs ==\> sorted (takeWhile P xs)
  by (subst takeWhile-eq-take) (auto dest: sorted-take)

lemma sorted-filter:
sorted (map f xs) ==\> sorted (map f (filter P xs))
by (induct xs) simp-all

lemma foldr-max-sorted:
assumes sorted (rev xs)
shows foldr max xs y = (if xs = [] then y else max (xs ! 0) y)
using assms
proof (induct xs)
  case (Cons x xs)
  then have sorted (rev xs) using sorted-append by auto
  with Cons show ?case
    by (cases xs) (auto simp add: sorted-append max-def)
qed simp

lemma filter-equals-takeWhile-sorted-rev:
assumes sorted: sorted (rev (map f xs))
shows filter (\lambda x. t < f x) xs = takeWhile (\lambda x. t < f x) xs
(is filter ?P xs = ?tW)
proof (rule takeWhile-eq-filter[symmetric])
  let ?dW = dropWhile ?P xs
  fix x assume x \in set ?dW
  then obtain i where i: i < length ?dW and nth-i: x = ?dW ! i
    unfolding in-set-conv-nth by auto
  hence length ?tW + i < length (?tW @ ?dW)
    unfolding length-append by simp
  hence i': length (map f ?tW) + i < length (map f xs) by simp
  have (map f ?tW @ map f ?dW) ! (length (map f ?tW) + i) \leq
    (map f ?tW @ map f ?dW) ! (length (map f ?tW) + 0)
    using sorted-rev-nth-mono[OF sorted - i', of length ?tW]
  unfolding map-append[symmetric] by simp
  hence f x \leq f (?dW ! 0)
unfolding nth-append-length-plus nth-i
using i preorder-class.le-less-trans[OF le0 i] by simp
also have ... ≤ t
using hd-dropWhile[of ?P xs] le0[THEN preorder-class.le-less-trans, OF i]
using hd-cone-nth[of ?dW] by simp
finally show ¬ t < f x by simp
qed

lemma sorted-map-same:
  sorted (map f (filter (λx. f x = g xs) xs))
proof (induct xs arbitrary: g)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  then have sorted (map f (filter (λy. f y = (λxs. f x) xs) xs)) .
  moreover from Cons have sorted (map f (filter (λy. f y = (g o Cons x) xs) xs)) .
  ultimately show ?case by simp-all
qed

lemma sorted-same:
  sorted (filter (λx. x = g xs) xs)
using sorted-map-same [of λx. x] by simp
end

lemma sorted-upt[simp]: sorted [m..<n]
by(simp add: sorted-sorted-wrt sorted-wrt-mono-rel[OF - sorted-wrt-upt])

lemma sorted-upto[simp]: sorted [m..n]
by(simp add: sorted-sorted-wrt sorted-wrt-mono-rel[OF - sorted-wrt-upto])

66.2.3 Sorting functions
Currently it is not shown that sort returns a permutation of its input because
the nicest proof is via multisets, which are not part of Main. Alternatively
one could define a function that counts the number of occurrences of an
element in a list and use that instead of multisets to state the correctness
property.
context linorder
begin

lemma set-insort-key:
  set (insort-key f x xs) = insert x (set xs)
by (induct xs) auto

lemma length-insort [simp]:
  length (insort-key f x xs) = Suc (length xs)
theory "List"

by (induct xs) simp-all

lemma insort-key-left-comm:
  assumes f x ≠ f y
  shows insort-key f y (insort-key f x xs) = insort-key f x (insort-key f y xs)
by (induct xs) (auto simp add: assms dest: antisym)

lemma insort-left-comm:
  insort x (insort y xs) = insort y (insort x xs)
by (cases x = y) (auto intro: insort-key-left-comm)

lemma comp-fun-commute-insort: comp-fun-commute insort
proof
  qed (simp add: insort-left-comm fun-eq-iff)

lemma sort-key-simps [simp]:
  sort-key f [] = []
  sort-key f (x#xs) = insort-key f x (sort-key f xs)
by (simp-all add: sort-key-def)

lemma sort-key-conv-fold:
  assumes inj-on f (set xs)
  shows sort-key f xs = fold (insort-key f) xs []
proof —
  have fold (insort-key f) (rev xs) = fold (insort-key f) xs []
  proof (rule fold-rev, rule ext)
    fix zs
    fix x y
    assume x ∈ set xs y ∈ set xs
    with assms have *: f y = f x ⟹ y = x by (auto dest: inj-onD)
    have **: x = y ⟹ y = x by auto
    show (insort-key f y o insort-key f x) zs = (insort-key f x o insort-key f y) zs
      by (induct zs) (auto intro: * simp add: **) 
  qed
  then show ?thesis by (simp add: sort-key-def foldr-conv-fold)
  qed

lemma sort-conv-fold:
  sort xs = fold insort xs []
by (rule sort-key-conv-fold) simp

lemma length-sort[simp]: length (sort-key f xs) = length xs
by (induct xs, auto)

lemma set-sort[simp]: set(sort-key f xs) = set xs
by (induct xs) (simp-all add: set-insort-key)

lemma distinct-insort: distinct (insort-key f x xs) = (x ∉ set xs ∧ distinct xs)
by(induct xs)(auto simp: set-insort-key)
lemma distinct-sort [simp]: distinct (sort-key f xs) = distinct xs
by (induct xs) (simp-all add: distinct-insort)

lemma sorted-insort-key: sorted (map f (insort-key f x xs)) = sorted (map f xs)
by (induct xs) (auto simp: sorted-insort-key)

lemma sorted-insort: sorted (insort x xs) = sorted xs
using sorted-insort-key [where f=λx. x] by simp

theorem sorted-sort-key [simp]: sorted (map f (sort-key f xs))
by (induct xs) (auto simp: sorted-insort-key)

theorem sorted-sort [simp]: sorted (sort xs)
using sorted-sort-key [where f=λx. x] by simp

lemma insort-not-Nil [simp]:
insort-key f a xs ≠ []
by (induction xs) simp-all

lemma insort-is-Cons: ∀x∈set xs. f a ≤ f x ⇒ insort-key f a xs = a # xs
by (cases xs) auto

lemma sorted-sort-id: sorted xs ⇒ sort xs = xs
by (induct xs) (auto simp add: insort-is-Cons)

lemma insort-key-remove1:
assumes a ∈ set xs and sorted (map f xs) and hd (filter (λx. f a = f x) xs) = a
shows insort-key f a (remove1 a xs) = xs
using assms proof (induct xs)
case (Cons x xs)
then show ?case
proof (cases x = a)
case False
then have f x ≠ f a using Cons.prems by auto
then have f x < f a using Cons.prems by auto
with (f x ≠ f a) show ?thesis using Cons by (auto simp: insort-is-Cons)
qed (auto simp: insort-is-Cons)
qed simp

lemma insort-remove1:
assumes a ∈ set xs and sorted xs
shows insort a (remove1 a xs) = xs
proof (rule insort-key-remove1)
define n where n = length (filter (λx. a = x) xs) - 1
from (a ∈ set xs) show a ∈ set xs .
from (sorted xs) show sorted (map (λx. a) xs) by simp
from (a ∈ set xs) have a ∈ set (filter (λx. a = x) xs) by auto
then have set (filter ((=) a) xs) ≠ {} by auto
then have filter ((=) a) xs ≠ [] by (auto simp only: set-empty)
then have length (filter ((=) a) xs) > 0 by simp
then have n: Suc n = length (filter ((=) a) xs) by (simp add: n-def)
moreover have replicate (Suc n) a = a # replicate n a
  by simp
ultimately show hd (filter ((=) a) xs) = a by (simp add: replicate-length-filter)
qed

lemma finite-sorted-distinct-unique:
  assumes finite A shows ∃!xs. set xs = A ∧ sorted xs ∧ distinct xs
proof –
  obtain xs where distinct xs A = set xs
    using finite-distinct-list [OF assms]
  then show ?thesis
    by (rule-tac a=sort xs in ex1I)
  (auto simp add: sorted-distinct-set-unique)
qed

lemma insort-insert-key-triv:
  f x ∈ f ' set xs ⇒ insort-insert-key f x xs = xs
  by (simp add: insort-insert-key-def)

lemma insort-insert-triv:
  x ∈ set xs ⇒ insort-insert x xs = xs
  using insort-insert-key-triv [of λx. x] by simp

lemma insort-insert-insort-key:
  f x /∈ f ' set xs ⇒ insort-insert-key f x xs = insort-key f x xs
  by (simp add: insort-insert-key-def)

lemma insort-insert-insort:
  x /∈ set xs ⇒ insort-insert x xs = insort x xs
  using insort-insert-insort-key [of λx. x] by simp

lemma set-insort-insert:
  set (insort-insert x xs) = insert x (set xs)
  by (auto simp add: insort-insert-key-def set-insert-key)

lemma distinct-insort-insert:
  assumes distinct xs
  shows distinct (insort-insert-key f xs)
  using assms by (induct xs) (auto simp add: insort-insert-key-def set-insert-key)

lemma sorted-insort-insert-key:
  assumes sorted (map f xs)
  shows sorted (map f (insort-insert-key f xs))
  using assms by (simp add: insort-insert-key-def sorted-insert-key)

lemma sorted-insort-insert:
assumes sorted xs
shows sorted (insert-insert x xs)
using assms sorted-insert-insert-key [of \lambda x. x] by simp

lemma filter-insert-triv:
\neg P x \implies filter P (insert-key f x xs) = filter P xs
by (induct xs) simp-all

lemma filter-insert:
sorted (map f xs) \implies P x \implies filter P (insert-key f x xs) = insert-key f x (filter P xs)
by (induct xs) (auto, subst insert-is-Cons, auto)

lemma filter-sort:
filter P (sort-key f xs) = sort-key f (filter P xs)
by (induct xs) (simp-all add: filter-insert-triv filter-insert)

lemma remove1-insert [simp]:
remove1 x (insert x xs) = xs
by (induct xs) simp-all

end

lemma sort-upt [simp]: sort [m..<n] = [m..<n]
by (rule sorted-sort-id) simp

lemma sort-upto [simp]: sort [i..j] = [i..j]
by (rule sorted-sort-id) simp

lemma sorted-find-Min:
sorted xs \implies \exists x \in set xs. P x \implies List.find P xs = Some (Min \{ x \in set xs. P x \})
proof (induct xs)
case Nil then show \?case by simp
next
case (Cons x xs) show \?case proof (cases P x)
case True
with Cons show \?thesis by (auto intro: Min-eqI [symmetric])
next
case False then have \{ y. (y = x \lor y \in set xs) \land P y \} = \{ y \in set xs. P y \}
by auto
with Cons False show \?thesis by (simp-all)
qed
qed

lemma sorted-enumerate [simp]: sorted (map fst (enumerate n xs))
by (simp add: enumerate-eq-zip)

Stability of sort-key:
lemma sort-key-stable: filter (λy. f y = k) (sort-key f xs) = filter (λy. f y = k) xs
by (induction xs) (auto simp: filter-insert sort-is-Cons filter-insert-triv)

corollary stable-sort-key-sort-key: stable-sort-key sort-key
by (simp add: stable-sort-key-def sort-key-stable)

lemma sort-key-const: sort-key (λx. c) xs = xs
by (metis (mono-tags) filter-True sort-key-stable)

66.2.4 transpose on sorted lists

lemma sorted-transpose[simp]: sorted (rev (map length (transpose xs)))
by (auto simp: sorted-iff-nth-mono rev-nth nth-transpose
length-filter-conv-card intro: card-mono)

lemma transpose-max-length:
foldr (λzs. max (length zs)) (transpose xs) 0 = length (filter (λx. x ≠ []) xs)
(is ?L = ?R)
proof (cases transpose xs = [])
case False
have ?L = foldr max (map length (transpose xs)) 0
  by (simp add: foldr-map comp-def)
also have ... = length (transpose xs ! 0)
  using False sorted-transpose by (simp add: foldr-max-sorted)
finally show ?thesis
  using False by (simp add: nth-transpose)
next
case True
hence filter (λx. x ≠ []) xs = []
  by (auto intro!: filter-False simp: transpose-empty)
thus ?thesis by (simp add: transpose-empty True)
qed

lemma length-transpose-sorted:
fixes xs :: 'a list list
assumes sorted: sorted (rev (map length xs))
shows length (transpose xs) = (if xs = [] then 0 else length (xs ! 0))
proof (cases xs = [])
case False
thus ?thesis
  using foldr-max-sorted[of sorted] False
  unfolding length-transpose foldr-map comp-def
  by simp
qed simp

lemma nth-nth-transpose-sorted[simp]:
fixes xs :: 'a list list
assumes sorted: sorted (rev (map length xs))
and i: i < length (transpose xs)
and \( j < \text{length} \) (filter (\( \lambda ys. \ i < \text{length} \ ys \) \( xs \))

\[ j \text{ filter-equals-takeWhile-sorted-rev} \] (OF sorted, of \( i \))

\[ \text{nth-transpose} [\{ i \}] \ \text{nth-map} [\{ j \}] \]

by (simp add: takeWhile-nth)

lemma transpose-column-length:

fixes \( xs :: \ a \ \text{list list} \)

assumes sorted: sorted (\( \text{rev} (\text{map} \ \text{length} \ xs) \)) and \( i < \text{length} \ xs \)

shows \( \text{length} \) (\( \lambda ys. \ i < \text{length} \ ys \) (\( \text{transpose} \ xs \))) = \( \text{length} \) (\( xs ! i \))

proof

have \( \{ j. \ j < \text{length} \ (\text{transpose} \ xs) \land i < \text{length} \ (\text{transpose} \ xs ! j) \} \)

= \( \{..<\text{length} \ (xs ! i) \} \)

proof safe

fix \( j \)

assume \( j < \text{length} \ (\text{transpose} \ xs) \) and \( i < \text{length} \ (\text{transpose} \ xs ! j) \)

with \( \text{this(2)} \) \( \text{nth-transpose} [\{ \text{this(1)} \}] \)

have \( i < \text{length} \ (\text{takeWhile} (\\lambda ys. j < \text{length} ys) xs) \) by simp

from \( \text{nth-mem} [\{ \text{this} \}] \ \text{takeWhile-nth} [\{ \text{this} \}] \)

show \( j < \text{length} \ (xs ! i) \) by (auto dest: set-takeWhileD)

next

fix \( j \)

assume \( j < \text{length} \ (xs ! i) \)

thus \( j < \text{length} \ (\text{transpose} \ xs) \)

using \( \text{foldr-max-sorted} [\{ \text{OF \ sorted} \}] \ \{xs \neq []\} \ \text{sortedE} [\{ \text{OF \ le0} \}] \)

by (auto simp: length-transpose comp-def foldr-map)

have \( \text{Suc} \ i \leq \text{length} \ (\text{takeWhile} (\\lambda ys. j < \text{length} ys) xs) \)

using \( i < \text{length} xs \) \( \langle j < \text{length} (xs ! i) \rangle \) \( \text{less-Suc-eq-le} \)

by (auto intro!: length-takeWhile-less-P-nth \ dest!: sortedE)

with \( \text{nth-transpose} [\{ j \}] \ (\text{transpose} \ xs ! j) \)

show \( i < \text{length} (\text{transpose} xs ! j) \) by simp

qed

thus \( \text{thesis} \) by (simp add: length-filter-conv-card)

qed

lemma transpose-column:

fixes \( xs :: \ a \ \text{list list} \)

assumes sorted: sorted (\( \text{rev} (\text{map} \ \text{length} \ xs) \)) and \( i < \text{length} \ xs \)

shows \( \text{map} (\\lambda ys. ys ! i) \ (\text{filter} (\\lambda ys. i < \text{length} ys) (\text{transpose} \ xs)) \)

= \( xs ! i \) (is \( ?R = - \))

proof (rule nth-equalityI)

show \( \text{length} \) (\( \text{length} ?R = \text{length} \ (xs ! i) \))

using transpose-column-length [\{ \text{OF \ assms} \}] by simp
fix j assume j: j < length ?R

note * = less-le-trans[OF this, unfolded length-map, OF length-filter-le]

from j have j-less: j < length (xs ! i) using length by simp

have i-less-tW: Suc i ≤ length (takeWhile (λys Suc j ≤ length ys) xs)

proof (rule length-takeWhile-less-P-nth)

  show Suc i ≤ length xs using i < length xs by simp

  fix k assume k < Suc i
  hence k ≤ i by auto

  with sorted-rev-nth-mono[OF sorted this] i < length xs

  have length (xs ! i) ≤ length (xs ! k) by simp

  thus Suc j ≤ length (xs ! k) using j-less by simp
qed

lemma transpose-transpose:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs))
  shows transpose (transpose xs) = takeWhile (λx. x ≠ []) xs (is ?trans = ?map)

proof -

  have sorted: sorted (rev (map length xs)) using length-map

  from foldr-max-sorted[OF this assms]

  have length ?L = length ?R

  unfolding length-transpose transpose-max-length

  using filter-equals-takeWhile-sorted-rev[OF sorted, of 0]

  by simp

  { fix i assume i < length ?R
    with less-le-trans[OF - length-takeWhile-le[of - xs]]
    have i < length xs by simp
  }

  note * = this

  show ?thesis
    by (rule nth-equalityI)

    (simp-all add: len nth-transpose transpose-column[OF sorted] * takeWhile-nth)

qed

theorem transpose-rectangle:
  assumes xs = [] ⇒ n = 0
  assumes rect: ∀ i. i < length xs ⇒ length (xs ! i) = n
  shows transpose xs = map (λ i. map (λ j. xs ! j ! i) [0..<length xs]) [0..<n]

  (is ?trans = ?map)

proof (rule nth-equalityI)

  have sorted: sorted (rev (map length xs)) using length-map

  from foldr-max-sorted[OF this assms]
show \text{len} : \text{length} ?\text{trans} = \text{length} ?\text{map}

  \text{by} (\text{simp-all add: length-transpose foldr-map comp-def})

moreover

  \{ \text{fix} i \text{ assume} i < n \text{ hence} \text{filter} (\lambda\text{ys} . i < \text{length} \text{ys}) \text{xs} = \text{xs} \\
      \text{using} \text{rect} \text{by} (\text{auto simp: in-set-conv-nth intro: filter-True}) \} 

ultimately show \(\forall i. i < \text{length} (\text{transpose} \text{xs}) \Rightarrow ?\text{trans} ! i = ?\text{map} ! i\)

  \text{by} (\text{auto simp: nth-transpose intro: nth-equalityI})

qed

66.2.5 sorted-list-of-set

This function maps (finite) linearly ordered sets to sorted lists. Warning: in most cases it is not a good idea to convert from sets to lists but one should convert in the other direction (via \text{set}).

c\text{ontext linorder

begin

definition sorted-list-of-set :: \text{'}a set \Rightarrow \text{'}a list where

  sorted-list-of-set = \text{folding.}F \text{insort} []

sublocale sorted-list-of-set: folding insort Nil

rewrites

  folding.\text{F insort} [] = sorted-list-of-set

proof

  \text{interpret} \text{comp-fun-commute insort} \text{by} (\text{fact comp-fun-commute-insort})

  show \text{folding insort} by \text{standard} (\text{fact comp-fun-commute})

  show folding.\text{F insort} [] = \text{sorted-list-of-set} by (\text{simp only: sorted-list-of-set-def})

qed

lemma sorted-list-of-set-empty:

  sorted-list-of-set \{\} = []

  \text{by} (\text{fact sorted-list-of-set.empty})

lemma sorted-list-of-set-insert \[simp\]:

  \text{finite} A \Rightarrow \text{sorted-list-of-set} (\text{insert} x A) = \text{insort} x (\text{sorted-list-of-set} (A - \{x\}))

  \text{by} (\text{fact sorted-list-of-set.insert-remove})

lemma sorted-list-of-set-eq-Nil-iff \[simp\]:

  \text{finite} A \Rightarrow \text{sorted-list-of-set} A = [] \longleftrightarrow A = \{\}

  \text{by} (\text{auto simp: sorted-list-of-set.remove})

lemma set-sorted-list-of-set \[simp\]:

  \text{finite} A \Rightarrow \text{set} (\text{sorted-list-of-set} A) = A

  \text{by(induct A rule: finite-induct) (simp-all add: set-insort-key})

lemma sorted-sorted-list-of-set \[simp\]:

  \text{sorted (sorted-list-of-set} A)

proof (cases \text{finite} A)

  case True thus \text{thesis} \text{by(induction A) (simp-all add: sorted-insort)}
next
  case False thus ?thesis by simp
qed

lemma distinct-sorted-list-of-set [simp]: distinct (sorted-list-of-set A)
proof (cases finite A)
  case True thus ?thesis by (induction A) (simp-all add: distinct-insort)
next
  case False thus ?thesis by simp
qed

lemmas sorted-list-of-set = set-sorted-list-of-set sorted-sorted-list-of-set distinct-sorted-list-of-set

lemma sorted-list-of-set-sort-remdups [code]:
  sorted-list-of-set (set xs) = sort (remdups xs)
proof –
  interpret comp-fun-commute insort by (fact comp-fun-commute-insort)
  show ?thesis by (simp add: sorted-list-of-set.eq-fold sort-conv-fold fold-set-fold-remdups)
qed

lemma sorted-list-of-set-remove: 
  assumes finite A
  shows sorted-list-of-set (A − {x}) = remove1 x (sorted-list-of-set A)
proof (cases x ∈ A)
  case False with assms have x /∈ set (sorted-list-of-set A) by simp
  with False show ?thesis by (simp add: remove1-idem)
next
  case True then obtain B where A: A = insert x B by (rule Set.set-insert)
  with assms show ?thesis by simp
qed

end

lemma sorted-list-of-set-range [simp]:
  sorted-list-of-set {m..<n} = [m..<n]
by (rule sorted-distinct-set-unique) simp-all

lemma strict-sorted-iff: strict-sorted l ⟷ sorted l ∧ distinct l
  by (induction l) (use less-le in auto)

lemma strict-sorted-list-of-set [simp]: strict-sorted (sorted-list-of-set A)
  by (simp add: strict-sorted-iff)

lemma finite-set-strict-sorted: 
  fixes A :: 'a::linorder set
  assumes finite A
  obtains l where strict-sorted l List.set l = A length l = card A
  by (metis assms distinct-card distinct-sorted-list-of-set set-sorted-list-of-set strict-sorted-list-of-set)
66.2.6  lists: the list-forming operator over sets

inductive-set
  lists :: 'a set => 'a list set
  for A :: 'a set
where
  Nil [intro!, simp]: [] ∈ lists A
  | Cons [intro!, simp]: [a ∈ A; l ∈ lists A] ==> a#l ∈ lists A

inductive-cases listsE [elim!]: x#l ∈ lists A
inductive-cases listspE [elim!]: listsp A (x # l)

inductive-simps listsp-simps [code]:
  listsp A []
  listsp A (x # xs)

lemma listsp-mono [mono]: A ≤ B ==> listsp A ≤ listsp B
by (rule predicate1I, erule listsp.induct, blast+)

lemmas lists-mono = listsp-mono [to-set]

lemma listsp-infI:
  assumes l: listsp A l shows listsp B l ==> listsp (inf A B) l using l
by induct blast+

lemmas lists-IntI = listsp-infI [to-set]

lemma listsp-inf-eq [simp]: listsp (inf A B) = inf (listsp A) (listsp B)
proof (rule mono-inf [where f=listsp, THEN order-antisym])
  show mono listsp by (simp add: mono-def listsp-mono)
  show inf (listsp A) (listsp B) ≤ listsp (inf A B) by (blast intro!: listsp-infI)
qed

lemmas listsp-conj-eq [simp] = listsp-inf-eq [simplified inf-fun-def inf-bool-def]

lemmas lists-Int-eq [simp] = listsp-inf-eq [to-set]

lemma Cons-in-lists-iff [simp]: x#xs ∈ lists A <-> x ∈ A ∧ xs ∈ lists A
by auto

lemma append-in-lists-conv [iff]:
  (listsp A (xs @ ys)) = (listsp A xs ∧ listsp A ys)
by (induct xs) auto


lemma in-lists-conv-set: (listsp A xs) = (∀ x ∈ set xs. A x)
— eliminate listsp in favour of set
by (induct xs) auto

lemma in-listspD [dest!]: listsp A xs ==> \( \forall x \in \text{set } xs. A x \)
by (rule in-listsp-conv-set [THEN iffD1])

lemmas in-listsD [dest!] = in-listspD [to-set]

lemma in-listspI [intro!]: \( \forall x \in \text{set } xs. A x \implies \text{listsp } A xs \)
by (rule in-listsp-conv-set [THEN iffD2])

lemmas in-listsI [intro!] = in-listspI [to-set]

lemma lists-eq-set: \( \text{lists } A = \{ xs. \text{set } xs \leq A \} \)
by auto

lemma lists-empty [simp]: \( \text{lists } \{\} = \{[]\} \)
by auto

lemma lists-UNIV [simp]: \( \text{lists } \text{UNIV} = \text{UNIV} \)
by auto

lemma lists-image: \( \text{lists } (f' A) = \text{map } f' \text{ lists } A \)
proof -
  \{ fix xs have \( \forall x \in \text{set } xs. x \in f' A \implies xs \in \text{map } f' \text{ lists } A \)
  by (induct set: ListMem) auto \}
  then show \(?thesis by auto \)
qed

66.2.7 Inductive definition for membership

inductive ListMem :: 'a => 'a list => bool
where
  elem: ListMem x (x # xs)
| insert: ListMem x xs ==> ListMem x (y # xs)

lemma ListMem-iff: (ListMem x xs) = (x \in \text{set } xs)
proof
  show ListMem x xs ==> x \in \text{set } xs
  by (induct set: ListMem) auto
  show x \in \text{set } xs ==> ListMem x xs
  by (induct xs) (auto intro: ListMem.intros)
qed

66.2.8 Lists as Cartesian products

set-Cons A Xs: the set of lists with head drawn from A and tail drawn from Xs.

definition set-Cons :: 'a set => 'a list set => 'a list set where
THEORY “List”

set-Cons A XS = \{ z. \exists x xs. z = x \# xs \land x \in A \land xs \in XS \}

lemma set-Cons-sing-Nil [simp]: set-Cons A \{\} = (%x. \{x\})'A
by (auto simp add: set-Cons-def)

Yields the set of lists, all of the same length as the argument and with elements drawn from the corresponding element of the argument.

primrec listset :: 'a set list \Rightarrow 'a list set
where
listset [] = {[]} | listset (A # As) = set-Cons A (listset As)

66.3 Relations on Lists

66.3.1 Length Lexicographic Ordering

These orderings preserve well-foundedness: shorter lists precede longer lists. These ordering are not used in dictionaries.

primrec — The lexicographic ordering for lists of the specified length
lexn :: ('a \times 'a) set \Rightarrow nat \Rightarrow ('a list \times 'a list) set where
lexn r 0 = {} |
lexn r (Suc n) = 
(map-prod (%(x, xs), x#xs) (%(x, xs), x#xs) \cdot (r <\cdot lex\cdot> lexn r n)) Int
\{(xs, ys). length xs = Suc n \land length ys = Suc n\}

definition lex :: ('a \times 'a) set \Rightarrow ('a list \times 'a list) set
where
lex r = (\union n. lexn r n) — Holds only between lists of the same length

definition lenlex :: ('a \times 'a) set => ('a list \times 'a list) set where
lenlex r = inv-image (less-than \cdot lex\cdot> lex r) (\lambda xs. (length xs, xs))
— Compares lists by their length and then lexicographically

lemma wf-lexn: assumes wf r shows wf (lexn r n)
proof (induct n)
case (Suc n)
have inj: inj (\lambda(x, xs). x \# xs)
using assms by (auto simp: inj-on-def)
have wf: wf (map-prod (\lambda(x, xs). x \# xs) (\lambda(x, xs). x \# xs) \cdot (r <\cdot lex\cdot> lexn r n))
by (simp add: Suc.hyps assms wf-map-prod-image [OF - inj])
then show ?case
by (rule wf-subset) auto
qed auto

lemma lexn-length:
(xs, ys) \in lexn r n \Longrightarrow length xs = n \land length ys = n
by (induct n arbitrary: xs ys) auto

lemma wf-lex [intro!]: wf r ==> wf (lex r)
unfolding lex-def

lemma wf-lexn [intro!]: wf r ==> wf (lexn r)
unfolding lexn-def
apply (rule wf-UN)
    apply (simp add: wf-lexn)
apply (metis DomainE Int-emptyI RangeE lexn-length)
done

lemma lexn-conv:
    lexn r n = 
    { (xs,ys). length xs = n ∧ length ys = n ∧ 
       (∃ xs x y xs' ys'. xs = xs @ x#xs' ∧ ys = ys @ y # ys' ∧ (x, y) ∈ r) }
apply (induct n, simp)
apply (simp add: image-Collect lex-prod-def, safe, blast)
apply (rule_tac x = ab # xys in exI, simp)
apply (case_tac xys, simp-all, blast)
done

By Mathias Fleury:

proposition lexn-transI:
    assumes trans r 
    shows trans (lexn r n)
unfolding trans-def
proof (intro allI impI)
    fix as bs cs
    assume asbs: (as, bs) ∈ lexn r n and bcs: (bs, cs) ∈ lexn r n
    obtain abs a b as' where
        n: length as = n and length bs = n and
        as: as = abs @ a # as' and
        bs: bs = abs @ b # bs' and
        abr: (a, b) ∈ r
        using asbs unfolding lexn-cone by blast
    obtain bcs b' c' cs' bs' where
        n': length cs = n and length bs = n and
        bs': bs = bcs @ b' # bs' and
        cs: cs = bcs @ c' # cs' and
        b'c'r: (b', c') ∈ r
        using bcs unfolding lexn-cone by blast
    consider (le) length bcs < length bs
        | (eq) length bcs = length bs
        | (ge) length bcs > length bs by linarith
    thus (as, cs) ∈ lexn r n
proof cases
    let ?k = length bcs
    case le
    hence as ! ?k = bs ! ?k unfolding as bs by (simp add: nth-append)
    hence (as ! ?k, cs ! ?k) ∈ r using b'c'r unfolding bs' cs by auto
    moreover
    have length bcs < length as using le unfolding as by simp
    from id-take-nth-drop[OF this]
    have as = take ?k as @ as ! ?k # drop (Suc ?k) as .
    moreover
    have length bcs < length cs unfolding cs by simp

    case ge
    hence as ! ?k = bs ! ?k unfolding as bs by (simp add: nth-append)
    hence (as ! ?k, cs ! ?k) ∈ r using b'c'r unfolding bs' cs by auto
    moreover
    have length bcs < length as using le unfolding as by simp
    from id-take-nth-drop[OF this]
    have as = take ?k as @ as ! ?k # drop (Suc ?k) as .
    moreover
    have length bcs < length cs unfolding cs by simp


from \textit{id-take-nth-drop}[OF this]

have \(cs = \text{take } \bar{k} \, cs \odot cs! \bar{k} \# \, \text{drop} (\text{Suc } \bar{k}) \, cs\).

moreover have \(\text{take } \bar{k} \, cs = \text{take } \bar{k} \, cs\)

using \textit{le arg-cong}[OF \, bs, \, \text{of take} (\text{length} \, bs)]

unfolding \(cs \, \text{as} \, bs'\) by auto

ultimately show \(?\text{thesis}\) using \(n \, n'\) unfolding \textit{lexn-conv} by auto

next

let \(\bar{k} = \text{length} \, abs\)

case \ge

hence \(bs! \bar{k} = cs! \bar{k}\) unfolding \(bs' \, cs\) by \(\text{simp add: nth-append}\)

hence \((as! \bar{k}, \, cs! \bar{k}) \in r\) using \textit{abr} unfolding \(as \, bs\) by auto

moreover

have \(\text{length} \, abs < \text{length} \, cs\) using \textit{ge unfolding} \(as \, \text{by simp}\)

from \textit{id-take-nth-drop}[OF this]

have \(\text{as} = \text{take } \bar{k} \, \text{as} \odot \text{as}! \bar{k} \# \text{drop} (\text{Suc } \bar{k}) \, \text{as}\).

moreover have \(\text{length} \, abs < \text{length} \, cs\) using \(n \, n'\) unfolding \(as \, \text{by simp}\)

from \textit{id-take-nth-drop}[OF this]

have \(cs = \text{take } \bar{k} \, cs \odot cs! \bar{k} \# \text{drop} (\text{Suc } \bar{k}) \, cs\).

moreover have \(\text{take } \bar{k} \, cs = \text{take } \bar{k} \, cs\)

using \textit{ge arg-cong}[OF \, bs', \, \text{of take} (\text{length} \, abs)]

unfolding \(cs \, \text{as} \, bs\) by auto

ultimately show \(?\text{thesis}\) using \(n \, n'\) unfolding \textit{lexn-conv} by auto

next

let \(\bar{k} = \text{length} \, abs\)

case \eq

hence \(\ast \) unfolding \(bs \, b' = bs' \, c\) by auto

hence \((a, c') \in r\)

using \textit{abr} \(b'c'r\) \textit{assms unfolding} \textit{trans-def} by blast

with \(\ast\) show \(?\text{thesis}\) using \(n \, n'\) unfolding \textit{lexn-conv} as \(bs \, cs\) by auto

qed

qed

corollary \textit{lex-transI}:

assumes \textit{trans} \(r\)

shows \textit{trans} \((\text{lex} \, r)\)

using \textit{lex-transI}[OF \, \textit{assms}]

by \textit{(clarsimp simp add: lex-def trans-def) (metis \textit{lexn-length})}

lemma \textit{lex-conv}:

\(\textit{lex} \, r =\)

\(\{(xs,ys). \, \text{length} \, xs = \text{length} \, ys \wedge\)

\(\exists xys \, x \, x's \, ys'. \, x = xys \odot x \# x's \wedge y = xys \odot y \# y's' \wedge (x, y) \in r\}\}

by \textit{(force simp add: lex-def lexn-conv)}

lemma \textit{wf-lenlex} [intro!]: \(\textit{wf} \, r \Longrightarrow \textit{wf} \, (\text{lenlex} \, r)\)

by \textit{(unfold lenlex-def) blast}

lemma \textit{lenlex-conv}:

\(\textit{lenlex} \, r =\)

\(\{(xs,ys). \, \text{length} \, xs < \text{length} \, ys \vee\)

\(\text{length} \, xs = \text{length} \, ys \wedge (xs, ys) \in \text{lex} \, r\}\}

by \textit{(force simp add: lex-def lexn-conv)}

lemma \textit{lex-transI}:

assumes \textit{trans} \(r\)

shows \textit{trans} \((\text{lex} \, r)\)

using \textit{lex-transI}[OF \, \textit{assms}]

by \textit{(clarsimp simp add: lex-def trans-def) (metis \textit{lexn-length})}

lemma \textit{lex-conv}:

\(\textit{lex} \, r =\)

\(\{(xs,ys). \, \text{length} \, xs = \text{length} \, ys \wedge\)

\(\exists xys \, x \, x's \, ys'. \, x = xys \odot x \# x's \wedge y = xys \odot y \# y's' \wedge (x, y) \in r\}\}

by \textit{(force simp add: lex-def lexn-conv)}

lemma \textit{wf-lenlex} [intro!]: \(\textit{wf} \, r \Longrightarrow \textit{wf} \, (\text{lenlex} \, r)\)

by \textit{(unfold lenlex-def) blast}

lemma \textit{lenlex-conv}:

\(\textit{lenlex} \, r =\)

\(\{(xs,ys). \, \text{length} \, xs < \text{length} \, ys \vee\)

\(\text{length} \, xs = \text{length} \, ys \wedge (xs, ys) \in \text{lex} \, r\}\}

by \textit{(force simp add: lex-def lexn-conv)}

lemma \textit{lex-transI}:

assumes \textit{trans} \(r\)

shows \textit{trans} \((\text{ lex} \, r)\)

using \textit{lex-transI}[OF \, \textit{assms}]

by \textit{(clarsimp simp add: lex-def trans-def) (metis \textit{lexn-length})}
by (simp add: lenlex-def Id-on-def lex-prod-def inv-image-def)

lemma total-lenlex:
assumes total r
shows total (lenlex r)
proof
  have (xs,ys) ∈ lexn r (length xs) ∨ (ys,xs) ∈ lexn r (length ys)
    if xs ≠ ys and
      length xs = length ys for xs ys
proof
  obtain pre x xs' y ys' where
    x ≠ y and
    xs = pre @ [x] @ xs' and
    ys = pre @ [y] @ ys'
    by (meson len ⟨xs ≠ ys⟩ same-length-different)
then consider (x,y) ∈ r | (y,x) ∈ r
  by (meson UNIV-I assms total-on-def)
then show ?thesis
  by cases (use len in (force simp add: lexn-conv xs ys)+)
qed

then show ?thesis
  by (fastforce simp: lenlex-def total-on-def lex-def)
qed

lemma lenlex-transI [intro]: trans r ⇒ trans (lenlex r)
  unfolding lenlex-def
  by (meson lex-transI trans-inv-image trans-less-than trans-lex-prod)

lemma Nil-notin-lex [iff]: ([], ys) ∉ lex r
  by (simp add: lex-conv)
lemma Nil2-notin-lex [iff]: (xs, []) ∉ lex r
  by (simp add: lex-conv)

lemma Cons-in-lex [simp]:
((x ≠ xs, y ≠ ys) ∈ lex r) =
  ((x, y) ∈ r ∧ length xs = length ys ∨ x = y ∧ (xs, ys) ∈ lex r)
apply (simp add: lex-conv)
apply (rule iffI)
prefer 2 apply (blast intro: Cons-eq-appendI, clarify)
by (metis hd-append append-Nil list.sel(1) list.sel(3) tl-append2)

lemma Nil-lenlex-iff1 [simp]: ([], ns) ∈ lenlex r ←→ ns ≠ []
and Nil-lenlex-iff2 [simp]: (ns,[]) ∉ lenlex r
  by (auto simp: lenlex-def)

lemma Cons-lenlex-iff:
((m ≠ ms, n ≠ ns) ∈ lenlex r) ←→
  length ms < length ns
∨ length ms = length ns ∧ (m,n) ∈ r
∨ (m = n ∧ (ms,ns) ∈ lenlex r)
  by (auto simp: lenlex-def)
THEORY "List"

lemma lenlex-length: \((ms, ns) \in \text{lenlex } r \implies \text{length } ms \leq \text{length } ns\)
by (auto simp: lenlex-def)

lemma lex-append-rightI:
\((xs, ys) \in \text{lex } r \implies \text{length } vs = \text{length } us \implies (xs @ us, ys @ vs) \in \text{lex } r\)
by (fastforce simp: lex-def lexn-conv)

lemma lex-append-leftI:
\((ys, zs) \in \text{lex } r \implies (xs @ ys, xs @ zs) \in \text{lex } r\)
by (induct xs)

lemma lex-append-leftD:
\(\forall x. (x, x) \notin r \implies (xs @ ys, xs @ zs) \in \text{lex } r \implies (ys, zs) \in \text{lex } r\)
by (induct xs)

lemma lex-append-left-iff:
\(\forall x. (x, x) \notin r \implies (xs @ ys, xs @ zs) \in \text{lex } r \iff (ys, zs) \in \text{lex } r\)
by (metis lex-append-leftD lex-append-leftI)

lemma lex-take-index:
assumes \((xs, ys) \in \text{lex } r\)
obtains \(i\) where \(i < \text{length } xs\) and \(i < \text{length } ys\) and take \(i\) xs = take \(i\) ys
and \((xs ! i, ys ! i) \in r\)
proof -
  obtain \(n us x xs' y ys'\) where \((xs, ys) \in \text{lex } n r\) and \(\text{length } xs = n\) and \(\text{length } ys = n\)
  and \(xs = us @ x # xs'\) and \(ys = us @ y # ys'\) and \((x, y) \in r\)
  using assms by (fastforce simp: lex-def lexn-conv)
  then show \(\text{thesis}\) by (intro that [of length us])
qed

66.3.2 Lexicographic Ordering

Classical lexicographic ordering on lists, ie. "a" ¡ "ab" ¡ "b". This ordering does not preserve well-foundedness. Author: N. Voelker, March 2005.

definition lexord :: ('a x 'a) set => ('a list x 'a list) set where
lexord \(r = \{(x, y). \exists a v. y = x @ a \# v \lor \exists u a b v w. (a, b) \in r \land x = u @ (a \# v) \land y = u @ (b \# w)\}\)

lemma lexord-Nil-left[simp]: \([[], y] \in \text{lexord } r = (\exists a. y = a \# x)\)
by (unfold lexord-def, induct-tac y, auto)

lemma lexord-Nil-right[simp]: \([x, []] \notin \text{lexord } r\)
by (unfold lexord-def, induct-tac x, auto)

lemma lexord-cons-cons[simp]:
\((a \# x, b \# y) \in \text{lexord } r \implies ((a, b) \in r \lor (a = b \land (x, y) \in \text{lexord } r))\)
unfolding lexord-def
apply (safe, simp-all)
  apply (metis hd-append list.sel(1))
  apply (metis hd-append list.sel(1) list.sel(3) tl-append2)
apply blast
by (meson Cons-eq-appendI)

lemmas lexord-simps = lexord-Nil-left lexord-Nil-right lexord-cons-cons

lemma lexord-append-rightI: \( \exists\ b\ z\ .\ y = b \# z \implies (x, x@y) \in \text{lexord}\ r \)
by (induct-tac x, auto)

lemma lexord-append-left-rightI:
\( (a,b) \in r \implies u@a#x, u@b#y) \in \text{lexord}\ r \)
by (induct-tac u, auto)

lemma lexord-append-leftI: \( (u,v) \in \text{lexord}\ r \implies (x@u, x@v) \in \text{lexord}\ r \)
by (induct x, auto)

lemma lexord-append-leftD:
\[ \frac{(x@u, x@v) \in \text{lexord}\ r; (\forall a. (a,a) \not\in r)}{(u,v) \in \text{lexord}\ r}\]
by (erule rev-mp, induct-tac x, auto)

lemma lexord-take-index-conv:
\( ((x,y) \in \text{lexord}\ r) = \)
\[ ((\text{length}\ x < \text{length}\ y \land \text{take}\ (\text{length}\ x)\ y = x) \lor \]
\( (\exists\ i. i < \min(\text{length}\ x)(\text{length}\ y) \land \text{take}\ i\ x = \text{take}\ i\ y \land (x!i,y!i) \in r))\]
apply (unfold lexord-def Let-def, clarsimp)
apply (rule-tac f = (% a b. a \lor b) in arg-cong2)
apply (metis Cons-nth-drop-Suc append-eq-conv-conj drop-all list.simps(3) not-le)
apply auto
apply (rule-tac x =length u in exI, simp)
by (metis id-take-nth-drop)

— lexord is extension of partial ordering List.lex

lemma lexord-lex: \( (x,y) \in \text{lex}\ r = ((x,y) \in \text{lexord}\ r \land \text{length}\ x = \text{length}\ y)\)
proof (induction x arbitrary: y)
case (Cons a x y) then show ?case
  by (cases y) (force+)
qed auto

lemma lexord-irreflexive: \( \forall x. (x,x) \not\in r \implies (xs,xs) \not\in \text{lexord}\ r \)
by (induct xs) auto

By René Thiemann:

lemma lexord-partial-trans:
\( \frac{\forall x y z. x \in \text{set}\ xs \implies (x,y) \in r \implies (y,z) \in r \implies (x,z) \in r)}{(xs,ys) \in \text{lexord}\ r \implies (ys,zs) \in \text{lexord}\ r \implies (xs,zs) \in \text{lexord}\ r}\)
proof (induct xs arbitrary: ys zs)
case Nil
from Nil(3) show ?case unfolding lexord-def by (cases zs, auto)

next
case (Cons x xs yys zzs)
from Cons(3) obtain y ys where yys: yys = y # ys unfolding lexord-def
by (cases yys, auto)

note Cons = Cons[unfolded yys]
from Cons(4) have one: (x,y) ∈ r ∨ x = y ∧ (xs,ys) ∈ lexord r by auto
from Cons(4) obtain z zs where zzs: zzs = z # zs unfolding lexord-def
by (cases zzs, auto)

note Cons = Cons[unfolded zzs]
from Cons(4) have two: (y,z) ∈ r ∨ y = z ∧ (ys, zs) ∈ lexord r by auto

{ assume (xs,ys) ∈ lexord r and (ys,zs) ∈ lexord r
from Cons(1)|OF - this| Cons(2)
have (xs, zs) ∈ lexord r by auto
}

note ind1 = this

{ assume (x,y) ∈ r and (y,z) ∈ r
from Cons(2)|OF - this| have (x,z) ∈ r by auto
}

note ind2 = this
from one two ind1 ind2
have (x,z) ∈ r ∨ x = z ∧ (xs, zs) ∈ lexord r by blast

thus ?case unfolding zzs by auto

qed
assumes asym R
shows asym (lexord R)

proof
from assms obtain irrefl R by (blast elim: asym.cases)
then show irrefl (lexord R) by (rule lexord-irrefl)

next
fix xs ys
assume (xs, ys) ∈ lexord R
then show (ys, xs) /∈ lexord R
proof (induct xs arbitrary: ys)
  case Nil
    then show ?case by simp
next
  case (Cons x xs)
    then obtain z zs where ys: ys = z # zs by (cases ys) auto
    with assms Cons show ?case by (auto elim: asym.cases)
qed

lemma lexord-asymmetric:
assumes asym R
assumes hyp: (a, b) ∈ lexord R
shows (b, a) /∈ lexord R

proof
  from ⟨asym R |⟩ have asym (lexord R) by (rule lexord-asym)
  then show ?thesis by (rule asym.cases) (auto simp add: hyp)
qed

Predicate version of lexicographic order integrated with Isabelle’s order type classes. Author: Andreas Lochbihler

category ord

begin

context notes [[inductive-internals]]
begin

inductive lexordp :: 'a list ⇒ 'a list ⇒ bool
where
  Nil: lexordp [] (y ≠ ys)
| Cons: x < y ⇒ lexordp (x # xs) (y # ys)
| Cons-eq:
  [¬ x < y; ¬ y < x; lexordp xs ys ] ⇒ lexordp (x # xs) (y # ys)
end

lemma lexordp-simps [simp]:
lexordp [] ys = (ys ≠ [])
lexordp xs [] = False
THEORY "List"

let lexordp (x ≠ xs) (y ≠ ys) ←→ x < y ∨ ¬ y < x ∧ lexordp xs ys
by(subst lexordp.simps, fastforce simp add: neq-Nil-conv)+

inductive lexordp-eq :: ℓa list ⇒ ℓa list ⇒ bool where
  Nil: lexordp-eq [] ys
| Cons: x < y ⇒ lexordp-eq (x ≠ xs) (y ≠ ys)
| Cons-eq: [¬ x < y; ¬ y < x; lexordp-eq xs ys ] ⇒ lexordp-eq (x ≠ xs) (y ≠ ys)

lemma lexordp-eq-simps [simp]:
  lexordp-eq [] ys = True
  lexordp-eq xs [] ←→ xs = []
  lexordp-eq (x ≠ xs) [] = False
  lexordp-eq (x ≠ xs) (y ≠ ys) ←→ x < y ∨ ¬ y < x ∧ lexordp-eq xs ys
by(subst lexordp-eq.simps, fastforce)+

lemma lexordp-append-rightI: ys ≠ Nil ⇒ lexordp xs (xs @ ys)
by(induct xs) (auto simp add: neq-Nil-conv)

lemma lexordp-append-left-rightI: x < y ⇒ lexordp (us @ x ≠ xs) (us @ y ≠ ys)
by(induct us) auto

lemma lexordp-eq-refl: lexordp-eq xs xs
by(induct xs) simp-all

lemma lexordp-append-leftI: lexordp us vs ⇒ lexordp (xs @ us) (xs @ vs)
by(induct xs) auto

lemma lexordp-append-leftD: [ lexordp (xs @ us) (xs @ vs); ∀ a. ¬ a < a ] ⇒ lexordp us vs
by(induct xs) auto

lemma lexordp-irreflexive:
  assumes irrefl: ∀ x. ¬ x < x
  shows ¬ lexordp xs xs
proof
  assume lexordp xs xs
  thus False by(induct xs ys≡xs)(simp-all add: irrefl)
qed

lemma lexordp-into-lexordp-eq:
  assumes lexordp xs ys
  shows lexordp-eq xs ys
using assms by induct simp-all

end

declare ord.lexordp-simps [simp, code]
THEORY "List"

declare ord.lexordp-eq-simps [code, simp]

lemma lexord-code [code, code-unfold]: lexordp = ord.lexordp less
unfolding lexordp-def ord.lexordp-def ..

class order

begin

lemma lexordp-antisym:
assumes lexordp xs ys lexordp ys xs
shows False
using assms by (induct auto)

lemma lexordp-irreflexive': ¬ lexordp xs xs
by (rule lexordp-irreflexive) simp

end

context linorder begin

lemma lexordp-cases [consumes 1, case-names Nil Cons Cons-eq, cases pred: lexordp]:
assumes lexordp xs ys
obtains (Nil) y ys' where xs = [] ys = y # ys'
| (Cons) x xs' y ys' where xs = x # xs' ys = y # ys' x < y
| (Cons-eq) x xs' ys' where xs = x # xs' ys = xs # ys' lexordp xs' ys'
using assms by cases (fastforce simp add: not-less-iff-gr-or-eq)+

lemma lexordp-induct [consumes 1, case-names Nil Cons Cons-eq, induct pred: lexordp]:
assumes major: lexordp xs ys
and Nil: ∀y ys. P [] (y # ys)
and Cons: ∀x xs y ys. x < y ⇒ P (x # xs) (y # ys)
and Cons-eq: ∀x xs ys. lexordp xs ys; P xs ys ⇒ P (x # xs) (x # ys)
shows P xs ys
using major by (induct (simp-all add: Nil Cons not-less-iff-gr-or-eq Cons-eq)

lemma lexordp-iff:
lexordp xs ys ⟷ (∃x vs. xs = x # vs) ∨ (∃ys. a < b ∧ ∃us. a # us ∧ xs = us # b ∧ ys = ys)
(is ?lhs = ?rhs)
proof
assume ?lhs thus ?rhs
proof induct
  case Cons-eq thus ?case by simp (metis append.simps(2))
qed (fastforce intro: disjI2 del: disjCI intro: ext[where x=[]])+
next
assume ?rhs thus ?lhs
by (auto intro: lexordp-append-leftI[where us=[], simplified] lexordp-append-leftI)
qed

lemma lexordp-conv-lexord:
  lexordp xs ys ←→ (xs, ys) ∈ lexord {x, y. x < y}
by (simp add: lexordp-iff lexord-def)

lemma lexordp-eq-antisym:
  assumes lexordp-eq xs ys lexordp-eq ys xs
  shows xs = ys
using assms by (induct simp-all)

lemma lexordp-eq-trans:
  assumes lexordp-eq xs ys and lexordp-eq ys zs
  shows lexordp-eq xs zs
using assms
by (induct arbitrary: zs) (case-tac zs; auto)+

lemma lexordp-linear:
  lexordp xs ys ∨ xs = ys ∨ lexordp ys xs
by (induct xs arbitrary: ys; case-tac ys; fastforce)

lemma lexordp-conv-lexordp-eq:
  lexordp xs ys ←→ lexordp-eq xs ys ∧ ¬ lexordp-eq ys xs
(is ?lhs ←→ ?rhs)
proof
  assume ?lhs
  hence ¬ lexordp-eq ys xs by (induct simp-all)
with (?lhs) show ?rhs by (simp add: lexordp-into-lexordp-eq)
next
  assume ?rhs
  hence lexordp-eq xs ys ¬ lexordp-eq ys xs by simp-all
  thus ?lhs by (induct simp-all)
qed

lemma lexordp-eq-linear:
  lexordp-eq xs ys ∨ lexordp-eq ys xs
by (induct xs arbitrary: ys) (case-tac ys; auto)+

lemma lexordp-linorder: class.linorder lexordp-eq lexordp
by unfold-locales
end

lemma sorted-insert-is-snoc: sorted xs \implies \forall x \in set xs. a \geq x \implies insert a xs = xs @ [a]
by (induct xs) (auto dest: insert-is-Cons)

66.3.3 Lexicographic combination of measure functions

These are useful for termination proofs

definition measures fs = inv-image (lex less-than) (%a. map (%f. f a) fs)

lemma wf-measures[simp]: wf (measures fs)
unfolding measures-def by blast

lemma in-measures[simp]:
(x, y) \in measures [] = False
(x, y) \in measures (f # fs)
  = (f x < f y \lor (f x = f y \land (x, y) \in measures fs))
unfolding measures-def by auto

lemma measures-less: f x < f y ==> (x, y) \in measures (f#fs)
by simp

lemma measures-lesequ: f x \leq f y ==> (x, y) \in measures fs ==> (x, y) \in measures (f#fs)
by auto

66.3.4 Lifting Relations to Lists: one element

definition listrel1 :: ('a x 'a) set \Rightarrow ('a list x 'a list) set where
listrel1 r = {
  \exists z z' vs. xs = us @ z # vs \land (z,z') \in r \land ys = us @ z' # vs
}

lemma listrel1I:
[(x, y) \in r; xs = us @ x # vs; ys = us @ y # vs] \implies (xs, ys) \in listrel1 r
unfolding listrel1-def by auto

lemma listrel1IE:
[(xs, ys) \in listrel1 r;
  \forall x y us vs. [(x, y) \in r; xs = us @ x # vs; ys = us @ y # vs] \implies P]
\implies P
unfolding listrel1-def by auto

lemma not-Nil-listrel1 [iff]: ([], xs) \notin listrel1 r
unfolding listrel1-def by blast
lemma not-listrel1-Nil [iff]: \( (xs, []) \notin \text{listrel1 } r \)
unfolding listrel1-def by blast

lemma Cons-listrel1-Cons [iff]:
\( (x \# xs, y \# ys) \in \text{listrel1 } r \iff (x,y) \in r \land x = y \land (xs, ys) \in \text{listrel1 } r \)
by (simp add: listrel1-def Cons-eq-append-conv) (blast)

lemma listrel1I1: \( (x,y) \in r \Rightarrow (x \# xs, y \# xs) \in \text{listrel1 } r \)
by fast

lemma listrel1I2: \( (xs, ys) \in \text{listrel1 } r \Rightarrow (xs, y \# ys) \in \text{listrel1 } r \)
by fast

lemma append-listrel1I:
\( (xs, ys) \in \text{listrel1 } r \land us = vs \lor xs = ys \land (us, vs) \in \text{listrel1 } r \)
\( \Rightarrow (xs @ us, ys @ vs) \in \text{listrel1 } r \)
unfolding listrel1-def
by auto (blast intro: append-eq-appendI)

lemma Cons-listrel1E1 [elim!]:
assumes \( (x \# xs, y \# ys) \in \text{listrel1 } r \)
and \( \forall y. ys = y \# xs \Rightarrow (x,y) \in r \Rightarrow R \)
and \( \forall zs. zs = x \# zs \Rightarrow (xs, zs) \in \text{listrel1 } r \Rightarrow R \)
shows \( R \)
using assms by (cases ys) blast+

lemma Cons-listrel1E2 [elim!]:
assumes \( (xs, y \# ys) \in \text{listrel1 } r \)
and \( \forall x. xs = x \# ys \Rightarrow (x,y) \in r \Rightarrow R \)
and \( \forall zs. xs = y \# zs \Rightarrow (zs, ys) \in \text{listrel1 } r \Rightarrow R \)
shows \( R \)
using assms by (cases xs) blast+

lemma snoc-listrel1-snoc-iff:
\( (xs @ [x], ys @ [y]) \in \text{listrel1 } r \)
\( \iff (xs, ys) \in \text{listrel1 } r \land x = y \lor xs = ys \land (x,y) \in r \) (is \( ?L \leftrightarrow ?R \))
proof
assume \( ?L \) thus \( ?R \)
by (fastforce simp: listrel1-def snoc-eq-iff-butlast butlast-append)
next
assume \( ?R \) then show \( ?L \) unfolding listrel1-def by force
qed

lemma listrel1-eq-len: \( (xs, ys) \in \text{listrel1 } r \Rightarrow \text{length } xs = \text{length } ys \)
unfolding listrel1-def by auto

lemma listrel1-mono:
\[ r \subseteq s \implies \text{listrel1 } r \subseteq \text{listrel1 } s \]

**unfolding** \text{listrel1-def} by blast

**lemma** \text{listrel1-converse}: \text{listrel1 } (r^{-1}) = (\text{listrel1 } r)^{-1}

**unfolding** \text{listrel1-def} by blast

**lemma** \text{in-listrel1-converse}:
\[(x,y) \in \text{listrel1 } (r^{-1}) \iff (x,y) \in (\text{listrel1 } r)^{-1}\]

**unfolding** \text{listrel1-def} by blast

**lemma** \text{listrel1-iff-update}:
\[(xs,ys) \in \text{listrel1 } r \iff (\exists n. (xs!n, y) \in r \land n < \text{length } xs \land ys = xs[n:=y]) \ (\text{is } ?L \iff ?R)\]

**proof**

assume \(?L\)
then obtain \(x y v\) where
\[xs = u \cons x \# v \quad ys = u \cons y \# v \quad (x,y) \in r\]

**unfolding** \text{listrel1-def} by auto

then have \(ys = xs[\text{length } u := y]\) and \(\text{length } u < \text{length } xs\)
and \((xs \cons \text{length } u, y) \in r\) by auto

then show \(?R\) by auto

next
assume \(?R\)
then obtain \(x y n\) where
\((xs!n, y) \in r \land n < \text{size } xs \land ys = xs[n:=y]\)

\(x = xs!n\)

by auto

then obtain \(u v\) where
\(xs = u \cons x \# v\) and \(ys = u \cons y \# v\) and \((x,y) \in r\)

by (auto intro: upd-conv-take-nth-drop id-take-nth-drop)
then show \(?L\) by (auto simp: \text{listrel1-def})

qed

Accessible part and wellfoundedness:

**lemma** \text{Cons-acc-listrel1I} [intro!]:
\[x \in \text{Wellfounded.acc } r \implies xs \in \text{Wellfounded.acc } (\text{listrel1 } r) \implies (x \# xs) \in \text{Wellfounded.acc } (\text{listrel1 } r)\]

**apply** (induct arbitrary: \(xs\) set: Wellfounded.acc)

**apply** (erule thin-rf)

**apply** (erule acc-induct)

**apply** (rule accI)

**apply** (blast)

**done**

**lemma** \text{lists-accD}: \(xs \in \text{lists } (\text{Wellfounded.acc } r) \implies xs \in \text{Wellfounded.acc } (\text{listrel1 } r)\)

**proof** (induct set: \text{lists})

**case** Nil
then show \(?case\)
by (meson acc.intros not-listrel1-Nil)

**next**

**case** (Cons \(a\) \(l\))
then show \( \text{?case} \)
  by blast
qed

lemma lists-accI: \( xs \in \text{Wellfounded.} \text{acc} (\text{listrel1} r) \implies xs \in \text{lists (Wellfounded.} \text{acc} r) \)
apply (induct set: Wellfounded.\text{acc})
apply clarify
apply (rule accI)
apply (fastforce dest!: in-set-conv-decomp[THEN iffD1] simp: listrel1-def)
done

lemma wf-listrel1-iff[simp]: \( \text{wf (listrel1} r) = \text{wf r} \)
by (auto simp: wf-acc-iff intro: lists-accD lists-accI[THEN Cons-in-lists-iff[THEN iffD1, THEN conjunct1]])

66.3.5 Lifting Relations to Lists: all elements

inductive-set
  listrel :: (\'a × \'b) set ⇒ (\'a \text{ list} × \'b \text{ list}) set
for r :: (\'a × \'b) set
where
  Nil: \( ([],[]) \in \text{listrel} r \)
| Cons: \( [(x,y)] \in r; (xs,ys) \in \text{listrel} r \implies (x\#xs, y\#ys) \in \text{listrel} r \)

inductive-cases listrel-Nil1 [elim!]: \( ([],xs) \in \text{listrel} r \)
inductive-cases listrel-Nil2 [elim!]: \( (xs,[]) \in \text{listrel} r \)
inductive-cases listrel-Cons1 [elim!]: \( (y\#ys,xs) \in \text{listrel} r \)
inductive-cases listrel-Cons2 [elim!]: \( (xs,y\#ys) \in \text{listrel} r \)

lemma listrel-eq-len: \( (xs, ys) \in \text{listrel} r \implies \text{length} xs = \text{length} ys \)
by(induct rule: listrel.induct) auto

lemma listrel-iff-zip [code-unfold]: \( (xs,ys) \in \text{listrel} r \iff \text{length} xs = \text{length} ys \land (\forall (x,y) \in \text{set(zip} xs ys). (x,y) \in r) \)
proof
  assume \( \text{?L thus ?R by induct (auto intro: listrel-eq-len)} \)
next
  assume \( \text{?R thus ?L} \)
  apply (clarify)
  by (induct rule: list-induct2) (auto intro: listrel.intros)
qed

lemma listrel-iff-nth: \( (xs,ys) \in \text{listrel} r \iff \text{length} xs = \text{length} ys \land (\forall n < \text{length} xs. (xs!n, ys!n) \in r) \)
by (auto simp add: all-set-cone-all-nth listrel-iff-zip)
lemma listrel-mono: $r \subseteq s \implies listrel r \subseteq listrel s$
   by (meson listrel-iff-nth subrelI subset-eq)

lemma listrel-subset: $r \subseteq A \times A \implies listrel r \subseteq lists A \times lists A$
apply clarify
apply (erule listrel.induct, auto)
done

lemma listrel-refl-on: refl-on A r = \implies refl-on (lists A) (listrel r)
apply (simp add: refl-on-def listrel-subset Ball-def)
apply (rule allI)
apply (erule refl-on.induct)
apply (auto)
done

lemma listrel-sym: sym r = \implies sym (listrel r)
by (simp add: listrel-iff-nth sym-def)

lemma listrel-trans: trans r = \implies trans (listrel r)
apply (simp add: trans-def)
apply (intro allI)
apply (rule reflI)
apply (erule listrel.induct)
apply (auto intro: listrel.intros)+
done

theorem equiv-listrel: equiv A r = \implies equiv (lists A) (listrel r)
by (simp add: equiv-def listrel-refl-on listrel-sym listrel-trans)

lemma listrel-rtrancl-refl: (xs, xs) \in listrel(r^*)
using listrel-refl-on[of UNIV, OF refl-rtrancl]
by(auto simp: refl-on-def)

lemma listrel-rtrancl-trans:
\[ [(xs, ys) \in listrel(r^*); (ys, zs) \in listrel(r^*)] \implies (xs, zs) \in listrel(r^*) \]
by (metis listrel-trans trans-def trans-rtrancl)

lemma listrel-nil [simp]: listrel r "[]" = "[]
by (blast intro: listrel.intros)

lemma listrel-Cons:
\[ listrel r "\{x\} \in set-Cons (r^* \{x\}) \] (listrel r "\{xs\})
by (auto simp add: set-Cons-def intro: listrel.intros)

Relating listrel1, listrel and closures:

lemma listrel1-rtrancl-subset-rtrancl-listrel1:
listrel1 \ (r^*) \subseteq (listrel1 \ r)^*

proof (rule subrelI)
  fix \ x s y s assume 1: \ (xs,ys) \in listrel1 \ (r^*)
  \{ \ fix \ x y \ us \ vs \\ n \ have \ (x,y) \in \ r^* \implies (us \ @ \ x \ # \ vs, \ us \ @ \ y \ # \ vs) \in (listrel1 \ r)^* \\ n \ proof (induct rule: rtrancl.induct) \\ n \ case \ rtrancl-refl \ show \ ?case \ by \ simp \\ n \ next \\ n \ case \ rtrancl-into-rtrancl \ thus \ ?case \\ n \ by \ (metis \ listrel1I \ rtrancl \ rtrancl-into-rtrancl) \\ qed \} 
  thus \ (xs,ys) \in (listrel1 \ r)^* \ using \ 1 \ by (blast \ elim: \ listrel1E) 
  qed 

lemma \ rtrancl-listrel1-eq-len: \ (x,y) \in (listrel1 \ r)^* \implies \ length \ x = \ length \ y 
  by (induct rule: rtrancl.induct) (auto intro: listrel1-eq-len)

lemma \ rtrancl-listrel1-ConsI1: 
  \ (xs,ys) \in (listrel1 \ r)^* \implies \ (x \# \ xs, x \# \ ys) \in (listrel1 \ r)^*
  apply (induct rule: rtrancl.induct) 
  apply simp 
  by (metis \ listrel1I2 \ rtrancl.rtrancl-into-rtrancl)

lemma \ rtrancl-listrel1-ConsI2: 
  \ (x,y) \in \ r^* \implies \ (x,y) \in (listrel1 \ r)^*
  \implies \ (x \# \ xs, y \# \ ys) \in (listrel1 \ r)^*
  by ( blast \ intro: \ rtrancl-trans \ rtrancl-listrel1-ConsI1 
    subsetD[OF \ listrel1-rtrancl-subset-rtrancl-listrel1 \ listrel1I1])

lemma \ listrel1-subset-listrel: 
  \ r \subseteq \ r' \implies \ refl \ r' \implies \ listrel1 \ r \subseteq \ listrel(r')
  by (auto \ elim!: \ listrel1E \ simp \ add: \ listrel-iff-zip \ set-zip \ refl-on-def)

lemma \ listrel-reflcl-if-listrel1: 
  \ (xs,ys) \in \ listrel1 \ r \implies \ (xs,ys) \in \ listrel(r^*)
  by (erule \ listrel1E) (auto \ simp \ add: \ listrel-iff-zip \ set-zip)

lemma \ listrel-rtrancl-eq-rtrancl-listrel1: \ listrel \ (r^*) = (listrel1 \ r)^*
proof 
  \{ \ fix \ x y \ assume \ (x,y) \in \ listrel \ (r^*) \\ n \ then \ have \ (x,y) \in \ (listrel1 \ r)^* \\ n \ by \ induct \ (auto \ intro: \ rtrancl-listrel1-ConsI2) \} 
  then \ show \ listrel \ (r^*) \subseteq (listrel1 \ r)^* \\ n \ by \ (rule \ subrelI) 
  next 
  show \ listrel \ (r^*) \supseteq (listrel1 \ r)^* 
proof (rule subrelI)
  fix \ xs \ ys \ assume \ (xs,ys) \in \ (listrel1 \ r)^* 
  then \ show \ (xs,ys) \in \ listrel \ (r^*)
proof induct
  case base show ?case by (auto simp add: listrel-iff-zip set-zip)
next
  case (step ys zs)
  thus ?case by (metis listrel-refl-if-listrel1 listrel-rtrancl-trans)
qed
qed
qed

lemma rtrancl-listrel1-if-listrel:
  \((xs,ys) \in \text{listrel } r \implies (zs,ys) \in (\text{listrel1 } r)^*\)
by (metis listrel-rtrancl-eq-rtrancl-listrel1 subsetD [OF listrel-mono] r-into-rtrancl subsetI)

lemma listrel-subset-rtrancl-listrel1: \(\text{listrel } r \subseteq (\text{listrel1 } r)^*\)
by (fast intro: rtrancl-listrel1-if-listrel)

66.4 Size function

lemma [measure-function]: is-measure \(f\) \implies is-measure (size-list \(f\))
by (rule is-measure-trivial)

lemma [measure-function]: is-measure \(f\) \implies is-measure (size-option \(f\))
by (rule is-measure-trivial)

lemma size-list-estimation[termination-simp]:
  \(x \in \text{set } xs \implies y < f \ x \implies y < \text{size-list } f \ xs\)
by (induct xs) auto

lemma size-list-estimation'[termination-simp]:
  \(x \in \text{set } xs \implies y \leq f \ x \implies y \leq \text{size-list } f \ xs\)
by (induct xs) auto

lemma size-list-map[simp]: size-list \(f\) (map \(g\) \(xs\)) = size-list (\(f \circ g\)) \(xs\)
by (induct \(xs\)) auto

lemma size-list-append[simp]: size-list \(f\) (\(xs \oplus\) \(ys\)) = size-list \(f\) \(xs\) + size-list \(f\) \(ys\)
by (induct \(xs\), auto)

lemma size-list-pointwise[termination-simp]:
  \(\forall x. x \in \text{set } xs \implies f \ x \leq g \ x \implies \text{size-list } f \ xs \leq \text{size-list } g \ xs\)
by (induct \(xs\)) force+

66.5 Monad operation

definition bind :: '\a list ⇒ ('a ⇒ 'b list) ⇒ 'b list where
  bind \(xs\) \(f\) = concat (map \(f\) \(xs\))

hide-const (open) bind
lemma bind-simps [simp]:
List.bind [] f = []
List.bind (x # xs) f = f x @ List.bind xs f
by (simp-all add: bind-def)

lemma list-bind-cong [fundef-cong]:
assumes xs = ys (∀x. x ∈ set xs → f x = g x)
shows List.bind xs f = List.bind ys g
proof –
  from assms(2) have List.bind xs f = List.bind xs g
  by (induction xs) simp-all
  with assms(1) show ?thesis by simp
qed

lemma set-list-bind: set (List.bind xs f) = (∪x∈set xs. set (f x))
by (induction xs) simp-all

66.6 Code generation

Optional tail recursive version of map. Can avoid stack overflow in some
target languages.

fun map-tailrec-rev :: ('a ⇒ 'b) ⇒ 'a list ⇒ 'b list where
map-tailrec-rev f [] bs = bs |
map-tailrec-rev f (a#as) bs = map-tailrec-rev f as (f a # bs)

lemma map-tailrec-rev:
map-tailrec-rev f as bs = rev(map f as) @ bs
by (induction as arbitrary: bs) simp-all

definition map-tailrec :: ('a ⇒ 'b) ⇒ 'a list ⇒ 'b list where
map-tailrec f as = rev (map-tailrec-rev f as [])

Code equation:
lemma map-eq-map-tailrec: map = map-tailrec
by(simp add: fun-eq-iff map-tailrec-def map-tailrec-rev)

66.6.1 Counterparts for set-related operations

definition member :: 'a list ⇒ 'a ⇒ bool where
[code-abbrev]: member xs x ←→ x ∈ set xs

Use member only for generating executable code. Otherwise use x ∈ set xs
instead — it is much easier to reason about.

lemma member-rec [code]:
member (x # xs) y ←→ x = y ∨ member xs y
member [] y ←→ False
by (auto simp add: member-def)
lemma in-set-member :
  \( x \in \text{ set } xs \) \( \iff \) member \( xs \) \( x \)
by (simp add: member-def)

lemmas list-all-iff [code-abbrev] = fun-cong[OF list.pred-set]

definition list-ex :: ('a \Rightarrow bool) \Rightarrow 'a list \Rightarrow bool where
list-ex-iff [code-abbrev]: list-ex \( P \) \( xs \) \( \iff \) Bex (set \( xs \)) \( P \)

definition list-ex1 :: ('a \Rightarrow bool) \Rightarrow 'a list \Rightarrow bool where
list-ex1-iff [code-abbrev]: list-ex1 \( P \) \( xs \) \( \iff \) (\( \exists! \) \( x \). \( x \in \text{ set } xs \) \( \land \) \( P x \))

Usually you should prefer \( \forall x \in \text{ set } xs \), \( \exists x \in \text{ set } xs \) and \( \exists! x. x \in \text{ set } xs \) \( \land \) - over list-all, list-ex and list-ex1 in specifications.

lemma list-all-simps [code]:
  list-all \( P \) (\( x \# \) \( xs \)) \( \iff \) \( P x \) \( \land \) list-all \( P \) \( xs \)
  list-all \( P \) [] \( \iff \) True
by (simp-all add: list-all-iff)

lemma list-ex-simps [simp, code]:
  list-ex \( P \) (\( x \# \) \( xs \)) \( \iff \) \( P x \) \( \lor \) list-ex \( P \) \( xs \)
  list-ex \( P \) [] \( \iff \) False
by (simp-all add: list-ex-iff)

lemma list-ex1-simps [simp, code]:
  list-ex1 \( P \) [] = False
  list-ex1 \( P \) (\( x \# \) \( xs \)) \( \equiv \) (if \( P x \) then list-all (\( \lambda y. \neg \) \( P y \) \( \lor \) \( x = y \)) \( xs \) else list-ex1 \( P \) \( xs \))
by (auto simp add: list-ex1-iff list-all-iff)

lemma Ball-set-list-all:
  Ball (set \( xs \)) \( P \) \( \iff \) list-all \( P \) \( xs \)
by (simp add: list-all-iff)

lemma Bex-set-list-ex:
  Bex (set \( xs \)) \( P \) \( \iff \) list-ex \( P \) \( xs \)
by (simp add: list-ex-iff)

lemma list-all-append [simp]:
  list-all \( P \) (\( xs \) @ \( ys \)) \( \iff \) list-all \( P \) \( xs \) \( \land \) list-all \( P \) \( ys \)
by (auto simp add: list-all-iff)

lemma list-ex-append [simp]:
  list-ex \( P \) (\( xs \) @ \( ys \)) \( \iff \) list-ex \( P \) \( xs \) \( \lor \) list-ex \( P \) \( ys \)
by (auto simp add: list-ex-iff)

lemma list-all-rev [simp]:
  list-all \( P \) (rev \( xs \)) \( \iff \) list-all \( P \) \( xs \)
by (simp add: list-all-iff)
lemma list-ex-rev [simp]:
list-ex P (rev xs) \iff list-ex P xs
by (simp add: list-ex-iff)

lemma list-all-length:
list-all P xs \iff (∀ n < length xs. P (xs ! n))
by (auto simp add: list-all-iff set-conv-nth)

lemma list-ex-length:
list-ex P xs \iff (∃ n < length xs. P (xs ! n))
by (auto simp add: list-ex-iff set-conv-nth)

lemmas list-all-cong [fundef-cong] = list.pred-cong

lemma list-ex-cong [fundef-cong]:
xs = ys \implies (∀ x. x ∈ set ys \implies f x = g x) \implies list-ex f xs = list-ex g ys
by (simp add: list-ex-iff)

definition can-select :: ('a ⇒ bool) ⇒ 'a set ⇒ bool
[code-abbrev]: can-select P A = (∃!x∈A. P x)

lemma can-select-set-list-ex1 [code]:
can-select P (set A) = list-ex1 P A
by (simp add: list-ex1-iff can-select-def)

Executable checks for relations on sets

definition listrel1p :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ bool
[code-unfold]: listrel1p r xs ys = ((xs, ys) ∈ listrel1 {((x, y), r x y)})

lemma [code-unfold]:
(xs, ys) ∈ listrel1 r = listrel1p (λx y. (x, y) ∈ r) xs ys
unfolding listrel1p-def by auto

lemma [code]:
listrel1p r [] xs = False
listrel1p r xs [] = False
listrel1p r (x # xs) (y # ys) \iff
  r x y ∧ xs = ys ∨ x = y ∧ listrel1p r xs ys
by (simp add: listrel1p-def)+

definition lexordp :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ bool
[code-unfold]: lexordp r xs ys = ((xs, ys) ∈ lexord {((x, y), r x y)})

lemma [code-unfold]:
(xs, ys) ∈ lexord r = lexordp (λx y. (x, y) ∈ r) xs ys
unfolding lexordp-def by auto
THEORY “List”

lemma [code]:
  lexordp r xs [] = False
  lexordp r [] (y # ys) = True
  lexordp r (x # xs) (y # ys) = (r x y ∨ (x = y ∧ lexordp r xs ys))
unfolding lexordp-def by auto

Bounded quantification and summation over nats.

lemma atMost-upto [code-unfold]:
  {..n} = set [0..<Suc n]
  by auto

lemma atLeast-upt [code-unfold]:
  {..<n} = set [0..<n]
  by auto

lemma greaterThanLessThan-upt [code-unfold]:
  {n..<m} = set [Suc n..<m]
  by auto

lemmas atLeastLessThan-upt [code-unfold] = set-upt [symmetric]

lemma greaterThanAtMost-upt [code-unfold]:
  {n..<m} = set [Suc n..<Suc m]
  by auto

lemma atLeastAtMost-upt [code-unfold]:
  {n..m} = set [n..<Suc m]
  by auto

lemma all-nat-less-eq [code-unfold]:
  (∀ m<n::nat. P m) ⟷ (∀ m ∈ {0..<n}. P m)
  by auto

lemma ex-nat-less-eq [code-unfold]:
  (∃ m<n::nat. P m) ⟷ (∃ m ∈ {0..<n}. P m)
  by auto

lemma all-nat-less [code-unfold]:
  (∀ m≤n::nat. P m) ⟷ (∀ m ∈ {0..n}. P m)
  by auto

lemma ex-nat-less [code-unfold]:
  (∃ m≤n::nat. P m) ⟷ (∃ m ∈ {0..n}. P m)
  by auto

Bounded LEAST operator:

definition Bleast S P = (LEAST x. x ∈ S ∧ P x)

definition abort-Bleast S P = (LEAST x. x ∈ S ∧ P x)
declare [[code abort: abort-Bleast]]

lemma Bleast-code [code]:
Bleast (set xs) P = (case filter P (sort xs) of
  x#xs ⇒ x |
  [] ⇒ abort-Bleast (set xs) P)

proof (cases filter P (sort xs))
case Nil thus thesis by (simp add: Bleast-def abort-Bleast-def)
  
next
case (Cons x ys)
  have (LEAST x. x ∈ set xs ∧ P x) = x

proof (rule Least-equality)
  show x ∈ set xs ∧ P x
    by (metis Cons Cons-eq-filter-iff in-set-conv-decomp set-sort)

next
given y assume y ∈ set xs ∧ P y
  hence y ∈ set (filter P xs) by auto
  thus x ≤ y
    by (metis Cons eq-iff filter-sort set-ConsD set-sort sorted simps(2) sorted-sort)

qed

thus thesis using Cons by (simp add: Bleast-def)

qed

declare Bleast-def[symmetric, code-unfold]

Summation over ints.

lemma greaterThanLessThan-upto [code-unfold]:
{ i..<j::int } = set [i+1..j - 1]
by auto

lemma atLeastLessThan-upto [code-unfold]:
{ i..<j::int } = set [i..j - 1]
by auto

lemma greaterThanAtMost-upto [code-unfold]:
{ i..<j::int } = set [i+1..j]
by auto

lemmas atLeastAtMost-upto [code-unfold] = set-upto [symmetric]

66.6.2 Optimizing by rewriting

definition null :: 'a list ⇒ bool where
  [code-abbrev]: null xs ←→ xs = []

Efficient emptyness check is implemented by null.

lemma null-rec [code]:
null (x # xs) ←→ False
null [] ←→ True
by (simp-all add: null-def)

lemma eq-Nil-null:
x$s = [] ←→ null x$s
by (simp add: null-def)

lemma equal-Nil-null [code-unfold]:
HOL.equal x$s [] ←→ null x$s
HOL.equal [] = null
by (auto simp add: equal null-def)

definition maps :: ('a ⇒ 'b list) ⇒ 'a list ⇒ 'b list where
[code-abbrev]: maps f x$s = concat (map f x$s)

definition map-filter :: ('a ⇒ 'b option) ⇒ 'a list ⇒ 'b list where
[code-post]: map-filter f x$s = map (the ◦ f) (filter (λx. f x ≠ None) x$s)

Operations maps and map-filter avoid intermediate lists on execution – do not use for proving.

lemma maps-simps [code]:
maps f (x # x$s) = f x @ maps f x$s
maps f [] = []
by (simp-all add: maps-def)

lemma map-filter-simps [code]:
map-filter f (x # x$s) = (case f x of None ⇒ map-filter f x$s | Some y ⇒ y # map-filter f x$s)
map-filter f [] = []
by (simp-all add: map-filter-def split: option.split)

lemma concat-map-maps:
concat (map f x$s) = maps f x$s
by (simp add: maps-def)

lemma map-filter-map-filter [code-unfold]:
map f (filter P x$s) = map-filter (λx. if P x then Some (f x) else None) x$s
by (simp add: map-filter-def)

Optimized code for ∀ i∈{a..b::int} and ∀ n:{a..<b::nat} and similiarly for ∃.

definition all-interval-nat :: (nat ⇒ bool) ⇒ nat ⇒ nat ⇒ bool where
all-interval-nat P i j ←→ (∀ n ∈ {i..<j}. P n)

lemma [code]:
all-interval-nat P i j ←→ i ≥ j ∨ P i ∧ all-interval-nat P (Suc i) j

proof −
have *: ∀ n. P i −> ∀ n∈{Suc i..<j}. P n −> i ≤ n −> n < j −> P n
proof −
THEORY “List”

fix n
assume P i \forall n \in \{\text{Suc } i..< j\}. P n i \leq n n < j
then show P n by (cases n = i) simp-all
qed
show ?thesis by (auto simp add: all-interval-nat-def intro: *)
qed

lemma list-all-iff-all-interval-nat [code-unfold]:
list-all P [i..<j] \iff all-interval-nat P i j
by (simp add: list-all-iff all-interval-nat-def)

lemma list-ex-iff-not-all-interval-nat [code-unfold]:
list-ex P [i..<j] \iff \neg (all-interval-nat (Not o P) i j)
by (simp add: list-ex-iff all-interval-nat-def)

definition all-interval-int :: (int \Rightarrow bool) \Rightarrow int \Rightarrow int \Rightarrow bool where
all-interval-int P i j \iff (\forall k \in \{i..j\}. P k)

lemma [code]:
all-interval-int P i j \iff i > j \lor P i \land all-interval-int P (i + 1) j
proof -
have *: \\land k. P i \Longrightarrow \forall k \in \{i+1..j\}. P k \Longrightarrow i \leq k \Longrightarrow k \leq j \Longrightarrow P k
  proof -
  fix k
  assume P i \\forall k \in \{i+1..j\}. P k i \leq k k \leq j
  then show P k by (cases k = i) simp-all
  qed
show ?thesis by (auto simp add: all-interval-int-def intro: *)
qed

lemma list-all-iff-all-interval-int [code-unfold]:
list-all P [i..j] \iff all-interval-int P i j
by (simp add: list-all-iff all-interval-int-def)

lemma list-ex-iff-not-all-interval-int [code-unfold]:
list-ex P [i..j] \iff \neg (all-interval-int (Not o P) i j)
by (simp add: list-ex-iff all-interval-int-def)

optimized code (tail-recursive) for length

definition gen-length :: (int \Rightarrow bool) \Rightarrow int \Rightarrow int \Rightarrow bool where
gen-length n xs = n + length xs

lemma gen-length-code [code]:
gen-length n [] = n
  gen-length n (x # xs) = gen-length (Suc n) xs
by(simp-all add: gen-length-def)

declare list.size(3-4)[code del]
lemma length-code [code]: length = gen-length 0
by(simp add: gen-length-def fun-eq-iff)

hide-const (open) member null maps map-filter all-interval-nat all-interval-int

66.6.3 Pretty lists

ML ⟨
(* Code generation for list literals. *)
signature LIST-CODE =
sig
  val add-literal-list: string -> theory -> theory
end;
structure List-Code : LIST-CODE =
struct
open Basic-Code-Thingol;

fun implode-list t =
  let
    fun dest-cons (IConst { sym = Code-Symbol.Constant const-name (Cons), ... }) 't1 't2 = SOME (t1, t2)
    | dest-cons - = NONE;
    val (ts, t') = Code-Thingol.unfoldr dest-cons t;
    in case t'
      of IConst { sym = Code-Symbol.Constant const-name (Nil), ... } => SOME ts
        | - => NONE
  end;

fun print-list (target-fxy, target-cons) pr fxy t1 t2 =
  Code-Printer.brackify-infix (target-fxy, Code-Printer.R) fxy (pr (Code-Printer.INFX (target-fxy, Code-Printer.X)) t1,
  Code-Printer.str target-cons,
  pr (Code-Printer.INFX (target-fxy, Code-Printer.R)) t2);

fun add-literal-list target =
  let
    fun pretty literals pr - vars fxy [(t1, -), (t2, -)] =
      case Option.map (cons t1) (implode-list t2)
        of SOME ts =>
          Code-Printer.literal-list literals (map (pr vars Code-Printer.NOBR) ts)
        | NONE =>
          print-list (Code-Printer.infix-cons literals) (pr vars) fxy t1 t2;
in
THEORY "List"

Code-Target.set-printings (Code-Symbol.Constant (const-name 'Cons),
  [[(target, SOME (Code-Printer.complex-const-syntax (2, pretty))))]])
end

end;
)

code-printing
type-constructor list →
  (SML) - list
  and (OCaml) - list
  and (Haskell) !(\(-\))
  and (Scala) List\!(\(-\))]
| constant Nil →
  (SML) []
  and (OCaml) []
  and (Haskell) []
  and (Scala) !Nil
| class-instance list :: equal →
  (Haskell) –
| constant HOL.equal :: 'a list ⇒ 'a list ⇒ bool →
  (Haskell) infix 4 ==

setup :fold (List-Code.add-literal-list) [SML, OCaml, Haskell, Scala]

code-reserved SML
list

code-reserved OCaml
list

66.6.4 Use convenient predefined operations

code-printing
constant (@) →
  (SML) infixr 7 @
  and (OCaml) infixr 6 @
  and (Haskell) infixr 5 ++
  and (Scala) infixl 7 ++
| constant map →
  (Haskell) map
| constant filter →
  (Haskell) filter
| constant concat →
  (Haskell) concat
| constant List.maps →
  (Haskell) concatMap
| constant rev →
  (Haskell) reverse
THEORY "List"

| constant zip ⇀ (Haskell) zip
| constant List.null ⇀ (Haskell) null
| constant takeWhile ⇀ (Haskell) takeWhile
| constant dropWhile ⇀ (Haskell) dropWhile
| constant list-all ⇀ (Haskell) all
| constant list-ex ⇀ (Haskell) any

66.6.5 Implementation of sets by lists

lemma is-empty-set [code]:
Set.is-empty (set xs) ←→ List.null xs
by (simp add: Set.is-empty-def null-def)

lemma empty-set [code]:
{} = set []
by simp

lemma UNIV-coset [code]:
UNIV = List.coset []
by simp

lemma compl-set [code]:
− set xs = List.coset xs
by simp

lemma compl-coset [code]:
− List.coset xs = set xs
by simp

lemma [code]:
x ∈ set xs ←→ List.member xs x
x ∈ List.coset xs ←→ ¬ List.member xs x
by (simp-all add: member-def)

lemma insert-code [code]:
insert x (set xs) = set (List.insert x xs)
insert x (List.coset xs) = List.coset (removeAll x xs)
by simp-all

lemma remove-code [code]:
Set.remove x (set xs) = set (removeAll x xs)
Set.remove x (List.coset xs) = List.coset (List.insert x xs)
by (simp-all add: remove-def Compl-insert)
lemma filter-set [code]:
  $\text{Set.filter } P \ (\text{set } xs) = \text{set } (\text{filter } P \ xs)$
  by auto

lemma image-set [code]:
  $\text{image } f \ (\text{set } xs) = \text{set } (\text{map } f \ xs)$
  by simp

lemma subset-code [code]:
  set $xs \leq B \iff (\forall x \in \text{set } xs. \ x \in B)$
  $A \leq \text{List.coset } ys \iff (\forall y \in \text{set } ys. \ y \notin A)$
  $\text{List.coset } [] \subseteq \text{set } [] \iff \text{False}$
  by auto

A frequent case – avoid intermediate sets

lemma [code-unfold]:
  set $xs \subseteq \text{set } ys \iff \text{list-all } (\lambda x. \ x \in \text{set } ys) \ xs$
  by (auto simp: list-all-iff)

lemma Ball-set [code]:
  $\text{Ball } (\text{set } xs) \ P \iff \text{list-all } P \ xs$
  by (simp add: list-all-iff)

lemma Bex-set [code]:
  $\text{Bex } (\text{set } xs) \ P \iff \text{list-ex } P \ xs$
  by (simp add: list-ex-iff)

lemma card-set [code]:
  $\text{card } (\text{set } xs) = \text{length } (\text{remdups } xs)$
proof –
  have $\text{card } (\text{set } (\text{remdups } xs)) = \text{length } (\text{remdups } xs)$
    by (rule distinct-card) simp
  then show thesis by simp
qed

lemma the-elem-set [code]:
  $\text{the-elem } (\text{set } [x]) = x$
  by simp

lemma Pow-set [code]:
  $\text{Pow } (\text{set } []) = \{\{}\}$
  $\text{Pow } (\text{set } (x \# xs)) = (\text{let } A = \text{Pow } (\text{set } xs) \text{ in } A \cup \text{insert } x \ A)$
  by (simp-all add: Pow-insert Let-def)

definition map-project :: (‘a ⇒ ‘b option) ⇒ ‘a set ⇒ ‘b set where
  map-project $f A = \{b. \ \exists \ a \in A. \ f \ a = \text{Some } b\}$

lemma [code]:
map-project f (set xs) = set (List.map-filter f xs)
by (auto simp add: map-project-def map-filter-def image-def)

hide-const (open) map-project

Operations on relations

lemma product-code [code]:
Product-Type.product (set xs) (set ys) = set [(x, y). x ← xs, y ← ys]
by (auto simp add: Product-Type.product-def)

lemma Id-on-set [code]:
Id-on (set xs) = set [(x, x). x ← xs]
by (auto simp add: Id-on-def)

lemma [code]:
R "" S = List.map-project (λ(x, y). if x ∈ S then Some y else None) R
unfolding map-project-def by (auto split: prod.split if-split-asm)

lemma trancl-set-ntrancl [code]:
trancl (set xs) = ntrancl (card (set xs) - 1) (set xs)
by (simp add: finite-trancl-ntranl)

lemma set-relcomp [code]:
set xys O set yzs = set [(fst xy, snd yz). xy ← xys, yz ← yzs, snd xy = fst yz]
by (auto simp add: Bex-def image-def)

lemma wf-set [code]:
wf (set xs) = acyclic (set xs)
by (simp add: wf-iff-acyclic-if-finite)

66.7 Setup for Lifting/Transfer

66.7.1 Transfer rules for the Transfer package

category includes lifting-syntax
begin

lemma tl-transfer [transfer-rule]:
(list-all2 A ===> list-all2 A) tl tl
unfolding tl-def[abs-def] by transfer-prover

lemma butlast-transfer [transfer-rule]:
(list-all2 A ===> list-all2 A) butlast butlast
by (rule rel-funI, erule list-all2-induct, auto)

lemma map-rec: map f xs = rec-list Nil (%x - y. Cons (f x) y) xs
by (induct xs) auto

lemma append-transfer [transfer-rule]:
(list-all2 A ===> list-all2 A ===> list-all2 A) append append
unfolding List.append-def by transfer-prover

lemma rev-transfer [transfer-rule]:
  (list-all2 A ===> list-all2 A) rev rev
unfolding List.rev-def by transfer-prover

lemma filter-transfer [transfer-rule]:
  ((A ===> (=)) ===> list-all2 A ===> list-all2 A) filter filter
unfolding List.filter-def by transfer-prover

lemma fold-transfer [transfer-rule]:
  ((A ===> B ===> B) ===> list-all2 A ===> B ===> B) fold fold
unfolding List.fold-def by transfer-prover

lemma foldr-transfer [transfer-rule]:
  ((A ===> B ===> B) ===> list-all2 A ===> B ===> B) foldr foldr
unfolding List.foldr-def by transfer-prover

lemma foldl-transfer [transfer-rule]:
  ((B ===> A ===> B) ===> list-all2 A ===> B ===> B) foldl foldl
unfolding List.foldl-def by transfer-prover

lemma concat-transfer [transfer-rule]:
  (list-all2 (list-all2 A) ===> list-all2 A) concat concat
unfolding List.concat-def by transfer-prover

lemma drop-transfer [transfer-rule]:
  ((=) ===> list-all2 A ===> list-all2 A) drop drop
unfolding List.drop-def by transfer-prover

lemma take-transfer [transfer-rule]:
  ((=) ===> list-all2 A ===> list-all2 A) take take
unfolding List.take-def by transfer-prover

lemma list-update-transfer [transfer-rule]:
  (list-all2 A ===> (=) ===> A ===> list-all2 A) list-update list-update
unfolding list-update-def by transfer-prover

lemma takeWhile-transfer [transfer-rule]:
  ((A ===> (=)) ===> list-all2 A ===> list-all2 A) takeWhile takeWhile
unfolding takeWhile-def by transfer-prover

lemma dropWhile-transfer [transfer-rule]:
  ((A ===> (=)) ===> list-all2 A ===> list-all2 A) dropWhile dropWhile
unfolding dropWhile-def by transfer-prover

lemma zip-transfer [transfer-rule]:
  (list-all2 A ===> list-all2 B ===> list-all2 (rel-prod A B)) zip zip
unfolding zip-def by transfer-prover
lemma product-transfer [transfer-rule]:
(list-all2 A ===> list-all2 B ===> list-all2 (rel-prod A B)) List.product List.product
unfolding List.product-def by transfer-prover

lemma product-lists-transfer [transfer-rule]:
(list-all2 (list-all2 A) ===> list-all2 (list-all2 A)) product-lists product-lists
unfolding product-lists-def by transfer-prover

lemma insert-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A
shows (A ===> list-all2 A ===> list-all2 A) List.insert List.insert
unfolding List.insert-def [abs-def] by transfer-prover

lemma find-transfer [transfer-rule]:
((A ===> (=)) ===> list-all2 A ===> rel-option A) List.find List.find
unfolding List.find-def by transfer-prover

lemma those-transfer [transfer-rule]:
(list-all2 (rel-option P) ===> rel-option (list-all2 P)) those those
unfolding List.those-def by transfer-prover

lemma remove1-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A
shows (A ===> list-all2 A ===> list-all2 A) remove1 remove1
unfolding remove1-def by transfer-prover

lemma removeAll-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A
shows (A ===> list-all2 A ===> list-all2 A) removeAll removeAll
unfolding removeAll-def by transfer-prover

lemma distinct-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A
shows (list-all2 A ===> (=)) distinct distinct
unfolding distinct-def by transfer-prover

lemma remdups-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A
shows (list-all2 A ===> list-all2 A) remdups remdups
unfolding remdups-def by transfer-prover

lemma remdups-adj-transfer [transfer-rule]:
assumes [transfer-rule]: bi-unique A
shows (list-all2 A ===> list-all2 A) remdups-adj remdups-adj
proof (rule rel-funI, erule list-all2-induct)
qed (auto simp: remdups-adj-Cons assms[unfolded bi-unique-def] split: list.splits)

lemma replicate-transfer [transfer-rule]:
\[(=) \Longrightarrow A \Longrightarrow \text{list-all2 } A\] replicate replicate

unfolding replicate-def by transfer-prover

lemma length-transfer [transfer-rule]:
\[(\text{list-all2 } A \Longrightarrow (=)) \text{ length length}\]

unfolding size-list-overloaded-def size-list-def by transfer-prover

lemma rotate1-transfer [transfer-rule]:
\[(\text{list-all2 } A \Longrightarrow \text{list-all2 } A) \text{ rotate1 rotate1}\]

unfolding rotate1-def by transfer-prover

lemma rotate-transfer [transfer-rule]:
\[(=) \Longrightarrow \text{list-all2 } A \Longrightarrow \text{list-all2 } A) \text{ rotate rotate}\]

unfolding rotate-def [abs-def] by transfer-prover

lemma nths-transfer [transfer-rule]:
\[(\text{list-all2 } A \Longrightarrow \text{rel-set } (=) \Longrightarrow \text{list-all2 } A) \text{ nths nths}\]

unfolding nths-def [abs-def] by transfer-prover

lemma subseqs-transfer [transfer-rule]:
\[(\text{list-all2 } A \Longrightarrow \text{list-all2 (list-all2 } A)) \text{ subseqs subseqs}\]

unfolding subseqs-def [abs-def] by transfer-prover

lemma partition-transfer [transfer-rule]:
\[(\text{partition} A \Longrightarrow (=)) \Longrightarrow \text{list-all2 } A \Longrightarrow \text{rel-prod (list-all2 } A) (\text{list-all2 } A))\]

partition partition

unfolding partition-def by transfer-prover

lemma lists-transfer [transfer-rule]:
\[(\text{rel-set A \Longrightarrow rel-set (list-all2 } A)) \text{ lists lists}\]

apply (rule rel-funI, rule rel-setI)

apply (erule lists.induct, simp)

apply (simp only: rel-set-def list-all2-Cons1, metis lists.Cons)

apply (erule lists.induct, simp)

apply (simp only: rel-set-def list-all2-Cons2, metis lists.Cons)

done

lemma set-Cons-transfer [transfer-rule]:
\[(\text{rel-set A \Longrightarrow rel-set (list-all2 } A) \Longrightarrow \text{rel-set (list-all2 } A))\]

set-Cons set-Cons

unfolding rel-fun-def rel-set-def set-Cons-def

by (fastforce simp add: list-all2-Cons1 list-all2-Cons2)

lemma listset-transfer [transfer-rule]:
\[(\text{list-all2 (rel-set A) \Longrightarrow rel-set (list-all2 } A)) \text{ listset listset}\]

unfolding listset-def by transfer-prover

lemma null-transfer [transfer-rule]:
\[(\text{list-all2 A \Longrightarrow ( =)) List.null List.null}\]
unfolding rel-fun-def List.null-def by auto

lemma list-all-transfer [transfer-rule]:
  \[ \text{((A \Longrightarrow (\_)) \Longrightarrow list-all2 A \Longrightarrow (\_)) list-all list-all} \]
unfolding list-all-iff [abs-def] by transfer-prover

lemma list-ex-transfer [transfer-rule]:
  \[ \text{((A \Longrightarrow (\_)) \Longrightarrow list-all2 A \Longrightarrow (\_)) list-ex list-ex} \]
unfolding list-ex-iff [abs-def] by transfer-prover

lemma splice-transfer [transfer-rule]:
  \[ (\text{list-all2 A \Longrightarrow list-all2 A \Longrightarrow list-all2 A}) \text{ splice splice} \]
apply (rule rel-funI, erule list-all2-induct, simp add: rel-fun-def, simp)
apply (rule rel-funI)
apply (erule-tac xs ys xs ys)
done

lemma shuffles-transfer [transfer-rule]:
  \[ (\text{list-all2 A \Longrightarrow list-all2 A \Longrightarrow rel-set (list-all2 A)}) \text{ shuffles shuffles} \]
proof (intro rel-funI, goal-cases)
  case (1 xs xs' ys ys')
  thus ?case
proof (induction xs ys arbitrary: xs' ys' rule: shuffles.induct)
  case (3 x xs y ys x's y's')
from 3.prems obtain x' xs'' where xs': xs' = x' # xs'' by (cases xs') auto
from 3.prems obtain y' ys'' where ys': ys' = y' # ys'' by (cases ys') auto
have [transfer-rule]: A x x' A y y' list-all2 A xs xs'' list-all2 A ys ys''
using 3.prems by (simp-all add: x's y's')
have [transfer-rule]: rel-set (list-all2 A) (shuffles xs (y # ys)) (shuffles xs'' ys'')
and
[transfer-rule]: rel-set (list-all2 A) (shuffles (x # xs) ys) (shuffles xs' y's'')
using 3.prems by (auto intro: 3.IH simp: x's y's')
have rel-set (list-all2 A) ((#) x' shuffles xs (y # ys) \cup (#) y' shuffles (x # xs) ys)
((#) x' \cdot shuffles xs'' ys' \cup (#) y' \cdot shuffles xs' y's') by transfer-prover
thus ?case by (simp add: x's y's')
qed (auto simp: rel-set-def)
qed

lemma rtrancl-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-unique A bi-total A
  shows (rel-set (rel-prod A A) \Longrightarrow rel-set (rel-prod A A)) rtrancl rtrancl
unfolding rtrancl-def by transfer-prover

lemma monotone-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-total A
  shows ((A \Longrightarrow A) \Longrightarrow (\_)) \Longrightarrow (B \Longrightarrow B) \Longrightarrow (\_)) \Longrightarrow (A \Longrightarrow B) \Longrightarrow (\_)) monotone monotone
unfolding monotone-def[abs-def] by transfer-prover
lemma fun-ord-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-total C
  shows \((A \Rightarrow B \Rightarrow (\approx)) \Rightarrow (C \Rightarrow A) \Rightarrow (C \Rightarrow B)\) 
  \(\Rightarrow (\approx)\) fun-ord fun-ord
unfolding fun-ord-def[abs-def] by transfer-prover

lemma fun-lub-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-total A bi-unique A
  shows \((\rel-set A \Rightarrow B) \Rightarrow \rel-set (C \Rightarrow A) \Rightarrow C \Rightarrow B)\) 
  fun-lub fun-lub
unfolding fun-lub-def[abs-def] by transfer-prover
end
end

67 Sum and product over lists

theory Groups-List
imports List
begin
locale monoid-list = monoid
begin

definition F :: 'a list ⇒ 'a
where
  eq-foldr [code]: F xs = foldr f xs 1
lemma Nil [simp]:
  F [] = 1
  by (simp add: eq-foldr)
lemma Cons [simp]:
  F (x # xs) = x * F xs
  by (simp add: eq-foldr)
lemma append [simp]:
  F (xs @ ys) = F xs * F ys
  by (induct xs) (simp-all add: assoc)
end
locale comm-monoid-list = comm-monoid + monoid-list
begin
lemma rev [simp]:
  F (rev xs) = F xs
by (simp add: eq-foldr foldr-fold fold-rev fun-eq-iff assoc left-commute)

end

locale comm-monoid-list-set = list: comm-monoid-list + set: comm-monoid-set

begin

lemma distinct-set-conv-list:
  distinct xs =⇒ set.F g (set xs) = list.F (map g xs)
  by (induct xs) simp-all

lemma set-conv-list [code]:
  set.F g (set xs) = list.F (map g (remdups xs))
  by (simp add: distinct-set-conv-list [symmetric])

end

67.1 List summation

context monoid-add

begin

sublocale sum-list: monoid-list plus 0

defines
  sum-list = sum-list.F ..

end

context comm-monoid-add

begin

sublocale sum-list: comm-monoid-list plus 0
rewrites
  monoid-list.F plus 0 = sum-list
proof --
  show comm-monoid-list plus 0 ..
  then interpret sum-list: comm-monoid-list plus 0 .
  from sum-list-def show monoid-list.F plus 0 = sum-list by simp
  qed

sublocale sum: comm-monoid-list-set plus 0
rewrites
  monoid-list.F plus 0 = sum-list
  and comm-monoid-set.F plus 0 = sum
proof --
  show comm-monoid-list-set plus 0 ..
  then interpret sum: comm-monoid-list-set plus 0 .
  from sum-list-def show monoid-list.F plus 0 = sum-list by simp
  from sum-def show comm-monoid-set.F plus 0 = sum by (auto intro: sym)
Some syntactic sugar for summing a function over a list:

\[
\sum x \leftarrow xs. \ b \rightarrow \ \mbox{CONST sum-list} \ (\mbox{CONST map} \ (\lambda x. \ b) \ xs)
\]

context
includes lifting-syntax

begin

lemma sum-list-transfer [transfer-rule]:
(list-all2 A \Longrightarrow A) sum-list sum-list
  if [transfer-rule]: A 0 0 (A \Longrightarrow A \Longrightarrow A) (+) (+)
unfolding sum-list.eq-foldr [abs-def]
by transfer-prover

end

TODO duplicates

lemmas sum-list-simps = sum-list.Nil sum-list.Cons
lemmas sum-list-append = sum-list.append
lemmas sum-list-rev = sum-list.rev

lemma (in monoid-add) fold-plus-sum-list-rev:
fold plus xs \ = \ plus \ (sum-list \ (rev \ xs))
proof
  fix x
  have fold plus xs x \ = \ sum-list \ (rev \ xs \ @ \ [x])
    by (simp add: foldr-conv-fold sum-list.eq-foldr)
  also have \ldots \ = \ sum-list \ (rev \ xs) + x
    by simp
  finally show fold plus xs x \ = \ sum-list \ (rev \ xs) + x
  .
qed

lemma (in comm-monoid-add) sum-list-map-remove1:
x \in set xs \Longrightarrow sum-list \ (map \ f \ xs) \ = \ f \ x + sum-list \ (map \ f \ (remove1 \ x \ xs))
by (induct xs) (auto simp add: ac-simps)

lemma (in monoid-add) size-list-conv-sum-list:
size-list \ f \ xs \ = \ sum-list \ (map \ f \ xs) + size \ xs
by (induct xs) auto
lemma (in monoid-add) length-concat:
  length (concat xss) = sum-list (map length xss)
  by (induct xss) simp-all

lemma (in monoid-add) length-product-lists:
  length (product-lists xss) = foldr (∗) (map length xss) 1
proof (induct xss)
  case (Cons xs xss) then show ?case by (induct xs) (auto simp: length-concat o-def)
qed simp

lemma (in monoid-add) sum-list-map-filter:
  assumes \( \forall x. x \in \text{set } xs \Rightarrow \neg P x \Rightarrow f x = 0 \)
  shows sum-list (map f (filter P xs)) = sum-list (map f xs)
  using assms by (induct xs) auto

lemma sum-list-filter-te-nat:
  fixes f :: 'a ⇒ nat
  shows sum-list (map f (filter P xs)) ≤ sum-list (map f xs)
  by (induction xs; simp)

lemma (in comm-monoid-add) distinct-sum-list-conv-Sum:
  distinct xs ⇒ sum-list xs = Sum (set xs)
  by (induct xs) simp-all

lemma sum-list-upt[simp]:
  \( m \leq n \Rightarrow \text{sum-list } [m..<n] = \sum \{m..<n\} \)
  by (simp add: distinct-sum-list-conv-Sum)

context ordered-comm-monoid-add
begin

lemma sum-list-nonneg: (\( \forall x. x \in \text{set } xs \Rightarrow 0 \leq x \) \( \Rightarrow 0 \leq \text{sum-list } xs \))
  by (induction xs) auto

lemma sum-list-nonpos: (\( \forall x. x \in \text{set } xs \Rightarrow x \leq 0 \) \( \Rightarrow \text{sum-list } xs \leq 0 \))
  by (induction xs) (auto simp: add-nonpos-nonpos)

lemma sum-list-nonneg-eq-0-iff:
  (\( \forall x. x \in \text{set } xs \Rightarrow 0 \leq x \) \( \Rightarrow \text{sum-list } xs = 0 \) \( \leftrightarrow (\forall x \in \text{set } xs. x = 0) \))
  by (induction xs) (simp-all add: add-nonneg-eq-0-iff sum-list-nonneg)
end

context canonically-ordered-monoid-add
begin

lemma sum-list-eq-0-iff [simp]:
  sum-list ns = 0 \( \leftrightarrow (\forall n \in \text{set } ns. n = 0) \)
lemma member-le-sum-list:
x \in \text{set} \; \text{xs} \implies x \leq \text{sum-list} \; \text{xs}
by (induction \; \text{xs}) \; (auto \; simp: \; \text{add-increasing} \; \text{add-increasing2})

lemma elem-le-sum-list:
k < \text{size} \; \text{ns} \implies \text{ns}!k \leq \text{sum-list} \; (\text{ns})
by (rule \; \text{member-le-sum-list}) \; \text{simp}

end

lemma \text{(in \; ordered-cancel-comm-monoid-diff \; \text{sum-list-update})}:
k < \text{size} \; \text{xs} \implies \text{sum-list} \; (\text{xs}[k := x]) = \text{sum-list} \; \text{xs} + x - \text{xs}!k
apply (induction \; \text{xs} \; \text{arbitrary}:k)
apply (auto \; simp: \text{add-ac} \; \text{split:} \; \text{nat-split})
apply (drule \; \text{elem-le-sum-list})
by (simp \; add: \text{local.add-diff-assoc} \; \text{local.add-increasing})

lemma \text{(in \; monoid-add \; \text{sum-list-triv})}:
(\sum x \leftarrow \text{xs}. \; r) = \text{of-nat} \; (\text{length} \; \text{xs}) \times r
by (induct \; \text{xs}) \; (simp-all \; add: \text{distrib-right})

lemma \text{(in \; monoid-add \; \text{sum-list-0} \; \text{[simp]})}:
(\sum x \leftarrow \text{xs}. \; 0) = 0
by (induct \; \text{xs}) \; (simp-all \; add: \text{distrib-right})

For non-Abelian groups \text{xs} needs to be reversed on one side:

lemma \text{(in \; ab-group-add \; \text{uminus-sum-list-map})}:
- \text{sum-list} \; (\text{map} \; f \; \text{xs}) = \text{sum-list} \; (\text{map} \; (\text{uminus} \circ f) \; \text{xs})
by (induct \; \text{xs}) \; \text{simp-all}

lemma \text{(in \; comm-monoid-add \; \text{sum-list-addf})}:
(\sum x \leftarrow \text{xs}. \; f \; x + g \; x) = \text{sum-list} \; (\text{map} \; f \; \text{xs}) + \text{sum-list} \; (\text{map} \; g \; \text{xs})
by (induct \; \text{xs}) \; (simp-all \; \text{add: algebra-simps})

lemma \text{(in \; ab-group-add \; \text{sum-list-subtractf})}:
(\sum x \leftarrow \text{xs}. \; f \; x - g \; x) = \text{sum-list} \; (\text{map} \; f \; \text{xs}) - \text{sum-list} \; (\text{map} \; g \; \text{xs})
by (induct \; \text{xs}) \; (simp-all \; \text{add: algebra-simps})

lemma \text{(in \; semiring-0 \; \text{sum-list-const-mult})}:
(\sum x \leftarrow \text{xs}. \; c \times f \; x) = c \times (\sum x \leftarrow \text{xs}. \; f \; x)
by (induct \; \text{xs}) \; (simp-all \; \text{add: algebra-simps})

lemma \text{(in \; semiring-0 \; \text{sum-list-mult-const})}:
(\sum x \leftarrow \text{xs}. \; f \; x \times c) = (\sum x \leftarrow \text{xs}. \; f \; x) \times c
by (induct \; \text{xs}) \; (simp-all \; \text{add: algebra-simps})

lemma \text{(in \; ordered-ab-group-add-abs \; \text{sum-list-abs})}:
\[ \text{sum-list } xs \leq \text{sum-list } (\text{map } \text{abs} \; xs) \]

by (induct \( xs \)) (simp-all add: order-trans \([\text{OF abs-triangle-ineq}]\))

**lemma sum-list-mono:**

fixes \( f \) \( g \) :: \( 'a \rightarrow 'b \):

\{\text{monoid-add, ordered-ab-semigroup-add}\}

shows \( \forall x. x \in \text{set } xs \Rightarrow f x \leq g x \Rightarrow (\sum x \leftarrow xs. f x) \leq (\sum x \leftarrow xs. g x) \)

by (induct \( xs \)) (simp, simp add: add-mono)

**lemma sum-list-strict-mono:**

fixes \( f \) \( g \) :: \( 'a \rightarrow 'b \):

\{\text{monoid-add, strict-ordered-ab-semigroup-add}\}

shows \( \text{if } xs \neq [] \text{ then } \forall x. x \in \text{set } xs \Rightarrow f x < g x \text{ \Rightarrow sum-list } (\text{map } f \; xs) < \text{sum-list } (\text{map } g \; xs) \)

proof (induction \( xs \))

case Nil thus \(?case\) by simp

next

case C: \((\text{Cons } - xs)\)

show \(?case\)

proof (cases \( xs \))

  case Nil thus \(?thesis\) using C.prems by simp

next

  case Cons thus \(?thesis\) using C by (simp add: add-strict-mono)

qed

**lemma (in monoid-add) sum-list-distinct-conv-sum-set:**

\( \text{distinct } xs \Rightarrow \text{sum-list } (\text{map } f \; xs) = \text{sum } f \; (\text{set } xs) \)

by (induct \( xs \)) simp-all

**lemma (in monoid-add) interv-sum-list-conv-sum-set-nat:**

\( \text{sum-list } (\text{map } f \; [m..<n]) = \text{sum } f \; (\text{set } [m..<n]) \)

by (simp add: sum-list-distinct-conv-sum-set)

**lemma (in monoid-add) interv-sum-list-conv-sum-set-int:**

\( \text{sum-list } (\text{map } f \; [k..l]) = \text{sum } f \; (\text{set } [k..l]) \)

by (simp add: sum-list-distinct-conv-sum-set)

General equivalence between \( \text{sum-list} \) and \( \text{sum} \)

**lemma (in monoid-add) sum-list-sum-nth:**

\( \text{sum-list } xs = (\sum i = 0 ..< \text{length } xs. xs ! i) \)

using interv-sum-list-conv-sum-set-nat \([\text{of } (!)]\) \( xs \theta \text{ length } xs \) by (simp add: map-nth)

**lemma sum-list-map-eq-sum-count:**

\( \text{sum-list } (\text{map } f \; xs) = \text{sum } (\lambda x. \text{count-list } xs x * f x) \; (\text{set } xs) \)

proof (induction \( xs \))

  case (Cons \( x \; xs \))

  show \(?case\) \( \text{is } !l = !r \)

  proof cases

    assume \( x \in \text{set } xs \)
have \( \forall l = f x + (\sum x \in \text{set} \, \text{xs} \cdot \text{count-list} \, \text{xs} \, x \cdot f \, x) \) by (simp add: Cons.IH)
also have \( \text{set} \, \text{xs} = \text{insert} \, x \, (\text{set} \, \text{xs} \, - \{x\}) \) using \( x \in \text{set} \, \text{xs} \) by blast
also have \( f \, x + (\sum x \in \text{insert} \, x \, (\text{set} \, \text{xs} \, - \{x\}) \cdot \text{count-list} \, \text{xs} \, x \cdot f \, x) = ?r \)
by (simp add: sum.insert-remove eq-commute)
finally show \( \?\text{thesis} \).

next
assume \( x \notin \text{set} \, \text{xs} \)
hence \( \forall xa. \, xa \in \text{set} \, \text{xs} \implies x \neq xa \) by blast
thus \( \?\text{thesis} \) by (simp add: Cons.IH (\( x \notin \text{set} \, \text{xs} \)))
qed

qed simp

lemma sum-list-map-eq-sum-count2:
assumes \( \text{set} \, \text{xs} \subseteq X \, \text{finite} \, \text{X} \)
shows \( \text{sum-list} \, (\text{map} \, f \, \text{xs}) = \text{sum} \, (\lambda x. \, \text{count-list} \, \text{xs} \, x \cdot f \, x) \, X \)
proof
let \( \?F = \lambda x. \, \text{count-list} \, \text{xs} \, x \cdot f \, x \)
have \( \text{sum} \, \?F \, X = \text{sum} \, \?F \, (\text{set} \, \text{xs} \cup (X - \text{set} \, \text{xs})) \)
using Un-absorb1[OF assms(1)] by (simp)
also have \( ... = \text{sum} \, \?F \, (\text{set} \, \text{xs}) \)
using assms(2)
by (simp add: sum.union-disjoint[OF - - Diff-disjoint] del: Un-Diff-cancel)
finally show \( \?\text{thesis} \) by (simp add: sum-list-map-eq-sum-count)

qed

lemma sum-list-nonneg:
(\( \forall x. \, x \in \text{set} \, \text{xs} \implies x :: 'a :: ordered-comm-monoid-add \geq 0 \) \( \implies \) sum-list \( \text{xs} \geq 0 \))
by (induction \( \text{xs} \)) simp-all

lemma sum-list-Suc:
sum-list (\( \text{map} \, (\lambda x. \, \text{Suc} \, (f \, x)) \, \text{xs} \)) = sum-list (\( \text{map} \, f \, \text{xs} \)) + length \( \text{xs} \)
by (induction \( \text{xs} \); simp)

lemma (in monoid-add) sum-list-map-filter4:
sum-list (\( \text{map} \, (f \, (\text{filter} \, P \, \text{xs})) \)) = sum-list (\( \text{map} \, (\lambda x. \, \text{if} \, P \, x \, \text{then} \, f \, x \, \text{else} \, 0) \, \text{xs} \))
by (induction \( \text{xs} \)) simp-all

Summation of a strictly ascending sequence with length \( n \) can be upper-bounded by summation over \( \{0..<\, n\} \).

lemma sorted-wrt-less-sum-mono-lowerbound:
fixes \( f :: \text{nat} \Rightarrow (b :: \text{ordered-comm-monoid-add}) \)
assumes mono: \( \forall x, \, y. \, x \leq y \implies f \, x \leq f \, y \)
shows \( \text{sorted-wrt} \, (<) \, \text{ns} \implies (\sum i \in \{0..<\, \text{length} \, \text{ns}\}. \, f \, i) \leq (\sum i \in \text{ns}. \, f \, i) \)
proof (induction \( \text{ns} \) rule: rev-induct)
case Nil
then show \( \?\text{case} \) by simp
next
THEORY “Groups-List”

case (snoc n ns)
have sum f {0..<length (ns @ [n])}
  = sum f {0..<length ns} + f (length ns)
  by simp
also have sum f {0..<length ns} ≤ sum-list (map f ns)
  using snoc by (auto simp: sorted-wrt-append)
also have length ns ≤ n
  using sorted-wrt-less-idx[OF snoc.prems(1), of length ns] by auto
finally have sum f {0..<length (ns @ [n])} ≤ sum-list (map f ns) + f n
  using mono add-mono by blast
thus ?case by simp
qed

67.2 Further facts about List·n-lists

lemma length-n-lists: length (List·n-lists n xs) = length xs ^ n
  by (induct n) (auto simp add: comp-def length-concat sum-list-triv)

lemma distinct-n-lists:
  assumes distinct xs
  shows distinct (List·n-lists n xs)
  proof (rule card-distinct)
    from assms have card-length: card (set xs) = length xs by (rule distinct-card)
    have card (set (List·n-lists n xs)) = card (set xs) ^ n
      proof (induct n)
        case 0 then show ?case by simp
        next
        case (Suc n)
        moreover have card (∪ys∈set (List·n-lists n xs). (λy. y # ys) ' set xs)
          = (∑ys∈set (List·n-lists n xs). card ((λy. y # ys) ' set xs))
          by (rule card-UN-disjoint) auto
        moreover have (∀ys. card ((λy. y # ys) ' set xs) = card (set xs)
          by (rule card-image) (simp add: inj-on-def)
        ultimately show ?case by auto
      qed
    also have . . = length xs ^ n by (simp add: card-length)
    finally show card (set (List·n-lists n xs)) = length (List·n-lists n xs)
      by (simp add: length-n-lists)
  qed

67.3 Tools setup

lemmas sum-code = sum-set-conv-list

lemma sum-set-upto-conv-sum-list-int [code-unfold]:
  sum f (set [i..j::int]) = sum-list (map f [i..j])
  by (simp add: interv-sum-list-conv-sum-set-int)

lemma sum-set-upto-conv-sum-list-nat [code-unfold]:
  sum f (set [m..<n]) = sum-list (map f [m..<n])
by (simp add: interv-sum-list-cone-sum-set-nat)

67.4 List product

context monoid-mult
begin

sublocale prod-list: monoid-list times 1
defines prod-list = prod-list.
.
end

context comm-monoid-mult
begin

sublocale prod-list: comm-monoid-list times 1
rewrites monoid-list.F times 1 = prod-list
proof –
  then interpret prod-list: comm-monoid-list times 1.
  from prod-list-def show monoid-list.F times 1 = prod-list by simp
qed

sublocale prod: comm-monoid-list-set times 1
rewrites monoid-list.F times 1 = prod-list
and comm-monoid-set.F times 1 = prod
proof –
  then interpret prod: comm-monoid-list-set times 1.
  from prod-list-def show monoid-list.F times 1 = prod-list by simp
  from prod-def show comm-monoid-set.F times 1 = prod by (auto intro: sym)
qed

end

Some syntactic sugar:

syntax (ASCII)
  -prod-list :: pttrn => 'a list => 'b => 'b ((3PROD -<- - .) [0, 51, 10] 10)
syntax
  -prod-list :: pttrn => 'a list => 'b => 'b ((3∏ -<-. -) [0, 51, 10] 10)
translations — Beware of argument permutation!
  ∏ x←xs. b ⇔ CONST prod-list (CONST map (λx. b) xs)

context
  includes lifting-syntax
begin
lemma prod-list-transfer [transfer-rule]:
(list-all2 A ===> A) prod-list prod-list
  if [transfer-rule]: A 1 (A ===> A ===> A) (+) (+)
unfolding prod-list.eq-foldr [abs-def]
by transfer-prover
end

lemma prod-list-zero-iff:
  prod-list xs = 0 <-> (0 :: 'a :: {semiring-no-zero-divisors, semiring-1}) ∈ set xs
by (induction xs) simp-all
end

68  A HOL random engine

theory Random
imports List Groups-List
begin

notation fcomp (infixl ◦ 60)
notation scomp (infixl ◦→ 60)

68.1  Auxiliary functions

fun log :: natural ⇒ natural ⇒ natural where
  log b i = (if b ≤ 1 ∨ i < b then 1 else 1 + log b (i div b))

definition inc-shift :: natural ⇒ natural ⇒ natural where
  inc-shift v k = (if v = k then 1 else k + 1)

definition minus-shift :: natural ⇒ natural ⇒ natural ⇒ natural where
  minus-shift r k l = (if k < l then r + k − l else k − l)

68.2  Random seeds

type-synonym seed = natural × natural

primrec next :: seed ⇒ natural × seed where
next (v, w) = (let
  k = v div 53668;
  v' = minus-shift 2147483563 ((v mod 53668) * 40014) (k * 12211);
  l = w div 52774;
  w' = minus-shift 2147483399 ((w mod 52774) * 40692) (l * 3791);
  z = minus-shift 2147483562 v' (w' + 1) + 1
in (z, (v', w')))

definition split-seed :: seed ⇒ seed × seed where
split-seed s = (let
    (v, w) = s;
    (v', w') = snd (next s);
    v'' = inc-shift 2147483562 v;
    w'' = inc-shift 2147483398 w
in ((v'', w'), (v', w'')))

68.3 Base selectors

fun iterate :: natural ⇒ (′b ⇒ ′a ⇒ ′b × ′a) ⇒ ′b ⇒ ′a ⇒ ′b × ′a where
iterate k f x = (if k = 0 then Pair x else f x ◦→ iterate (k - 1) f)

definition range :: natural ⇒ seed ⇒ natural × seed where
range k = iterate (log 2147483561 k)
  (λ. next ◦→ (λv. Pair (v + l * 2147483561))) 1
  ◦→ (λv. Pair (v mod k))

definition select :: ′a list ⇒ seed ⇒ ′a × seed where
select xs = range (natural-of-nat (length xs))
  ◦→ (λk. Pair (nth xs (nat-of-natural k)))

lemma range:
k > 0 ⇒ fst (range k s) < k
by (simp add: range-def split-def less-natural-def del: log.simps iterate.simps)

lemma select:
assumes xs ≠ []
shows fst (select xs s) ∈ set xs
proof –
  from assms have natural-of-nat (length xs) > 0 by (simp add: less-natural-def)
  with range have
    fst (range (natural-of-nat (length xs)) s) < natural-of-nat (length xs) by best
  then have
    nat-of-natural (fst (range (natural-of-nat (length xs)) s)) < length xs by (simp add: less-natural-def)
  then show ?thesis
    by (simp add: split-beta select-def)
qed

primrec pick :: (natural × ′a) list ⇒ natural ⇒ ′a where
pick (x # xs) i = (if i < fst x then snd x else pick xs (i - fst x))

lemma pick-member:
i < sum-list (map fst xs) ⇒ pick xs i ∈ set (map snd xs)
by (induct xs arbitrary: i) (simp-all add: less-natural-def)

lemma pick-drop-zero:
pick (filter (λ(k, -). k > 0) xs) = pick xs
by (induct xs) (auto simp add: fun-eq-iff less-natural-def minus-natural-def)
lemma pick-same:
  \( l < \text{length} \ x s \implies \text{Random.pick} (\text{map} (\text{Pair} \ 1) \ x s) (\text{natural-of-nat} \ l) = \text{nth} \ x s \ l \)

proof (induct \ x s \ arbitrary: \ l)
  case Nil then show \(?case\) by simp
next
  case (Cons \ x \ x s) then show \(?case\) by (cases \ l\) (simp-all add: less-natural-def)
qed

definition select-weight :: \((\text{natural} \times \ 'a)\ \text{list} \Rightarrow \ 'a \times \text{seed}\) where
  select-weight \ x s = \text{range} (sum-list (map \ fst \ x s))
  \circ \rightarrow (\lambda k. \text{Pair} (\text{pick} \ x s \ k))

lemma select-weight-member:
  assumes \( 0 < \text{sum-list} (\text{map} \ fst \ x s) \)
  shows \( \text{fst} (\text{select-weight} \ x s \ s) \in \text{set} (\text{map} \ \text{snd} \ x s) \)

proof —
  from range assms have \( \text{fst} (\text{range} (\text{sum-list} (\text{map} \ fst \ x s)) \ s) < \text{sum-list} (\text{map} \ fst \ x s) \).
  with pick-member have \( \text{pick} \ x s (\text{fst} (\text{range} (\text{sum-list} (\text{map} \ fst \ x s)) \ s)) \in \text{set} (\text{map} \ \text{snd} \ x s) \).
  then show \(?thesis\) by (simp add: select-weight-def scomp-def split-def)
qed

lemma select-weight-cons-zero:
  select-weight ((0, x) \# x s) = select-weight x s
by (simp add: select-weight-def less-natural-def)

lemma select-weight-drop-zero:
  select-weight (filter (\( \lambda(k, \cdot). \ k > 0 \) \ x s) \ x s) = select-weight x s

proof —
  have \( \text{sum-list} (\text{map} \ \text{fst} \ [(k, \cdot)\leftarrow x s. \ 0 < k]) = \text{sum-list} (\text{map} \ \text{fst} \ x s) \)
  by (induct \ x s\) (auto simp add: less-natural-def natural-eq-iff)
  then show \(?thesis\) by (simp only: select-weight-def pick-drop-zero)
qed

lemma select-weight-select:
  assumes \( \ x s \neq [] \)
  shows select-weight (map (Pair 1) x s) = select x s

proof —
  have \( \text{less}: \text{\bigwedge} s. \text{fst} (\text{range} (\text{natural-of-nat} (\text{length} x s)) \ s) < \text{natural-of-nat} (\text{length} x s) \)
  using assms by (intro range) (simp add: less-natural-def)
  moreover have \( \text{sum-list} (\text{map} \ \text{fst} (\text{map} (\text{Pair} 1) x s)) = \text{natural-of-nat} (\text{length} x s) \)
  by (induct \ x s\) simp-all
  ultimately show \(?thesis\)
  by (auto simp add: select-weight-def select-def scomp-def split-def fun-eq-iff pick-same [symmetric] less-natural-def)
68.4 ML interface

code-reflect Random-Engine
  functions range select select-weight

ML |
structure Random-Engine =
  struct
  open Random-Engine;
  type seed = Code-Numeral.natural * Code-Numeral.natural;
  local
  val seed = Unsynchronized.ref
    (let
     val (now) = Time.toMilliseconds (Time.now ());
     val (q, s1) = IntInf.divMod (now, 2147483562);
     val s2 = q mod 2147483398;
     in apply2 Code-Numeral.natural-of-integer (s1 + 1, s2 + 1) end);
  in
  fun next-seed () =
    let
      val (seed1, seed') = @code-split-seed (! seed)
      val _ = seed := seed'
      in
        seed1
      end
  fun run f =
    let
      val (x, seed') = f (! seed);
      val _ = seed := seed'
      in x end;
    end;
  end;
  }

hide-type (open) seed
hide-const (open) inc-shift minus-shift log next split-seed
  iterate range select pick select-weight
hide-fact (open) range-def
no-notation \textit{fcomp} \ (<text{infixl} \circ \rightarrow 60)\no-notation \textit{scomp} \ (<text{infixl} \circ \rightarrow 60)\end

69 Maps

theory \textit{Map} 
imports \textit{List} 
abbrevs (= = \subseteq_m 
begin 
type-synonym \textit{('}a, \textit{'}b) map = \textit{'}a \Rightarrow \textit{'}b \ \text{option} \ (<text{infixr} \ 	riangleright 0)\abbreviation \textit{empty} :: \textit{'}a \Rightarrow \textit{'}b \ \text{where} 
\textit{empty} \equiv \lambda x. \text{None} 
definition \textit{map-comp} :: (\textit{'}b \Rightarrow \textit{'}c) \Rightarrow (\textit{'}a \Rightarrow \textit{'}b) \Rightarrow (\textit{'}a \Rightarrow \textit{'}c) \ (<text{infixl} \circ_m 55) \ \text{where} 
f \circ_m g = (\lambda k. \text{case } g k \text{ of } \text{None } \Rightarrow \text{None } | \text{Some } v \Rightarrow \text{f } v) 
definition \textit{map-add} :: (\textit{'}a \Rightarrow \textit{'}b) \Rightarrow (\textit{'}a \Rightarrow \textit{'}b) \Rightarrow (\textit{'}a \Rightarrow \textit{'}b) \ (<text{infixl} \ ++ 100) \ \text{where} 
m1 ++ m2 = (\lambda x. \text{case } m2 x \text{ of } \text{None } \Rightarrow \text{m1 } x | \text{Some } y \Rightarrow \text{Some } y) 
definition \textit{restrict-map} :: (\textit{'}a \Rightarrow \textit{'}b) \Rightarrow \textit{'}a \ \text{set} \Rightarrow (\textit{'}a \Rightarrow \textit{'}b) \ (<text{infixl} \ | \ 110) \ \text{where} 
m|A = (\lambda x. \text{if } x \in A \text{ then } m x \text{ else } \text{None}) 
notation (latex output) 
\textit{restrict-map} \ (<text{infixl} \ [-] [111,110] 110) 
definition \textit{dom} :: (\textit{'}a \Rightarrow \textit{'}b) \Rightarrow \textit{'}a \ \text{set} \ \text{where} 
\textit{dom} m = \{a. \text{m } a \neq \text{None}\} 
definition \textit{ran} :: (\textit{'}a \Rightarrow \textit{'}b) \Rightarrow \textit{'}b \ \text{set} \ \text{where} 
\textit{ran} m = \{b. \exists a. \text{m } a = \text{Some } b\} 
definition \textit{map-le} :: (\textit{'}a \Rightarrow \textit{'}b) \Rightarrow (\textit{'}a \Rightarrow \textit{'}b) \Rightarrow \text{bool} \ (<text{infix} \subseteq_m 50) \ \text{where} 
(m_1 \subseteq_m m_2) \longleftrightarrow (\forall a \in \text{dom } m_1. \text{m }_1 a = \text{m }_2 a) 
nonterminal maplets and maplet 
syntax
THEORY "Map"

-maplet :: ['a, 'a] ⇒ maplet (- /⇒/ -)
-maplets :: ['a, 'a] ⇒ maplet (- /⇒/ -)
 : maplet ⇒ maplets (-)
-Maplets :: [maplet, maplets] ⇒ maplets (-,/ -)
-MapUpd :: ['a → 'b, maplets] ⇒ 'a → 'b (-/(-) [900, 0] 900)
-Map :: maplets ⇒ 'a → 'b (((1|-}))

syntax (ASCII)
-maplet :: ['a, 'a] ⇒ maplet (- /⇒/ -)
-maplets :: ['a, 'a] ⇒ maplet (- /⇒/ -)

translations
-MapUpd m (-Maplets xy ms) ⇐ -MapUpd (-MapUpd m xy) ms
-MapUpd m (-maplet x y) ⇐ m(x := CONST Some y)
-Map ms ⇐ -MapUpd (CONST empty) ms
-Map (-Maplets ms1 ms2) ⇐ -MapUpd (-Map ms1) ms2
-Maplets ms1 (-Maplets ms2 ms3) ⇐ -Maplets (-Maplets ms1 ms2) ms3

primrec map-of :: ('a × 'b) list ⇒ 'a → 'b
where
map-of [] = empty
| map-of (p # ps) = (map-of ps)(fst p ↦→ snd p)

definition map-upds :: ('a → 'b) ⇒ 'a list ⇒ 'b list ⇒ 'a → 'b
where map-upds m xs ys = m ++ map-of (rev (zip xs ys))

definition map-upds :: ('a → 'b) ⇒ 'a list ⇒ 'b list ⇒ 'a → 'b
where map-upds m xs ys = m ++ map-of (rev (zip xs ys))

definition map-upds :: ('a → 'b) ⇒ 'a list ⇒ 'b list ⇒ 'a → 'b
where map-upds m xs ys = m ++ map-of (rev (zip xs ys))

translations
-MapUpd m (-maplets x y) ⇐ CONST map-upds m x y

lemma map-of-Cons-code [code]:
map-of [] k = None
map-of ((l, v) # ps) k = (if l = k then Some v else map-of ps k)
by simp-all

69.1 empty
lemma empty-upd-none [simp]: empty(x := None) = empty
by (rule ext) simp

69.2 map-upd
lemma map-upd-triv: t k = Some x ⇒ t(k→x) = t
by (rule ext) simp

lemma map-upd-nonempty [simp]: t(k→x) ≠ empty
proof
  assume t(k→x) = empty
  then have (t(k→x)) k = None by simp
  then show False by simp
qed
lemma `map-upd-eqD1`: assumes `m(a↦x) = n(a↦y)` shows `x = y` proof - from assms have `(m(a↦x)) a = (n(a↦y)) a` by simp then show ?thesis by simp qed

lemma `map-upd-Some-unfold`: `((m(a↦b)) x = Some y) = (x = a ∧ b = y ∨ x ≠ a ∧ m x = Some y)` by auto

lemma `image-map-upd [simp]`: `x ∉ A ⟹ m(x↦y) A = m A` by auto

lemma `finite-range-updI [simp]`: `finite (range f) ⟹ finite (range (f(a↦b)))` unfolding `image-def` apply (simp (no-asm-use) add:full-SetCompr-eq) apply (rule finite-subset) prefer 2 apply assumption apply (auto) done

69.3 `map-of`

lemma `map-of-eq-empty-iff [simp]`: `map-of xys = empty ⟷ xys = []` proof show `map-of xys = empty ⟹ xys = []` by (induction xys) simp-all qed simp

lemma `empty-eq-map-of-iff [simp]`: `empty = map-of xys ⟷ xys = []` by (subst eq-commute) simp

lemma `map-of-eq-None-iff`: `(map-of xys x = None) = (x ∉ fst '(set xys))` by (induct xys) simp-all

lemma `Some-eq-map-of-iff [symmetric]`: `distinct(map fst xys) ⟹ (map-of xys x = Some y) = ((x,y) ∈ set xys)` apply (induct xys) apply simp apply (auto simp: map-of-eq-None-iff) done

lemma `Some-eq-map-of-iff [symmetric]`: `distinct(map fst xys) ⟹ (Some y = map-of xys x) = ((x,y) ∈ set xys)`
by (auto simp del: map-of-eq-Some-iff simp: map-of-eq-Some-iff [symmetric])

lemma map-of-is-SomeI [simp]: \[ \text{distinct}(\text{map \text{fst} xys}); (x,y) \in \text{set xys} \] \[ \implies \text{map-of} \ xys \ x = \text{Some} \ y \]
apply (induct xys)
apply simp
apply force
done

lemma map-of-zip-is-None [simp]:
\[ \text{length} \ \text{xs} = \text{length} \ ys \implies (\text{map-of} \ (\text{zip} \ \text{xs} \ ys) \ x = \text{None}) = (x \notin \text{set} \ \text{xs}) \]
by (induct rule: list-induct2) simp-all

lemma map-of-zip-is-Some:
assumes \[ \text{length} \ \text{xs} = \text{length} \ ys \]
shows \[ x \in \text{set} \ \text{xs} \iff (\exists y. \text{map-of} \ (\text{zip} \ \text{xs} \ ys) \ x = \text{Some} \ y) \]
using assms by (induct rule: list-induct2) simp-all

lemma map-of-zip-upd:
fixes \[ x :: 'a \text{ and} \ \text{xs} :: 'a \text{ list and} \ ys \ zs :: 'b \text{ list} \]
assumes \[ \text{length} \ ys = \text{length} \ \text{xs} \]
and \[ \text{length} \ zs = \text{length} \ \text{xs} \]
and \[ x \notin \text{set} \ \text{xs} \]
and \[ \text{map-of} \ (\text{zip} \ \text{xs} \ ys)(x \mapsto y) = \text{map-of} \ (\text{zip} \ \text{xs} \ zs)(x \mapsto z) \]
shows \[ \text{map-of} \ (\text{zip} \ \text{xs} \ ys) = \text{map-of} \ (\text{zip} \ \text{xs} \ zs) \]
proof
fix \[ x' :: 'a \]
show \[ \text{map-of} \ (\text{zip} \ \text{xs} \ ys) \ x' = \text{map-of} \ (\text{zip} \ \text{xs} \ zs) \ x' \]
proof (cases \[ x = x' \])
  case True
  from assms True map-of-zip-is-None [of \[ \text{xs} \ ys \ x' \]
  have \[ \text{map-of} \ (\text{zip} \ \text{xs} \ ys) \ x' = \text{None} \]
  moreover from assms True map-of-zip-is-None [of \[ \text{xs} \ zs \ x' \]
  have \[ \text{map-of} \ (\text{zip} \ \text{xs} \ zs) \ x' = \text{None} \]
  ultimately show \[ ?\text{thesis} \]
  by simp
next
  case False from assms
  have \[ (\text{map-of} \ (\text{zip} \ \text{xs} \ ys)(x \mapsto y)) \ x' = (\text{map-of} \ (\text{zip} \ \text{xs} \ zs)(x \mapsto z)) \ x' \]
  by auto
  with False show \[ ?\text{thesis} \]
  by simp
qed

lemma map-of-zip-inject:
assumes \[ \text{length} \ ys = \text{length} \ \text{xs} \]
and \[ \text{length} \ zs = \text{length} \ \text{xs} \]
and \[ \text{distinct} \ \text{xs} \]
and \[ \text{map-of}: \text{map-of} \ (\text{zip} \ \text{xs} \ ys) = \text{map-of} \ (\text{zip} \ \text{xs} \ zs) \]
shows \[ ys = zs \]
using assms\(1\) assms\(2\)[symmetric]

proof (induct ys xs zs rule: list-induct3)
  case Nil show ?case by simp
next
  case (Cons y ys x xs z zs)
  from ⟨map-of (zip (x#xs) (y#ys)) = map-of (zip (x#xs) (z#zs))⟩
  have map-of: map-of (zip xs ys)(x ↦→ y) = map-of (zip xs zs)(x ↦→ z) by simp
  from Cons have length ys = length xs and length zs = length xs
  and \(x \notin set xs\) by simp-all
  then have map-of (zip xs ys) = map-of (zip xs zs) using map-of by (rule map-of-zip-upd)
  with Cons.hyps ⟨distinct (x # xs)⟩ have ys = zs by simp
  moreover from map-of have y = z by (rule map-upd-eqD1)
  ultimately show ?case by simp
qed

lemma map-of-zip-nth:
  assumes length xs = length ys
  assumes distinct xs
  assumes i < length ys
  shows map-of (zip xs ys) (xs ! i) = Some (ys ! i)
  using assms proof (induct arbitrary: \(i\) rule: list-induct2)
    case Nil
    then show ?case by simp
  next
    case (Cons x xs y ys)
    then show ?case
    using less-Suc-eq-0-disj by auto
  qed

lemma map-of-zip-map:
  map-of (zip xs (map f xs)) = (λx. if x ∈ set xs then Some (f x) else None)
  by (induct xs) (simp-all add: fun-eq-iff)

lemma finite-range-map-of: finite (range (map-of xys))
apply (induct xys)
apply (simp-all add: image-constant)
apply (rule finite-subset)
prefer 2 apply assumption
apply auto
done

lemma map-of-SomeD: map-of xs k = Some \(y\) ⇒ (k, y) ∈ set xs
by (induct xs) (auto split: if-splits)

lemma map-of-mapk-SomeI:
  inj \(f\) ⇒ map-of t k = Some \(x\) ⇒
  map-of (map (case-prod (λk. Pair (f k))) t) (f k) = Some \(x\)
by (induct t) (auto simp: inj-eq)

lemma weak-map-of-SomeI: \((k, x) \in \text{set } l \Rightarrow \exists x. \text{map-of } l \ k = \text{Some } x\)
by (induct l) auto

lemma map-of-filter-in:
\(\text{map-of } xs \ k = \text{Some } z \Rightarrow \text{P } k \ z \Rightarrow \text{map-of } (\text{filter } (\text{case-prod } P) \ xs) \ k = \text{Some } z\)
by (induct xs) auto

lemma map-of-map:
\(\text{map-of } (\lambda(k, v). (k, f \ v)) \ xs = \text{map-option } f \circ \text{map-of } xs\)
by (induct xs) (auto simp: fun-eq-iff)

lemma dom-map-option:
\(\text{dom } (\lambda k. \text{map-option } (f k) \ (m k)) = \text{dom } m\)
by (simp add: dom-def)

lemma dom-map-option-comp [simp]:
\(\text{dom } (\text{map-option } g \circ m) = \text{dom } m\)
using dom-map-option [of \(\lambda\cdot g\) m] by (simp add: comp-def)

69.4 \text{ map-option related}

lemma map-option-o-empty [simp]: \(\text{map-option } f \circ \text{empty} = \text{empty}\)
by (rule ext) simp

lemma map-option-o-map-upd [simp]:
\(\text{map-option } f \circ m(a \mapsto b) = (\text{map-option } f \circ m)(a \mapsto f b)\)
by (rule ext) simp

69.5 \text{ map-comp related}

lemma map-comp-empty [simp]:
\(m \circ_m \text{empty} = \text{empty}\)
\(\text{empty} \circ_m m = \text{empty}\)
by (auto simp: map-comp-def split: option.splits)

lemma map-comp-simps [simp]:
\(m_2 k = \text{None} \Rightarrow (m_1 \circ_m m_2) \ k = \text{None}\)
\(m_2 k = \text{Some } k' \Rightarrow (m_1 \circ_m m_2) \ k = m_1 k'\)
by (auto simp: map-comp-def)

lemma map-comp-None-iff:
\((m_1 \circ_m m_2) \ k = \text{None} \Rightarrow (\exists k'. m_2 k = \text{Some } k' \land m_1 k' = \text{Some } v)\)
by (auto simp: map-comp-def split: option.splits)

lemma map-comp-None-iff:
\((m_1 \circ_m m_2) \ k = \text{None} \Rightarrow (m_2 k = \text{None} \lor (\exists k'. m_2 k = \text{Some } k' \land m_1 k' = \text{None}))\)
by (auto simp: map-comp-def split: option.splits)

69.6  ++

lemma map-add-empty[simp]: m ++ empty = m
by(simp add: map-add-def)

lemma empty-map-add[simp]: empty ++ m = m
by (rule ext) (simp add: map-add-def split: option.split)

lemma map-add-assoc[simp]: m1 ++ (m2 ++ m3) = (m1 ++ m2) ++ m3
by (rule ext) (simp add: map-add-def split: option.split)

lemma map-add-Some-iff:
  ((m ++ n) k = Some x) = (n k = Some x ∨ n k = None ∧ m k = Some x)
by (simp add: map-add-def split: option.split)

lemma map-add-SomeD [dest!]:
  (m ++ n) k = Some x ⇒ n k = Some x ∨ n k = None ∧ m k = Some x
by (rule map-add-Some-iff [THEN iffD1])

lemma map-add-find-right [simp]: n k = Some x ⇒ (m ++ n) k = Some x
by (subst map-add-Some-iff) fast

lemma map-add-None [iff]: ((m ++ n) k = None) = (n k = None ∧ m k = None)
by (simp add: map-add-def split: option.split)

lemma map-add-upd[simp]: f ++ g(x→y) = (f ++ g)(x→y)
by (rule ext) (simp add: map-add-def)

lemma map-add-upds[simp]: m1 ++ (m2(xs→ys)) = (m1++m2)(xs→ys)
by (simp add: map-upds-def)

lemma map-add-upd-left: m∉ dom e2 ⇒ e1(m → u1) ++ e2 = (e1 ++ e2)(m → u1)
by (rule ext) (auto simp: map-add-def dom-def split: option.split)

lemma map-of-append[simp]: map-of (xs @ ys) = map-of ys ++ map-of xs

unfolding map-add-def
apply (induct xs)
  apply simp
  apply (rule ext)
  apply (simp split: option.split)
done

lemma finite-range-map-of-map-add:
  finite (range f) ⇒ finite (range (f ++ map-of l))
apply (induct l)
apply (auto simp del: fun-upd-apply)
apply (erule finite-range-updI)
done

lemma inj-on-map-add-dom [iff]:
inj-on (m ++ m') (dom m') = inj-on m (dom m')
by (fastforce simp: map-add-def dom-def inj-on-def split: option.splits)

lemma map-upds-fold-map-upd:
m(ks [↦→] vs) = foldl (λm (k, v). m(k [→] v)) m (zip ks vs)
unfolding map-upds-def proof (rule sym, rule zip-obtain-same-length)
  fix ks :: 'a list and vs :: 'b list
  assume length ks = length vs
  then show foldl (λm (k, v). m(k [→] v)) m (zip ks vs) = m ++ map-of (rev (zip ks vs))
    by (induct arbitrary: m rule: list-induct2) simp-all
qed

lemma restrict-map-to-empty [simp]: m{|{}} = empty
by (simp add: restrict-map-def)

lemma restrict-map-insert: f |' (insert a A) = (f |' A)(a := f a)
by (auto simp: restrict-map-def)

lemma restrict-map-empty [simp]: empty|'D = empty
by (simp add: restrict-map-def)

lemma restrict-in [simp]: x ∈ A =⇒ (m|'A) x = m x
by (simp add: restrict-map-def)

lemma restrict-out [simp]: x ∉ A =⇒ (m|'A) x = None
by (simp add: restrict-map-def)

lemma ran-restrictD: y ∈ ran (m|'A) =⇒ ∃x∈A. m x = Some y
by (auto simp: restrict-map-def ran-def split: if-split_asm)

lemma dom-restrict [simp]: dom (m|'A) = dom m ∩ A
by (auto simp: restrict-map-def dom-def split: if-split_asm)

lemma restrict-upd-same [simp]: m(x [↦→] y)|'(-{x}) = m|'(-{x})
by (rule ext) (auto simp: restrict-map-def)

lemma restrict-restrict [simp]: m|'A|'B = m|'A∩B
THEORY "Map"

by (rule ext) (auto simp: restrict-map-def)

lemma restrict-fun-upd [simp]:
  \( m(x := y)|'D = (\text{if } x \in D \text{ then } (m|'(D\setminus\{x\}))(x := y) \text{ else } m|'D) \)
by (simp add: restrict-map-def fun-eq-iff)

lemma fun-upd-None-restrict [simp]:
  \( (m|'D)(x := \text{None}) = (\text{if } x \in D \text{ then } m|'(D\setminus\{x\}) \text{ else } m|'D) \)
by (simp add: restrict-map-def fun-eq-iff)

lemma fun-upd-restrict [simp]:
  \( (m|'D)(x := y) = (m|'(D\setminus\{x\}))(x := y) \)
by (simp add: restrict-map-def fun-eq-iff)

lemma fun-upd-restrict-conv [simp]:
  \( x \in D \Longrightarrow (m|'D)(x := y) = (m|'(D\setminus\{x\}))(x := y) \)
by (simp add: restrict-map-def fun-eq-iff)

lemma map-of-map-restrict:
  \( \text{map-of } (\lambda k. (k, f k))\ ks = (\text{Some } \circ f) \restriction \text{set } ks \)
by (induct ks) (simp-all add: fun-eq-iff restrict-map-insert)

lemma restrict-complement-singleton-eq:
  \( f \restriction (-\{x\}) = f(x := \text{None}) \)
by (simp add: restrict-map-def fun-eq-iff)

69.8  map-upds

lemma map-upds-Nil1 [simp]: \( m(\text{[]} \mapsto bs) = m \)
by (simp add: map-upds-def)

lemma map-upds-Nil2 [simp]: \( m(\text{as} \mapsto \text{[]} \restriction bs) = m \)
by (simp add: map-upds-def)

lemma map-upds-Cons [simp]: \( m(a\#\text{as} \mapsto b\#bs) = (m(a\mapsto b))\restriction as\mapsto bs \)
by (simp add: map-upds-def)

lemma map-upds-append1 [simp]: size xs < size ys \(\Longrightarrow \)
  \( m(xs@[x] \mapsto y) = m(xs \mapsto ys)(x \mapsto ys!\text{size } xs) \)
apply (induct xs arbitrary: ys m)
apply (clarsimp simp add: neq-Nil-conv)
apply (case-tac ys)
apply simp
apply simp
done

lemma map-upds-list-update2-drop [simp]:
  size xs \(\leq i \Longrightarrow \)
  \( m(xs[i := y]) = m(xs[i := y]) \)
apply (induct xs arbitrary: m ys i)
apply simp
apply (case-tac ys)
apply simp
apply (simp split: nat.split)
done

lemma map-upd-upds-conv-if:
(f(x\rightarrow y))(xs [|\rightarrow|] ys) =
(if x \in set(take (length ys) xs) then f(xs [|\rightarrow|] ys)
else (f(xs [|\rightarrow|] ys))(x\rightarrow y))
apply (induct xs arbitrary: x y ys f)
apply simp
apply (case-tac ys)
apply (auto split: if-split simp: fun-upd-twist)
done

lemma map-upds-twist [simp]:
a \notin set as \implies m(a\rightarrow b)(as\rightarrow|bs) = m(as\rightarrow|bs)(a\rightarrow b)
using set-take-subset by (fastforce simp add: map-upd-upds-conv-if)

lemma map-upds-apply-nontin [simp]:
x \notin set xs \implies (f(xs\rightarrow|ys))(x) = f x
apply (induct xs arbitrary: ys)
apply simp
apply (case-tac ys)
apply (auto simp: map-upd-upds-conv-if)
done

lemma fun-upds-append-drop [simp]:
size xs = size ys \implies m(xs@zs\rightarrow|ys) = m(xs\rightarrow|ys)
apply (induct xs arbitrary: m ys)
apply simp
apply (case-tac ys)
apply simp-all
done

lemma fun-upds-append2-drop [simp]:
size xs = size ys \implies m(xs\rightarrow|ys@zs) = m(xs\rightarrow|ys)
apply (induct xs arbitrary: m ys)
apply simp
apply (case-tac ys)
apply simp-all
done

lemma restrict-map-upds[simp]:
[ length xs = length ys; set xs \subseteq D ]
\implies m(xs \rightarrow| ys)|'D = (m|'(D - set xs))(xs \rightarrow| ys)
apply (induct xs arbitrary: m ys)
apply simp
apply (case-tac ys)
apply simp
apply (simp add: Diff-insert [symmetric] insert-absorb)
apply (simp add: map-upd-upds-cone-if)
done

69.9 dom

lemma dom-eq-empty-cone [simp]: dom \( f = \{ \} \) \( \iff \) \( f = \text{empty} \)
by (auto simp: dom-def)

lemma domI: \( m \ a = \text{Some} \ b \implies a \in \text{dom} \ m \)
by (simp add: dom-def)

lemma domD: \( a \in \text{dom} \ m \implies \exists b. \ m \ a = \text{Some} \ b \)
by (cases m a) (auto simp add: dom-def)

lemma domIff [iff, simp del, code-unfold]: \( a \in \text{dom} \ m \iff m \ a \neq \text{None} \)
by (simp add: dom-def)

lemma dom-empty [simp]: \( \text{dom} \ \text{empty} = \{ \} \)
by (auto simp: dom-def)

lemma dom-fun-upd [simp]:
\( \text{dom} (f(x := y)) = (\text{if} \ y = \text{None} \ \text{then} \ \text{dom} \ f - \{x\} \ \text{else} \ \text{insert} \ x \ (\text{dom} \ f)) \)
by (auto simp: dom-def)

lemma dom-if:
\( \text{dom} (\lambda x. \ \text{if} \ P \ x \ \text{then} \ f \ x \ \text{else} \ g \ x) = \text{dom} \ f \cap \{x. \ P \ x\} \cup \text{dom} \ g \cap \{x. \ \neg \ P \ x\} \)
by (auto split: if-splits)

lemma dom-map-of-conv-image-fst:
\( \text{dom} \ (\text{map-of} \ xys) = \text{fst} \cdot \text{set} \ xys \)
by (induct xys) (auto simp: dom-if)

lemma dom-map-of-zip [simp]: \( \text{length} \ \text{xs} = \text{length} \ \text{ys} \implies \text{dom} \ (\text{map-of} (\text{zip} \ \text{xs} \ \text{ys})) = \text{set} \ \text{xs} \)
by (induct rule: list-induct2) (auto simp: dom-if)

lemma finite-dom-map-of: finite (\text{dom} (\text{map-of} \ l))
by (induct l) (auto simp: dom-def insert-Collect [symmetric])

lemma dom-map-upds [simp]:
\( \text{dom}(m(\text{xs}[\rightarrow]\text{ys})) = \text{set}(\text{take} \ (\text{length} \ \text{ys}) \ \text{xs}) \cup \text{dom} \ m \)
apply (induct \text{xs} arbitrary: \text{m} \ \text{ys})
apply simp
apply (case-tac \text{ys})
apply auto
done

lemma dom-map-add [simp]: \text{dom} \ (m ++ n) = \text{dom} \ n \cup \text{dom} \ m
  by (auto simp: dom-def)

lemma dom-override-on [simp]:
  \text{dom} \ (\text{override-on} \ f \ g \ A) =
  (\text{dom} \ f \ - \ \{a. \ a \in A \ - \ \text{dom} \ g\}) \cup \{a. \ a \in A \cap \text{dom} \ g\}
  by (auto simp: dom-def override-on-def)

lemma map-add-comm: \text{dom} \ m1 \cap \text{dom} \ m2 = \{\} \implies \ m1 ++ \ m2 = \ m2 ++ \ m1
  by (rule ext) (force simp: map-add-def dom-def split: option.split)

lemma map-add-dom-app-simps:
  \ m \ \in \ \text{dom} \ l2 \ \Rightarrow \ (l1 ++ l2) \ m = l2 \ m
  \ m \ /\ \in \ \text{dom} \ l1 \ \Rightarrow \ (l1 ++ l2) \ m = l2 \ m
  \ m \ /\ \in \ \text{dom} \ l2 \ \Rightarrow \ (l1 ++ l2) \ m = l1 \ m
  by (auto simp add: map-add-def split: option.split-asm)

lemma dom-const [simp]:
  \text{dom} \ (\lambda x. \text{Some} (f x)) = \text{UNIV}
  by auto

lemma finite-map-freshness:
  finite \ (\text{dom} \ (f :: 'a \Rightarrow 'b)) \implies \neg \ finite \ (\text{UNIV} :: 'a \ set)
  \exists x. f x = \text{None}
  by (bestsimp dest: ex-new-if-finite)

lemma dom-minus:
  \ f \ x = \text{None} \ \Rightarrow \ \text{dom} \ f - \ \{x\} = \text{dom} \ f - \ A
  unfolding dom-def by simp

lemma insert-dom:
  \ f \ x = \text{Some} \ y \ \Rightarrow \ \text{insert} \ x \ (\text{dom} \ f) = \text{dom} \ f
  unfolding dom-def by auto

lemma map-of-map-keys:
  \text{set} \ xs = \text{dom} \ m \ \Rightarrow \ \text{map-of} \ (\text{map} \ (\lambda k. \ (k, \ \text{the} \ (m \ k))) \ xs) = m
  by (rule ext) (auto simp add: map-of-map-restrict restrict-map-def)

lemma map-of-eqI:
  assumes set-eq: \text{set} \ (\text{map} \ \text{fst} \ xs) = \text{set} \ (\text{map} \ \text{fst} \ ys)
  assumes map-eq: \forall k \in \text{set} \ (\text{map} \ \text{fst} \ xs) \ . \ \text{map-of} \ xs \ k = \text{map-of} \ ys \ k
  shows \text{map-of} \ xs = \text{map-of} \ ys
  proof (rule ext)
    fix k show \text{map-of} \ xs \ k = \text{map-of} \ ys \ k
    proof (cases map-of xs k)
      case None
then have \( k \notin \{a \in set (map fst xs)\} \) by (simp add: map-of-eq-None-iff)
with set-eq have \( k \notin \{a \in set (map fst ys)\} \) by simp
then have \( \map{ys} k = None \) by (simp add: map-of-eq-None-iff)
with None show ?thesis by simp

next
  case (Some \( v \))
then have \( k \in \{a \in set (map fst xs)\} \) by (auto simp add: dom-map-of-conv-image-fst)
with map-eq show ?thesis by auto
qed

lemma map-of-eq-dom:
  assumes \( \map{xs} = \map{ys} \)
  shows \( \{a \in set \map{xs} \} = \{a \in set \map{ys} \} \)
proof
  from assms have \( \dom(\map{xs}) = \dom(\map{ys}) \) by simp
  then show ?thesis by (simp add: dom-map-of-conv-image-fst)
qed

lemma finite-set-of-finite-maps:
  assumes \( A \text{ finite } B \)
  shows \( \{a \in set \{m. \dom m = A \land \ran m \subseteq B\}\} \) (is finite ?S)
proof
  let \( \text{S'} = \{a. \forall x. (x \in A \longrightarrow m x \in \text{Some } B) \land (x \notin A \longrightarrow m x = \text{None})\} \)
  have \( ?S = ?S' \)
  proof
    show \( ?S \subseteq ?S' \) by (auto simp: dom-def ran-def image-def)
    show \( ?S' \subseteq ?S \)
    proof
      fix \( m \) assume \( m \in ?S' \)
      hence \( 1: \dom m = A \) by force
      hence \( 2: \ran m \subseteq B \) using \( \{m \in ?S'\} \) by (auto simp: dom-def ran-def)
      from \( 1 2 \) show \( m \in ?S \) by blast
    qed
  qed
  with assms show ?thesis by(simp add: finite-set-of-finite-funs)
qed

69.10  \( \ran \)

lemma ranI: \( m \ a = \text{Some } b \implies b \in \ran m \)
  by (auto simp: ran-def)

lemma ran-empty [simp]: \( \ran empty = \{\} \)
  by (auto simp: ran-def)

lemma ran-map-upd [simp]: \( m \ a = \text{None } \implies \ran(m(a\mapsto b)) = \text{insert } b \ (\ran m) \)
unfolding ran-def
apply auto
apply (subgoal-tac aa ≠ a)
apply auto
done

lemma ran-map-add:
  assumes dom m1 ∩ dom m2 = {}
  shows ran (m1 ++ m2) = ran m1 ∪ ran m2
proof
  show ran (m1 ++ m2) ⊆ ran m1 ∪ ran m2
    unfolding ran-def by auto
next
  show ran m1 ∪ ran m2 ⊆ ran (m1 ++ m2)
    proof
      have (m1 ++ m2) x = Some y if m1 x = Some y for x y
        using assms map-add-comm that by fastforce
      moreover have (m1 ++ m2) x = Some y if m2 x = Some y for x y
        using assms that by auto
      ultimately show ?thesis
        unfolding ran-def by blast
    qed
qed

lemma finite-ran:
  assumes finite (dom p)
  shows finite (ran p)
proof
  have ran p = (λx. the (p x)) ' dom p
    unfolding ran-def by force
  from this (finite (dom p)) show ?thesis by auto
qed

lemma ran-distinct:
  assumes dist: distinct (map fst al)
  shows ran (map-of al) = snd ' set al
using assms
proof (induct al)
  case Nil
  then show ?case by simp
next
  case (Cons kv al)
  then have ran (map-of al) = snd ' set al by simp
  moreover from Cons.prems have map-of al (fst kv) = None
    by (simp add: map-of-eq-None-iff)
  ultimately show ?case by (simp only: map-of.simps ran-map-upd) simp
qed

lemma ran-map-of-zip:
assumes length xs = length ys distinct xs
shows ran (map-of (zip xs ys)) = set ys
using assms by (simp add: ran-distinct set-map[symmetric])

lemma ran-map-option: ran (λx. map-option f (m x)) = f ' ran m
by (auto simp add: ran-def)

69.11 map-le
lemma map-le-empty [simp]: empty ⊆ₘ g
by (simp add: map-le-def)

lemma upd-None-map-le [simp]: f(x := None) ⊆ₘ f
by (force simp add: map-le-def)

lemma map-le-upd [simp]: f ⊆ₘ g ==> f(a := b) ⊆ₘ g(a := b)
by (fastforce simp add: map-le-def)

lemma map-le-upd-le [simp]: m1 ⊆ₘ m2 ==> m1(x := None) ⊆ₘ m2(x \mapsto y)
by (force simp add: map-le-def)

lemma map-le-upds [simp]: f ⊆ₘ g ==> f(as \mapsto bs) ⊆ₘ g(as \mapsto bs)
apply (induct as arbitrary: f g bs)
apply simp
apply (case-tac bs)
apply auto
done

lemma map-le-implies-dom-le: (f ⊆ₘ g) ==> (dom f ⊆ dom g)
by (fastforce simp add: map-le-def dom-def)

lemma map-le-refl [simp]: f ⊆ₘ f
by (simp add: map-le-def)

lemma map-le-trans [trans]: [ m1 ⊆ₘ m2; m2 ⊆ₘ m3 ] ==> m1 ⊆ₘ m3
by (auto simp add: map-le-def dom-def)

lemma map-le-antisym: [ f ⊆ₘ g; g ⊆ₘ f ] ==> f = g
unfolding map-le-def
apply (rule ext)
apply (case-tac x ∈ dom f, simp)
apply (case-tac x ∈ dom g, simp, fastforce)
done

lemma map-le-map-add [simp]: f ⊆ₘ g ++ f
by (fastforce simp: map-le-def)
lemma map-le-iff-map-add-commute: \( f \subseteq_m f ++ g \iff f ++ g = g ++ f \)
by (fastforce simp: map-add-def map-le-def fun-eq-iff split: option.splits)

lemma map-add-le-mapE: \( f ++ g \subseteq_m h \implies g \subseteq_m h \)
by (fastforce simp: map-le-def map-add-def dom-def)

lemma map-add-le-mapI: \( \{ f \subseteq_m h \mid g \subseteq_m h \} \implies f ++ g \subseteq_m h \)
by (auto simp: map-le-def map-add-def dom-def split: option.splits)

lemma map-add-subsumed1: \( f \subseteq_m g \implies f ++ g = g \)
by (simp add: map-add-le-mapI map-le-antisym)

lemma map-add-subsumed2: \( f \subseteq_m g \implies g ++ f = g \)
by (metis map-add-subsumed1 map-le-iff-map-add-commute)

lemma dom-eq-singleton-conv: \( \text{dom } f = \{ x \} \iff (\exists v. f = \{ x \mapsto v \}) \)
(is ?lhs \iff ?rhs)
proof
  assume ?rhs
  then show ?lhs by (auto split: if-split-asm)
next
  assume ?lhs
  then obtain v where v: \( f x = \text{Some } v \) by auto
  show ?rhs
  proof
    show \( f = \{ x \mapsto v \} \)
    proof (rule map-le-antisym)
      show \( \{ x \mapsto v \} \subseteq_m f \)
      using v by (auto simp add: map-le-def)
      show \( f \subseteq_m \{ x \mapsto v \} \)
      using \( \text{dom } f = \{ x \} \) \( f x = \text{Some } v \) by (auto simp add: map-le-def)
      qed
    qed
    qed
    qed

lemma map-add-eq-empty-iff[simp]:
\( f ++ g = \text{empty} \iff f = \text{empty} \land g = \text{empty} \)
by (metis map-add-None)

lemma empty-eq-map-add-iff[simp]:
\( \text{empty} = f ++ g \iff f = \text{empty} \land g = \text{empty} \)
by (subst map-add-eq-empty-iff[symmetric])(rule eq-commute)

69.12 Various

lemma set-map-of-compr:
  assumes distinct: \( \text{distinct } (\text{map } \text{fst } x z) \)
  shows \( \text{set } x z = \{ (k, v). \text{map-of } x z k = \text{Some } v \} \)
  using assms
proof (induct xs)
case Nil then show ?case by simp
next
case (Cons x xs)
obtain k v where x = (k, v) by (cases x) blast
with Cons.prems have k \notin dom (map-of xs)
  by (simp add: dom-map-of-conv-image-fst)
then have \*: insert (k, v) \{(k, v). map-of xs k = Some v\} =
  \{(k', v'). (map-of xs(k \mapsto v)) k' = Some v'\}
  by (auto split: if-splits)
from Cons have set xs = \{(k, v). map-of xs k = Some v\} by simp
with \* (x = (k, v)) show ?case by simp
qed

lemma eq-key-imp-eq-value:
v1 = v2
if distinct (map fst xs) (k, v1) \in set xs (k, v2) \in set xs
proof –
from that have inj-on fst (set xs)
  by (simp add: distinct-map)
moreover have \{ (k, v1) = \{ (k, v2) \}
  by simp
ultimately have (k, v1) = (k, v2)
  by (rule inj-onD) (fact that)+
then show \?thesis
  by simp
qed

lemma map-of-inject-set:
assumes distinct: distinct (map fst xs) distinct (map fst ys)
show set xs = set ys \longleftrightarrow \{ (k, v). map-of xs k = Some v\}
  \longleftrightarrow \{ (k, v). map-of ys k = Some v\}
proof
assume \?lhs
moreover from (distinct (map fst xs)): have set xs = \{(k, v). map-of xs k = Some v\}
  by (rule set-map-of-compr)
moreover from (distinct (map fst ys)): have set ys = \{(k, v). map-of ys k = Some v\}
  by (rule set-map-of-compr)
ultimately show \?rhs by simp
next
assume \?rhs show \?lhs
proof
fix k
show map-of xs k = map-of ys k
  proof (cases map-of xs k)
  case None
  with \(?rhs\) have map-of ys k = None
by (simp add: map-of-eq-None iff)
with None show ?thesis by simp
next
case (Some v)
with distinct (?rhs) have map-of ys k = Some v
  by simp
with Some show ?thesis by simp
qed
qed

hide-const (open) Map.empty

end

70 Finite types as explicit enumerations

theory Enum
imports Map Groups-List
begin

70.1 Class enum

class enum =
  fixes enum :: 'a list
  fixes enum-all :: ('a ⇒ bool) ⇒ bool
  fixes enum-ex :: ('a ⇒ bool) ⇒ bool
  assumes UNIV-enum: UNIV = set enum
  and enum-distinct: distinct enum
  assumes enum-all-UNIV: enum-all P ←→ Ball UNIV P
  assumes enum-ex-UNIV: enum-ex P ←→ Bex UNIV P
  — tailored towards simple instantiation
begin

subclass finite proof
qed (simp add: UNIV-enum)

lemma enum-UNIV:
  set enum = UNIV
by (simp only: UNIV-enum)

lemma in-enum: x ∈ set enum
by (simp add: enum-UNIV)

lemma enum-eq-I:
  assumes ∀x. x ∈ set xs
  shows set enum = set xs
proof –
  from assms UNIV-eq-I have UNIV = set xs by auto
with enum-UNIV show ?thesis by simp
qed

lemma card-UNIV-length-enum:
  card (UNIV :: 'a set) = length enum
  by (simp add: UNIV-enum distinct-card enum-distinct)

lemma enum-all [simp]:
  enum-all = HOL.All
  by (simp add: fun-eq-iff enum-all-UNIV)

lemma enum-ex [simp]:
  enum-ex = HOL.Ex
  by (simp add: fun-eq-iff enum-ex-UNIV)

end

70.2 Implementations using enum

70.2.1 Unbounded operations and quantifiers

lemma Collect-code [code]:
  Collect P = set (filter P enum)
  by (simp add: enum-UNIV)

lemma vimage-code [code]:
  f −' B = set (filter (λx. f x ∈ B) enum-class.enum)
  unfolding vimage-def Collect-code ..

definition card-UNIV :: 'a itself ⇒ nat
  where
  [code del]: card-UNIV TYPE('a) = card (UNIV :: 'a set)

lemma [code]:
  card-UNIV TYPE('a :: enum) = card (set (Enum.enum :: 'a list))
  by (simp only: card-UNIV-def enum-UNIV)

lemma all-code [code]: (∀x. P x) ↔ enum-all P
  by simp

lemma exists-code [code]: (∃x. P x) ↔ enum-ex P
  by simp

lemma exists1-code [code]: (∃!x. P x) ↔ list-ex1 P enum
  by (auto simp add: list-ex1-iff enum-UNIV)

70.2.2 An executable choice operator

definition
  [code del]: enum-the = The
lemma [code]:

The \( P = \) (case filter \( P \) \( \text{enum} \) of \[ x \]\( \Rightarrow \) \( x \) | \( - \) \( \Rightarrow \) \( \text{enum-the} \) \( P \) )

proof –

\{ 
  fix \( a \)
  assume filter-enum: filter \( P \) \( \text{enum} \) = \[ a \]
  have \( \text{The} \) \( P \) = \( a \)
    proof (rule the-equality)
      fix \( x \)
      assume \( P \) \( x \)
      show \( x = a \)
        proof (rule ccontr)
          assume \( x \neq a \)
          from filter-enum obtain \( \text{us} \) \( \text{vs} \) 
            where \( \text{enum-eq} \): \( \text{enum} = \text{us} @ [a] @ \text{vs} \)
            and \( \forall \) \( x \in \text{set} \) \( \text{us} \). \( \neg \) \( P \) \( x \)
            and \( \forall \) \( x \in \text{set} \) \( \text{vs} \). \( \neg \) \( P \) \( x \)
            and \( P \) \( a \)
          by (auto simp add: filter-eq-Cons-iff) (simp only: filter-empty-conv[symmetric])
          with \( \langle P \rangle \) in-enum[of \( x \), unfolded enum-eq] \( \langle x \neq a \rangle \)
          show False by auto
        qed
    qed
  next
  from filter-enum show \( P \) \( a \) by (auto simp add: filter-eq-Cons-iff)
  qed
\}
from this show \( \text{?thesis} \)
  unfolding enum-the-def by (auto split: list.split)
qed

declare \( \langle \text{code abort: enum-the} \rangle \)

code-printing
  constant \( \text{enum-the} \to \) (Eval) \( \langle \text{fn }'\to'\Rightarrow\rangle \) raise Match

70.2.3 Equality and order on functions

instantiation \( \text{fun} :: \) \( \langle \text{enum}, \text{equal} \rangle \) equal
begin

definition
  \( \text{HOL.equal} \) \( f \) \( g \) \( \leftarrow\rightarrow \) (\( \forall \) \( x \in \text{set} \) \( \text{enum} \). \( f \) \( x \) \( = \) \( g \) \( x \) )

instance proof
  qed (simp-all add: equal-fun-def fun-eq-iff enum-UNIV)
end

lemma [code]:
HOL.equal \( f \leftrightarrow \text{enum-all} \ (\forall x. f x = g x) \) 
by (auto simp add: equal fun-eq-iff)

**lemma** [code nbe]:

**HOL.equal** \((f :: - \Rightarrow -) \leftrightarrow True\) 
by (fact equal-refl)

**lemma** order-fun [code]:

fixes \( f, g :: 'a ::\text{enum} \Rightarrow 'b ::\text{order} \)
shows \( f \leq g \leftrightarrow \text{enum-all} \ (\lambda x. f x \leq g x) \) 
and \( f < g \leftrightarrow f \leq g \wedge \text{enum-ex} \ (\lambda x. f x \neq g x) \) 
by (simp-all add: fun-eq-iff le-fun-def order-less-le)

70.2.4 Operations on relations

**lemma** [code]:

\( \text{Id} = \text{image} \ (\lambda x. (x, x)) \) \ (set \text{Enum.enum}) 
by (auto intro: imageI in-enm)

**lemma** trancl-unfold [code]:

\( \text{tranclp} \ r \ a \ b \leftrightarrow (a, b) \in \text{trancl} \ \{(x, y). r x y\} \) 
by (simp add: trancl-def)

**lemma** rtranclp-rtrancl-eq [code]:

\( \text{rtranclp} \ r \ x \ y \leftrightarrow (x, y) \in \text{rtrancl} \ \{(x, y). r x y\} \) 
by (simp add: rtrancl-def)

**lemma** max-ext-eq [code]:

\( \text{max-ext} \ R = \{(X, Y). \text{finite} X \wedge \text{finite} Y \wedge Y \neq \{\} \wedge (\forall x. x \in X \longrightarrow (\exists xa \in Y. (x, xa) \in R))\} \) 
by (auto simp add: max-ext-simps)

**lemma** max-extp-eq [code]:

\( \text{max-extp} \ r \ x \ y \leftrightarrow (x, y) \in \text{max-ext} \ \{(x, y). r x y\} \) 
by (simp add: max-ext-def)

**lemma** mlex-eq [code]:

\( f <^{\text{mlex}} R = \{(x, y). f x < f y \lor (f x \leq f y \wedge (x, y) \in R)\} \) 
by (auto simp add: mlex-prod-def)

70.2.5 Bounded accessible part

**primrec** bacc :: \('a \times 'a\) set \Rightarrow \text{nat} \Rightarrow 'a set
where
\( \text{bacc} \ r \ 0 = \{x. \forall y. (y, x) \notin r\} \) 
\| \( \text{bacc} \ r \ (\text{Suc} \ n) = (\text{bacc} \ r \ n \cup \{x. \forall y. (y, x) \in r \longrightarrow y \in \text{bacc} \ r \ n\}) \) 

**lemma** bacc-subseteq-acc:

\( \text{bacc} \ r \ n \subseteq \text{Wellfounded.acc} \ r \) 
by (induct n) (auto intro: acc.intros)
THEORY "Enum"

lemma bacc-mono:
  \( n \leq m \implies \text{bacc } r n \subseteq \text{bacc } r m \)
by (induct rule: dec-induct) auto

lemma bacc-upper-bound:
\( \text{bacc } (r :: ('a \times 'a) set) \ (\text{card } (\text{UNIV} :: 'a::finite set)) = (\bigcup n. \text{bacc } r n) \)
proof –
  have mono (bacc r) unfolding mono-def by (simp add: bacc-mono)
  moreover have \( \forall n. \text{bacc } r n = \text{bacc } r (\text{Suc } n) \implies \text{bacc } r (\text{Suc } n) = \text{bacc } r (\text{Suc } n) \) by auto
  moreover have finite (range (bacc r)) by auto
  ultimately show ?thesis by (intro finite-mono-strict-prefix-implies-finite-fixpoint)
  (auto intro: finite-mono-remains-stable-implies-strict-prefix)
qed

lemma acc-subseteq-bacc:
  assumes finite r
shows Wellfounded. acc r \subseteq (\bigcup n. \text{bacc } r n)
proof
fix x
assume x \in Wellfounded. acc r
then have \( \exists n. x \in \text{bacc } r n \)
proof (induct x arbitrary: rule: acc.induct)
  case (accI x)
  then have \( \forall y. \exists n. (y, x) \in r \implies y \in \text{bacc } r n \) by simp
  from choice[OF this] obtain n where n: \( \forall y. (y, x) \in r \implies y \in \text{bacc } r (n y) \)
  ..
  obtain n where \( \forall y. (y, x) \in r \implies y \in \text{bacc } r n \)
proof
  fix y assume y: \( (y, x) \in r \)
  with n have y \in bacc r (n y) by auto
  moreover have n y \leq Max ((\lambda (y, x). n y) ' r) using y (finite r) by (auto intro!: Max-ge)
  note bacc-mono[OF this, of r]
  ultimately show y \in bacc r (Max ((\lambda (y, x). n y) ' r)) by auto
  qed
  then show ?case
  by (auto simp add: Let-def intro!: exI[of - Suc n])
  qed
  then show x \in (\bigcup n. \text{bacc } r n) by auto
  qed

lemma acc-bacc-eq:
  fixes A :: ('a :: finite \times 'a) set
  assumes finite A
shows Wellfounded. acc A = bacc A (card (UNIV :: 'a set))
using assms by (metis acc-subseteq-bacc bacc-subseteq-acc bacc-upper-bound order-eq-iff)
lemma [code]:
  fixes xs :: ('a::finite × 'a) list
  shows Wellfounded.acc (set xs) = bacc (set xs) (card-UNIV TYPE('a))
  by (simp add: card-UNIV-def acc-bacc-eq)

70.3 Default instances for enum

lemma map-of-zip-enum-is-Some:
  assumes length ys = length (enum :: 'a::enum list)
  shows ∃ y. map-of (zip (enum :: 'a::enum list) ys) x = Some y
  proof
    from assms have x ∈ set (enum :: 'a::enum list) ⟷
      (∃ y. map-of (zip (enum :: 'a::enum list) ys) x = Some y)
    by (auto intro: map-of-zip-is-Some)
    then show ?thesis using enum-UNIV by auto
  qed

lemma map-of-zip-enum-inject:
  fixes xs ys :: 'b::enum list
  assumes length: length xs = length (enum :: 'a::enum list)
  length ys = length (enum :: 'a::enum list)
  and map-of: the ◦ map-of (zip (enum :: 'a::enum list) xs) = the ◦ map-of (zip
  (enum :: 'a::enum list) ys)
  shows xs = ys
  proof
    have map-of (zip (enum :: 'a list) xs) = map-of (zip (enum :: 'a list) ys)
      proof
        fix x :: 'a
        from length map-of-zip-enum-is-Some obtain y1 y2
        where map-of (zip (enum :: 'a list) xs) x = Some y1
        and map-of (zip (enum :: 'a list) ys) x = Some y2 by blast
        moreover from map-of
        have the (map-of (zip (enum :: 'a::enum list) xs) x) = the (map-of (zip
        (enum :: 'a::enum list) ys) x)
          by (auto dest: fun-cong)
        ultimately show map-of (zip (enum :: 'a::enum list) xs) x = map-of (zip
        (enum :: 'a::enum list) ys) x
          by simp
      qed
    with length enum-distinct show xs = ys by (rule map-of-zip-inject)
  qed

definition all-n-lists :: (('a :: enum) list ⇒ bool) ⇒ nat ⇒ bool
  where
    all-n-lists P n ⟷ (∀ xs ∈ set (List.n-lists n enum). P xs)

lemma [code]:
  all-n-lists P n ⟷ (if n = 0 then P [] else enum-all (%x. all-n-lists (%xs. P (x
### THEORY “Enum”

```plaintext
# xs)) (n - 1)))
unfolding all-n-lists-def enum-all
by (cases n) (auto simp add: enum-UNIV)

definition ex-n-lists :: (('a :: enum) list ⇒ bool) ⇒ nat ⇒ bool
where
ex-n-lists P n ←→ (∃xs ∈ set (List.n-lists n enum). P xs)

lemma [code]:
ex-n-lists P n ←→ (if n = 0 then P [] else enum-ex (%xs. ex-n-lists (%xs. P (xs # xs)) (n - 1)))
unfolding ex-n-lists-def enum-ex
by (cases n) (auto simp add: enum-UNIV)

instantiation fun :: (enum, enum) enum
begin

definition enum = map (λys. the ◦ map-of (zip (enum::'a list) ys)) (List.n-lists (length (enum::'a::enum list))) enum

definition enum-all P = all-n-lists (λbs. P (the ◦ map-of (zip enum bs))) (length (enum :: 'a list))

definition enum-ex P = ex-n-lists (λbs. P (the ◦ map-of (zip enum bs))) (length (enum :: 'a list))

instance proof
show UNIV = set (enum :: ('a ⇒ 'b) list)
proof (rule UNIV-eq-I)
fix f :: 'a ⇒ 'b
have f = the ◦ map-of (zip (enum :: 'a::enum list)) (map f enum)
by (auto simp add: map-of-zip-map fun-eq-iff intro: in-enum)
then show f ∈ set enum
by (auto simp add: enum-fun-def set-n-lists intro: in-enum)
qed

next
from map-of-zip-enum-inject
show distinct (enum :: ('a ⇒ 'b) list)
by (auto intro!: inj-onI simp add: enum-fun-def
distinct-map distinct-n-lists enum-distinct set-n-lists)

next
fix P
show enum-all (P :: ('a ⇒ 'b) ⇒ bool) = Ball UNIV P
proof
assume enum-all P
show Ball UNIV P
```

```plaintext
```
proof
  fix f :: 'a ⇒ 'b
  have f: f = the ◦ map-of (zip enum :: 'a::enum list) (map f enum))
    by (auto simp add: map-of-zip-map fan-eq-iff intro: in-num)
  from ⟨enum-all P⟩ have P (the ◦ map-of (zip enum (map f enum)))
  unfolding enum-all-fun-def all-n-lists-def
  apply (simp add: set-n-lists)
  apply (erule-tac x=map f enum in allE)
  apply (auto intro!: in-num)
  done
  from this f show P f by auto
qed
next
  assume Ball UNIV P
  from this show enum-all P
    unfolding enum-all-fun-def all-n-lists-def
    by auto
qed
next
  fix P
  show enum-ex (P :: ('a ⇒ 'b) ⇒ bool) = Bex UNIV P
  proof
    assume enum-ex P
    from this show Bex UNIV P
      unfolding enum-ex-fun-def ex-n-lists-def by auto
  next
    assume Bex UNIV P
    from this obtain f where P f ..
    have f: f = the ◦ map-of (zip enum :: 'a::enum list) (map f enum))
      by (auto simp add: map-of-zip-map fan-eq-iff intro: in-num)
    from ⟨P f⟩ this have P (the ◦ map-of (zip enum :: 'a::enum list) (map f enum)))
      by auto
    from this show enum-ex P
      unfolding enum-ex-fun-def ex-n-lists-def
      apply (auto simp add: set-n-lists)
      apply (rule-tac x=map f enum in exI)
      apply (auto intro!: in-num)
      done
    qed
  qed
end

lemma enum-fun-code [code]: enum = (let enum-a = (enum :: 'a::{enum, equal} list)
  in map (λys. the ◦ map-of (zip enum-a ys)) (List.n-lists (length enum-a) enum))
  by (simp add: enum-fun-def Let-def)

lemma enum-all-fun-code [code]:
enum-all $P = \text{(let } \text{enum-a = (enum :: }'a::\text{enum, equal} \text{ list)}$
\text{ in all-n-lists (λbs. } P \text{ (the ◦ map-of (zip enum-a bs))) (length enum-a))$
\text{ by (simp only: enum-all-fun-def Let-def)}$

\text{lemma enum-ex-fun-code [code]:}
enum-ex $P = \text{(let } \text{enum-a = (enum :: }'a::\text{enum, equal} \text{ list)}$
\text{ in ex-n-lists (λbs. } P \text{ (the ◦ map-of (zip enum-a bs))) (length enum-a))$
\text{ by (simp only: enum-ex-fun-def Let-def)}$

\text{instantiation set :: (enum) enum}
\text{begin}
\text{definition enum = map set (subseqs enum)}$
\text{definition enum-all $P \leftrightarrow (\forall A \in \text{set enum}. P (A::'a set))$
\text{definition enum-ex $P \leftrightarrow (\exists A \in \text{set enum}. P (A::'a set))$
\text{instance proof qed (simp-all add: enum-set-def enum-all-set-def enum-ex-set-def subseqs-powset distinct-set-subseqs enum-distinct enum-UNIV)}$
\text{end}

\text{instantiation unit :: enum}
\text{begin}
\text{definition enum = [()]$
\text{definition enum-all $P = P ()$
\text{definition enum-ex $P = P ()$
\text{instance proof qed (auto simp add: enum-unit-def enum-all-unit-def enum-ex-unit-def)}$
\text{end}

\text{instantiation bool :: enum}
\text{begin}
\text{definition}
enum = [False, True]

definition
enum-all P ↔ P False ∧ P True

definition
enum-ex P ↔ P False ∨ P True

instance proof
qed (simp-all only: enum-bool-def enum-all-bool-def enum-ex-bool-def UNIV-bool, simp-all)

end

instantiation prod :: (enum, enum) enum begin

definition
enum = List.product enum enum

definition
enum-all P = enum-all (%x. enum-all (%y. P (x, y)))

definition
enum-ex P = enum-ex (%x. enum-ex (%y. P (x, y)))

instance
by standard
  (simp-all add: enum-prod-def distinct-product
   enum-UNIV enum-distinct enum-all-prod-def enum-ex-prod-def)

end

instantiation sum :: (enum, enum) enum begin

definition
enum = map Inl enum @ map Inr enum

definition
enum-all P ↔ enum-all (lx. P (Inl x)) ∧ enum-all (lx. P (Inr x))

definition
enum-ex P ↔ enum-ex (lx. P (Inl x)) ∨ enum-ex (lx. P (Inr x))

instance proof
qed (simp-all only: enum-sum-def enum-all-sum-def enum-ex-sum-def UNIV-sum, auto simp add: enum-UNIV distinct-map enum-distinct)
THEORY "Enum"

end

instantiation option :: (enum) enum
begin

definition
  enum = None ≠ map Some enum

definition
  enum-all P ←→ P None ∧ enum-all (λx. P (Some x))

definition
  enum-ex P ←→ P None ∨ enum-ex (λx. P (Some x))

instance proof
qed (simp-all only: enum-option-def enum-all-option-def enum-ex-option-def UNIV-option-conv,
auto simp add: distinct-map enum-UNIV enum-distinct)

end

70.4 Small finite types

We define small finite types for use in Quickcheck

datatype (plugins only: code quickcheck extraction) finite-1 =
  a₁

notation (output) a₁ (a₁)

lemma UNIV-finite-1:
  UNIV = {a₁}
by (auto intro: finite-1.exhaust)

instantiation finite-1 :: enum
begin

definition
  enum = [a₁]

definition
  enum-all P = P a₁

definition
  enum-ex P = P a₁

instance proof
qed (simp-all only: enum-finite-1-def enum-all-finite-1-def enum-ex-finite-1-def UNIV-finite-1,
simp-all)
Theory "Enum"

end

instantiation finite-1 :: linorder
begin

definition less-finite-1 :: finite-1 ⇒ finite-1 ⇒ bool
where
  \( x < (y :: \text{finite-1}) \) \( \mapsto \) False

definition less-eq-finite-1 :: finite-1 ⇒ finite-1 ⇒ bool
where
  \( x \leq (y :: \text{finite-1}) \) \( \mapsto \) True

instance
apply (intro-classes)
apply (auto simp add: less-finite-1-def less-eq-finite-1-def)
apply (metis (full-types) finite-1.exhaust)
done

end

instance finite-1 :: \{dense-linorder, wellorder\}
by intro-classes (simp-all add: less-finite-1-def)

instantiation finite-1 :: complete-lattice
begin

definition [simp]: Inf = (\( \lambda \cdot \ a_1 \) )
definition [simp]: Sup = (\( \lambda \cdot \ a_1 \) )
definition [simp]: bot = a_1
definition [simp]: top = a_1
definition [simp]: inf = (\( \lambda \cdot \cdot \ a_1 \) )
definition [simp]: sup = (\( \lambda \cdot \cdot \ a_1 \) )

instance by intro-classes(simp-all add: less-eq-finite-1-def)
end

instance finite-1 :: complete-distrib-lattice
  by standard simp-all

instance finite-1 :: complete-linorder ..

lemma finite-1-eq: \( x = a_1 \)
by(cases x) simp

simproc-setup finite-1-eq (x::finite-1) =:
  fn _ => fn _ => fn ct =>
  (case Thm.term_of ct of
   Const (const-name\( a_1 \), _) => NONE
instantiation finite-1 :: complete-boolean-algebra
begin
  definition [simp]: (-) = (λ - . a₁)
  definition [simp]: uminus = (λ - . a₁)
instance by intro-classes simp-all
end

instantiation finite-2 :: enum
begin
  definition enum = [a₁, a₂]
definition enum-all P ↔ P a₁ ∧ P a₂
definition
enum-ex \( P \) \( \longleftrightarrow P \; a_1 \lor P \; a_2 \)

instance proof
qed (simp-all only: enum-finite-2-def enum-all-finite-2-def enum-ex-finite-2-def UNIV-finite-2, simp-all)
end

instantiation finite-2 :: linorder
begin

definition less-finite-2 :: finite-2 \( \Rightarrow \) finite-2 \( \Rightarrow \) bool
where
\( x < y \) \( \longleftrightarrow \) \( x = a_1 \land y = a_2 \)

definition less-eq-finite-2 :: finite-2 \( \Rightarrow \) finite-2 \( \Rightarrow \) bool
where
\( x \leq y \) \( \longleftrightarrow \) \( x = y \lor x < (y \; :: \; finite-2) \)

instance
apply (intro-classes)
apply (auto simp add: less-finite-2-def less-eq-finite-2-def)
apply (metis finite-2.nchotomy)+
done
end

instance finite-2 :: wellorder
by (rule wf-wellorderI)(simp add: less-finite-2-def, intro-classes)

instantiation finite-2 :: complete-lattice
begin

definition \( \sqcap \) \( A \) \( = \) (if \( a_1 \in A \) then \( a_1 \) else \( a_2 \))

definition \( \sqcup \) \( A \) \( = \) (if \( a_2 \in A \) then \( a_2 \) else \( a_1 \))

definition [simp]: bot = \( a_1 \)

definition [simp]: top = \( a_2 \)

definition \( x \sqcap y \) \( = \) (if \( x = a_1 \lor y = a_1 \) then \( a_1 \) else \( a_2 \))

definition \( x \sqcup y \) \( = \) (if \( x = a_2 \lor y = a_2 \) then \( a_2 \) else \( a_1 \))

lemma neq-finite-2-a1-iff [simp]: \( x \neq a_1 \longleftrightarrow x = a_2 \)
by (cases \( x \)) simp-all

lemma neq-finite-2-a1-iff' [simp]: \( a_1 \neq x \longleftrightarrow x = a_2 \)
by (cases \( x \)) simp-all

lemma neq-finite-2-a2-iff [simp]: \( x \neq a_2 \longleftrightarrow x = a_1 \)
by (cases $x$) simp-all

lemma neq-finite-2-a2-iff [simp]: $a_2 \neq x \iff x = a_1$
by (cases $x$) simp-all

instance
proof
  fix $x :: \text{finite-2}$ and $A$
  assume $x \in A$
  then show $\bigwedge A \leq x \leq \bigvee A$
    by (cases $x$; auto simp add: less-eq-finite-2-def less-finite-2-def Inf-finite-2-def Sup-finite-2-def)+
qed (auto simp add: less-eq-finite-2-def less-finite-2-def inf-finite-2-def sup-finite-2-def Inf-finite-2-def Sup-finite-2-def)
end

instance finite-2 :: complete-linorder ..

instance finite-2 :: complete-distrib-lattice ..

instantiation finite-2 :: \{field, idom-abs-sgn, idom-modulo\} begin
definition $0 = a_1$
definition $1 = a_2$
definition $x + y = (\text{case } (x, y) \text{ of } (a_1, a_1) \Rightarrow a_1 | (a_2, a_2) \Rightarrow a_1 | - \Rightarrow a_2)$
definition $\text{uminus} = (\lambda x :: \text{finite-2}. x)$
definition $(-) = ((+) :: \text{finite-2} \Rightarrow -)$
definition $x * y = (\text{case } (x, y) \text{ of } (a_2, a_2) \Rightarrow a_2 | - \Rightarrow a_1)$
definition $\text{inverse} = (\lambda x :: \text{finite-2}. x)$
definition $\text{divide} = ((*) :: \text{finite-2} \Rightarrow -)$
definition $x \mod y = (\text{case } (x, y) \text{ of } (a_2, a_1) \Rightarrow a_2 | - \Rightarrow a_1)$
definition $\text{abs} = (\lambda x :: \text{finite-2}. x)$
definition $\text{sgn} = (\lambda x :: \text{finite-2}. x)$
instance
  by standard
  (subproofs
   simp-all add: plus-finite-2-def uminus-finite-2-def minus-finite-2-def
times-finite-2-def
inverse-finite-2-def divide-finite-2-def modulo-finite-2-def
abs-finite-2-def sgn-finite-2-def
split: finite-2.splits)
end

lemma two-finite-2 [simp]:
  $2 = a_1$
by (simp add: numeral.simps plus-finite-2-def)

lemma dvd-finite-2-unfold:
  $x \text{ dvd } y \iff x = a_2 \lor y = a_1$
by (auto simp add: dvd-def times-finite-2-def split: finite-2.splits)
instanitiation finite-2 :: {normalization-semidom, unique-euclidean-semiring} begin
definition [simp]: normalize = (id :: finite-2 ⇒ _) 
definition [simp]: unit-factor = (id :: finite-2 ⇒ -) 
definition [simp]: euclidean-size x = (case x of a₁ ⇒ 0 | a₂ ⇒ 1) 
definition [simp]: division-segment (x :: finite-2) = 1 
instance 
by standard 
(subproofs 
(auto simp add: divide-finite-2-def times-finite-2-def dvd-finite-2-unfold 
split: finite-2.splits;)
end

hide-const (open) a₁ a₂

datatype (plugins only: code quickcheck extraction) finite-3 =
a₁ | a₂ | a₃
notation (output) a₁ (a₁) 
notation (output) a₂ (a₂) 
notation (output) a₃ (a₃)

lemma UNIV-finite-3; 
UNIV = {a₁, a₂, a₃} 
by (auto intro: finite-3.exhaust)

instanitiation finite-3 :: enum
begin

definition 
enum = [a₁, a₂, a₃]

definition 
enum-all P ←→ P a₁ ∧ P a₂ ∧ P a₃

definition 
enum-ex P ←→ P a₁ ∨ P a₂ ∨ P a₃

instance proof
qed (simp-all only: enum-finite-3-def enum-all-finite-3-def enum-ex-finite-3-def UNIV-finite-3, simp-all)

end

lemma finite-3-not-eq-unfold:
x ≠ a₁ ←→ x ∈ {a₂, a₃} 
x ≠ a₂ ←→ x ∈ {a₁, a₃} 
x ≠ a₃ ←→ x ∈ {a₁, a₂}
by (cases x; simp)+

instantiation finite-3 :: linorder
begin

definition less-finite-3 :: finite-3 ⇒ finite-3 ⇒ bool
where
  x < y = (case x of a1 ⇒ y ≠ a1 | a2 ⇒ y = a3 | a4 ⇒ False)

definition less-eq-finite-3 :: finite-3 ⇒ finite-3 ⇒ bool
where
  x ≤ y ←→ x = y ∨ x < (y :: finite-3)

instance proof (intro-classes)
qed (auto simp add: less-finite-3-def less-eq-finite-3-def split: finite-3.split-asm)
end

instance finite-3 :: wellorder
proof (rule wf-wellorderI)
  have inv-image less-than (case-finite-3 0 1 2) = {(x, y). x < y}
    by (auto simp add: less-finite-3-def split: finite-3.splits)
  from this[symmetric] show wf ...
  by simp
qed intro-classes

class finite-lattice = finite + lattice + Inf + Sup + bot + top +
  assumes Inf-finite-empty: Inf { } = Sup UNIV
  assumes Inf-finite-insert: Inf (insert a A) = a ∩ Inf A
  assumes Sup-finite-empty: Sup { } = Inf UNIV
  assumes Sup-finite-insert: Sup (insert a A) = a ⊔ Sup A
  assumes bot-finite-def: bot = Inf UNIV
  assumes top-finite-def: top = Sup UNIV
begin

subclass complete-lattice
proof
  fix x A
  show x ∈ A ⇒ ⨍ A ≤ x
    by (metis Set.set-insert abel-semigroup.commute local.Inf-finite-insert local.inf.abel-semigroup-axioms
        local.inf.left-idem local.inf.orderI)
  show x ∈ A ⇒ x ≤ ⨆ A
    by (metis Set.set-insert insert-absorb2 local.Sup-finite-insert local.sup.absorb-iff2)
next
  fix A z
  have ⨁ UNIV = z ⊔ ⨁ UNIV
    by (subst Sup-finite-insert [symmetric], simp add: insert-UNIV)
  from this have [simp]: z ≤ ⨁ UNIV
  using local.le-iff-sup by auto
  have (∀ x. x ∈ A → z ≤ x) → z ≤ ⨋ A
class finite-distrib-lattice = finite-lattice + distrib-lattice

lemma finite-inf-Sup: a ∩ (Sup A) = Sup {a ∩ b | b ∈ A}
proof (rule finite-induct [of A λ A . (∀ x. x ∈ A ⇒ z ≤ x) ⇒ z ≤ ⋃ A], simp-all add: inf-sup-distrib1)
fix x::'a
fix F
assume x ∉ F
assume [simp]: a ∩ ∪ F = ∪ {a ∩ b | b ∈ F}
have [simp]: insert (a ∩ x) {a ∩ b | b ∈ F} = {a ∩ b | b = x ∨ b ∈ F}
  by blast
have a ∩ (x ∪ ∪ F) = a ∩ x ∪ a ∩ ∪ F
  by (simp add: inf-sup-distrib1)
also have ... = a ∩ x ∪ {a ∩ b | b ∈ F}
  by simp
also have ... = ∪ {a ∩ b | b = x ∨ b ∈ F}
  by (unfold Sup-insert[THEN sym], simp)
finally show a ∩ (x ∪ ∪ F) = ∪ {a ∩ b | b = x ∨ b ∈ F}
  by simp
qed

lemma finite-Inf-Sup: ⋃ (Sup ' A) ≤ ⋃ (Inf ' {f ' A | f. ∀ Y∈A. f Y ∈ Y})
proof (rule finite-induct [of A λ A . ⋃ (Sup ' A) ≤ ⋃ (Inf ' {f ' A | f. ∀ Y∈A. f Y ∈ Y}), simp-all add: finite-UnionD])
fix x::'a set
fix F
assume x ∉ F
have \([\text{simp}]\): \(\bigcup x \cap b . b \in \text{Inf} \cdot \{f \cdot F \mid f \cdot (\forall Y \in F. f Y \in Y)\} = \{\bigcup x \cap \text{Inf} (f \cdot F) \mid f \cdot (\forall Y \in F. f Y \in Y)\}\)

by auto

define \(fa\) where \(fa = (\lambda (b::')a f Y . (if Y = x then b else f Y))\)

have \(\bigcup f b. \forall Y \in F. f Y \in Y \Longrightarrow b \in x \Longrightarrow \text{insert} b (f \cdot (F \cap \{Y. Y \neq x\})) = \text{insert} (fa \cdot b \cdot x) (fa \cdot f \cdot F) \cdot \text{fa} \cdot b \cdot f \cdot x \in x \land (\forall Y \in F. \text{fa} \cdot b \cdot f \cdot Y \in Y)\)

by (auto simp add: fa-def)

from this have \(\bigcup B. \forall f b. \forall Y \in F. f Y \in Y \Longrightarrow b \in x \Longrightarrow fa b f \cdot ((x) \cup F) \in \{\text{insert} f x \cdot (f \cdot F) \mid f \cdot f x \in x \land (\forall Y \in F. f Y \in Y)\}\)

by blast

have \([\text{simp}]\): \(\bigcup f b. \forall Y \in F. f Y \in Y \Longrightarrow b \in x \Longrightarrow b \cap (\bigcup x \in F. f x) \leq \bigcup (\text{Inf} \cdot \{f \cdot F \mid f \cdot (\forall Y \in F. f Y \in Y)\})\)

using \(B\) apply (rule SUP-upper2)

using \((x \notin F)\) apply (simp-all add: fa-def \text{Inf-union-distrib})

apply (simp add: image-mono \text{Inf-superset-mono} \text{inf.coboundedI2})

done

assume \(\bigcup (\text{Sup} \cdot F) \leq \bigcup (\text{Inf} \cdot \{f \cdot F \mid f \cdot (\forall Y \in F. f Y \in Y)\})\)

from this have \(\bigcup x \cap \bigcup (\text{Sup} \cdot F) \leq \bigcup x \cap \bigcup (\text{Inf} \cdot \{f \cdot F \mid f \cdot (\forall Y \in F. f Y \in Y)\})\)

using \text{inf.coboundedI2} by auto

also have \(\ldots = \text{Sup} \{\bigcup x \cap (\text{Inf} \cdot \{f \cdot F\}) \mid f \cdot (\forall Y \in F. f Y \in Y)\}\)

by (simp add: finite-inf-Sup)

also have \(\ldots = \text{Sup} \{\text{Inf} (f \cdot F) \cap b \cdot b \in x\} \mid f \cdot (\forall Y \in F. f Y \in Y)\}\)

by (subst inf-commute) (simp add: finite-inf-Sup)

also have \(\ldots \leq \bigcup (\text{Inf} \cdot \{\text{insert} f x \cdot (f \cdot F) \mid f \cdot f x \in x \land (\forall Y \in F. f Y \in Y)\})\)

apply (rule Sup-least, clarsimp)+

apply (subst inf-commute, simp)

done

finally show \(\bigcup x \cap \bigcup (\text{Sup} \cdot F) \leq \bigcup (\text{Inf} \cdot \{\text{insert} f x \cdot (f \cdot F) \mid f \cdot f x \in x \land (\forall Y \in F. f Y \in Y)\})\)

by simp

qed

subclass complete-distrib-lattice

by (standard, rule finite-Inf-Sup)
end

instantiation finite-3 :: finite-lattice

begin

definition \(\bigcap A = (if a_1 \in A then a_1 else if a_2 \in A then a_2 else a_3)\)

definition \(\bigcup A = (if a_3 \in A then a_3 else if a_2 \in A then a_2 else a_1)\)

definition \([\text{simp}]\): \(\text{bot} = a_1\)

definition \([\text{simp}]\): \(\text{top} = a_3\)

definition \([\text{simp}]\): \(\text{inf} = (\text{min} :: \text{finite-3} \Rightarrow -)\)
definition [simp]: sup = (max :: finite-3 ⇒ )

instance
proof
qed (auto simp add: Inf-finite-3-def Sup-finite-3-def max-def min-def less-eq-finite-3-def less-finite-3-def split: finite-3.split)
end

instance finite-3 :: complete-lattice ..

instance finite-3 :: finite-distrib-lattice
proof
qed (auto simp add: min-def max-def)

instance finite-3 :: complete-distrib-lattice ..

instance finite-3 :: complete-linorder ..

instantiation finite-3 :: {field, idom-abs-sgn, idom-modulo} begin
definition [simp]: 0 = a1
definition [simp]: 1 = a2

definition x + y = (case (x, y) of
  (a1, a1) ⇒ a1 | (a2, a3) ⇒ a1 | (a3, a2) ⇒ a1
  | (a1, a2) ⇒ a2 | (a2, a1) ⇒ a2 | (a3, a3) ⇒ a2
  | - ⇒ a3)
definition - x = (case x of a1 ⇒ a1 | a2 ⇒ a3 | a3 ⇒ a2)
definition x * y = (case (x, y) of (a2, a2) ⇒ a2 | (a3, a3) ⇒ a2 | (a2, a3) ⇒ a3 | (a3, a2) ⇒ a3 | - ⇒ a1)
definition inverse = (λx :: finite-3. x)
definition x div y = x * inverse (y :: finite-3)
definition x mod y = (case y of a1 ⇒ x | - ⇒ a1)
definition abs = (λx. case x of a3 ⇒ a2 | - ⇒ x)
definition sgn = (λx :: finite-3. x)

instance
by standard
(subproofs
  simp-all add: plus-finite-3-def uminus-finite-3-def minus-finite-3-def times-finite-3-def inverse-finite-3-def divide-finite-3-def modulo-finite-3-def abs-finite-3-def sgn-finite-3-def less-finite-3-def
split: finite-3.splits)
end

lemma two-finite-3 [simp]:
  2 = a3
by (simp add: numeral.simps plus-finite-3-def)
lemma dvd-finite-3-unfold:
\[ x \text{ dvd } y \iff x = a_2 \lor x = a_3 \lor y = a_1 \]
by (cases x) (auto simp add: dvd-def times-finite-3-def split: finite-3.splits)

instantiation finite-3 :: {normalization-semidom, unique-euclidean-semiring} begin

definition norm [simp]: normalize x = (case x of a3 ⇒ a2 | - ⇒ x)
definition unit-factor [simp]: unit-factor = (id :: finite-3 ⇒ -)
definition euclidean-size [simp]: euclidean-size x = (case x of a1 ⇒ 0 | - ⇒ 1)
definition division-segment [simp]: division-segment (x :: finite-3) = 1

instance proof
fix x :: finite-3
assume x ≠ 0
then show is-unit (unit-factor x)
  by (cases x) (simp-all add: dvd-finite-3-unfold)
qed

end

hide-const (open) a1 a2 a3

datatype (plugins only: code quickcheck extraction) finite-4 =
  a1 | a2 | a3 | a4

notation (output) a1 (a1)
notation (output) a2 (a2)
notation (output) a3 (a3)
notation (output) a4 (a4)

lemma UNIV-finite-4:
UNIV = {a1, a2, a3, a4}
  by (auto intro: finite-4.exhaust)

instantiation finite-4 :: enum begin

definition enum = [a1, a2, a3, a4]

definition enum-all P ⇔ P a1 ∧ P a2 ∧ P a3 ∧ P a4

definition enum-ex P ⇔ P a1 ∨ P a2 ∨ P a3 ∨ P a4
instance proof

qed (simp-all only: enum-finite-4-def enum-all-finite-4-def enum-ex-finite-4-def UNIV-finite-4, simp-all)

end

instantiation finite-4 :: finite-distrib-lattice begin

\(a_1 < a_2, a_3 < a_4\), but \(a_2\) and \(a_3\) are incomparable.

definition 
\(x < y \leftrightarrow \text{case } (x, y) \text{ of}
\begin{align*}
(a_1, a_1) &\Rightarrow \text{False} | (a_1, \cdot) \Rightarrow \text{True} \\
(a_2, a_4) &\Rightarrow \text{True} \\
(a_3, a_4) &\Rightarrow \text{True} | \cdot \Rightarrow \text{False}
\end{align*}

\end{definition}

definition 
\(x \leq y \leftrightarrow \text{case } (x, y) \text{ of}
\begin{align*}
(a_1, \cdot) &\Rightarrow \text{True} \\
(a_2, a_2) &\Rightarrow \text{True} | (a_2, a_4) \Rightarrow \text{True} \\
(a_3, a_3) &\Rightarrow \text{True} | (a_3, a_4) \Rightarrow \text{True} \\
(a_4, a_4) &\Rightarrow \text{True} | \cdot \Rightarrow \text{False}
\end{align*}

\end{definition}

definition 
\(\prod A = (\text{if } a_1 \in A \lor a_2 \in A \land a_3 \in A \text{ then } a_1 \text{ else if } a_2 \in A \text{ then } a_2 \text{ else if } a_3 \in A \text{ then } a_3 \text{ else } a_4)\)

definition 
\(\bigsqcap A = (\text{if } a_4 \in A \lor a_2 \in A \land a_3 \in A \text{ then } a_4 \text{ else if } a_2 \in A \text{ then } a_2 \text{ else if } a_3 \in A \text{ then } a_3 \text{ else } a_1)\)

definition \[\text{[simp]}: \bot = a_1\]

definition \[\text{[simp]}: \top = a_4\]

definition 
\(x \sqcap y = (\text{case } (x, y) \text{ of}
\begin{align*}
(a_1, \cdot) &\Rightarrow a_1 | (\cdot, a_1) \Rightarrow a_1 | (a_2, a_3) \Rightarrow a_1 | (a_3, a_2) \Rightarrow a_1 \\
(a_2, \cdot) &\Rightarrow a_2 | (\cdot, a_2) \Rightarrow a_2 \\
(a_3, \cdot) &\Rightarrow a_3 | (\cdot, a_3) \Rightarrow a_3 \\
\cdot &\Rightarrow a_4
\end{align*}
\)

definition 
\(x \sqcup y = (\text{case } (x, y) \text{ of}
\begin{align*}
(a_4, \cdot) &\Rightarrow a_4 | (\cdot, a_4) \Rightarrow a_4 | (a_2, a_3) \Rightarrow a_4 | (a_3, a_2) \Rightarrow a_4 \\
(a_2, \cdot) &\Rightarrow a_2 | (\cdot, a_2) \Rightarrow a_2 \\
(a_3, \cdot) &\Rightarrow a_3 | (\cdot, a_3) \Rightarrow a_3 \\
\cdot &\Rightarrow a_1
\end{align*}
\)

instance

by standard

(subproofs
  (auto simp add: less-finite-4-def less-eq-finite-4-def Inf-finite-4-def Sup-finite-4-def inf-finite-4-def sup-finite-4-def split: finite-4.splits)
end

instance finite-4 :: complete-lattice ..

instance finite-4 :: complete-distrib-lattice ..

instantiation finite-4 :: complete-boolean-algebra begin
definition $-x = (\text{case } x \text{ of } a_1 \Rightarrow a_4 | a_2 \Rightarrow a_3 | a_3 \Rightarrow a_2 | a_4 \Rightarrow a_1)$
definition $x - y = x \cap -(y :: \text{finite-4})$
instantiation
by standard
(subproofs
(simp-all add: inf-finite-4-def sup-finite-4-def uminus-finite-4-def minus-finite-4-def
split: finite-4.splits))
end

hide-const (open) $a_1 \ a_2 \ a_3 \ a_4$

datatype (plugins only: code quickcheck extraction) finite-5 = $a_1 | a_2 | a_3 | a_4 | a_5$

notation (output) $a_1$ ($a_1$)
notation (output) $a_2$ ($a_2$)
notation (output) $a_3$ ($a_3$)
notation (output) $a_4$ ($a_4$)
notation (output) $a_5$ ($a_5$)

lemma $\text{UNIV-finite-5}$:
$\text{UNIV} = \{a_1, a_2, a_3, a_4, a_5\}$
by (auto intro: finite-5.exhaust)

instantiation finite-5 :: enum begin
definition $\text{enum} = [a_1, a_2, a_3, a_4, a_5]$
definition $\text{enum-all } P \iff P \ a_1 \land P \ a_2 \land P \ a_3 \land P \ a_4 \land P \ a_5$
definition $\text{enum-ex } P \iff P \ a_1 \lor P \ a_2 \lor P \ a_3 \lor P \ a_4 \lor P \ a_5$
instance proof
qed (simp-all only: enum-finite-5-def enum-all-finite-5-def enum-ex-finite-5-def UNIV-finite-5, simp-all)
end
The non-distributive pentagon lattice \( N_5 \)

**definition**

\[ x < y \iff (\text{case } (x, y) \text{ of} \]

\[ (a_1, a_1) \Rightarrow \text{False} \mid (a_1, \cdot) \Rightarrow \text{True} \]
\[ (a_2, a_3) \Rightarrow \text{True} \mid (a_2, a_5) \Rightarrow \text{True} \]
\[ (a_3, a_5) \Rightarrow \text{True} \]
\[ (a_4, a_5) \Rightarrow \text{True} \mid - \Rightarrow \text{False} \]

**definition**

\[ x \leq y \iff (\text{case } (x, y) \text{ of} \]

\[ (a_1, \cdot) \Rightarrow \text{True} \]
\[ (a_2, a_2) \Rightarrow \text{True} \mid (a_2, a_3) \Rightarrow \text{True} \mid (a_2, a_5) \Rightarrow \text{True} \]
\[ (a_3, a_3) \Rightarrow \text{True} \mid (a_3, a_5) \Rightarrow \text{True} \]
\[ (a_4, a_4) \Rightarrow \text{True} \mid (a_4, a_5) \Rightarrow \text{True} \]
\[ (a_5, a_5) \Rightarrow \text{True} \mid - \Rightarrow \text{False} \]

**definition**

\[ \bigwedge A = \]

\[ (\text{if } a_1 \in A \lor a_4 \in A \land (a_2 \in A \lor a_3 \in A) \text{ then } a_1 \]
\[ \quad \text{else if } a_2 \in A \text{ then } a_2 \]
\[ \quad \text{else if } a_3 \in A \text{ then } a_3 \]
\[ \quad \text{else if } a_4 \in A \text{ then } a_4 \]
\[ \quad \text{else } a_5) \]

**definition**

\[ \bigvee A = \]

\[ (\text{if } a_5 \in A \lor a_4 \in A \land (a_2 \in A \lor a_3 \in A) \text{ then } a_5 \]
\[ \quad \text{else if } a_3 \in A \text{ then } a_3 \]
\[ \quad \text{else if } a_2 \in A \text{ then } a_2 \]
\[ \quad \text{else if } a_4 \in A \text{ then } a_4 \]
\[ \quad \text{else } a_1) \]

**definition** [simp]: \( \bot = a_1 \)

**definition** [simp]: \( \top = a_5 \)

**definition**

\[ x \cap y = (\text{case } (x, y) \text{ of} \]

\[ (a_1, \cdot) \Rightarrow a_1 \mid (\cdot, a_1) \Rightarrow a_1 \mid (a_2, a_4) \Rightarrow a_1 \mid (a_4, a_2) \Rightarrow a_1 \mid (a_3, a_4) \Rightarrow a_1 \mid \]
\[ (a_4, a_3) \Rightarrow a_1 \]
\[ (a_2, \cdot) \Rightarrow a_2 \mid (\cdot, a_2) \Rightarrow a_2 \]
\[ (a_3, \cdot) \Rightarrow a_3 \mid (\cdot, a_3) \Rightarrow a_3 \]
\[ (a_4, \cdot) \Rightarrow a_4 \mid (\cdot, a_4) \Rightarrow a_4 \]
\[ \cdot \Rightarrow a_5) \]

**definition**

\[ x \cup y = (\text{case } (x, y) \text{ of} \]

\[ (a_5, \cdot) \Rightarrow a_5 \mid (\cdot, a_5) \Rightarrow a_5 \mid (a_2, a_4) \Rightarrow a_5 \mid (a_4, a_2) \Rightarrow a_5 \mid (a_3, a_4) \Rightarrow a_5 \mid \]
\[ (a_4, a_3) \Rightarrow a_5 \]
\[ (a_3, \cdot) \Rightarrow a_3 \mid (\cdot, a_3) \Rightarrow a_3 \]
THEORY "String"

| (a₂, ·) ⇒ a₂ | (·, a₂) ⇒ a₂
| (a₄, ·) ⇒ a₄ | (·, a₄) ⇒ a₄
| · ⇒ a₁)

instance
by standard
(subproofs
(auto simp add: less-eq-finite-5-def less-finite-5-def inf-finite-5-def sup-finite-5-def
Inf-finite-5-def Sup-finite-5-def split: finite-5.splits if-split-asm)
end

instance finite-5 :: complete-lattice ..

hide-const (open) a₁ a₂ a₃ a₄ a₅

70.5 Closing up
hide-type (open) finite-1 finite-2 finite-3 finite-4 finite-5
hide-const (open) enum enum-all enum-ex all-n-lists ex-n-lists ntrancl
end

71 Character and string types

theory String imports Enum begin

71.1 Strings as list of bytes

When modelling strings, we follow the approach given in https://utf8everywhere.org/:

- Strings are a list of bytes (8 bit).
- Byte values from 0 to 127 are US-ASCII.
- Byte values from 128 to 255 are uninterpreted blobs.

71.1.1 Bytes as datatype

context unique-euclidean-semiring-with-bit-shifts
begin
lemma bit-horner-sum-iff:
\[ \langle \text{bit} (\text{foldr} (\lambda b k. \text{of-bool } b + k \cdot 2) \text{bs } 0) \rangle |\! n \iff n < \text{length bs} \wedge \text{bs} !\! n \rangle \]
proof (induction bs arbitrary: n)
  case Nil
  then show ?case
  by simp
next
  case (Cons b bs)
  show ?case
  proof (cases n)
    case \theta
    then show ?thesis
    by simp
  next
    case (Suc m)
    with bit-rec [of \!\! n] Cons.prems Cons.IH [of m]
    show ?thesis by simp
  qed
qed

lemma take-bit-horner-sum-eq:
\[ \langle \text{take-bit } n (\text{foldr} (\lambda b k. \text{of-bool } b + k \cdot 2) \text{bs } 0) = \text{foldr} (\lambda b k. \text{of-bool } b + k \cdot 2) (\text{take } n \text{ bs}) 0 \rangle \]
proof (induction bs arbitrary: n)
  case Nil
  then show ?case
  by simp
next
  case (Cons b bs)
  show ?case
  proof (cases n)
    case \theta
    then show ?thesis
    by simp
  next
    case (Suc m)
    with take-bit-rec [of \!\! n] Cons.prems Cons.IH [of m]
    show ?thesis by (simp add: ac-simps)
  qed
qed

lemma (in semiring-bit-shifts) take-bit-eq-horner-sum:
\[ \langle \text{take-bit } n \text{ a} = \text{foldr} (\lambda b k. \text{of-bool } b + k \cdot 2) (\text{map} (\text{bit} \text{ a}) [0..<n]) 0 \rangle \]
proof (induction a arbitrary: n rule: bits-induct)
  case (stable a)
  have \*: \langle (\lambda k. k \cdot 2)^\wedge n \rangle \Theta \equiv \Theta \rangle
  by (induction n) simp-all
  from stable have \langle \text{bit a} = (\lambda -. \text{odd a}) \rangle
  by (simp add: stable-imp-bit-iff-odd fun-eq-iff)
then have \( \langle \text{map} \ (\text{bit} \ a) \ [0..<n] = \text{replicate} \ n \ (\text{odd} \ a) \rangle \)
  by (simp add: \text{map-replicate-const})
with stable show \( ?\text{case} \)
  by (simp add: \text{stable-imp-take-bit-eq mask-eq-seq-sum} *)
next
  case (rec \( a \) \( b \))
  show \( ?\text{case} \)
  proof (cases \( n \))
    case 0
    then show \( ?\text{thesis} \)
    by simp
  next
    case (Suc \( m \))
    have \( \langle \text{map} \ (\text{bit} \ (\text{of-bool} \ b + 2 \ * \ a)) \ [0..<\text{Suc} \ m] = b \ \# \ \text{map} \ (\text{bit} \ (\text{of-bool} \ b + 2 \ * \ a)) \ [\text{Suc} \ 0..<\text{Suc} \ m] \rangle \)
      by (simp only: \text{upt-conv-Cons}) simp
    also have \( \ldots = b \ \# \ \text{map} \ (\text{bit} \ a) \ [0..<m] \rangle \)
      by (simp only: \text{flip: map-Suc-upt}) (simp add: \text{bit-Suc rec. hyps})
    finally show \( ?\text{thesis} \)
      using Suc rec.IH \( \text{of} \ m \) by (simp add: \text{take-bit-Suc rec. hyps}, simp add: \text{ac-simps})
  qed
qed
end

datatype char =
Char (digit0: bool) (digit1: bool) (digit2: bool) (digit3: bool)
  (digit4: bool) (digit5: bool) (digit6: bool) (digit7: bool)
context comm-semiring-1
begin

definition of-char :: \( \text{char} \Rightarrow \alpha \)
  where of-char \( c \) = foldr (\( \lambda \ b \ k. \ \text{of-bool} \ b + k \ * \ 2 \))
    [digit0 \( c \), digit1 \( c \), digit2 \( c \), digit3 \( c \), digit4 \( c \), digit5 \( c \), digit6 \( c \), digit7 \( c \) \( 0 \) 0)

lemma of-char-Char [simp]:
  of-char (Char b0 b1 b2 b3 b4 b5 b6 b7) =
  foldr (\( \lambda \ b \ k. \ \text{of-bool} \ b + k \ * \ 2 \)) \ [b0, b1, b2, b3, b4, b5, b6, b7] \ 0;
  by (simp add: of-char-def)
end

context unique-euclidean-semiring-with-bit-shifts
begin

definition char-of :: \( \alpha \Rightarrow \text{char} \)
  where \( \text{char-of} \ n = \text{Char} \ (\text{odd} \ n) \ (\text{bit} \ n \ 1) \ (\text{bit} \ n \ 2) \ (\text{bit} \ n \ 3) \ (\text{bit} \ n \ 4) \ (\text{bit} \ n \ 5) \ (\text{bit} \ n \ 6) \ (\text{bit} \ n \ 7) \ (\text{bit} \ n \ 8) \)

5) \((\text{bit } n 6)(\text{bit } n 7)\)

**lemma** char-of-take-bit-eq:
\(<\text{char-of } (\text{take-bit } n m) = \text{char-of } m \text{ if } n \geq 8>\)
using that by \((\text{simp add: char-of-def bit-take-bit-iff})\)

**lemma** char-of-char [simp]:
\(<\text{char-of } (\text{of-char } c) = c>\)
by \((\text{simp only: of-char-def char-of-def bit-horner-sum-iff})\) simp

**lemma** char-of-comp-of-char [simp]:
\(<\text{char-of } \circ \text{of-char } = \text{id}>\)
by \((\text{simp add: fun-eq-iff})\)

**lemma** inj-of-char:
\(<\text{inj of-char}>\)
**proof** (rule injI)
fix \(c\) \(d\)
assume \(<\text{of-char } c = \text{of-char } d>\)
then have \(<\text{char-of } (\text{of-char } c) = \text{char-of } (\text{of-char } d)>\)
by simp
then show \(<c = d>\)
by simp
qed

**lemma** of-char-eqI:
\(<c = d \text{ if } (\text{of-char } c = \text{of-char } d)>\)
using that inj-of-char by \((\text{simp add: inj-eq})\)

**lemma** of-char-eq-iff [simp]:
\(<\text{of-char } c = \text{of-char } d \leftrightarrow c = d>\)
by \((\text{auto intro: of-char-eqI})\)

**lemma** of-char-of [simp]:
\(<\text{of-char } (\text{char-of } a) = a \bmod 256>\)
**proof** –
have \(<[0..<8] = [0, \text{Suc } 0, 2, 3, 4, 5, 6, 7 :: \text{nat}]>\)
by \((\text{simp add: upt-Cons-conv})\)
then have \(<[\text{odd } a, \text{bit } a 1, \text{bit } a 2, \text{bit } a 3, \text{bit } a 4, \text{bit } a 5, \text{bit } a 6, \text{bit } a 7] = \text{map } (\text{bit } a) [0..<8]>\)
by simp
then have \(<\text{of-char } (\text{char-of } a) = \text{take-bit } 8 a>\)
by \((\text{simp only: char-of-def of-char-def char.sel take-bit-eq-horner-sum})\)
then show \(<?thesis>\)
by \((\text{simp add: take-bit-eq-mod})\)
qed

**lemma** char-of-mod-256 [simp]:
\(<\text{char-of } (n \bmod 256) = \text{char-of } n>\)
THEORY "String"
proof (rule; rule)
  fix n :: nat
  assume n ∈ range of-char
  then show n ∈ {0..<256}
    by auto
next
  fix n :: nat
  assume n ∈ {0..<256}
  then have n = of-char (char-of n)
    by simp
  then show n ∈ range of-char
    by (rule range-eqI)
qed

lemma UNIV-char-of-nat:
  UNIV = char-of ' {0::nat..<256}
proof –
  have range (of-char :: char ⇒ nat) = of-char ' char-of ' {0::nat..<256}
    by (auto simp add: range-nat-of-char intro: image-eqI)
  with inj-of-char [where ?'a = nat] show ?thesis
    by (simp add: inj-image-eq-iff)
qed

lemma card-UNIV-char:
  card (UNIV :: char set) = 256
by (auto simp add: UNIV-char-of-nat card-image)

context
  includes lifting-syntax integer.lifting natural.lifting
begin
lemma [transfer-rule]:
  ⟨(pcr-integer ===> (=)) char-of char-of⟩
by (unfold char-of-def) transfer-prover
lemma [transfer-rule]:
  ⟨((=) ===> pcr-integer) of-char of-char⟩
by (unfold of-char-def) transfer-prover
lemma [transfer-rule]:
  ⟨(pcr-natural ===> (=)) char-of char-of⟩
by (unfold char-of-def) transfer-prover
lemma [transfer-rule]:
  ⟨((=) ===> pcr-natural) of-char of-char⟩
by (unfold of-char-def) transfer-prover
end
lifting-update integer.lifting
lifting-forget integer.lifting

lifting-update natural.lifting
lifting-forget natural.lifting

syntax
-Char :: str-position ⇒ char  (CHR -)
-Char-ord :: num-const ⇒ char  (CHR -)

type-synonym string = char list

syntax
-String :: str-position ⇒ string  (-)

ML-file (Tools/string-syntax.ML)

instantiation char :: enum
begin

definition
Enum.enum = [
  CHR 0x00, CHR 0x01, CHR 0x02, CHR 0x03,
  CHR 0x04, CHR 0x05, CHR 0x06, CHR 0x07,
  CHR 0x08, CHR 0x09, CHR 0x0A, CHR 0x0B,
  CHR 0x0C, CHR 0x0D, CHR 0x0E, CHR 0x0F,
  CHR 0x10, CHR 0x11, CHR 0x12, CHR 0x13,
  CHR 0x14, CHR 0x15, CHR 0x16, CHR 0x17,
  CHR 0x18, CHR 0x19, CHR 0x1A, CHR 0x1B,
  CHR 0x1C, CHR 0x1D, CHR 0x1E, CHR 0x1F,
  CHR '"', CHR '"', CHR 0x22, CHR '#',
  CHR '$', CHR '%', CHR '&', CHR 0x27,
  CHR '(', CHR ')', CHR '*', CHR '+',
  CHR ',', CHR '-', CHR '.', CHR '/')',
  CHR '0', CHR '1', CHR '2', CHR '3',
  CHR '4', CHR '5', CHR '6', CHR '7',
  CHR '8', CHR '9', CHR ':', CHR ';',
  CHR '<', CHR '=', CHR '>', CHR '?',
  CHR '@', CHR 'A', CHR 'B', CHR 'C',
  CHR 'D', CHR 'E', CHR 'F', CHR 'G',
  CHR 'H', CHR 'I', CHR 'J', CHR 'K',
  CHR 'L', CHR 'M', CHR 'N', CHR 'O',
  CHR 'P', CHR 'Q', CHR 'R', CHR 'S',
  CHR 'T', CHR 'U', CHR 'V', CHR 'W',
  CHR 'X', CHR 'Y', CHR 'Z', CHR 'a',
  CHR 'b', CHR 'c', CHR 'd', CHR 'e',
  CHR 'f', CHR 'g', CHR 'h',
  CHR 'i', CHR 'j', CHR 'k',
]
THEORY “String” 1421

CHR "I", CHR "m", CHR "n", CHR "o", CHR "p", CHR "q", CHR "r", CHR "s", CHR "t", CHR "u", CHR "v", CHR "w", CHR "x", CHR "y", CHR "z", CHR ",", CHR ",", CHR ",", CHR ",", CHR ",", CHR "~", CHR 0x7F,
CHR 0x80, CHR 0x81, CHR 0x82, CHR 0x83, CHR 0x84, CHR 0x85, CHR 0x86, CHR 0x87, CHR 0x88, CHR 0x89, CHR 0x8A, CHR 0x8B, CHR 0x8C, CHR 0x8D, CHR 0x8E, CHR 0x8F, CHR 0x90, CHR 0x91, CHR 0x92, CHR 0x93, CHR 0x94, CHR 0x95, CHR 0x96, CHR 0x97, CHR 0x98, CHR 0x99, CHR 0x9A, CHR 0x9B, CHR 0x9C, CHR 0x9D, CHR 0x9E, CHR 0x9F, CHR 0xA0, CHR 0xA1, CHR 0xA2, CHR 0xA3, CHR 0xA4, CHR 0xA5, CHR 0xA6, CHR 0xA7, CHR 0xA8, CHR 0xA9, CHR 0xAA, CHR 0xAB, CHR 0xAC, CHR 0xAD, CHR 0xAE, CHR 0xAF, CHR 0xB0, CHR 0xB1, CHR 0xB2, CHR 0xB3, CHR 0xB4, CHR 0xB5, CHR 0xB6, CHR 0xB7, CHR 0xB8, CHR 0xB9, CHR 0xBA, CHR 0xBB, CHR 0xBC, CHR 0xBD, CHR 0xBE, CHR 0xBF, CHR 0xC0, CHR 0xC1, CHR 0xC2, CHR 0xC3, CHR 0xC4, CHR 0xC5, CHR 0xC6, CHR 0xC7, CHR 0xC8, CHR 0xC9, CHR 0xCA, CHR 0xCB, CHR 0xCC, CHR 0xCD, CHR 0xCE, CHR 0xCF, CHR 0xD0, CHR 0xD1, CHR 0xD2, CHR 0xD3, CHR 0xD4, CHR 0xD5, CHR 0xD6, CHR 0xD7, CHR 0xD8, CHR 0xD9, CHR 0xDA, CHR 0xDB, CHR 0xDC, CHR 0xDD, CHR 0xDE, CHR 0xDF, CHR 0xE0, CHR 0xE1, CHR 0xE2, CHR 0xE3, CHR 0xE4, CHR 0xE5, CHR 0xE6, CHR 0xE7, CHR 0xE8, CHR 0xE9, CHR 0xEA, CHR 0xEB, CHR 0xEC, CHR 0xED, CHR 0xEE, CHR 0xEF, CHR 0xF0, CHR 0xF1, CHR 0xF2, CHR 0xF3, CHR 0xF4, CHR 0xF5, CHR 0xF6, CHR 0xF7, CHR 0xF8, CHR 0xF9, CHR 0xFA, CHR 0xFB, CHR 0xFC, CHR 0xFD, CHR 0xFE, CHR 0xFF]

definition
Enum.enum-all P ⇔ list-all P (Enum.enum :: char list)

definition
Enum.enum-ex P ⇔ list-ex P (Enum.enum :: char list)

lemma enum-char-unfold:
Enum.enum = map char-of [0..<256]

proof –
  have map (of-char :: char ⇒ nat) Enum.enum = [0..<256]
    by (simp add: enum-char-def of-char-def apt-cons-Cons-Cons numeral-2-eq-2
THEORY "String"

[symmetric]
then have map char-of (map (of-char :: char ⇒ nat) Enum.enum) =
  map char-of [0..<256]
  by simp
then show ?thesis
  by simp
qed

instance proof
show UNIV: UNIV = set (Enum.enum :: char list)
  by (simp add: enum-char-unfold UNIV-char-of-nat atLeast0LessThan)
show distinct (Enum.enum :: char list)
  by (auto simp add: enum-char-unfold distinct-map intro: inj-onI)
show ∃P. Enum.enum-all P ←→ Ball (UNIV :: char set) P
  by (simp add: UNIV enum-all-char-def list-all-iff)
show ∃P. Enum.enum-ex P ←→ Bex (UNIV :: char set) P
  by (simp add: UNIV enum-ex-char-def list-ex-iff)
qed

end

lemma linorder-char:
class linorder (λ c d. of-char c ≤ (of-char d :: nat)) (λ c d. of-char c < (of-char d :: nat))
  by standard auto

Optimized version for execution

definition char-of-integer :: integer ⇒ char
  where [code-abbrev]: char-of-integer = char-of

definition integer-of-char :: char ⇒ integer
  where [code-abbrev]: integer-of-char = of-char

lemma char-of-integer-code [code]:
  char-of-integer k = (let
    (q0, b0) = bit-cut-integer k;
    (q1, b1) = bit-cut-integer q0;
    (q2, b2) = bit-cut-integer q1;
    (q3, b3) = bit-cut-integer q2;
    (q4, b4) = bit-cut-integer q3;
    (q5, b5) = bit-cut-integer q4;
    (q6, b6) = bit-cut-integer q5;
    (-, b7) = bit-cut-integer q6
    in Char b0 b1 b2 b3 b4 b5 b6 b7)

lemma integer-of-char-code [code]:
  integer-of-char (Char b0 b1 b2 b3 b4 b5 b6 b7) =
71.2 Strings as dedicated type for target language code generation

71.2.1 Logical specification

begin

qualified definition ascii-of :: char \Rightarrow char
  where ascii-of c = Char (digit0 c) (digit1 c) (digit2 c) (digit3 c) (digit4 c) (digit5 c) (digit6 c) False

qualified lemma ascii-of-Char [simp]:
  ascii-of (Char b0 b1 b2 b3 b4 b5 b6 b7) = Char b0 b1 b2 b3 b4 b5 b6 False
  by (simp add: ascii-of-def)

qualified lemma not-digit7-ascii-of [simp]:
  \neg digit7 (ascii-of c)
  by (simp add: ascii-of-def)

qualified lemma ascii-of-idem:
  ascii-of c = c if \neg digit7 c
  using that by (cases c) simp

qualified lemma char-of-ascii-of [simp]:
  of-char (ascii-of c) = take-bit 7 (of-char c :: nat)
  by (cases c) (simp only: ascii-of-Char of-char-Char take-bit-horner-sum-eq, simp)

qualified typedef literal = \{cs. \forall c \in set cs. \neg digit7 c\}
  morphisms explode Abs-literal
proof
  show \[] \in \{cs. \forall c \in set cs. \neg digit7 c\}
  by simp
qed

qualified setup-lifting type-definition-literal

qualified lift-definition implode :: string \Rightarrow literal
  is map ascii-of
  by auto

qualified lemma implode-explode-eq [simp]:
  String.implode (String.explode s) = s
proof transfer
THEORY "String"

fix cs
show map ascii-of cs = cs if \( \forall c \in \text{set } cs. \neg \text{digit7 } c \)
using that
by \((\text{induction } cs) \ (\text{simp-all add: ascii-of-idem})\)
qed

qualified lemma explode-implode-eq [simp]:
\( \text{String.explode } (\text{String.implode } cs) = \text{map ascii-of } cs \)
by transfer rule
end

71.2.2 Syntactic representation

Logical ground representations for literals are:

1. \(0\) for the empty literal;
2. \(\text{Literal } b0 \ldots b6 s\) for a literal starting with one character and continued by another literal.
   Syntactic representations for literals are:
3. Printable text as string prefixed with \(\text{STR}\);
4. A single ascii value as numerical hexadecimal value prefixed with \(\text{STR}\).

instantiation \(\text{String.literal} :: \text{zero}\)
begin
context
begin
qualified lift-definition zero-literal :: \(\text{String.literal}\)
   is Nil
   by simp
instance ..
end
end
context
begin
qualified abbreviation (output) empty-literal :: \(\text{String.literal}\)
   where empty-literal \(\equiv 0\)
qualified lift-definition Literal :: bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ String.literal ⇒ String.literal
  is λb0 b1 b2 b3 b4 b5 b6 cs. Char b0 b1 b2 b3 b4 b5 b6 False ≠ cs
  by auto

qualified lemma Literal-eq-iff [simp];
  Literal b0 b1 b2 b3 b4 b5 b6 s = Literal c0 c1 c2 c3 c4 c5 c6 t
  ℓ→ (b0 ↔ c0) ∧ (b1 ↔ c1) ∧ (b2 ↔ c2) ∧ (b3 ↔ c3)
  ∧ (b4 ↔ c4) ∧ (b5 ↔ c5) ∧ (b6 ↔ c6) ∧ s = t
  by transfer simp

qualified lemma empty-neq-Literal [simp];
  empty-literal ≠ Literal b0 b1 b2 b3 b4 b5 b6 s
  by transfer simp

qualified lemma Literal-neq-empty [simp];
  Literal b0 b1 b2 b3 b4 b5 b6 s ≠ empty-literal
  by transfer simp

end

code-datatype 0 :: String.literal String.Literal

syntax
  -Literal :: str-position ⇒ String.literal (STR -)
  -Ascii :: num-const ⇒ String.literal (STR -)

ML-file ⟨Tools/literal.ML⟩

71.2.3 Operations

instantiation String.literal :: plus
begin

context
begin

qualified lift-definition plus-literal :: String.literal ⇒ String.literal ⇒ String.literal
  is (@)
  by auto

instance ..

end

end

instance String.literal :: monoid-add
  by (standard; transfer) simp-all
instantiation String.literal :: size
begin

context
  includes literal.lifting
begin

lift-definition size-literal :: String.literal ⇒ nat
  is length .
end

instance ..
end

instantiation String.literal :: equal
begin

context
begin

qualified lift-definition equal-literal :: String.literal ⇒ String.literal ⇒ bool
  is HOL.equal .

instance
  by (standard; transfer) (simp add: equal)
end
end

instantiation String.literal :: linorder
begin

context
begin

qualified lift-definition less-eq-literal :: String.literal ⇒ String.literal ⇒ bool
  is ord.lexordp-eq (λc d. of-char c < (of-char d :: nat))
  .

qualified lift-definition less-literal :: String.literal ⇒ String.literal ⇒ bool
  is ord.lexordp (λc d. of-char c < (of-char d :: nat))
  .

instance proof −
  from linorder-char interpret linorder ord.lexordp-eq (λc d. of-char c < (of-char

lemma infinite-literal:
  infinite (UNIV :: String.literal set)
proof –
define S where S = range (\n. replicate n CHR "A")
have inj-on String.implode S 
proof (rule inj-onI)
  fix cs ds
  assume String.implode cs = String.implode ds
  then have String.explode (String.implode cs) = String.explode (String.implode ds)
  by simp
moreover assume cs \in S and ds \in S
ultimately show cs = ds
  by (auto simp add: S-def)
qed
moreover have infinite S 
  by (auto simp add: S-def dest: finite-range-imageI [of - length])
ultimately have infinite (String.implode " S)
then show \thesis
  by (auto intro: finite-subset)
qed

71.2.4 Executable conversions
context
begin
qualified lift-definition ascii-of-literal :: String.literal \Rightarrow integer list
  is map of-char
.

qualified lemma ascii-of-zero [simp, code]:
  ascii-of-literal 0 = []
  by transfer simp

qualified lemma ascii-of-Literal [simp, code]:
  ascii-of-literal (String.Literal b0 b1 b2 b3 b4 b5 b6 s) =
of-char (Char b0 b1 b2 b3 b4 b5 b6 False) ≠ ascii-of-literal s  
by transfer simp

qualified lift-definition literal-of-asciis :: integer list ⇒ String.literal  
is map (String.ascii-of o char-of)  
by auto

qualified lemma literal-of-asciis-Nil [simp, code]:  
literal-of-asciis [] = 0  
by transfer simp

qualified lemma literal-of-asciis-Cons [simp, code]:  
literal-of-asciis (k # ks) = (case char-of k  
of Char b0 b1 b2 b3 b4 b5 b6 b7 ⇒ String.Literal b0 b1 b2 b3 b4 b5 b6  
(literal-of-asciis ks))  
by (simp add: char-of-def) (transfer, simp add: char-of-def)

qualified lemma literal-of-asciis-of-literal [simp]:  
literal-of-asciis (ascii-of-literal s) = s  
proof transfer  
fix cs  
assume ∀ c∈set cs. ¬ digit7 c  
then show map (String.ascii-of o char-of) (map of-char cs) = cs  
by (induction cs) (simp-all add: String.ascii-of-idem)  
qed

qualified lemma explode-code [code]:  
String.explode s = map char-of (ascii-of-literal s)  
by transfer simp

qualified lemma implode-code [code]:  
String.implode cs = literal-of-asciis (map of-char cs)  
by transfer simp

qualified lemma equal-literal [code]:  
HOL.equal (String.Literal b0 b1 b2 b3 b4 b5 b6 s)  
(String.Literal a0 a1 a2 a3 a4 a5 a6 r)  
←→ (b0 ←→ a0) ∧ (b1 ←→ a1) ∧ (b2 ←→ a2) ∧ (b3 ←→ a3)  
∧ (b4 ←→ a4) ∧ (b5 ←→ a5) ∧ (b6 ←→ a6) ∧ (s = r)  
by (simp add: equal)

end

71.2.5 Technical code generation setup

Alternative constructor for generated computations

case
begin
qualified definition Literal' :: bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ bool ⇒ String.literal ⇒ String.literal
  where [simp]: Literal' = String.Literal

lemma [code]:
 ⟨Literal' b0 b1 b2 b3 b4 b5 b6 s = String.literal-of-asciis
  [foldr (λb k. of-bool b + k * 2) [b0, b1, b2, b3, b4, b5, b6] 0] + s⟩

proof −
 have ⟨foldr (λb k. of-bool b + k * 2) [b0, b1, b2, b3, b4, b5, b6] 0 = of-char (Char b0 b1 b2 b3 b4 b5 b6 False)⟩
  by simp

moreover have ⟨Literal' b0 b1 b2 b3 b4 b5 b6 s = String.literal-of-asciis
  [of-char (Char b0 b1 b2 b3 b4 b5 b6 False)] + s⟩
  by (unfold Literal' def) (transfer, simp only: list.simps comp-apply char-of-char, simp)

ultimately show ?thesis
  by simp
qed

lemma [code-computation-unfold]:
 String.Literal = Literal'
 by simp
end

code-reserved SML string String Char List
code-reserved OCaml string String Char List
code-reserved Haskell Prelude
code-reserved Scala string

code-printing
type-constructor String.literal ⇒
  (SML) string
  and (OCaml) string
  and (Haskell) String
  and (Scala) String
| constant STR """" ⇒
  (SML)
  and (OCaml)
  and (Haskell)
  and (Scala)

setup ⟨
  fold Literal.add-code [SML, OCaml, Haskell, Scala]
⟩

code-printing
constant (+) :: String.literal ⇒ String.literal ⇒ String.literal ⇒
  (SML) infixl 18 "

and (OCaml) infixr 6 
and (Haskell) infixr 5 ++
and (Scala) infixl 7 +

| constant String.literal-of-ascii ⇒
  (SML) ![String.implode/ o List.map (fn k => if 0 <= k andalso k < 128 then Char.chr o IntInf.toInt) k else raise Fail Non-ASCII character in literal])
and (OCaml) ![let zs = -
  and chr k =
  let l = Z.to'-int k
  in if 0 <= l && l < 128
  then Char.l
  else failwith Non-ASCII character in literal
  in String.init (List.length xs) (List.nth (List.map chr xs)))
and (Haskell) map/ (let chr k | (0 <= k && k < 128) = Prelude.toEnum k :: Prelude.Char in chr . Prelude.fromInteger)
and (Scala) / /+/-map(k: BigInt) => if (BigInt(0) <= k && k < BigInt(128)) k.charValue else sys.error(Non-ASCII character in literal))

| constant String.asciis-of-literal ⇒
  (SML) ![List.map (fn c => let val k = Char.ord c in if k < 128 then IntInf.fromInt k else raise Fail Non-ASCII character in literal end) / o String.explode)
and (OCaml) ![let s = - in let rec exp i l = if i < 0 then l else exp (i - 1) (let k = Char.code (String.get s i) in
  if k < 128 then Z.of'-int k :: l else failwith Non-ASCII character in literal)
in exp (String.length s - 1) []]
and (Haskell) map/ (let ord k | (k < 128) = Prelude.toInteger k in ord . (Prelude.fromEnum :: Prelude.Char -> Prelude.Int))
and (Scala) ![-.toList(c => { val k = Int = c.toInt; if (k < 128) BigInt(k)
  else sys.error(Non-ASCII character in literal) }))

| class-instance String.literal :: equal →
  (Haskell) –
| constant HOL.equal :: String.literal ⇒ String.literal ⇒ bool ⇒
  (SML) ![(!: : string) = -]
and (OCaml) ![(!: : string) = -]
and (Haskell) infix 4 ==
and (Scala) infixl 5 ==
| constant (≤) :: Stringliteral ⇒ String.literal ⇒ bool ⇒
  (SML) ![(!: : string) <= -]
and (OCaml) ![(!: : string) <= -]
and (Haskell) infix 4 <=
| constant (<) :: String.literal ⇒ String.literal ⇒ bool ⇒
  (SML) ![(!: : string) < -]
and (OCaml) ![(!: : string) < -]
and (Haskell) infix 4 <
and (Scala) infixl 4 <

— Order operations for String.literal work in Haskell only if no type class instance needs to be generated, because String = [Char] in Haskell and char list need not have the same order as String.literal.
and (Eval) infixl 6 <

71.2.6 Code generation utility

setup (Sign.map-naming (Name-Space.mandatory-path Code))

definition abort :: String.literal ⇒ (unit ⇒ 'a) ⇒ 'a
  where [simp]: abort - f = f ()
declare [[code drop: Code.abort]]

lemma abort-cong:
  msg = msg' ⇒ Code.abort msg f = Code.abort msg' f
  by simp

setup (Sign.map-naming Name-Space.parent-path)

setup (Code-Simp.map-ss (Simplifier.add-cong @{thm Code.abort-cong}))

code-printing
constant Code.abort ⇒
  (SML) !(raise/ Fail/ -)
  and (OCaml) failwith
  and (Haskell) !{(error/ ::/ forall a./ String → (() → a) → a)
  and (Scala) !{/ sys.error((-))/((-))}.apply(())

71.2.7 Finally

lifting-update literal.lifting
lifting-forget literal.lifting

end

72 Reflecting Pure types into HOL

theory Typerep
imports String
begin

datatype typerep = Typerep String.literal typerep list

class typerep =
  fixes typerep :: 'a itself ⇒ typerep
begin

definition typerep-of :: 'a ⇒ typerep where
  [simp]: typerep-of x = typerep TYPE('a)
end
THEORY "Typerep"

syntax
-|TYPEREP :: type => logic ((|TYPEREP|/(|\'|(-*))))

parse-translation :
let
  fun typerep-tr (*-TYPEREP*) [ty] =
    Syntax.const const-syntax ⟨typerep⟩ $ (Syntax.const syntax-const ⟨-constr⟩ $ Syntax.const const-syntax ⟨Pure.type⟩ $ Syntax.const type-syntax ⟨itself⟩ $ ty)
  | typerep-tr (*-TYPEREP*) ts = raise TERM (typerep-tr, ts);
in [(syntax-const ⟨-TYPEREP⟩), K typerep-tr] end

typed-print-translation :
let
  fun typerep-tr’ ctxt (*typerep*)
    (Type ⟨typename⟩ [fun], [Type ⟨typename⟩ ⟨itself⟩, [T], -]])
    (Const ⟨const-syntax⟩ ⟨Pure.type⟩, -) :: ts =
      Term.list-comb
        (Syntax.const syntax-const ⟨-TYPEREP⟩, $ Syntax-Phases.term-of-typ ctxt T, ts)
    | typerep-tr’ - T ts = raise Match;
in [(const-syntax ⟨typerep⟩, typerep-tr’)] end

setup :
let

fun add-typerep tyco thy =
let
  val sorts = replicate (Sign.arity-number thy tyco) sort ⟨typerep⟩;
  val vs = Name.invent-names Name.context 'a sorts;
  val ty = Type ⟨typerep⟩, map TFree vs);
  val lhs = Const ⟨const-name⟩ ⟨typerep⟩, Term.itselfT ty ---> typ ⟨typerep⟩;
    $ Free (T, Term.itselfT ty);
  val rhs = term ⟨Typerep⟩ $ HLogic.mk-literal tyco
    $ HLogic.mk-list ty ⟨typerep⟩ (map (HLogic.mk-typerep o TFree) vs);
  val eq = HLogic.mk-Trueprop (HLogic.mk-eq (lhs, rhs));
in thy
  |> Class.instantiation ([tyco], vs, sort ⟨typerep⟩)
  |> (fn lthy => Syntax.check-term lthy eq)
    |> (fn eq => Specification.definition NONE [] [] (Binding.empty-atts, eq))
  |> snd
  |> Class.prove-instantiation-exit (fn ctxt => Class.intro-classes-tac ctxt [])
end;
fun ensure-typerep tyco thy =
  if not (Sorts.has-instance (Sign.classes-of thy) tyco sort typerep)
    andalso Sorts.has-instance (Sign.classes-of thy) tyco sort type
  then add-typerep tyco thy else thy;

in

add-typerep type-name (fun)
  #> Typedef.interpretation (Local-Theory.background-theory o ensure-typerep)
  #> Code.type-interpretation ensure-typerep

end


lemma [code]:
  HOL.equal (Typerep tyco1 tys1) (Typerep tyco2 tys2) ←→ HOL.equal tyco1 tyco2
    ∧ list-all2 HOL.equal tys1 tys2
  by (auto simp add: eq-equal [symmetric] list-all2-eq [symmetric])

lemma [code nbe]:
  HOL.equal (x :: typerep) x ←→ True
  by (fact equal-refl)

code-printing
  type-constructor typerep → (Eval) Term.typ
  | constant Typerep → (Eval) Term.Type/ (-, -)

code-reserved Eval Term

hide-const (open) typerep Typerep

end

73 Predicates as enumerations

theory Predicate
imports String
begin

73.1 The type of predicate enumerations (a monad)

datatype (plugins only: extraction) (dead 'a) pred = Pred (eval: 'a ⇒ bool)

lemma pred-eqI:
  (∀w. eval P w ←→ eval Q w) ⇒ P = Q
  by (cases P, cases Q) (auto simp add: fun-eq-iff)

lemma pred-eq-iff:
  P = Q ⇒ (∀w. eval P w ←→ eval Q w)
THEORY “Predicate”

by (simp add: pred-eqI)

instantiation pred :: (type) complete-lattice
begin

definition
\( P \leq Q \iff \text{eval } P \leq \text{eval } Q \)

definition
\( P < Q \iff \text{eval } P < \text{eval } Q \)

definition
\( \bot = \text{Pred } \bot \)

lemma eval-bot [simp]:
\( \text{eval } \bot = \bot \)
by (simp add: bot-pred-def)

definition
\( \top = \text{Pred } \top \)

lemma eval-top [simp]:
\( \text{eval } \top = \top \)
by (simp add: top-pred-def)

definition
\( P \cap Q = \text{Pred } (\text{eval } P \cap \text{eval } Q) \)

lemma eval-inf [simp]:
\( \text{eval } (P \cap Q) = \text{eval } P \cap \text{eval } Q \)
by (simp add: inf-pred-def)

definition
\( P \sqcup Q = \text{Pred } (\text{eval } P \sqcup \text{eval } Q) \)

lemma eval-sup [simp]:
\( \text{eval } (P \sqcup Q) = \text{eval } P \sqcup \text{eval } Q \)
by (simp add: sup-pred-def)

definition
\( \prod A = \text{Pred } (\prod (\text{eval } \, A)) \)

lemma eval-Inf [simp]:
\( \text{eval } (\prod A) = \prod (\text{eval } \, A) \)
by (simp add: Inf-pred-def)

definition
\( \bigsqcup A = \text{Pred } (\bigsqcup (\text{eval } \, A)) \)
lemma eval-Sup [simp]:
  eval (∪ A) = ∪ (eval ' A)
by (simp add: Sup-pred-def)

instance proof
qed (auto intro!: pred-eqI simp add: less-eq-pred-def less-pred-def le-fun-def less-fun-def)

end

lemma eval-INF [simp]:
  eval (∩ (f ' A)) = ∩ ((eval ◦ f) ' A)
by (simp add: image-comp)

lemma eval-SUP [simp]:
  eval (∪ (f ' A)) = ∪ ((eval ◦ f) ' A)
by (simp add: image-comp)

instantiation pred :: (type) complete-boolean-algebra
begin

definition − P = Pred (− eval P)

lemma eval-compl [simp]:
  eval (− P) = − eval P
by (simp add: uminus-pred-def)

definition P − Q = Pred (eval P − eval Q)

lemma eval-minus [simp]:
  eval (P − Q) = eval P − eval Q
by (simp add: minus-pred-def)

instance proof
  fix A::'a pred set set
  show ∪(Sup ' A) ≤ ∪(Inf ' {f ' A | f. ∀ Y∈A. f Y ∈ Y})
  proof (simp add: less-eq-pred-def Sup-fun-def Inf-fun-def, safe)
    fix w
    assume A: ∀ x∈A. ∃ f∈x. eval f w
    define F where F = (λ x. SOME f . f ∈ x ∧ eval f w)
    have [simp]: (∀ f∈ (F ' A). eval f w)
      by (metis (no-types, lifting) A F-def image-iff some-eq-ex)
    have (∃ f. F ' A = f ' A ∧ (∀ Y∈A. f Y ∈ Y)) ∧ (∀ f∈(F ' A). eval f w)
      using A by (simp, metis (no-types, lifting) F-def someI)
    from this show (∃ x. (∃ f. x = f ' A ∧ (∀ Y∈A. f Y ∈ Y)) ∧ (∀ f∈x. eval f w)
      by (rule exI [of - F ' A])
    qed
  qed (auto intro!: pred-eqI)
end

definition single :: 'a ⇒ 'a pred where
  single x = Pred ((=) x)

lemma eval-single [simp]:
  eval (single x) = (=) x
  by (simp add: single-def)

definition bind :: 'a pred ⇒ ('a ⇒ 'b pred) ⇒ 'b pred (infixl ≫ 70) where
  P ≫ f = (⨆ (f ' {x. eval P x}))

lemma eval-bind [simp]:
  eval (P ≫ f) = eval (⨆ (f ' {x. eval P x}))
  by (simp add: bind-def)

lemma bind-bind:
  (P ≫ Q) ≫ R = P ≫ (λx. Q x ≫ R)
  by (rule pred-eqI) auto

lemma bind-single:
  P ≫ single = P
  by (rule pred-eqI) auto

lemma single-bind:
  single x ≫ P = P x
  by (rule pred-eqI) auto

lemma bottom-bind:
  ⊥ ≫ P = ⊥
  by (rule pred-eqI) auto

lemma sup-bind:
  (P ⊔ Q) ≫ R = P ≫ R ⊔ Q ≫ R
  by (rule pred-eqI) auto

lemma Sup-bind:
  (⨆ A ≫ f) = ⨆ ((λx. x ≫ f) ' A)
  by (rule pred-eqI) auto

lemma pred-iffI:
  assumes \( \forall x. \text{eval} A x \implies \text{eval} B x \) and \( \forall x. \text{eval} B x \implies \text{eval} A x \)
  shows A = B
  using assms by (auto intro: pred-eqI)

lemma singleI: eval (single x) x
  by simp
lemma singleI-unit: eval (single ()) x by simp

lemma singleI: eval (single x) y \implies (y = x \implies P) \implies P by simp

lemma singleE': eval (single x) y \implies (x = y \implies P) \implies P by simp

lemma bindI: eval P x \implies eval (Q x) y \implies eval (P >\> Q) y by auto

lemma bindE: eval (R >\> Q) y \implies (\forall x. eval R x \implies eval (Q x) y \implies P) \implies P by auto

lemma botE: eval \bot x \implies P by auto

lemma supI1: eval A x \implies eval (A \⊔ B) x by auto

lemma supI2: eval B x \implies eval (A \⊔ B) x by auto

lemma supE: eval (A \⊔ B) x \implies (eval A x \implies P) \implies (eval B x \implies P) \implies P by auto

lemma single-not-bot [simp]: single x \neq \bot by (auto simp add: single-def bot-pred-def fun-eq-iff)

lemma not-bot: assumes A \neq \bot obtains x where eval A x using assms by (cases A) (auto simp add: bot-pred-def)

73.2 Emptiness check and definite choice

definition is-empty :: 'a pred \Rightarrow bool where
is-empty A \iff A = \perp

lemma is-empty-bot: is-empty \perp by (simp add: is-empty-def)

lemma not-is-empty-single: \neg is-empty (single x)
by (auto simp add: is-empty-def single-def bot-pred-def fun-eq-iff)

lemma is-empty-sup:
is-empty \((A \sqcup B)\) \iff is-empty A \land is-empty B  
by (auto simp add: is-empty-def)

definition singleton :: (unit \Rightarrow \,'a pred \Rightarrow \,'a) where
singleton default A = (if \(\exists!x. \text{eval A x}\) \then \text{THE x. eval A x else default ()}\) for default

lemma singleton-eqI:
\(\exists!x. \text{eval A x} = \Rightarrow \text{eval A x } = \Rightarrow \text{singleton default A } = x\) for default
by (auto simp add: singleton-def)

lemma eval-singletonI:
\(\exists!x. \text{eval A x} = \Rightarrow \text{eval A (singleton default A)} = A\) for default
proof –
assume assm: \(\exists!x. \text{eval A x}\)
then obtain x where x: \text{eval A x} ..
with assm have singleton default A = x by (rule singleton-eqI)
with x show thesis by simp

qed

lemma single-singleton:
\(\exists!x. \text{eval A x} = \Rightarrow \text{single (singleton default A)} = A\) for default
proof –
assume assm: \(\exists!x. \text{eval A x}\)
then have eval A (singleton default A) 
by (rule eval-singletonI)
moreover from assm have \(\forall x. \text{eval A x } =\Rightarrow \text{singleton default A } = x\) 
by (rule singleton-eqI)
ultimately have eval (single (singleton default A)) = eval A 
by (simp (no-asm-use) add: single-def fun-eq-iff) blast 
then have \(\forall x. \text{eval (singleton default A)} x = \text{eval A x}\) 
by simp 
then show thesis by (rule pred-eqI)

qed

lemma singleton-undefinedI:
\(\neg (\exists!x. \text{eval A x}) = \Rightarrow \text{singleton default A } = \text{default ()}\) for default
by (simp add: singleton-def)

lemma singleton-bot:
singleton default \(\bot\) = default () for default
by (auto simp add: bot-pred-def intro: singleton-undefinedI)

lemma singleton-single:
singleton default (single x) = x for default
by (auto simp add: intro: singleton-eqI singleI elim: singleE)
lemma singleton-sup-single-single:
  singleton default (single x ⊔ single y) = (if x = y then x else default ()) for default
proof (cases x = y)
case True then show ?thesis by (simp add: singleton-single)
next
case False
have eval (single x ⊔ single y) x
  and eval (single x ⊔ single y) y
  by (auto intro: supI1 supI2 singleI)
with False have ¬ (∃!z. eval (single x ⊔ single y) z)
  by blast
then have singleton default (single x ⊔ single y) = default ()
  by (rule singleton-undefinedI)
with False show ?thesis by simp
qed

lemma singleton-sup-aux:
  singleton default (A ⊔ B) = (if A = ⊥ then singleton default B
  else if B = ⊥ then singleton default A
  else singleton default
  (single (singleton default A) ⊔ single (singleton default B))) for default
proof (cases (∃!x. eval A x) ∧ (∃!y. eval B y))
case True then show ?thesis by (simp add: single-singleton)
next
case False
from False have A-or-B:
  singleton default A = default () ∨ singleton default B = default ()
  by (auto intro!: singleton-undefinedI)
then have rhs: singleton default
  (single (singleton default A) ⊔ single (singleton default B)) = default ()
  by (auto simp add: singleton-sup-single-single singleton-single)
from False have not-unique:
  ¬ (∃!x. eval A x) ∨ ¬ (∃!y. eval B y) by simp
show ?thesis proof (cases A ≠ ⊥ ∧ B ≠ ⊥)
case True
  then obtain a b where a: eval A a and b: eval B b
    by (blast elim: not-bot)
  with True not-unique have ¬ (∃!x. eval (A ⊔ B) x)
    by (auto simp add: sup-pred-def bot-pred-def)
  then have singleton default (A ⊔ B) = default ()
    by (rule singleton-undefinedI)
  with True rhs show ?thesis by simp
next
  case False then show ?thesis by auto
qed

lemma singleton-sup:
73.3 Derived operations

**definition if-pred :: bool ⇒ unit pred where**

if-pred-eq: if-pred b = (if b then single () else ⊥)

**definition holds :: unit pred ⇒ bool where**

holds-eq: holds P = eval P ()

**definition not-pred :: unit pred ⇒ unit pred where**

not-pred-eq: not-pred P = (if eval P () then ⊥ else single ()

**lemma if-predI: P =⇒ eval (if-pred P) ()**

unfolding if-pred-eq by (auto intro: singleI)

**lemma if-predE: eval (if-pred b) x =⇒ (b =⇒ x = () =⇒ P) =⇒ P**

unfolding if-pred-eq by (cases b) (auto elim: botE)

**lemma not-predI: ¬ P =⇒ eval (not-pred (Pred (λu. P))) ()**

unfolding not-pred-eq by (auto intro: singleI)

**lemma not-predI’: ¬ eval P () =⇒ eval (not-pred P) ()**

unfolding not-pred-eq by (auto intro: singleI)

**lemma not-predE: eval (not-pred (Pred (λu. P))) x =⇒ (¬ P =⇒ thesis) =⇒ thesis**

unfolding not-pred-eq by (auto split: if-split-asm elim: botE)

**lemma not-predE’: eval (not-pred P) x =⇒ (¬ eval P x =⇒ thesis) =⇒ thesis**

unfolding not-pred-eq by (auto split: if-split-asm elim: botE)

**lemma f () = False ∨ f () = True**

by simp

**lemma closure-of-bool-cases [no-atp]:**

fixes f :: unit ⇒ bool

assumes f = (λu. False) =⇒ P f

assumes f = (λu. True) =⇒ P f

shows P f

proof –

have f = (λu. False) ∨ f = (λu. True)

apply (cases f ())

apply (rule disjI2)
apply (rule ext)
apply (simp add: unit-eq)
apply (rule disjI1)
apply (rule ext)
apply (simp add: unit-eq)
done
from this assms show ?thesis by blast
qed

lemma unit-pred-cases:
  assumes P ⊥
  assumes P (single ())[118x644]
says P Q
using assms unfolding bot-pred-def bot-fun-def bot-bool-def empty-def single-def
proof (cases Q)
  fix f
  assume P (Pred (λu. False)) P (Pred (λu. () = u))
  then have P (Pred f)
    by (cases - f rule: closure-of-bool-cases) simp-all
  moreover assume Q = Pred f
  ultimately show P Q by simp
qed

lemma holds-if-pred:
  holds (if-pred b) = b
unfolding if-pred-eq holds-eq
by (cases b) (auto intro: singleI elim: botE)

lemma if-pred-holds:
  if-pred (holds P) = P
unfolding if-pred-eq holds-eq
by (rule unit-pred-cases) (auto intro: singleI elim: botE)

lemma is-empty-holds:
  is-empty P ↔ ¬ holds P
unfolding is-empty-def holds-eq
by (rule unit-pred-cases) (auto elim: botE intro: singleI)

definition map :: ('a ⇒ 'b) ⇒ 'a pred ⇒ 'b pred where
  map f P = P ≫ (single o f)

lemma eval-map [simp]:
  eval (map f P) = (⊔ x∈{x. eval P x}. (λy. f x = y))
by (simp add: map-def comp-def image-comp)

functor map: map
  by (rule ext, rule pred-eqI, auto)
73.4 Implementation

datatype (plugins only: code extraction) (dead 'a) seq =
  Empty
| Insert 'a 'a pred
| Join 'a pred 'a seq

primrec pred-of-seq :: 'a seq ⇒ 'a pred where
  pred-of-seq Empty = ⊥
| pred-of-seq (Insert x P) = single x ⊔ P
| pred-of-seq (Join P xq) = P ⊔ pred-of-seq xq

definition Seq :: (unit ⇒ 'a seq) ⇒ 'a pred where
  Seq f = pred-of-seq (f ())

code-datatype Seq

primrec member :: 'a seq ⇒ 'a ⇒ bool where
  member Empty x ←→ False
| member (Insert y P) x ←→ x = y ∨ eval P x
| member (Join P xq) x ←→ eval P x ∨ member xq x

lemma eval-member:
  member xq = eval (pred-of-seq xq)
proof (induct xq)
  case Empty show ?case
  by (auto simp add: fun-eq-iff elim: botE)
next
  case Insert show ?case
  by (auto simp add: fun-eq-iff elim: supE singleE intro: supI1 supI2 singleI)
next
  case Join then show ?case
  by (auto simp add: fun-eq-iff elim: supE intro: supI1 supI2)
qed

lemma eval-code [code]: eval (Seq f) = member (f ())
unfolding Seq-def by (rule sym, rule eval-member)

lemma single-code [code]:
  single x = Seq (λu. Insert x ⊥)
unfolding Seq-def by simp

primrec apply :: ('a ⇒ 'b pred) ⇒ 'a seq ⇒ 'b seq where
  apply f Empty = Empty
| apply f (Insert x P) = Join (f x) (Join (P ≪= f) Empty)
| apply f (Join P xq) = Join (P ≪= f) (apply f xq)

lemma apply-bind:
  pred-of-seq (apply f xq) = pred-of-seq xq ≪= f
proof (induct xq)
case Empty show ?case
  by (simp add: bottom-bind)
next
case Insert show ?case
  by (simp add: single-bind sup-bind)
next
case Join then show ?case
  by (simp add: sup-bind)
qed

lemma bind-code [code]:
  Seq g >>= f = Seq (λu. apply f (g ()))
unfolding Seq-def by (rule sym, rule apply-bind)

lemma bot-set-code [code]:
  ⊥ = Seq (λu. Empty)
unfolding Seq-def by simp

primrec adjunct :: 'a pred ⇒ 'a seq ⇒ 'a seq where
  adjunct P Empty = Join P Empty
| adjunct P (Insert x Q) = Insert x (Q ⊔ P)
| adjunct P (Join Q xq) = Join Q (adjunct P xq)

lemma adjunct-sup:
  pred-of-seq (adjunct P xq) = P ⊔ pred-of-seq xq
by (induct xq) (simp-all add: sup-assoc sup-commute sup-left-commute)

lemma sup-code [code]:
  Seq f ⊔ Seq g = Seq (λu. case f ()
  of Empty ⇒ g ()
      | Insert x P ⇒ Insert x (P ⊔ Seq g)
      | Join P xq ⇒ adjunct (Seq g) (Join P xq))
proof (cases f ())
case Empty
  thus ?thesis
  unfolding Seq-def by (simp add: sup-commute [of ⊥])
next
case Insert
  thus ?thesis
  unfolding Seq-def by (simp add: sup-assoc)
next
case Join
  thus ?thesis
  unfolding Seq-def
  by (simp add: adjunct-sup sup-assoc sup-commute sup-left-commute)
qed

primrec contained :: 'a seq ⇒ 'a pred ⇒ bool where
  contained Empty Q ←→ True
lemma single-less-eq-eval:
  single x ≤ P ←→ eval P x
  by (auto simp add: less-eq-pred-def le-fun-def)

lemma contained-less-eq:
  contained xq Q ←→ pred-of-seq xq ≤ Q
  by (induct xq) (simp-all add: single-less-eq-eval)

lemma less-eq-pred-code [code]:
  Seq f ≤ Q = (case f ()
  of Empty ⇒ True
  | Insert x P ⇒ eval Q x ∧ P ≤ Q
  | Join P xq ⇒ P ≤ Q ∧ contained xq Q)
  by (cases f ())(simp-all add: Seq-def single-less-eq-eval contained-less-eq)

instantiation pred :: (type) equal
begin

definition equal-pred
  where [simp]: HOL.equal P Q ←→ P = (Q :: 'a pred)

instance by standard simp
end

lemma [code]:
  HOL.equal P Q ←→ P ≤ Q ∧ Q ≤ P for P Q :: 'a pred
  by auto

lemma [code nbe]:
  HOL.equal P P ←→ True for P :: 'a pred
  by (fact equal-refl)

lemma [code]:
case-pred f P = f (eval P)
  by (fact pred.case-eq-if)

lemma [code]:
  rec-pred f P = f (eval P)
  by (cases P) simp

inductive eq :: 'a ⇒ 'a ⇒ bool where eq x x

lemma eq-is-eq: eq x y ≡ (x = y)
  by (rule eq-reflection) (auto intro: eq.intros elim: eq.cases)
primrec null :: 'a seq ⇒ bool where
  null Empty ←→ True
  | null (Insert x P) ←→ False
  | null (Join P xq) ←→ is-empty P ∧ null xq

lemma null-is-empty:
  null xq ←→ is-empty (pred-of-seq xq)
by (induct xq) (simp-all add: is-empty-bot not-is-empty-single is-empty-sup)

lemma is-empty-code [code]:
  is-empty (Seq f) ←→ null (f ())
by (simp add: null-is-empty Seq-def)

primrec the-only :: (unit ⇒ 'a) ⇒ 'a seq ⇒ 'a where
  the-only Default Empty = default () for default
  | the-only Default (Insert x P) =
  (if is-empty P then x else let y = singleton default P in if x = y then x else default ()) for default
  | the-only Default (Join P xq) =
  (if is-empty P then the-only default xq else if null xq then singleton default P
  else let x = singleton default P; y = the-only default xq in
  if x = y then x else default ()) for default

lemma the-only-singleton:
  the-only default xq = singleton default (pred-of-seq xq) for default
by (induct xq)
  (auto simp add: singleton-bot singleton-single is-empty-def
  null-is-empty Let-def singleton-sup)

lemma singleton-code [code]:
  singleton default (Seq f) =
  (case f () of
    Empty ⇒ default ()
  | Insert x P ⇒ if is-empty P then x
  else let y = singleton default P in
  if x = y then x else default ()
  | Join P xq ⇒ if is-empty P then the-only default xq
  else if null xq then singleton default P
  else let z = singleton default P; y = the-only default xq in
  if x = y then x else default ()) for default
by (cases f ());
  (auto simp add: Seq-def the-only-singleton is-empty-def
  null-is-empty singleton-bot singleton-single singleton-sup Let-def)

definition the :: 'a pred ⇒ 'a where
  the A = (THE x. eval A x)

lemma the-eqI:
THEORY “Predicate”

(THE \( x \), eval \( P \) \( x \)) = \( x \) \( \Rightarrow \) the \( P \) = \( x \)
by \( \text{simp add: the-def} \)

lemma \( \text{the-eq [code]: the A = singleton (\( \lambda x. \text{Code.abort (STR "not-unique") (\( \lambda \). the A) A} \)) \)
by \( \text{rule the-eqI} \) \( \text{simp add: singleton-def the-def} \)

code-reflect Predicate
datatypes pred = Seq and seq = Empty | Insert | Join

ML |
signature PREDICATE =
sig
  val anamorph: ('a -> ('b * 'a option) -> int -> 'a -> 'b list * 'a
datatype 'a pred = Seq of (unit -> 'a seq)
and 'a seq = Empty | Insert of 'a * 'a pred | Join of 'a pred * 'a seq
val map: ('a -> 'b) -> 'a pred -> 'b pred
val yield: 'a pred -> ('a * 'a pred) option
val yieldn: int -> 'a pred -> 'a list * 'a pred[end];
structure Predicate : PREDICATE =
struct
  fun anamorph f k x =
    (if k = 0 then ([], x)
    else case f x
      of NONE => ([], x)
      | SOME (v, y) => let
        val k' = k - 1;
        val (vs, z) = anamorph f k' y
      in (v :: vs, z) end);
datatype pred = datatype Predicate.pred
datatype seq = datatype Predicate.seq
fun map f = @\{code Predicate.map\} f;

fun yield (Seq f) = next (f ());
and next Empty = NONE
| next (Insert (x, P)) = SOME (x, P)
| next (Join (P, xq)) = (case yield P
  of NONE => next xq
  | SOME (x, Q) => SOME (x, Seq (fn - => Join (Q, xq))));

fun yieldn k = anamorph yield k;

end;
Conversion from and to sets

**Definition** `pred-of-set` :: 'a set ⇒ 'a pred

\[
\text{pred-of-set } = \text{Pred} \circ (\lambda A. x. x \in A)
\]

**Lemma** `eval-pred-of-set` [simp]:
\[
\text{eval (pred-of-set A) } x \leftrightarrow x \in A
\]
by (simp add: pred-of-set-def)

**Definition** `set-of-pred` :: 'a pred ⇒ 'a set


\[
\text{set-of-pred } = \text{Collect} \circ \text{eval}
\]

**Lemma** `member-set-of-pred` [simp]:
\[
x \in \text{set-of-pred } P \leftrightarrow \text{Predicate.eval } P x
\]
by (simp add: set-of-pred-def)

**Definition** `set-of-seq` :: 'a seq ⇒ 'a set

\[
\text{set-of-seq } = \text{set-of-pred } \circ \text{pred-of-seq}
\]

**Lemma** `member-set-of-seq` [simp]:
\[
x \in \text{set-of-seq } xq \leftrightarrow \text{Predicate.member } xq x
\]
by (simp add: set-of-seq-def eval-member)

**Lemma** `of-pred-code` [code]:
\[
\begin{align*}
\text{set-of-pred } (\text{Predicate.Seq } f) &= (\text{case } f () \text{ of}\) \\
& \quad \mid \text{Predicate.Empty } \Rightarrow \{\} \\
& \quad \mid \text{Predicate.Insert } x P \Rightarrow \text{insert } x \text{ (set-of-pred } P) \\
& \quad \mid \text{Predicate.Join } P xq \Rightarrow \text{set-of-pred } P \cup \text{set-of-seq } xq
\end{align*}
\]
by (auto split: seq.split simp add: eval-code)

**Lemma** `of-seq-code` [code]:
\[
\begin{align*}
\text{set-of-seq } \text{Predicate.Empty} &= \{\} \\
\text{set-of-seq } (\text{Predicate.Insert } x P) &= \text{insert } x \text{ (set-of-pred } P) \\
\text{set-of-seq } (\text{Predicate.Join } P xq) &= \text{set-of-pred } P \cup \text{set-of-seq } xq
\end{align*}
\]
by auto

Lazy Evaluation of an indexed function

**Function** `iterate-upto` :: (natural ⇒ 'a) ⇒ natural ⇒ natural ⇒ 'a Predicate.pred

\[
\begin{align*}
\text{iterate-upto } f n m &= \text{Predicate.Seq } (\%u. \text{ if } n > m \text{ then Predicate.Empty} \\
& \quad \quad \quad \text{ else Predicate.Insert } (f n) \text{ (iterate-upto } f (n + 1) \text{ } m))
\end{align*}
\]
by pat-completeness auto

**Termination** by (relation measure (\% (f, n, m). nat-of-natural (m + 1 - n)))
(auto simp add: less-natural-def)

Misc

**Declare** `Inf-set-fold` [where 'a = 'a Predicate.pred, code]

**Declare** `Sup-set-fold` [where 'a = 'a Predicate.pred, code]
lemma pred-of-set-fold-sup:
assumes finite A
shows pred-of-set A = Finite-Set.fold sup bot (Predicate.single ' A) (is ?lhs = ?rhs)
proof (rule sgm)
  interpret comp-fun-idem sup :: 'a Predicate.pred ⇒ 'a Predicate.pred ⇒ 'a Predicate.pred
  by (fact comp-fun-idem-sup)
from ⟨finite A⟩ show ?rhs = ?lhs by (induct A) (auto intro!: pred-eqI)
qed

lemma pred-of-set-set-fold-sup:
pred-of-set (set xs) = fold sup (List.map Predicate.single xs) bot
proof -
  interpret comp-fun-idem sup :: 'a Predicate.pred ⇒ 'a Predicate.pred ⇒ 'a Predicate.pred
  by (fact comp-fun-idem-sup)
  show ?thesis by (simp add: pred-of-set-fold-sup fold-set-fold [symmetric])
qed

lemma pred-of-set-set-foldr-sup [code]:
pred-of-set (set xs) = foldr sup (List.map Predicate.single xs) bot
by (simp add: pred-of-set-set-fold-sup ac-simps foldr-fold fun-eq-iff)

no-notation
  bind (infixl ≫= 70)

hide-type (open) pred seq
hide-const (open) Pred eval single bind is-empty singleton if-pred not-pred holds
  Empty Insert Join Seq member pred-of-seq apply adjunct null the-only eq map the
  iterate-upto
hide-fact (open) null-def member-def

end

74 Lazy sequences

theory Lazy-Sequence
imports Predicate
begin

74.1 Type of lazy sequences

datatype (plugins only: code extraction) (dead 'a) lazy-sequence =
lazy-sequence-of-list 'a list
primrec list-of-lazy-sequence :: 'a lazy-sequence ⇒ 'a list
where
  list-of-lazy-sequence (lazy-sequence-of-list xs) = xs

lemma lazy-sequence-of-list-of-lazy-sequence [simp]:
  lazy-sequence-of-list (list-of-lazy-sequence xq) = xq
  by (cases xq) simp-all

lemma lazy-sequence-eqI:
  list-of-lazy-sequence xq = list-of-lazy-sequence yq ⟹ xq = yq
  by (cases xq, cases yq) simp

lemma lazy-sequence-eq-iff:
  xq = yq ⟷ list-of-lazy-sequence xq = list-of-lazy-sequence yq
  by (auto intro: lazy-sequence-eqI)

lemma case-lazy-sequence [simp]:
  case-lazy-sequence f xq = f (list-of-lazy-sequence xq)
  by (cases xq) auto

lemma rec-lazy-sequence [simp]:
  rec-lazy-sequence f xq = f (list-of-lazy-sequence xq)
  by (cases xq) auto

definition Lazy-Sequence :: (unit ⇒ ('a × 'a lazy-sequence) option) ⇒ 'a lazy-sequence
where
  Lazy-Sequence f = lazy-sequence-of-list (case f () of
    None ⇒ []
    | Some (x, xq) ⇒ x # list-of-lazy-sequence xq)

code_datatype Lazy-Sequence

declare list-of-lazy-sequence.simps [code del]
declare lazy-sequence.case [code del]
declare lazy-sequence.rec [code del]

lemma list-of-Lazy-Sequence [simp]:
  list-of-lazy-sequence (Lazy-Sequence f) = (case f () of
    None ⇒ []
    | Some (x, xq) ⇒ x # list-of-lazy-sequence xq)
  by (simp add: Lazy-Sequence-def)

definition yield :: 'a lazy-sequence ⇒ ('a × 'a lazy-sequence) option
where
  yield xq = (case list-of-lazy-sequence xq of
    [] ⇒ None
    | x # xs ⇒ Some (x, lazy-sequence-of-list xs))

lemma yield-Seq [simp, code]:
yield (Lazy-Sequence f) = f ()
by (cases f ()) (simp-all add: yield-def split-def)

lemma case-yield-eq [simp]: case-option g h (yield xq) =
case-list g (λx. curry h x o lazy-sequence-of-list) (list-of-lazy-sequence xq)
by (cases list-of-lazy-sequence xq) (simp-all add: yield-def)

lemma equal-lazy-sequence-code [code]:
HOL.equal xq yq = (case (yield xq, yield yq) of
  (None, None) ⇒ True
  | (Some (x, xq'), Some (y, yq')) ⇒ HOL.equal x y ∧ HOL.equal xq yq
  | - ⇒ False)
by (simp-all add: lazy-sequence-eq-iff equal-eq split: list.splits)

lemma [code nbe]:
HOL.equal (x :: 'a lazy-sequence) x ←→ True
by (fact equal-refl)

definition empty :: 'a lazy-sequence
where
empty = lazy-sequence-of-list []

lemma list-of-lazy-sequence-empty [simp]:
list-of-lazy-sequence empty = []
by (simp add: empty-def)

lemma empty-code [code]:
empty = Lazy-Sequence (λ-. None)
by (simp add: lazy-sequence-eq-iff)

definition single :: 'a ⇒ 'a lazy-sequence
where
single x = lazy-sequence-of-list [x]

lemma list-of-lazy-sequence-single [simp]:
list-of-lazy-sequence (single x) = [x]
by (simp add: single-def)

lemma single-code [code]:
single x = Lazy-Sequence (λ-. Some (x, empty))
by (simp add: lazy-sequence-eq-iff)

definition append :: 'a lazy-sequence ⇒ 'a lazy-sequence ⇒ 'a lazy-sequence
where
append xq yq = lazy-sequence-of-list (list-of-lazy-sequence xq @ list-of-lazy-sequence yq)

lemma list-of-lazy-sequence-append [simp]:
list-of-lazy-sequence (append xq yq) = list-of-lazy-sequence xq @ list-of-lazy-sequence yq
lemma append-code [code]:
append $xq$ $yq$ = Lazy-Sequence $(\lambda\. \text{case yield } xq \text{ of}
| \text{None} \Rightarrow \text{yield } yq
| \text{Some } (x, xq') \Rightarrow \text{Some } (x, \text{append } xq' yq))$
by (simp-all add: lazy-sequence-eq-iff split: list.splits)

definition map :: '('a ⇒ 'b lazy-sequence ⇒ 'b lazy-sequence
where
map $f$ $xq$ = lazy-sequence-of-list (List.map $f$ (list-of-lazy-sequence $xq$))

lemma list-of-lazy-sequence-map [simp]:
list-of-lazy-sequence (map $f$ $xq$) = List.map $f$ (list-of-lazy-sequence $xq$)
by (simp add: map-def)

lemma map-code [code]:
map $f$ $xq$ = Lazy-Sequence $(\lambda\. \text{map-option } (\lambda(x, xq'). (f x, map f xq')) \text{ (yield } xq))$
by (simp-all add: lazy-sequence-eq-iff split: list.splits)

definition flat :: 'a lazy-sequence lazy-sequence ⇒ 'a lazy-sequence
where
flat $xqq$ = lazy-sequence-of-list (concat (List.map list-of-lazy-sequence (list-of-lazy-sequence $xqq$)))

lemma list-of-lazy-sequence-flat [simp]:
list-of-lazy-sequence (flat $xqq$) = concat (List.map list-of-lazy-sequence (list-of-lazy-sequence $xqq$))
by (simp add: flat-def)

lemma flat-code [code]:
flat $xqq$ = Lazy-Sequence $(\lambda\. \text{case yield } xqq \text{ of}
| \text{None} \Rightarrow \text{None}
| \text{Some } (xq, xqq') \Rightarrow \text{yield } (\text{append } xq (\text{flat } xqq'))$
by (simp add: lazy-sequence-eq-iff split: list.splits)

definition bind :: 'a lazy-sequence ⇒ ('a ⇒ 'b lazy-sequence) ⇒ 'b lazy-sequence
where
bind $xq$ $f$ = flat (map $f$ $xq$)

definition if-seq :: bool ⇒ unit lazy-sequence
where
if-seq $b$ = (if $b$ then single () else empty)

definition those :: 'a option lazy-sequence ⇒ 'a lazy-sequence option
where
those $xq$ = map-option lazy-sequence-of-list (List.those (list-of-lazy-sequence $xq$))
function iterate-upto :: (natural ⇒ 'a) ⇒ natural ⇒ natural ⇒ 'a lazy-sequence
where
iterate-upto f n m = Lazy-Sequence (λ-. if n > m then None else Some (f n, iterate-upto f (n + 1) m))

by pat-completeness auto

termination by (relation measure (λ(f, n, m). nat-of-natural (m + 1 − n)))
(auto simp add: less-natural-def)

definition not-seq :: unit lazy-sequence ⇒ unit lazy-sequence
where
not-seq xq = (case yield xq of
  None ⇒ single ()
  | Some (((), xq)) ⇒ empty)

74.2 Code setup

code-reflect Lazy-Sequence
datatypes lazy-sequence = Lazy-Sequence

ML:
signature LAZY-SEQUENCE =
sig
datatype 'a lazy-sequence = Lazy-Sequence of (unit ⇒ ('a * 'a Lazy-Sequence.lazy-sequence) option)
  val map: ('a ⇒ 'b) ⇒ 'a lazy-sequence ⇒ 'b lazy-sequence
  val yield: 'a lazy-sequence ⇒ ('a * 'a lazy-sequence) option
  val yieldn: int ⇒ 'a lazy-sequence ⇒ 'a list * 'a lazy-sequence
end;

structure Lazy-Sequence : LAZY-SEQUENCE =
struct
datatype lazy-sequence = datatype Lazy-Sequence.lazy-sequence;

fun map f = @$\{\text{code Lazy-Sequence.map}\} f$;

fun yield P = @$\{\text{code Lazy-Sequence.yield}\} P$;

fun yieldn k = Predicate.anamorph yield k;
end;
74.3 Generator Sequences

74.3.1 General lazy sequence operation

definition product :: 'a lazy-sequence ⇒ 'b lazy-sequence ⇒ ('a × 'b) lazy-sequence
where
  product s1 s2 = bind s1 (λa. bind s2 (λb. single (a, b)))

74.3.2 Small lazy typeclasses

class small-lazy =
  fixes small-lazy :: natural ⇒ 'a lazy-sequence

instantiation unit :: small-lazy
begin
  definition small-lazy d = single ()
  instance ..
end

instantiation int :: small-lazy
begin
  maybe optimise this expression -¿ append (single x) xs == cons x xs Performance difference?
  function small-lazy' :: int ⇒ int ⇒ int lazy-sequence
where
  small-lazy' d i = (if d < i then empty
    else append (single i) (small-lazy d (i + 1)))
  by pat-completeness auto

termination
  by (relation measure (%(d, i). nat (d + I - i))) auto

definition small-lazy d = small-lazy' (int (nat-of-natural d)) (− (int (nat-of-natural d)))

instance ..
end

instantiation prod :: (small-lazy, small-lazy) small-lazy
begin
  definition small-lazy d = product (small-lazy d) (small-lazy d)
  instance ..
end

instantiation list :: (small-lazy) small-lazy
begin

fun small-lazy-list :: natural ⇒ 'a list lazy-sequence
where
small-lazy-list d = append (single [])
  (if d > 0 then bind (product (small-lazy (d - 1)))
   (small-lazy (d - 1))) (λ(x, xs). single (x # xs)) else empty)

instance ..
end

74.4 With Hit Bound Value
assuming in negative context
type-synonym 'a hit-bound-lazy-sequence = 'a option lazy-sequence
definition hit-bound :: 'a hit-bound-lazy-sequence
where
hit-bound = Lazy-Sequence (λ- . Some (None, empty))

lemma list-of-lazy-sequence-hit-bound [simp];
  list-of-lazy-sequence hit-bound = [None]
  by (simp add: hit-bound-def)
definition hb-single :: 'a ⇒ 'a hit-bound-lazy-sequence
where
hb-single x = Lazy-Sequence (λ- . Some (Some x, empty))
definition hb-map :: ('a ⇒ 'b) ⇒ 'a hit-bound-lazy-sequence ⇒ 'b hit-bound-lazy-sequence
where
hb-map f xq = map (map-option f) xq
lemma hb-map-code [code]:
  hb-map f xq =
  Lazy-Sequence (λ- . map-option (λ(x, xq'). (map-option f x, hb-map f xq')) (yield xq))
  using map-code [of map-option f xq] by (simp add: hb-map-def)
definition hb-flat :: 'a hit-bound-lazy-sequence hit-bound-lazy-sequence ⇒ 'a hit-bound-lazy-sequence
where
hb-flat xqq = lazy-sequence-of-list (concat
  (List.map ((λx. case x of None ⇒ [None] | Some xs ⇒ xs) o map-option
    list-of-lazy-sequence) (list-of-lazy-sequence xqq)))
lemma list-of-lazy-sequence-hb-flat [simp]:
  list-of-lazy-sequence (hb-flat xqq) =
  concat (List.map ((x. case x of None ⇒ [None] | Some xs ⇒ xs) o map-option
  list-of-lazy-sequence) (list-of-lazy-sequence xqq))
  by (simp add: hb-flat-def)

lemma hb-flat-code [code]:
  hb-flat xqq = Lazy-Sequence (λ-. case yield xqq of
    None ⇒ None
    | Some (xq, xqq') ⇒ yield
      (append (case xq of None ⇒ hit-bound | Some xq ⇒ xq) (hb-flat xqq')))
  by (simp add: lazy-sequence-eq-iff split: list.splits option.splits)

definition hb-bind :: 'a hit-bound-lazy-sequence ⇒ ('a ⇒ 'b hit-bound-lazy-sequence)
⇒ 'b hit-bound-lazy-sequence
where
  hb-bind xq f = hb-flat (hb-map f xq)

definition hb-if-seq :: bool ⇒ unit hit-bound-lazy-sequence
where
  hb-if-seq b = (if b then hb-single () else empty)

definition hb-not-seq :: unit hit-bound-lazy-sequence ⇒ unit lazy-sequence
where
  hb-not-seq xq = (case yield xq of
    None ⇒ single ()
    | Some (x, xq) ⇒ empty)

hide-const (open) yield empty single append flat map bind
if-seq those iterate-upto not-seq product

hide-fact (open) yield-def empty-def def single-def append-def flat-def map-def bind-def
if-seq-def those-def not-seq-def product-def

end

75 Depth-Limited Sequences with failure element

theory Limited-Sequence imports Lazy-Sequence
begin

75.1 Depth-Limited Sequence

type-synonym 'a dseq = natural ⇒ bool ⇒ 'a lazy-sequence option

definition empty :: 'a dseq
where
  empty = (λ - -. Some Lazy-Sequence.empty)
definition single :: 'a ⇒ 'a dseq
where
  single x = (λ- -. Some (Lazy-Sequence.single x))

definition eval :: 'a dseq ⇒ natural ⇒ bool ⇒ 'a lazy-sequence option
where
  [simp]: eval f i pol = f i pol

definition yield :: 'a dseq ⇒ natural ⇒ bool ⇒ ('a × 'a dseq) option
where
  yield f i pol = (case eval f i pol of
      None ⇒ None | Some s ⇒ (map-option o apsnd) (λr -. Some r) (Lazy-Sequence.yield s))

definition map-seq :: ('a ⇒ 'b dseq) ⇒ 'a lazy-sequence ⇒ 'b dseq
where
  map-seq f xq i pol = map-option Lazy-Sequence.flat (Lazy-Sequence.those (Lazy-Sequence.map (λx. f x i pol) xq))

lemma map-seq-code [code]:
  map-seq f xq i pol = (case Lazy-Sequence.yield xq of
                   None ⇒ Some Lazy-Sequence.empty | Some (x, xq') ⇒ (case eval (f x) i pol of
                       None ⇒ None | Some yq ⇒ (case map-seq f xq' i pol of
                           None ⇒ None | Some zq ⇒ Some (Lazy-Sequence.append yq zq))))
  by (cases xq)
    (auto simp add: map-seq-def Lazy-Sequence.those-def lazy-sequence-eq-iff split: list.splits option.splits)

definition bind :: 'a dseq ⇒ ('a ⇒ 'b dseq) ⇒ 'b dseq
where
  bind x f = (λi pol.
      if i = 0 then
        (if pol then Some Lazy-Sequence.empty else None)
      else
        (case x (i − 1) pol of
          None ⇒ None | Some xq ⇒ map-seq f xq i pol))

definition union :: 'a dseq ⇒ 'a dseq ⇒ 'a dseq
where
  union x y = (λi pol. case (x i pol, y i pol) of
    (Some xq, Some yq) ⇒ Some (Lazy-Sequence.append xq yq) | - ⇒ None)

definition if-seq :: bool ⇒ unit dseq
THEORY “Limited-Sequence”

where
if-seq b = (if b then single () else empty)

definition not-seq :: unit dseq ⇒ unit dseq
where
not-seq x = (λi. case x i of
  None ⇒ Some Lazy-Sequence.empty
| Some xq ⇒ (case Lazy-Sequence.yield xq of
    None ⇒ Some (Lazy-Sequence.single ()))
  | Some _ ⇒ Some (Lazy-Sequence.empty)))

definition map :: ('a ⇒ 'b) ⇒ 'a dseq ⇒ 'b dseq
where
map f g = (λi. case g i of
  None ⇒ None
| Some xq ⇒ Some (Lazy-Sequence.map f xq))

75.2 Positive Depth-Limited Sequence

type-synonym 'a pos-dseq = natural ⇒ 'a Lazy-Sequence.lazy-sequence

definition pos-empty :: 'a pos-dseq
where
pos-empty = (λi. Lazy-Sequence.empty)

definition pos-single :: 'a ⇒ 'a pos-dseq
where
pos-single x = (λi. Lazy-Sequence.single x)

definition pos-bind :: 'a pos-dseq ⇒ ('a ⇒ 'b pos-dseq) ⇒ 'b pos-dseq
where
pos-bind x f = (λi. Lazy-Sequence.bind (x i) (λa. f a i))

definition pos-decr-bind :: 'a pos-dseq ⇒ ('a ⇒ 'b pos-dseq) ⇒ 'b pos-dseq
where
pos-decr-bind x f = (λi.
  if i = 0 then
    Lazy-Sequence.empty
  else
    Lazy-Sequence.bind (x (i - 1)) (λa. f a i))

definition pos-union :: 'a pos-dseq ⇒ 'a pos-dseq ⇒ 'a pos-dseq
where
pos-union xq yq = (λi. Lazy-Sequence.append (xq i) (yq i))

definition pos-if-seq :: bool ⇒ unit pos-dseq
where
pos-if-seq b = (if b then pos-single () else pos-empty)
**THEORY** “Limited-Sequence” 1458

**definition** pos-iterate-upto :: (natural ⇒ 'a) ⇒ natural ⇒ natural ⇒ 'a pos-dseq

where

pos-iterate-upto f n m = (λi. Lazy-Sequence.iterate-upto f n m)

**definition** pos-map :: ('a ⇒ 'b) ⇒ 'a pos-dseq ⇒ 'b pos-dseq

where

pos-map f xq = (λi. Lazy-Sequence.map f (xq i))

### 75.3 Negative Depth-Limited Sequence

**type-synonym** 'a neg-dseq = natural ⇒ 'a Lazy-Sequence.hit-bound-lazy-sequence

**definition** neg-empty :: 'a neg-dseq

where

neg-empty = (λi. Lazy-Sequence.empty)

**definition** neg-single :: 'a ⇒ 'a neg-dseq

where

neg-single x = (λi. Lazy-Sequence.hb-single x)

**definition** neg-bind :: 'a neg-dseq ⇒ ('a ⇒ 'b neg-dseq) ⇒ 'b neg-dseq

where

neg-bind x f = (λi. hb-bind (x i) (λa. f a i))

**definition** neg-decr-bind :: 'a neg-dseq ⇒ ('a ⇒ 'b neg-dseq) ⇒ 'b neg-dseq

where

neg-decr-bind x f = (λi.
    if i = 0 then
        Lazy-Sequence.hit-bound
    else
        hb-bind (x (i - 1)) (λa. f a i))

**definition** neg-union :: 'a neg-dseq ⇒ 'a neg-dseq ⇒ 'a neg-dseq

where

neg-union x y = (λi. Lazy-Sequence.append (x i) (y i))

**definition** neg-if-seq :: bool ⇒ unit neg-dseq

where

neg-if-seq b = (if b then neg-single () else neg-empty)

**definition** neg-iterate-upto

where

neg-iterate-upto f n m = (λi. Lazy-Sequence.iterate-upto (λi. Some (f i)) n m)

**definition** neg-map :: ('a ⇒ 'b) ⇒ 'a neg-dseq ⇒ 'b neg-dseq

where

neg-map f xq = (λi. Lazy-Sequence.hb-map f (xq i))
75.4 Negation

**definition** `pos-not-seq :: unit dseq ⇒ unit pos-dseq`

**where**

`pos-not-seq xq = (λi. Lazy-Sequence.hb-not-seq (xq (3 * i)))`

**definition** `neg-not-seq :: unit pos-dseq ⇒ unit neg-dseq`

**where**

`neg-not-seq xq = (λi. case Lazy-Sequence.yield (xq i) of
  None ⇒ Lazy-Sequence.hb-single ()
  | Some (((), xq) ⇒ Lazy-Sequence.empty))`

**ML**

```ml
signature LIMITED-SEQUENCE =
  sig
    type 'a dseq = Code-Natural.¬ bool ⇒ 'a Lazy-Sequence.lazy-sequence option
    val map : ('a ⇒ 'b) ⇒ 'a dseq ⇒ 'b dseq
    val yield : 'a dseq ⇒ Code-Natural.¬ bool ⇒ ('a * 'a dseq) option
    val yieldn : int ⇒ 'a dseq ⇒ Code-Natural.¬ bool ⇒ 'a list * 'a dseq
  end;

structure Limited-Sequence : LIMITED-SEQUENCE =
  struct
    type 'a dseq = Code-Natural.¬ bool ⇒ 'a Lazy-Sequence.lazy-sequence option
    fun map f = @{code Limited-Sequence.map} f;
    fun yield f = @{code Limited-Sequence.yield} f;
    fun yieldn n f i pol = (case f i pol of
      NONE ⇒ ([], fn - ⇒ fn - ⇒ NONE)
      | SOME s ⇒ let val (xs, s') = Lazy-Sequence.yieldn n s in (xs, fn - ⇒ fn - ⇒ SOME s') end);
  end;
```

**code-reserved** Eval Limited-Sequence

**hide-const (open)** yield empty single eval map-seq bind union if-seq not-seq map
pos-empty pos-single pos-bind pos-decr-bind pos-union pos-if-seq pos-iterate-upto
pos-not-seq pos-map
neg-empty neg-single neg-bind neg-decr-bind neg-union neg-if-seq neg-iterate-upto
neg-not-seq neg-map

end

76  Term evaluation using the generic code generator

theory Code-Evaluation
imports Typerep Limited-Sequence
keywords value :: diag
begin

76.1  Term representation

76.1.1  Terms and class term-of
datatype (plugins only: extraction) term = dummy-term
definition Const :: String.literal ⇒ typerep ⇒ term where
    Const - - = dummy-term
definition App :: term ⇒ term ⇒ term where
    App - - = dummy-term
definition Abs :: String.literal ⇒ typerep ⇒ term ⇒ term where
    Abs - - - = dummy-term
definition Free :: String.literal ⇒ typerep ⇒ term where
    Free - - = dummy-term
code-datatype Const App Abs Free
class term-of = typerep +
  fixes term-of :: 'a ⇒ term
lemma term-of-anything: term-of x ≡ t
  by (rule eq-reflection) (cases term-of x, cases t, simp)
definition valapp :: ('a ⇒ 'b) × (unit ⇒ term) ⇒ 'a × (unit ⇒ term) ⇒ 'b × (unit ⇒ term) where
valapp f x = (fst f (fst x), λu. App (snd f ()) (snd x ()))
lemma valapp-code [code, code-unfold]:
valapp \( f, tf \) \((x, tx)\) = \((f x, \lambda u. \text{App}(tf())(tx()))\)

by (simp only: valapp-def fst-conv snd-conv)

76.1.2 Syntax

definition termify :: 'a ⇒ term where
  [code del]: termify \( x \) = dummy-term

abbreviation valtermify :: 'a ⇒ 'a × (unit ⇒ term) where
  valtermify \( x \) ≡ \((x, \lambda u. \text{termify} x)\)

locale term-syntax
begin

notation App (infixl <·> 70)
  and valapp (infixl {·} 70)

end

interpretation term-syntax.

no-notation App (infixl <·> 70)
  and valapp (infixl {·} 70)

76.2 Tools setup and evaluation

context
begin

qualified definition TERM-OF :: 'a::term-of itself
where
  TERM-OF = snd \((\text{Code-Evaluation}.\text{term-of} :: 'a ⇒ -, \text{TYPE}('a))\)

qualified definition TERM-OF-EQUAL :: 'a::term-of itself
where
  TERM-OF-EQUAL = snd \((λ (a::'a). \text{Code-Evaluation}.\text{term-of} a, \text{HOL.eq} a), \text{TYPE}('a))\)

end

lemma eq-eq-TrueD;
  fixes \( x \) \( y \) :: 'a::{}
  assumes \((x \equiv y) \equiv \text{Trueprop True}\)
  shows \( x \equiv y \)
  using assms by simp

code-printing
  type-constructor \text{term} → (Eval) \text{Term.term}
| \text{constant Const} → (Eval) \text{Term.Const}/ ((-), (-))
| \text{constant App} → (Eval) \text{Term.$}/ ((-), (-))
THEORY "Code-Evaluation"

| constant \( \text{Abs} \rightarrow (\text{Eval}) \text{Term.Abs}/ ((-), (-), (-)) \)  
| constant \( \text{Free} \rightarrow (\text{Eval}) \text{Term.Free}/ ((-), (-)) \)  

ML-file ⟨Tools/code-evaluation.ML⟩

code-reserved Eval Code-Evaluation

ML-file ⟨/~src/HOL/Tools/value-command.ML⟩

76.3 Dedicated term-of instances

instantiation fun :: (typerep, typerep) term-of  
begin

definition
  term-of (f :: 'a ⇒ 'b) =  
    Const (STR "Pure.dummy-pattern")  
    (Typerep.Typerep (STR "fun") [Typerep.typerep TYPE('a), Typerep.typerep TYPE('b)])

instance ..

end

declare [[code drop: rec-term case-term  
  term-of :: typerep ⇒ - term-of :: term ⇒ - term-of :: String.literal ⇒ -  
  term-of :: - Predicate.pred ⇒ term term-of :: - Predicate.seq ⇒ term]]

code-printing  
  constant term-of :: integer ⇒ term ⇒ (Eval) HOLogic.mk'-'number/ HO-Logic.code'-integerT  
| constant term-of :: String.literal ⇒ term ⇒ (Eval) HOLogic.mk'-'literal

declare [[code drop: term-of :: integer ⇒ -]]

lemma term-of-integer [unfolded typerep-fun-def typerep-num-def typerep-integer-def, code]:
  term-of (i :: integer) =  
  (if \( i > 0 \) then  
    App (Const (STR "Num.numeral-class.numeral") (TYPEREPT(num ⇒ integer)))  
    (term-of (num-of-integer i))  
  else if \( i = 0 \) then Const (STR "Groups.zero-class.zero") TYPEREPT(integer)  
  else  
    App (Const (STR "Groups.uminus-class.uminus") TYPEREPT(integer ⇒ integer))  
    (term-of (− i)))  
by (rule term-of-anything [THEN meta-eq-to-obj-eq])
code-reserved Eval HOLogic

76.4 Generic reification
ML-file (~~/src/HOL/Tools/reification.ML)

76.5 Diagnostic
definition tracing :: String.literal ⇒ 'a ⇒ 'a where
    [code del]: tracing s x = x
code-printing
    constant tracing :: String.literal ⇒ 'a ⇒ 'a ⇒ (Eval) Code’-Evaluation.tracing

hide-const dummy-term valapp
hide-const (open) Const App Abs Free termify valtermify term-of tracing
end

77 A simple counterexample generator performing random testing

theory Quickcheck-Random
imports Random Code-Evaluation Enum
begin

notation fcomp (infixl ◦ > 60)
notation scomp (infixl ◦→ 60)
setup :Code-Target.add-derived-target (Quickcheck, [(Code-Runtime.target, I)])

77.1 Catching Match exceptions
axiomatization catch-match :: 'a ⇒ 'a ⇒ 'a

code-printing
    constant catch-match ⇒ (Quickcheck) ((-) handle Match ⇒ -)

code-reserved Quickcheck Match

77.2 The random class
class random = typerep +
    fixes random :: natural ⇒ Random.seed ⇒ ('a × (unit ⇒ term)) × Random.seed

77.3 Fundamental and numeric types
instantiation bool :: random
THEORY "Quickcheck-Random"

begin

definition
  random i = Random.range 2 \to
  (\lambda k. Pair (if k = 0 then Code-Evaluation.valtermify False else Code-Evaluation.valtermify True))

instance ..

end

instantiation itself :: (typerep) random
begin

definition
  random-itself :: natural \to Random.seed \to ('a itself \times (unit \to term)) \times Random.seed
where random-itself - = Pair (Code-Evaluation.valtermify TYPE('a))

instance ..

end

instantiation char :: random
begin

definition
  random = Random.select (Enum.enum :: char list) \to (\lambda c. Pair (c, \lambda u. Code-Evaluation.term-of c))

instance ..

end

instantiation String.literal :: random
begin

definition
  random - = Pair (STR """, \lambda u. Code-Evaluation.term-of (STR ""))

instance ..

end

instantiation nat :: random
begin

definition random-nat :: natural \to Random.seed
\to (nat \times (unit \to Code-Evaluation.term)) \times Random.seed
where
random-nat i = Random.range (i + 1) o→ (λk. Pair (let n = nat-of-natural k in (n, λ-. Code-Evaluation.term-of n)))

instance ..

end

instantiation int :: random

begin

definition
random i = Random.range (2 * i + 1) o→ (λk. Pair (let j = (if k ≥ i then int (nat-of-natural (k - i)) else - (int (nat-of-natural (i - k)))) in (j, λ-. Code-Evaluation.term-of j)))

instance ..

end

instantiation natural :: random

begin

definition
random-natural :: natural ⇒ Random.seed ⇒ (natural × (unit ⇒ Code-Evaluation.term)) × Random.seed
where
random-natural i = Random.range (i + 1) o→ (λn. Pair (n, λ-. Code-Evaluation.term-of n))

instance ..

end

instantiation integer :: random

begin

definition
random-integer :: natural ⇒ Random.seed ⇒ (integer × (unit ⇒ Code-Evaluation.term)) × Random.seed
where
random-integer i = Random.range (2 * i + 1) o→ (λk. Pair (let j = (if k ≥ i then integer-of-natural (k - i)) else - (integer-of-natural (i - k))) in (j, λ-. Code-Evaluation.term-of j)))

instance ..

end
77.4 Complex generators

Towards \( 'a \Rightarrow 'b \)

**axiomatization** random-fun-aux :: typerep \( \Rightarrow \) typerep \( \Rightarrow \) (\( 'a \Rightarrow 'a \Rightarrow \) bool) \( \Rightarrow \) (\( 'a \Rightarrow \) term)

\( \Rightarrow \) (Random.seed \( \Rightarrow \) (\( 'b \times (\) unit \( \Rightarrow \) term\)\)) \times Random.seed)

\( \Rightarrow \) (Random.seed \( \Rightarrow \) Random.seed \times Random.seed)

\( \Rightarrow \) Random.seed \( \Rightarrow \) ((\( 'a \Rightarrow 'b \) \times (\( unit \Rightarrow \) term)) \times Random.seed)

**definition** random-fun-aux \( :: \) (Random.seed \( \Rightarrow \) (\( 'b \times (\) unit \( \Rightarrow \) term\)\)) \times Random.seed)

\( \Rightarrow \) Random.seed \( \Rightarrow \) ((\( 'a::\) term-of \( \Rightarrow 'b::\) typerep\) \times (\( unit \Rightarrow \) term)) \times Random.seed)

**where**

random-fun-aux TYPEREP('a) TYPEREP('b) \( = \) Code-Evaluation.term-of f

Random.split-seed

**instantiation** fun \( :: \) (\{equal, term-of\}, random) random

begin

**definition**

random-fun \( :: \) natural \( \Rightarrow \) Random.seed \( \Rightarrow \) ((\( 'a \Rightarrow 'b \) \times (\( unit \Rightarrow \) term)) \times Random.seed)

**where**

random i \( = \) random-fun-lift (random i)

instance ..

end

Towards type copies and datatypes

**definition** collapse :: (\( 'a \Rightarrow (\( 'b \Rightarrow (\) 'b \times 'a) \times 'a\)) \Rightarrow 'a \Rightarrow 'b \times 'a

**where** collapse f \( = \) (f \( \circ \) id)

**definition** beyond :: natural \( \Rightarrow \) natural \( \Rightarrow \) natural

**where**

beyond k l \( = \) (if l \( > \) k then l else 0)

**lemma** beyond-zero: beyond k 0 \( = \) 0

by (simp add: beyond-def)

**definition** (in term-syntax) \[\text{code-unfold}]:

valterm-emptyset = Code-Evaluation.valtermify (\{\} :: ('a :: typerep) set)

**definition** (in term-syntax) \[\text{code-unfold}]:

valtermify-insert x s = Code-Evaluation.valtermify insert \{\} \( (x :: ('a :: typerep \times -)) \times \) s

**instantiation** set :: (random) random

begin
fun random-aux-set
where
random-aux-set 0 j = collapse (Random.select-weight [(1, Pair valterm-emptyset)])
| random-aux-set (Code-Numeral.Suc i) j =
collapse (Random.select-weight
[(1, Pair valterm-emptyset),
(Code-Numeral.Suc i,
random j o→ (%x. random-aux-set i j o→ (%s. Pair (valtermify-insert x s)))))

lemma [code]:
random-aux-set i j =
collapse (Random.select-weight [(1, Pair valterm-emptyset),
(i, random j o→ (%x. random-aux-set (i − 1) j o→ (%s. Pair (valtermify-insert x s))))])

proof (induct i rule: natural.induct)
case zero
  show ?case by (subst select-weight-drop-zero [symmetric])
    (simp add: random-aux-set.simps [simplified] less-natural-def)
next
case (Suc i)
  show ?case by (simp only: random-aux-set.simps(2) [of i] Suc-natural-minus-one)
qed

definition random-set i = random-aux-set i i

instance ..

end

lemma random-aux-rec:
  fixes random-aux :: natural ⇒ 'a
  assumes random-aux 0 = rhs 0
  and ∃k. random-aux (Code-Numeral.Suc k) = rhs (Code-Numeral.Suc k)
  shows random-aux k = rhs k
  using assms by (rule natural.induct)

77.5 Deriving random generators for datatypes

ML-file ⟨Tools/Quickcheck/quickcheck-common.ML⟩
ML-file ⟨Tools/Quickcheck/random-generators.ML⟩

77.6 Code setup

code-printing
  constant random-fun-aux ⇒ (Quickcheck) Random-'Generators.random-'fun
  — With enough criminal energy this can be abused to derive False; for this
  reason we use a distinguished target Quickcheck not spoiling the regular trusted
code generation
code-reserved Quickcheck Random-Generators

no-notation fcomp (infixl o > 60)
no-notation scomp (infixl o → 60)

hide-const (open) catch-match random collapse beyond random-fun-aux random-fun-lift
hide-fact (open) collapse-def beyond-def random-fun-lift-def

end

78 The Random-Predicate Monad

theory Random-Pred
imports Quickcheck-Random
begin

fun iter' :: 'a itself ⇒ natural ⇒ natural ⇒ Random.seed ⇒ ('a::random) Predicate.pred
where
iter' T nrandom sz seed = (if nrandom = 0 then bot-class.bot else
  let ((x, -), seed') = Quickcheck-Random.random sz seed
  in Predicate.Seq (%a. Predicate.Insert x (iter' T (nrandom - 1) sz seed')))

definition iter :: natural ⇒ natural ⇒ Random.seed ⇒ ('a::random) Predicate.pred
where
iter nrandom sz seed = iter' (TYPE('a)) nrandom sz seed

lemma [code]:
iter nrandom sz seed = (if nrandom = 0 then bot-class.bot else
  let ((x, -), seed') = Quickcheck-Random.random sz seed
  in Predicate.Seq (%a. Predicate.Insert x (iter (nrandom - 1) sz seed')))
unfolding iter-def iter’..simps [of - nrandom] ..

type-synonym 'a random-pred = Random.seed ⇒ ('a Predicate.pred × Random.seed)

definition empty :: 'a random-pred
where empty = Pair bot

definition single :: 'a ⇒ 'a random-pred
where single x = Pair (Predicate.single x)

definition bind :: 'a random-pred ⇒ ('a ⇒ 'b random-pred) ⇒ 'b random-pred
where
bind R f = (λs. let
  (P, s') = R s;
  (s1, s2) = Random.split-seed s'
  in f P s)
in (Predicate.bind P (%a. fst (f a s1)), s2))

**definition** union :: 'a random-pred ⇒ 'a random-pred ⇒ 'a random-pred

where
 union R1 R2 = (λs. let
 (P1, s') = R1 s;
 (P2, s'') = R2 s'
 in (sup-class.sup P1 P2, s''))

**definition** if-randompred :: bool ⇒ unit random-pred

where
 if-randompred b = (if b then single () else empty)

**definition** iterate upto :: (natural ⇒ 'a) ⇒ natural ⇒ natural ⇒ 'a random-pred

where
 iterate upto f n m = Pair (Predicate.iterate upto f n m)

**definition** not-randompred :: unit random-pred ⇒ unit random-pred

where
 not-randompred P = (λs. let
 (P', s') = P s
 in if Predicate.eval P' () then (Orderings.bot, s') else (Predicate.single (), s'))

**definition** Random :: (Random.seed ⇒ ('a × (unit ⇒ term)) × Random.seed) ⇒ 'a random-pred

where
 Random g = scomp g (Pair o (Predicate.single o fst))

**definition** map :: ('a ⇒ 'b) ⇒ 'a random-pred ⇒ 'b random-pred

where
 map f P = bind P (single o f)

hide-const (open) iter' iter empty single bind union if-randompred
iterate upto not-randompred Random map

hide-fact iter'.simp

hide-fact (open) iter-def empty-def single-def bind-def union-def
if-randompred-def iterate upto-def not-randompred-def Random-def map-def

end

79 Various kind of sequences inside the random monad

theory Random-Sequence
imports Random-Pred
begin

type-synonym 'a random-dseq = natural ⇒ natural ⇒ Random.seed ⇒ ('a Limited-Sequence.dseq × Random.seed)
fun Random :: (natural ⇒ Random.seed ⇒ (('a × (unit ⇒ term)) × Random.seed)) ⇒ ('a random-dseq)

where
Random g nrandom = (%size. if nrandom <= 0 then (Pair Limited-Sequence.empty) else 
(scomp (g size) (%r. scomp (Random g (nrandom − 1) size) (%rs. Pair (Limited-Sequence.union (Limited-Sequence.single (fst r)) rs)))))

type-synonym 'a pos-random-dseq = natural ⇒ natural ⇒ Random.seed ⇒ 'a Limited-Sequence.pos-dseq

definition pos-empty :: 'a pos-random-dseq
where

pos-empty = (%|nrandom size seed. Limited-Sequence.pos-empty)

definition pos-single :: 'a => 'a pos-random-dseq
where
pos-single x = (%|nrandom size seed. Limited-Sequence.pos-single x)

definition pos-bind :: 'a pos-random-dseq => ('a => 'b pos-random-dseq) => 'b pos-random-dseq
where
pos-bind R f = (\nrandom size seed. Limited-Sequence.pos-bind (R nrandom size seed) (%a. f a nrandom size seed))

definition pos-decr-bind :: 'a pos-random-dseq => ('a => 'b pos-random-dseq) => 'b pos-random-dseq
where
pos-decr-bind R f = (\nrandom size seed. Limited-Sequence.pos-decr-bind (R nrandom size seed) (%a. f a nrandom size seed))

definition pos-union :: 'a pos-random-dseq => 'a pos-random-dseq => 'a pos-random-dseq
where
pos-union R1 R2 = (\nrandom size seed. Limited-Sequence.pos-union (R1 nrandom size seed) (R2 nrandom size seed))

definition pos-if-random-dseq :: bool => unit pos-random-dseq
where
pos-if-random-dseq b = (if b then pos-single () else pos-empty)

definition pos-iterate-upto :: (natural => 'a) => natural => natural => 'a pos-random-dseq
where
pos-iterate-upto f n m = (\nrandom size seed. Limited-Sequence.pos-iterate-upto f n m)

definition pos-map :: ('a => 'b) => 'a pos-random-dseq => 'b pos-random-dseq
where
pos-map f P = pos-bind P (pos-single o f)

fun iter :: (Random.seed => ('a x (unit => term)) x Random.seed) => natural => Random.seed => 'a Lazy-Sequence.lazy-sequence
where
iter random nrandom seed =
(if nrandom = 0 then Lazy-Sequence.empty else Lazy-Sequence.Lazy-Sequence(%a. let ((x, -), seed') = random seed in Some (x, iter random (nrandom - 1) seed'))

definition pos-Random :: (natural => Random.seed => ('a x (unit => term)) x Random.seed) => 'a pos-random-dseq
where

\[ \text{pos-Random } g = (%\text{nrandom size seed iter (g size) nrandom seed}) \]

type-synonym 'a neg-random-dseq = natural ⇒ natural ⇒ \text{Random.seed} ⇒ 'a Limited-Sequence.neg-dseq

definition neg-empty :: 'a neg-random-dseq
where

\[ \text{neg-empty} = (%\text{nrandom size seed. Limited-Sequence.neg-empty}) \]

definition neg-single :: 'a => 'a neg-random-dseq
where

\[ \text{neg-single } x = (%\text{nrandom size seed. Limited-Sequence.neg-single } x) \]

definition neg-bind :: 'a neg-random-dseq => ('a => 'b neg-random-dseq) => 'b neg-random-dseq
where

\[ \text{neg-bind } R f = (\lambda \text{nrandom size seed. Limited-Sequence.neg-bind } (R \text{nrandom size seed}) (%a. f a \text{nrandom size seed})) \]

definition neg-decr-bind :: 'a neg-random-dseq => ('a => 'b neg-random-dseq) => 'b neg-random-dseq
where

\[ \text{neg-decr-bind } R f = (\lambda \text{nrandom size seed. Limited-Sequence.neg-decr-bind } (R \text{nrandom size seed}) (%a. f a \text{nrandom size seed})) \]

definition neg-union :: 'a neg-random-dseq => 'a neg-random-dseq => 'a neg-random-dseq
where

\[ \text{neg-union } R1 R2 = (\lambda \text{nrandom size seed. Limited-Sequence.neg-union } (R1 \text{nrandom size seed}) (R2 \text{nrandom size seed})) \]

definition neg-if-random-dseq :: bool => unit neg-random-dseq
where

\[ \text{neg-if-random-dseq } b = (\text{if } b \text{ then neg-single } () \text{ else neg-empty}) \]

definition neg-iterate-upto :: (natural => 'a) => natural => natural => 'a neg-random-dseq
where

\[ \text{neg-iterate-upto } f n m = (\lambda \text{nrandom size seed. Limited-Sequence.neg-iterate-upto } f n m) \]

definition neg-not-random-dseq :: unit pos-random-dseq => unit neg-random-dseq
where

\[ \text{neg-not-random-dseq } R = (\lambda \text{nrandom size seed. Limited-Sequence.neg-not-seq } (R \text{nrandom size seed})) \]

definition neg-map :: ('a => 'b) => 'a neg-random-dseq => 'b neg-random-dseq
where
THEORY "Quickcheck-Exhaustive"

neg-map f P = neg-bind P (neg-single o f)

definition pos-not-random-dseq :: unit neg-random-dseq => unit pos-random-dseq
where
pos-not-random-dseq R = (λnrandom size seed. Limited-Sequence.pos-not-seq (R nrandom size seed))

hide-const (open)
empty single bind union if-random-dseq not-random-dseq map Random
pos-empty pos-single pos-bind pos-decr-bind pos-union pos-if-random-dseq pos-iterate-upto
pos-not-random-dseq pos-map iter pos-Random
neg-empty neg-single neg-bind neg-decr-bind neg-union neg-if-random-dseq neg-iterate-upto
neg-not-random-dseq neg-map

hide-fact (open) empty-def single-def bind-def union-def if-random-dseq-def not-random-dseq-def
map-def Random.simps
pos-iterate-upto-def pos-not-random-dseq-def pos-map-def iter.simps pos-Random-def
neg-empty-def neg-single-def neg-bind-def neg-decr-bind-def neg-union-def neg-if-random-dseq-def
neg-iterate-upto-def neg-not-random-dseq-def neg-map-def

eend

80 A simple counterexample generator performing exhaustive testing

theory Quickcheck-Exhaustive
imports Quickcheck-Random
keywords quickcheck-generator :: thy-decl
begin

80.1 Basic operations for exhaustive generators
definition orelse :: 'a option => 'a option => 'a option
(infixr orelse 55)
where [code-unfold]: x orelse y = (case x of Some x' => Some x' | None => y)

80.2 Exhaustive generator type classes
class exhaustive = term-of +
  fixes exhaustive :: ('a => (bool × term list) option) => natural => (bool × term list) option

class full-exhaustive = term-of +
  fixes full-exhaustive ::
    ('a × (unit => term) => (bool × term list) option) => natural => (bool × term list) option

instantiation natural :: full-exhaustive
begin

function full-exhaustive-natural' ::
  (natural × (unit ⇒ term) ⇒ (bool × term list) option) ⇒
  natural ⇒ natural ⇒ (bool × term list) option
where full-exhaustive-natural' f d i =
  (if d < i then None
   else (f (i, λ-. Code-Evaluation.term-of i)) orelse (full-exhaustive-natural' f d (i + 1)))
by pat-completeness auto

termination
  by (relation measure (λ(-, d, i). nat-of-natural (d + I – i))) (auto simp add: less-natural-def)

definition full-exhaustive f d = full-exhaustive-natural' f d 0

instance ..

end

instantiation natural :: exhaustive
begin

function exhaustive-natural' ::
  (natural ⇒ (bool × term list) option) ⇒ natural ⇒ natural ⇒ (bool × term list) option
where exhaustive-natural' f d i =
  (if d < i then None
   else (f i orelse exhaustive-natural' f d (i + 1)))
by pat-completeness auto

termination
  by (relation measure (λ(-, d, i). nat-of-natural (d + I – i))) (auto simp add: less-natural-def)

definition exhaustive f d = exhaustive-natural' f d 0

instance ..

end

instantiation integer :: exhaustive
begin

function exhaustive-integer' ::
  (integer ⇒ (bool × term list) option) ⇒ integer ⇒ integer ⇒ (bool × term list) option
where exhaustive-integer' f d i =
(if \(d < i\) then None else \(f\ orelse\ exhaustive-integer' f\ d\ (i + 1)\))

by pat-completeness auto

termination

by (relation measure \((\lambda\ (\cdot, d, i).\ \text{nat-of-integer} (d + 1 - i))\))

(auto simp add: less-integer-def nat-of-integer-def)

definition exhaustive f d = exhaustive-integer' f (integer-of-natural d) (- (integer-of-natural d))

instance ..

end

instantiation integer :: full-exhaustive

begin

function full-exhaustive-integer' ::

(integer \times\ (\unit\ \Rightarrow\ \text{term}) \Rightarrow\ (\bool \times\ \text{term list} \ \Rightarrow\ \text{option} ) \Rightarrow\ integer \Rightarrow\ integer \Rightarrow\ (\bool \times\ \text{term list} \ \Rightarrow\ \text{option})

where full-exhaustive-integer' f d i =

(if \(d < i\) then None

else

(case f\ (i, \\lambda\ \cdot.\ \text{Code-Evaluation.\term-of\ i})\ of

Some t \Rightarrow\ Some t

\mid\ None \Rightarrow\ full-exhaustive-integer' f\ d\ (i + 1))\))

by pat-completeness auto

termination

by (relation measure \((\lambda\ (\cdot, d, i).\ \text{nat-of-integer} (d + 1 - i))\))

(auto simp add: less-integer-def nat-of-integer-def)

definition full-exhaustive f d =

full-exhaustive-integer' f (integer-of-natural d) (- (integer-of-natural d))

instance ..

end

instantiation nat :: exhaustive

begin

definition exhaustive f d = exhaustive (\lambda x. f\ (\text{nat-of-natural} x))\ d

instance ..

end

instantiation nat :: full-exhaustive
begin

definition full-exhaustive f d =
full-exhaustive (λ(x, xt). f (nat-of-natural x, λ-. Code-Evaluation.term-of (nat-of-natural x))) d

instance ..

end

instantiation int :: exhaustive
begin

function exhaustive-int' ::
  (int ⇒ (bool × term list) option) ⇒ int ⇒ int ⇒ (bool × term list) option
where exhaustive-int' f d i =
  (if d < i then None else (f i orelse exhaustive-int' f d (i + 1)))
by pat-completeness auto

termination
  by (relation measure (λ(-, d, i). nat (d + 1 - i))) auto

definition exhaustive f d =
exhaustive-int' f (int-of-integer (integer-of-natural d))
  (¬ (int-of-integer (integer-of-natural d)))

instance ..

end

instantiation int :: full-exhaustive
begin

function full-exhaustive-int' ::
  (int × (unit ⇒ term) ⇒ (bool × term list) option) ⇒
  int ⇒ int ⇒ (bool × term list) option
where full-exhaustive-int' f d i =
  (if d < i then None
   else
    (case f (i, λ-. Code-Evaluation.term-of i) of
     Some t ⇒ Some t
     | None ⇒ full-exhaustive-int' f d (i + 1)))
by pat-completeness auto

termination
  by (relation measure (λ(-, d, i). nat (d + 1 - i))) auto

definition full-exhaustive f d =
full-exhaustive-int' f (int-of-integer (integer-of-natural d))
THEORY “Quickcheck-Exhaustive”

(\neg (\text{int-of-integer} (\text{integer-of-natural} d)))

instance ..

end

instantiation prod :: (exhaustive, exhaustive) exhaustive
begin

definition exhaustive f d = exhaustive (\lambda x. exhaustive (\lambda y. f ((x, y))) d) d

instance ..

end

definition (in \text{term-syntax})
\[\text{code-unfold}::\]
valtermify-pair x y =
Code-Evaluation.valtermify (Pair :: 'a::typerep \Rightarrow 'b::typerep \Rightarrow 'a \times 'b) \{\cdot\} x
\{\cdot\} y

instantiation prod :: (full-exhaustive, full-exhaustive) full-exhaustive
begin

definition full-exhaustive f d =
full-exhaustive (\lambda x. full-exhaustive (\lambda y. f (valtermify-pair x y)) d) d

instance ..

end

instantiation set :: (exhaustive) exhaustive
begin

fun exhaustive-set
where
exhaustive-set f i =
(if i = 0 then None
else
f {} orelse
exhaustive-set
(\lambda A. f A orelse exhaustive (\lambda x. if x \in A then None else f (insert x A)) (i - 1)) (i - 1))

instance ..

end

instantiation set :: (full-exhaustive) full-exhaustive
begin
fun full-exhaustive-set
where
  full-exhaustive-set f i =
  (if i = 0 then None
   else
     f valterm-emptyset orelse
     full-exhaustive-set
     (λA.
         f A orelse Quickcheck-Exhaustive.full-exhaustive
         (λx.
             if fst x ∈ fst A then None else f (valterm-insert x A)) (i − 1)) (i
         − 1))

instance ..
end

instantiation fun :: {equal,exhaustive}, exhaustive exhaustion
begin
fun exhaustive-fun' ::
  (((′a ⇒ ′b) ⇒ (bool × term list) option) ⇒ natural ⇒ natural ⇒ (bool × term
  list) option)
where
  exhaustive-fun' f i d =
  (exhaustive (λb.
      f (λ−.
          b)) d) orelse
  (if i > 1 then
   exhaustive-fun'
   (λg.
        exhaustive (λa. exhaustive (λb.
          f (g(a := b))) d) d) (i − 1) d else
    None)

definition exhaustive-fun ::
  (((′a ⇒ ′b) ⇒ (bool × term list) option) ⇒ natural ⇒ (bool × term list) option)
where exhaustive-fun f d = exhaustive-fun' f d d

instance ..
end

definition [code-unfold]:
  valtermify-absdummy =
    (λ(v, t).
      (λ::′a. v,
        λa::unit. Code-Evaluation.Abs (STR "x") (Typerep.typerep TYPE(′a::typerep))
         (t ())))

definition (in term-syntax)
  [code-unfold]: valtermify-fun-upd g a b =
    Code-Evaluation.valtermify
    (fun-upd :: (′a::typerep ⇒ ′b::typerep) ⇒ ′a ⇒ ′b ⇒ ′a ⇒ ′b) {·} g {·} a {·}
instantiation fun :: \{\text{equal,full-exhaustive}\}, full-exhaustive \} full-exhaustive

begin

fun full-exhaustive-fun' ::
(('a ⇒ 'b) × (unit ⇒ term) ⇒ (bool × term list) option) ⇒
natural ⇒ natural ⇒ (bool × term list) option

where
full-exhaustive-fun' f i d =
full-exhaustive (λv. f (valtermify-absdummy v)) d or else
(if i > 1 then
full-exhaustive-fun' (λg. full-exhaustive (λa. full-exhaustive (λb. f (valtermify-fun-upd g a b)) d) d) (i - 1) d
else None)

definition full-exhaustive-fun ::
((a × (unit ⇒ term) ⇒ (bool × term list) option) ⇒
natural ⇒ (bool × term list) option

where full-exhaustive-fun f d = full-exhaustive-fun' f d d

instance ..

end

80.2.1 A smarter enumeration scheme for functions over finite datatypes

class check-all = enum + term-of +

fixes check-all :: (a × (unit ⇒ term) ⇒ (bool × term list) option) ⇒ (bool * term list) option

fixes enum-term-of :: 'a itself ⇒ unit ⇒ term list

fun check-all-n-lists :: (a::check-all list × (unit ⇒ term list) ⇒
(bool × term list) option) ⇒ natural ⇒ (bool * term list) option

where
check-all-n-lists f n =
(if n = 0 then f ([], (λ_. []))
else check-all (λ(x, xt).
check-all-n-lists (λ(xs, xst). f ((x # xs), (λ_. (xt () # xst ()'))) (n - 1))))

definition (in term-syntax)
[code-unfold]: termify-fun-upd g a b =
(Code-Evaluation.termify
(fun-upd :: (a::typerep ⇒ 'b::typerep) ⇒ 'a ⇒ 'b ⇒ 'a ⇒ 'b) <$> g <$> a
<*> b)

definition mk-map-term ::
THEORY "Quickcheck-Exhaustive"

```
(unit ⇒ typerep) ⇒ (unit ⇒ typerep) ⇒
(unit ⇒ term list) ⇒ (unit ⇒ term list) ⇒ unit ⇒ term
where mk-map-term T1 T2 domm rng =
(λ-.
  let
    T1 = T1 ();
    T2 = T2 ();
    update-term =
      (λg (a, b).
          (Code-Evaluation.Const (STR "Fun.fun-upd")
            (Typerep.Typerep (STR "fun") [Typerep.Typerep (STR "fun") [T1,
              T2]]))
            Typerep.Typerep (STR "fun") [T1,
              Typerep.Typerep (STR "fun") [T2, Typerep.Typerep (STR "fun")
                [T1, T2]]]))
        g) a b)
in
      List.foldl update-term
      (Code-Evaluation.Abs (STR "x") T1
        (Code-Evaluation.Const (STR "HOL.undefined") T2)) (zip (domm ()))
      (rng ()))

instantiation fun :: ({equal,check-all}, check-all) check-all begin

definition check-all f =
(λ-.
  mk-term =
    mk-map-term
    (λ-. Typerep.typerep (TYPE('a)))
    (λ-. Typerep.typerep (TYPE('b)))
    (enum-term-of (TYPE('a)));
  enum = (Enum.enum :: 'a list)
in
  check-all-n-lists
    (λ(xs, yst). f (the ∘ map-of (zip enum ys), mk-term yst))
    (natural-of-nat (length enum)))

definition enum-term-of-fun :: ('a ⇒ 'b) itself ⇒ unit ⇒ term list
where enum-term-of-fun =
(λ-.
  let
    enum-term-of-a = enum-term-of (TYPE('a));
    mk-term =
      mk-map-term
      (λ-. Typerep.typerep (TYPE('a)))
      (λ-. Typerep.typerep (TYPE('b)))
```
enum-term-of-a
in
map (λys. mk-term (λ-. ys))
(List.n-lists (length (enum-term-of-a () (enum-term-of (TYPE(′b)) ()))))

instance ..
end

fun (in term-syntax) check-all-subsets ::
((′a::typerep) set × (unit ⇒ term) ⇒ (bool × term list) option) ⇒
(′a × (unit ⇒ term)) list ⇒ (bool × term list) option
where
check-all-subsets f [] = f valterm-emptyset
| check-all-subsets f (x # xs) =
  check-all-subsets (λs. case f s of Some ts ⇒ Some ts | None ⇒ f (valtermify-insert x s)) xs

definition (in term-syntax)
[code-unfold]:
term-emptyset = Code-Evaluation.termify ({}) :: (′a::typerep) set

definition (in term-syntax)
[code-unfold]:
termify-insert x s =
  Code-Evaluation.termify (insert :: (′a::typerep) ⇒ ′a set ⇒ ′a set) <-> x <-> s

definition (in term-syntax)
setify :: (′a::typerep) setify (term list) ⇒ term
where
setify T ts = foldr (termify-insert T) ts (term-emptyset T)
instantiation set :: (check-all) check-all
begin

definition
check-all-set f =
  check-all-subsets f
  (zip (Enum.enum :: ′a list)
    (map (λa. λu :: unit. a) (Quickcheck-Exhaustive.enum-term-of (TYPE (′a)) ()))))

definition
enum-term-of-set :: ′a set itself ⇒ unit ⇒ term list
where
enum-term-of-set - - =
  map (setify (TYPE(′a))) (subseqs (Quickcheck-Exhaustive.enum-term-of (TYPE(′a)) () ()))
instance ..
end
THEORY “Quickcheck-Exhaustive”

instantiation unit :: check-all
begin

  definition check-all f = f (Code-Evaluation.valtermify ())

  definition enum-term-of-unit :: unit itself ⇒ unit ⇒ term list
  where enum-term-of-unit = (λ· ·. [Code-Evaluation.term-of ()])

  instance ..

end

instantiation bool :: check-all
begin

  definition check-all f =
    (case f (Code-Evaluation.valtermify False) of
      Some x′ ⇒ Some x′
    | None ⇒ f (Code-Evaluation.valtermify True))

  definition enum-term-of-bool :: bool itself ⇒ unit ⇒ term list
    where enum-term-of-bool = (λ· ·. map Code-Evaluation.term-of (Enum.enum :: bool list))

  instance ..

end

definition (in term-syntax) [code-unfold]:
  termify-pair x y =
    Code-Evaluation.termify (Pair :: 'a::typerep ⇒ 'b :: typerep ⇒ 'a * 'b) <$> x <$> y

instantiation prod :: (check-all, check-all) check-all
begin

definition check-all f = check-all (λx. check-all (λy. f (valtermify-pair x y)))

  definition enum-term-of-prod :: ('a * 'b) itself ⇒ unit ⇒ term list
    where enum-term-of-prod =
      (λ· ·. map (λ(x, y). termify-pair TYPE('a) TYPE('b) x y)
        (List.product (enum-term-of (TYPE('a)) ()) (enum-term-of (TYPE('b)) ()))))

  instance ..
end

definition (in term-syntax)  
[code-unfold]: valtermify-Inl x =  
Code-Evaluation.valtermify (Inl :: 'a::typerep ⇒ 'a + 'b :: typerep) {} x

definition (in term-syntax)  
[code-unfold]: valtermify-Inr x =  
Code-Evaluation.valtermify (Inr :: 'b::typerep ⇒ 'a::typerep + 'b) {} x

instantiation sum :: (check-all, check-all) check-all  
begin
check-all f = check-all (λa. f (valtermify-Inl a)) orelse check-all (λb. f (valtermify-Inr b))

definition enum-term-of-sum :: ('a + 'b) itself ⇒ unit ⇒ term list  
where enum-term-of-sum =  
(λ- -.  
let  
T1 = Typerep.typerep (TYPE('a));  
T2 = Typerep.typerep (TYPE('b))  
in  
map  
(Typerep.Typerep (STR "fun") [T1, Typerep.Typerep (STR "Sum-Type.sum")])  
[T1, T2]))  
(map  
(Typerep.Typerep (STR "fun") [T2, Typerep.Typerep (STR "Sum-Type.sum")])  
[T1, T2])))  
enum-term-of (TYPE('a)) () @  
map  
enum-term-of (TYPE('b)) ()

instance ..

end

instantiation char :: check-all  
begin
primrec check-all-char' ::  
(char × (unit ⇒ term) ⇒ (bool × term list) option) ⇒ char list ⇒ (bool × term list) option  
where check-all-char' f [] = None  
| check-all-char' f (c # cs) = f (c, λ-. Code-Evaluation.term-of c)  
orelse check-all-char' f cs
**THEORY** “Quickcheck-Exhaustive”

**definition** check-all-char ::

(char × (unit ⇒ term) ⇒ (bool × term list) option) ⇒ (bool × term list) option

**where** check-all f = check-all-char' f Enum.enum

**definition** enum-term-of-char :: char itself ⇒ unit ⇒ term list

**where**

enum-term-of-char = (λ -. map Code-Evaluation.term-of (Enum.enum :: char list))

**instance ..**

**end**

**instantiation** option :: (check-all) check-all

**begin**

**definition**

check-all f =

f (Code-Evaluation.valtermify (None :: 'a option)) orelse
check-all

(λ(x, t).

f

(Some x,

λ -. Code-Evaluation.App

(Code-Evaluation.Const (STR "Option.option.Some")
(Typerep.Typerep (STR "fun")
[Typerep.typerep TYPE('a),
Typerep.Typerep (STR "Option.option") [Typerep.typerep TYPE('a)]]))

(t ()))))

**definition** enum-term-of-option :: 'a option itself ⇒ unit ⇒ term list

**where**

enum-term-of-option =

(λ -. 

Code-Evaluation.term-of (None :: 'a option) #
(map

(Code-Evaluation.App

(Code-Evaluation.Const (STR "Option.option.Some")
(Typerep.Typerep (STR "fun")
[Typerep.typerep TYPE('a),
Typerep.Typerep (STR "Option.option") [Typerep.typerep TYPE('a)]]))

(arm-term-of (TYPE('a)) ())))

**instance ..**

**end**

**instantiation** Enum.finite-1 :: check-all
begin

definition check-all \( f = f \) (Code-Evaluation.valtermify Enum.finite-1.a_1) 

definition enum-term-of-finite-1 :: Enum.finite-1 itself \Rightarrow \text{unit} \Rightarrow \text{term list} 
  where enum-term-of-finite-1 = \( \lambda - . \) [Code-Evaluation.term-of Enum.finite-1.a_1]) 

instance ..

end

instantiation Enum.finite-2 :: check-all 
begin 

definition
  check-all \( f = \) 
    \begin{align*}
    f \ (\text{Code-Evaluation.valtermify Enum.finite-2.a_1}) \lor \\
    f \ (\text{Code-Evaluation.valtermify Enum.finite-2.a_2})
    \end{align*} 

definition enum-term-of-finite-2 :: Enum.finite-2 itself \Rightarrow \text{unit} \Rightarrow \text{term list} 
  where enum-term-of-finite-2 = 
    \( \lambda - . \) map Code-Evaluation.term-of (Enum.enum :: Enum.finite-2 list)) 

instance ..

end 

instantiation Enum.finite-3 :: check-all 
begin 

definition
  check-all \( f = \) 
    \begin{align*}
    f \ (\text{Code-Evaluation.valtermify Enum.finite-3.a_1}) \lor \\
    f \ (\text{Code-Evaluation.valtermify Enum.finite-3.a_2}) \lor \\
    f \ (\text{Code-Evaluation.valtermify Enum.finite-3.a_3})
    \end{align*} 

definition enum-term-of-finite-3 :: Enum.finite-3 itself \Rightarrow \text{unit} \Rightarrow \text{term list} 
  where enum-term-of-finite-3 = 
    \( \lambda - . \) map Code-Evaluation.term-of (Enum.enum :: Enum.finite-3 list)) 

instance ..

end 

instantiation Enum.finite-4 :: check-all 
begin 

definition
  check-all \( f = \) 

\begin{verbatim}
valtermify Enum.finite-4.
a1 orelse
valtermify Enum.finite-4.
a2 orelse
valtermify Enum.finite-4.
a3 orelse
valtermify Enum.finite-4.
a4

definition enum-term-of-finite-4 :: Enum.finite-4 itself ⇒ unit ⇒ term list
where enum-term-of-finite-4 =
  (λ·. map Code-Evaluation.term-of (Enum.enum :: Enum.finite-4 list))

instance ..
end

80.3 Bounded universal quantifiers

class bounded-forall =
  fixes bounded-forall :: ('a ⇒ bool) ⇒ natural ⇒ bool

80.4 Fast exhaustive combinators

class fast-exhaustive = term-of +
  fixes fast-exhaustive :: ('a ⇒ unit) ⇒ natural ⇒ unit

axiomatization throw-Counterexample :: term list ⇒ unit
axiomatization catch-Counterexample :: unit ⇒ term list option

code-printing
  constant throw-Counterexample →
  (Quickcheck) raise (Exhaustive'-Generators.Counterexample -)
  | constant catch-Counterexample →
  (Quickcheck) ((λ·; NONE) handle Exhaustive'-Generators.Counterexample ts ⇒ SOME ts)

80.5 Continuation passing style functions as plus monad

type-synonym 'a cps = ('a ⇒ term list option) ⇒ term list option

definition cps-empty :: 'a cps
  where cps-empty = (λcont. None)

definition cps-single :: 'a ⇒ 'a cps
  where cps-single v = (λcont. cont v)

definition cps-bind :: 'a cps ⇒ ('a ⇒ 'b cps) ⇒ 'b cps
  where cps-bind m f = (λcont. m (λa. (f a) cont))

definition cps-plus :: 'a cps ⇒ 'a cps ⇒ 'a cps
  where cps-plus a b = (λc. case a c of None ⇒ b c | Some x ⇒ Some x)

definition cps-if :: bool ⇒ unit cps
\end{verbatim}
where \( \text{cps-if } b = (\text{if } b \text{ then } \text{cps-single } () \text{ else } \text{cps-empty}) \)

**definition** \( \text{cps-not } :: \text{unit} \times \text{unit} \Rightarrow \text{unit} \times \text{unit} \)

where \( \text{cps-not } n = (\lambda c. \text{case } n \left( \lambda u. \text{Some } [] \right) \Rightarrow c () | \text{Some } - \Rightarrow \text{None}) \)

**type-synonym** \( \forall a \Rightarrow \text{pos-bound-cps} = \left( \forall a \Rightarrow (\text{bool} \times \text{term list}) \Rightarrow \text{natural} \Rightarrow (\text{bool} \times \text{term list}) \Rightarrow \text{option} \right) \)

**definition** \( \text{pos-bound-cps-empty} :: \forall a \Rightarrow \text{pos-bound-cps} \)

where \( \text{pos-bound-cps-empty} = (\lambda \text{cont } i. \text{None}) \)

**definition** \( \text{pos-bound-cps-single} :: \forall a \Rightarrow \forall a \Rightarrow \text{unit} \times \text{pos-bound-cps} \)

where \( \text{pos-bound-cps-single } v = (\lambda \text{cont } i. \text{cont } v) \)

**definition** \( \text{pos-bound-cps-bind} :: \forall a \Rightarrow \forall b \Rightarrow \forall a \Rightarrow \forall b \Rightarrow \text{unit} \times \text{pos-bound-cps} \)

where \( \text{pos-bound-cps-bind } m f = (\lambda \text{cont } i. \text{if } i = 0 \text{ then } \text{cont } \text{Unknown} \text{ else } m (\lambda a. \text{case } a \left( \text{of } \text{Unknown } \Rightarrow \text{cont } \text{Unknown } | \text{Known } a' \Rightarrow f a' \text{ cont } i) \text{ cont } i - 1)) \)

**definition** \( \text{pos-bound-cps-plus} :: \forall a \Rightarrow \forall a \Rightarrow \forall a \Rightarrow \text{unit} \times \text{pos-bound-cps} \)

where \( \text{pos-bound-cps-plus } a b = (\lambda \text{cont } i. \text{case } a \left( \text{cont } i \right) \left( \text{some } x \Rightarrow \text{cont } i \right) \text{ cont } i) \)

**definition** \( \text{pos-bound-cps-if} :: \text{bool} \Rightarrow \text{unit} \times \text{pos-bound-cps} \)

where \( \text{pos-bound-cps-if } b = (\text{if } b \text{ then } \text{pos-bound-cps-single } () \text{ else } \text{pos-bound-cps-empty}) \)

**datatype** (plugins only: code extraction) (dead \( \forall a \Rightarrow \text{unknown} = \text{Unknown } \mid \text{Known } a \))

**datatype** (plugins only: code extraction) (dead \( \forall a \Rightarrow \text{three-valued} = \text{Unknown-value } \mid \text{Value } a \mid \text{No-value} \))

**type-synonym** \( \forall a \Rightarrow \text{neg-bound-cps} = \left( \forall a \Rightarrow \text{term list three-valued} \Rightarrow \text{natural} \Rightarrow \text{term list three-valued} \right) \)

**definition** \( \text{neg-bound-cps-empty} :: \forall a \Rightarrow \text{neg-bound-cps} \)

where \( \text{neg-bound-cps-empty} = (\lambda \text{cont } i. \text{No-value}) \)

**definition** \( \text{neg-bound-cps-single} :: \forall a \Rightarrow \forall a \Rightarrow \forall a \Rightarrow \text{unit} \times \text{neg-bound-cps} \)

where \( \text{neg-bound-cps-single } v = (\lambda \text{cont } i. \text{cont } (\text{Known } v)) \)

**definition** \( \text{neg-bound-cps-bind} :: \forall a \Rightarrow \forall b \Rightarrow \forall a \Rightarrow \forall b \Rightarrow \text{unit} \times \text{neg-bound-cps} \)

where \( \text{neg-bound-cps-bind } m f = \\
(\lambda \text{cont } i. \text{if } i = 0 \text{ then } \text{cont } \text{Unknown} \text{ else } m (\lambda a. \text{case } a \left( \text{of } \text{Unknown } \Rightarrow \text{cont } \text{Unknown } | \text{Known } a' \Rightarrow f a' \text{ cont } i) \text{ cont } i - 1)) \)
**THEORY “Quickcheck-Exhaustive”**

**definition** neg-bound-cps-plus :: 'a neg-bound-cps ⇒ 'a neg-bound-cps ⇒ 'a neg-bound-cps
where neg-bound-cps-plus a b =
(λc i.
  (case a c i of
    No-value ⇒ b c i
  | Value x ⇒ Value x
  | Unknown-value ⇒
    (case b c i of
      No-value ⇒ Unknown-value
    | Value x ⇒ Value x
    | Unknown-value ⇒ Unknown-value)))

**definition** neg-bound-cps-if :: bool ⇒ unit neg-bound-cps
where neg-bound-cps-if b = (if b then neg-bound-cps-single () else neg-bound-cps-empty)

**definition** neg-bound-cps-not :: unit pos-bound-cps ⇒ unit neg-bound-cps
where neg-bound-cps-not n =
(λc i.
  (case n (λu. Some (True, [])) i of None ⇒ c (Known ())) | Some _ ⇒ No-value)

**definition** pos-bound-cps-not :: unit neg-bound-cps ⇒ unit pos-bound-cps
where pos-bound-cps-not n =
(λc i.
  (case n (λu. Value []) i of No-value ⇒ c () | Value _ ⇒ None | Unknown-value ⇒ None))

### 80.6 Defining generators for any first-order data type

**axiomatization** unknown :: 'a

**notation** (output) unknown (?)

**ML-file** ⟨Tools/Quickcheck/exhaustive-generators.ML⟩

**declare** [[quickcheck-batch-tester = exhaustive]]

### 80.7 Defining generators for abstract types

**ML-file** ⟨Tools/Quickcheck/abstract-generators.ML⟩

**hide-fact** (open) orelse-def

**no-notation** orelse (infixr orelse 55)

**hide-const** valtermify-undefined valtermify-fun-upd
valtermify-emptysert valtermify-insert
valtermify-pair valtermify-Inl valtermify-Inr
termify-fun-upd term-emptysert termify-insert termify-pair setify

**hide-const** (open)
exhaustive full-exhaustive
exhaustive-int' full-exhaustive-int'
81 A compiler for predicates defined by introduction rules

theory Predicate-Compile
imports Random-Sequence Quickcheck-Exhaustive
keywords
code-pred :: thy-goal and
values :: diag
begin

ML-file (Tools/Predicate-Compile/predicate-compile-aux.ML)
ML-file (Tools/Predicate-Compile/predicate-compile-compilations.ML)
ML-file (Tools/Predicate-Compile/core-data.ML)
ML-file (Tools/Predicate-Compile/mode-inference.ML)
ML-file (Tools/Predicate-Compile/predicate-compile-proof.ML)
ML-file (Tools/Predicate-Compile/predicate-compile-core.ML)
ML-file (Tools/Predicate-Compile/predicate-compile-data.ML)
ML-file (Tools/Predicate-Compile/predicate-compile-fun.ML)
ML-file (Tools/Predicate-Compile/predicate-compile-pred.ML)
ML-file (Tools/Predicate-Compile/predicate-compile-specialisation.ML)
ML-file (Tools/Predicate-Compile/predicate-compile.ML)

81.1 Set membership as a generator predicate

Introduce a new constant for membership to allow fine-grained control in code equations.

definition contains :: 'a set => 'a => bool
where contains A x ←→ x ∈ A

definition contains-pred :: 'a set => 'a => unit Predicate.pred
where contains-pred A x = (if x ∈ A then Predicate.single () else bot)
lemma pred-of-setE:
  assumes Predicate.eval (pred-of-set A) x
  obtains contains A x
using assms by (simp add: contains-def)

lemma pred-of-setI: contains A x ==> Predicate.eval (pred-of-set A) x
by (simp add: contains-def)

lemma pred-of-set-eq: pred-of-set A x
by (simp add: contains-def [abs-def] pred-of-set-def o-def)

lemma containsI: x ∈ A ==> contains A x
by (simp add: contains-def)

lemma containsE: assumes contains A x
  obtains A' x' where A = A' x = x' x ∈ A
using assms by (simp add: contains-def)

lemma contains-predI: contains A x ==> Predicate.eval (contains-pred A x) ()
by (simp add: contains-pred-def contains-def)

lemma contains-predE:
  assumes Predicate.eval (contains-pred A x) y
  obtains contains A x
using assms by (simp add: contains-pred-def contains-def split: if-split-asm)

lemma contains-pred-eq: contains-pred A x

lemma contains-pred-notI:
  ~ contains A x ==> Predicate.eval (Predicate.not-pred (contains-pred A x)) ()
by (simp add: contains-pred-def contains-def not-pred-eq)

setup :
let
  val Fun = Predicate-Compile-Aux.Fun
  val Input = Predicate-Compile-Aux.Input
  val Output = Predicate-Compile-Aux.Output
  val Bool = Predicate-Compile-Aux.Bool
  val io = Fun (Input, Fun (Output, Bool))
  val ii = Fun (Input, Fun (Input, Bool))
in
Core-Data.PredData.map (Graph.new-node
  (const-name (contains),
  Core-Data.PredData {
    pos = Position.thread-data (),
  },
THEORY “Quickcheck-Narrowing”

intros = [(NONE, @{thm containsI})],
elim = SOME @{thm containsE},
preprocessed = true,
function-names = [(Predicate-Compile-Aux.Pred,
[(io, const-name: pred-of-set), (ii, const-name: contains-pred)])],
predfun-data = [
(io, Core-Data.PredfunData {
   elim = @(thm pred-of-setE), intro = @(thm pred-of-setI),
   neg-intro = NONE, definition = @(thm pred-of-set-eq)
}),
(ii, Core-Data.PredfunData {
   elim = @(thm contains-predE), intro = @(thm contains-predI),
   neg-intro = SOME @(thm contains-pred-notI), definition = @(thm contains-pred-eq)
})],
needs-random = []]
end

hide-const (open) contains contains-pred
hide-fact (open) pred-of-setE pred-of-setI pred-of-set-eq
   containsI containsE contains-predI contains-predE contains-pred-eq contains-pred-notI
end

82 Counterexample generator performing narrowing-based testing

theory Quickcheck-Narrowing
imports Quickcheck-Random
keywords find-unused-assms :: diag
begin

82.1 Counterexample generator

82.1.1 Code generation setup

setup : Code-Target.add-derived-target (Haskell-Quickcheck, [(Code-Haskell.target, I))]

code-printing
code-module Typerep → (Haskell-Quickcheck) (module Typerep(Typerep(..)) where
data Typerep = Typerep String [Typerep]
   for type-constructor typerep constant Typerep.Typerep
   | type-constructor typerep → (Haskell-Quickcheck) Typerep.Typerep
   | constant Typerep.Typerep → (Haskell-Quickcheck) Typerep.Typerep


code-reserved Haskell-Quickcheck Typerep

code-printing
type-constructor integer \to (Haskell-Quickcheck) Prelude.Int
| constant 0::integer \to (Haskell-Quickcheck) !(0/:: Prelude.Int)

setup 
let 
val target = Haskell-Quickcheck;
fan print - = Code-Haskell.print-numeral Prelude.Int;
in
  Numeral.add-code const-name (Code-Numeral.Pos) I print target
  #> Numeral.add-code const-name (Code-Numeral.Neg) (~) print target
end

82.1.2 Narrowing’s deep representation of types and terms

datatype (plugins only: code extraction) narrowing-type =
  Narrowing-sum-of-products narrowing-type list list

datatype (plugins only: code extraction) narrowing-term =
  Narrowing-variable integer list narrowing-type
  | Narrowing-constructor integer narrowing-term list

datatype (plugins only: code extraction) (dead 'a) narrowing-cons =
  Narrowing-cons narrowing-type (narrowing-term list \Rightarrow 'a) list

primrec map-cons :: ('a => 'b) => 'a narrowing-cons => 'b narrowing-cons
where
  map-cons f (Narrowing-cons ty cs) = Narrowing-cons ty (map (\c. f \circ c) cs)

82.1.3 From narrowing’s deep representation of terms to HOL.Code-Evaluation’s terms

class partial-term-of = typerep +
  fixes partial-term-of :: 'a itself \Rightarrow narrowing-term \Rightarrow Code-Evaluation.term

lemma partial-term-of-anything: partial-term-of x nt \equiv t
  by (rule eq-reflection) (cases partial-term-of x nt, cases t, simp)

82.1.4 Auxilary functions for Narrowing

consts nth :: 'a list \Rightarrow integer \Rightarrow 'a

code-printing constant nth \to (Haskell-Quickcheck) infixl 9 !!
code-printing constant error → (Haskell-Quickcheck) error

consts toEnum :: integer => char

code-printing constant toEnum → (Haskell-Quickcheck) Prelude.toEnum

consts marker :: char

code-printing constant marker → (Haskell-Quickcheck) "\0"

82.1.5 Narrowing’s basic operations
type-synonym 'a narrowing = integer => 'a narrowing-cons
definition cons :: 'a => 'a narrowing
where
  cons a d = (Narrowing-cons (Narrowing-sum-of-products [[]])) (λa. a)

fun conv :: (narrowing-term list => 'a) list => narrowing-term => 'a
where
  conv cs (Narrowing-variable p -) = error (marker # map toEnum p)
| conv cs (Narrowing-constructor i xs) = (nth cs i) xs

fun non-empty :: narrowing-type => bool
where
  non-empty (Narrowing-sum-of-products ps) = (¬ (List.null ps))

definition apply :: ('a => 'b) narrowing => 'a narrowing => 'b narrowing
where
  apply f a d = (if d > 0 then
    (case f d of Narrowing-cons (Narrowing-sum-of-products ps) cfs ⇒
      case a (d - 1) of Narrowing-cons ta cas ⇒
        let
          shallow = non-empty ta;
          cs = [(λx # xx) ⇒ cf xs (conv cas x), shallow, cf ← cfs]
        in Narrowing-cons (Narrowing-sum-of-products [ta # p. shallow, p ← ps])
    cs)
  else Narrowing-cons (Narrowing-sum-of-products [[]] [])

definition sum :: 'a narrowing => 'a narrowing => 'a narrowing
where
  sum a b d =
    (case a d of Narrowing-cons (Narrowing-sum-of-products ssa) ca ⇒
      case b d of Narrowing-cons (Narrowing-sum-of-products ssb) cb ⇒
        Narrowing-cons (Narrowing-sum-of-products (ssa @ ssb)) (ca @ cb))

lemma [fundef-cong]:
  assumes a d = a’ d b d = b’ d d = d’
  shows sum a b d = sum a’ b’ d’
using assms unfolding sum-def by (auto split: narrowing-cons.split narrowing-type.split)

lemma [fundef-cong]:
assumes f d = f' d
\( \land d' \leq d' \land d' < d \Longrightarrow a d' = a' d' \)
assumes d = d'
shows apply f a d = apply f' a' d'
proof -
  note assms
moreover have \( 0 < d' \Longrightarrow 0 \leq d' - 1 \)
  by (simp add: less-integer-def less-eq-integer-def)
ultimately show \( \text{thesis} \)
  by (auto simp add: apply-def Let-def split: narrowing-cons.split narrowing-type.split)
qed

82.1.6 Narrowing generator type class

class narrowing =
  fixes narrowing :: integer => 'a narrowing-cons

datatype (plugins only: code extraction) property =
  Universal narrowing-type (narrowing-term => property) narrowing-term =>
  Code-Evaluation.term |
  Existential narrowing-type (narrowing-term => property) narrowing-term =>
  Code-Evaluation.term |
  Property bool

definition exists :: ('a :: {narrowing, partial-term-of} => property) => property
where
  exists f = (case narrowing (100 :: integer) of Narrowing-cons ty cs => Existential ty (\t. f (conv cs t)) (partial-term-of (TYPE('a))))

definition all :: ('a :: {narrowing, partial-term-of} => property) => property
where
  all f = (case narrowing (100 :: integer) of Narrowing-cons ty cs => Universal ty (\t. f (conv cs t)) (partial-term-of (TYPE('a))))

82.1.7 class is-testable

The class is-testable ensures that all necessary type instances are generated.
class is-testable

instance bool :: is-testable ..

instance fun :: (\{term-of, narrowing, partial-term-of\}, is-testable) is-testable ..

definition ensure-testable :: 'a :: is-testable => 'a :: is-testable
where
82.1.8 Defining a simple datatype to represent functions in an incomplete and redundant way

datatype (plugins only: code quickcheck-narrowing extraction) (dead 'a, dead 'b) ffun =
    Constant 'b
  | Update 'a 'b ('a, 'b) ffun

primrec eval-ffun :: ('a, 'b) ffun => 'a => 'b
where
    eval-ffun (Constant c) x = c
  | eval-ffun (Update x' y f) x = (if x = x' then y else eval-ffun f x)

hide-type (open) ffun
hide-const (open) Constant Update eval-ffun

datatype (plugins only: code quickcheck-narrowing extraction) (dead 'b) cfun =
    Constant 'b

primrec eval-cfun :: 'b cfun => 'a => 'b
where
    eval-cfun (Constant c) y = c

hide-type (open) cfun
hide-const (open) Constant eval-cfun Abs-cfun Rep-cfun
unfolding Let-def ensure-testable-def ..

82.2  Narrowing for sets

instantiation set :: (narrowing) narrowing
begin

definition narrowing-set = Quickcheck-Narrowing.apply (Quickcheck-Narrowing.cons set) narrowing

instance ..
end

82.3  Narrowing for integers

definition drawn-from :: 'a list ⇒ 'a narrowing-cons
where
    drawn-from xs = Narrowing-cons (Narrowing-sum-of-products (map (λ-. [])) xs) (map (λx -. x) xs)

function around-zero :: int ⇒ int list
where
    around-zero i = (if i < 0 then [] else (if i = 0 then [0] else around-zero (i - 1) @ [i, -i]))
    by pat-completeness auto
termination by (relation measure nat) auto

declare around-zero.simps [simp del]

lemma length-around-zero:
assumes i >= 0
shows length (around-zero i) = 2 * nat i + 1
proof (induct rule: int-ge-induct [OF assms])
case 1
from 1 show ?case by (simp add: around-zero.simps)
next
case (2 i)
from 2 show ?case
    by (simp add: around-zero.simps [of i + 1])
qed

instantiation int :: narrowing
begin

definition narrowing-int d = (let u :: - ⇒ - ⇒ unit) = conv; i = int-of-integer d
    in drawn-from (around-zero i))
instance ..
end
declare [[code drop: partial-term-of :: int itself ⇒ -]]

lemma [code]:
partial-term-of (ty :: int itself) (Narrowing-variable p t) ≡
Code-Evaluation.Free (STR "\{\cdot\}') (Typerep.Typerep (STR "Int.int")) []
partial-term-of (ty :: int itself) (Narrowing-constructor i []) ≡
(if i mod 2 = 0
     then Code-Evaluation.term-of (− (int-of-integer i) div 2)
     else Code-Evaluation.term-of (((int-of-integer i + 1) div 2))
by (rule partial-term-of-anything)+

instantiation integer :: narrowing
begin
definition
narrowing-integer d = (let (u :: - ⇒ - ⇒ unit) = conv; i = int-of-integer d
      in drawn-from (map integer-of-int (around-zero i)))
instance ..
end
declare [[code drop: partial-term-of :: integer itself ⇒ -]]

lemma [code]:
partial-term-of (ty :: integer itself) (Narrowing-variable p t) ≡
partial-term-of (ty :: integer itself) (Narrowing-constructor i []) ≡
(if i mod 2 = 0
     then Code-Evaluation.term-of (− i div 2)
     else Code-Evaluation.term-of (((i + 1) div 2))
by (rule partial-term-of-anything)+

code-printing constant Code-Evaluation.term-of :: integer ⇒ term → (Haskell-Quickcheck)

(let { t = Typerep.Typerep Code\-'Numerals.integer [];
mkFunT s t = Typerep.Typerep fun [s, t];
umT = Typerep.Typerep Num.num [];
mkB i 0 = Generated\-'Code.Const Num.num.Bit0 (mkFunT numT numT numT);
mkB i 1 = Generated\-'Code.Const Num.num.Bit1 (mkFunT numT numT numT);
mkNumeral 0 = Generated\-'Code.Const Num.num.One numT;
mkNumeral i = let { q = i 'Prelude.div' 2; r = i 'Prelude.mod' 2 }
      in Generated\-'Code.App (mkBit r) (mkNumeral q);
mkNumber 0 = Generated\-'Code.Const Groups.zero\-'class.zero t;
82.4 The find-unused-assms command

ML-file ⟨Tools/Quickcheck/find-unused-assms.ML⟩

82.5 Closing up

hide-type narrowing-type narrowing-term narrowing-cons property
hide-const map-cons nth error toEnum marker empty Narrowing-cons conv non-empty
ensure-testable all exists drawn-from around-zero
hide-const (open) Narrowing-variable Narrowing-constructor apply sum cons
hide-fact empty-def cons-def conv simps non-empty simps apply-def sum-def ensure-testable-def
all-def exists-def

end

83 Program extraction for HOL

theory Extraction
imports Option
begin

83.1 Setup

setup :
  Extraction.add-types
  [(bool, ([], NONE))] #>
  Extraction.set-preprocessor (fn thy =>
    Proofterm.rewrite-proof-notypes
    ([], Rewrite-HOL-Proof.elim-cong :: Proof-Rewrite-Rules.rprocs true) o
    Proofterm.rewrite-proof thy
    (Rewrite-HOL-Proof.rews,
      Proof-Rewrite-Rules.rprocs true @ [Proof-Rewrite-Rules.expand-of-class thy])
  )

lemmas [extraction-expand] =
  meta-spec atomize-eq atomize-all atomize-imp atomize-conj
THEORY “Extraction”

allE rev-mp conjE Eq-TrueI Eq-FalseI eqTrueI eqTrueE eq-cong2
notE' impE' impE iffE imp-cong simp-thms eq-True eq-False
induct-forall-eq induct-implies-eq induct-equal-eq induct-conj-eq
induct-atomize induct-atomize' induct-rulify induct-rulify
induct-rulify-fallback induct-trueI

True-implies-equals implies-True-equals TrueE
False-implies-equals implies-False-swap

lemmas [extraction-expand-def] =
HOL.induct-forall-def HOL.induct-implies-def HOL.induct-equal-def
HOL.induct-conj-def

datatype (plugins only: code extraction) sumbool = Left | Right

83.2 Type of extracted program

extract-type

\begin{align*}
\text{typeof}\ (\text{Trueprop}\ P) & \equiv\ \text{typeof}\ P \\
\text{typeof}\ P & \equiv\ \text{Type}\ (\text{TYPE}(\text{Null})) \Rightarrow\ \text{typeof}\ Q & \equiv\ \text{Type}\ (\text{TYPE}(\text{'Q})) \Rightarrow \\
\text{typeof}\ (P \rightarrow Q) & \equiv\ \text{Type}\ (\text{TYPE}(\text{'Q})) \\
\text{typeof}\ Q & \equiv\ \text{Type}\ (\text{TYPE}(\text{Null})) \Rightarrow\ \text{typeof}\ (P \rightarrow Q) & \equiv\ \text{Type}\ (\text{TYPE}(\text{Null})) \\
\text{typeof}\ P & \equiv\ \text{Type}\ (\text{TYPE}(\text{'P})) \Rightarrow\ \text{typeof}\ Q & \equiv\ \text{Type}\ (\text{TYPE}(\text{'Q})) \Rightarrow \\
\text{typeof}\ (P \rightarrow Q) & \equiv\ \text{Type}\ (\text{TYPE}(\text{'P} \Rightarrow \text{'Q})) \\
(\lambda x. \text{typeof}\ (P\ x)) & \equiv (\lambda x. \text{Type}\ (\text{TYPE}(\text{Null}))) \Rightarrow \\
\text{typeof}\ (\forall x. P\ x) & \equiv\ \text{Type}\ (\text{TYPE}(\text{Null})) \\
(\lambda x. \text{typeof}\ (P\ x)) & \equiv (\lambda x. \text{Type}\ (\text{TYPE}(\text{'P}))) \Rightarrow \\
\text{typeof}\ (\forall x::a. P\ x) & \equiv\ \text{Type}\ (\text{TYPE}(\text{'a} \Rightarrow \text{'P})) \\
(\lambda x. \text{typeof}\ (P\ x)) & \equiv (\lambda x. \text{Type}\ (\text{TYPE}(\text{Null}))) \Rightarrow \\
\text{typeof}\ (\exists x::a. P\ x) & \equiv\ \text{Type}\ (\text{TYPE}(\text{'a})) \\
(\lambda x. \text{typeof}\ (P\ x)) & \equiv (\lambda x. \text{Type}\ (\text{TYPE}(\text{'P}))) \Rightarrow \\
\text{typeof}\ (\exists x::a. P\ x) & \equiv\ \text{Type}\ (\text{TYPE}(\text{'a} \times \text{'P})) \\
\text{typeof}\ P & \equiv\ \text{Type}\ (\text{TYPE}(\text{Null})) \Rightarrow\ \text{typeof}\ Q & \equiv\ \text{Type}\ (\text{TYPE}(\text{Null})) \Rightarrow \\
\text{typeof}\ (P \lor Q) & \equiv\ \text{Type}\ (\text{TYPE}(\text{sumbool})) \\
\text{typeof}\ P & \equiv\ \text{Type}\ (\text{TYPE}(\text{Null})) \Rightarrow\ \text{typeof}\ Q & \equiv\ \text{Type}\ (\text{TYPE}(\text{'Q})) \Rightarrow \\
\text{typeof}\ (P \lor Q) & \equiv\ \text{Type}\ (\text{TYPE}(\text{'Q option})) \\
\text{typeof}\ P & \equiv\ \text{Type}\ (\text{TYPE}(\text{'P})) \Rightarrow\ \text{typeof}\ Q & \equiv\ \text{Type}\ (\text{TYPE}(\text{Null})) \Rightarrow \\
\text{typeof}\ (P \lor Q) & \equiv\ \text{Type}\ (\text{TYPE}(\text{'P option})) \\
\text{typeof}\ P & \equiv\ \text{Type}\ (\text{TYPE}(\text{'P})) \Rightarrow\ \text{typeof}\ Q & \equiv\ \text{Type}\ (\text{TYPE}(\text{'Q})) \Rightarrow \\
\end{align*}
THEORY “Extraction”

\[
\text{typeof} (P \lor Q) \equiv \text{Type} (\text{TYPE}('P + 'Q))
\]
\[
\text{typeof} P \equiv \text{Type} (\text{TYPE}(\text{Null})) \implies \text{typeof} Q \equiv \text{Type} (\text{TYPE}'Q) \implies
\]
\[
\text{typeof} (P \land Q) \equiv \text{Type} (\text{TYPE}'P)
\]
\[
\text{typeof} P \equiv \text{Type} (\text{TYPE}'P) \implies \text{typeof} Q \equiv \text{Type} (\text{TYPE}('Q)) \implies
\]
\[
\text{typeof} (P \land Q) \equiv \text{Type} (\text{TYPE}'P) \implies
\]
\[
\text{typeof} (P = Q) \equiv \text{typeof} ((P \rightarrow Q) \land (Q \rightarrow P))
\]
\[
\text{typeof} (x \in P) \equiv \text{typeof} P
\]

83.3 Realizability

realizability
\[
(\text{realizes} \ t \ (\text{Trueprop} \ P)) \equiv (\text{Trueprop} \ (\text{realizes} \ t \ P))
\]
\[
(\text{typeof} P) \equiv (\text{Type} (\text{TYPE}(\text{Null}))) \implies
\]
\[
(\text{realizes} \ t \ (P \rightarrow Q)) \equiv (\text{realizes} \ \text{Null} \ P \rightarrow \text{realizes} \ t \ Q)
\]
\[
(\text{typeof} P) \equiv (\text{Type} (\text{TYPE}'P)) \implies
\]
\[
(\text{typeof} Q) \equiv (\text{Type} (\text{TYPE}(\text{Null}))) \implies
\]
\[
(\text{realizes} \ t \ (P \rightarrow Q)) \equiv (\forall x::'P. \text{realizes} \ x \ P \rightarrow \text{realizes} \ \text{Null} \ Q)
\]
\[
(\text{realizes} \ t \ (P \rightarrow Q)) \equiv (\forall x. \text{realizes} \ x \ P \rightarrow \text{realizes} \ (t \ x) \ Q)
\]
\[
(\lambda x. \text{typeof} \ (P \ x)) \equiv (\lambda x. \text{Type} (\text{TYPE}(\text{Null}))) \implies
\]
\[
(\text{realizes} \ t \ (\forall x. \ P \ x)) \equiv (\forall x. \text{realizes} \ \text{Null} \ (P \ x))
\]
\[
(\text{realizes} \ t \ (\forall x. \ P \ x)) \equiv (\forall x. \text{realizes} \ (t \ x) \ (P \ x))
\]
\[
(\lambda x. \text{typeof} \ (P \ x)) \equiv (\lambda x. \text{Type} (\text{TYPE}(\text{Null}))) \implies
\]
\[
(\text{realizes} \ t \ (\exists x. \ P \ x)) \equiv (\text{realizes} \ \text{Null} \ (P \ t))
\]
\[
(\text{realizes} \ t \ (\exists x. \ P \ x)) \equiv (\text{realizes} \ (\text{snd} \ t) \ (P \ (\text{fst} \ t)))
\]
\[
(\text{typeof} P) \equiv (\text{Type} (\text{TYPE}(\text{Null}))) \implies
\]
\[
(\text{typeof} Q) \equiv (\text{Type} (\text{TYPE}(\text{Null}))) \implies
\]
\[
(\text{realizes} \ t \ (P \lor Q)) \equiv
\]
\[
(\text{case} \ t \ \text{of} \ \text{Left} \Rightarrow \text{realizes} \ \text{Null} \ P \ | \ \text{Right} \Rightarrow \text{realizes} \ \text{Null} \ Q)
\]
\[
(\text{typeof} P) \equiv (\text{Type} (\text{TYPE}(\text{Null}))) \implies
\]
\[
(\text{realizes} \ t \ (P \lor Q)) \equiv
\]
\[
(\text{case} \ t \ \text{of} \ \text{None} \Rightarrow \text{realizes} \ \text{Null} \ P \ | \ \text{Some} \ q \Rightarrow \text{realizes} \ q \ Q)
\]
\[
(\text{typeof} Q) \equiv (\text{Type} (\text{TYPE}(\text{Null}))) \implies
\]
(realizes $t \ (P \lor Q)) \equiv$
(case $t$ of None $\Rightarrow$ realizes Null $Q$ | Some $p$ $\Rightarrow$ realizes $p \ P$)

(remains $t \ (P \lor Q)) \equiv$
(case $t$ of Inl $p$ $\Rightarrow$ realizes $p \ P$ | Inr $q$ $\Rightarrow$ realizes $q \ Q$)

$(\text{typeof } P) \equiv (\text{Type} \ (\text{TYPE}(\text{Null}))) \implies$
(realizes $t \ (P \land Q)) \equiv (\text{realizes} \ \text{Null} \ P \land \text{realizes} \ t \ Q)$

$(\text{typeof } Q) \equiv (\text{Type} \ (\text{TYPE}(\text{Null}))) \implies$
(realizes $t \ (P \land Q)) \equiv (\text{realizes} \ t \ P \land \text{realizes} \ \text{Null} \ Q)$

(realizes $t \ (P \land Q)) \equiv (\text{realizes} \ \text{fst} \ t \ P \land \text{realizes} \ \text{snd} \ t \ Q)$

(typeof $P$) $\equiv$ (Type (TYPE (Null))) $\Rightarrow$
(realizes $t \ (\neg \ P)) \equiv \neg \text{realizes} \ P$

(typeof $Q$) $\equiv$ (Type (TYPE (Null))) $\Rightarrow$
(realizes $t \ (\neg \ P)) \equiv \neg \text{realizes} \ Q$

\[
\text{realizes} \ t \ (P = Q) \equiv \text{realizes} \ (\text{fst} \ t \ P \land \text{realizes} \ \text{snd} \ t \ Q)
\]

\[
\text{realizes} \ t \ (P = Q) \equiv \text{realizes} \ ((P \to Q) \land (Q \to P))
\]

### 83.4 Computational content of basic inference rules

**Theorem disjE-realizer:**
- **Assumptions:** $r$: case $x$ of Inl $p$ $\Rightarrow$ $P \ p$ | Inr $q$ $\Rightarrow$ $Q \ q$
- **And $r1$:** $\forall p. \ P \ p \Rightarrow R \ (f \ p)$ and $r2$: $\forall q. \ Q \ q \Rightarrow R \ (g \ q)$
- **Shows:** $R \ (\text{case } x \ of \ \text{Inl } p \Rightarrow f \ p | \ \text{Inr } q \Rightarrow g \ q)$
- **Proof:** (cases $x$)
  - Case Inl
    - With $r$ show ?thesis by simp (rule $r1$)
  - Next
    - Case Inr
      - With $r$ show ?thesis by simp (rule $r2$)
  qed

**Theorem disjE-realizer2:**
- **Assumptions:** $r$: case $x$ of None $\Rightarrow$ $P \ | \ Some \ q \Rightarrow Q \ q$
- **And $r1$:** $P \Rightarrow R \ f$ and $r2$: $\forall q. \ Q \ q \Rightarrow R \ (g \ q)$
- **Shows:** $R \ (\text{case } x \ of \ \text{None} \Rightarrow f | \ \text{Some } q \Rightarrow g \ q)$
- **Proof:** (cases $x$)
  - Case None
    - With $r$ show ?thesis by simp (rule $r1$)
  - Next
    - Case Some
with r show ?thesis by simp (rule r2)
qed

theorem disjE-realizer3:
  assumes r: case x of Left ⇒ P | Right ⇒ Q
  and r1: P ⇒ R f and r2: Q ⇒ R g
  shows R (case x of Left ⇒ f | Right ⇒ g)
proof (cases x)
  case Left
  with r show ?thesis by simp (rule r1)
next
  case Right
  with r show ?thesis by simp (rule r2)
qed

theorem conjI-realizer:
P p =⇒ Q q =⇒ P (fst (p, q)) ∧ Q (snd (p, q))
by simp

theorem exI-realizer:
P y x =⇒ P (snd (x, y)) (fst (x, y)) by simp

theorem exE-realizer: P (snd p) (fst p) =⇒
(∀ x y. P y x =⇒ Q (f x y)) =⇒ Q (let (x, y) = p in f x y)
by (cases p) (simp add: Let-def)

realizers
impl (P, Q): λpq. pq
  λ(c: -) (d: -) P Q pq (h: -). allI - - c · (λx. impl - - - (h · x))
impl (P): Null
  λ(c: -) P Q (h: -). allI - - c · (λx. impl - - - (h · x))
impl (Q): λq. q λ(c: -) P Q q. impl - - -
impl: Null impl

mp (P, Q): λpq. pq
  λ(c: -) (d: -) P Q pq (h: -) p. mp · - · · (spec · · p · c · h)
mp (P): Null
  λ(c: -) P Q (h: -) p. mp · - · · (spec · · p · c · h)
mp (Q): λq. q λ(c: -) P Q q. mp · - · -
mp: Null mp
allI (P): \( \lambda p \, \lambda x. p \, \lambda (c:\d) \, P \, (d:\d) \, p \) allI \(
\) allI: Null allI

spec (P): \( \lambda x. \, \lambda p \, x \, \lambda (c:\d) \, P \, (d:\d) \, p \) spec \cdot \cdot \cdot x \cdot d

spec: Null spec

exI (P): \( \lambda x. \, \lambda (x, p) \, \lambda (c:\d) \, P \, (d:\d) \, p \) exI-realizer \cdot P \cdot p \cdot x \cdot c \cdot d

exI: \( \lambda x. \, \lambda x \lambda P \, x \) \( (c:\d) \, h:\d \) h

exE (P, Q): \( \lambda p \, pq \) let \( (x, y) = p \) in \( pq \, x \, y \)
\( \lambda (c:\d) \, (d:\d) \, P \, Q \) \( (e:\d) \, pq \) exE-realizer \cdot P \cdot p \cdot Q \cdot pq \cdot c \cdot e \cdot d \cdot h

exE (P): Null
\( \lambda (c:\d) \, (d:\d) \, P \) \( Q \) \( (e:\d) \, pq \, x \) exE: Null

conjI (P, Q): Pair
\( \lambda (c:\d) \, (d:\d) \, P \, Q \) \( (h:\d) \, q \) conjI-realizer \cdot P \cdot p \cdot Q \cdot q \cdot c \cdot d \cdot h

conjI (P): \( \lambda p \)
\( \lambda (c:\d) \, (d:\d) \, P \) \( Q \) \( p \) conjI \cdot \cdot \cdot 

conjI (Q): \( \lambda q \, q \)
\( \lambda (c:\d) \, (d:\d) \, P \, Q \) \( (h:\d) \, q \) conjI \cdot \cdot \cdot h

conjI: Null conjI

conjunct1 (P, Q): fst
\( \lambda (c:\d) \, (d:\d) \, P \, Q \) \( pq \) conjunct1 \cdot \cdot \cdot 

conjunct1 (P): \( \lambda p \)
\( \lambda (c:\d) \, (d:\d) \, P \) \( Q \) \( p \) conjunct1 \cdot \cdot \cdot 

conjunct1 (Q): Null
\( \lambda (c:\d) \, (d:\d) \, P \) \( Q \) \( q \) conjunct1 \cdot \cdot \cdot 

conjunct1: Null conjunct1

conjunct2 (P, Q): snd
\( \lambda (c:\d) \, (d:\d) \, P \) \( Q \) \( pq \) conjunct2 \cdot \cdot \cdot
conject2 \ (P) \ \text{Null}
\ \lambda (c: \cdot) \ P \ Q \ p. \ \text{conject2} \ \cdot \ \cdot \ \cdot$

\text{conject2} \ (Q) \ \lambda p. \ p
\ \lambda (c: \cdot) \ P \ Q \ p. \ \text{conject2} \ \cdot \ \cdot \ \cdot

\text{conject2} \ \text{Null conject2}

\text{disj1} \ (P, Q) : \ \text{Inl}
\ \lambda (c: \cdot) \ (d: \cdot) \ P \ Q \ p. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{sum. case-1} \cdot \ P \cdot \ \cdot \ p \cdot \ \text{arity-type-bool} \cdot \ c \cdot \ d)$

\text{disj1} \ (P) : \ \text{Some}
\ \lambda (c: \cdot) \ P \ Q \ p. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{option. case-2} \cdot \ \cdot \ P \cdot \ \cdot \ p \cdot \ \text{arity-type-bool} \cdot \ c)$

\text{disj1} \ (Q) : \ \text{None}
\ \lambda (c: \cdot) \ P \ Q. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{option. case-1} \cdot \ \cdot \ \cdot \ \text{arity-type-bool} \cdot \ c)$

\text{disj1} : \ \text{Left}
\ \lambda P \ Q. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{sumbool. case-1} \cdot \ \cdot \ \cdot \ \text{arity-type-bool})

\text{disj2} \ (P, Q) : \ \text{Inr}
\ \lambda (d: \cdot) \ (c: \cdot) \ P \ Q \ q. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{sum. case-2} \cdot \ \cdot \ Q \cdot \ q \cdot \ \text{arity-type-bool} \cdot \ c \cdot \ d)$

\text{disj2} \ (P) : \ \text{None}
\ \lambda (c: \cdot) \ Q \ P. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{option. case-1} \cdot \ \cdot \ \cdot \ \text{arity-type-bool} \cdot \ c)$

\text{disj2} \ (Q) : \ \text{Some}
\ \lambda (c: \cdot) \ Q \ P \ q. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{option. case-2} \cdot \ \cdot \ Q \cdot \ q \cdot \ \text{arity-type-bool} \cdot \ c)$

\text{disj2} : \ \text{Right}
\ \lambda Q \ P. \ \text{iffD2} \ \cdot \ \cdot \ \cdot \ (\text{sumbool. case-2} \cdot \ \cdot \ \cdot \ \text{arity-type-bool})

\text{disjE} \ (P, Q, R) : \ \lambda pq \ pr \ qr.
\ \text{(case pq of Inl p \Rightarrow pr p | Inr q \Rightarrow qr q)}
\ \lambda (c: \cdot) \ (d: \cdot) \ (e: \cdot) \ P \ Q \ R \ pq \ (h1: \cdot) \ pr \ (h2: \cdot) \ qr.
\ \text{disjE-realizer} \ \cdot \ \cdot \ \cdot \ pq \cdot R \cdot pr \cdot qr \cdot c \cdot d \cdot e \cdot h1 \cdot h2

\text{disjE} \ (Q, R) : \ \lambda pq \ pr \ qr.
\ \text{(case pq of None \Rightarrow pr | Some q \Rightarrow qr q)}
\ \lambda (c: \cdot) \ (d: \cdot) \ P \ Q \ R \ pq \ (h1: \cdot) \ pr \ (h2: \cdot) \ qr.
\ \text{disjE-realizer2} \ \cdot \ \cdot \ \cdot \ pq \cdot R \cdot pr \cdot qr \cdot c \cdot d \cdot e \cdot h1 \cdot h2

\text{disjE} \ (P, R) : \ \lambda pq \ pr \ qr.
\ \text{(case pq of None \Rightarrow qr | Some p \Rightarrow pr p)}
\ \lambda (c: \cdot) \ (d: \cdot) \ P \ Q \ R \ pq \ (h1: \cdot) \ pr \ (h2: \cdot) \ qr \ (h3: \cdot).
\ \text{disjE-realizer2} \ \cdot \ \cdot \ \cdot \ pq \cdot R \cdot qr \cdot pr \cdot c \cdot d \cdot h1 \cdot h3 \cdot h2
disjE (R): \( \lambda pq \) pr qr.
(case \( pq \) of Left \( \Rightarrow \) pr | Right \( \Rightarrow \) qr)
\[ \lambda (c:) P Q R pq (h1: -) pr (h2: -) qr. \]
disjE-realizer3 \( \cdot \cdot \cdot \cdot \cdot \) pq \( \cdot \) R \( \cdot \) pr \( \cdot \) qr \( \cdot \) c \( \cdot \) h1 \( \cdot \) h2

disjE (P, Q): Null
\[ \lambda (c:) (d:) P Q R pq. \text{disjE-realizer} \cdot \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot c \cdot d \cdot \text{arity-type-bool} \]

disjE (Q): Null
\[ \lambda (c:) P Q R pq. \text{disjE-realizer2} \cdot \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot c \cdot \text{arity-type-bool} \]

disjE (P): Null
\[ \lambda (c:) P Q R pq (h1: -) (h2: -) (h3: -). \]
disjE-realizer2 \( \cdot \cdot \cdot \cdot \cdot \) pq \( \cdot \) (\( \lambda x. R \)) \( \cdot \cdot \cdot \cdot \cdot c \cdot \text{arity-type-bool} \cdot h1 \cdot h3 \cdot h2

disjE: Null
\[ \lambda P Q R pq. \text{disjE-realizer3} \cdot \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot \text{arity-type-bool} \]

FalseE (P): default
\[ \lambda (c:) P. \text{FalseE} \cdot - \]

FalseE: Null FalseE

notI (P): Null
\[ \lambda (c:) P (h: -). \text{allI} \cdot - \cdot - c \cdot (\lambda x. \text{notI} \cdot - \cdot - (h \cdot x)) \]

notI: Null notI

notE (P, R): \( \lambda p. \) default
\[ \lambda (c:) (d:) P R (h: -) p. \text{notE} \cdot - \cdot - (\text{spec} \cdot - \cdot p \cdot c \cdot h) \]

notE (P): Null
\[ \lambda (c:) P R (h: -) p. \text{notE} \cdot - \cdot - (\text{spec} \cdot - \cdot p \cdot c \cdot h) \]

notE (R): default
\[ \lambda (c:) P R. \text{notE} \cdot - \cdot - \]

notE: Null notE

subst (P): \( \lambda s t ps. ps \)
\[ \lambda (c:) s t P (d:) (h:) -) ps. \text{subst} \cdot s \cdot t \cdot P \cdot ps \cdot d \cdot h \]

subst: Null subst

iffD1 (P, Q): fst
\[ \lambda (d:) (c:) Q P pq (h:) -) p. \]
\[ \text{mp} \cdot - \cdot - (\text{spec} \cdot - \cdot p \cdot d \cdot (\text{conjunct1} \cdot - \cdot - h)) \]
iffD1 (P): λp. p
\[ \lambda(c: -) \quad P \quad \lambda(p :: (conjunct1 \cdot \cdot \cdot h)) \]

iffD1 (Q): Null
\[ \lambda(c: -) \quad Q \quad \lambda(p :: (conjunct1 \cdot \cdot \cdot h)) \]

iffD2 (P, Q); snd
\[ \lambda(c: -) \quad P \quad \lambda(p :: (conjunct1 \cdot \cdot \cdot h)) \]

iffD2 (Q): Null
\[ \lambda(c: -) \quad Q \quad \lambda(p :: (conjunct1 \cdot \cdot \cdot h)) \]

iffI (P, Q): Pair
\[ \lambda(c: -) \quad (d: -) \quad P \quad Q \quad \lambda(p :: (conjunct1 \cdot \cdot \cdot h)) \]

iffI (P): λp. p
\[ \lambda(c: -) \quad P \quad \lambda(p :: (conjunct1 \cdot \cdot \cdot h)) \]

iffI (Q): λq. q
\[ \lambda(c: -) \quad P \quad \lambda(q :: (conjunct1 \cdot \cdot \cdot h)) \]

iffI: Null iffI

end

84 Extensible records with structural subtyping

theory Record
84.1 Introduction

Records are isomorphic to compound tuple types. To implement efficient records, we make this isomorphism explicit. Consider the record access/update simplification $\text{alpha } (\text{beta-update } f \text{ rec}) = \text{alpha rec}$ for distinct fields alpha and beta of some record rec with n fields. There are $n^2$ such theorems, which prohibits storage of all of them for large n. The rules can be proved on the fly by case decomposition and simplification in $O(n)$ time. By creating $O(n)$ isomorphic-tuple types while defining the record, however, we can prove the access/update simplification in $O(\log(n)^2)$ time.

The $O(n)$ cost of case decomposition is not because $O(n)$ steps are taken, but rather because the resulting rule must contain $O(n)$ new variables and an $O(n)$ size concrete record construction. To sidestep this cost, we would like to avoid case decomposition in proving access/update theorems.

Record types are defined as isomorphic to tuple types. For instance, a record type with fields 'a, 'b, 'c and 'd might be introduced as isomorphic to 'a × ('b × ('c × 'd)). If we balance the tuple tree to ('a × 'b) × ('c × 'd) then accessors can be defined by converting to the underlying type then using $O(\log(n))$ fst or snd operations. Updators can be defined similarly, if we introduce a fst-update and snd-update function. Furthermore, we can prove the access/update theorem in $O(\log(n))$ steps by using simple rewrites on fst, snd, fst-update and snd-update.

The catch is that, although $O(\log(n))$ steps were taken, the underlying type we converted to is a tuple tree of size $O(n)$. Processing this term type wastes performance. We avoid this for large n by taking each subtree of size K and defining a new type isomorphic to that tuple subtree. A record can now be defined as isomorphic to a tuple tree of these $O(n/K)$ new types, or, if $n > K*K$, we can repeat the process, until the record can be defined in terms of a tuple tree of complexity less than the constant K.

If we prove the access/update theorem on this type with the analogous steps to the tuple tree, we consume $O(\log(n)^2)$ time as the intermediate terms are $O(\log(n))$ in size and the types needed have size bounded by K. To enable this analogous traversal, we define the functions seen below: iso-tuple-fst, iso-tuple-snd, iso-tuple-fst-update and iso-tuple-snd-update. These functions generalise tuple operations by taking a parameter that encapsulates a tuple isomorphism. The rewrites needed on these functions now need an additional assumption which is that the isomorphism works.
These rewrites are typically used in a structured way. They are here presented as the introduction rule isomorphic-tuple.intros rather than as a rewrite rule set. The introduction form is an optimisation, as net matching can be performed at one term location for each step rather than the simplifier searching the term for possible pattern matches. The rule set is used as it is viewed outside the locale, with the locale assumption (that the isomorphism is valid) left as a rule assumption. All rules are structured to aid net matching, using either a point-free form or an encapsulating predicate.

84.2 Operators and lemmas for types isomorphic to tuples

datatype (dead 'a, dead 'b, dead 'c) tuple-isomorphism =
  Tuple-Isomorphism 'a ⇒ 'b × 'c ⇒ 'a

primrec
  repr :: ('a, 'b, 'c) tuple-isomorphism ⇒ 'a ⇒ 'b × 'c where
  repr (Tuple-Isomorphism r a) = r

primrec
  abst :: ('a, 'b, 'c) tuple-isomorphism ⇒ 'b × 'c ⇒ 'a where
  abst (Tuple-Isomorphism r a) = a

definition
  iso-tuple-fst :: ('a, 'b, 'c) tuple-isomorphism ⇒ 'a ⇒ 'b where
  iso-tuple-fst isom = fst ◦ repr isom

definition
  iso-tuple-snd :: ('a, 'b, 'c) tuple-isomorphism ⇒ 'a ⇒ 'c where
  iso-tuple-snd isom = snd ◦ repr isom

definition
  iso-tuple-fst-update ::
    ('a, 'b, 'c) tuple-isomorphism ⇒ ('b ⇒ 'b) ⇒ ('a ⇒ 'a) where
  iso-tuple-fst-update isom f = abst isom ◦ apfst f ◦ repr isom

definition
  iso-tuple-snd-update ::
    ('a, 'b, 'c) tuple-isomorphism ⇒ ('c ⇒ 'c) ⇒ ('a ⇒ 'a) where
  iso-tuple-snd-update isom f = abst isom ◦ apsnd f ◦ repr isom

definition
  iso-tuple-cons ::
    ('a, 'b, 'c) tuple-isomorphism ⇒ 'b ⇒ 'c ⇒ 'a where
  iso-tuple-cons isom = curry (abst isom)

84.3 Logical infrastructure for records

definition
iso-tuple-surjective-proof-assist :: 'a ⇒ 'b ⇒ ('a ⇒ 'b) ⇒ bool where
iso-tuple-surjective-proof-assist x y f ←→ f x = y

definition
iso-tuple-update-accessor-cong-assist ::
   (('b ⇒ 'b) ⇒ ('a ⇒ 'a)) ⇒ ('a ⇒ 'b) ⇒ bool where
iso-tuple-update-accessor-cong-assist upd ac ←→
   (∀v. upd (λx. f (ac v)) v = upd f v) ∧ (∀v. upd id v = v)

definition
iso-tuple-update-accessor-eq-assist ::
   (('b ⇒ 'b) ⇒ ('a ⇒ 'a)) ⇒ ('a ⇒ 'b) ⇒ bool where
iso-tuple-update-accessor-eq-assist upd ac v f v'
   ←→ upd f r = upd f' r'∧ ac v = x ∧ iso-tuple-update-accessor-cong-assist upd ac

lemma update-accessor-congruence-foldE:
assumes uac: iso-tuple-update-accessor-cong-assist upd ac
   and r: r = r' and v: ac r' = v'
   and f: ∃v. v' = v → f v = f' v
shows upd f r = upd f' r'
using uac v [symmetric]
apply (erule (2) update-accessor-congruence-foldE)
done

lemma update-accessor-congruence-unfoldE:
iso-tuple-update-accessor-cong-assist upd ac
   ⇒ r = r' ⇒ ac r' = v' ⇒ (∃v. v' = v → f v = f' v)
   ⇒ upd f r = upd f' r'
apply (erule(2) update-accessor-congruence-foldE)
apply simp
done

lemma iso-tuple-update-accessor-cong-assist-id:
iso-tuple-update-accessor-cong-assist upd ac ⇒ upd id = id
by rule (simp add: iso-tuple-update-accessor-cong-assist-def)

lemma update-accessor-noopE:
assumes uac: iso-tuple-update-accessor-cong-assist upd ac
   and ac: f (ac x) = ac x
shows upd f x = x
using uac
by (simp add: ac iso-tuple-update-accessor-cong-assist-id [OF uac, unfolded id-def]
   cong: update-accessor-congruence-unfoldE [OF uac])

lemma update-accessor-noop-compE:
assumes uac: iso-tuple-update-accessor-cong-assist upd ac
and \( ac : f \ (ac \ x) = ac \ x \)

shows \( \text{upd} \ (g \circ f) \ x = \text{upd} \ g \ x \)

by (simp add: ac cong: update-accessor-congruence-unfoldE[OF uac])

**lemma** update-accessor-cong-assist-idI:
iso-tuple-update-accessor-cong-assist id id
by (simp add: iso-tuple-update-accessor-cong-assist-def)

**lemma** update-accessor-cong-assist-triv:
iso-tuple-update-accessor-cong-assist upd ac \( \Rightarrow \)
iso-tuple-update-accessor-cong-assist upd ac
by assumption

**lemma** update-accessor-accessor-eqE:
iso-tuple-update-accessor-eq-assist upd ac v f v' x \( \Rightarrow \) ac v = x
by (simp add: iso-tuple-update-accessor-eq-assist-def)

**lemma** update-accessor-updator-eqE:
iso-tuple-update-accessor-eq-assist upd ac v f v' x \( \Rightarrow \) upd f v = v'
by (simp add: iso-tuple-update-accessor-eq-assist-def)

**lemma** iso-tuple-update-accessor-eq-assist-idI:
v' = f v \( \Rightarrow \) iso-tuple-update-accessor-eq-assist id id v f v'

**lemma** iso-tuple-update-accessor-eq-assist-triv:
iso-tuple-update-accessor-eq-assist upd ac v f v' x \( \Rightarrow \)
iso-tuple-update-accessor-eq-assist upd ac v f v' x
by assumption

**lemma** iso-tuple-update-accessor-cong-from-eq:
iso-tuple-update-accessor-eq-assist upd ac v f v' x \( \Rightarrow \)
iso-tuple-update-accessor-cong-assist upd ac
by (simp add: iso-tuple-update-accessor-eq-assist-def)

**lemma** iso-tuple-surjective-proof-assistI:
f x = y \( \Rightarrow \) iso-tuple-surjective-proof-assist x y f
by (simp add: iso-tuple-surjective-proof-assist-def)

**lemma** iso-tuple-surjective-proof-assist-idE:
iso-tuple-surjective-proof-assist x y id \( \Rightarrow \) x = y
by (simp add: iso-tuple-surjective-proof-assist-def)

locale isomorphic-tuple =
fixes isom :: \('a, 'b, 'c\) tuple-isomorphism
assumes repr-inv: \( \forall x. \\text{abst} \ \text{isom} \ (\text{repr} \ \text{isom} \ x) = x \)
and abst-inv: \( \forall y. \text{repr} \ \text{isom} \ (\text{abst} \ \text{isom} \ y) = y \)
begin
THEORY “Record”

**Lemma** repr-inj: \( \text{repr isom } x = \text{repr isom } y \iff x = y \)

**By** (auto dest: arg-cong [of \( \text{repr isom } x \text{ repr isom } y \text{ abst isom} \])

**Simp add:** repr-inv

**Lemma** abst-inj: \( \text{abst isom } x = \text{abst isom } y \iff x = y \)

**By** (auto dest: arg-cong [of \( \text{abst isom } x \text{ abst isom } y \text{ repr isom} \])

**Simp add:** abst-inv

**Lemmas** simps = Let-def repr-inv abst-inv repr-inj abst-inj

**Lemma** iso-tuple-access-update-fst-fst:

\[
 f \circ h \circ g = j \circ f \implies \\
 (f \circ (\text{iso-tuple-fst isom}) \circ (\text{iso-tuple-update-fst isom} \circ h) \circ g = \\
 j \circ (f \circ (\text{iso-tuple-fst isom})
\]

**By** (clarsimp simp: iso-tuple-update-fst-def iso-tuple-fst-def simps fun-eq-iff)

**Lemma** iso-tuple-access-update-snd-snd:

\[
 f \circ h \circ g = j \circ f \implies \\
 (f \circ (\text{iso-tuple-snd isom}) \circ (\text{iso-tuple-update-snd isom} \circ h) \circ g = \\
 j \circ (f \circ (\text{iso-tuple-snd isom})
\]

**By** (clarsimp simp: iso-tuple-update-snd-def iso-tuple-snd-def simps fun-eq-iff)

**Lemma** iso-tuple-access-update-fst-snd:

\[
 (f \circ (\text{iso-tuple-fst isom}) \circ (\text{iso-tuple-update-fst isom} \circ h) \circ g = \\
 id \circ (f \circ (\text{iso-tuple-fst isom})
\]

**By** (clarsimp simp: iso-tuple-update-fst-def iso-tuple-fst-def simps fun-eq-iff)

**Lemma** iso-tuple-access-update-snd-fst:

\[
 (f \circ (\text{iso-tuple-snd isom}) \circ (\text{iso-tuple-update-snd isom} \circ h) \circ g = \\
 id \circ (f \circ (\text{iso-tuple-snd isom})
\]

**By** (clarsimp simp: iso-tuple-update-snd-def iso-tuple-snd-def simps fun-eq-iff)

**Lemma** iso-tuple-update-swap-fst-fst:

\[
 h \circ f \circ j \circ g = j \circ g \circ h \circ f \implies \\
 (\text{iso-tuple-update-fst isom} \circ h) \circ f \circ (\text{iso-tuple-update-fst isom} \circ j) \circ g = \\
 (\text{iso-tuple-fst-update isom} \circ j) \circ g \circ (\text{iso-tuple-fst-update isom} \circ h) \circ f
\]

**By** (clarsimp simp: iso-tuple-update-fst-def simps apfst-compose fun-eq-iff)

**Lemma** iso-tuple-update-swap-snd-snd:

\[
 h \circ f \circ j \circ g = j \circ g \circ h \circ f \implies \\
 (\text{iso-tuple-update-snd isom} \circ h) \circ f \circ (\text{iso-tuple-update-snd isom} \circ j) \circ g = \\
 (\text{iso-tuple-snd-update isom} \circ j) \circ g \circ (\text{iso-tuple-snd-update isom} \circ h) \circ f
\]

**By** (clarsimp simp: iso-tuple-update-snd-def simps apsnd-compose fun-eq-iff)

**Lemma** iso-tuple-update-swap-fst-snd:

(To be continued)
(iso-tuple-snd-update isom \circ h) f \circ (iso-tuple-fst-update isom \circ j) g =
(iso-tuple-fst-update isom \circ j) g \circ (iso-tuple-snd-update isom \circ h) f
by (clarsimp simp: iso-tuple-fst-update-def iso-tuple-snd-update-def
    simps fun-eq-iff)

lemma iso-tuple-update-swap-snd-fst:
  (iso-tuple-fst-update isom \circ h) f \circ (iso-tuple-snd-update isom \circ j) g =
  (iso-tuple-snd-update isom \circ j) g \circ (iso-tuple-fst-update isom \circ h) f
by (clarsimp simp: iso-tuple-fst-update-def iso-tuple-snd-update-def simps
    fun-eq-iff)

lemma iso-tuple-update-compose-fst-fst:
  h f \circ j g = k (f \circ g) =
  (iso-tuple-fst-update isom \circ h) f \circ (iso-tuple-fst-update isom \circ j) g =
  (iso-tuple-fst-update isom \circ k) (f \circ g)
by (clarsimp simp: iso-tuple-fst-update-def simps apfst-compose fun-eq-iff)

lemma iso-tuple-update-compose-snd-snd:
  h f \circ j g = k (f \circ g) =
  (iso-tuple-snd-update isom \circ h) f \circ (iso-tuple-snd-update isom \circ j) g =
  (iso-tuple-snd-update isom \circ k) (f \circ g)
by (clarsimp simp: iso-tuple-snd-update-def simps apsnd-compose fun-eq-iff)

lemma iso-tuple-surjective-proof-assist-step:
  iso-tuple-surjective-proof-assist v a (iso-tuple-fst isom \circ f) =
  iso-tuple-surjective-proof-assist v b (iso-tuple-snd isom \circ f) =
  iso-tuple-surjective-proof-assist v (iso-tuple-cons isom a b) f
by (clarsimp simp: iso-tuple-surjective-proof-assist-def simps
    iso-tuple-fst-def iso-tuple-snd-def iso-tuple-cons-def)

lemma iso-tuple-fst-update-accessor-cong-assist:
assumes iso-tuple-update-accessor-cong-assist f g
shows iso-tuple-update-accessor-cong-assist
  (iso-tuple-fst-update isom \circ f) (g \circ iso-tuple-fst isom)
proof --
from assms have f id = id
by (rule iso-tuple-update-accessor-cong-assist-id)
with assms show ?thesis
  by (clarsimp simp: iso-tuple-update-accessor-cong-assist-def simps
      iso-tuple-fst-update-def iso-tuple-fst-def)
qed

lemma iso-tuple-snd-update-accessor-cong-assist:
assumes iso-tuple-update-accessor-cong-assist f g
shows iso-tuple-update-accessor-cong-assist
  (iso-tuple-snd-update isom \circ f) (g \circ iso-tuple-snd isom)
proof --
from assms have f id = id
by (rule iso-tuple-update-accessor-cong-assist-id)
with assms show \(?thesis
  by (clarsimp simp: iso-tuple-update-accessor-cong-assist-def simps
       iso-tuple-snd-update-def iso-tuple-snd-def)
qed

lemma iso-tuple-fst-update-accessor-eq-assist:
  assumes iso-tuple-update-accessor-eq-assist f g a u a' v
  shows iso-tuple-update-accessor-eq-assist
       (iso-tuple-fst-update isom \(\circ\) f) (g \(\circ\) iso-tuple-fst isom)
       (iso-tuple-cons isom a b) u (iso-tuple-cons isom a' b) v
proof –
  from assms have \(f \text{id} = \text{id}\)
  by (auto simp add: iso-tuple-update-accessor-eq-assist-def
        intro: iso-tuple-update-accessor-cong-assist-id)
with assms show \(?thesis
  by (clarsimp simp: iso-tuple-update-accessor-eq-assist-def
        iso-tuple-fst-update-def iso-tuple-fst-def
        iso-tuple-update-accessor-cong-assist-def iso-tuple-cons-def simps)
qed

lemma iso-tuple-snd-update-accessor-eq-assist:
  assumes iso-tuple-update-accessor-eq-assist f g b u b' v
  shows iso-tuple-update-accessor-eq-assist
       (iso-tuple-snd-update isom \(\circ\) f) (g \(\circ\) iso-tuple-snd isom)
       (iso-tuple-cons isom a b) u (iso-tuple-cons isom a' b') v
proof –
  from assms have \(f \text{id} = \text{id}\)
  by (auto simp add: iso-tuple-update-accessor-eq-assist-def
        intro: iso-tuple-update-accessor-cong-assist-id)
with assms show \(?thesis
  by (clarsimp simp: iso-tuple-update-accessor-eq-assist-def
        iso-tuple-snd-update-def iso-tuple-snd-def
        iso-tuple-update-accessor-cong-assist-def iso-tuple-cons-def simps)
qed

lemma iso-tuple-cons-conj-eqI:
  \(a = c \land b = d \land P \longleftrightarrow Q\)
  \(\Longrightarrow\)
  iso-tuple-cons isom a b = iso-tuple-cons isom c d \land P \longleftrightarrow Q
by (clarsimp simp: iso-tuple-cons-def simps)

lemmas intros =
  iso-tuple-access-update-fst-fst
  iso-tuple-access-update-snd-snd
  iso-tuple-access-update-fst-snd
  iso-tuple-access-update-snd-fst
  iso-tuple-update-swap-fst-fst
  iso-tuple-update-swap-snd-snd
  iso-tuple-update-swap-fst-snd
  iso-tuple-update-swap-snd-fst
iso-tuple-update-compose-fst-fst
iso-tuple-update-compose-snd-snd
iso-tuple-surjective-proof-assist-step
iso-tuple-fst-update-accessor-eq-assist
iso-tuple-snd-update-accessor-eq-assist
iso-tuple-fst-update-accessor-cong-assist
iso-tuple-snd-update-accessor-cong-assist
iso-tuple-cons-conj-eqI
end

lemma isomorphic-tuple-intro:
  fixes repr abst
  assumes repr-inj: \( \forall x \, y. \, \text{repr} x = \text{repr} y \iff x = y \)
  and abst-inv: \( \forall z. \, \text{repr} (\text{abst} z) = z \)
  and v: v \equiv \text{Tuple-Isomorphism} \, \text{repr} \, \text{abst}
  shows isomorphic-tuple v
proof
  fix x
  have \( \text{repr} (\text{abst} (\text{repr} x)) = \text{repr} x \)
    by (simp add: abst-inv)
  then show \( \text{Record} \, \text{abst} \, v \, (\text{Record} \, \text{repr} \, v \, x) = x \)
    by (simp add: v repr-inj)
next
  fix y
  show \( \text{Record} \, \text{repr} \, v \, (\text{Record} \, \text{abst} \, v \, y) = y \)
    by (simp add: v) (fact abst-inv)
qed

definition tuple-iso-tuple \equiv \text{Tuple-Isomorphism} \, \text{id} \, \text{id}

lemma tuple-iso-tuple:
  isomorphic-tuple tuple-iso-tuple
  by (simp add: isomorphic-tuple-intro \[ OF \, - \, reflexive \] tuple-iso-tuple-def)

lemma refl-conj-eq: \( Q = R \implies P \land Q \iff P \land R \)
  by simp

lemma iso-tuple-UNIV-I: \( x \in \text{UNIV} \equiv \text{True} \)
  by simp

lemma iso-tuple-True-simp: \( (\text{True} \implies \text{PROP} \, P) \equiv \text{PROP} \, P \)
  by simp

lemma prop-sabst: \( s = t \implies \text{PROP} \, P \, t \implies \text{PROP} \, P \, s \)
  by simp

lemma K-record-comp: \( (\lambda x. \, c) \circ f = (\lambda x. \, c) \)
  by (simp add: comp-def)
84.4 Concrete record syntax

nonterminal
ident and
field-type and
field-types and
field and
fields and
field-update and
field-updates

syntax
-constify :: id => ident 
-constify :: longid => ident 
-field-type :: ident => type => field-type ((2- :/ -))
:: field-type => field-types (-)
-field-types :: field-type => field-types => field-types (-/ -)
-record-type :: field-types => type ((\{[\]::\}))
-record-type-scheme :: field-types => type => type ((\{[\]::\} (2... ::= /-))))
-field :: ident => 'a => field ((2- /= -))
:: field => fields (-)
-fields :: field => fields => fields (-/ -)
-record :: fields => 'a ((\{[\]::\}))
-record-scheme :: fields => 'a => 'a ((\{[\]::\} (2... /= -))))
-field-update :: ident => 'a => field-update ((2- :/ -))
:: field-update => field-updates (-)
-field-updates :: field-update => field-updates => field-updates (-/ -)
-record-update :: 'a => field-updates => 'b (-/(\{[\]::\} [900, 0] 900))

syntax (ASCII)
-record-type :: field-types => type ((\{[\}]-/\}))
-record-type-scheme :: field-types => type => type ((\{[\}]-/\} (2... ::= /- \}))
-record :: fields => 'a ((\{[\}]-/\}))
-record-scheme :: fields => 'a => 'a ((\{[\}]-/\} (2... -= - \}))
-record-update :: 'a => field-updates => 'b (-/(\{[\}]-/\} [900, 0] 900)

84.5 Record package

ML-file (Tools/record.ML)

hide-const (open) Tuple-Isomorphism repr abst iso-tuple-fst iso-tuple-snd
iso-tuple-fst-update iso-tuple-snd-update iso-tuple-cons
iso-tuple-surjective-proof-assist iso-tuple-update-accessor-cong-assist
iso-tuple-update-accessor-eq-assist tuple-iso-tuple

end
85 Greatest common divisor and least common multiple

theory GCD
  imports Groups-List
begin

85.1 Abstract bounded quasi semilattices as common foundation

locale bounded-quasi-semilattice
  = abel-semigroup +
  fixes top :: 'a (⊤) and bot :: 'a (⊥)
  and normalize :: 'a ⇒ 'a
  assumes idem-normalize [simp]: a ∗ a = normalize a
  and normalize-left-idem [simp]: normalize a ∗ b = a ∗ b
  and normalize-idem [simp]: normalize (a ∗ b) = a ∗ b
  and normalize-top [simp]: normalize ⊤ = ⊤
  and normalize-bottom [simp]: normalize ⊥ = ⊥
  and top-left-normalize [simp]: ⊤ ∗ a = normalize a
  and bottom-left-bottom [simp]: ⊥ ∗ a = ⊥
begin

lemma left-idem [simp]:
  a ∗ (a ∗ b) = a ∗ b
  using assoc [of a a b, symmetric] by simp

lemma right-idem [simp]:
  (a ∗ b) ∗ b = a ∗ b
  using left-idem [of b a] by (simp add: ac-simps)

lemma comp-fun-idem: comp-fun-idem f
  by standard (simp-all add: fun-eq-iff ac-simps)

interpretation comp-fun-idem f
  by (fact comp-fun-idem)

lemma top-right-normalize [simp]:
  a ∗ ⊤ = normalize a
  using top-left-normalize [of a] by (simp add: ac-simps)

lemma bottom-right-bottom [simp]:
  a ∗ ⊥ = ⊥
  using bottom-left-bottom [of a] by (simp add: ac-simps)

lemma normalize-right-idem [simp]:
  a ∗ normalize b = a ∗ b
  using normalize-left-idem [of b a] by (simp add: ac-simps)

end
locale bounded-quasi-semilattice-set = bounded-quasi-semilattice

begin

interpretation comp-fun-idem f
by (fact comp-fun-idem)

definition F :: 'a set ⇒ 'a
where
eq-fold: F A = (if finite A then Finite-Set.fold f ⊤ A else ⊥)

lemma infinite [simp]:
infinite A ⇒ F A = ⊥
by (simp add: eq-fold)

lemma set-eq-fold [code]:
F (set xs) = fold f xs ⊤
by (simp add: eq-fold fold-set-fold)

lemma empty [simp]:
F {} = ⊤
by (simp add: eq-fold)

lemma insert [simp]:
F (insert a A) = a * F A
by (cases finite A) (simp-all add: eq-fold)

lemma normalize [simp]:
normalize (F A) = F A
by (induct rule: infinite-finite-induct simp-all)

lemma in-idem:
assumes a ∈ A
shows a * F A = F A
using assms by (induct A rule: infinite-finite-induct)
(auto simp: left-commute [of a])

lemma union:
F (A ∪ B) = F A * F B
by (induct A rule: infinite-finite-induct)
(simp-all add: ac-simps)

lemma remove:
assumes a ∈ A
shows F A = a * F (A - {a})
proof -
from assms obtain B where A = insert a B and a ∉ B
by (blast dest: mk-disjoint-insert)
with assms show ?thesis by simp
**THEORY “GCD”**

```
qed

lemma insert-remove:
  F (insert a A) = a * F (A - {a})
  by (cases a ∈ A) (simp-all: insert-absorb remove)

lemma subset:
  assumes B ⊆ A
  shows F B * F A = F A
  using assms by (simp add: union [symmetric] Un-absorb1)
```

### 85.2 Abstract GCD and LCM

**class gcd = zero + one + dvd +**

- fixes gcd :: 'a ⇒ 'a ⇒ 'a
- and lcm :: 'a ⇒ 'a ⇒ 'a

**class Gcd = gcd +**

- fixes Gcd :: 'a set ⇒ 'a
- and Lcm :: 'a set ⇒ 'a

**syntax**

- **GCD1** :: pttrns ⇒ 'b ⇒ 'b (((3GCD -./ -) [0, 10] 10))
- **GCD** :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((3GCD -∈./ -) [0, 0, 10] 10)
- **LCM1** :: pttrns ⇒ 'b ⇒ 'b ((3LCM -./ -) [0, 10] 10)
- **LCM** :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b ((3LCM -∈./ -) [0, 0, 10] 10)

**translations**

- **GCD x y. f** = GCD x. GCD y. f
- **GCD x. f** = CONST Gcd (CONST range (\lambda x. f))
- **GCD x ∈ A. f** = CONST Gcd ((\lambda x. f) · 'A)
- **LCM x y. f** = LCM x. LCM y. f
- **LCM x. f** = CONST Lcm (CONST range (\lambda x. f))
- **LCM x ∈ A. f** = CONST Lcm ((\lambda x. f) · 'A)

**class semiring-gcd = normalization-semidom + gcd +**

- assumes gcd-dvd1 [iff]: gcd a b dvd a
- and gcd-dvd2 [iff]: gcd a b dvd b
- and gcd-greatest: c dvd a ⇒ c dvd b ⇒ c dvd gcd a b
- and normalize-gcd [simp]: normalize (gcd a b) = gcd a b
- and lcm-gcd: lcm a b = normalize (a * b div gcd a b)

**begin**

**lemma gcd-greatest-iff [simp]:** a dvd gcd b c ↔ a dvd b ∧ a dvd c
  by (blast intro!: gcd-greatest intro: dvd-trans)

**lemma gcd-dvdI1:** a dvd c ⇒ gcd a b dvd c
by (rule dvd-trans) (rule gcd-dvd1)

lemma gcd-dvd2: b dvd c ⟹ gcd a b dvd c
  by (rule dvd-trans) (rule gcd-dvd2)

lemma dvd-gcdD1: a dvd gcd b c ⟹ a dvd b
  using gcd-dvd1 [of b c] by (blast intro: dvd-trans)

lemma dvd-gcdD2: a dvd gcd b c ⟹ a dvd c
  using gcd-dvd2 [of b c] by (blast intro: dvd-trans)

lemma gcd-0-left [simp]: gcd 0 a = normalize a
  by (rule associated-eqI) simp-all

lemma gcd-0-right [simp]: gcd a 0 = normalize a
  by (rule associated-eqI) simp-all

lemma gcd-eq-0-iff [simp]: gcd a b = 0 ⟷ a = 0 ∧ b = 0
  (is ?P ⟷ ?Q)
proof
  assume ?P
  then have 0 dvd gcd a b
    by simp
  then have 0 dvd a and 0 dvd b
    by (blast intro: dvd-trans)+
  then show ?Q
    by simp
next
  assume ?Q
  then show ?P
    by simp
qed

lemma unit-factor-gcd: unit-factor (gcd a b) = (if a = 0 ∧ b = 0 then 0 else 1)
proof (cases gcd a b = 0)
  case True
  then show ?thesis by simp
next
  case False
  have unit-factor (gcd a b) * normalize (gcd a b) = gcd a b
    by (rule unit-factor-mult-normalize)
  then have unit-factor (gcd a b) * gcd a b = gcd a b
    by simp
  then have unit-factor (gcd a b) * gcd a b div gcd a b = gcd a b div gcd a b
    by simp
  with False show ?thesis
    by simp
qed
lemma is-unit-gcd-iff [simp]:
is-unit (gcd a b) ⟷ gcd a b = 1
by (cases a = 0 ∧ b = 0) (auto simp: unit-factor-gcd dest: is-unit-unit-factor)

sublocale gcd: abel-semigroup gcd
proof
fix a b c
show gcd a b = gcd b a
by (rule associated-eqI) simp-all
from gcd-dvd1 have gcd (gcd a b) c dvd a
by (rule dvd-trans) simp
moreover from gcd-dvd1 have gcd (gcd a b) c dvd b
by (rule dvd-trans) simp
ultimately have P1: gcd (gcd a b) c dvd gcd a (gcd b c)
by (auto intro!: gcd-greatest)
from gcd-dvd2 have gcd a (gcd b c) dvd b
by (rule dvd-trans) simp
moreover from gcd-dvd2 have gcd a (gcd b c) dvd c
by (rule dvd-trans) simp
ultimately have P2: gcd a (gcd b c) dvd gcd (gcd a b) c
by (auto intro!: gcd-greatest)
from P1 P2 show gcd (gcd a b) c = gcd a (gcd b c)
by (rule associated-eqI) simp-all
qed

sublocale gcd: bounded-quasi-semilattice gcd 0 1 normalize
proof
show gcd a a = normalize a for a
proof –
  have a dvd gcd a a
by (rule gcd-greatest) simp-all
  then show ?thesis
  by (auto intro: associated-eqI)
qed

lemma gcd-self: gcd a a = normalize a
by (fact gcd.idem-normalize)

lemma gcd-left-idem: gcd a (gcd a b) = gcd a b
by (fact gcd.left-idem)

lemma gcd-right-idem: gcd (gcd a b) b = gcd a b
by (fact gcd.right-idem)
lemma gcdI:
  assumes c dvd a and c dvd b
  and greatest: \( \forall d. d \mbox{ dvd } a \implies d \mbox{ dvd } b \implies d \mbox{ dvd } c \)
  and normalize c = c
  shows c = gcd a b
  by (rule associated-eqI) (auto simp: assms intro: gcd-greatest)

lemma gcd-unique:
  \( d \mbox{ dvd } a \land d \mbox{ dvd } b \land \mbox{ normalize } d = d \land (\forall e. e \mbox{ dvd } a \land e \mbox{ dvd } b \implies e \mbox{ dvd } d) \)
  \( \iff \)
  \( d = \gcd a b \)
  by rule (auto intro: gcdI simp: gcd-greatest)

lemma gcd-dvd-prod: gcd a b dvd k * b
  using mult-dvd-mono[of 1] by auto

lemma gcd-proj2-if-dvd: b dvd a \( \implies \) gcd a b = normalize b
  by (rule gcdI [symmetric]) simp-all

lemma gcd-proj1-if-dvd: a dvd b \( \implies \) gcd a b = normalize a
  by (rule gcdI [symmetric]) simp-all

lemma gcd-proj1-iff: gcd m n = normalize m \( \iff \) m dvd n
  proof
    assume *: gcd m n = normalize m
    show m dvd n
      proof (cases m = 0)
        case True
        with * show ?thesis by simp
      next
        case [simp]: False
        from * have **: m = gcd m n * unit-factor m
          by (simp add: unit-eq-div2)
        show ?thesis
          by (subst **) (simp add: mult-unit-dvd-iff)
      qed
  next
    assume m dvd n
    then show gcd m n = normalize m
      by (rule gcd-proj1-if-dvd)
  qed

lemma gcd-proj2-iff: gcd m n = normalize n \( \iff \) n dvd m
  using gcd-proj1-iff[of n m] by (simp add: ac-simps)

lemma gcd-mult-left: gcd (c * a) (c * b) = normalize (c * gcd a b)
  proof (cases c = 0)
    case True
    then show ?thesis by simp
next
  case False
  then have \( \ast \): \( c \ast \gcd a b \) dvd \( \gcd (c \ast a) (c \ast b) \)
    by (auto intro: gcd-greatest)
  moreover from False \( \ast \) have \( \gcd (c \ast a) (c \ast b) \) dvd \( c \ast \gcd a b \)
    by (metis div-dvd-iff-mult dvd-mult-left gcd-dvd1 gcd-dvd2 gcd-greatest mult-commute)
  ultimately have normalize \( \gcd (c \ast a) (c \ast b) \) = normalize \( c \ast \gcd a b \)
    by (auto intro: associated-eqI)
  then show ?thesis
    by (simp add: normalize-mult)
qed

lemma gcd-mult-right: \( \gcd (a \ast c) (b \ast c) \) = normalize \( \gcd b a \ast c \)
using gcd-mult-left [of c a b]
by (simp add: ac-simps)

lemma dvd-lcm1 [iff]: \( a \) dvd lcm \( a \) \( b \)
by (metis dvd-mult-swap dvd-mult-left dvd-normalize-iff dvd-refl gcd-dvd2 lcm-gcd)

lemma dvd-lcm2 [iff]: \( b \) dvd lcm \( a \) \( b \)
by (metis dvd-div-mult dvd-mult dvd-normalize-iff dvd-refl gcd-dvd1 lcm-gcd)

lemma dvd-lcmI1: \( a \) dvd \( b \) =\( \Rightarrow \) \( a \) dvd lcm \( b \) \( c \)
by (rule dvd-trans) (assumption, blast)

lemma dvd-lcmI2: \( a \) dvd \( c \) =\( \Rightarrow \) \( a \) dvd lcm \( b \) \( c \)
by (rule dvd-trans) (assumption, blast)

lemma lcm-dvdD1: lcm \( a \) \( b \) dvd \( c \) =\( \Rightarrow \) \( a \) dvd \( c \)
using dvd-lcm1 [of a b]
by (blast intro: dvd-trans)

lemma lcm-dvdD2: lcm \( a \) \( b \) dvd \( c \) =\( \Rightarrow \) \( b \) dvd \( c \)
using dvd-lcm2 [of a b]
by (blast intro: dvd-trans)

lemma lcm-least:
  assumes \( a \) dvd \( c \) and \( b \) dvd \( c \)
shows \( \text{lcm} a b \) dvd \( c \)
proof (cases \( c = 0 \))
  case True
  then show ?thesis by simp
next
  case False
  then have \( \ast \): is-unit \( \langle \text{unit-factor} c \rangle \)
    by simp
  show ?thesis
proof (cases gcd \( a \) \( b \) = \( 0 \))
  case True
  with assms show ?thesis by simp
next
  case False
have a * b dvd normalize (c * gcd a b)
using assms by (subst gcd-mult-left [symmetric]) (auto intro!: gcd-greatest simp: mult-ac)
with False have (a * b div gcd a b) dvd c
by (subst div-dvd-iff-mult) auto
thus thesis by (simp add: lcm-gcd)
qed

qed

lemma lcm-least-iff [simp]: lcm a b dvd c ←→ a dvd c ∧ b dvd c
by (blast intro!: lcm-least intro: ded-trans)

lemma normalize-lcm [simp]: normalize (lcm a b) = lcm a b
by (simp add: lcm-gcd dvd-normalize-div)

lemma lcm-0-left [simp]: lcm 0 a = 0
by (simp add: lcm-gcd)

lemma lcm-0-right [simp]: lcm a 0 = 0
by (simp add: lcm-gcd)

lemma lcm-eq-0-iff: lcm a b = 0 ←→ a = 0 ∨ b = 0 (is ?P ←→ ?Q)
proof
assume ?P
then have 0 dvd lcm a b
by simp
also have lcm a b dvd (a * b)
by simp
finally show ?Q
by auto
next
assume ?Q
then show ?P
by auto
qed

lemma zero-eq-lcm-iff: 0 = lcm a b ←→ a = 0 ∨ b = 0
using lcm-eq-0-iff[of a b] by auto

lemma lcm-eq-1-iff [simp]: lcm a b = 1 ←→ is-unit a ∧ is-unit b
by (auto intro: associated-eqI)

lemma unit-factor-lcm: unit-factor (lcm a b) = (if a = 0 ∨ b = 0 then 0 else 1)
using lcm-eq-0-iff[of a b] by (cases lcm a b = 0) (auto simp: lcm-gcd)

sublocale lcm: abel-semigroup lcm
proof
fix a b c
show \( \text{lcm\ a\ b = lcm\ b\ a} \)
by (simp add: lcm-gcd ac-simps normalize-mult dvd-normalize-div)

have \( \text{lcm\ (lcm\ a\ b\ c\ d) = dvd\ lcm\ a\ (lcm\ b\ c)\ and\ lcm\ a\ (lcm\ b\ c)\ dvd\ lcm\ (lcm\ a\ b)\ c} \)
by (auto intro: lcm-least
dvd-trans[of b (lcm b c) lcm a (lcm b c)]
dvd-trans[of c (lcm b c) lcm a (lcm b c)]
dvd-trans[of a lcm a b lcm (lcm a b) c]
dvd-trans[of b lcm a b lcm (lcm a b) c])
then show \( \text{lcm\ (lcm\ a\ b\ c) = lcm\ a\ (lcm\ b\ c)} \)
by (rule associated-eqI) simp-all
qed

sublocale lcm: bounded-quasi-semilattice lcm 1 0 normalize
proof
show \( \text{lcm\ a\ a = normalize\ a\ for\ a} \)
proof –
  have \( \text{lcm\ a\ a\ dvd\ a} \)
  by (rule lcm-least) simp-all
then show \( \text{?thesis} \)
  by (auto intro: associated-eqI)
qed

show \( \text{lcm\ (normalize\ a\ b) = lcm\ a\ b\ for\ a\ b} \)
using dvd-lcm1[of normalize a b] unfolding normalize-dvd-iff
by (auto intro: associated-eqI)

show \( \text{lcm\ 1\ a = normalize\ a\ for\ a} \)
by (rule associated-eqI) simp-all
qed simp-all

lemma lcm-self: \( \text{lcm\ a\ a = normalize\ a} \)
by (fact lcm.idem-normalize)

lemma lcm-left-idem: \( \text{lcm\ a\ (lcm\ a\ b) = lcm\ a\ b} \)
by (fact lcm.left-idem)

lemma lcm-right-idem: \( \text{lcm\ (lcm\ a\ b)\ b = lcm\ a\ b} \)
by (fact lcm.right-idem)

lemma gcd-lcm:
assumes \( a \neq 0\ and\ b \neq 0\)
shows \( \text{gcd\ a\ b = normalize\ (a\ *\ b\ div\ lcm\ a\ b)} \)
proof –
from assms have [simp]: \( a\ *\ b\ div\ gcd\ a\ b\neq 0\)
  by (subst dvd-div-eq-0-iff) auto
let \(?u = unit-factor\ (a\ *\ b\ div\ gcd\ a\ b)\)
have \( \text{gcd\ a\ b\ +\ normalize\ (a\ *\ b\ div\ gcd\ a\ b) = gcd\ a\ b\ +\ ((a\ *\ b\ div\ gcd\ a\ b)\ *\ (1\ div\ ?u))} \)
  by simp
also have \( \ldots = a\ *\ b\ div\ ?u \)
by (subst mult.assoc [symmetric]) auto
also have \ldots \ dvd a \cdot b
  by (subst div-unit-dvd-iff) auto
finally have \text{gcd} \ a \ b \ dvd \ ((a \cdot b) \ div \ lcm \ a \ b)
  by (intro ded-mult-imp-div) (auto simp: lcm-gcd)
moreover have a \cdot b \ div \ lcm \ a \ b \ dvd \ a \ and \ a \cdot b \ div \ lcm \ a \ b \ dvd \ b
  using assms by (subst div-dvd-iff-mult) simp add: lcm-eq-0-iff mult.commute[of b lcm a b])+
ultimately have normalize (gcd a \ b) = normalize (a \cdot b \ div \ lcm \ a \ b)
  apply -
  apply (rule associated-eqI)
  using assms
  apply (auto simp: div-dvd-iff-mult zero-eq-lcm-iff)
done
thus \?thesis by simp
qed

lemma lcm-1-left: lcm 1 \ a = normalize \ a
  by (fact lcm.top-left-normalize)

lemma lcm-1-right: lcm \ a \ 1 = normalize \ a
  by (fact lcm.top-right-normalize)

lemma lcm-mult-left: lcm (c \cdot a) (c \cdot b) = normalize (c \cdot lcm \ a \ b)
proof (cases c = 0)
  case True
  then show \?thesis by simp
next
  case False
  then have \*: lcm (c \cdot a) (c \cdot b) \ dvd \ c \cdot lcm \ a \ b
    by (auto intro: lcm-least)
  moreover have lcm \ a \ b \ dvd \ lcm (c \cdot a) (c \cdot b) \ div \ c
    by (intro lcm-least) (auto intro!: ded-mult-imp-div simp: mult-ac)
  hence c \cdot lcm \ a \ b \ dvd \ lcm (c \cdot a) (c \cdot b)
    using False by (subst (asm) dvd-div-iff-mult) (auto simp: mult-ac intro: dvd-lcmI1)
ultimately have normalize \ (lcm (c \cdot a) (c \cdot b)) = normalize \ (c \cdot lcm \ a \ b)
  by (auto intro: associated-eqI)
then show \?thesis
  by (simp add: normalize-mult)
qed

lemma lcm-mult-right: lcm (a \cdot c) (b \cdot c) = normalize \ (lcm \ b \ a \cdot c)
  using lcm-mult-left [of c a b] by (simp add: ac-simps)

lemma lcm-mult-unit1: is-unit \ a \implies \ lcm \ (b \cdot a) \ c = lcm \ b \ c
  by (rule associated-eqI) (simp-all add: mult-unit-dvd-iff dvd-lcmI1)

lemma lcm-mult-unit2: is-unit \ a \implies \ lcm \ b \ (c \cdot a) = lcm \ b \ c
using \( \text{lcm-mult-unit1} \) [of \( a \) \( c \) \( b \)] by (simp add: \( \text{ac-simps} \))

lemma \( \text{lcm-div-unit1} \): 
\( \text{is-unit} \ a \implies \text{lcm} \ (b \ \text{div} \ a) \ c = \text{lcm} \ b \ c \)
by (erule is-unitE [of - b]) (simp add: \( \text{lcm-mult-unit1} \))

lemma \( \text{lcm-div-unit2} \): \( \text{is-unit} \ a \implies \text{lcm} \ b \ (c \ \text{div} \ a) = \text{lcm} \ b \ c \)
by (erule is-unitE [of - c]) (simp add: \( \text{lcm-mult-unit2} \))

lemma \( \text{normalize-lcm-left} \): \( \text{lcm} \ (\text{normalize} \ a) \ b = \text{lcm} \ a \ b \)
by (fact \( \text{lcm-normalize-left-idem} \))

lemma \( \text{normalize-lcm-right} \): \( \text{lcm} \ a \ (\text{normalize} \ b) = \text{lcm} \ a \ b \)
by (fact \( \text{lcm-normalize-right-idem} \))

lemma \( \text{comp-fun-idem-gcd} \): \( \text{comp-fun-idem} \ \text{gcd} \)
by standard (simp-all add: \( \text{fun-eq-iff ac-simps} \))

lemma \( \text{comp-fun-idem-lcm} \): \( \text{comp-fun-idem} \ \text{lcm} \)
by standard (simp-all add: \( \text{fun-eq-iff ac-simps} \))

lemma \( \text{gcd-dvd-antisym} \): \( \text{gcd} \ a \ b \ \text{dvd} \ \text{gcd} \ c \ d \implies \text{gcd} \ c \ d \ \text{dvd} \ \text{gcd} \ a \ b \implies \text{gcd} \ a \ b = \text{gcd} \ c \ d \)
proof (rule gcdI)
assume \( \ast \): \( \text{gcd} \ a \ b \ \text{dvd} \ \text{gcd} \ c \ d \)
and \( \ast\ast \): \( \text{gcd} \ c \ d \ \text{dvd} \ \text{gcd} \ a \ b \)
have \( \text{gcd} \ c \ d \ \text{dvd} \ c \)
by simp
with \( \ast \) show \( \text{gcd} \ a \ b \ \text{dvd} \ c \)
by (rule dvd-trans)
have \( \text{gcd} \ c \ d \ \text{dvd} \ d \)
by simp
with \( \ast \) show \( \text{gcd} \ a \ b \ \text{dvd} \ d \)
by (rule dvd-trans)
show \( \text{normalize} \ (\text{gcd} \ a \ b) = \text{gcd} \ a \ b \)
by simp
fix \( l \) assume \( l \ \text{dvd} \ c \) and \( l \ \text{dvd} \ d \)
then have \( l \ \text{dvd} \ \text{gcd} \ c \ d \)
by (rule gcd-greatest)
from this and \( \ast\ast \) show \( l \ \text{dvd} \ \text{gcd} \ a \ b \)
by (rule dvd-trans)
qed
declare \( \text{unit-factor-lcm} \) [simp]

lemma \( \text{lcmI} \):
assumes \( a \ \text{dvd} \ c \) and \( b \ \text{dvd} \ c \) and \( \forall d. \ a \ \text{dvd} \ d \implies b \ \text{dvd} \ d \implies c \ \text{dvd} \ d \)
and normalize \( c = c \)
shows \( c = \text{lcm} \ a \ b \)
by (rule associated-eqI) (auto simp: assms intro: lcm-least)

lemma gcd-dvd-lcm [simp]: \( \text{gcd } a \ b \ dvd \ lcm \ a \ b \)
using gcd-dvd2 by (rule dvd-lcmI2)

lemmas lcm-0 = lcm-0-right

lemma lcm-unique:
  \( a \ dvd \ d \ \land \ b \ dvd \ d \ \land \ \text{normalize} \ d = d \ \land \ (\forall e. \ a \ dvd \ e \ \land \ b \ dvd \ e \ \rightarrow \ d \ dvd \ e) \)
\( \iff \) \( d = \text{lcm } a \ b \)
by rule (auto intro: lcmI simp: lcm-least lcm-eq-0-iff)

lemma lcm-proj1-if-dvd:
  assumes \( b \ dvd \ a \)
shows \( \text{lcm } a \ b = \text{normalize } a \)
proof
  have normalize (lcm a b) = normalize a
    by (rule associatedI) (use assms in auto)
  thus ?thesis by simp
qed

lemma lcm-proj2-if-dvd: \( a \ dvd \ b \ \Longrightarrow \ \text{lcm } a \ b = \text{normalize } b \)
using lcm-proj1-if-dvd [of a b] by (simp add: ac-simps)

lemma lcm-proj1-iff: lcm m n = normalize m \( \iff \) \( n \ dvd \ m \)
proof
  assume \( *: \text{lcm } m \ n = \text{normalize } m \)
show \( n \ dvd \ m \)
proof (cases \( m = 0 \))
  case True
  then show ?thesis by simp
next
case [simp]: \( \text{False} \)
  from \( * \) have \( **: \text{lcm } m \ n = \text{unit-factor } m \)
    by (simp add: unit-eq-div2)
  show ?thesis by (subst **) simp
qed

lemma lcm-proj2-iff: lcm m n = normalize n \( \iff \) \( m \ dvd \ n \)
using lcm-proj1-iff [of n m] by (simp add: ac-simps)

lemma gcd-mono: \( a \ dvd \ c \ \Longrightarrow \ b \ dvd \ d \ \Longrightarrow \ \text{gcd } a \ b \ dvd \ \text{gcd } c \ d \)
by (simp add: gcd-dvd11 gcd-dvd12)

lemma lcm-mono: \( a \ dvd \ c \ \Longrightarrow \ b \ dvd \ d \ \Longrightarrow \ \text{lcm } a \ b \ dvd \ \text{lcm } c \ d \)
lemma dvd-productE:
assumes p dvd a * b
obtains x y where p = x * y x dvd a y dvd b
proof (cases a = 0)
case True
thus ?thesis by (intro that[of p 1]) simp-all
next
case False
define x y where x = gcd a p and y = p div x
have p = x * y by (simp add: x-def y-def)
moreover have x dvd a by (simp add: x-def)
moreover from assms have p dvd gcd (b * a) (b * p)
  by (intro gcd-greatest) (simp-all add: mult.commute)
hence p dvd b * gcd a p by (subst (asm) gcd-mult-left) auto
with False have y dvd b
  by (simp add: x-def y-def div-dvd-iff-mult assms)
ultimately show ?thesis by (rule that)
qed

lemma gcd-mult-unit1:
assumes is-unit a shows gcd (b * a) c = gcd b c
proof –
  have gcd (b * a) c dvd b
    using assms dvd-mult-unit-iff by blast
  then show ?thesis
    by (rule gcdI) simp-all
qed

lemma gcd-mult-unit2: is-unit a ==> gcd (c * a) b = gcd b c
using gcd.commute gcd-mult-unit1 by auto

lemma gcd-div-unit1: is-unit a ==> gcd (b div a) c = gcd b c
by (erule is-unitE[of - b]) (simp add: gcd-mult-unit1)

lemma gcd-div-unit2: is-unit a ==> gcd b (c div a) = gcd b c
by (erule is-unitE[of - c]) (simp add: gcd-mult-unit2)

lemma normalize-gcd-left: gcd (normalize a) b = gcd a b
by (fact gcd.normalize-left-idem)

lemma normalize-gcd-right: gcd a (normalize b) = gcd a b
by (fact gcd.normalize-right-idem)

lemma gcd-add1 [simp]: gcd (m + n) n = gcd m n
by (rule gcdI [symmetric]) (simp-all add: dvd-add-left-iff)

lemma gcd-add2 [simp]: gcd m (m + n) = gcd m n
using gcd-add1 [of n m] by (simp add: ac-simps)

lemma gcd-add-mult: \( \gcd m (k \times m + n) = \gcd m n \)
  by (rule gcdI [symmetric]) (simp-all add: dvd-add-right-iff)

end

class ring-gcd = comm-ring-1 + semiring-gcd
begin
lemma gcd-neg1 [simp]: \( \gcd (-a) b = \gcd a b \)
  by (rule sym, rule gcdI) (simp-all add: gcd-greatest)

lemma gcd-neg2 [simp]: \( \gcd a (-b) = \gcd a b \)
  by (rule sym, rule gcdI) (simp-all add: gcd-greatest)

lemma gcd-neg-numeral-1 [simp]: \( \gcd (-\text{numeral} n) a = \gcd (\text{numeral} n) a \)
  by (fact gcd-neg1)

lemma gcd-neg-numeral-2 [simp]: \( \gcd a (-\text{numeral} n) = \gcd a (\text{numeral} n) \)
  by (fact gcd-neg2)

lemma lcm-neg1 [simp]: \( \lcm (-a) b = \lcm a b \)
  by (rule sym, rule lcmI) (simp-all add: lcm-least lcm-eq-0-iff)

lemma lcm-neg2 [simp]: \( \lcm a (-b) = \lcm a b \)
  by (rule sym, rule lcmI) (simp-all add: lcm-least lcm-eq-0-iff)

lemma lcm-neg-numeral-1 [simp]: \( \lcm (-\text{numeral} n) a = \lcm (\text{numeral} n) a \)
  by (fact lcm-neg1)

lemma lcm-neg-numeral-2 [simp]: \( \lcm a (-\text{numeral} n) = \lcm a (\text{numeral} n) \)
  by (fact lcm-neg2)

end

class semiring-Gcd = semiring-gcd + Gcd +
assumes Gcd-dvd: \( a \in A \Longrightarrow \Gcd A \, \text{dvd} \, a \)
  and Gcd-greatest: \( \forall b. \ b \in A \Longrightarrow a \, \text{dvd} \, b \) \( \Longrightarrow a \, \text{dvd} \, \Gcd A \)
  and normalize-Gcd [simp]: \( \text{normalize} (\Gcd A) = \Gcd A \)
assumes dvd-Lcm: \( a \in A \Longrightarrow a \, \text{dvd} \, \lcm A \)
  and Lcm-least: \( \forall b. \ b \in A \Longrightarrow b \, \text{dvd} \, a \) \( \Longrightarrow \lcm A \, \text{dvd} \, a \)
  and normalize-Lcm [simp]: \( \text{normalize} (\lcm A) = \lcm A \)
begin

lemma Lcm-Gcd: Lcm A = Gcd { b. ∀ a∈A. a dvd b}
  by (rule associated-eqI) (auto intro: Gcd-dvd dvd-Lcm Gcd-greatest Lcm-least)

lemma Gcd-Lcm: Gcd A = Lcm { b. ∀ a∈A. b dvd a}
  by (rule associated-eqI) (auto intro: Gcd-dvd dvd-Lcm Gcd-greatest Lcm-least)

lemma Gcd-empty [simp]: Gcd {} = 0
  by (rule dvd-0-left, rule Gcd-greatest) simp

lemma Lcm-empty [simp]: Lcm {} = 1
  by (auto intro: associated-eqI Lcm-least)

lemma Gcd-insert [simp]: Gcd (insert a A) = gcd a (Gcd A)
proof
  have Gcd (insert a A) dvd gcd a (Gcd A)
    by (auto intro: Gcd-dvd Gcd-greatest)
  moreover have gcd a (Gcd A) dvd Gcd (insert a A)
    proof (rule Gcd-greatest)
      fix b
      assume b ∈ insert a A
      then show gcd a (Gcd A) dvd b
        proof
          assume b = a
          then show ?thesis
            by simp
        next
        assume b ∈ A
        then have Gcd A dvd b
          by (rule Gcd-dvd)
        moreover have gcd a (Gcd A) dvd Gcd A
          by simp
        ultimately show ?thesis
          by (blast intro: dvd-trans)
    qed
  qed
  ultimately show ?thesis
    by (auto intro: associated-eqI)
  qed

lemma Lcm-insert [simp]: Lcm (insert a A) = lcm a (Lcm A)
proof (rule sym)
  have lcm a (Lcm A) dvd Lcm (insert a A)
    by (auto intro: dvd-Lcm Lcm-least)
  moreover have Lcm (insert a A) dvd lcm a (Lcm A)
    proof (rule Lcm-least)
      fix b
      assume b ∈ insert a A
  qed
then show \( b \mid \text{lcm} \ a \) (\( \text{lcm} \ A \))

proof
  assume \( b = a \)
  then show ?thesis by simp
next
  assume \( b \in A \)
  then have \( b \mid \text{lcm} \ A \)
    by (rule dvd-Lcm)
  moreover have \( \text{lcm} \ A \mid \text{lcm} \ a \) (\( \text{lcm} \ A \))
    by simp
  ultimately show ?thesis
    by (blast intro: dvd-trans)
qed

ultimately show \( \text{lcm} \ a \) (\( \text{lcm} \ A \)) = \( \text{lcm} \) (\( \text{insert} \ a \ A \))
  by (rule associated-eqI) (simp-all add: lcm-eq-0-iff)

lemma \( \text{lcm-I} \):
  assumes \[ \forall a. \ a \in A \Longrightarrow a \mid b \]
  and \[ \forall c. (\forall a. \ a \in A \Longrightarrow a \mid c) \Longrightarrow b \mid c \]
  and normalize \( b = b \)
  shows \( b = \text{lcm} \ A \)
  by (rule associated-eqI) (auto simp: assms dvd-Lcm intro: Lcm-least)

lemma \( \text{lcm-subset} \): \( A \subseteq B \Longrightarrow \text{lcm} \ A \mid \text{lcm} \ B \)
  by (blast intro: Lcm-least dvd-Lcm)

lemma \( \text{lcm-Un} \): \( \text{lcm} \ (A \cup B) = \text{lcm} \) (\( \text{lcm} \ A \)) (\( \text{lcm} \ B \))

proof
  have \[ \forall d. [\text{lcm} \ A \mid d; \text{lcm} \ B \mid d] \Longrightarrow \text{lcm} \ (A \cup B) \mid d \]
    by (meson UnE Lcm-least dvd-Lcm dvd-trans)
  then show ?thesis
    by (meson Lcm-subset lcm-unique normalize-Lcm sup.cobounded1 sup.cobounded2)
qed

lemma \( \text{Gcd-0-iff} \) \[ \text{simp} \]: \( \text{Gcd} \ A = 0 \iff A \subseteq \{0\} \)
(is \( ?P \iff ?Q \))

proof
  assume ?P
  show ?Q
    proof
      fix \( a \)
      assume \( a \in A \)
      then have \( \text{Gcd} \ A \mid a \)
        by (rule Gcd-dvd)
      with \( ?P \) have \( a = 0 \)
        by simp
      then show \( a \in \{0\} \)
THEORY “GCD”

by simp
qed

next

assume ?Q
have 0 dvd Gcd A
proof (rule Gcd-greatest)
  fix a
  assume a ∈ A
  with ⟨?Q⟩ have a = 0
    by auto
  then show 0 dvd a
    by simp
qed
then show ?P
  by simp
qed

lemma Lcm-1-iff [simp]: Lcm A = 1 ⟷ (∀ a∈A. is-unit a)
(is ?P ⟷ ?Q)
proof
  assume ?P
  show ?Q
  proof
    fix a
    assume a ∈ A
    then have a dvd Lcm A
      by (rule dvd-Lcm)
    with ⟨?P⟩ show is-unit a
      by simp
  qed
next
  assume ?Q
  then have is-unit (Lcm A)
    by (blast intro: Lcm-least)
  then have normalize (Lcm A) = 1
    by (rule is-unit-normalize)
  then show ?P
    by simp
qed

lemma unit-factor-Lcm: unit-factor (Lcm A) = (if Lcm A = 0 then 0 else 1)
proof (cases Lcm A = 0)
  case True
  then show ?thesis
    by simp
next
  case False
  with unit-factor-normalize have unit-factor (normalize (Lcm A)) = 1
    by blast
with False show thesis
  by simp
qed

lemma unit-factor-Gcd: unit-factor (Gcd A) = (if Gcd A = 0 then 0 else 1)
  by (simp add: Gcd-Lcm unit-factor-Lcm)

lemma GcdI:
  assumes \( \bigwedge a. \ a \in A \implies b \ dvd a \)
  \( \bigwedge c. (\bigwedge a. \ a \in A \implies c \ dvd a) \implies c \ dvd b \)
  and normalize b = b
  shows b = Gcd A
  by (rule associated-eqI) (auto simp: assms Gcd-dvd intro: Gcd-greatest)

lemma Gcd-eq-1-I:
  assumes is-unit a and a \in A
  shows Gcd A = 1
proof –
  from assms have is-unit (Gcd A)
    by (blast intro: Gcd-dvd dvd-unit-imp-unit)
  then have normalize (Gcd A) = 1
    by (rule is-unit-normalize)
  then show thesis
    by simp
qed

lemma Lcm-eq-0-I:
  assumes \( 0 \in A \)
  shows Lcm A = 0
proof –
  from assms have \( 0 \ dvd Lcm A \)
    by (rule dvd-Lcm)
  then show thesis
    by simp
qed

lemma Gcd-UNIV [simp]: Gcd UNIV = 1
  using dvd-refl by (rule Gcd-eq-1-I) simp

lemma Lcm-UNIV [simp]: Lcm UNIV = 0
  by (rule Lcm-eq-0-I) simp

lemma Lcm-0-iff:
  assumes \( \text{finite} \ A \)
  shows \( Lcm A = 0 \iff 0 \in A \)
proof (cases A = \{\})
  case True
  then show thesis by simp
next
case False
with assms show ?thesis
  by (induct A rule: finite-ne-induct) (auto simp: lcm-eq-0-iff)
qed

lemma Gcd-image-normalize [simp]: Gcd (normalize ' A) = Gcd A
proof -
  have Gcd (normalize ' A) dvd a if a ∈ A for a
  proof -
    from that obtain B where A = insert a B
    by blast
    moreover have gcd (normalize a) (Gcd (normalize ' B)) dvd normalize a
    by (rule gcd-dvd1)
    ultimately show Gcd (normalize ' A) dvd a
    by simp
  qed
  then have Gcd (normalize ' A) dvd Gcd A and Gcd A dvd Gcd (normalize ' A)
  by (auto intro!: Gcd-greatest intro: Gcd-dvd)
  then show ?thesis
  by (auto intro: associated-eqI)
qed

lemma Gcd-eqI:
  assumes normalize a = a
  assumes ‹∀ b. b ∈ A ⇒ a dvd b›
  and ‹∀ c. (∀ b. b ∈ A ⇒ c dvd b) ⇒ c dvd a›
  shows Gcd A = a
  using assms by (blast intro: associated-eqI Gcd-greatest Gcd-dvd normalize-Gcd)

lemma dvd-GcdD: x dvd Gcd A ⇒ y ∈ A ⇒ x dvd y
  using Gcd-dvd dvd-trans by blast

lemma dvd-Gcd-iff: x dvd Gcd A ⇐⇒ (∀ y ∈ A. x dvd y)
  by (blast dest: dvd-GcdD intro: Gcd-greatest)

lemma Gcd-mult: Gcd ((*) c * A) = normalize (c * Gcd A)
proof (cases c = 0)
  case True
  then show ?thesis by auto
next
case [simp]: False
  have Gcd ((*) c * A) div c dvd Gcd A
  by (intro Gcd-greatest, subst div-dvd-iff-mult)
  (auto intro!: Gcd-greatest Gcd-dvd simp: mult.commute[of - c])
  then have Gcd ((*) c * A) dvd c * Gcd A
  by (subst (asm) div-dvd-iff-mult) (auto intro: Gcd-greatest simp: mult-ac)
  moreover have c * Gcd A dvd Gcd ((*) c * A)
  by (intro Gcd-greatest) (auto intro: mult-dvd-mono Gcd-dvd)
  ultimately have normalize (Gcd ((*) c * A)) = normalize (c * Gcd A)
THEORY "GCD"

by (rule associatedI)
then show ?thesis by simp
qed

lemma Lcm-eqI:
assumes normalize a = a
and \( \forall b. b \in A \implies b \text{ dvd } a \)
and \( \forall c. (\forall b. b \in A \implies b \text{ dvd } c) \implies a \text{ dvd } c \)
shows \( \text{Lcm } A = a \)
using assms by (blast intro: associated-eqI Lcm-least dvd-Lcm normalize-Lcm)

lemma Lcm-dvdD: \( \text{Lcm } A \text{ dvd } x \implies y \in A \implies y \text{ dvd } x \)
using dvd-Lcm dvd-trans by blast

lemma Lcm-dvd-iff: \( \text{Lcm } A \text{ dvd } x \iff (\forall y \in A. y \text{ dvd } x) \)
by (blast dest: Lcm-dvdD intro: Lcm-least)

lemma Lcm-mult:
assumes \( A \neq \{\} \)
shows \( \text{Lcm } ((*) \ c \cdot A) = \text{normalize } (c \ast \text{Lcm } A) \)
proof (cases c = 0)
case True
with assms have \((*) \ c \cdot A = \{0\}\)
by auto
with True show ?thesis by auto
next
case [simp]: False
from assms obtain \( x \) where \( x \in A \)
by blast
have \( c \text{ dvd } c \cdot x \)
by simp
also from \( x \) have \( c \ast x \text{ dvd } \text{Lcm } ((*) \ c \cdot A) \)
by (intro dvd-Lcm) auto
finally have dvd: \( c \text{ dvd } \text{Lcm } ((*) \ c \cdot A) \).
moreover have \( \text{Lcm } A \text{ dvd } \text{Lcm } ((*) \ c \cdot A) \text{ div } c \)
by (intro Lcm-least dvd-mult-imp-div)
(auto intro!: Lcm-least dvd-Lcm simp: mult.commute[of - c])
ultimately have \( c \ast \text{Lcm } A \text{ dvd } \text{Lcm } ((*) \ c \cdot A) \)
by auto
moreover have \( \text{Lcm } ((*) \ c \cdot A) \text{ dvd } c \ast \text{Lcm } A \)
by (intro Lcm-least) (auto intro: mult-dvd-mono dvd-Lcm)
ultimately have normalize \( (c \ast \text{Lcm } A) = \text{normalize } (\text{Lcm } ((*) \ c \cdot A)) \)
by (rule associatedI)
then show ?thesis by simp
qed

lemma Lcm-no-units: \( \text{Lcm } A = \text{Lcm } (A - \{a. \text{is-unit } a\}) \)
proof -
have \( (A - \{a. \text{is-unit } a\}) \cup \{a \in A. \text{is-unit } a\} = A \)
THEORY "GCD"

by blast
then have \( \text{lcm} \ A = \text{lcm} \ (\text{lcm} \ (A - \{a. \text{is-unit} \ a\})) \ (\text{lcm} \ \{a \in A. \text{is-unit} \ a\}) \)
by (simp add: Lcm-Un [symmetric])
also have \( \text{lcm} \ \{a \in A. \text{is-unit} \ a\} = 1 \)
by simp
finally show \(?thesis\)
by simp
qed

lemma Lcm-0-iff': \( \text{lcm} \ A = 0 \iff (\exists l. \ l \neq 0 \land (\forall a \in A. \ a \text{ dvd} \ l)) \)
by (metis Lcm-least dvd-0-left dvd-Lcm)

lemma Lcm-no-multiple: \( (\forall m. \ m \neq 0 \rightarrow (\exists a \in A. \ a \not\text{ dvd} \ m)) \rightarrow \text{lcm} \ A = 0 \)
by (auto simp: Lcm-0-iff')

lemma Lcm-singleton [simp]: \( \text{lcm} \ \{a\} = \text{normalize} \ a \)
by simp

lemma Lcm-2 [simp]: \( \text{lcm} \ \{a, b\} = \text{lcm} \ a \ b \)
by simp

lemma Gcd-1: \( 1 \in A \rightarrow \text{gcd} \ A = 1 \)
by (auto intro!: Gcd-eq-1-I)

lemma Gcd-singleton [simp]: \( \text{gcd} \ \{a\} = \text{normalize} \ a \)
by simp

lemma Gcd-2 [simp]: \( \text{gcd} \ \{a, b\} = \text{gcd} \ a \ b \)
by simp

lemma Gcd-mono:
assumes \( \forall x. \ x \in A \rightarrow f \ x \text{ dvd} \ g \ x \)
shows \( (\text{GCD} \ x \in A. \ f \ x) \text{ dvd} \ (\text{GCD} \ x \in A. \ g \ x) \)
proof (intro Gcd-greatest, safe)
fix \( x \)
assume \( x \in A \)
hence \( (\text{GCD} \ x \in A. \ f \ x) \text{ dvd} \ f \ x \)
by (intro Gcd-dvd) auto
also have \( f \ x \text{ dvd} \ g \ x \)
using \( x \in A \) assms by blast
finally show \( (\text{GCD} \ x \in A. \ f \ x) \text{ dvd} \ldots . \)
qed

lemma Lcm-mono:
assumes \( \forall x. \ x \in A \rightarrow f \ x \text{ dvd} \ g \ x \)
shows \( (\text{LCM} \ x \in A. \ f \ x) \text{ dvd} \ (\text{LCM} \ x \in A. \ g \ x) \)
proof (intro Lcm-least, safe)
fix \( x \)
assume \( x \in A \)
hence \( f \ x \text{ dvd} \ g \ x \) by (rule assms)
also have \( g \ x \text{ dvd} \ (\text{LCM} \ x \in A. \ g \ x) \)
using (x ∈ A) by (intro dvd-Lcm) auto
finally show f x dvd . . . .
qed
end

85.3 An aside: GCD and LCM on finite sets for incomplete gcd rings

category semiring-gcd
begin

sublocale Gcd-fin: bounded-quasi-semilattice-set gcd 0 1 normalize
defines
Gcd-fin (Gcd f in) = Gcd-fin.F :: 'a set ⇒ 'a ..

abbreviation gcd-list :: 'a list ⇒ 'a
where gcd-list xs ≡ Gcd f in (set xs)

sublocale Lcm-fin: bounded-quasi-semilattice-set lcm 1 0 normalize
defines
Lcm-fin (Lcm f in) = Lcm-fin.F ..

abbreviation lcm-list :: 'a list ⇒ 'a
where lcm-list xs ≡ Lcm f in (set xs)

lemma Gcd-fin-dvd:
  a ∈ A ⇒ Gcd f in A dvd a
  by (induct A rule: infinite-finite-induct)
  (auto intro: dvd-trans)

lemma dvd-Lcm-fin:
  a ∈ A ⇒ a dvd Lcm f in A
  by (induct A rule: infinite-finite-induct)
  (auto intro: dvd-trans)

lemma Gcd-fin-greatest:
  a dvd Gcd f in A if finite A and \( \forall b. b ∈ A \implies a \mid b \)
  using that by (induct A) simp-all

lemma Lcm-fin-least:
  Lcm f in A dvd a if finite A and \( \forall b. b ∈ A \implies b \mid a \)
  using that by (induct A) simp-all

lemma gcd-list-greatest:
  a dvd gcd-list bs if \( \forall b. b ∈ set bs \implies a \mid b \)
  by (rule Gcd-fin-greatest) (simp-all add: that)

lemma lcm-list-least:
lemma dvd-Gcd-fin-iff:
  \( b \text{ dvd } \text{Gcd}_{f in} A \iff (\forall a \in A. b \text{ dvd } a) \) if finite \( A \)
  using that by (auto intro: Gcd-fin-greatest Gcd-fin-dvd dvd-trans [of \( b \text{ Gcd}_{f in} A \)])

lemma dvd-gcd-list-iff:
  \( b \text{ dvd } \text{gcd-list } xs \iff (\forall a \in \text{set } xs. b \text{ dvd } a) \)
  by (simp add: dvd-Gcd-fin-iff)

lemma Lcm-fin-dvd-iff:
  \( \text{Lcm}_{f in} A \text{ dvd } b \iff (\forall a \in A. a \text{ dvd } b) \) if finite \( A \)
  using that by (auto intro: Lcm-fin-least dvd-Lcm-fin dvd-trans [of - \( \text{Lcm}_{f in} A \) \( b \)])

lemma lcm-list-dvd-iff:
  \( \text{lcm-list } xs \text{ dvd } b \iff (\forall a \in \text{set } xs. a \text{ dvd } b) \)
  by (simp add: Lcm-fin-dvd-iff)

lemma Gcd-fin-mult:
  \( \text{Gcd}_{f in} (\text{image } (\text{times } b) A) = \text{normalize } (b * \text{Gcd}_{f in} A) \) if finite \( A \)
  using that by induction (auto simp: gcd-mult-left)

lemma Lcm-fin-mult:
  \( \text{Lcm}_{f in} (\text{image } (\text{times } b) A) = \text{normalize } (b * \text{Lcm}_{f in} A) \) if \( A \neq \{\} \)
  proof (cases \( b = 0 \))
    case True
    moreover from that have times 0 :: \( A = \{0\} \)
    by auto
    ultimately show ?thesis
      by simp
  next
  case False
  show ?thesis proof (cases finite \( A \))
    case False
    moreover have inj-on (\( \text{times } b \)) \( A \)
      using \( b \neq 0 \) by (rule inj-on-mult)
    ultimately have infinite (\( \text{times } b \) :: \( A \))
      by (simp add: finite-image-iff)
    with False show ?thesis
      by simp
  next
  case True
  then show ?thesis using that
    by (induct \( A \) rule: finite-ne-induct) (auto simp: lcm-mult-left)
  qed
qed
lemma unit-factor-Gcd-fin:
  unit-factor (Gcd_fin A) = of_bool (Gcd_fin A ≠ 0)
  by (rule normalize-idem-imp-unit-factor-eq) simp

lemma unit-factor-Lcm-fin:
  unit-factor (Lcm_fin A) = of_bool (Lcm_fin A ≠ 0)
  by (rule normalize-idem-imp-unit-factor-eq) simp

lemma is-unit-Gcd-fin-iff
  simp:
  is-unit (Gcd_fin A) ←→ Gcd_fin A = 1
  by (rule normalize-idem-imp-is-unit-iff) simp

lemma is-unit-Lcm-fin-iff
  simp:
  is-unit (Lcm_fin A) ←→ Lcm_fin A = 1
  by (rule normalize-idem-imp-is-unit-iff) simp

lemma Gcd-fin-0-iff:
  Gcd_fin A = 0 ←→ A ⊆ {0} ∧ finite A
  by (induct A rule: infinite-finite-induct) simp-all

lemma Lcm-fin-0-iff:
  Lcm_fin A = 0 ←→ 0 ∈ A if finite A
  using that by (induct A) (auto simp: lcm-eq-0-iff)

lemma Lcm-fin-1-iff:
  Lcm_fin A = 1 ←→ (∀ a ∈ A. is-unit a) ∧ finite A
  by (induct A rule: infinite-finite-induct) simp-all
end

context semiring-Gcd

begin

lemma Gcd-fin-eq-Gcd [simp]:
  Gcd_fin A = Gcd A if finite A for A :: 'a set
  using that by induce simp-all

lemma Gcd-set-eq-fold [code-unfold]:
  Gcd (set xs) = fold gcd xs 0
  by (simp add: Gcd-fin.set-eq-fold [symmetric])

lemma Lcm-fin-eq-Lcm [simp]:
  Lcm_fin A = Lcm A if finite A for A :: 'a set
  using that by induce simp-all

lemma Lcm-set-eq-fold [code-unfold]:
  Lcm (set xs) = fold lcm xs 1
  by (simp add: Lcm-fin.set-eq-fold [symmetric])
85.4 Coprimality

context semiring-gcd
begin

lemma coprime-imp-gcd-eq-1 [simp]:
gcd a b = 1 if coprime a b
proof -
define t r s where t = gcd a b and r = a div t and s = b div t
then have a = t * r and b = t * s
  by simp-all
with that have coprime (t * r) (t * s)
  by simp
then show ?thesis
  by (simp add: t-def)
qed

lemma gcd-eq-1-imp-coprime [dest!]:
coprime a b if gcd a b = 1
proof (rule coprimeI)
  fix c
  assume c dvd a and c dvd b
  then have c dvd gcd a b
    by (rule gcd-greatest)
  with that show is-unit c
    by simp
qed

lemma coprime-iff-gcd-eq-1 [presburger, code]:
coprime a b ←→ gcd a b = 1
by rule (simp-all add: gcd-eq-1-imp-coprime)

lemma is-unit-gcd [simp]:
is-unit (gcd a b) ←→ coprime a b
by (simp add: coprime-iff-gcd-eq-1)

lemma coprime-add-one-left [simp]: coprime (a + 1) a
by (simp add: gcd-eq-1-imp-coprime ac-simps)

lemma coprime-add-one-right [simp]: coprime a (a + 1)
using coprime-add-one-left [of a] by (simp add: ac-simps)

lemma coprime-mult-left-iff [simp]:
coprime (a * b) c ←→ coprime a c ∧ coprime b c
proof
  assume coprime (a * b) c
with coprime-common-divisor [of a * b c]
have *: is-unit d if d dvd a * b and d dvd c for d
using that by blast
have coprime a c
  by (rule coprimeI, rule *) simp-all
moreover have coprime b c
  by (rule coprimeI, rule *) simp-all
ultimately show coprime a c ∧ coprime b c ..
next
assume coprime a c ∧ coprime b c
then have coprime a c coprime b c
  by simp-all
show coprime (a * b) c
proof (rule coprimeI)
  fix d
  assume d dvd a * b
  then obtain r s where d: d = r * s r dvd a s dvd b
    by (rule dvd-productE)
  assume d dvd c
  with d have r * s dvd c
    by simp
  ultimately show is-unit d
    by (simp add: d is-unit-mult-iff)
qed

lemma coprime-mult-right-iff [simp]:
coprime c (a * b) ←→ coprime c a ∧ coprime c b
using coprime-mult-left-iff [of a b c] by (simp add: ac-simps)

lemma coprime-power-left-iff [simp]:
coprime (a ^ n) b ←→ coprime a b ∨ n = 0
proof (cases n = 0)
case True
  then show ?thesis
    by simp
next
case False
  then have n > 0
    by simp
  then show ?thesis
by (induction n rule: nat-induct-non-zero) simp-all

qed

lemma coprime-power-right-iff [simp]:
coprime a (b ^ n) ↔ coprime a b ∨ n = 0
using coprime-power-left-iff [of b n a] by (simp add: ac-simps)

lemma prod-coprime-left:
coprime (∏ i∈A. f i) a if ∏ i. i ∈ A ⇒ coprime (f i) a
using that by (induct A rule: infinite-finite-induct) simp-all

lemma prod-coprime-right:
coprime a (∏ i∈A. f i) if ∏ i. i ∈ A ⇒ coprime a (f i)
using that prod-coprime-left [of A f a] by (simp add: ac-simps)

lemma prod-list-coprime-left:
coprime (∏ x∈set xs) a if ∏ x. x ∈ set xs ⇒ coprime x a
using that by (induct xs) simp-all

lemma prod-list-coprime-right:
coprime a (∏ x∈set xs) if ∏ x. x ∈ set xs ⇒ coprime a x
using that prod-list-coprime-left [of xs a] by (simp add: ac-simps)

lemma coprime-dvd-mult-left-iff:
a dvd b * c ↔ a dvd b if coprime a c

proof
  assume a dvd b
  then show a dvd b * c
    by simp

next
  assume a dvd b * c
  show a dvd b

proof (cases b = 0)
  case True
  then show ?thesis
    by simp

next
  case False
  then have unit: is-unit (unit-factor b)
    by simp
  from coprime a c have gcd (b * a) (b * c) * unit-factor b = b
    by (subst gcd-mult-left) (simp add: ac-simps)
  moreover from (a dvd b * c)
  have a dvd gcd (b * a) (b * c) * unit-factor b
    by (simp add: dvd-mult-unit-iff unit)
  ultimately show ?thesis
    by simp

qed
lemma coprime-dvd-mult-right-iff:
    \( a \mid c \cdot b \iff a \mid b \quad \text{if} \quad \text{coprime} \ a \ c \)
using that coprime-dvd-mult-left-iff \[[a \ c \ b]\] by (simp add: ac-simps)

lemma divides-mult:
    \( a \cdot b \mid c \quad \text{if} \quad a \mid c \quad \text{and} \quad b \mid c \quad \text{and} \quad \text{coprime} \ a \ b \)
proof -
  from \( \langle b \mid c \rangle \) obtain \( b' \) where \( c = b \cdot b' \) ..
  with \( \langle a \mid c \rangle \) have \( a \mid b' \cdot b \)
  by (simp add: ac-simps)
  with \( \langle \text{coprime} \ a \ b \rangle \) have \( a \mid b' \)
  by (simp add: coprime-dvd-mult-left-iff)
  then obtain \( a' \) where \( b' = a \cdot a' \) ..
  with \( \langle c = b \cdot b' \rangle \) have \( c = (a \cdot b) \cdot a' \)
  by (simp add: ac-simps)
  then show \( ?\text{thesis} \) ..
qed

lemma div-gcd-coprime:
    assumes \( a \not= 0 \lor b \not= 0 \)
    shows \( \text{coprime} \ (a \div \gcd a \ b) \ (b \div \gcd a \ b) \)
proof -
  let \( ?g = \gcd a \ b \)
  let \( ?a' = a \div ?g \)
  let \( ?b' = b \div ?g \)
  let \( ?g' = \gcd ?a \ ?b' \)
  have \( \text{dvdg:} \ ?g \mid a \ ?g \mid b \)
  by simp-all
  have \( \text{dvdg':} \ ?g' \mid ?a' \ ?g' \mid ?b' \)
  by simp-all
  from \( \text{dvdg dvdg'} \) obtain \( ka \ kb \ ka' \ kb' \) where
    \( \text{kab:} \ a = ?g \cdot ka \ b = ?g \cdot kb \ ?a' = ?g' \cdot ka' \ ?b' = ?g' \cdot kb' \)
  unfolding dvd-def by blast
  from this \( \text{[symmetric]} \) have \( ?g \cdot ?a' = (\text{?g } \cdot ?g') \cdot \text{?ka' } ?g \cdot ?b' = (\text{?g } \cdot ?g') \cdot \text{?kb'} \)
  by (simp-all add: mult.assoc mult.left-commute [of \( \gcd a \ b \)])
  then have \( \text{dvdgg':} ?g \cdot ?g' \mi a \ ?g' \cdot ?g \mi b \)
  by (auto simp: dvd-mult-div-cancel [OF \( \text{dvdg(1)} \) ] dvd-mult-div-cancel [OF \( \text{dvdg(2)} \) ] dvd-def)
  have \( ?g \not= 0 \)
  using \( \text{assms by simp} \)
  moreover from \( \gcd\text{-greatest} \) \( \text{[OF dvdgg']} \) have \( ?g \cdot ?g' \mi ?g \).
  ultimately show \( ?\text{thesis} \)
    using \( \text{dvd-times-left-cancel-iff} \) \( \text{[of \gcd a b - 1]} \)
    by simp (simp only: coprime-iff-gcd-eq-1)
qed
lemma \textit{gcd-coprime}:
assumes \( c : \gcd a b \neq 0 \)
and \( a : a = a' \ast \gcd a b \)
and \( b : b = b' \ast \gcd a b \)
shows \( \text{coprime} a' b' \)
proof --
from \( c \) have \( a \neq 0 \lor b \neq 0 \)
  by simp
with \( \text{div-gcd-coprime} \) have \( \text{coprime} \ (a \ \text{div} \ \gcd a b) \ (b \ \text{div} \ \gcd a b) \).
also from \( \text{assms} \) have \( a \ \text{div} \ \gcd a b = a' \)
  using \( \text{dvd-div-eq-mult} \ \text{gcd-dvd1} \) by blast
also from \( \text{assms} \) have \( b \ \text{div} \ \gcd a b = b' \)
  using \( \text{dvd-div-eq-mult} \ \text{gcd-dvd1} \) by blast
finally show \( ?\text{thesis} \).
qed

lemma \textit{gcd-coprime-exists}:
assumes \( \gcd a b \neq 0 \)
shows \( \exists a' b'. \ a = a' \ast \gcd a b \land b = b' \ast \gcd a b \land \text{coprime} a' b' \)
proof --
have \( \text{coprime} \ (a \ \text{div} \ \gcd a b) \ (b \ \text{div} \ \gcd a b) \)
  using \( \text{assms} \ \text{div-gcd-coprime} \) by auto
then show \( ?\text{thesis} \)
  by force
qed

lemma \textit{pow-divides-pow-iff} \[\text{simp}\]:
\( a \ ^ n \ \text{dvd} \ b \ ^ n \ \Longleftrightarrow \ a \ \text{dvd} \ b \) if \( n > 0 \)
proof \( \text{(cases} \ \gcd a b = 0) \)
  case \( \text{True} \)
  then show \( ?\text{thesis} \)
    by simp
next
  case \( \text{False} \)
  show \( ?\text{thesis} \)
proof
  let \( ?d = \gcd a b \)
  from \( \text{n > 0} \) obtain \( m : n = \text{Suc} m \)
    by \( \text{(cases} \ n) \ \text{simp-all} \)
  from \( \text{False} \) have \( zn : ?d \cdot n \neq 0 \)
    by \( \text{(rule} \ \text{power-not-zero}) \)
  from \( \text{gcd-coprime-exists} \) \[\text{OF False}\]
  obtain \( a' b' \) where \( ab' : a = a' \ast ?d b = b' \ast ?d \ \text{coprime} a' b' \)
    by blast
  assume \( a \ \text{\_} n \ \text{dvd} \ b \ \text{\_} n \)
  then have \( (a' \ast ?d) \ \text{\_} n \ \text{dvd} \ (b' \ast ?d) \ \text{\_} n \)
    by \( \text{(simp} \ \text{add:} \ ab'(1,2)[\text{symmetric}]) \)
  then have \( ?d \cdot n \ast a' \cdot n \ \text{dvd} \ ?d \cdot n \ast b' \cdot n \)
    by \( \text{(simp} \ \text{only:} \ \text{power-mult-distrib} \ \text{ac-simps}) \)
with \( zn \) have \( a' \cdot n \mid d \cdot b' \cdot n \)
  by simp
then have \( a' \mid d \cdot b' \cdot n \)
  using dvd-trans[of \( a' \cdot n \) \( b' \cdot n \)] by (simp add: \( m \))
then have \( a' \mid d \cdot b' \cdot m \cdot b' \)
  by (simp add: \( m \) ac-simps)
moreover have \( \text{coprime } a' \) \( (b' \cdot n) \)
  using \( \{ \text{coprime } a' \text{ } b' \cdot n \} \)
  by simp
then have \( a' \mid d \cdot b' \cdot m \cdot b' \)
  using \( \{ a' \mid d \cdot b' \cdot n \} \) coprime-dvd-mult-left-iff dvd-mult by blast
with \( ab'(1,2) \) show \( a \mid d \cdot b \)
  by simp
next
assume \( a \mid d \cdot b \)
with \( \{ n > 0 \} \) show \( a \cdot n \mid d \cdot b \cdot n \)
  by (induction rule: nat-induct-non-zero)
  (simp-all add: mult-dvd-mono)
qed

lemma coprime-crossproduct:
  fixes \( a \text{ } b \text{ } c \text{ } d \) :: \text{'}a
  assumes \( \text{coprime } a \text{ } d \) \( \text{and } \text{coprime } b \text{ } c \)
  shows \( \text{normalize } a \ast \text{normalize } c = \text{normalize } b \ast \text{normalize } d \iff \text{normalize } a = \text{normalize } b \land \text{normalize } c = \text{normalize } d \)
  (is \( \text{?lhs } \iff \text{?rhs } \))
proof
  assume \text{?rhs}
  then show \text{?lhs} by simp
next
assume \text{?lhs}
  from \( \text{?lhs} \) have \( \text{normalize } a \mid d \text{ normalize } b \ast \text{normalize } d \)
    by (auto intro: dvdI dest: sym)
  with \( \text{coprime } a \text{ } d \) have \( a \mid d \cdot b \)
    by (simp add: coprime-dvd-mult-left-iff normalize-mult [symmetric])
  from \( \text{?lhs} \) have \( \text{normalize } b \mid d \text{ normalize } a \ast \text{normalize } c \)
    by (auto intro: dvdI dest: sym)
  with \( \text{coprime } b \text{ } c \) have \( b \mid d \cdot a \)
    by (simp add: coprime-dvd-mult-left-iff normalize-mult [symmetric])
  from \( \text{?lhs} \) have \( \text{normalize } c \mid d \text{ normalize } d \ast \text{normalize } b \)
    by (auto intro: dvdI dest: sym simp add: mult.commute)
  with \( \text{coprime } b \text{ } c \) have \( c \mid d \cdot d \)
    by (simp add: coprime-dvd-mult-left-iff normalize-mult [symmetric])
  from \( \text{?lhs} \) have \( \text{normalize } d \mid d \text{ normalize } c \ast \text{normalize } a \)
    by (auto intro: dvdI dest: sym simp add: mult.commute)
  with \( \text{coprime } a \text{ } d \) have \( d \mid d \cdot c \)
    by (simp add: coprime-dvd-mult-left-iff normalize-mult [symmetric])
from (a dvd b) (b dvd a) have normalize a = normalize b
  by (rule associatedI)
moreover from (c dvd d) (d dvd c) have normalize c = normalize d
  by (rule associatedI)
ultimately show ?rhs ..
qed

lemma gcd-mult-left-left-cancel:
gcd (c * a) b = gcd a b if coprime b c
proof -
  have coprime (gcd b (a * c)) c
    by (rule coprimeI) (auto intro: that coprime-common-divisor)
  then have gcd b (a * c) dvd a
    using coprime-dvd-mult-left-iff [of gcd b (a * c) c a]
    by simp
  then show ?thesis
    by (auto intro: associated-eqI simp add: ac-simps)
qed

lemma gcd-mult-left-right-cancel:
gcd (a * c) b = gcd a b if coprime b c
using that gcd-mult-left-left-cancel [of b c a]
by (simp add: ac-simps)

lemma gcd-mult-right-left-cancel:
gcd a (c * b) = gcd a b if coprime a c
using that gcd-mult-left-left-cancel [of a c b]
by (simp add: ac-simps)

lemma gcd-mult-right-right-cancel:
gcd a (b * c) = gcd a b if coprime a c
using that gcd-mult-right-left-cancel [of a c b]
by (simp add: ac-simps)

lemma gcd-exp-weak:
gcd (a ^ n) (b ^ n) = normalize (gcd a b ^ n)
proof (cases a = 0 ∧ b = 0 ∨ n = 0)
  case True
  then show ?thesis
    by (cases n) simp-all
next
  case False
  then have coprime (a dvd gcd a b) (b dvd gcd a b) and n > 0
    by (auto intro: div-gcd-coprime)
  then have coprime ((a dvd gcd a b) ^ n) ((b dvd gcd a b) ^ n)
    by simp
  then have 1 = gcd ((a dvd gcd a b) ^ n) ((b dvd gcd a b) ^ n)
    by simp
  then have normalize (gcd a b ^ n) = normalize (gcd a b ^ n * ...
by simp
also have \ldots = \gcd (\gcd a b \sim n \ast (a \div \gcd a b) \sim n) (\gcd a b \sim n \ast (b \div \gcd a b) \sim n)
  by (rule gcd-mult-left [symmetric])
also have (\gcd a b) \sim n \ast (a \div \gcd a b) \sim n = a \sim n
  by (simp add: ac-simps dvd-power, dvd-power-same)
also have (\gcd a b) \sim n \ast (b \div \gcd a b) \sim n = b \sim n
  by (simp add: ac-simps dvd-power, dvd-power-same)
finally show ?thesis by simp
qed

lemma division-decomp:
  assumes a \mid b \ast c
  shows \exists b' c'. a = b' \ast c' \land b' \mid d \land c' \mid d
proof (cases \gcd a b = 0)
case True
then have a = 0 \land b = 0
  by simp
then have a = 0 \ast c \land 0 \mid d \land c \mid d
  by simp
then show ?thesis by blast
next
case False
let \?d = \gcd a b
from \gcd-coprime-exists [OF False]
  obtain a' b' where ab': a = a' \ast \?d \ast b = b' \ast \?d
  by blast
from ab'(1) have a' \mid d \ast a ..
  with assms have a' \mid d \land b \ast c
    by blast
from assms ab'(1,2) have a' \ast \?d \mid d \land (b' \ast \?d) \ast c
  by simp
then have \?d \ast a' \mid d \ast (b' \ast c)
    by (simp add: mult-ac)
with \(\?d \neq 0\) have a' \mid d \land b' \ast c
  by simp
then have a' \mid d \ast c
  using coprime a' b' by (simp add: coprime-dvd-mult-right-iff)
with ab'(1) have a = \?d \ast a' \land \?d \mid d \land a' \mid d \land c
  by (simp add: ac-simps)
then show ?thesis by blast
qed

lemma lcm-coprime: coprime a b \implies \lcm a b = \normalize (a \ast b)
  by (subst lcm-gcd) simp

end

context ring-gcd
THEORY “GCD”

begin

lemma coprime-minus-left-iff [simp]:
  coprime (− a) b ←→ coprime a b
  by (rule; rule coprimeI) (auto intro: coprime-common-divider)

lemma coprime-minus-right-iff [simp]:
  coprime a (− b) ←→ coprime a b
  using coprime-minus-left-iff [of b a] by (simp add: ac-simps)

lemma coprime-diff-one-left [simp]:
  coprime (a − 1) a
  using coprime-add-one-right [of a − 1] by simp

lemma coprime-diff-one-right [simp]:
  coprime a (a − 1)
  using coprime-diff-one-left [of a] by (simp add: ac-simps)

end

context semiring-Gcd

begin

lemma Lcm-coprime:
  assumes finite A
    and A ≠ {} and \( \forall a b. a \in A \implies b \in A \implies a \neq b \implies \text{coprime } a \ b \)
  shows Lcm A = normalize (\( \prod A \))
  using assms
  proof (induct rule: finite-ne-induct)
    case singleton
    then show ?case by simp
  next
    case (insert a A)
    have Lcm (insert a A) = lcm a (Lcm A)
      by simp
    also from insert have Lcm A = normalize (\( \prod A \))
      by blast
    also have lcm a ... = lcm a (\( \prod A \))
      by (cases \( \prod A = 0 \)) (simp-all add: lcm-div-unit2)
    also from insert have coprime a (\( \prod A \))
      by (subst coprime-commute, intro prod-coprime-left) auto
    with insert have lcm a (\( \prod A \)) = normalize (\( \prod (\text{insert } a \ A) \))
      by (simp add: lcm-coprime)
    finally show ?case .
  qed

lemma Lcm-coprime':
  card A ≠ 0 \implies
  (\( \forall a b. a \in A \implies b \in A \implies a \neq b \implies \text{coprime } a \ b \)) \implies
  Lcm A = normalize (\( \prod A \))
85.5 GCD and LCM for multiplicative normalisation functions

class semiring-gcd-mult-normalize = semiring-gcd + normalization-semidom-multiplicative

begin

lemma mult-gcd-left: c * gcd a b = unit-factor c * gcd (c * a) (c * b)
by (simp add: gcd-mult-left normalize-mult mult.assoc [symmetric])

lemma mult-gcd-right: gcd a b * c = gcd (a * c) (b * c) * unit-factor c
using mult-gcd-left [of c a b] by (simp add: ac-simps)

lemma gcd-mult-distrib': normalize c * gcd a b = gcd (c * a) (c * b)
by (subst gcd-mult-left) (simp-all add: normalize-mult)

lemma gcd-mult-distrib: k * gcd a b = gcd (k * a) (k * b) * unit-factor k
proof -
  have normalize k * gcd a b = gcd (k * a) (k * b)
    by (simp add: gcd-mult-distrib')
  then have normalize k * unit-factor k * gcd a b = gcd (k * a) (k * b) * unit-factor k
    by simp
  then have normalize k * unit-factor k * gcd a b = gcd (k * a) (k * b) * unit-factor k
    by (simp only: ac-simps)
  then show ?thesis
    by simp
qed

lemma gcd-mult-lcm [simp]: gcd a b * lcm a b = normalize a * normalize b
by (simp add: lcm-gcd normalize-mult dvd-normalize-div)

lemma lcm-mult-gcd [simp]: lcm a b * gcd a b = normalize a * normalize b
using gcd-mult-lcm [of a b] by (simp add: ac-simps)

lemma mult-lcm-left: c * lcm a b = unit-factor c * lcm (c * a) (c * b)
by (simp add: lcm-mult-left mult.assoc [symmetric] normalize-mult)

lemma mult-lcm-right: lcm a b * c = lcm (a * c) (b * c) * unit-factor c
using mult-lcm-left [of c a b] by (simp add: ac-simps)

lemma lcm-gcd-prod: lcm a b * gcd a b = normalize (a * b)
by (simp add: lcm-gcd dvd-normalize-div normalize-mult)

lemma lcm-mult-distrib': normalize c * lcm a b = lcm (c * a) (c * b)
by (subst lcm-mult-left) (simp add: normalize-mult)

lemma lcm-mult-distrib: 
  k * lcm a b = lcm (k * a) (k * b) * unit-factor k
proof
  have normalize k * lcm a b = lcm (k * a) (k * b)
    by (simp add: lcm-mult-distrib')
  then have normalize k * lcm a b * unit-factor k = lcm (k * a) (k * b) * unit-factor k
    by simp
  then have normalize k * unit-factor k * lcm a b = lcm (k * a) (k * b) * unit-factor k
    by simp
  then show ?thesis
    by simp
qed

lemma coprime-crossproduct':
  fixes a b c d
  assumes b ≠ 0
  assumes unit-factors: unit-factor b = unit-factor d
  assumes coprime: coprime a b coprime c d
  shows a * d = b * c ←→ a = c ∧ b = d
proof safe
  assume eq: a * d = b * c
  hence normalize a * normalize d = normalize c * normalize b
    by (simp only: normalize-mult [symmetric] mult-ac)
  with coprime have normalize b = normalize d
    by (subst (asm) coprime-crossproduct) simp-all
  from this and unit-factors show b = d
    by (rule normalize-unit-factor-eqI)
  from eq have a * d = c * d by (simp only: b = d; mult-ac)
  with ⟨b ≠ 0⟩ ⟨b = d⟩ show a = c by simp
qed (simp-all add: mult-ac)

lemma gcd-exp [simp]:
  gcd (a ^ n) (b ^ n) = gcd a b ^ n
using gcd-exp-weak[of a n b] by (simp add: normalize-power)

end

85.6 GCD and LCM on nat and int

instantiation nat :: gcd
begin

  fun gcd-nat :: nat ⇒ nat ⇒ nat
    where gcd-nat x y = (if y = 0 then x else gcd y (x mod y))

definition lcm-nat :: nat ⇒ nat ⇒ nat
where \( \text{lcm-nat } x \ y = x \ast y \div (\text{gcd } x \ y) \)

instance ..

end

instantiation int :: gcd
begin

definition gcd-int :: int \Rightarrow int \Rightarrow int
where gcd-int \( x \ y = \text{int } (\text{gcd } \text{nat } |x|) \ (\text{nat } |y|) \)

definition lcm-int :: int \Rightarrow int \Rightarrow int
where lcm-int \( x \ y = \text{int } (\text{lcm } \text{nat } |x|) \ (\text{nat } |y|) \)

instance ..

end

lemma gcd-int-int-eq [simp]:
\( \text{gcd } \text{int } m \ (\text{int } n) = \text{int } (\text{gcd } m \ n) \)
by (simp add: gcd-int-def)

lemma gcd-nat-abs-left-eq [simp]:
\( \text{gcd } \text{nat } |k| \ n = \text{nat } (\text{gcd } \text{int } n) \)
by (simp add: gcd-int-def)

lemma gcd-nat-abs-right-eq [simp]:
\( \text{gcd } n \ (\text{nat } |k|) = \text{nat } (\text{gcd } \text{int } n) \ k \)
by (simp add: gcd-int-def)

lemma abs-gcd-int [simp]:
\( |\text{gcd } x \ y| = \text{gcd } x \ y \)
for \( x \ y :: \text{int} \)
by (simp only: gcd-int-def)

lemma gcd-abs1-int [simp]:
\( \text{gcd } |x| \ y = \text{gcd } x \ y \)
for \( x \ y :: \text{int} \)
by (simp only: gcd-int-def) simp

lemma gcd-abs2-int [simp]:
\( \text{gcd } x \ |y| = \text{gcd } x \ y \)
for \( x \ y :: \text{int} \)
by (simp only: gcd-int-def) simp

lemma lcm-int-int-eq [simp]:
\( \text{lcm } \text{int } m \ (\text{int } n) = \text{int } (\text{lcm } m \ n) \)
by (simp add: lcm-int-def)
lemma lcm-nat-abs-left-eq [simp]:
\[ \text{lcm} (\text{nat} \mid k) \cdot n = \text{nat} (\text{lcm} \ k \ (\text{int} \ n)) \]
by (simp add: lcm-int-def)

lemma lcm-nat-abs-right-eq [simp]:
\[ \text{lcm} \ n \ (\text{nat} \mid k) = \text{nat} (\text{lcm} \ (\text{int} \ n) \ k) \]
by (simp add: lcm-int-def)

lemma lcm-abs1-int [simp]:
\[ \text{lcm} \mid x \mid y = \text{lcm} \ x \ y \]
for \( x \ y :: \text{int} \)
by (simp only: lcm-int-def) simp

lemma lcm-abs2-int [simp]:
\[ \text{lcm} \ x \mid y \mid = \text{lcm} \ x \ y \]
for \( x \ y :: \text{int} \)
by (simp only: lcm-int-def) simp

lemma abs-lcm-int [simp]:
\[ \mid \text{lcm} \ i \ j \mid = \text{lcm} \ i \ j \]
for \( i \ j :: \text{int} \)
by (simp only: lcm-int-def)

lemma gcd-nat-induct [case-names base step]:
fixes \( m \ n :: \text{nat} \)
assumes \( \forall m. \ P m 0 \)
\( \land \forall m. \ 0 < n \Longrightarrow P n (m \ mod \ n) \Longrightarrow P m n \)
shows \( P m n \)
proof (induction \( m \ n \) rule: gcd-nat.induct)
case (1 \( x \ y \))
then show \( \forall \text{case} \)
using assms neq0-conv by blast
qed

lemma gcd-neg1-int [simp]: \( \text{gcd} \ (\text{−} x) \ y = \text{gcd} \ x \ y \)
for \( x \ y :: \text{int} \)
by (simp only: gcd-int-def) simp

lemma gcd-neg2-int [simp]: \( \text{gcd} \ x \ (\text{−} y) = \text{gcd} \ x \ y \)
for \( x \ y :: \text{int} \)
by (simp only: gcd-int-def) simp

lemma gcd-cases-int:
fixes \( x \ y :: \text{int} \)
assumes \( x \geq 0 \Longrightarrow y \geq 0 \Longrightarrow P \ (\text{gcd} \ x \ y) \)
\( \land x \geq 0 \Longrightarrow y \leq 0 \Longrightarrow P \ (\text{gcd} \ x \ (\text{−} y)) \)
\( \land x \leq 0 \Longrightarrow y \geq 0 \Longrightarrow P \ (\text{gcd} \ (\text{−} x) \ y) \)
\( \land x \leq 0 \Longrightarrow y \leq 0 \Longrightarrow P \ (\text{gcd} \ (\text{−} x) \ (\text{−} y)) \)
shows \( P \ (\text{gcd} \ x \ y) \)
THEORY "GCD"

using assms by auto arith

lemma gcd-ge-0-int [simp]: gcd (x::int) y >= 0
  for x y :: int
  by (simp add: gcd-int-def)

lemma lcm-neg1-int: lcm (- x) y = lcm x y
  for x y :: int
  by (simp only: lcm-int-def) simp

lemma lcm-neg2-int: lcm x (- y) = lcm x y
  for x y :: int
  by (simp only: lcm-int-def) simp

lemma lcm-cases-int:
  fixes x y :: int
  assumes x ≥ 0 ⇒ y ≥ 0 ⇒ P (lcm x y)
  and x ≥ 0 ⇒ y ≤ 0 ⇒ P (lcm x (- y))
  and x ≤ 0 ⇒ y ≥ 0 ⇒ P (lcm (- x) y)
  and x ≤ 0 ⇒ y ≤ 0 ⇒ P (lcm (- x) (- y))
  shows P (lcm x y)
  using assms by (auto simp: lcm-neg1-int lcm-neg2-int) arith

lemma lcm-ge-0-int [simp]: lcm x y ≥ 0
  for x y :: int
  by (simp only: lcm-int-def)

lemma gcd-0-nat: gcd x 0 = x
  for x :: nat
  by simp

lemma gcd-0-int [simp]: gcd x 0 = |x|
  for x :: int
  by (auto simp: gcd-int-def)

lemma gcd-0-left-nat: gcd 0 x = x
  for x :: nat
  by simp

lemma gcd-0-left-int [simp]: gcd 0 x = |x|
  for x :: int
  by (auto simp: gcd-int-def)

lemma gcd-red-nat: gcd x y = gcd y (x mod y)
  for x y :: nat
  by (cases y = 0) auto

Weaker, but useful for the simplifier.

lemma gcd-non-0-nat: y ≠ 0 ⇒ gcd x y = gcd y (x mod y)
THEORY "GCD"

for x y :: nat
by simp

lemma gcd-1-nat [simp]: gcd m 1 = 1
for m :: nat
by simp

lemma gcd-Suc-0 [simp]: gcd m (Suc 0) = Suc 0
for m :: nat
by simp

lemma gcd-1-int [simp]: gcd m 1 = 1
for m :: int
by (simp add: gcd-int-def)

lemma gcd-idem-nat: gcd x x = x
for x :: nat
by simp

lemma gcd-idem-int: gcd x x = |x|
for x :: int
by (auto simp: gcd-int-def)

declare gcd-nat.simps [simp del]

gcd m n divides m and n. The conjunctions don’t seem provable separately.

instance nat :: semiring-gcd

proof
  fix m n :: nat
  show gcd m n dvd m and gcd m n dvd n
  proof (induct m n rule: gcd-nat-induct)
    case (step m n)
    then have gcd n (m mod n) dvd m
      by (metis dvd-mod-imp-dvd)
    with step show gcd m n dvd m
      by (simp add: gcd-non-0-nat)
  qed (simp-all add: gcd-0-nat gcd-non-0-nat)

next
  fix m n k :: nat
  assume k dvd m and k dvd n
  then show k dvd gcd m n
    by (induct m n rule: gcd-nat-induct) (simp-all add: gcd-non-0-nat dvd-mod gcd-0-nat)
  qed (simp-all add: lcm-nat-def)

instance int :: ring-gcd

proof
  fix k l r :: int
  show [simp]: gcd k l dvd k gcd k l dvd l
using gcd-dvd1 [of nat |k| nat |l|]
gcd-dvd2 [of nat |k| nat |l|]
by simp-all
show lcm k l = normalize (k * l div gcd k l)
using lcm-gcd [of nat |k| nat |l|]
by (simp add: nat-eq-iff of-nat-div abs-mult abs-div)
assume r dvd k r dvd l
then show r dvd gcd k l
using gcd-greatest [of nat |r| nat |k| nat |l|]
by simp
qed simp

lemma gcd-le1-nat [simp]: a ≠ 0 ⇒ gcd a b ≤ a
for a b :: nat
by (rule dvd-imp-le) auto

lemma gcd-le2-nat [simp]: b ≠ 0 ⇒ gcd a b ≤ b
for a b :: nat
by (rule dvd-imp-le) auto

lemma gcd-le1-int [simp]: a > 0 ⇒ gcd a b ≤ a
for a b :: int
by (rule zdvd-imp-le) auto

lemma gcd-le2-int [simp]: b > 0 ⇒ gcd a b ≤ b
for a b :: int
by (rule zdvd-imp-le) auto

lemma gcd-pos-nat [simp]: gcd m n > 0 ←→ m ≠ 0 ∨ n ≠ 0
for m n :: nat
using gcd-eq-0-iff [of m n] by arith

lemma gcd-pos-int [simp]: gcd m n > 0 ←→ m ≠ 0 ∨ n ≠ 0
for m n :: int
using gcd-eq-0-iff [of m n] gcd-ge-0-int [of m n] by arith

lemma gcd-unique-nat: d dvd a ∧ d dvd b ∧ (∀ e. e dvd a ∧ e dvd b → e dvd d) ←→ d = gcd a b
for d a :: nat
using gcd-unique by fastforce

lemma gcd-unique-int:
d ≥ 0 ∧ d dvd a ∧ d dvd b ∧ (∀ e. e dvd a ∧ e dvd b → e dvd d) ←→ d = gcd a b
for d a :: int
using zdvd-antisym-nonneg by auto

interpretation gcd-nat:
semilattice-neutr-order gcd 0::nat Rings.ded λm n. m dvd n ∧ m ≠ n

lemma gcd-proj1-if-dvd-int [simp]: \( x \mid y \implies \gcd(x, y) = |x| \)
  for \( x, y :: \text{int} \)
  by (metis gcd-proj1-if-dvd-int gcd.commute)

Multiplication laws.

lemma gcd-mult-distrib-nat: \( k \cdot \gcd(m, n) = \gcd(k \cdot m, k \cdot n) \)
  for \( k, m, n :: \text{nat} \)
  — [1, page 27]
  by (simp add: gcd-mult-left)

lemma gcd-mult-distrib-int: \( \lvert k \rvert \cdot \gcd(m, n) = \gcd(k \cdot m, k \cdot n) \)
  for \( k, m, n :: \text{int} \)
  by (simp add: gcd-mult-left abs-mult)

Addition laws.

lemma gcd-diff1-nat: \( m \geq n \implies \gcd(m - n, n) = \gcd(m, n) \)
  for \( m, n :: \text{nat} \)
  by (subst gcd-add1 [symmetric]) auto

lemma gcd-diff2-nat: \( n \geq m \implies \gcd(n - m, n) = \gcd(m, n) \)
  for \( m, n :: \text{nat} \)
  by (metis gcd.gcd.commute gcd-diff2-nat gcd-diff1-nat)

lemma gcd-non-0-int:
  fixes \( x, y :: \text{int} \)
  assumes \( y > 0 \)
  shows \( \gcd(x, y) = \gcd(y, (x \mod y)) \)
proof (cases \( x \mod y = 0 \))
  case False
  then have neg: \( x \mod y = y - (-x) \mod y \)
    by (simp add: zmod-zminus1-eq-if)
  have xy: \( 0 \leq x \mod y \)
    by (simp add: assms)
  show \( ?thesis \)
proof (cases \( x < 0 \))
  case True
  have nat: \( -x \mod y \leq \text{lhd} \)
    by (simp add: assms dual-order.order-iff-strict)
  moreover have \( \gcd(-x) = \gcd(-x \mod y) \)
    using True assms gcd-non-0-nat nat-mod-distrib by auto
  ultimately have \( \gcd(nat(-x)) = \gcd(nat(-x \mod y)) \)
    using assms
    by (simp add: neg nat-diff-distrib’ (metis gcd.gcd.commute gcd-diff2-nat)

THEORY “GCD”
theory "GCD"

with assms ⟨0 ≤ x mod y⟩ show thesis
  by (simp add: True dual-order.order-iff-strict gcd-int-def)
next
  case False
  with assms xy have gcd (nat x) (nat y) = gcd (nat y) (nat x mod nat y)
  using gcd-red-nat by blast
  with False assms show thesis
    by (simp add: gcd-int-def nat-mod-distrib)
qed

lemma gcd-red-int: gcd x y = gcd y (x mod y)
  for x y :: int
proof (cases y 0::int rule: linorder-cases)
  case less
  with gcd-non-0-int [of −y −x] show thesis
    by auto
next
  case greater
  with gcd-non-0-int [of y x] show thesis
    by auto
qed auto

lemma finite-divisors-nat [simp]:
  fixes m :: nat
  assumes m > 0
  shows finite {d. d dvd m}
proof
  from assms have {d. d dvd m} ⊆ {d. d ≤ m}
    by (auto dest: dvd-imp-le)
  then show thesis
    using finite-Collect-le-nat by (rule finite-subset)
qed

lemma finite-divisors-int [simp]:
  fixes i :: int
  assumes i ≠ 0
  shows finite {d. d dvd i}
proof
  have {d. |d| ≤ |i|} = {− |i|..|i|}
    by (auto simp: abs-if)
  then have finite {d. |d| ≤ |i|}
    by simp
  from finite-subset [OF - this] show thesis
    using assms by (simp add: dvd-imp-le-int subset-iff)
theory "GCD"

lemma Max-divisors-self-nat [simp]: \( n \neq 0 \implies \text{Max}\{d::\text{nat} \mid d \mid n\} = n \)
by (fastforce intro: antisym Max-le-iff[D2] simp: dvd-imp-le)

lemma Max-divisors-self-int [simp]:
assumes \( n \neq 0 \)
shows \( \text{Max}\{d::\text{int} \mid d \mid n\} = \vert n \vert \)
proof (rule antisym)
  show \( \text{Max}\{d::\text{int} \mid d \mid n\} \leq \vert n \vert \)
  using assms by (auto intro: abs-le-D1 dvd-imp-le-int intro: Max-le-iff[D2])
qed (simp add: assms)

lemma gcd-is-Max-divisors-nat:
fixes \( m, n \) :: \text{nat}
assumes \( n > 0 \)
shows \( \text{gcd}\ m n = \text{Max}\{d::\text{nat} \mid d \mid m \land d \mid n\} \)
proof (rule Max-eqI[D2] simp)
  show finite \{d::\text{nat} \mid d \mid m \land d \mid n\} by (simp add: \langle n > 0 \rangle)
  show \( \forall y. y \mid m \land y \mid n \implies y \leq \text{gcd}\ m n \)
  by (simp add: \langle n > 0 \rangle dvd-imp-le)
qed

lemma gcd-is-Max-divisors-int:
fixes \( m, n \) :: \text{int}
assumes \( n \neq 0 \)
shows \( \text{gcd}\ m n = \text{Max}\{d::\text{int} \mid d \mid m \land d \mid n\} \)
proof (rule Max-eqI[D2] simp)
  show finite \{d::\text{int} \mid d \mid m \land d \mid n\} by (simp add: \langle n > 0 \rangle)
  show \( \forall y. y \mid m \land y \mid n \implies y \leq \text{gcd}\ m n \)
  by (simp add: \langle n > 0 \rangle zdvd-imp-le)
qed

lemma gcd-code-int [code]: \( \text{gcd}\ k l = \text{if}\ l = 0\ then\ \text{gcd}\ l\ (\vert k \mid \mod\ \vert l\vert)\ \text{else}\ \text{gcd}\ l\ (\vert k \mid \mod\ \vert l\vert)\)
for \( k, l :: \text{int}\)
using gcd-red-int [of \vert k \mid \vert l\vert] by simp

lemma coprime-Suc-left-nat [simp]:
coprime (Suc n) n
using coprime-add-one-left [of n] by simp

lemma coprime-Suc-right-nat [simp]:
coprime n (Suc n)
using coprime-Suc-left-nat [of n] by (simp add: ac-simps)

lemma coprime-diff-one-left-nat [simp]:
coprime (n - 1) n if \( n > 0 \)
for \( n :: \text{nat}\)
using that coprime-Suc-right-nat [of n - 1] by simp

qed
lemma coprime-diff-one-right-nat [simp]:
\[ \text{coprime } n \ (n - 1) \text{ if } n > 0 \text{ for } n :: \nat \]
using that coprime-diff-one-left-nat [of \( n \)] by (simp add: ac-simps)

lemma coprime-crossproduct-nat:
fixes \( a \ b \ c \ d :: \nat \)
assumes coprime \( a \ d \) and coprime \( b \ c \)
shows \( a \ast c = b \ast d \leftrightarrow a = b \land c = d \)
using assms coprime-crossproduct [of \( a \ d \ b \ c \)] by simp

lemma coprime-crossproduct-int:
fixes \( a \ b \ c \ d :: \int \)
assumes coprime \( a \ d \) and coprime \( b \ c \)
shows \(|a| \ast |c| = |b| \ast |d| \leftrightarrow |a| = |b| \land |c| = |d| \)
using assms coprime-crossproduct [of \( a \ d \ b \ c \)] by simp

85.7 Bezout’s theorem

Function \( \text{bezw} \) returns a pair of witnesses to Bezout’s theorem – see the theorems that follow the definition.

fun \( \text{bezw} :: \nat \Rightarrow \nat \Rightarrow \int \ast \int \)
where \( \text{bezw} \ x \ y = \)
\( (\text{if } y = 0 \text{ then } (1, 0) \text{ else } (\text{snd } (\text{bezw} \ y \ (x \mod y)), \\
\quad \text{fst } (\text{bezw} \ y \ (x \mod y)) - \text{snd } (\text{bezw} \ y \ (x \mod y))^\ast \text{int}(x \div y))) \)

lemma bezw-0 [simp]: \( \text{bezw} \ x \ 0 = (1, 0) \)
by simp

lemma bezw-non-0:
\( y > 0 \implies \text{bezw} \ x \ y = \)
\( (\text{snd } (\text{bezw} \ y \ (x \mod y)), \text{fst } (\text{bezw} \ y \ (x \mod y)) - \text{snd } (\text{bezw} \ y \ (x \mod y))^\ast \text{int}(x \div y)) \)
by simp

declare bezw.simps [simp del]

lemma bezw-aux: int (gcd \( x \ y \)) = \( \text{fst } (\text{bezw} \ x \ y) \ast \text{int } x + \text{snd } (\text{bezw} \ x \ y) \ast \text{int } y \)
proof (induct \( x \ y \) rule: gcd-nat-induct)

  case (step \( m \ n \))
  then have \( \text{fst } (\text{bezw} \ m \ n) \ast \text{int } m + \text{snd } (\text{bezw} \ m \ n) \ast \text{int } n - \text{int } (\text{gcd } m \ n) \)
  \( = \text{int } m \ast \text{snd } (\text{bezw} \ n \ (m \mod n)) - \text{(int } (m \mod n) \ast \text{snd } (\text{bezw} \ n \ (m \mod n)) + \text{int } n \ast (\text{int } (m \div n) \ast \text{snd } (\text{bezw} \ n \ (m \mod n)))) \)
  by (simp add: bezw-non-0 gcd-non-0-nat field-simps)
  also have \( \ldots = \text{int } m \ast \text{snd } (\text{bezw} \ n \ (m \mod n)) - (\text{int } (m \mod n) + \text{int } (n \ast (m \div n))) \ast \text{snd } (\text{bezw} \ n \ (m \mod n)) \)
by (simp add: distrib-right)
also have ... = 0
by (metis cancel-comm-monoid-add-class.diff-cancel mod-mult-div-eq of-nat-add)
finally show ?case
by simp
qed auto

lemma bezout-int: \( \exists u \, v \cdot u \ast x + v \ast y = \gcd x y \) for \( x \, y :: \text{int} \)
proof
have aux: \( x \geq 0 \Longrightarrow y \geq 0 \Longrightarrow \exists u \, v \cdot u \ast x + v \ast y = \gcd x y \) for \( x \, y :: \text{int} \)
apply (rule-tac x = fst (bezw (nat x) (nat y)) in exI)
apply (rule-tac x = snd (bezw (nat x) (nat y)) in exI)
by (simp add: bezw-aux gcd-int-def)
consider \( x \geq 0 \) \( y \geq 0 \) \( | x \geq 0 \) \( y \leq 0 \) \( | x \leq 0 \) \( y \geq 0 \) \( | x \leq 0 \) \( y \leq 0 \)
using linear by blast
then show ?thesis
proof cases
  case 1
  then show ?thesis by (rule aux)
next
  case 2
  then show ?thesis
  using aux [of x \(-\) y]
  by (metis gcd-neg2-int mult.commute mult-minus-right neg-0-le-iff-le)
next
  case 3
  then show ?thesis
  using aux [of \(-\) x y]
  by (metis gcd.commute gcd-neg2-int mult.commute mult-minus-right neg-0-le-iff-le)
next
  case 4
  then show ?thesis
  using aux [of \(-\) x \(-\) y]
  by (metis diff-0 diff-ge-0-iff-ge gcd-neg1-int gcd-neg2-int mult.commute mult-minus-right)
qed

Versions of Bezout for \text{nat}, by Amine Chaieb.

lemma Euclid-induct [case-names swap zero add]:
fixes \( P :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool} \)
assumes c: \( \forall a \, b. \ P \ a \ b \longleftrightarrow \ P \ b \ a \)
and z: \( \forall a. \ P \ a \ 0 \)
and add: \( \forall a \, b. \ P \ a \ b \longrightarrow \ P \ a \ (a + b) \)
shows \( P \ a \ b \)
proof (induct a + b arbitrary: a b rule: less-induct)
case less
consider (eq) \( a = b \) \( (lt) \) \( a < b \) \( a + b - a < a + b \) \( | b = 0 \) \( b + a - b < a \)
THEORY "GCD"

+ b
  by arith
show ?case
proof (cases a b rule: linorder-cases)
case equal
  with add [rule-format, OF z [rule-format, of a]] show ?thesis by simp
next
case lt: less
then consider a = 0 | a + b - a < a + b by arith
then show ?thesis
proof cases
  case 1
  with z c show ?thesis by blast
next
case 2
also have *: a + b - a = a + (b - a) using lt by arith
finally have a + (b - a) < a + b .
then have P a (a + (b - a)) by (rule add [rule-format, OF less])
then show ?thesis by (simp add: *[symmetric])
qed
next
case gt: greater
then consider b = 0 | b + a - b < a + b by arith
then show ?thesis
proof cases
  case 1
  with z c show ?thesis by blast
next
case 2
also have *: b + a - b = b + (a - b) using gt by arith
finally have b + (a - b) < a + b .
then have P b (b + (a - b)) by (rule add [rule-format, OF less])
then have P b a by (simp add: *[symmetric])
  with c show ?thesis by blast
qed
qed

lemma bezout-lemma-nat:
fixes d::nat
shows [d dvd a; d dvd b; a * x = b * y + d ∨ b * x = a * y + d]
  ⇒ ∃ x y. d dvd a ∧ d dvd a + b ∧ (a * x = (a + b) * y + d ∨ (a + b) * x
  = a * y + d)
apply auto
apply (metis add-mult-distrib2 left-add-mult-distrib)
apply (rule_tac x=x in ex1)
by (metis add-mult-distrib2 mult.commute add.assoc)

lemma bezout-add-nat:
∃(d :: nat) x y. d dvd a ∧ d dvd b ∧ (a * x = b * y + d ∨ b * x = a * y + d)

proof (induct a b rule: Euclid-induct)
  case (swap a b)
  then show ?case
    by blast
next
  case (zero a)
  then show ?case
    by fastforce
next
  case (add a b)
  then show ?case
    by (meson bezout-lemma-nat)
qed

lemma bezout1-nat: ∃(d :: nat) x y. d dvd a ∧ d dvd b ∧ (a * x − b * y = d ∨ b * x − a * y = d)
  using bezout-add-nat[of a b]
  by (metis add-diff-cancel-left)

lemma bezout-add-strong-nat:
  fixes a b :: nat
  assumes a: a ≠ 0
  shows ∃d x y. d dvd a ∧ d dvd b ∧ a * x = b * y + d
  proof
    consider d x y where d dvd a d dvd b a * x = b * y + d
      | d x y where d dvd a d dvd b b * x = a * y + d
    using bezout-add-nat[of a b]
    by blast
  then show ?thesis
    proof cases
      case 1
      then show ?thesis
        by blast
    next
      case H: 2
      show ?thesis
      proof (cases b = 0)
        case True
        with H show ?thesis
        by simp
      next
        case False
        then have bp: b > 0 by simp
        with dvd-imp-le[OF H(2)] consider d = b | d < b
        by atomize-elim auto
      then show ?thesis
      proof cases
        case 1
        with a H show ?thesis
        by (metis Suc-pred add.commute mult.commute mult-Suc-right neq0-conv)
      next
        case 2
show \( ?\text{thesis} \)
proof (cases \( x = 0 \))
  case True
  with \( a \) \( ?\text{thesis} \) by simp
next
case 0: False
then have \( xp: x > 0 \) by simp
from \( d < b \) have \( d \leq b - 1 \) by simp
then have \( d \cdot b \leq b \cdot (b - 1) \) by simp
with \( xp \text{ multi-mono[of } 1 \cdot d \cdot b \cdot (b - 1) \text{]} \)
  have \( db: d \cdot b \leq x \cdot b \cdot (b - 1) \) using \( bp \) by simp
from \( H(3) \) have \( d + (b - 1) \cdot (b \cdot x) = d + (b - 1) \cdot (a \cdot y + d) \)
  by simp
then have \( \text{dble} = d \cdot b \leq x \cdot b \cdot (b - 1) \) by simp
then have \( \text{dble} = d + (b - 1) \cdot a \cdot y + (b - 1) \cdot d = d + (b - 1) \cdot b \cdot x \)
  by \( \text{simp only: \text{multi-assoc distrib-left}} \)
then have \( \text{dble} = a \cdot ((b - 1) \cdot y) + d \cdot (b - 1 + 1) = d + x \cdot b \cdot (b - 1) \)
  by \( \text{algebra} \)
then have \( \text{dble} = a \cdot ((b - 1) \cdot y) = d + x \cdot b \cdot (b - 1) - d \cdot b \)
  using \( bp \) by simp
then have \( \text{dble} = a \cdot ((b - 1) \cdot y) = d + (x \cdot b \cdot (b - 1) - d \cdot b) \)
  by \( \text{simp only: \text{diff-add-assoc[of db, of d, symmetric]}} \)
then have \( \text{dble} = a \cdot ((b - 1) \cdot y) = b \cdot (x \cdot (b - 1) - d) + d \)
  by \( \text{simp only: \text{diff-mult-distrib2 ac-simps}} \)
with \( H(1,2) \) show \( ?\text{thesis} \)
  by blast
qed
qed
qed

lemma bezout-nat:
  fixes \( a :: \text{nat} \)
  assumes \( a: a \neq 0 \)
  shows \( \exists x \cdot y. \ a \cdot x = b \cdot y + \gcd a \cdot b \)
proof
  obtain \( d \cdot x \cdot y \) where \( d \cdot dvd a \cdot d \cdot d \cdot d \cdot b \) and \( eq: a \cdot x = b \cdot y + d \)
    using \( \text{bezout-add-strong-nat[of } a, of b \text{]} \) by blast
from \( d \cdot dvd \cdot \gcd a \cdot b \)
  by \( \text{simp} \)
then obtain \( k \) where \( \gcd a \cdot b = d \cdot k \)
  unfolding \( \text{dvd-def} \) by blast
from \( eq \) have \( a \cdot x \cdot k = (b \cdot y + d) \cdot k \)
  by \( \text{auto} \)
then have \( a \cdot (x \cdot k) = b \cdot (y \cdot k) + \gcd a \cdot b \)
  by \( \text{algebra add: } k \)
then show \( ?\text{thesis} \)
  by blast
qed
85.8 LCM properties on nat and int

lemma lcm-altdef-int [code]: \( \text{lcm} \ a \ b = |a| * |b| \ div \ \text{gcd} \ a \ b \)
  for \( \ a \ b :: \ \text{int} \)
  by (simp add: abs-mult lcm-gcd abs-div)

lemma prod-gcd-lcm-nat: \( m * n = \text{gcd} \ m \ n * \text{lcm} \ m \ n \)
  for \( \ m \ n :: \ \text{nat} \)
  by (simp add: lcm-gcd)

lemma prod-gcd-lcm-int: \( |m| * |n| = \text{gcd} \ m \ n * \text{lcm} \ m \ n \)
  for \( \ m \ n :: \ \text{int} \)
  by (simp add: lcm-gcd abs-div abs-mult)

lemma lcm-pos-nat: \( m > 0 \Rightarrow n > 0 \Rightarrow \text{lcm} \ m \ n > 0 \)
  for \( \ m \ n :: \ \text{nat} \)
  using lcm-eq-0-iff \[of \ m \ n\] by auto

lemma lcm-pos-int: \( m \neq 0 \Rightarrow n \neq 0 \Rightarrow \text{lcm} \ m \ n > 0 \)
  for \( \ m \ n :: \ \text{int} \)
  by (simp add: lcm-eq-0-iff)

lemma dvd-pos-nat: \( n > 0 \Rightarrow m \ \text{dvd} \ n \Rightarrow m > 0 \)
  for \( \ m \ n :: \ \text{nat} \)
  by auto

lemma lcm-unique-nat:
  \([\ a \ \text{dvd} \ d \ \land \ b \ \text{dvd} \ d \ \land \ (\forall \ e. \ a \ \text{dvd} \ e \ \land \ b \ \text{dvd} \ e \ \Rightarrow \ d \ \text{dvd} \ e) \ \iff \ d = \text{lcm} \ a \ b \])
  for \( \ a \ b \ d :: \ \text{nat} \)
  by (auto intro: dvd-antisym lcm-least)

lemma lcm-unique-int:
  \([\ d \geq 0 \ \land \ a \ \text{dvd} \ d \ \land \ b \ \text{dvd} \ d \ \land \ (\forall \ e. \ a \ \text{dvd} \ e \ \land \ b \ \text{dvd} \ e \ \Rightarrow \ d \ \text{dvd} \ e) \ \iff \ d = \text{lcm} \ a \ b \])
  for \( \ a \ b \ d :: \ \text{int} \)
  using lcm-least zdvd-antisym-nonneg by auto

lemma lcm-proj2-if-dvd-nat [simp]: \( x \ \text{dvd} \ y \Rightarrow \text{lcm} \ x \ y = y \)
  for \( \ x \ y :: \ \text{nat} \)
  by (simp add: lcm-proj2-if-dvd)

lemma lcm-proj2-if-dvd-int [simp]: \( x \ \text{dvd} \ y \Rightarrow \text{lcm} \ x \ y = |y| \)
  for \( \ x \ y :: \ \text{int} \)
  by (simp add: lcm-proj2-if-dvd)

lemma lcm-proj1-if-dvd-nat [simp]: \( x \ \text{dvd} \ y \Rightarrow \text{lcm} \ y \ x = y \)
  for \( \ x \ y :: \ \text{nat} \)
  by (subst lcm.commute) (erule lcm-proj2-if-dvd-nat)

lemma lcm-proj1-if-dvd-int [simp]: \( x \ \text{dvd} \ y \Rightarrow \text{lcm} \ y \ x = |y| \)
for \( x \) \( y \) :: int
by (subst lcm.commute) (erule lcm-proj2-if-dvd-int)

lemma lcm-proj1-iff-nat [simp]: \( \text{lcm} \ m \ n = m \iff n \text{ dvd m} \)
for \( m \ n \) :: nat
by (metis lcm-proj1-if-dvd-nat lcm-unique-nat)

lemma lcm-proj2-iff-nat [simp]: \( \text{lcm} \ m \ n = n \iff m \text{ dvd n} \)
for \( m \ n \) :: nat
by (metis lcm-proj2-if-dvd-nat lcm-unique-nat)

lemma lcm-proj1-iff-int [simp]: \( \text{lcm} \ m \ n = |m| \iff n \text{ dvd m} \)
for \( m \ n \) :: int
by (metis dvd-abs-iff lcm-proj1-if-dvd-int lcm-unique-int)

lemma lcm-proj2-iff-int [simp]: \( \text{lcm} \ m \ n = |n| \iff m \text{ dvd n} \)
for \( m \ n \) :: int
by (metis dvd-abs-iff lcm-proj2-if-dvd-int lcm-unique-int)

lemma lcm-1-iff-nat [simp]: \( \text{lcm} \ m \ n = \text{Suc} \ 0 \iff \text{m = Suc} \ 0 \land \text{n = Suc} \ 0 \)
for \( m \ n \) :: nat
using lcm-eq-1-iff [of m n] by simp

lemma lcm-1-iff-int [simp]: \( \text{lcm} \ m \ n = 1 \iff (m = 1 \lor m = -1) \land (n = 1 \lor n = -1) \)
for \( m \ n \) :: int
by auto

85.9 The complete divisibility lattice on nat and int
Lifting \( \gcd \) and \( \text{lcm} \) to sets \( (\text{Gcd} / \text{Lcm}) \). \( \text{Gcd} \) is defined via \( \text{Lcm} \) to facilitate the proof that we have a complete lattice.

instantiation nat :: semiring-Gcd
begin

interpretation semilattice-neutr-set lcm 1::nat
by standard simp-all

definition \( \text{Lcm} \ M = (\text{if finite} \ M \ \text{then} \ \text{F} \ M \ \text{else} \ 0) \) for \( M :: \text{nat set} \)

lemma Lcm-nat-empty: \( \text{Lcm} \ \{\} = (1::nat) \)
by (simp add: Lcm-nat-def del: One-nat-def)

lemma Lcm-nat-insert: \( \text{Lcm} \ (\text{insert} \ n \ M) = \text{lcm} \ n \ (\text{Lcm} \ M) \) for \( n :: \text{nat} \)
by (cases finite \( M \)) (auto simp: Lcm-nat-def simp del: One-nat-def)

lemma Lcm-nat-infinite: infinite \( M \implies \text{Lcm} \ M = 0 \) for \( M :: \text{nat set} \)
by (simp add: Lcm-nat-def)
lemma dvd-Lcm-nat [simp]:
fixes M :: nat set
assumes m ∈ M
shows m dvd Lcm M
proof –
  from assms have insert m M = M
    by auto
moreover have m dvd Lcm (insert m M)
    by (simp add: Lcm-nat-insert)
ultimately show ?thesis
  by simp
qed

lemma Lcm-dvd-nat [simp]:
fixes M :: nat set
assumes ∀m ∈ M. m dvd n
shows Lcm M dvd n
proof (cases n > 0)
  case False
  then show ?thesis
    by simp
next
  case True
  then have finite {d. d dvd n}
    by (rule finite-divisors-nat)
moreover have M ⊆ {d. d dvd n}
  using assms by fast
ultimately have finite M
  by (rule rev-finite-subset)
then show ?thesis
  using assms by (induct M) (simp-all add: Lcm-nat-empty Lcm-nat-insert)
qed

definition Gcd M = Lcm {d. ∀m ∈ M. d dvd m} for M :: nat set

instance
proof
  fix N :: nat set
  fix n :: nat
  show Gcd N dvd n if n ∈ N
    using that by (induct N rule: infinite-finite-induct) (auto simp: Gcd-nat-def)
  show n dvd Gcd N if ∀m. m ∈ N ⇒ n dvd m
    using that by (induct N rule: infinite-finite-induct) (auto simp: Gcd-nat-def)
  show n dvd Lcm N if n ∈ N
    using that by (induct N rule: infinite-finite-induct) auto
  show Lcm N dvd n if ∀m. m ∈ N ⇒ m dvd n
    using that by (induct N rule: infinite-finite-induct) auto
  show normalize (Gcd N) = Gcd N and normalize (Lcm N) = Lcm N
    by simp-all
qed
end

lemma Gcd-nat-eq-one: 1 ∈ N ⇒ Gcd N = 1
  for N :: nat set
  by (rule Gcd-eq-1-I) auto

instance nat :: semiring-gcd-mult-normalize
  by intro_classes (auto simp: unit-factor-nat-def)

Alternative characterizations of Gcd:

lemma Gcd-eq-Max:
  fixes M :: nat set
  assumes finite (M::nat set) and M ≠ {} and 0 ∉ M
  shows Gcd M = Max (⋂ m∈M. {d. d dvd m})
proof (rule antisym)
  from assms obtain m where m ∈ M and m > 0
    by auto
  from ⟨m > 0⟩ have finite {d. d dvd m}
    by (blast intro: finite-divisors-nat)
  with ⟨m ∈ M⟩ have fin: finite (⋂ m∈M. {d. d dvd m})
    by blast
  from fin show Gcd M ≤ Max (⋂ m∈M. {d. d dvd m})
    by (auto intro: Max-ge Gcd-dvd)
  from fin show Max (⋂ m∈M. {d. d dvd m}) ≤ Gcd M
  proof (rule Max.boundedI, simp-all)
    show (⋂ m∈M. {d. d dvd m}) ≠ {} by auto
    show ∀ a. ∀ x∈M. a dvd x ⇒ a ≤ Gcd M
      by (meson Gcd-dvd Gcd-greatest ⟨0 < m⟩ ⟨m ∈ M⟩ dvd-imp-le dvd-pos-nat)
  qed
qed

lemma Gcd-remove0-nat: finite M ⇒ Gcd M = Gcd (M - {0})
  for M :: nat set
proof (induct pred: finite)
  case (insert x M)
  then show ?case
    by (simp add: insert-Diff-if)
qed auto

lemma Lcm-in-lcm-closed-set-nat:
  fixes M :: nat set
  assumes finite M M ≠ {} and 0 ∉ M
  shows Lcm M ∈ M
  using assms
proof (induction M rule: finite-linorder-min-induct)
  case (insert x M)
  then have ∃ m n. m ∈ M ⇒ n ∈ M ⇒ lcm m n ∈ M
by (metis dvd-lcm1 gr0I insert-iff lcm-pos-nat nat-dvd-not-less)
with insert show ?case
by simp (metis Lcm-nat-empty One-nat-def dvd-1-left dvd-lcm2)
qed auto

lemma Lcm-eq-Max-nat:
fixes M :: nat set
assumes M : finite M M \not= {} 0 \notin M \land \forall m. \forall n. \exists m \in M. n \in M \implies lcm m n
shows Lcm M = Max M
proof (rule antisym)
show Lcm M \leq Max M
by (simp add: Lcm-in-lcm-closed-set-nat finite M M \not= {})
show Max M \leq Lcm M
by (meson Lcm-0-iff Max-in M dvd-Lcm dvd-imp-le le-0-eq not-le)
qed

lemma mult-inj-if-coprime-nat:
inj-on f A \implies inj-on g B \implies (\land \forall a b. \exists a \in A. \exists b \in B \implies \coprime (f a) (g b)) \implies
inj-on (\lambda (a, b). f a * g b) (A \times B)
for f :: 'a \Rightarrow nat and g :: 'b \Rightarrow nat
by (auto simp: inj-on-def coprime-crossproduct-nat simp del: One-nat-def)

85.9.1 Setwise GCD and LCM for integers

instantiation int :: Gcd
begin

definition Gcd-int :: int set \Rightarrow int
where Gcd K = int (GCD k \in K. (nat \circ abs) k)
definition Lcm-int :: int set \Rightarrow int
where Lcm K = int (LCM k \in K. (nat \circ abs) k)

instance ..
end

lemma Gcd-int-eq[simp]:
(GCD n \in N. int n) = int (Gcd N)
by (simp add: Gcd-int-def image-image)

lemma Gcd-nat-abs-eq[simp]:
(GCD k \in K. nat \abs k) = nat (Gcd K)
by (simp add: Gcd-int-def)

lemma abs-Gcd-eq[simp]:
|Gcd K| = Gcd K for K :: int set
by (simp only: Gcd-int-def)
lemma Gcd-int-greater-eq-0 [simp]:
  \( \text{Gcd } K \geq 0 \)
for \( K :: \text{int set} \)
using abs-ge-zero [of Gcd K] by simp

lemma Gcd-abs-eq [simp]:
  \((\text{GCD } k \in K. |k|) = \text{Gcd } K\)
for \( K :: \text{int set} \)
by (simp only: Gcd-int-def image-image) simp

lemma Lcm-int-eq [simp]:
  \((\text{LCM } n \in N. \text{int } n) = \text{int } (\text{Lcm } N)\)
by (simp add: Lcm-int-def image-image)

lemma Lcm-nat-abs-eq [simp]:
  \(|\text{LCM } k \in K. \text{nat } |k|) = \text{nat } (\text{Lcm } K)\)
by (simp add: Lcm-int-def)

lemma abs-Lcm-eq [simp]:
  \(|\text{Lcm } K| = \text{Lcm } K\) for \( K :: \text{int set} \)
by (simp only: Lcm-int-def)

lemma Lcm-int-greater-eq-0 [simp]:
  \(\text{Lcm } K \geq 0\)
for \( K :: \text{int set} \)
using abs-ge-zero [of Lcm K] by simp

lemma Lcm-abs-eq [simp]:
  \((\text{LCM } k \in K. |k|) = \text{Lcm } K\) for \( K :: \text{int set} \)
by (simp only: Lcm-int-def)

instance int :: semiring-Gcd proof
  fix \( K :: \text{int set} \) and \( k :: \text{int} \)
  show \( \text{Gcd } K \text{ dvd } k \text{ and } k \text{ dvd } \text{Lcm } K \) if \( k \in K \)
    using That Gcd-dvd [of nat \( |k|\) (nat o abs) ' K]
    dvd-Lcm [of nat \( |k|\) (nat o abs) ' K]
    by (simp-all add: comp-def)
  show \( k \text{ dvd } \text{Gcd } K \) if \( \forall l. l \in K \implies k \text{ dvd } l \)
    proof
      have \( \text{nat } |k| \text{ dvd } (\text{GCD } k \in K. \text{nat } |k|) \)
        by (rule Gcd-greatest) (use that in auto)
      then show \?thesis by simp
    qed
  show \( \text{Lcm } K \text{ dvd } k \) if \( \forall l. l \in K \implies l \text{ dvd } k \)
    proof
      have \( (\text{LCM } k \in K. \text{nat } |k|) \text{ dvd } \text{nat } |k| \)
  qed
by (rule Lcm-least) (use that in auto)
then show \( \text{thesis} \) by simp
qed
qed (simp-all add: sgn-mult)

instance \( \text{int} :: \text{semiring-gcd-mult-normalize} \)
by intra-classes (auto simp: sgn-mult)

85.10 GCD and LCM on \( \text{integer} \)

instantiation \( \text{integer} :: \text{gcd} \)
begin

context
  includes \( \text{integer} \).lifting
begin

lift-definition \( \text{gcd-integer} :: \text{integer} \Rightarrow \text{integer} \Rightarrow \text{integer} \) is \( \text{gcd} \).
lift-definition \( \text{lcm-integer} :: \text{integer} \Rightarrow \text{integer} \Rightarrow \text{integer} \) is \( \text{lcm} \).
end

instance ..
end

lifting-update \( \text{integer}.lifting \)
lifting-forget \( \text{integer}.lifting \)

context
  includes \( \text{integer} \).lifting
begin

lemma \( \text{gcd-code-integer} \) [code]: \( \text{gcd} \ k \ l = | \text{if } l = (0::\text{integer}) \text{ then } k \text{ else } \text{gcd} \ l \ (|k| \mod |l|)| \)
  by transfer (fact gcd-code-int)

lemma \( \text{lcm-code-integer} \) [code]: \( \text{lcm} \ a \ b = |a| \ast |b| \div \text{gcd} \ a \ b \)
  for \( a \ b :: \text{integer} \)
  by transfer (fact lcm-altdef-int)
end

code-printing

constant \( \text{gcd} :: \text{integer} \Rightarrow - \Rightarrow \)
  (OCaml) \(!\text{fan } k \ l -> \text{if } \text{Z.equal } k \text{ Z.zero } \text{then/ } \text{Z.abs } l \text{ else } \text{if } \text{Z.equal/ } l \text{ Z.zero \then } \text{Z.abs } k \text{ else } \text{Z.gcd } k \ l\)
  and (Haskell) Prelude.gcd
and (Scala) - `gcd'((-))`

— There is no gcd operation in the SML standard library, so no code setup for SML.

Some code equations

lemmas Lcm-nat-set-eq-fold [code] = Lcm-set-eq-fold [where ?'a = nat]
lemmas Lcm-int-set-eq-fold [code] = Lcm-set-eq-fold [where ?'a = int]

Fact aliases.

lemma lcm-0-iff-nat [simp]: lcm m n = 0 ←→ m = 0 ∨ n = 0
for m n :: nat
by (fact lcm-eq-0-iff)

lemma lcm-0-iff-int [simp]: lcm m n = 0 ←→ m = 0 ∨ n = 0
for m n :: int
by (fact lcm-eq-0-iff)

lemma dvd-lcm-I1-nat [simp]: k dvd m =⇒ k dvd lcm m n
for k m n :: nat
by (fact dvd-lcmI1)

lemma dvd-lcm-I2-nat [simp]: k dvd n =⇒ k dvd lcm m n
for k m n :: nat
by (fact dvd-lcmI2)

lemma dvd-lcm-I1-int [simp]: i dvd m =⇒ i dvd lcm m n
for i m n :: int
by (fact dvd-lcmI1)

lemma dvd-lcm-I2-int [simp]: i dvd n =⇒ i dvd lcm m n
for i m n :: int
by (fact dvd-lcmI2)

lemmas Gcd-dvd-int [simp] = Gcd-dvd [where ?'a = int]
lemmas Gcd-greatest-nat [simp] = Gcd-greatest [where ?'a = nat]
lemmas Gcd-greatest-int [simp] = Gcd-greatest [where ?'a = int]

lemma dvd-Lcm-int [simp]: m ∈ M =⇒ m dvd Lcm M
for M :: int set
by (fact dvd-Lcm)

lemma gcd-neg-natural-1-int [simp]: gcd (− numeral n :: int) x = gcd (numeral n) x
by (fact gcd-neg1-int)

lemma gcd-neg-natural-2-int [simp]: gcd x (− numeral n :: int) = gcd x (numeral n) x
n)
  by (fact gcd-neg2-int)

lemma gcd-proj1-if-dvd-nat [simp]: \( x \mid y \Rightarrow \gcd(x, y) = x \)
  for \( x, y :: \text{nat} \)
  by (fact gcd-nat.absorb1)

lemma gcd-proj2-if-dvd-nat [simp]: \( y \mid x \Rightarrow \gcd(x, y) = y \)
  for \( x, y :: \text{nat} \)
  by (fact gcd-nat.absorb2)

lemmas Lcm-eq-0-I-nat [simp] = Lcm-eq-0-I [where ?'a = nat]
lemmas Lcm-0-iff-nat [simp] = Lcm-0-iff [where ?'a = nat]
lemmas Lcm-least-int [simp] = Lcm-least [where ?'a = int]

end

86 Nitpick: Yet Another Counterexample Generator for Isabelle/HOL

theory Nitpick
  imports Record GCD
  keywords
    nitpick :: diag and
    nitpick-params :: thy-decl
  begin

  datatype (plugins only: extraction) (dead 'a, dead 'b) fun-box = FunBox 'a ⇒ 'b
  datatype (plugins only: extraction) (dead 'a, dead 'b) pair-box = PairBox 'a 'b
  datatype (plugins only: extraction) (dead 'a) word = Word 'a set

  typedef bisim-iterator
  typedef unsigned-bit
  typedef signed-bit

  consts
    unknown :: 'a
    is-unknown :: 'a ⇒ bool
    bisim :: bisim-iterator ⇒ 'a ⇒ 'a ⇒ bool
    bisim-iterator-max :: bisim-iterator
    Quot :: 'a ⇒ 'b
    safe-The :: ('a ⇒ bool) ⇒ 'a

  Alternative definitions.

  lemma Ex1-unfold[nitpick-unfold]: \( \exists x. \{x. P x\} = \{x\} \)
  apply (rule eq-reflection)
  apply (simp add: Ex1-def set-eq-iff)
  apply (rule iffI)
apply (erule exE)
apply (erule conjE)
apply (rule-tac x = x in exI)
apply (rule allI)
apply (rename-tac y)
apply (erule-tac x = y in allE)
by auto

lemma rtrancl-unfold[nitpick-unfold]: r* ≡ (r+)−
by (simp only: rtrancl-trancl-refl)

lemma rtranclp-unfold[nitpick-unfold]: rtranclp r a b ≡ (a = b ∨ tranclp r a b)
by (rule eq-reflection) (auto dest: rtranclpD)

lemma tranclp-unfold[nitpick-unfold]:
tranclp r a b ≡ (a, b) ∈ trancl { (x, y). r x y }
by (simp add: trancl-def)

lemma [nitpick-simp]:
of-nat n = (if n = 0 then 0 else 1 + of-nat (n − 1))
by (cases n) auto

definition prod :: 'a set ⇒ 'b set ⇒ ('a × 'b) set where
prod A B = { (a, b). a ∈ A ∧ b ∈ B }

definition refl' :: ('a × 'a) set ⇒ bool where
refl' r ≡ ∀ x. (x, x) ∈ r

definition wf' :: ('a × 'a) set ⇒ bool where
wf' r ≡ acyclic r ∧ (finite r ∨ unknown)

definition card' :: 'a set ⇒ nat where
card' A ≡ if finite A then length (SOME xs. set xs = A ∧ distinct xs) else 0

definition sum' :: ('a ⇒ 'b::comm-monoid-add) ⇒ 'a set ⇒ 'b where
sum' f A ≡ if finite A then sum-list (map f (SOME xs. set xs = A ∧ distinct xs)) else 0

inductive fold-graph' :: ('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b ⇒ bool where
fold-graph' f z { } z | [x ∈ A; fold-graph' f z (A − {x}) y] ⇒ fold-graph' f z A (f x y)

The following lemmas are not strictly necessary but they help the specialize optimization.

lemma The-psimp[nitpick-psimp]: P = (=) x ⇒ The P = x
by auto

lemma Eps-psimp[nitpick-psimp]:
[P x; ¬ P y; Eps P = y] ⇒ Eps P = x
apply (cases P (Eps P))
apply auto
apply (erule contrapos-np)
by (rule someI)

lemma case-unit-unfold[nitpick-unfold]:
case-unit x u ≡ x
apply (subgoal-tac u = ())
apply (simp only: unit.case)
by simp

declare unit.case[nitpick-simp del]

lemma case-nat-unfold[nitpick-unfold]:
case-nat x f n ≡ if n = 0 then x else f (n - 1)
apply (rule eq-reflection)
by (cases n) auto

declare nat.case[nitpick-simp del]

lemma size-list-simp[nitpick-simp]:
size-list f xs = (if xs = [] then 0 else Suc (f (hd xs) + size-list f (tl xs)))
size xs = (if xs = [] then 0 else Suc (size (tl xs)))
by (cases xs) auto

Auxiliary definitions used to provide an alternative representation for \texttt{rat} and \texttt{real}.
fun \texttt{nat-gcd} :: nat ⇒ nat ⇒ nat where
\texttt{nat-gcd} x y = (if y = 0 then x else \texttt{nat-gcd} y (x mod y))

declare \texttt{nat-gcd.simps} [simp del]

definition \texttt{nat-lcm} :: nat ⇒ nat ⇒ nat where
\texttt{nat-lcm} x y = x * y div (\texttt{nat-gcd} x y)

lemma gcd-eq-nitpick-gcd [nitpick-unfold]:
gcd x y = Nitpick.nat-gcd x y
by (induct x y rule: nat-gcd.induct)
(simp add: gcd-nat.simps Nitpick.nat-gcd.simps)

lemma lcm-eq-nitpick-lcm [nitpick-unfold]:
lcm x y = Nitpick.nat-lcm x y
by (simp only: lcm-nat-def Nitpick.nat-lcm-def gcd-eq-nitpick-gcd)

definition \texttt{Frac} :: int × int ⇒ bool where
\texttt{Frac} ≡ \lambda(a, b). b > 0 ∧ coprime a b

consts
Abs-\texttt{Frac} :: int × int ⇒ 'a
Rep-Frac :: 'a ⇒ int × int

definition zero-frac :: 'a where
  zero-frac ≡ Abs-Frac (0, 1)

definition one-frac :: 'a where
  one-frac ≡ Abs-Frac (1, 1)

definition num :: 'a ⇒ int where
  num ≡ fst ◦ Rep-Frac

definition denom :: 'a ⇒ int where
  denom ≡ snd ◦ Rep-Frac

function norm-frac :: int ⇒ int ⇒ int × int where
  norm-frac a b =
  (if b < 0 then norm-frac (−a) (−b)
   else if a = 0 ∨ b = 0 then (0, 1)
   else let c = gcd a b in (a div c, b div c))
  by pat-completeness auto
termination by (relation measure (λ(−, b). if b < 0 then 1 else 0)) auto

declare norm-frac.simps[simp del]

definition frac :: int ⇒ int ⇒ 'a where
  frac a b ≡ Abs-Frac (norm-frac a b)

definition plus-frac :: 'a ⇒ 'a ⇒ 'a where
  [nitpick-simp]: plus-frac q r =
  (let d = lcm (denom q) (denom r) in
   frac (num q * (d div denom q) + num r * (d div denom r)) d)

definition times-frac :: 'a ⇒ 'a ⇒ 'a where
  [nitpick-simp]: times-frac q r = frac (num q * num r) (denom q * denom r)

definition uminus-frac :: 'a ⇒ 'a where
  uminus-frac q ≡ Abs-Frac (− num q, denom q)

definition number-of-frac :: int ⇒ 'a where
  number-of-frac n ≡ Abs-Frac (− n, 1)

definition inverse-frac :: 'a ⇒ 'a where
  inverse-frac q ≡ frac (denom q) (num q)

definition less-frac :: 'a ⇒ 'a ⇒ bool where
  [nitpick-simp]: less-frac q r ←→ num (plus-frac q (uminus-frac r)) < 0

definition less-eq-frac :: 'a ⇒ 'a ⇒ bool where
  [nitpick-simp]: less-eq-frac q r ←→ num (plus-frac q (uminus-frac r)) ≤ 0
ML-file (Tools/Nitpick/kodkod.ML)
ML-file (Tools/Nitpick/kodkod-sat.ML)
ML-file (Tools/Nitpick/nitpick-impl.ML)
ML-file (Tools/Nitpick/nitpick-bisim.ML)
ML-file (Tools/Nitpick/nitpick-preproc.ML)
ML-file (Tools/Nitpick/nitpick-scope.ML)
ML-file (Tools/Nitpick/nitpick-sim.ML)
ML-file (Tools/Nitpick/nitpick-tests.ML)

setup :
Nitpick-HOL.register-ersatz-global
[ const-name card, const-name card',
  const-name sum, const-name sum',
  const-name fold-graph, const-name fold-graph',
  const-name wf, const-name wf',
  const-name wf-rec, const-name wf-rec',
  const-name wf-rec', const-name wf-rec']

hide-const (open) unknown is-unknown bisim bisim-iterator-max Quot safe-The FunBox PairBox Word prod
refl' w' card' sum' fold-graph' nat-gcd nat-lcm Frac Abs-Frac Rep-Frac
zero-frac one-frac num denom norm-frac frac plus-frac times-frac uminus-frac
number-of-frac
inverse-frac less-frac less-eq-frac of-frac wf-wfrec wf-wfrec wfrec'

hide-type (open) bisim-iterator fun-box pair-box unsigned-bit signed-bit word

hide-fact (open) Ex1-unfold rtrancl-unfold rtranclp-unfold tranclp-unfold prod-def
refl'-def w'-'def
87 Greatest Fixpoint (Codatatype) Operation on Bounded Natural Functors
begin

alias proj = Equiv-Relations.proj

lemma one-pointE: ![\forall x. s = x \Rightarrow P] \Rightarrow P
  by simp

lemma obj-sumE: ![\forall x. s = Inl x \to P; \forall x. s = Inr x \to P] \Rightarrow P
  by (cases s) auto

lemma not-TrueE: ¬ True =⇒ P
  by (erule notE, rule TrueI)

lemma neq-eq-eq-contradict: ![t \neq u; s = t; s = u] \Rightarrow P
  by fast

lemma converse-Times: (A × B)^{-1} = B × A
  by fast

lemma equiv-proj:
  assumes e: equiv A R and m: z \in R
  shows (proj R \circ fst) z = (proj R \circ snd) z
proof
  from m have z: (fst z, snd z) \in R by auto
  with e have \\[x. (fst z, x) \in R \to (snd z, x) \in R \land (snd z, x) \in R \to (fst z, x) \in R\]
    unfolding equiv-def sym-def trans-def by blast+
  then show ?thesis unfolding proj-def[abs-def] by auto
qed

definition image2 where image2 A f g = \{(f a, g a) | a. a \in A\}

lemma Id-on-Gr: Id-on A = Gr A id
  unfolding Id-on-def Gr-def by auto

lemma image2-eqI: ![b = f x; c = g x; x \in A] \Rightarrow (b, c) \in image2 A f g
  unfolding image2-def by auto

lemma IdD: (a, b) \in Id =⇒ a = b
  by auto

lemma image2-Gr: image2 A f g = (Gr A f)^{-1} O (Gr A g)
  unfolding image2-def Gr-def by auto

lemma GrD1: (x, fx) \in Gr A f =⇒ x \in A
  unfolding Gr-def by simp

lemma GrD2: (x, fx) \in Gr A f =⇒ f x = fx
unfolding Gr-def by simp

lemma Gr-incl: \( A \subseteq A \times B \iff f \cdot A \subseteq B \)
unfolding Gr-def by auto

lemma subset-Collect-iff: \( B \subseteq A \implies (B \subseteq \{ x \in A. \ P x \}) = (\forall x \in B. \ P x) \)
  by blast

lemma subset-CollectI: \( B \subseteq A \implies (\forall x \in B. \ P x) \implies \{ x \in A. \ P x \} \subseteq \{ x \in B. \ P x \} \)
  by blast

lemma in-rel-Collect-case-prod-eq: \( \text{in-rel}\ (\text{Collect}\ (\text{case-prod}\ X)) = X \)
unfolding fun-eq-iff by auto

lemma Collect-case-prod-in-rel-leI: \( X \subseteq Y \implies X \subseteq \text{Collect}\ (\text{case-prod}\ (\text{in-rel}\ Y)) \)
  by auto

lemma Collect-case-prod-in-rel-leE: \( X \subseteq \text{Collect}\ (\text{case-prod}\ (\text{in-rel}\ Y)) \implies (X \subseteq Y \implies R) \implies R \)
  by force

lemma conversep-in-rel: \( (\text{in-rel}\ R)^{-1}^{-1} = \text{in-rel}\ (R^{-1}) \)
unfolding fun-eq-iff by auto

lemma relcompp-in-rel: \( \text{in-rel}\ R \circ \text{in-rel}\ S = \text{in-rel}\ (R \circ S) \)
unfolding fun-eq-iff by auto

lemma in-rel-Gr: \( \text{in-rel}\ (\text{Gr}\ A f) = \text{Grp}\ A f \)
unfolding Gr-def Grp-def fun-eq-iff by auto

definition relImage where
  \( \text{relImage}\ R\ f \equiv \{ (f\ a1, f\ a2) \mid a1\ a2. \ (a1, a2) \in R \} \)

definition relInvImage where
  \( \text{relInvImage}\ A\ R\ f \equiv \{ (a1, a2) \mid a1\ a2. \ a1 \in A \land a2 \in A \land (f\ a1, f\ a2) \in R \} \)

lemma relImage-Gr:
  \( [R \subseteq A \times A] \implies \text{relImage}\ R\ f = (\text{Gr}\ A\ f)^{-1}\ O\ R\ O\ \text{Gr}\ A\ f \)
unfolding relImage-def Gr-def relcomp-def by auto

lemma relInvImage-Gr:
  \( [R \subseteq B \times B] \implies \text{relInvImage}\ A\ R\ f = \text{Gr}\ A\ f\ O\ R\ O\ (\text{Gr}\ A\ f)^{-1} \)
  unfolding Gr-def relcomp-def image-def relInvImage-def by auto

lemma relImage-mono:
  \( R1 \subseteq R2 \implies \text{relImage}\ R1\ f \subseteq \text{relImage}\ R2\ f \)
unfolding relImage-def by auto
lemma relInvImage-mono:
\[ R_1 \subseteq R_2 \implies \text{relInvImage } A \ R_1 f \subseteq \text{relInvImage } A \ R_2 f \]
unfolding relInvImage-def by auto

lemma relInvImage-Id-on:
\[ (\forall a_1 a_2. \ f a_1 = f a_2 \iff a_1 = a_2) \implies \text{relInvImage } A \ (\text{Id-on } B) \ f \subseteq \text{Id} \]
unfolding relInvImage-def Id-on-def by auto

lemma relInvImage-UNIV-relImage:
\[ R \subseteq \text{relInvImage } \text{UNIV} \ (\text{relImage } R \ f) \ f \]
unfolding relInvImage-def relImage-def by auto

lemma relImage-proj:
assumes equiv A R
shows \( \text{relImage } R \ (\text{proj } R) \subseteq \text{Id-on } (A/\{R}) \)
unfolding relImage-def Id-on-def
using proj-iff[OF assms] equiv-class-eq-iff[OF assms]
by (auto simp: proj-preserves)

lemma relImage-relInvImage:
assumes \( R \subseteq f \ \text{‘} A \times f \ \text{‘} A \)
shows \( \text{relImage } (\text{relInvImage } A \ R \ f) \ f = R \)
using assms unfolding relImage-def relInvImage-def by fast

lemma subst-Pair:
\[ P x y \implies a \quad (x, y) \implies P (\text{fst } a) (\text{snd } a) \]
by simp

lemma fst-diag-id:
\[ \text{fst} \circ (\lambda x. (x, x))) z = \text{id } z \]
by simp

lemma snd-diag-id:
\[ \text{snd} \circ (\lambda x. (x, x))) z = \text{id } z \]
by simp

lemma fst-diag-fst:
\[ \text{fst} \circ ((\lambda x. (x, x))) \circ \text{fst} = \text{fst} \]
by auto

lemma snd-diag-fst:
\[ \text{snd} \circ ((\lambda x. (x, x))) \circ \text{fst} = \text{fst} \]
by auto

lemma fst-diag-snd:
\[ \text{fst} \circ ((\lambda x. (x, x))) \circ \text{snd} = \text{snd} \]
by auto

lemma snd-diag-snd:
\[ \text{snd} \circ ((\lambda x. (x, x))) \circ \text{snd} = \text{snd} \]
by auto

definition Succ where Succ Kl kl = \{ k . kl \in [k] \in Kl \}
definition Shift where Shift Kl k = \{ kl. k \neq kl \in Kl \}
definition shift where shift lab k = (\lambda kl. lab (k \neq kl))

lemma empty-Shift:
\[ [][] \in Kl; k \in \text{Succ } Kl \[[]\] \implies [] \in \text{Shift } Kl k \]
unfolding Shift-def Succ-def by simp

lemma SuccD:
\[ k \in \text{Succ } Kl kl \implies kl \in [k] \in Kl \]
unfolding Succ-def by simp

lemmas SuccE = SuccD[elim-format]

lemma SuccI:
\[ kl \in [k] \in Kl \implies k \in \text{Succ } Kl kl \]
\textbf{unfolding} Succ-def by simp

\textbf{lemma} ShiftD: \(kl \in \text{Shift} Kl k \iff k \# kl \in Kl\)

\textbf{unfolding} Shift-def by simp

\textbf{lemma} Succ-Shift: \(\text{Succ} (\text{Shift} Kl k) kl = \text{Succ} Kl (k \# kl)\)

\textbf{unfolding} Succ-def Shift-def by auto

\textbf{lemma} length-Cons: \(\text{length} (x \# xs) = \text{Suc} (\text{length} xs)\)

by simp

\textbf{lemma} length-append-singleton: \(\text{length} (xs @ [x]) = \text{Suc} (\text{length} xs)\)

by simp

\textbf{definition} toCard-pred A r f \equiv \text{inj-on } f A \land f ' A \subseteq \text{Field} r \land \text{Card-order } r

\textbf{definition} toCard A r \equiv \text{SOME } f. \text{toCard-pred } A r f

\textbf{lemma} ex-toCard-pred:
\[|A| \leq o r; \text{Card-order } r \implies \exists f. \text{toCard-pred } A r f\]

\textbf{unfolding} toCard-pred-def

\textbf{using} card-of-ordLeq[of A Field r]

ordLeq-ordIso-trans[OF - card-of-unique[of Field r r], of |A|]

by blast

\textbf{lemma} toCard-inj:
\[|A| \leq o r; \text{Card-order } r; x \in A; y \in A \implies \text{toCard } A r x \leftrightarrow \text{toCard } A r y \leftrightarrow x = y\]

\textbf{using} toCard-pred-toCard

\textbf{unfolding} inj-on-def toCard-pred-def by blast

\textbf{definition} fromCard A r k \equiv \text{SOME } b. b \in A \land \text{toCard } A r b = k

\textbf{lemma} fromCard-toCard:
\[|A| \leq o r; \text{Card-order } r; b \in A \implies \text{fromCard } A r (\text{toCard } A r b) = b\]

\textbf{unfolding} fromCard-def by (rule some-equality) (auto simp add: toCard-inj)

\textbf{lemma} Inl-Field-csum: \(a \in \text{Field } r \implies \text{Inl } a \in \text{Field } (r + c s)\)

\textbf{unfolding} Field-card-of csum-def by auto

\textbf{lemma} Inr-Field-csum: \(a \in \text{Field } s \implies \text{Inr } a \in \text{Field } (r + c s)\)

\textbf{unfolding} Field-card-of csum-def by auto

\textbf{lemma} rec-nat-0-imp: \(f = \text{rec-nat } f1 (\lambda n \text{ rec. } f2 n \text{ rec}) \implies f 0 = f1\)

by auto

\textbf{lemma} rec-nat-Suc-imp: \(f = \text{rec-nat } f1 (\lambda n \text{ rec. } f2 n \text{ rec}) \implies f (\text{Suc } n) = f2 n (f\)
lemma rec-list-Nil-imp: \( f = \text{rec-list } f_1 \ (\lambda x \ xs \ rec. \ f_2 \ x \ xs \ rec) \implies f' = f_1 \) by auto

lemma rec-list-Cons-imp: \( f = \text{rec-list } f_1 \ (\lambda x \ xs \ rec. \ f_2 \ x \ xs \ rec) \implies f' (x \# xs) = f_2 \ x \ xs (f' \ xs) \) by auto

lemma not-arg-cong-Inr: \( x \neq y \implies \text{Inr } x \neq \text{Inr } y \) by simp

definition image2p where
image2p \( f \ g \ R \) = \( (\lambda x \ y. \ \exists x' \ y'. \ R \ x' \ y' \land f \ x' = x \land g \ y' = y) \)

lemma image2pI: \( R \ x \ y \implies \text{image2p } f \ g \ R \ (f \ x) \ (g \ y) \) unfolding image2p-def by blast

lemma image2pE: \([(\text{image2p } f \ g \ R \ (f \ x) \ (g \ y) \implies P)] \implies P \) unfolding image2p-def by blast

lemma rel-fun-iff-geq-image2p: \( \text{rel-fun } R \ S \ f \ g = (\text{image2p } f \ g \ R \leq S) \) unfolding rel-fun-def image2p-def by auto

lemma rel-fun-image2p: \( \text{rel-fun } R \ (\text{image2p } f \ g \ R) \ f \ g \) unfolding rel-fun-def image2p-def by auto

87.1 Equivalence relations, quotients, and Hilbert’s choice

lemma equiv-Eps-in:
\[\text{equiv } A \ r \ X : X \in A/\!/r \implies \text{Eps } (\lambda x. \ x \in X) \in X\]
apply (rule someI2-ex)
using in-quotient-imp-non-empty by blast

lemma equiv-Eps-preserves:
assumes ECH: \( \text{equiv } A \ r \ \text{and } X : X \in A/\!/r \)
shows \( \text{Eps } (\lambda x. \ x \in X) \in A \)
apply (rule in-mono[rule-format])
using assms apply (rule in-quotient-imp-subset)
by (rule equiv-Eps-in) (rule assms)+

lemma proj-Eps:
assumes equiv A r and X : X \in A/\!/r
shows proj r (\text{Eps } (\lambda x. \ x \in X)) = X
unfolding proj-def
proof auto
fix x assume x: x \in X
thus \((\text{Eps } (\lambda x. x \in X), x) \in r\) using assms equiv-Eps-in in-quotient-imp-in-rel by fast

next
fix \(x\) assume \((\text{Eps } (\lambda x. x \in X), x) \in r\)
thus \(x \in X\) using in-quotient-imp-closed[of assms equiv-Eps-in[of assms]] by fast

qed

definition \(\text{univ}\) where \(\text{univ } f \ X = f \ (\text{Eps } (\lambda x. x \in X))\)

lemma \(\text{univ-commute}\):
assumes \(\text{ECH} : \text{equiv } A \ r\) and \(\text{RES} : f \text{ respects } r\) and \(x : x \in A\)
shows \((\text{univ } f) \ (\text{proj } r \ x) = f \ x\)
proof (unfold \text{univ-def})
  have \(\text{prj} : \text{proj } r \ x \in A/\ r\) using \(x\) proj-preserves by fast
  hence \(\text{Eps } (\lambda y. \ y \in \text{proj } r \ x) \in A\) using \(\text{ECH} \ \text{equiv-Eps-preserves}\) by fast
  moreover have \(\text{proj } r \ (\text{Eps } (\lambda y. \ y \in \text{proj } r \ x)) = \text{proj } r \ x\) using \(\text{ECH} \ \text{prj proj-Eps}\) by fast
  ultimately have \((x, \text{Eps } (\lambda y. \ y \in \text{proj } r \ x)) \in r\) using \(x\) \(\text{ECH} \ \text{proj-iff}\) by fast
  thus \(f \ (\text{Eps } (\lambda y. \ y \in \text{proj } r \ x)) = f \ x\) using \(\text{RES}\) unfolding congruent-def by fastforce

qed

lemma \(\text{univ-preserves}\):
assumes \(\text{ECH} : \text{equiv } A \ r\) and \(\text{RES} : f \text{ respects } r\) and \(\text{PRES} : \forall x : x \in A. \ f \ x \in B\)
shows \(\forall X : X \in A/\ r. \ \text{univ } f \ X \in B\)
proof
fix \(X\) assume \(X : X \in A/\ r\)
then obtain \(x\) where \(x : x \in A\) and \(X : X = \text{proj } r \ x\) using \(\text{ECH} \ \text{proj-image[of r A]}\) by blast
  hence \(\text{univ } f \ X = f \ x\) using \(\text{ECH} \ \text{RES} \ \text{univ-commute}\) by fastforce
  thus \(\text{univ } f \ X \in B\) using \(x\) \(\text{PRES}\) by simp

qed

ML-file ⟨Tools/BNF/bnf-gfp-util.ML⟩
ML-file ⟨Tools/BNF/bnf-gfp-tactics.ML⟩
ML-file ⟨Tools/BNF/bnf-gfp.ML⟩
ML-file ⟨Tools/BNF/bnf-gfp-rec-sugar-tactics.ML⟩
ML-file ⟨Tools/BNF/bnf-gfp-rec-sugar.ML⟩

end

88 Filters on predicates

theory Filter
imports Set-Interval Lifting-Set
begin
88.1 Filters

This definition also allows non-proper filters.

\texttt{locale is-filter =}
\texttt{\hskip 1em fixes F :: ('a ⇒ bool) ⇒ bool}
\texttt{\hskip 1em assumes True: F (λx. True)}
\texttt{\hskip 1em assumes conj: F (λx. P x) ⇒ F (λx. Q x) ⇒ F (λx. P x ∧ Q x)}
\texttt{\hskip 1em assumes mono: \forall x. P x \implies Q x \implies F (λx. P x) \implies F (λx. Q x)}

\texttt{typedef 'a filter =}
\texttt{\hskip 1em \{ F :: ('a ⇒ bool) ⇒ bool. is-filter F \}}

\texttt{proof}
\texttt{\hskip 1em show (λx. True) ∈ ?filter by (auto intro: is-filter.intro)}
\texttt{\hskip 1em qed}

\texttt{lemma is-filter-Rep-filter: is-filter (Rep-filter F)}
\texttt{\hskip 1em using Rep-filter[of F] by simp}

\texttt{lemma Abs-filter-inverse':}
\texttt{\hskip 1em assumes is-filter F shows Rep-filter (Abs-filter F) = F}
\texttt{\hskip 1em using assms by (simp add: Abs-filter-inverse)}

88.1.1 Eventually

\texttt{definition eventually :: ('a ⇒ bool) ⇒ 'a filter ⇒ bool}
\texttt{\hskip 1em where eventually P F ←→ Rep-filter F P}

\texttt{syntax}
\texttt{\hskip 1em -eventually :: pttrn => 'a filter => bool => bool ((∀F - in -./ -) [0, 0, 10] 10)}
\texttt{translations}
\texttt{\hskip 1em \forall_F x \ in \ F. P => CONST eventually (λx. P) F}

\texttt{lemma eventually-Abs-filter:}
\texttt{\hskip 1em assumes is-filter F shows eventually P (Abs-filter F) = F P}
\texttt{\hskip 1em unfolding eventually-def using assms by (simp add: Abs-filter-inverse)}

\texttt{lemma filter-eq-iff:}
\texttt{\hskip 1em shows F = F' ←→ (∀P. eventually P F = eventually P F')}
\texttt{\hskip 1em unfolding Rep-filter-inject [symmetric] fun-eq-iff eventually-def \...}

\texttt{lemma eventually-True [simp]: eventually (λx. True) F}
\texttt{\hskip 1em unfolding eventually-def}
\texttt{\hskip 1em by (rule is-filter.True [OF is-filter-Rep-filter])}

\texttt{lemma always-eventually: \forall x. P x \implies eventually P F}
\texttt{\hskip 1em proof}
\texttt{\hskip 1em assume \forall x. P x hence P = (λx. True) by (simp add: ext)}
\texttt{\hskip 1em thus eventually P F by simp}
\texttt{\hskip 1em qed}
theory "Filter"

lemma eventuallyI: (∀ x. P x) ===> eventually P F
  by (auto intro: always-eventually)

lemma eventually-mono:
  [eventually P F; ∀ x. P x ===> Q x] ===> eventually Q F
  unfolding eventually-def
  by (blast intro: is-filter.mono [OF is-filter-Rep-filter])

lemma eventually-conj:
  assumes P: eventually (λx. P x) F
  assumes Q: eventually (λx. Q x) F
  shows eventually (λx. P x ∧ Q x) F
  using assms unfolding eventually-def
  by (rule is-filter.conj [OF is-filter-Rep-filter])

lemma eventually-mp:
  assumes eventually (λx. P x → Q x) F
  assumes eventually (λx. P x) F
  shows eventually (λx. Q x) F

proof –
  have eventually (λx. (P x → Q x) ∧ P x) F
    using assms by (rule eventually-conj)
  then show ?thesis
    by (blast intro: eventually-mono)
qed

lemma eventually-rev-mp:
  assumes eventually (λx. P x) F
  assumes eventually (λx. P x → Q x) F
  shows eventually (λx. Q x) F
  using assms(2) assms(1) by (rule eventually-mp)

lemma eventually-conj-iff:
  eventually (λx. P x ∧ Q x) F ↔ eventually P F ∧ eventually Q F
  by (auto intro: eventually-conj elim: eventually-rev-mp)

lemma eventually-elim2:
  assumes eventually (λi. P i) F
  assumes eventually (λi. Q i) F
  assumes (∀ i. P i ===> Q i ===> R i)
  shows eventually (λi. R i) F
  using assms by (auto elim!: eventually-rev-mp)

lemma eventually-ball-finite-distrib:
  finite A ===> (eventually (λx. ∀ y∈A. P x y) net) ↔ (∀ y∈A. eventually (λx. P x y) net)
  by (induction A rule: finite-induct) (auto simp: eventually-conj-iff)
lemma eventually-ball-finite:
finite A \implies \forall y \in A. eventually (\lambda x. P x y) net \implies eventually (\lambda x. \forall y \in A. P x y) net
by (auto simp: eventually-ball-finite-distrib)

lemma eventually-all-finite:
fixes P :: 'a \Rightarrow 'b :: finite \Rightarrow bool
assumes \forall y. eventually (\lambda x. P x y) net
shows eventually (\lambda x. \forall y. P x y) net
using eventually-ball-finite [of UNIV P] assms by simp

lemma eventually-ex:
(\forall F x in F. \exists y. P x y) \iff (\exists Y. \forall F x in F. P x (Y x))
proof
assume \forall F x in F. \exists y. P x y
then have \forall F x in F. P x (SOME y. P x y)
  by (auto intro: someI-ex eventually-mono)
then show \exists Y. \forall F x in F. P x (Y x)
  by auto
qed (auto intro: eventually-mono)

lemma not-eventually-impl:
eventually P F \implies \neg eventually Q F \implies \neg eventually (\lambda x. P x \rightarrow Q x) F
by (auto intro: eventually-mp)

lemma not-eventuallyD:
\neg eventually P F \implies \exists x. \neg P x
by (metis always-eventually)

lemma eventually-subst:
assumes eventually (\lambda n. P n = Q n) F
shows eventually P F = eventually Q F (is ?L = ?R)
proof
from assms have eventually (\lambda x. P x \rightarrow Q x) F
  and eventually (\lambda x. Q x \rightarrow P x) F
  by (auto elim: eventually-mono)
then show ?thesis by (auto elim: eventually-elim2)
qed

88.2 Frequently as dual to eventually

definition frequently :: ('a \Rightarrow bool) \Rightarrow 'a filter \Rightarrow bool
  where frequently P F \iff \neg eventually (\lambda x. \neg P x) F

syntax
  \exists F x in F. P == CONST frequently (\lambda x. P) F

translations
  \exists F x in F. P == CONST frequently (\lambda x. P) F

lemma not-frequently-False [simp]: \neg (\exists F x in F. False)
  by (simp add: frequently-def)
lemma frequently-ex: \( \exists F x \in F. P x \rightarrow \exists x. P x \)
by (auto simp: frequently-def dest: not-eventuallyD)

lemma frequentlyE: assumes frequently \( P \) \( F \) obtains \( x \) where \( P x \)
using frequently-ex[of assms] by auto

lemma frequently-mp:
assumes ev: \( \forall_F x \in F. P x \rightarrow Q x \) and \( P: \exists_F x \in F. P x \) shows \( \exists_F x \in F. Q x \)
proof –
from ev have eventually (\( \lambda x. \neg Q x \rightarrow \neg P x \)) \( F \)
by (rule eventually-rev-mp) (auto intro!: always-eventually)
from eventually-mp[of this] \( P \) show \( \forall \)thesis
by (auto simp: frequently-def)
qed

lemma frequently-rev-mp:
assumes \( \exists_F x \in F. P x \)
assumes \( \forall_F x \in F. P x \rightarrow Q x \)
shows \( \exists_F x \in F. Q x \)
using assms(2) assms(1) by (rule frequently-mp)

lemma frequently-mono: \((\forall x. P x \rightarrow Q x) \rightarrow \forall x. F x \rightarrow \forall x. Q x \)
using frequently-mp[of P Q] by (simp add: always-eventually)

lemma frequently-elim1: \( \exists_F x \in F. P x \rightarrow (\forall i. P i \rightarrow Q i) \rightarrow \exists_F x \in F. Q x \)
by (metis frequently-mono)

lemma frequently-disj-iff: \( \exists_F x \in F. P x \lor Q x \leftrightarrow (\exists_F x \in F. P x) \lor (\exists_F x \in F. Q x) \)
by (simp add: frequently-def eventually-conj-iff)

lemma frequently-disj: \( \exists_F x \in F. P x \rightarrow \exists_F x \in F. Q x \rightarrow \exists_F x \in F. P x \lor Q x \)
by (simp add: frequently-disj-iff)

lemma frequently-bex-finite-distrib:
assumes finite A shows \( \exists_F x \in F. \exists y \in A. P x y \leftrightarrow (\exists y \in A. \exists_F x \in F. P x y) \)
using assms by induction (auto simp: frequently-disj-iff)

lemma frequently-bex-finite: finite A \( \rightarrow \exists_F x \in F. \exists y \in A. P x y \rightarrow \exists y \in A. \exists_F x \in F. P x y \)
by (simp add: frequently-bex-finite-distrib)

lemma frequently-all: \( \exists_F x \in F. \forall y. P x y \leftrightarrow (\forall Y. \exists_F x \in F. P x (Y y)) \)
THEORY "Filter"

using eventually-ex[of λx y. ¬ P x y F] by (simp add: frequently-def)

lemma
shows not-eventually: ¬ eventually P F ←→ (∃ F. x y. ¬ P x y F)
and not-frequently: ¬ frequently P F ←→ (∀ F. x y. ¬ P x y F)
by (auto simp: frequently-def)

lemma frequently-imp-iff:
(∃ F. x y. P x y Q x) ←→ (eventually P F → frequently Q F)
unfolding imp-conj-disj frequently-disj-iff not-eventually[symmetric] ..

lemma eventually-frequently-const-simps:
(∃ F. x y. P x y C) ←→ (∃ F. x y. P x y) ∧ C
(∃ F. x y. P x y C) ←→ C ∧ (∃ F. x y. P x y)
(∀ F. x y. P x y C) ←→ (∀ F. x y. P x y) ∨ C
(∀ F. x y. P x y C) ←→ C ∨ (∀ F. x y. P x y)
(∀ F. x y. P x y C) ←→ ((∃ F. x y. P x y) → C)
(∀ F. x y. P x y C) ←→ (C → (∃ F. x y. P x y))
by (cases C; simp add: not-frequently)+

lemmas eventually-frequently-simps =
eventually-frequently-const-simps
not-eventually
eventually-conj-iff
eventually-ball-finite-distrib
eventually-ex
not-frequently
frequently-disj-iff
frequently-bex-finite-distrib
frequently-all
frequently-imp-iff

ML
fun eventually-elim-tac facts =
  CONTEXT-SUBGOAL (fn (goal, i) => fn (ctxt, st) =>
    let
      val mp-facts = facts RL @{thms eventually-rev-mp}
      val rule =
        @{thm eventuallyI}
      |> fold (fn mp-fact => fn th => th RS mp-fact) mp-facts
      |> funpow (length facts) (fn th => @{thm impI} RS th)
      val cases-prop =
        Thm.prop-of (Rule-Cases.internalize-params (rule RS Goal.init (Thm.cterm_of ctxt goal))
      val cases = Rule-Cases.make-common ctxt cases-prop ([|elim, []|], [[]])
    in CONTEXT-CASES cases (resolve-tac ctxt [rule] i) (ctxt, st) end)
  )

method-setup eventually-elim = (}
88.2.1 Finer-than relation

$F \leq F'$ means that filter $F$ is finer than filter $F'$.

**instantiation**

```plaintext
filter :: (type) complete-lattice
begin

definition le-filter-def:
  $F \leq F'$ $\iff$ ($\forall P. \; \text{eventually } P F' \rightarrow \text{eventually } P F$)

definition
  ($F :: 'a \text{ filter}$) $< F' \iff F \leq F' \land \neg F' \leq F$

definition
  top = Abs-filter ($\lambda P. \; \forall x. \; P x$)

definition
  bot = Abs-filter ($\lambda P. \; \text{True}$)

definition
  sup $F F' = Abs-filter (\lambda P. \; \text{eventually } P F \land \text{eventually } P F')$

definition
  inf $F F' = Abs-filter
    (\lambda P. \; \exists Q R. \; \text{eventually } Q F \land \text{eventually } R F' \land (\forall x. \; Q x \land R x \rightarrow P x))$

definition
  Sup $S = Abs-filter (\lambda P. \; \forall F \in S. \; \text{eventually } P F)$

definition
  Inf $S = Sup \{F :: 'a \text{ filter}. \; \forall F' \in S. \; F \leq F\}'$

lemma eventually-top [simp]: eventually $P$ top $\iff$ ($\forall x. \; P x$)
  unfolding top-filter-def
  by (rule eventually-Abs-filter, rule is-filter.intro, auto)

lemma eventually-bot [simp]: eventually $P$ bot
  unfolding bot-filter-def
  by (subst eventually-Abs-filter, rule is-filter.intro, auto)

lemma eventually-sup:
  eventually $P$ (sup $F F'$) $\iff$ eventually $P F \land$ eventually $P F'$
  unfolding sup-filter-def
  by (rule eventually-Abs-filter, rule is-filter.intro)
    (auto elim!: eventually-rev-mp)
```
lemma eventually-inf:
eventually P (inf F F') ↔
(∃ Q R. eventually Q F ∧ eventually R F' ∧ (∀ x. Q x ∧ R x → P x))
unfolding inf-filter-def
apply (rule eventually-Abs-filter, rule is-filter.intro)
apply (fast intro: eventually-True)
apply clarify
apply (intro exI conjI)
apply (erule (1) eventually-conj)
apply simp
apply auto
done

lemma eventually-Sup:
eventually P (Sup S) ↔ (∀ F ∈ S. eventually P F)
unfolding Sup-filter-def
apply (rule eventually-Abs-filter, rule is-filter.intro)
apply (auto intro: eventually-conj elim!: eventually-rev-mp)
done

instance proof
fix F F' F'' :: 'a filter and S :: 'a filter set
{ show F < F' ↔ F ≤ F' ∧ ¬ F' ≤ F
  by (rule less-filter-def) }
{ show F ≤ F
  unfolding le-filter-def by simp }
{ assume F ≤ F' and F' ≤ F'' thus F ≤ F''
  unfolding le-filter-def by simp }
{ assume F ≤ F' and F ≤ F'' thus F ≤ inf F F''
  unfolding le-filter-def eventually-inf
  by (auto intro: eventually-mono [OF eventually-conj]) }
{ show F ≤ sup F F' and inf F F' ≤ F'
  unfolding le-filter-def eventually-sup by simp-all }
{ assume F ≤ F'' and F' ≤ F'' thus sup F F' ≤ F''
  unfolding le-filter-def eventually-sup by simp }
{ assume F'' ∈ S thus Inf S ≤ F''
  unfolding le-filter-def Inf-filter-def eventually-Sup Ball-def by simp }
{ assume ∃ F', F'' ∈ S. F ≤ F' thus F ≤ Inf S
  unfolding le-filter-def Inf-filter-def eventually-Sup Ball-def by simp }
{ assume F ∈ S thus F ≤ sup S
  unfolding le-filter-def eventually-Sup by simp }
{ assume ∃ F. F ∈ S. F ≤ F' thus sup S ≤ F'
  unfolding le-filter-def eventually-Sup by simp }
{ show Inf {} = (top::'a filter) }
by (auto simp: top-filter-def Inf-filter-def Sup-filter-def)
  (metis (full-types) top-filter-def always-eventually eventually-top)

{ show Sup {} = (bot::'a filter)
  by (auto simp: bot-filter-def Sup-filter-def)
qed

end

instance filter :: (type) distrib-lattice
proof
  fix F G H :: 'a filter
  show sup F (inf G H) = inf (sup F G) (sup F H)
proof (rule order.antisym)
    show inf (sup F G) (sup F H) ≤ sup F (inf G H)
      unfolding le-filter-def eventually-sup
    proof safe
      fix P assume 1: eventually P F and 2: eventually P (inf G H)
      from 2 obtain Q R
        where QR: eventually Q G eventually R H \x. Q x \implies R x \implies P x
        by (auto simp: eventually-inf)
      define Q' where Q' = (\x. Q x \or P x)
      define R' where R' = (\x. R x \or P x)
      from 1 have eventually Q' F
        by (elim eventually-mono) (auto simp: Q'-def)
      moreover from 1 have eventually R' F
        by (elim eventually-mono) (auto simp: R'-def)
      moreover from QR(1) have eventually Q' G
        by (elim eventually-mono) (auto simp: Q'-def)
      moreover from QR(2) have eventually R' H
        by (elim eventually-mono)(auto simp: R'-def)
      moreover from QR have P x if Q' x R' x for x
        using that by (auto simp: Q'-def R'-def)
      ultimately show eventually P (inf (sup F G) (sup F H))
        by (auto simp: eventually-inf eventually-sup)
    qed
  qed (auto intro: inf.coboundedI1 inf.coboundedI2)
qed

lemma filter-leD:
  F ≤ F' \implies eventually P F' \implies eventually P F
unfolding le-filter-def by simp

lemma filter-leI:
  (∀P. eventually P F' \implies eventually P F) \implies F ≤ F'
unfolding le-filter-def by simp

lemma eventually-False:
  eventually (\x. False) F \iff F = bot
unfolding filter-eq-iff by (auto elim: eventually-rev-mp)

lemma eventually-frequently: \( F \neq \text{bot} \implies \text{eventually } P F \implies \text{frequently } P F \)
using eventually-conj[of \( P F \) \( \lambda x. \neg P x \)]
by (auto simp add: frequently-def eventually-False)

lemma eventually-frequentlyE:
assumes eventually \( P F \)
assumes eventually \( (\lambda x. \neg P x \lor Q x) \ F \ F \neq \text{bot} \)
shows frequently \( Q F \)
proof –
  have eventually \( Q F \)
  using eventually-conj[of \( \text{assms} \ (1,2) \) simplified] by (auto elim: eventually-mono)
  then show \( ?\text{thesis} \) using eventually-frequently[of \( F \neq \text{bot} \)] by auto
qed

lemma eventually-const-iff: eventually \( (\lambda x. P) F \) \( \iff \) \( P \lor F \neq \text{bot} \)
by (cases \( P \)) (auto simp: eventually-False)

lemma eventually-const[simp]: \( F \neq \text{bot} \implies \text{eventually } (\lambda x. P) F \implies P \)
by (simp add: eventually-const-iff)

lemma frequently-const-iff: frequently \( (\lambda x. P) F \implies P \land F \neq \text{bot} \)
by (simp add: frequently-def eventually-const-iff)

lemma frequently-const[simp]: \( F \neq \text{bot} \implies \text{frequently } (\lambda x. P) F \implies P \)
by (simp add: frequently-def eventually-const-iff)

lemma eventually-happens: eventually \( P \) net \( \implies \) net \( = \text{bot} \lor (\exists x. P x) \)
by (metis frequentlyE eventually-frequently)

lemma eventually-happens':
assumes \( F \neq \text{bot} \) eventually \( P F \)
shows \( \exists x. P x \)
using assms eventually-frequently frequentlyE by blast

abbreviation (input) trivial-limit :: 'a filter \( \Rightarrow \) bool
where trivial-limit \( F \equiv F = \text{bot} \)

lemma trivial-limit-def: trivial-limit \( F \equiv \) eventually \( (\lambda x. \text{False}) F \)
by (rule eventually-False [symmetric])

lemma False-imp-not-eventually: \( (\forall x. \neg P x ) \implies \neg \text{trivial-limit net} \implies \neg \) eventually \( (\lambda x. P x) \) net
by (simp add: eventually-False)

lemma eventually-Inf: eventually \( P (\text{Inf } B) \equiv \) \( (\exists X \subseteq B. \text{finite } X \land \) eventually \( P (\text{Inf } X)) \)
proof –
let \( ?F = \lambda P. \exists X \subseteq B. \text{finite } X \land \text{eventually } P \ (\text{Inf } X) \)

{ fix P have eventually P (Abs-filter ?F) \(\iff\) ?F P 
  proof (rule eventually-Abs-filter is-filter.intro) + 
  show ?F (\(\lambda x. \text{True}\)) 
    by (rule exI[of - {}]) (simp add: le-fun-def) 
  next 
    fix P Q 
    assume ?F P then guess X .. 
    moreover assume ?F Q then guess Y .. 
    ultimately show ?F (\(\lambda x. \ P x \land Q x\)) 
      by (intro exI[of - X \cup Y]) 
        (auto simp: Inf-union-distrib eventually-inf) 
  next 
    fix P Q 
    assume ?F P then guess X .. 
    moreover assume \(\forall x. \ P x \rightarrow Q x\) 
    ultimately show ?F Q 
      by (intro exI[of - X]) (auto elim: eventually-mono) 
  qed 
  note eventually-F = this 
}

have Inf B = Abs-filter ?F 
  proof (intro antisym Inf-greatest) 
    show Inf B \(\leq\) Abs-filter ?F 
      by (auto simp: le-filter-def eventually-F dest: Inf-superset-mono) 
  next 
    fix F assume F \(\in\) B then show Abs-filter ?F \(\leq\) F 
      by (auto simp add: le-filter-def eventually-F intro: exI[of - \{F\}]) 
  qed 
  then show ?thesis 
    by (simp add: eventually-F) 
  qed 

lemma eventually-INF: eventually P (\(\prod b \in B. \ F b\)) \(\iff\) \(\exists X \subseteq B. \text{finite } X \land \text{eventually } P \ (\text{Inf } X\)) 
  unfolding eventually-Inf[of P F:B] 
  by (metis finite-imageI image-mono finite-subset-image)

lemma Inf-filter-not-bot: 
  fixes B :: \('a\) filter set 
  shows \((\bigwedge X. \ X \subseteq B \implies \text{finite } X \implies \text{Inf } X \neq \text{bot}) \implies \text{Inf } B \neq \text{bot}\) 
  unfolding trivial-limit-def eventually-Inf[of - B] 
    bot-bool-def [symmetric] bot-fun-def [symmetric] bot-unique by simp

lemma INF-filter-not-bot: 
  fixes F :: \('i \Rightarrow \ 'a\) filter 
  shows \((\bigwedge X. \ X \subseteq B \implies \text{finite } X \implies (\prod b \in X. \ F b) \neq \text{bot}) \implies (\prod b \in B. \ F b)\)
\[ \neq \text{bot} \]

unfolding trivial-limit-def eventually-INF [of - B]

\[ \text{bot-bool-def [symmetric] bot-fun-def [symmetric] bot-unique by simp} \]

lemma eventually-Inf-base:
assumes \( B \neq \{ \} \) and base: \( \forall F G. F \in B \implies G \in B \implies \exists x \in B. \ x \leq \inf F G \)
shows eventually P (Inf B) \( \iff \) (\( \exists b \in B. \ \text{eventually P} \ b \) )
proof (subst eventually-Inf, safe)
fix X assume finite X X \( \subseteq \) B
then have \( \exists b \in B. \ \forall x \in X. \ b \leq x \)
proof induc
case empty then show \(?case\)
using \( \{ B \neq \{ \} \} \) by auto
next
case (insert x X)
then obtain b when b \( \in \) B \( \land \) \( \forall x. \ x \in X \implies b \leq x \)
by auto
with (insert x X \( \subseteq \) B) base[of b x] show \(?case\)
by (auto intro: order-trans)
qed
then obtain b when b \( \in \) B \( b \leq \) Inf X
by (auto simp: le-Inf-iff)
then show eventually P (Inf X) \( \implies \) Bex B (eventually P)
by (intro bexI[of - b]) (auto simp: le-filter-def)
qed (auto intro!: exI[of - \{ x \} for x])

lemma eventually-INF-base:
\( B \neq \{ \} \implies (\forall a \ b. \ a \in B \implies b \in B \implies \exists x \in B. \ F x \leq \inf (F a) (F b)) \implies \)
eventually P (\( \prod b \in B. \ F b \) ) \( \iff \) (\( \exists b \in B. \ \text{eventually P} \ (F b) \) )
by (subst eventually-Inf-base) auto

lemma eventually-INF1: \( i \in I \implies \text{eventually P} \ (F i) \implies \text{eventually P} \ (\prod i \in I. \ F i) \)

lemma eventually-INF-finite:
assumes finite A
shows eventually P (\( \prod x \in A. \ F x \) ) \( \iff \)
(\( \exists Q. \ (\forall x \in A. \ \text{eventually} \ (Q x) \ (F x) ) \land (\forall y. \ (\forall x \in A. \ Q x y \implies P y) \) )
using assms
proof (induction arbitrary: P rule: finite-induct)
case (insert a A P)
from insert.hyps have [simp]: \( x \neq a \) if \( x \in A \) for x
using that by auto
have eventually P (\( \prod x \in \text{insert a A. F x} \) ) \( \iff \)
(\( \exists Q R S. \ \text{eventually} \ Q (F a) \land ((\forall x \in A. \ \text{eventually} \ (S x) \ (F x)) \land \)
(\( \forall y. \ (\forall x \in A. \ S x y \implies R y) \land (\forall x. \ Q x \land R x \implies P x) \))
unfolding ex-simps by (simp add: eventually-inf insert.IH)
also have \( \ldots \iff (\exists Q. \ (\forall x \in \text{insert a A. eventually} \ (Q x) \ (F x)) \land \)
THEORY "Filter"

(∀ y. (∀ x∈ insert a A. Q x y) → P y))

proof (safe, goal-cases)
case (1 Q R S)
thus ?case using 1 by (intro exI[of - S(a := Q)]) auto
next
case (2 Q)
show ?case
  by (rule exI[of - Q], rule exI[of - λ y. ∀ x∈ A. Q x y],
      rule exI[of - Q(a := (λ -. True))]) (use 2 in auto)
qed
finally show ?case .
qed auto

88.2.2 Map function for filters

definition filtermap :: ('a ⇒ 'b) ⇒ 'a filter ⇒ 'b filter
where filtermap f F = Abs-filter (λP. eventually (λx. P (f x)) F)

lemma eventually-filtermap:
  eventually P (filtermap f F) = eventually (λx. P (f x)) F
unfolding filtermap-def
apply (rule eventually-Abs-filter)
apply (rule is-filter.intro)
apply (auto elim!: eventually-rev-mp)
done

lemma filtermap-iden: filtermap (λx. x) F = F
by (simp add: filter-eq-iff eventually-filtermap)

lemma filtermap-filtermap:
  filtermap f (filtermap g F) = filtermap (λx. f (g x)) F
by (simp add: filter-eq-iff eventually-filtermap)

lemma filtermap-mono: F ≤ F' =⇒ filtermap f F ≤ filtermap f F'
unfolding le-filter-def eventually-filtermap by simp

lemma filtermap-bot [simp]: filtermap f bot = bot
by (simp add: filter-eq-iff eventually-filtermap)

lemma filtermap-bot-iff: filtermap f F = bot =⇒ F = bot
by (simp add: trivial-limit-def eventually-filtermap)

lemma filtermap-sup: filtermap f (sup F1 F2) = sup (filtermap f F1) (filtermap f F2)
by (simp add: filter-eq-iff eventually-filtermap eventually-sup)

lemma filtermap-SUP: filtermap f (⨆ b∈B. F b) = (⨆ b∈B. filtermap f (F b))
by (simp add: filter-eq-iff eventually-Sup eventually-filtermap)
lemma filtermap-inf: filtermap f (inf F1 F2) ≤ inf (filtermap f F1) (filtermap f F2)
  by (intro inf-greatest filtermap-mono inf-sup-ord)

lemma filtermap-INF: filtermap f (\( \prod b\in B \). F b) ≤ (\( \prod b\in B \). filtermap f (F b))
  by (rule INF-greatest, rule filtermap-mono, erule INF-lower)

88.2.3 Contravariant map function for filters

definition filtercomap :: (′a ⇒ ′b) ⇒ ′b filter ⇒ ′a filter
  where
    filtercomap f F = Abs-filter (λP. \( \exists Q. \) eventually Q F \( \land \) (\( \forall x. \) Q (f x) → P x))

lemma eventually-filtercomap:
  eventually P (filtercomap f F) ←→ (\( \exists Q. \) eventually Q F \( \land \) (\( \forall x. \) Q (f x) → P x))

unfolding filtercomap-def
proof (intro eventually-Abs-filter, unfold-locales, goal-cases)
  case 1
  show ?case by (auto intro!: exI [of - λ- True])

next
  case (2 P Q)
  from 2(1) guess P' by (elim exE conjE) note P' = this
  from 2(2) guess Q' by (elim exE conjE) note Q' = this
  show ?case
    by (rule exI [of - λx. P' x ∧ Q' x])
    (insert P' Q', auto intro!: eventually-conj)

next
  case (3 P Q)
  thus ?case by blast
qed

lemma filtercomap-ident: filtercomap (λx. x) F = F
  by (auto simp: filter-eq-iff eventually-filtercomap elim!: eventually mono)

lemma filtercomap-filtercomap: filtercomap f (filtercomap g F) = filtercomap (λx. g (f x)) F
  unfolding filter-eq-iff by (auto simp: eventually-filtercomap)

lemma filtercomap-mono: F ≤ F' → filtercomap f F ≤ filtercomap f F'
  by (auto simp: eventually-filtercomap le-filter-def)

lemma filtercomap-bot [simp]: filtercomap f bot = bot
  by (auto simp: filter-eq-iff eventually-filtercomap)

lemma filtercomap-top [simp]: filtercomap f top = top
  by (auto simp: filter-eq-iff eventually-filtercomap)

lemma filtercomap-inf: filtercomap f (inf F1 F2) = inf (filtercomap f F1) (filtercomap f F2)
unfolding filter-eq-iff
proof safe
fix P
assume eventually P (filtercomap f (F1 ∩ F2))
then obtain Q R S where *:
eventually Q F1 eventually R F2 \(\forall x. Q x \implies R x \implies S x \land x. S (f x) \implies P x\)

unfolding eventually-filtercomap eventually-inf by blast
from * have eventually (\(\lambda x. Q (f x)\)) (filtercomap f F1)
eventually (\(\lambda x. R (f x)\)) (filtercomap f F2)
by (auto simp: eventually-filtercomap)
with * show eventually P (filtercomap f F1 \cap filtercomap f F2)
by (auto simp: eventually-inf)
with * show eventually P (filtercomap f (F1 ∩ F2))
qed

lemma filtercomap-sup: filtercomap f \((\sup F1 F2)\) ≥ \(\sup (\text{filtercomap } f F1) (\text{filtercomap } f F2)\)
by (intro sup-least filtercomap-mono inf-sup-ord)

lemma filtercomap-INF: filtercomap f \((\prod b\in B. F b)\) = \((\prod b\in B. \text{filtercomap } f (F b))\)
proof —
have *: filtercomap f \((\prod b\in B. F b)\) = \((\prod b\in B. \text{filtercomap } f (F b))\) if finite B
for B
using that by induction (simp-all add: filtercomap-inf)
show ?thesis unfolding filter-eq-iff
proof
fix P
have eventually P \((\prod b\in B. \text{filtercomap } f (F b))\) \iff
(\(\exists X. (X \subseteq B \land \text{finite } X) \land \text{eventually } P (\prod b\in X. \text{filtercomap } f (F b))\))
by (subst eventually-INF) blast
also have \ldots \iff \(\exists X. (X \subseteq B \land \text{finite } X) \land \text{eventually } P (\text{filtercomap } f (\prod b\in X. F b))\))
by (rule ex-cong) (simp add: *)
also have \ldots \iff eventually P (filtercomap f (\prod (F ^ i B)))
unfolding eventually-filtercomap by (subst eventually-INF) blast
finally show eventually P (filtercomap f (\prod (F ^ i B))) =
eventually P (\prod b\in B. \text{filtercomap } f (F b)) ..
qed
qed

lemma filtercomap-SUP:
filtercomap f (⨆ b∈B. F b) ≥ (⨆ b∈B. filtercomap f (F b))
by (intro SUP-least filtercomap-mono SUP-upper)

lemma filtermap-le-iff-le-filtercomap: filtermap f F ≤ G ←→ F ≤ filtercomap f G
unfolding le-filter-def eventually-filtermap eventually-filtercomap
using eventually-mono by auto

lemma filtercomap-neq-bot:
assumes ⋀ P. eventually P F =⇒ ∃ x. P (f x)
shows filtercomap f F ≠ bot
using assms by (auto simp: trivial-limit-def eventually-filtercomap)

lemma filtercomap-neq-bot-surj:
assumes F ≠ bot and surj f
shows filtercomap f F ≠ bot
proof (rule filtercomap-neq-bot)
fix P assume *: eventually P F
show ∃ x. P (f x)
proof (rule ccontr)
assume **: ¬(∃ x. P (f x))
from * have eventually (λ -. False) F
proof eventually-elim
  case (elim x)
  from (surj f) obtain y where x = f y by auto
  with elim and ** show False by auto
qed
with assms show False by (simp add: trivial-limit-def)
qed

lemma eventually-filtercomapI [intro]:
assumes eventually P F
shows eventually (λ x. P (f x)) (filtercomap f F)
using assms by (auto simp: eventually-filtercomap)

lemma filtermap-filtercomap: filtermap f (filtercomap f F) ≤ F
by (auto simp: le-filter-def eventually-filtermap eventually-filtercomap)

lemma filtercomap-filtermap: filtercomap f (filtermap f F) ≥ F
unfolding le-filter-def eventually-filtermap eventually-filtercomap
by (auto elim!: eventually-mono)

88.2.4 Standard filters

definition principal :: 'a set ⇒ 'a filter where
principal S = Abs-filter (λP. ∀ x∈S. P x)
lemma eventually-principal: \( \text{eventually} \ P \ (\text{principal} \ S) \iff (\forall x \in S. \ P \ x) \)

unfolding principal-def
by (rule eventually-Abs-filter, rule is-filter.intro) auto

lemma eventually-inf-principal: \( \text{eventually} \ P \ (\text{inf} \ F \ (\text{principal} \ s)) \iff \text{eventually} \ (\lambda x. \ x \in s \implies P \ x) \ F \)

unfolding eventually-inf eventually-principal by (auto elim: eventually-mono)

lemma principal-UNIV[simp]: \( \text{principal} \ \text{UNIV} = \text{top} \)
by (auto simp: filter-eq-iff eventually-principal)

lemma principal-empty[simp]: \( \text{principal} \ \{\} = \text{bot} \)
by (auto simp: filter-eq-iff eventually-principal)

lemma principal-eq-bot-iff: \( \text{principal} \ X = \text{bot} \iff X = \{\} \)
by (auto simp add: eq-iff)

lemma principal-le-iff: \( \text{principal} \ A \leq \text{principal} \ B \iff A \subseteq B \)
by (auto simp: le-filter-def eventually-principal)

lemma le-principal: \( F \leq \text{principal} \ A \iff \text{eventually} \ (\lambda x. \ x \in A) \ F \)

unfolding le-filter-def eventually-principal
apply safe
apply (erule_tac x = \lambda x. \ x \in A in allE)
apply (auto elim: eventually-mono)
done

lemma principal-inject[iff]: \( \text{principal} \ A = \text{principal} \ B \iff A = B \)
unfolding eq-iff by simp

lemma sup-principal[simp]: \( \text{sup} \ (\text{principal} \ A) \ (\text{principal} \ B) = \text{principal} \ (A \cup B) \)

unfolding filter-eq-iff eventually-sup eventually-principal by auto

lemma inf-principal[simp]: \( \text{inf} \ (\text{principal} \ A) \ (\text{principal} \ B) = \text{principal} \ (A \cap B) \)

unfolding filter-eq-iff eventually-inf eventually-principal
by (auto intro: exI[of \- \lambda x. \ x \in A] exI[of \- \lambda x. \ x \in B])

lemma SUP-principal[simp]: \( \bigsqcup \{ \text{principal} \ A \ (i) \ \mid i \in I \} = \text{principal} \ (\bigsqcup \{ A \ (i) \ \mid i \in I \}) \)
unfolding filter-eq-iff eventually-Sup by (auto simp: eventually-principal)

lemma INF-principal-finite: \( \text{finite} \ X \implies (\bigsqcap x \in X. \text{principal} \ (f \ x)) = \text{principal} \ (\bigsqcap x \in X. f \ x) \)
by (induct X rule: finite-induct) auto

lemma filtermap-principal[simp]: \( \text{filtermap} \ f \ (\text{principal} \ A) = \text{principal} \ (f \ A) \)
unfolding filter-eq-iff eventually-filtermap eventually-principal by simp

lemma filtercomap-principal[simp]: \( \text{filtercomap} \ f \ (\text{principal} \ A) = \text{principal} \ (f \ A) \)
A) unfolding filter-eq-iff eventually-filtercomap eventually-principal by fast

88.2.5 Order filters
definition at-top :: (′a::order) filter
  where at-top = (⋂ k. principal {k ..})

lemma at-top-sub: at-top = (⋂ k∈{c::′a::linorder..}. principal {k ..})
  by (auto intro!: INF-eq max.cobounded1 max.cobounded2 simp: at-top-def)

lemma eventually-at-top-linorder: eventually P at-top ⨾→ (∃ N::′a::linorder. ∀ n≥N. P n)
  unfolding at-top-def
  by (subst eventually-INF-base) (auto simp: eventually-principal intro: max.cobounded1 max.cobounded2)

lemma eventually-filtercomap-at-top-linorder:
  eventually P (filtercomap f at-top) ⨾→ (∃ N::′a::linorder. ∀ x. f x ≥ N −→ P x)
  by (auto simp: eventually-filtercomap eventually-at-top-linorder)

lemma eventually-at-top-linorderI:
  fixes c::′a::linorder
  assumes ∀ x. c ≤ x =⇒ P x
  shows eventually P at-top using assms by (auto simp: eventually-at-top-linorder)

lemma eventually-ge-at-top [simp]:
  eventually (λx. (c::::linorder) ≤ x) at-top
  unfolding eventually-at-top-linorder by auto

lemma eventually-at-top-dense: eventually P at-top ⨾→ (∃ N::′a::{no-top, linorder}. ∀ n>N. P n)
  proof –
    have eventually P (⋂ k. principal {k <..}) ⨾→ (∃ N::′a. ∀ n>N. P n)
      by (subst eventually-INF-base) (auto simp: eventually-principal intro: max.cobounded1 max.cobounded2)
    also have (⋂ k. principal {k::′a <..}) = at-top
      unfolding at-top-def
      by (intro INF-eq) (auto intro: less-imp-le simp: Ici-subset-Ioi-iff gt-ex)
    finally show ?thesis .
  qed

lemma eventually-filtercomap-at-top-dense:
  eventually P (filtercomap f at-top) ⨾→ (∃ N::′a::{no-top, linorder}. ∀ x. f x > N −→ P x)
  by (auto simp: eventually-filtercomap eventually-at-top-dense)

lemma eventually-at-top-not-equal [simp]: eventually (λx::′a::{no-top, linorder}.}
THEORY "Filter"

\[ x \neq c \] at-top

unfolding eventually-at-top-dense by auto

lemma eventually-gt-at-top [simp]: eventually (\( \lambda x. \ (c :::: \{\text{no-top, linorder}\}) < x \)) at-top

unfolding eventually-at-top-dense by auto

lemma eventually-all-ge-at-top:
  assumes eventually P (at-top :: (a :::: linorder) filter)
  shows eventually (\( \lambda x. \forall y \geq x. \ P y \)) at-top

proof –
  from assms obtain x where \( \forall y \geq x \Rightarrow P y \) by (auto simp: eventually-at-top-linorder)
  hence \( \forall z \geq y. \ P z \) if \( y \geq x \) for \( y \) using that by simp
  thus \?thesis by (auto simp: eventually-at-top-linorder)
qed

definition at-bot :: (a::order) filter
  where \( \text{at-bot} = (\bigcap k. \text{principal} \{.. k\}) \)

lemma at-bot-sub: at-bot = (\( \bigcap k\in\{.. c::a::linorder\}. \text{principal} \{.. k\} \))
  by (auto intro: INF-eq min.cobounded1 min.cobounded2 simp: at-bot-def)

lemma eventually-at-bot-linorder:
  fixes P :: a::linorder => bool shows eventually P at-bot \( \longleftrightarrow \( \exists N. \forall n \leq N. \ P n \) \)
  unfolding at-bot-def
  by (subst eventually-INF-base) (auto simp: eventually-principal intro: min.cobounded1 min.cobounded2)

lemma eventually-filtercomap-at-bot-linorder:
  eventually P (filtercomap f at-bot) \( \longleftrightarrow \( \exists N. a::\text{linorder}. \forall x. f x \leq N \Rightarrow P x \) \)
  by (auto simp: eventually-filtercomap eventually-at-bot-linorder)

lemma eventually-le-at-bot [simp]:
  eventually (\( \lambda x. \ x \leq (c::::linorder) \)) at-bot

unfolding eventually-at-bot-linorder by auto

lemma eventually-at-bot-dense: eventually P at-bot \( \longleftrightarrow \( \exists N::a::\{\text{no-bot, linorder}\}. \forall n < N. \ P n \) \)

proof –
  have eventually P (\( \bigcap k. \text{principal} \{.. k\} \)) \( \longleftrightarrow \( \exists N::a. \forall n < N. \ P n \) \)
    by (subst eventually-INF-base) (auto simp: eventually-principal intro: min.cobounded1 min.cobounded2)
  also have \( \bigcap k. \text{principal} \{.. k::a\} = \text{at-bot} \)
    unfolding at-bot-def
    by (intro INF-eq) (auto intro: less-imp-le simp: Iic-subset-Iio-iff lt-ex)
  finally show \?thesis .
qed

lemma eventually-filtercomap-at-bot-dense:
THEORY "Filter"

eventually $P$ (filtercomap $f$ at-bot) $\iff \exists N :: a::\{no-bot, linorder\}. \forall x. f x < N \rightarrow P x$
by (auto simp: eventually-filtercomap eventually-at-bot-dense)

lemma eventually-at-bot-not-equal [simp]: eventually $(\lambda x::a::\{no-bot, linorder\}. x \neq c)$ at-bot
  unfolding eventually-at-bot-dense by auto

lemma eventually-gt-at-bot [simp]:
  eventually $(\lambda x. x < (c:::unbounded-dense-linorder))$ at-bot
  unfolding eventually-at-bot-dense by auto

lemma trivial-limit-at-bot-linorder [simp]: $\neg$ trivial-limit (at-bot ::('a::linorder) filter)
  unfolding trivial-limit-def
  by (metis eventually-at-bot-linorder order-refl)

lemma trivial-limit-at-top-linorder [simp]: $\neg$ trivial-limit (at-top ::('a::linorder) filter)
  unfolding trivial-limit-def
  by (metis eventually-at-top-linorder order-refl)

88.3 Sequentially

abbreviation sequentially :: nat filter
  where sequentially $\equiv$ at-top

lemma eventually-sequentially:
  eventually $P$ sequentially $\iff (\exists N. \forall n \geq N. P n)$
  by (rule eventually-at-top-linorder)

lemma sequentially-bot [simp, intro]: sequentially $\neq$ bot
  unfolding filter-eq-iff eventually-sequentially by auto

lemmas trivial-limit-sequentially = sequentially-bot

lemma eventually-False-sequentially [simp]:
  $\neg$ eventually $(\lambda n. False)$ sequentially
  by (simp add: eventually-False)

lemma le-sequentially:
  $F \leq$ sequentially $\iff (\forall N. \text{eventually} (\lambda n. N \leq n) F)$
  by (simp add: at-top-def le-INF-iff le-principal)

lemma eventually-sequentiallyI [intro?]:
  assumes $\forall x. c \leq x \implies P x$
  shows eventually $P$ sequentially
  using assms by (auto simp: eventually-sequentially)
lemma eventually-sequentially-Suc [simp]: eventually (λi. P (Suc i)) sequentially ←→ eventually P sequentially
  unfolding eventually-sequentially by (metis Suc-le-D Suc-le-mono le-Suc-eq)

lemma eventually-sequentially-seg [simp]: eventually (λn. P (n + k)) sequentially ←→ eventually P sequentially
  using eventually-sequentially-Suc[of λn. P (n + k)] by (induction k) auto

lemma filtermap-sequentially-ne-bot: filtermap f sequentially ≠ bot
  by (simp add: filtermap-bot-iff)

88.4 Increasing finite subsets

definition finite-subsets-at-top where
  finite-subsets-at-top A = (∏ X∈{X. finite X ∧ X ⊆ A}, principal {Y. finite Y ∧ X ⊆ Y ∧ Y ⊆ A})

lemma eventually-finite-subsets-at-top:
  eventually P (finite-subsets-at-top A) ←→
  (∃ X. finite X ∧ X ⊆ A ∧ (∀ Y. finite Y ∧ X ⊆ Y ∧ Y ⊆ A → P Y))
  unfolding finite-subsets-at-top-def
proof (subst eventually-INF-base, goal-cases)
  show {X. finite X ∧ X ⊆ A} ≠ {} by auto
next
  case (2 B C)
  thus ?case by (intro bexI[of - B ∪ C]) auto
qed (simp-all add: eventually-principal)

lemma eventually-finite-subsets-at-top-weakI [intro]:
  assumes \( \forall X. \text{finite } X \implies X \subseteq A \implies P X \)
  shows eventually P (finite-subsets-at-top A)
proof –
  have eventually (λX. finite X ∧ X ⊆ A) (finite-subsets-at-top A)
    by (auto simp: eventually-finite-subsets-at-top)
  thus ?thesis by eventually-elim (use assms in auto)
qed

lemma finite-subsets-at-top-neq-bot [simp]: finite-subsets-at-top A ≠ bot
proof –
  have ¬eventually (λx. False) (finite-subsets-at-top A)
    by (auto simp: eventually-finite-subsets-at-top)
  thus ?thesis by auto
qed

lemma filtermap-image-finite-subsets-at-top:
  assumes inj-on f A
  shows filtermap (λx. f (finite-subsets-at-top A)) = finite-subsets-at-top (f " A)
  unfolding filter-eq-iff eventually-filtermap
proof (safe, goal-cases)
THEORY "Filter"

case (1 P)
then obtain X where X: finite X X ⊆ A \ Y. finite Y → X ⊆ Y → Y ⊆ A → P (f Y)
  unfolding eventually-finite-subsets-at-top by force
show ?case unfolding eventually-finite-subsets-at-top eventually-filtermap
proof (rule exI[of - f X], intro conjI allI impI, goal-cases)
  case (3 Y)
  with assms and X(1,2) have P (f Y Y ∩ A) using X(1,2)
  by (intro X(3) finite-vimage-IntI) auto
also have f Y Y ∩ A = Y using assms 3 by blast
finally show ?case .
qued (insert assms X(1,2), auto intro!: finite-vimage-IntI)
next
case (2 P)
then obtain X where X: finite X X ⊆ f A \ Y. finite Y → X ⊆ Y → Y ⊆ f A → P Y
  unfolding eventually-finite-subsets-at-top by force
show ?case unfolding eventually-finite-subsets-at-top eventually-filtermap
proof (rule exI[of - f - Y ∩ A], intro conjI allI impI, goal-cases)
  case (3 Y)
  with X(1,2) and assms show ?case by (intro X(3)) force+
qued (insert assms X(1), auto intro!: finite-vimage-IntI)

lemma eventually-finite-subsets-at-top-finite:
  assumes finite A
  shows eventually P (finite-subsets-at-top A) ←→ P A
  unfolding eventually-finite-subsets-at-top using assms by force

lemma finite-subsets-at-top-finite: finite A → finite-subsets-at-top A = principal {A}
  by (auto simp: filter-eq-iff eventually-finite-subsets-at-top-finite eventually-principal)

88.5 The cofinite filter

definition cofinite = Abs-filter (λP. finite {x. ¬ P x})

abbreviation Inf-many :: ('a ⇒ bool) ⇒ bool (binder ∃∞ 10)
  where Inf-many P ≡ frequently P cofinite

abbreviation Alm-all :: ('a ⇒ bool) ⇒ bool (binder ∀∞ 10)
  where Alm-all P ≡ eventually P cofinite

notation (ASCII)
f
Inf-many (binder INFM 10) and
Alm-all (binder MOST 10)

lemma eventually-cofinite: eventually P cofinite ←→ finite {x. ¬ P x}
  unfolding cofinite-def
proof (rule eventually-Abs-filter, rule is-filter.intro)
fix P Q :: 'a ⇒ bool
assume finite {x. ¬ P x}
finitesubset (OF this)
show finite {x. ¬ (P x ∧ Q x)}
  by (rule rev-finite-subset auto)
next
fix P Q :: 'a ⇒ bool
assume P: finite {x. ¬ P x}
and *: ∀x. P x ⇒ Q x
from * show finite {x.¬ Q x}
  by (intro finite-subset[OF - P]) auto
qed simp

lemma frequently-cofinite: frequently P cofinite ⇐¬ finite {x. P x}
  by (simp add: frequently-def eventually-cofinite)

lemma cofinite-bot[simp]: cofinite = (bot::'a filter) ⇐ finite (UNIV :: 'a)
  unfolding trivial-limit-def eventually-cofinite by simp

lemma cofinite-eq-sequentially: cofinite = sequentially
  unfolding filter-eq-iff eventually-sequentially eventually-cofinite
proof safe
fix P :: nat ⇒ bool
assume [simp]: finite {x. ¬ P x}
show ∃ N. ∀ n ≥ N. P n
proof cases
  assume [simp]: {x. ¬ P x} ≠ {} then show ?thesis
  by (intro exI[of - Suc (Max {x. ¬ P x})]) (auto simp: Suc-le-eq)
qed auto
next
fix P :: nat ⇒ bool and N :: nat
assume ∀ n ≥ N. P n
then have {x. ¬ P x} ⊆ {..< N}
  by (auto simp: not-le)
then show finite {x. ¬ P x}
  by (blast intro: finite-subset)
qed

88.5.1 Product of filters

definition prod-filter :: 'a filter ⇒ 'b filter ⇒ ('a × 'b) filter (infixr ×F 80)
where
prod-filter F G =
  (⨆ (P, Q) ∈ {(P, Q). eventually P F ∧ eventually Q G}, principal {(x, y). P x ∧ Q y})

lemma eventually-prod-filter: eventually P (F ×F G) ⇐
  (∃ Pf Pg. eventually Pf F ∧ eventually P g G ∧ (∀ x y. Pf x ⇒ P g y ⇒ P (x, y)))
  unfolding prod-filter-def
proof (subst eventually-INF-base, goal-cases)
  case 2
  moreover have eventually Pf F ⇒ eventually Qf F ⇒ eventually P g G ⇒ eventually Qg G ⇒
\[
\exists P \ Q, \ \text{eventually } P \ F \land \ \text{eventually } Q \ G \\
\text{Collect } P \times \text{Collect } Q \subseteq \text{Collect } P F \times \text{Collect } P g \cap \text{Collect } Q f \times \text{Collect } Q g
\]
for \( P f \ P g Q f Q g \)

by \((\text{intro conjI exI[of - inf Pf Qf] exI[of - inf Pg Qg])})

(auto simp: inf-fun-def eventually-conj)

ultimately show ?case

by auto

qed (auto simp: eventually-principal intro: eventually-True)

\begin{enumerate}
\item \textbf{lemma eventually-prod1:}
\begin{enumerate}
\item \textbf{assumes} \( B \neq \text{bot} \)
\item \textbf{shows} \((\forall F (x, y) \in A \times_F B. \ P x) \leftrightarrow (\forall F x \in A. \ P x)\)
\item \textbf{unfolding} \text{eventually-prod-filter}
\item \textbf{proof} safe
\item fix \( R \ Q \)
\item assume \( \ast: \forall F x \in A. \ R x \land \forall F x \in B. \ Q x \land x y. \ R x \rightarrow Q y \rightarrow P x \)
\item with \((B \neq \text{bot})\) obtain \( y \) where \( Q y \) by (auto dest: eventually-happens)
\item with \( \ast \) show \( \text{eventually } P A \)
\item by (force elim: eventually-mono)
\item next
\item assume \( \text{eventually } P A \)
\item then show \( \exists P f P g. \ \text{eventually } P f A \land \text{eventually } P g B \land (\forall x y. \ P f x \rightarrow P g y \rightarrow P x) \)
\item by (intro exI[of - P] exI[of - \lambda x. True]) auto
\item qed
\end{enumerate}
\item \textbf{lemma eventually-prod2:}
\begin{enumerate}
\item \textbf{assumes} \( A \neq \text{bot} \)
\item \textbf{shows} \((\forall F (x, y) \in A \times_F B. \ P y) \leftrightarrow (\forall F y \in B. \ P y)\)
\item \textbf{unfolding} \text{eventually-prod-filter}
\item \textbf{proof} safe
\item fix \( R \ Q \)
\item assume \( \ast: \forall F x \in A. \ R x \land \forall F x \in B. \ Q x \land x y. \ R x \rightarrow Q y \rightarrow P y \)
\item with \((A \neq \text{bot})\) obtain \( x \) where \( R x \) by (auto dest: eventually-happens)
\item with \( \ast \) show \( \text{eventually } P B \)
\item by (force elim: eventually-mono)
\item next
\item assume \( \text{eventually } P B \)
\item then show \( \exists P f P g. \ \text{eventually } P f A \land \text{eventually } P g B \land (\forall x y. \ P f x \rightarrow P g y \rightarrow P y) \)
\item by (intro exI[of - P] exI[of - \lambda x. True]) auto
\item qed
\end{enumerate}
\item \textbf{lemma INF-filter-bot-base:}
\begin{enumerate}
\item fixes \( F :: 'a \Rightarrow 'b \) \text{filter}
\item assumes \( \ast: \lambda i j. \ i \in I \Rightarrow j \in I \Rightarrow \exists k \in I. \ F k \leq F i \land F j \)
\item shows \((\prod i \in I. \ F i) = \text{bot} \leftrightarrow (\exists i \in I. \ F i = \text{bot})\)
\item \textbf{proof} (cases \( \exists i \in I. \ F i = \text{bot} \))
\item case True
\end{enumerate}
\end{enumerate}
then have $\bigcap i \in I. F_i \leq \text{bot}$
  by (auto intro: INF-lower2)
with True show ?thesis
  by (auto simp: bot-unique)
next
case False
moreover have $\bigcap i \in I. F_i \neq \text{bot}$
proof (cases $I = \{\}$)
  case True
then show ?thesis
  by (auto simp add: filter-eq-iff)
next
case False'
show ?thesis
proof (rule INF-filter-not-bot)
  fix $J$
  assume finite $J$. $J \subseteq I$
  then have $\exists k \in I. F_k \leq \bigcap i \in J. F_i$
  proof (induct $J$
    case empty
    then show ?case
      using $I \neq \{\}$ by auto
  next
    case (insert $i$ $J$
      then obtain $k$ where $k \in I. F_k \leq \bigcap i \in J. F_i$
        by auto
      with insert *[of $i$ $k$] show ?case
        by auto
    qed
  qed
  qed
  ultimately show ?thesis
    by auto
qed

lemma Collect-empty-eq-bot: Collect $P = \{\} \longleftrightarrow P = \bot$
  by auto

lemma prod-filter-eq-bot: $A \times_F B = \text{bot} \longleftrightarrow A = \text{bot} \lor B = \text{bot}$
  unfolding trivial-limit-def
proof
  assume $\forall x \in A. F x. A. False$
  then obtain $Pf Pg$
    where $Pf$: eventually $(\lambda x. P f x) A$ and $Pg$: eventually $(\lambda y. P g y) B$
    and $\ast$: $\forall x y. P f x \longrightarrow P g y \longrightarrow False$
    unfolding eventually-prod-filter by fast
  from $\ast$ have $(\forall x. \neg Pf x) \lor (\forall y. \neg Pg y)$ by fast
  with $Pf Pg$ show $(\forall x \in A. F x) \lor (\forall x \in B. F x)$ by auto
next
  assume $(\forall F \ x \ in \ A. \ False) \lor (\forall F \ x \ in \ B. \ False)$
  then show $\forall F \ x \ in \ A \times F \ B. \ False$
    unfolding eventually-prod-filter by (force intro: eventually-True)
  qed

lemma prod-filter-mono: $F \leq F' \implies G \leq G' \implies F \times F G \leq F' \times F G'$
  by (auto simp: le-filter-def eventually-prod-filter)

lemma prod-filter-mono-iff:
  assumes nAB: $A \neq bot \land B \neq bot$
  shows $A \times F B \leq C \times F D \iff A \leq C \land B \leq D$
proof safe
  assume *: $A \times F B \leq C \times F D$
  with assms have $A \times F B \neq bot$
    by (auto simp: bot-unique prod-filter-eq-bot)
  with * have $C \times F D \neq bot$
    by (auto simp: bot-unique)
  then have nCD: $C \neq bot \land D \neq bot$
    by (auto simp: prod-filter-eq-bot)
  show $A \leq C$
    proof (rule filter-leI)
      fix $P$
      assume eventually $P C$ with *
      show eventually $P A$
        using nAB nCD by (simp add: eventually-prod1 eventually-prod2)
    qed

  show $B \leq D$
    proof (rule filter-leI)
      fix $P$
      assume eventually $P D$ with *
      show eventually $P B$
        using nAB nCD by (simp add: eventually-prod1 eventually-prod2)
    qed
  qed (intro prod-filter-mono)

lemma eventually-prod-same: eventually $P (F \times F F)$ $\iff$ $(\exists Q. \mbox{ eventually } Q F \land (\forall x y. \ Q x \implies Q y \implies P (x, y)))$
  unfolding eventually-prod-filter
  apply safe
  apply (rule_tac $x=\inf Pf Pg$ in exI)
  apply (auto simp: inf-fun-def intro!: eventually-conj)
  done

lemma eventually-prod-sequentially:
  eventually $P (\mbox{ sequentially } \times F \ \mbox{ sequentially})$ $\iff (\exists N. \mbox{ $\forall m \geq N \implies \forall n \geq N. \ P (n, m)$})$
  unfolding eventually-prod-same eventually-sequentially by auto
THEORY “Filter”

lemma principal-prod-principal: principal $A \times F$ principal $B = principal (A \times B)$
unfolding filter-eq-iff eventually-prod-filter eventually-principal
by (fast intro: exI[of - λx. x ∈ A] exI[of - λx. x ∈ B])

lemma le-prod-filterI:
  filtermap fst $F \leq A \implies$ filtermap snd $F \leq B \implies F \leq A \times F \ B$
unfolding le-filter-def eventually-filtermap eventually-prod-filter
by (force elim: eventually-elim2)

lemma filtermap-fst-prod-filter: filtermap fst $(A \times F \ B) \leq A$
unfolding le-filter-def eventually-filtermap eventually-prod-filter
by (force intro: eventually-True)

lemma filtermap-snd-prod-filter: filtermap snd $(A \times F \ B) \leq B$
unfolding le-filter-def eventually-filtermap eventually-prod-filter
by (force intro: eventually-True)

lemma prod-filter-INF:
assumes $I \neq \{\}$ and $J \neq \{\}$
shows $(\prod i \in I. A i) \times_F (\prod j \in J. B j) = (\prod i \in I. \prod j \in J. A i \times_F B j)$
proof (rule antisym)
  from $I \neq \{\}$ obtain $i$ where $i \in I$ by auto
  from $J \neq \{\}$ obtain $j$ where $j \in J$ by auto

show $(\prod i \in I. \prod j \in J. A i \times_F B j) \leq (\prod i \in I. A i) \times_F (\prod j \in J. B j)$
  by (fast intro: le-prod-filterI INF-greatest INF-lower2
    order-trans[OF filtermap-INF] (i ∈ I \ j ∈ J)
    filtermap-fst-prod-filter filtermap-snd-prod-filter)
show $(\prod i \in I. A i) \times_F (\prod j \in J. B j) \leq (\prod i \in I. \prod j \in J. A i \times_F B j)$
  by (intro INF-greatest prod-filter-mono INF-lower)
qed

lemma filtermap-Pair: filtermap $(\lambda x. (f x, g x)) F \leq filtermap \ f \ F \times_F filtermap \ g \ F$
  by (rule le-prod-filterI, simp-all add: filtermap-filtermap)

lemma eventually-prodI: eventually $P \ F \implies$ eventually $Q \ G \implies$ eventually $(\lambda x. P \ (f x) \land Q \ (\snd x)) \ (F \times_F G)$
unfolding eventually-prod-filter by auto

lemma prod-filter-INF1: $I \neq \{\} \implies (\prod i \in I. A i) \times_F B = (\prod i \in I. A i \times_F B)$
using prod-filter-INF[of $I \ {B}$] $A \ \lambda x. \ x$ by simp

lemma prod-filter-INF2: $J \neq \{\} \implies A \times_F (\prod i \in J. B i) = (\prod i \in J. A \times_F B i)$
using prod-filter-INF[of $\{A\} \ J \ \lambda x. \ x \ B$] by simp

lemma prod-filtermap1: prod-filter $(\text{filtermap} \ f \ F) G = \text{filtermap} \ (\text{afst} \ f) \ (\text{prod-filter} \ F \ G)$
apply(clarsimp simp add: filter-eq-iff eventually-filtermap eventually-prod-filter;
THEORY "Filter"

1610

safe)
subgoal by auto
subgoal for P Q R by (rule exI [where x = λ y. ∃ x. y = f x ∧ Q x]) (auto intro: eventually-mono)
done

lemma prod-filtermap2: prod-filter F (filtermap g G) = filtermap (apsnd g) (prod-filter F G)
apply (clarsimp simp add: filter-eq-iff eventually-filtermap eventually-prod-filter; safe)
subgoal by auto
subgoal for P Q R by (auto intro: exI [where x = λ y. ∃ x. y = g x ∧ R x])
eventually-mono)
done

lemma prod-filter-assoc:
prod-filter (prod-filter F G) H = filtermap (λ (x, y, z). ((x, y), z)) (prod-filter F (prod-filter G H))
apply (clarsimp simp add: filter-eq-iff eventually-filtermap eventually-prod-filter; safe)
subgoal for P Q R S T by (auto 4 4 intro: exI [where x = λ (a, b). T a ∧ S b])
subgoal for P Q R S T by (auto 4 3 intro: exI [where x = λ (a, b). Q a ∧ S b])
done

lemma prod-filter-principal-singleton: prod-filter (principal {x}) F = filtermap (Pair x) F
by (fastforce simp add: filter-eq-iff eventually-prod-filter eventually-principal eventually-filtermap elim: eventually-mono intro: exI [where x = λ a. a = x])

lemma prod-filter-principal-singleton2: prod-filter F (principal {x}) = filtermap (λ a. (a, x)) F
by (fastforce simp add: filter-eq-iff eventually-prod-filter eventually-principal eventually-filtermap elim: eventually-mono intro: exI [where x = λ a. a = x])

lemma prod-filter-commute: prod-filter F G = filtermap prod.swap (prod-filter G F)
by (auto simp add: filter-eq-iff eventually-prod-filter eventually-filtermap)

88.6 Limits

definition filterlim :: (′a ⇒ ′b) ⇒ ′b filter ⇒ ′a filter ⇒ bool where
filterlim f F2 F1 ⇔ filtermap f F1 ≤ F2

syntax
-LIM :: pttrns ⇒ ′a ⇒ ′b ⇒ ′a ⇒ bool ((3LIM (-)/ (-))/ (-) ⇒ (-)) [1000, 10, 0, 10] 10

translations
LIM x F1. f :: F2 == CONST filterlim (λx. f) F2 F1
lemma filterlim-top [simp]: filterlim f top F
  by (simp add: filterlim-def)

lemma filterlim-iff:
  (LIM x F1. f x => F2) <-> (∀ P. eventually P F2 -> eventually (λx. P (f x)) F1)
  unfolding filterlim-def le-filter-def eventually-filtermap ..

lemma filterlim-compose:
  filterlim g F3 F2 => filterlim f F2 F1 => filterlim (λx. g (f x)) F3 F1
  unfolding filterlim-def filtermap-filtermap[symmetric] by (metis filtermap-mono order-trans)

lemma filterlim-mono:
  filterlim f F2 F1 => F2 <= F2' => F1' <= F1 => filterlim f F2' F1'
  unfolding filterlim-def by (metis filtermap-mono order-trans)

lemma filterlim-ident: LIM x F. x => F
  by (simp add: filterlim-def filtermap-ident)

lemma filterlim-cong:
  F1 = F1' => F2 = F2' => eventually (λx. f x = g x) F2 => filterlim f F1 F2 = filterlim g F1' F2'
  by (auto simp: filterlim-def le-filter-def eventually-filtermap elim: eventually-elim2)

lemma filterlim-mono-eventually:
  assumes filterlim f F G and ord: F <= F' G' <= G
  assumes eq: eventually (λx. f x = g x) F2 => filterlim f F1 F2 = filterlim g F1' F2'
  shows filterlim f' F' G'
  apply (rule filterlim-cong[OF refl refl eq, THEN iffD1])
  apply (rule filterlim-mono[OF - ord])
  apply fact
  done

lemma filtermap-mono-strong: inj f => filtermap f F <= filtermap f G <-> F <= G
  apply (safe intro: filtermap-mono)
  apply (auto simp: le-filter-def eventually-filtermap)
  apply (erule_tac x=λx. P (inv f x) in allE)
  apply auto
  done

lemma eventually-compose-filterlim:
  assumes eventually P F filterlim f F G
  shows eventually (λx. P (f x)) G
  using assms by (simp add: filterlim-iff)

lemma filtermap-eq-strong: inj f => filtermap f F = filtermap f G <-> F = G
by (simp add: filtermap-mono-strong eq-iff)

lemma filtermap-fun-inverse:
  assumes g: filterlim g F G
  assumes f: filterlim f G F
  assumes ev: eventually \((\lambda x. f (g x)) = x\) G
  shows filtermap f F = G
proof (rule antisym)
  show filtermap f F \leq G
    using f unfolding filterlim-def.
  have G = filtermap f (filtermap g G) using ev
    by (auto elim: eventually-elim2 simp: filter-eq-iff eventually-filtermap)
  also have ... \leq filtermap f F
    using g by (intro filtermap-mono) (simp add: filterlim-def)
  finally show G \leq filtermap f F .
qed

lemma filterlim-principal:
(LIM x F. f x :> principal S) \longleftrightarrow (eventually \((\lambda x. f x \in S)\) F)
unfolding filterlim-def eventually-filtermap le-principal ..

lemma filterlim-filtercomap [intro]: filterlim f G (filtercomap f F)
unfolding filterlim-def by (rule filtermap-filtercomap)

lemma filterlim-inf:
(LIM x F1. f x :> inf F2 F3) \longleftrightarrow ((LIM x F1. f x :> F2) \land (LIM x F1. f x :> F3))
unfolding filterlim-def by simp

lemma filterlim-INF:
(LIM x F. f x :> (\prod b \in B. G b)) \longleftrightarrow (\forall b \in B. LIM x F. f x :> G b)
unfolding filterlim-def le-INF-iff ..

lemma filterlim-INF-INF:
(\forall m. m \in J \Longrightarrow \exists i \in I. filtermap f (F i) \leq G m) \Longrightarrow LIM x (\prod i \in I. F i). f x :> (\prod j \in J. G j)
unfolding filterlim-def by (rule order-trans[OF filtermap-INF INF-mono])

lemma filterlim-INF': x \in A \Longrightarrow filterlim f F (G x) \Longrightarrow filterlim f F (\prod x \in A. G x)
unfolding filterlim-def by (rule order.trans[OF filtermap-mono[OF INF-lower]])

lemma filterlim-filtercomap-iff: filterlim f (filtercomap g G) F \longleftrightarrow filterlim (g o f) G F
  by (simp add: filterlim-def filtermap-le-iff-le-filtercomap filtercomap-filtercomap o-def)

lemma filterlim-iff-le-filtercomap: filterlim f F G \longleftrightarrow G \leq filtermap f F
  by (simp add: filterlim-def filtermap-le-iff-le-filtercomap)
lemma filterlim-base:
  \((\forall m. m \in J \Rightarrow i m \in I) \Rightarrow (\forall x. x \in F \Rightarrow f x \in G)\)

  \(LIM x (\bigcap_{i \in I} \text{principal} (F i)). f x : (\bigcap_{j \in J} \text{principal} (G j))\)

  by (force intro!: filterlim-INF-INF simp: image-subset-iff)

lemma filterlim-base-iff:
  assumes \(I \neq \{\}\) and chain: \(\bigwedge i. j \in I \Rightarrow j \in I \Rightarrow F i \subseteq F j \vee F j \subseteq F i\)

  shows \((LIM x (\bigcap_{i \in I} \text{principal} (F i)). f x : (\bigcap_{j \in J} \text{principal} (G j)) \iff\)

    \((\forall j. \exists i. \forall x \in F i. f x \in G j)\)

  unfolding filterlim-INF filterlim-principal

proof (subst eventually-INF-base)

  fix i j

  assume i \(\in I\) j \(\in I\)

  with chain[OF this] show \(\exists x \in I. \text{principal} (F x) \leq \inf (\text{principal} (F i)) (\text{principal} (F j))\)

    by auto

  qed (auto simp: eventually-principal \(\langle I \neq \{\}\rangle\))

lemma filterlim-filtermap: filterlim \(f F1 \ (\text{filtermap} \ g F2) = \text{filterlim} \ (\lambda x. f (g x)) F1 F2\)

  unfolding filterlim-def filtermap-filtermap ..

lemma filterlim-sup:

  filterlim \(f F1 \Rightarrow f F2 \Rightarrow f (\sup F1 F2)\)

  unfolding filterlim-def filtermap-sup by auto

lemma filterlim-sequentially-Suc:

  \((LIM x \text{sequentially}. f (\text{Suc} x) : F) \iff (LIM x \text{sequentially}. f x : F)\)

  unfolding filterlim-iff by (subst eventually-sequentially-Suc simp)

lemma filterlim-Suc: filterlim \(\text{Suc}\) sequentially sequentially

  by (simp add: filterlim-iff eventually-sequentially)

lemma filterlim-If:

  \(LIM x \text{inf} F \ (\text{principal} \{x. P x\}). f x : G \Rightarrow\)

  \(LIM x \text{inf} F \ (\text{principal} \{x. \neg P x\}). g x : G \Rightarrow\)

  \(LIM x F. \text{if} P x \text{ then} f x \text{ else} g x : G\)

  unfolding filterlim-iff eventually-inf-principal by (auto simp: eventually-conj-iff)

lemma filterlim-Pair:

  \(LIM x F. f x : G \Rightarrow LIM x F. g x : H \Rightarrow LIM x F. (f x, g x) : G \times F H\)

  unfolding filterlim-def

  by (rule order-trans[OF filtermap-Pair prod-filter-mono])

88.7 Limits to at-top and at-bot

lemma filterlim-at-top:

  fixes f :: \('a\ \Rightarrow \('b::linorder)
shows \((LIM x F. f x :> at-top) \iff (\forall Z. \text{eventually } (\lambda x. Z \leq f x) F)\)

by (auto simp: filterlim-iff eventually-at-top-linorder elim!: eventually-mono)

**lemma** filterlim-at-top-mono:
\[
LIM x F. f x :> at-top \implies \text{eventually } (\lambda x. f x \leq (g x :a::linorder)) F \implies LIM x F. g x :> at-top
\]

by (auto simp: filterlim-at-top elim: eventually-elim2 intro: order-trans)

**lemma** filterlim-at-top-dense:

fixes \(f::'a \Rightarrow ('b::unbounded-dense-linorder)

shows \((LIM x F. f x :> at-top) \iff (\forall Z. \text{eventually } (\lambda x. Z < f x) F)\)

by (metis eventually- mono unfolding at-top-sub[of c] filterlim-INF)

**lemma** filterlim-at-top-ge:

fixes \(f::'a::linorder \Rightarrow 'b::linorder\)

assumes \(\text{mono: } \lambda x. y. Q x \Rightarrow Q y \Rightarrow x \leq y \Rightarrow f x \leq f y\)

assumes \(\text{bij: } \lambda x. P x \Rightarrow f (g x) = x \\lambda x. P x \Rightarrow Q (g x)\)

assumes \(\text{Q: } \text{eventually } Q \text{ at-top}\)

assumes \(\text{P: } \text{eventually } P \text{ at-top}\)

shows \(\text{filterlim } f \text{ at-top at-top}\)

**proof**

from \(P\) obtain \(x\) where \(\lambda y. x \leq y \Rightarrow P y\)

unfolding eventually-at-top-linorder by auto

show \(?thesis\)

proof (intro filterlim-at-top-ge[THEN iffD2] allI impI)

fix \(z\) assume \(x \leq z\)

with \(x\) have \(P z\) by auto

have \(\text{eventually } (\lambda x. g z \leq x) \text{ at-top}\)

by (rule eventually-ge-at-top)

with \(Q\) show \(\text{eventually } (\lambda x. z \leq f x) \text{ at-top}\)

by eventually-elim (metis mono bij \(P z\))

qed

qed

**lemma** filterlim-at-top-gt:

fixes \(f::'a \Rightarrow ('b::unbounded-dense-linorder)\) and \(c::'b\)

shows \((LIM x F. f x :> at-top) \iff (\forall Z>c. \text{eventually } (\lambda x. Z \leq f x) F)\)

by (metis filterlim-at-top order-less-le-trans gt-ex filterlim-at-top-ge)

**lemma** filterlim-at-bot:

fixes \(f::'a \Rightarrow ('b::linorder)\)

shows \((LIM x F. f x :< at-bot) \iff (\forall Z. \text{eventually } (\lambda x. f x \leq Z) F)\)

by (auto simp: filterlim-iff eventually-at-bot-linorder elim!: eventually-mono)
lemma filterlim-at-bot-dense:
  fixes f :: 'a ⇒ ('b::{dense-linorder, no-bot})
  shows (LIM x F. f x :: at-bot) ⟷ (∀ Z. eventually (λ x. f x < Z) F)
proof (auto simp add: filterlim-at-bot[of f F])
fix Z :: 'b
  from lt-ex [of Z] obtain Z' where 1: Z' < Z ..
assume ∀ Z. eventually (λ x. f x ≤ Z) F
hence eventually (λ x. f x ≤ Z') F by auto
thus eventually (λ x. f x < Z) F
  apply (rule eventually-mono)
  using 1 by auto
next
fix Z :: 'b
  show ∀ Z. eventually (λ x. f x < Z) F ⟷ eventually (λ x. Z ≥ f x) F
  unfolding filterlim-at-bot
  using 1 unfolding eventually-mono, auto simp add: less-imp-le
qed

lemma filterlim-at-bot-le:
  fixes f :: 'a ⇒ ('b::linorder) and c :: 'b
  shows (LIM x F. f x :: at-bot) ⟷ (∀ Z ≤ c. eventually (λ x. Z ≥ f x) F)
unfolding filterlim-at-bot
proof safe
fix Z assume *: ∀ Z ≤ c. eventually (λ x. Z ≥ f x) F
with * THEN spec [of - Z], erule eventually-mono
  by (auto elim!: eventually-mono)
qed simp

lemma filterlim-at-bot-lt:
  fixes f :: 'a ⇒ ('b::unbounded-dense-linorder) and c :: 'b
  shows (LIM x F. f x :: at-bot) ⟷ (∀ Z < c. eventually (λ x. Z ≥ f x) F)
by (metis filterlim-at-bot filterlim-at-bot-le lt-ex order-le-less-trans)

lemma filterlim-finite-subsets-at-top:
  filterlim f (finite-subsets-at-top A) F ⟷
  (∀ X. finite X ∧ X ⊆ A → eventually (λ y. finite (f y) ∧ X ⊆ f y ∧ f y ⊆ A) F)
(is ?lhs = ?rhs)
proof
  assume ?lhs
  thus ?rhs
proof (safe, goal-cases)
  case (1 X)
  hence *: (∀ f x in F. P (f x)) if eventually P (finite-subsets-at-top A) for P
    using that by (auto simp: filterlim-def le-filter-def eventually-filtermap)
  have ∀ f Y in finite-subsets-at-top A. finite Y ∧ X ⊆ Y ∧ Y ⊆ A
    using / unfolding eventually-finite-subsets-at-top by force
  thus ?case by (intro *) auto
qed
THEORY “Filter”

next
assume rhs: ?rhs
show ?lhs unfolding filterlim-def le-filter-def eventually-finite-subsets-at-top
proof (safe, goal-cases)
case (1 P X)
with rhs have \( \forall F \ y in F. \ finite \ (f y) \land X \subseteq f y \land f y \subseteq A \) by auto
thus eventually P (filtermap f F) unfolding eventually-filtermap
  by eventually-elim (insert 1, auto)
qed
qed

lemma filterlim-atMost-at-top:
  filterlim (\( \lambda n. \{..n\} \)) (finite-subsets-at-top (UNIV :: nat set)) at-top
proof (safe, goal-cases)
case (1 X)
then obtain n where n: X \subseteq \{..n\} by (auto simp: finite-nat-set-iff-bounded-le)
show ?case using eventually-ge-at-top[of n]
  by eventually-elim (insert n, auto)
qed

lemma filterlim-lessThan-at-top:
  filterlim (\( \lambda n. \{..<n\} \)) (finite-subsets-at-top (UNIV :: nat set)) at-top
proof (safe, goal-cases)
case (1 X)
then obtain n where n: X \subseteq \{..<n\} by (auto simp: finite-nat-set-iff-bounded)
show ?case using eventually-ge-at-top[of n]
  by eventually-elim (insert n, auto)
qed

88.8 Setup 'a filter for lifting and transfer
lemma filtermap-id [simp, id-simps]: filtermap id = id
by (simp add: fun_eq_iff id_def filtermap-ident)

lemma filtermap-id' [simp]: filtermap (\( \lambda x. \)) = (\( \lambda F. \) F)
using filtermap-id unfolding id-def.

context includes lifting-syntax
begin

definition map-filter-on :: 'a set ⇒ ('a ⇒ 'b) ⇒ 'a filter ⇒ 'b filter where
  map-filter-on X f F = Abs-filter (\( \lambda P. \) eventually (\( \lambda x. \ P \ (f x) \land x \in X \) F))

lemma is-filter-map-filter-on:
  is-filter (\( \lambda P. \) \( \forall F \ x in F. \ P \ (f x) \land x \in X \) \( \iff \) eventually (\( \lambda x. \ x \in X \) F)
proof (rule iffI; unfold-locales)
  show \( \forall F \ x in F. \ True \land x \in X \) if eventually (\( \lambda x. \ x \in X \) F) using that by
simp

  show \( \forall F \ x \in F. (P \ (f \ x) \land Q \ (f \ x)) \land x \in X \) \( \forall F \ x \in F. P \ (f \ x) \land x \in X \) \( \forall F \ x \in F. Q \ (f \ x) \land x \in X \) \( \forall F \ x \in F. Q \ (f \ x) \land x \in X \) for \( P \ Q \)
  using eventually-conj[of that] by(auto simp add: conj-AC cong: conj-cong)

show \( \forall F \ x \in F. Q \ (f \ x) \land x \in X \) \( \forall x. P x \to Q x \forall F \ x \in F. P \ (f \ x) \land x \in X \) \( \forall F \ x \in F. Q \ (f \ x) \land x \in X \) for \( P \ Q \)
  using that(2) by(rule eventually-mono)(use that(1) in auto)

show eventually \((\lambda x. x \in X) \to \text{is-filter} \ (\lambda P. \forall F \ x \in F. P \ (f \ x) \land x \in X)\)
  using is-filter.\text{True}(OF that) by simp

qed

lemma eventually-map-filter-on: \( \text{eventually} \ P \ (\text{map-filter-on} X f F) = (\forall F \ x \in F. P \ (f \ x) \land x \in X) \)
  if eventually \((\lambda x. x \in X) \to F\)
  by(simp add: \text{is-filter-map-filter-on} map-filter-on-def eventually-Abs-filter that)

lemma map-filter-on-UNIV: \( \text{map-filter-on} \ \text{UNIV} = \text{filtermap} \)
  by(simp add: map-filter-on-def filtermap-def fun-eq-iff)

lemma map-filter-on-comp: \( \text{map-filter-on} X f \ (\text{map-filter-on} Y g F) = \text{map-filter-on} Y \ (f \circ g) F \)
  if \( g \cdot Y \subseteq X \) and eventually \((\lambda x. x \in Y) \to F\)

unfolding map-filter-on-def using that(1)
  by(auto simp add: eventually-Abs-filter that(2) \text{is-filter-map-filter-on} intro!: arg-cong[where \( f=\text{Abs-filter} \)] arg-cong2[where \( f=\text{eventually} \))]

inductive rel-filter :: \( ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a \to ('b \Rightarrow \text{bool}) \Rightarrow \text{bool} \to R \ F \ G \)
where
  rel-filter \( R \ F \ G \) if eventually \((\text{case-prod} R) \ Z \text{ map-filter-on} \ {(x, y). R x y} \ \text{fst} Z = F \ \text{map-filter-on} \ {(x, y). R x y} \ \text{snd} Z = G \)

lemma rel-filter-eq [relator-eq]: \( \text{rel-filter} \ (=} \ (=) \)
proof(intro ext iffI)+

  show \( F = G \) if rel-filter \( (=} \) \( F \) \( G \) for \( F \) \( G \) using that
    by cases(clarsimp simp add: filter-eq-iff eventually-map-filter-on split-def cong: rev-conj-cong)

  show rel-filter \( (=} \) \( F \) \( G \) if \( F = G \) for \( F \) \( G \) unfolding \( F = G \)
proof

  let \( ?Z = \text{map-filter-on} \ \text{UNIV} \ (\lambda x. \ (x, x)) \ G \)
  have [simp]: range \((\lambda x. \ (x, x))\) \( \subseteq \{(x, y). x = y\}\) by auto

  show \( \text{map-filter-on} \ {(x, y). x = y} \ \text{fst} \ ?Z = G \) and \( \text{map-filter-on} \ {(x, y). x = y} \ \text{snd} \ ?Z = G \)
    by(simp-all add: \text{map-filter-on-comp})(simp-all add: \text{map-filter-on-UNIV o-def})

  show \( \forall x. (x, y) \in ?Z \) \( x = y \) by(simp add: eventually-map-filter-on)

  qed

qed

lemma rel-filter-mono [relator-mono]: \( \text{rel-filter} A \leq \text{rel-filter} B \) if le: \( A \leq B \)
proof(clarify elim!: \text{rel-filter.cases})
show \( \text{rel-filter} \ B \ (\text{map-filter-on} \ \{(x, y). A \ x \ y\} \ \text{fst} \ Z) \ \text{(map-filter-on} \ \{(x, y). A \ x \ y\} \ \text{snd} \ Z) \)
\[(\text{is rel-filter} - \ ?X \ ?Y) \ if \ \forall \_F \ (x, y) \ in \ Z.\ A \ x \ y \ \text{for} \ Z\]
proof
let \( ?Z = \text{map-filter-on} \ \{(x, y). A \ x \ y\} \ \text{id} \ Z \)
show \( \forall \_F \ (x, y) \ in \ ?Z. B \ x \ y \ \text{using} \ \text{le that} \)
by(\text{simp add: eventually-map-filter-on le-fun-def split-def conj-commute cong: conj-cong})
have [simp]: \( \{(x, y). A \ x \ y\} \subseteq \{(x, y). B \ x \ y\} \ \text{using} \ \text{le by auto} \)
show map-filter-on \( \{(x, y). B \ x \ y\} \ \text{fst} \ ?Z = \ ?X \ \text{map-filter-on} \ \{(x, y). B \ x \ y\} \ \text{snd} \ ?Z = \ ?Y \)
\[\text{using le that by(simp-all add: le-fun-def map-filter-on-comp)}\]
qed
qed

lemma rel-filter-conversep: rel-filter \( A^{\sim} \ \sim \) = (rel-filter \( A \))^{\sim} \sim
proof(safe intro!: ext elim!: rel-filter.cases)
show \( \ast: \text{rel-filter} \ A \ (\text{map-filter-on} \ \{(x, y). A^{\sim} \ \sim \ x \ y\} \ \text{snd} \ Z) \ \text{map-filter-on} \ \{(x, y). A^{\sim} \ \sim \ x \ y\} \ \text{fst} \ Z \)
\[(\text{is rel-filter} - \ ?X \ ?Y) \ if \ \forall \_F \ (x, y) \ in \ Z. A^{\sim} \ \sim \ x \ y \ \text{for} \ A \ Z\]
proof
let \( ?Z = \text{map-filter-on} \ \{(x, y). A \ y \ x\} \ \text{prod.swap} \ Z \)
show \( \forall \_F \ (x, y) \ in \ ?Z. A \ x \ y \ \text{using that by(simp add: eventually-map-filter-on)} \)
have [simp]: \( \text{prod.swap} \ \{(x, y). A \ y \ x\} \subseteq \{(x, y). A \ x \ y\} \ \text{by auto} \)
show map-filter-on \( \{(x, y). A \ x \ y\} \ \text{fst} \ ?Z = \ ?X \ \text{map-filter-on} \ \{(x, y). A \ x \ y\} \ \text{snd} \ ?Z = \ ?Y \)
\[\text{using that by(simp-all add: map-filter-on-comp o-def)}\]
qed
show rel-filter \( A^{\sim} \ \sim \) (map-filter-on \ \{(x, y). A \ x \ y\} \ \text{snd} \ Z) \ \text{map-filter-on} \ \{(x, y). A \ x \ y\} \ \text{fst} \ Z \)
if \( \forall \_F \ (x, y) \ in \ Z. A \ x \ y \ \text{for} \ Z \ \text{using} \ \ast[\text{of} \ A^{\sim} \ \sim \ Z] \ \text{that by simp} \)
qed

lemma rel-filter-distr [relator-distr]:
\( \text{rel-filter} \ A \ \text{OO} \ \text{rel-filter} \ B = \ \text{rel-filter} \ (A \ \text{OO} \ B) \)
proof(safe intro!: ext elim!: rel-filter.cases)
let \( ?AB = \{(x, y). (A \ \text{OO} \ B) \ x \ y\} \)
show (\text{rel-filter} \ A \ \text{OO} \ \text{rel-filter} \ B)
\( \ \text{map-filter-on} \ \{(x, y). (A \ \text{OO} \ B) \ x \ y\} \ \text{fst} \ Z) \ \text{map-filter-on} \ \{(x, y). (A \ \text{OO} \ B) \ x \ y\} \ \text{snd} \ Z \)
\[(\text{is \ (- \ OO \ -) \ ?F \ ?H) \ if \ \forall \_F \ (x, y) \ in \ Z. (A \ \text{OO} \ B) \ x \ y \ \text{for} \ Z\]
proof
let \( ?G = \text{map-filter-on} \ ?AB \ (\lambda(x, y). \ \text{SOME} \ z. A \ x \ z \ \land \ B \ z \ y) \ Z \)
show rel-filter \( A \ ?F \ ?G \)
proof
let \( ?Z = \text{map-filter-on} \ ?AB \ (\lambda(x, y). (x, \ \text{SOME} \ z. A \ x \ z \ \land \ B \ z \ y)) \ Z \)
show \( \forall \_F \ (x, y) \ in \ ?Z. A \ x \ y \ \text{using that} \)
by(\text{auto simp add: eventually-map-filter-on split-def elim!: eventually-mono intro: some12})
THEORY “Filter”

have [simp]: (λp. (fst p, SOME z. A (fst p) z ∧ B z (snd p)))) :: {p. (A OO B) (fst p) (snd p)} ⊆ {p. A (fst p) (snd p)} by(auto intro: someI2)
using that by(simp-all add: map-filter-on-comp split-def o-def)
qed
show rel-filter B ?G ?H
proof
let ?Z = map-filter-on ?AB (λ(x, y). (SOME z. A x z ∧ B z y, y)) Z
show ∀ F (x, y) in ?Z. B x y using that
by(auto simp add: eventually-map-filter-on-split-def elim!: eventually-monono intro: someI2)
have [simp]: (λp. (SOME z. A (fst p) z ∧ B z (snd p), snd p)) :: {p. (A OO B) (fst p) (snd p)} ⊆ {p. B (fst p) (snd p)} by(auto intro: someI2)
show map-filter-on {(x, y). B x y} fst ?Z = ?G map-filter-on {(x, y). B x y} snd ?Z = ?H
using that by(simp-all add: map-filter-on-comp split-def o-def)
qed
qed

fix F G
assume F: ∀ F (x, y) in F. A x y and G: ∀ F (x, y) in G. B x y
and eq: map-filter-on {(x, y). B x y} fst G = map-filter-on {(x, y). A x y} snd F (is ?Y2 = ?Y1)
let ?X = map-filter-on {(x, y). A x y} fst F
and ?Z = (map-filter-on {(x, y). B x y} snd G)
have step: ∃ P ≤ P ∀ Q ≤ Q. eventually P F ∧ eventually Q G ∧ {y. ∃ x. P′ (x, y)} (x, y) = {y. ∃ z. Q′ (y, z)}
if P: eventually P F and Q: eventually Q G for P Q
proof
let ?P = λ(x, y). P (x, y) ∧ A x y and ?Q = λ(y, z). Q (y, z) ∧ B y z
define P′ where P′ ≡ λ(x, y). ?P (x, y) ∧ (∃ z. ?Q (y, z))
define Q′ where Q′ ≡ λ(y, z). ?Q (y, z) ∧ (∃ x. ?P (x, y))
have P′ ≤ P Q′ ≤ Q {y. ∃ x. P′ (x, y)} = {y. ∃ z. Q′ (y, z)}
by(auto simp add: P′-def Q′-def)

moreover
from P Q F G have P′: eventually ?P F and Q′: eventually ?Q G
by(simp-all add: eventually-conj-iff split-def)
from P′ F have ∀ F y in ?Y1. ∃ x. P (x, y) ∧ A x y
by(auto simp add: eventually-map-filter-on elim!: eventually-monono)
from this-folded eq obtain Q′ where Q′: eventually Q′ G
and Q′P: {y. ∃ z. Q′ (y, z)} ⊆ {y. ∃ x. ?P (x, y)}
using G by(fastforce simp add: eventually-map-filter-on)
have eventually (inf Q′ ?Q) G using Q′ Q′ by(auto intro: eventually-conj simp add: inf-fin-def)
then have eventually Q′ G using Q′P by(auto elim!: eventually-monono simp add: Q′-def)
moreover
from Q′ G have ∀ F y in ?Y2. ∃ z. Q (y, z) ∧ B y z
by (auto simp add: eventually-map-filter-on elim!: eventually-mono)
from this[unfolded eq] obtain $P''$ where $P'': \text{eventually } P'' \vdash F$
and $P''(?Y): \{ y. \exists x. P''(x, y) \} \subseteq \{ y. \exists z. ?Q (y, z) \}$
using $F$ by (fastforce simp add: eventually-map-filter-on)
have eventually (inf $P'' ?P$) $F$ using $P'' P' by (auto intro: eventually-conj simp add: inf-fun-def)
then have eventually $P' F$ using $P''?Q$ by (auto elim!: eventually-mono simp add: $P'\text{-def}$)
ultimately show ?thesis by blast
qed

show rel-filter $(A OO B) ?X ?Z$
proof
let $?Y = \lambda Y. \exists X Z. \text{eventually } X ?X \land \text{eventually } Z ?Z \land (\lambda(x, z). X x \land Z z \land (A OO B) x z) \leq Y$
have $Y$: is-filter $?Y$
proof
show $?Y (\lambda-. \text{True})$ by (auto simp add: le-fun-def intro: eventually-True)
show $?Y (\lambda x. P x \land Q x)$ if $?Y P ?Y Q for $P Q$ using that
apply clarify
apply (intro exI conjI; (elim eventually-rev-mp; fold imp-conjL; intro always-eventually allI; rule imp-refl)?)
apply auto
done
show $?Y Q$ if $?Y P \forall x. P x \longrightarrow Q x$ for $P Q$ using that by blast
qed

define $Y$ where $Y = \text{Abs-filter } ?Y$
have eventually-$Y$: eventually $P Y \iff ?Y P$ for $P$
using eventually-Abs-filter[OF $Y$, of $P$] by (simp add: $Y\text{-def}$)
show $YY: \forall F (x, y) \in Y. (A OO B) x y$ using $F G$
by (auto simp add: eventually-$Y$ eventually-map-filter-on eventually-conj-iff intro!: eventually-True)
have $?Y (\lambda(x, z). P x \land (A OO B) x z) \iff \forall F (x, y)$ in $F. P x \land A x y$)
(is $\text{?lhs = ?rhs}$ for $P$)
proof
show $\text{?lhs}$ if $\text{?rhs}$ using $G F$ that
by (auto 4 3 intro: exI[where $z=\lambda-$. True] simp add: eventually-map-filter-on split-def)
assume $\text{?lhs}$
then obtain $X Z$ where $ \forall F (x, y)$ in $F. X x \land A x y$
and $\forall F (x, y)$ in $G. Z y \land B x y$
and $(\lambda(x, z). X x \land Z z \land (A OO B) x z) \leq (\lambda(x, z). P x \land (A OO B) x z)$
using $F G$ by (auto simp add: eventually-map-filter-on split-def)
from step[OF this(1, 2)] this(3)
show $\text{?rhs}$ by (clarisimp elim!: eventually-rev-mp simp add: le-fun-def)(fastforce intro: always-eventually)
qed
then show map-filter-on $?AB \text{ fst } Y = ?X$
by (simp add: filter-eq-iff YY eventually-map-filter-on)(simp add: eventually-$Y$)
eventually-map-filter-on \( F; \) simp add: split-def)

have \(?Y \bin (\lambda(x, z). P z \land (A OO B) x z) \leftrightarrow (\forall_F(x, y) \in G. P y \land B x y)\) (is \(?lhs = ?rhs\) for \(P\))

proof
  show \(?lhs\) if \(?rhs\) using \(G F\) that 
  by(auto \(4\ 3\ intro: ext[where x=\lambda-. True]\) simp add: eventually-map-filter-on split-def) 
  assume \(?lhs\)
  then obtain \(X Z\) where \(\forall_F(x, y) \in F. X x \land A x y\)
  and \(\forall_F(x, y) \in G. Z y \land B x y\)
  and \((\lambda(x, z). X x \land Z z \land (A OO B) x z) \leq (\lambda(x, z). P z \land (A OO B) x z)\)
  using \(F G\) by(auto simp add: eventually-map-filter-on split-def) 
  from step[OF this(\(1\ 2\)] this(\(3\))
  show \(?rhs\) by(clarsimp elim!: eventually-rev-mp simp add: le-fun-def)(fastforce intro: always-eventually)
  qed
  then show map-filter-on \(?AB\) snd \(Y = ?Z\)
  qed
  qed

lemma filtermap-parametric: \(((A ===> B) ===> rel-filter A ===> rel-filter B) filtermap filtermap\)
proof(intro rel-funI; erule rel-filter.cases; hypsubst)
  fix \(f\ g\ Z\)
  assume \(f g\): \((A ===> B) f g\) and \(Z: \forall_F(x, y) \in Z. A x y\)
  have rel-filter \(B\) (map-filter-on \(\{x, y\}; A x y\) \((f o fst) Z\) (map-filter-on \(\{x, y\}\)
  is rel-filter - \(?F ?G\))
  proof
    let \(?Z = map-filter-on \(\{x, y\}; A x y\) (map-prod f g) Z\)
    show \(\forall_F(x, y) \in ?Z. B x y\) using \(f g\ Z\)
    by(auto simp add: eventually-map-filter-on split-def elim!: eventually-mono rel-funD)
    have \(\{simp\}: map-prod f g : \{p. A (fst p) (snd p)\} \subseteq \{p. B (fst p) (snd p)\}\)
    using \(f g\ by(auto dest: rel-funD)\)
    show map-filter-on \(\{x, y\}; B x y\) \(fst ?Z = ?F map-filter-on \(\{x, y\}; B x y\)\)
    snd \(?Z = ?G\)
    using \(Z\) by(auto simp add: map-filter-on-comp split-def)
    qed
  thus rel-filter \(B\) (filtermap f (map-filter-on \(\{x, y\}; A x y\) \(fst Z\)) (filtermap g (map-filter-on \(\{x, y\}; A x y\) \(snd Z\)))
  using \(Z\) by(simp add: map-filter-on-UNIV[symmetric] map-filter-on-comp)
  qed

lemma rel-filter-Grp: rel-filter \((Grp UNIV f) = Grp UNIV (filtermap f)\)
proof(intro antisym predicate2I; (elim GrpE; hypsubst) ?; rule GrpI[OF - UNIV-I])
fix $F, G$
assume rel-filter $(\text{Grp UNIV } f) F G$
hence rel-filter $(=) (\text{filtermap } f F) (\text{filtermap } \text{id } G)$
  by (rule filtermap-parametric [THEN rel-funD, THEN rel-funD, rotated])(simp add: Grp-def rel-fun-def)
thus filtermap $f$ $F = G$ by (simp add: rel-filter-eq)

next
fix $F :: 'a$ filter
have rel-filter $(=) F F$ by (simp add: rel-filter-eq)
hence rel-filter $(\text{Grp UNIV } f) (\text{filtermap } \text{id } F) (\text{filtermap } f F)$
  by (rule filtermap-parametric [THEN rel-funD, THEN rel-funD, rotated])(simp add: Grp-def rel-fun-def)
thus rel-filter $(\text{Grp UNIV } f) F (\text{filtermap } f F)$ by simp
qed

lemma Quotient-filter [quot-map]:
  Quotient $R$ $\text{Abs}$ $\text{Rep}$ $T$ $\Longrightarrow$ Quotient $(\text{rel-filter } R) (\text{filtermap } \text{Abs}) (\text{filtermap } \text{Rep}) (\text{rel-filter } T)$
unfolding Quotient-alt-def5 rel-filter-eq [symmetric] rel-filter-Grp [symmetric]

lemma left-total-rel-filter [transfer-rule]: left-total $A$ $\Longrightarrow$ left-total $(\text{rel-filter } A)$
unfolding left-total-alt-def rel-filter-eq [symmetric] rel-filter-conversep [symmetric]
rel-filter-distr
by (rule rel-filter-mono)

lemma right-total-rel-filter [transfer-rule]: right-total $A$ $\Longrightarrow$ right-total $(\text{rel-filter } A)$
using left-total-rel-filter [of $A^{-1}$]
by (simp add: rel-filter-conversep)

lemma bi-total-rel-filter [transfer-rule]: bi-total $A$ $\Longrightarrow$ bi-total $(\text{rel-filter } A)$
unfolding bi-total-alt-def
by (simp add: left-total-rel-filter right-total-rel-filter)

lemma left-unique-rel-filter [transfer-rule]: left-unique $A$ $\Longrightarrow$ left-unique $(\text{rel-filter } A)$
unfolding left-unique-alt-def rel-filter-eq [symmetric] rel-filter-conversep [symmetric]
rel-filter-distr
by (rule rel-filter-mono)

lemma right-unique-rel-filter [transfer-rule]:
  right-unique $A$ $\Longrightarrow$ right-unique $(\text{rel-filter } A)$
using left-unique-rel-filter [of $A^{-1}$]
by (simp add: rel-filter-conversep)

lemma bi-unique-rel-filter [transfer-rule]: bi-unique $A$ $\Longrightarrow$ bi-unique $(\text{rel-filter } A)$
by (simp add: bi-unique-alt-def left-unique-rel-filter right-unique-rel-filter)

lemma eventually-parametric [transfer-rule]:
  $(\text{(} A \Longrightarrow (=) \text{)} \Longrightarrow \text{rel-filter } A \Longrightarrow (=))$ $\text{eventually eventually}$
by (auto 4 4 intro: rel-funI elim!: rel-filter.cases simp add: eventually-map-filter-on)
dest: rel-funD intro: always-eventually elim!: eventually-rev-mp)

lemma frequently-parametric [transfer-rule]: \((A \implies (\sim)) \implies \sim\) frequently frequently
  unfolding frequently-def[abs-def] by transfer-prover

lemma is-filter-parametric [transfer-rule]:
  assumes [transfer-rule]: bi-total A
  assumes [transfer-rule]: bi-unique A
  shows ((\((A \implies (\sim)) \implies (\sim)\)) \implies (\sim)) is-filter is-filter
  unfolding is-filter-def by transfer-prover

lemma top-filter-parametric [transfer-rule]: rel-filter A top top if bi-total A
proof
  let ?Z = principal \{\(x, y\). A x y\}
  show \(\forall F \ (x, y) \in ?Z. A x y\) by(simp add: eventually-principal)
  show map-filter-on \{(x, y). A x y\} \(\text{fst} \ ?Z = \text{top map-filter-on} \ \{(x, y). A x y\}\)
  \(\text{snd} \ ?Z = \text{top}\)
  using (that by(auto simp add: filter-eq-iff eventually-map-filter-on eventually-principal
  bi-total-def))
  qed

lemma bot-filter-parametric [transfer-rule]: rel-filter A bot bot
proof
  show \(\forall F \ (x, y) \in \text{bot}. A x y\) by simp
  show map-filter-on \{(x, y). A x y\} \(\text{fst} \ \text{bot} = \text{bot map-filter-on} \ \{(x, y). A x y\}\)
  \(\text{snd} \ \text{bot} = \text{bot}\)
  by(simp-all add: filter-eq-iff eventually-map-filter-on)
  qed

lemma principal-parametric [transfer-rule]: \((\text{rel-set} A \implies \text{rel-filter} A)\) principal
proof(rule rel-funI rel-filter.intro)+
  fix S S'
  assume *: \text{rel-set} A \ S \ S'
  define \text{SS'} where \text{SS'} = S \times S' \cap \{(x, y). A x y\}
  have \text{SS'}: \text{SS'} \subseteq \{(x, y). A x y\} and [simp]: \text{S = \text{fst} \ ' SS'} \ S' = \text{snd} \ ' SS'
  using * by(auto 4 3 dest: rel-setD1 rel-setD2 intro: rev-image-eqI simp add: SS'-def)
  let ?Z = principal \text{SS'}
  show \(\forall F \ (x, y) \in ?Z. A x y\) using \text{SS'} by(auto simp add: eventually-principal)
  then show map-filter-on \{(x, y). A x y\} \(\text{fst} \ ?Z = \text{principal} S\)
  and map-filter-on \{(x, y). A x y\} \(\text{snd} \ ?Z = \text{principal} S'\)
  by(auto simp add: filter-eq-iff eventually-map-filter-on eventually-principal)
  qed

lemma sup-filter-parametric [transfer-rule]:
\((\text{rel-filter} A \implies \text{rel-filter} A \implies \text{rel-filter} A)\) sup sup
proof(intro rel-funI; elim rel-filter.cases; hypsubst)
THEORY "Filter"

lemma Sup-filter-parametric [transfer-rule]: (rel-set (rel-filter A) ==> rel-filter A) Sup Sup
  proof (rule rel-funI)
    fix S S'
    define SS' where SS' = S x S' \cap { (F, G). rel-filter A F G } 
    assume rel-set (rel-filter A) S S'
    then have SS': SS' \subseteq { (F, G). rel-filter A F G } and [simp]; S = fst ' SS' S'
    = snd ' SS'
      by (auto 4 3 dest: rel-setD1 rel-setD2 intro: rev-image-eqI simp add: SS'-def)
from SS' obtain Z where Z: \forall F. (F, G) \in SS' ==> 
  \forall F. (x, y) in Z F G. (A x y) 
  id F = map-filter-on { (x, y). A x y } \asmk (Z F G) 
  id G = map-filter-on { (x, y). A x y } \asmk (Z F G)
  unfolding rel-filter.simps by (atomize-elim (rule choice allI)) +; auto
have id: eventually P F = eventually P (id F) eventually Q G = eventually Q (id G)
  if (F, G) \in SS' for P Q F G by simp-all
show rel-filter A (Sup S) (Sup S')
  proof
    let \asmk Z = \bigsqcup (F, G) \in SS'. Z F G
    show \asmk \forall F. (x, y) in \asmk Z F G. A x y using Z by (auto simp add: eventually-Sup)
    show map-filter-on { (x, y). A x y } \asmk \forall F. (x, y) in \asmk Z F G. A x y \asmk \forall F. (x, y) in \asmk Z F G. A x y
      unfolding filter-eq-iff
        by (auto 4 4 simp add: id eventually-Sup eventually-map-filter-on *[simplified eventually-Sup] simp del: id-apply dest: Z)
  qed
qed

case
  fixes A :: 'a => 'b => bool
  assumes [transfer-rule]: bi-unique A
begin

lemma le-filter-parametric [transfer-rule]:
(rel-filter A ====> rel-filter A ====> (=)) (≤) (≤)
unfolding le-filter-def[abs-def] by transfer-prover

lemma less-filter-parametric [transfer-rule]:
(rel-filter A ====> rel-filter A ====> (=)) (<) (<)
unfolding less-filter-def[abs-def] by transfer-prover

context
  assumes [transfer-rule]: bi-total A
begin

lemma Inf-filter-parametric [transfer-rule]:
(rel-set (rel-filter A) ====> rel-filter A) Inf Inf
unfolding Inf-filter-def[abs-def] by transfer-prover

lemma inf-filter-parametric [transfer-rule]:
(rel-filter A (Inf {F, G}) (Inf {F', G'})) by transfer-prover
thus rel-filter A (inf F G) (inf F' G') by simp
qed
end
end
end

context
  includes lifting-syntax
begin

lemma prod-filter-parametric [transfer-rule]:
(rel-filter R ====> rel-filter S ====> rel-filter (rel-prod R S)) prod-filter prod-filter
proof(intro rel-funI; elim rel-filter; cases; hypsubst)
fix F G
assume F: ∀ F (x, y) in F. R x y and G: ∀ F (x, y) in G. S x y
show rel-filter (rel-prod R S)
  (map-filter-on {(x, y), R x y} fst F ×F map-filter-on {(x, y), S x y} fst G)
  (map-filter-on {(x, y), R x y} snd F ×F map-filter-on {(x, y), S x y} snd G)
  (is rel-filter ?RS ?F ?G)
proof
  let ?Z = filtermap (λ((a, b), (a', b')). ((a, a'), (b, b'))) (prod-filter F G)
  show ∀ F (x, y) in ?Z. rel-prod R S x y using F G
    by(auto simp add: eventually-filtermap split-beta eventually-prod-filter)
  show map-filter-on {(x, y), ?RS x y} fst ?Z = ?F
    using F G
apply(clarsimp simp add: filter-eq-iff eventually-map-filter-on *)
apply(simp add: eventually-filtermap split-beta eventually-prod-filter)
apply(subst eventually-map-filter-on; simp+)
apply(rule iffI; clarsimp)
subgoal for P P' P''
  apply(rule exI[where x=λa. ∃ b. P' (a, b) ∧ R a b]; rule conjI)
  subgoal by(fastforce elim: eventually-rev-mp eventually-mono)
  done
subgoal by fastforce
done
show map-filter-on {(x, y). ?RS x y} snd ?Z = ?G
  using F G
apply(clarsimp simp add: filter-eq-iff eventually-map-filter-on *)
apply(simp add: eventually-filtermap split-beta eventually-prod-filter)
apply(subst eventually-map-filter-on; simp+)
apply(rule iffI; clarsimp)
subgoal for P P' P''
  apply(rule exI[where x=λb. ∃ a. P'' (a, b) ∧ S a b]; rule conjI)
  subgoal by(fastforce elim: eventually-rev-mp eventually-mono)
  done
subgoal by fastforce
done
qed
done

end

Code generation for filters

definition abstract-filter :: (unit ⇒ 'a filter) ⇒ 'a filter
  where [simp]: abstract-filter f = f ()

code-datatype principal abstract-filter

hide-const (open) abstract-filter

declare [[code drop: filterlim prod-filter filtermap eventually
  inf :: - filter ⇒ - sup :: - filter ⇒ - less-eq :: - filter ⇒ -
  Abs-filter]]

declare filterlim-principal [code]
declare principal-prod-principal [code]
declare filtermap-principal [code]
declare filtercomap-principal [code]
THEORY "Conditionally-Complete-Lattices"

declare eventually-principal [code]
declare inf-principal [code]
declare sup-principal [code]
declare principal-le-iff [code]

lemma Rep-filter-iff-eventually [simp, code]:
  Rep-filter F P ⟷ eventually P F
  by (simp add: eventually-def)

lemma bot-eq-principal-empty [code]:
  bot = principal {}  
  by simp

lemma top-eq-principal-UNIV [code]:
  top = principal UNIV  
  by simp

instantiation filter :: (equal) equal
begin

definition equal-filter :: 'a filter ⇒ 'a filter ⇒ bool
  where equal-filter F F' ⟷ F = F'

lemma equal-filter [code]:
  HOL.equal (principal A) (principal B) ⟷ A = B  
  by (simp add: equal-filter-def)

instance
  by standard (simp add: equal-filter-def)

end

end

89 Conditionally-complete Lattices

theory Conditionally-Complete-Lattices
imports Finite-Set Lattices-Big Set-Interval
begin

context preorder
begin

definition bdd-above A ⟷ (∃ M. ∀ x ∈ A. x ≤ M)
definition bdd-below A ⟷ (∃ m. ∀ x ∈ A. m ≤ x)

lemma bdd-aboveI [intro]: (∀ x. x ∈ A ⟹ x ≤ M) ⟹ bdd-above A  
  by (auto simp: bdd-above-def)
lemma bdd-below2 [intro]: (\( \forall x \in A \Rightarrow m \leq x \)) \( \Rightarrow \) bdd-below A
  by (auto simp: bdd-below-def)

lemma bdd-aboveI2: (\( \forall x \in A \Rightarrow f x \leq M \)) \( \Rightarrow \) bdd-above (f\'A)
  by force

lemma bdd-above-Int1 [simp, intro]: bdd-above A \( \Rightarrow \) bdd-above (A \( \cap \) B)
  using bdd-above-mono by auto

lemma bdd-above-Int2 [simp, intro]: bdd-above B \( \Rightarrow \) bdd-above (A \( \cap \) B)
  using bdd-above-mono by auto

lemma bdd-above-Ioo [simp, intro]: bdd-above \{ a <..< b \}
  by (auto simp add: bdd-above-def intro!: exI[of - b] less-imp-le)

lemma bdd-above-Ico [simp, intro]: bdd-above \{ a ..< b \}
  by (auto simp add: bdd-above-def intro!: exI[of - b] less-imp-le)

lemma bdd-above-Iio [simp, intro]: bdd-above \{..< b \}
  by (auto simp add: bdd-above-def intro: exI[of - b] less-imp-le)

lemma bdd-above-Ioc [simp, intro]: bdd-above \{ a <.. b \}
  by (auto simp add: bdd-above-def intro: exI[of - b] less-imp-le)

lemma bdd-above-Iic [simp, intro]: bdd-above \{.. b \}
  by (auto simp add: bdd-above-def intro: exI[of - b] less-imp-le)
by (auto simp add: bdd-above_def intro: exI[of - b] less_imp_le)

lemma bdd-below-Ioo [simp, intro]: bdd-below {a <..< b}
  by (auto simp add: bdd-below_def intro: exI[of - a] less_imp_le)

lemma bdd-below-Ioc [simp, intro]: bdd-below {a <.. b}
  by (auto simp add: bdd-below_def intro: exI[of - a] less_imp_le)

lemma bdd-below-Ioi [simp, intro]: bdd-below {a <..}
  by (auto simp add: bdd-below_def intro: exI[of - a] less_imp_le)

lemma bdd-below-Ico [simp, intro]: bdd-below {a ..< b}
  by (auto simp add: bdd-below_def intro: exI[of - a] less_imp_le)

lemma bdd-below-Icc [simp, intro]: bdd-below {a .. b}
  by (auto simp add: bdd-below_def intro: exI[of - a] less_imp_le)

end

lemma (in order-top) bdd-above-top [simp, intro!]: bdd-above A
  by (rule bdd-aboveI[of - top]) simp

lemma (in order-bot) bdd-above-bot [simp, intro!]: bdd-below A
  by (rule bdd-belowI[of - bot]) simp

lemma bdd-above-image-mono: mono f \Rightarrow bdd-above A \Rightarrow bdd-above (f' A)
  by (auto simp: bdd-above-def mono_def)

lemma bdd-below-image-mono: mono f \Rightarrow bdd-below A \Rightarrow bdd-below (f' A)
  by (auto simp: bdd-below-def mono_def)

lemma bdd-above-image-antimono: antimono f \Rightarrow bdd-below A \Rightarrow bdd-above (f' A)
  by (auto simp: bdd-above-def antimono_def)

lemma bdd-below-image-antimono: antimono f \Rightarrow bdd-below A \Rightarrow bdd-below (f' A)
  by (auto simp: bdd-below-def antimono_def)

lemma fixes X :: 'a::ordered-ab-group-add set
    shows bdd-above-uminus [simp]: bdd-above (uminus ' X) \leftrightarrow bdd-below X
    and bdd-below-uminus [simp]: bdd-below (uminus ' X) \leftrightarrow bdd-above X
    using bdd-above-image-antimono[of uminus X] bdd-below-image-antimono[of uminus uminus X]
    using bdd-below-image-antimono[of uminus X] bdd-above-image-antimono[of umi-
nus uminus\'X]
  by (auto simp: antimono-def image-image)

context lattice
begin

lemma bdd-above-insert [simp]: bdd-above (insert a A) = bdd-above A
  by (auto simp: bdd-above-def intro: le-supI1 sup-le1)

lemma bdd-below-insert [simp]: bdd-below (insert a A) = bdd-below A
  by (auto simp: bdd-below-def intro: le-infI1 inf-le1)

lemma bdd-finite [simp]:
  assumes finite A shows bdd-above-finite: bdd-above A and bdd-below-finite: bdd-below A
  using assms by (induct rule: finite-induct, auto)

lemma bdd-above-Un [simp]: bdd-above (A ∪ B) = (bdd-above A ∧ bdd-above B)
proof
  assume bdd-above (A ∪ B)
  thus bdd-above A ∧ bdd-above B unfolding bdd-above-def by auto
next
  assume bdd-above A ∧ bdd-above B
  then obtain a b where ∀x∈A. x ≤ a ∀x∈B. x ≤ b unfolding bdd-above-def
  by auto
  hence ∀x ∈ A ∪ B. x ≤ sup a b by (auto intro: Un-iff le-supI1 le-supI2)
  thus bdd-above (A ∪ B) unfolding bdd-above-def ..
qed

lemma bdd-below-Un [simp]: bdd-below (A ∪ B) = (bdd-below A ∧ bdd-below B)
proof
  assume bdd-below (A ∪ B)
  thus bdd-below A ∧ bdd-below B unfolding bdd-below-def by auto
next
  assume bdd-below A ∧ bdd-below B
  then obtain a b where ∀x∈A. a ≤ x ∀x∈B. b ≤ x unfolding bdd-below-def
  by auto
  hence ∀x ∈ A ∪ B. inf a b ≤ x by (auto intro: Un-iff le-infI1 le-infI2)
  thus bdd-below (A ∪ B) unfolding bdd-below-def ..
qed

lemma bdd-above-image-sup[simp]:
  bdd-above ((λx. sup (f x) (g x)) ' A) ←→ bdd-above (f'A) ∧ bdd-above (g'A)
  by (auto simp: bdd-above-def intro: le-supI1 le-supI2)

lemma bdd-below-image-inf[simp]:
  bdd-below ((λx. inf (f x) (g x)) ' A) ←→ bdd-below (f'A) ∧ bdd-below (g'A)
  by (auto simp: bdd-below-def intro: le-infI1 le-infI2)
THEORY “Conditionally-Complete-Lattices”

lemma bdd-below-UN[simp]: finite I \implies bdd-below (\bigcup i \in I. A i) = (\forall i \in I. bdd-below (A i))
by (induction I rule: finite.induct) auto

lemma bdd-above-UN[simp]: finite I \implies bdd-above (\bigcup i \in I. A i) = (\forall i \in I. bdd-above (A i))
by (induction I rule: finite.induct) auto

end

To avoid name classes with the complete-lattice-class we prefix \bigcup and \bigcap in theorem names with c.

class conditionally-complete-lattice = lattice + Sup + Inf +
  assumes cInf-lower: x \in X \implies bdd-below X \implies Inf X \leq x
  and cInf-greatest: X \neq {} \implies (\forall x. x \in X \implies z \leq x) \implies z \leq Inf X
  assumes cSup-upper: x \in X \implies bdd-above X \implies x \leq Sup X
  and cSup-least: X \neq {} \implies (\forall x. x \in X \implies x \leq z) \implies Sup X \leq z
begin

lemma cSup-upper2: x \in X \implies y \leq x \implies bdd-above X \implies y \leq Sup X
by (metis cSup-upper order-trans)

lemma cInf-lower2: x \in X \implies x \leq y \implies bdd-below X \implies Inf X \leq y
by (metis cInf-lower order-trans)

lemma cSup-mono: B \neq {} \implies bdd-above A \implies (\forall b. b \in B \implies \exists a \in A. b \leq a)
\implies Sup B \leq Sup A
by (metis cSup-least cSup-upper2)

lemma cInf-mono: B \neq {} \implies bdd-below A \implies (\forall b. b \in B \implies \exists a \in A. a \leq b)
\implies Inf A \leq Inf B
by (metis cInf-greatest cInf-lower2)

lemma cSup-subset-mono: A \neq {} \implies bdd-above B \implies A \subseteq B \implies Sup A \leq Sup B
by (metis cSup-least cSup-upper subsetD)

lemma cInf-superset-mono: A \neq {} \implies bdd-below B \implies A \subseteq B \implies Inf B \leq Inf A
by (metis cInf-greatest cInf-lower subsetD)

lemma cSup-eq-maximum: z \in X \implies (\forall x. x \in X \implies x \leq z) \implies Sup X = z
by (intro antisym cSup-upper[of z X] cSup-least[of X z]) auto

lemma cInf-eq-minimum: z \in X \implies (\forall x. x \in X \implies z \leq x) \implies Inf X = z
by (intro antisym cInf-lower[of z X] cInf-greatest[of X z]) auto

lemma cSup-le-iff: S \neq {} \implies bdd-above S \implies Sup S \leq a \iff (\forall x \in S. x \leq a)
by (metis order-trans cSup-upper cSup-least)
lemma le-cInf-iff: $S \neq \{\} \implies \text{bdd-below } S \implies a \leq \text{Inf } S \iff (\forall x \in S. \ a \leq x)$
  by (metis order-trans cInf-lower cInf-greatest)

lemma cSup-eq-non-empty:
  assumes 1: $X \neq \{\}$
  assumes 2: $\forall x. \ x \in X \implies x \leq a$
  assumes 3: $\forall y. \ (\forall x. \ x \in X \implies x \leq y) \implies a \leq y$
  shows $\text{Sup } X = a$
  by (intro 3 1 antisym cSup-least) (auto intro: 2 1 cSup-upper)

lemma cInf-eq-non-empty:
  assumes 1: $X \neq \{\}$
  assumes 2: $\forall x. \ x \in X = \implies a \leq x$
  assumes 3: $\forall y. \ (\forall x. \ x \in X = \implies y \leq x) = \implies y \leq a$
  shows $\text{Inf } X = a$
  by (intro 3 1 antisym cInf-greatest) (auto intro: 2 1 cInf-lower)

lemma cInf-cSup:
  $S \neq \{\} = \implies \text{bdd-below } S = \implies \text{Inf } S = \text{Sup } \{x. \ \forall s \in S. \ x \leq s\}$
  by (rule cInf-eq-non-empty) (auto intro: cSup-upper cSup-least simp: bdd-below-def)

lemma cSup-cInf:
  $S \neq \{\} = \implies \text{bdd-above } S = \implies \text{Sup } S = \text{Inf } \{x. \ \forall s \in S. \ s \leq x\}$
  by (rule cSup-eq-non-empty) (auto intro: cInf-lower cInf-greatest simp: bdd-above-def)

lemma cSup-insert: $X \neq \{\} = \implies \text{bdd-above } X = \implies \text{Sup } (\text{insert } a X) = \text{sup } a (\text{Sup } X)$
  by (intro cSup-eq-non-empty) (auto intro: le-supI2 cSup-upper cSup-least)

lemma cInf-insert: $X \neq \{\} = \implies \text{bdd-below } X = \implies \text{Inf } (\text{insert } a X) = \text{inf } a (\text{Inf } X)$
  by (intro cInf-eq-non-empty) (auto intro: le-infI2 cInf-lower cInf-greatest)

lemma cSup-singleton [simp]: $\text{Sup } \{x\} = x$
  by (intro cSup-eq-maximum) auto

lemma cInf-singleton [simp]: $\text{Inf } \{x\} = x$
  by (intro cInf-eq-minimum) auto

lemma cSup-insert-If: $\text{bdd-above } X = \implies \text{Sup } (\text{insert } a X) = (\text{if } X = \{\} \text{ then } a \text{ else } \text{sup } a (\text{Sup } X))$
  using cSup-insert[of $X$] by simp

lemma cInf-insert-If: $\text{bdd-below } X = \implies \text{Inf } (\text{insert } a X) = (\text{if } X = \{\} \text{ then } a \text{ else } \text{inf } a (\text{Inf } X))$
  using cInf-insert[of $X$] by simp

lemma le-cSup-finite: $\text{finite } X = \implies x \in X \implies x \leq \text{Sup } X$

proof (induct $X$ arbitrary: $x$ rule: finite-induct)
  case (insert $x$ $X$ $y$) then show ?case
by (cases X = {}) (auto simp: cSup-insert intro: le-supI2)

qed simp

lemma cInf-le-finite: finite X \implies x \in X \implies \Inf X \leq x
proof (induct X arbitrary: x rule: finite-induct)
  case (insert x X y)
  then show ?case
  by (cases X = {}) (auto simp: cInf-insert intro: le-infI2)
qed simp

lemma cSup-eq-Sup-fin: finite X \implies X \neq {} \implies \Sup X = \Sup-fin X
by (induct X rule: finite-ne-induct) (simp-all add: cSup-insert)

lemma cInf-eq-Inf-fin: finite X \implies X \neq {} \implies \Inf X = \Inf-fin X
by (induct X rule: finite-ne-induct) (simp-all add: cInf-insert)

lemma cSup-atMost\[simp\]: \Sup \{..x\} = x
by (auto intro!: cSup-eq-maximum)

lemma cSup-greaterThanAtMost\[simp\]: y < x \implies \Sup \{y..<x\} = x
by (auto intro!: cSup-eq-maximum)

lemma cSup-atLeastAtMost\[simp\]: y \leq x \implies \Sup \{y..x\} = x
by (auto intro!: cSup-eq-maximum)

lemma cInf-atLeast\[simp\]: \Inf \{x..\} = x
by (auto intro!: cInf-eq-minimum)

lemma cInf-atLeastLessThan\[simp\]: y < x \implies \Inf \{y..<x\} = y
by (auto intro!: cInf-eq-minimum)

lemma cInf-atLeastAtMost\[simp\]: y \leq x \implies \Inf \{y..x\} = y
by (auto intro!: cInf-eq-minimum)

lemma cINF-lower: bdd-below (f ' A) \implies x \in A \implies \bigcap (f ' A) \leq f x
using cINF-lower [of - f ' A] by simp

lemma cINF-greatest: A \neq {} \implies (\bigwedge x. x \in A \implies m \leq f x) \implies m \leq \bigcap (f ' A)
using cINF-greatest [of f ' A] by auto

lemma cSUP-upper: x \in A \implies bdd-above (f ' A) \implies f x \leq \bigcup (f ' A)
using cSUP-upper [of - f ' A] by simp

lemma cSUP-least: A \neq {} \implies (\bigwedge x. x \in A \implies f x \leq M) \implies \bigcup (f ' A) \leq M
using cSUP-least [of f ' A] by auto

lemma cINF-lower2: bdd-below (f ' A) \implies x \in A \implies f x \leq u \implies \bigcap (f ' A) \leq u
by (auto intro: cINF-lower order-trans)

lemma cSUP-upper2: bdd-above (f ' A) \implies x \in A \implies u \leq f x \implies u \leq \bigcup (f ' A)
by (auto intro: cSUP-upper order-trans)

lemma cSUP-const [simp]: \( A \neq \{\} \implies (\bigcup_{x \in A} \cdot c) = c \) by (intro antisym cSUP-least) (auto intro: cSUP-upper)

lemma cINF-const [simp]: \( A \neq \{\} \implies (\bigcap_{x \in A} \cdot c) = c \) by (intro antisym cINF-greatest) (auto intro: cINF-lower)

lemma le-cINF-iff: \( A \neq \{\} \implies \text{bdd-below} (f \cdot A) \implies u \leq \bigcap (f \cdot A) \iff (\forall x \in A. \ u \leq f x) \) by (metis cINF-greatest cINF-lower order-trans)

lemma cSUP-le-iff: \( A \neq \{\} \implies \text{bdd-above} (f \cdot A) \implies u \geq \bigcup (f \cdot A) \iff (\forall x \in A. \ f x \leq u) \) by (metis cSUP-least cSUP-upper order-trans)

lemma less-cINF-D: \( \text{bdd-below} (f \cdot A) \implies y < (\bigcap i \in A. \ f i) \implies i \in A \implies y < f i \) by (metis cINF-lower less-le-trans)

lemma cSUP-lessD: \( \text{bdd-above} (f \cdot A) \implies (\bigcup i \in A. \ f i) < y \implies i \in A \implies f i < y \) by (metis cSUP-upper le-less-trans)

lemma cINF-insert: \( A \neq \{\} \implies \text{bdd-below} (f \cdot A) \implies \bigcap (f \cdot \text{insert} a A) = \inf (f a) (\bigcap (f \cdot A)) \) by (simp add: cInf-insert)

lemma cSUP-insert: \( A \neq \{\} \implies \text{bdd-above} (f \cdot A) \implies \bigcup (f \cdot \text{insert} a A) = \sup (f a) (\bigcup (f \cdot A)) \) by (simp add: cSup-insert)

lemma cINF-mono: \( B \neq \{\} \implies \text{bdd-below} (f \cdot A) \implies (\forall m. \ m \in B \implies \exists n \in A. \ f n \leq g m) \implies \bigcap (f \cdot A) \leq \bigcap (g \cdot B) \) using cInf-mono [of g \cdot B f \cdot A] by auto

lemma cSUP-mono: \( A \neq \{\} \implies \text{bdd-above} (g \cdot B) \implies (\forall n. \ n \in A \implies \exists m \in B. \ f n \leq g m) \implies \bigcup (f \cdot A) \leq \bigcup (g \cdot B) \) using cSup-mono [of f \cdot A g \cdot B] by auto

lemma cINF-superset-mono: \( A \neq \{\} \implies \text{bdd-below} (g \cdot B) \implies A \subseteq B \implies (\forall x. \ x \in B \implies g x \leq f x) \implies \bigcap (g \cdot B) \leq \bigcap (f \cdot A) \) by (rule cINF-mono) auto

lemma cSUP-subset-mono: \[ A \neq \{\}; \text{bdd-above} (g \cdot B); A \subseteq B; \forall x. \ x \in A \implies f x \leq g x \] \implies \bigcup (f \cdot A) \leq \bigcup (g \cdot B) by (rule cSUP-mono) auto

lemma less-eq-cInf-inter: \( \text{bdd-below} A \implies \text{bdd-below} B \implies A \cap B \neq \{\} \implies \inf (\text{Inf} A) (\text{Inf} B) \leq \text{Inf} (A \cap B) \)
proof

cSup-inter-less-eq: bdd-above A \implies bdd-above B \implies A \cap B \neq \{} \implies \text{Sup} (A \cap B) \leq \text{sup} (\text{Sup} A) (\text{Sup} B)

by (metis cSup-subset-mono lattice-class.inf-sup-ord(1) le-infI1)

lemma cSup-inter-less-eq: bdd-above A \implies bdd-above B \implies A \cap B \neq \{} \implies bdd-above (A \cap B) \leq bdd-above (\text{Sup} A) (\text{Sup} B)

by (metis cSup-subset-mono lattice-class.inf-sup-ord(1) le-infI1)

lemma cInf-union-distrib: A \neq \{} \implies bdd-below A \implies B \neq \{} \implies bdd-below (A \cup B) = bdd-below ((\text{Inf} A) (\text{Inf} B))

using cInf-union-distrib [of f' A f' B] by (simp add: image-Un)

lemma cSup-union-distrib: A \neq \{} \implies bdd-below (f' A) \implies B \neq \{} \implies bdd-below (f' A \cup f' B) = bdd-below (f' (A \cup B))

using cSup-union-distrib [of f' A f' B] by (simp add: image-Un)

lemma cINF-inf-distrib: A \neq \{} \implies bdd-below ((f' A) \cap (g' A)) \implies bdd-below (\text{Inf} (A \cap (f' A) \cap (g' A))) = \text{Inf} \cap (\text{Inf} (f' A) \cap (g' A))

by (intro antisym le-infI cINF-greatest cINF-lower2)

(auto intro: le-supI1 le-supI2 cSup-upper)

lemma cSUP-sup-distrib: A \neq \{} \implies bdd-below ((f' A) \cup (g' A)) \implies bdd-below (f' (A \cup (f' A) \cup (g' A))) = bdd-below \cup (f' A) \cup (g' A)

by (intro antisym le-supI cSUP-least cSUP-upper2)

(auto intro: le-supI1 le-supI2 cSUP-least cSUP-upper le-supI)

lemma cInf-le-cSup:

A \neq \{} \implies bdd-above A \implies bdd-below (A \cup B) \implies \text{Inf} A \leq \text{Sup} A

by (auto intro!: cSup-upper2[of SOME a. a \in A] intro: someI cInf-lower)

end

instance complete-lattice \subseteq \text{conditionally-complete-lattice}

by standard (auto intro: Sup-upper le-supI1 le-supI2 Inf-le-supI inf-sup-ord(1) cInf-superset-mono)

lemma cSup-eq:

fixes a :: 'a :: \{conditionally-complete-lattice, no-bot\}

assumes upper: \A x. x \in X \implies x \leq a

assumes least: \A y. (\A x. x \in X \implies x \leq y) \implies a \leq y

shows \text{Sup} X = a

proof cases
assume $X = \{\}$ with lt-ex[of a] least show ?thesis by (auto simp: less-le-not-le)
qed (intro cSup-eq-non-empty assms)

lemma cInf-eq:
fixed a :: 'a :: {conditionally-complete-lattice, no-top}
assumes upper: $\forall x. x \in X \Rightarrow a \leq x$
assumes least: $\forall y. (\forall x. x \in X \Rightarrow y \leq x) \Rightarrow y \leq a$
shows Inf $X = a$
proof cases
  assume $X = \{\}$ with gt-ex[of a] least show ?thesis by (auto simp: less-le-not-le)
qed (intro cInf-eq-non-empty assms)

class conditionally-complete-linorder = conditionally-complete-lattice + linorder
begin


lemma cINF-less-iff: $A \neq \{\} \Rightarrow bdd-below (f'$A) $\Rightarrow (\exists x \in A. f i < a) \leftarrow (\exists x \in A. f x < a)$
using less-cSupE[of f'$A] by auto

lemma less-cSupE: $y < Sup X X \neq \{\} \Rightarrow x \in X y \neq x$ by (metis cSup-least assms not-le that)

lemma less-cSupD: $z < Sup X \neq \{\} \Rightarrow z \in X \Rightarrow (\exists x \in X. z < x)$ by (metis less-cSup-iff not-le-imp-less bdd-above-def)

lemma cInf-lessD: $z < Inf X \neq \{\} \Rightarrow x \in X. z < x$
by (metis cInf-less-iff not-le-imp-less bdd-below-def)

lemma complete-interval:
assumes $a < b \text{ and } P a \text{ and } \neg P b$
shows $\exists c. a \leq c \land c \leq b \land (\forall x. a \leq x \land x < c \Rightarrow P x) \land (\forall d. (\forall x. a \leq x \land x < d \Rightarrow P x) \Rightarrow d \leq c)$$
proof (rule exI [where $x = \text{Sup} \{d. \forall x. a \leq x \land x < d \Rightarrow P x\}$], auto)
show $a \leq \text{Sup} \{d. \forall c. a \leq c \land c < d \Rightarrow P c\}$
by (rule cSup-upper, auto simp: bdd-above-def)
  (metis (a < b) (∀ P b) linear less-le)

next
show Sup {d. ∀ c. a ≤ c ∧ c < d → P c} ≤ b
  apply (rule cSup-least)
  apply auto
  apply (metis less-le-not-le)
  apply (metis (a < b) (∀ P b) linear less-le)
  done

next
fix x
assume x: a ≤ x and lt: x < Sup {d. ∀ c. a ≤ c ∧ c < d → P c}
show P x
  apply (rule less-cSupE [OF lt], auto)
  apply (metis less-le-not-le)
  apply (metis x)
  done

next
fix d
assume 0: ∃ x. a ≤ x ∧ x < d → P x
thus d ≤ Sup {d. ∀ c. a ≤ c ∧ c < d → P c}
  by (rule-tac cSup-upper, auto simp: bdd-above-def)
  (metis (a < b) (∀ P b) linear less-le)

qed

end

instance complete-linorder < conditionally-complete-linorder ..

lemma cSup-eq-Max: finite (X::a::conditionally-complete-linorder set) ⇒ X ≠ {}
  ⇒ Sup X = Max X
  using cSup-eq-Sup-fin[of X] by (simp add: Sup-fin-Max)

lemma cInf-eq-Min: finite (X::a::conditionally-complete-linorder set) ⇒ X ≠ {}
  ⇒ Inf X = Min X
  using cInf-eq-Inf-fin[of X] by (simp add: Inf-fin-Min)

lemma cSup-lessThan[simp]: Sup {..<x::a::conditionally-complete-linorder, no-bot, dense-linorder} = x
  by (auto intro!: cSup-eq-non-empty intro: dense-le)

lemma cSup-greaterThanLessThan[simp]: y < x ⇒ Sup {y..<x::a::conditionally-complete-linorder, dense-linorder} = x
  by (auto intro!: cSup-eq-non-empty intro: dense-le-bounded)

lemma cSup-atLeastLessThan[simp]: y < x ⇒ Sup {y..<x::a::conditionally-complete-linorder, dense-linorder} = x
  by (auto intro!: cSup-eq-non-empty intro: dense-le-bounded)
lemma cInf-greaterThan[simp]: \( \inf \{ x \::\:: \{ \text{conditionally-complete-linorder, no-top, dense-linorder} \ \} <.. \} = x \)
  by (auto intro!: cInf-eq-non-empty intro: dense-ge)

lemma cInf-greaterThanAtMost[simp]: \( y < x \implies \inf \{ y<..x \::\:: \{ \text{conditionally-complete-linorder, dense-linorder} \} \} = y \)
  by (auto intro!: cInf-eq-non-empty intro: dense-ge-bounded)

lemma cInf-greaterThanLessThan[simp]: \( y < x \implies \inf \{ y<..<x \::\:: \{ \text{conditionally-complete-linorder, dense-linorder} \} \} = y \)
  by (auto intro!: cInf-eq-non-empty intro: dense-ge-bounded)

lemma Inf-insert-finite:
  fixes \( S \::\:: \{ \text{conditionally-complete-linorder set} \} \)
  shows \( \text{finite } S \implies \inf (\text{insert } x \ S) = (\text{if } S = \{ \} \text{ then } x \text{ else } \min x (\inf S)) \)
  by (simp add: cInf-eq-Min)

lemma Sup-insert-finite:
  fixes \( S \::\:: \{ \text{conditionally-complete-linorder set} \} \)
  shows \( \text{finite } S \implies \sup (\text{insert } x \ S) = (\text{if } S = \{ \} \text{ then } x \text{ else } \max x (\sup S)) \)
  by (simp add: cSup-insert sup-max)

lemma finite-imp-less-Inf:
  fixes \( a \::\:: \{ \text{conditionally-complete-linorder} \} \)
  shows \( \text{finite } X, x \in X, \forall x. x \in X \implies a < x \implies a < \inf X \)
  by (induction X rule: finite-induct (simp-all add: cInf-eq-Min Inf-insert-finite))

lemma finite-less-Inf-iff:
  fixes \( a \::\:: \{ \text{conditionally-complete-linorder} \} \)
  shows \( \text{finite } X, X \neq \{ \} \implies a < \inf X \iff (\forall x. x \in X. a < x) \)
  by (auto simp: cInf-eq-Min)

lemma finite-imp-Sup-less:
  fixes \( a \::\:: \{ \text{conditionally-complete-linorder} \} \)
  shows \( \text{finite } X, x \in X, \forall x. x \in X \implies a > x \implies a > \sup X \)
  by (induction X rule: finite-induct (simp-all add: cSup-eq-Max Sup-insert-finite))

lemma finite-Sup-less-iff:
  fixes \( a \::\:: \{ \text{conditionally-complete-linorder} \} \)
  shows \( \text{finite } X, X \neq \{ \} \implies a > \sup X \iff (\forall x. x \in X. a > x) \)
  by (auto simp: cSup-eq-Max)

class linear-continuum = conditionally-complete-linorder + dense-linorder +
  assumes UNIV-not-singleton: \( \exists a \ b. \ a \neq b \)
begin

lemma ex-gt-or-lt: \( \exists b. \ a < b \lor b < a \)
  by (metis UNIV-not-singleton neq_iff)
definition \( \text{Sup} (X::\text{nat set}) = \text{Max} X \)

definition \( \text{Inf} (X::\text{nat set}) = (\text{LEAST} \ n \ n \in X) \)

lemma \( \text{bdd-above-nat}: \text{bdd-above} \ X \iff \text{finite} (X::\text{nat set}) \)

proof
  assume \( \text{bdd-above} \ X \)
  then obtain \( z \) where \( X \subseteq \{.. z\} \)
    by (auto simp: \text{bdd-above-def})
  then show \( \text{finite} \ X \)
    by (rule \text{finite-subset}) simp
qed simp

instance proof
  fix \( x :: \text{nat} \)
  fix \( X :: \text{nat set} \)
  show \( \text{Inf} \ X \leq x \) if \( x \in X \text{ bdd-below} X \)
    using that by (simp add: \text{Inf-def} Least-le)
  show \( x \leq \text{Inf} \ X \) if \( X \neq \{\} \) \( \forall y. y \in X \implies x \leq y \)
    using that unfolding \text{Inf-def ex-in-conv}[symmetric] by (rule LeastI2-ex)
  show \( \text{Sup} \ X \leq x \) if \( x \in X \text{ bdd-above} X \)
    using that by (simp add: \text{Sup-def} \text{bdd-above-nat})
  show \( \text{Sup} \ X \leq x \) if \( X \neq \{\} \) \( \forall y. y \in X \implies y \leq x \)
    proof -
      from that have \( \text{bdd-above} \ X \)
        by (auto simp: \text{bdd-above-def})
      with that show \( ?\text{thesis} \)
        by (simp add: \text{Sup-def} \text{bdd-above-nat})
    qed
  qed

end

lemma \text{Inf-def1}: fixes \( K::\text{nat set} \)
  assumes \( K \neq \{\} \)
  shows \( \text{Inf} \ K \in K \)
  by (auto simp add: Min-def \text{Inf-def}) (meson LeastI assms bot.extremum-unique subsetI)

instantiation \( \text{int} :: \text{conditionally-complete-linorder} \)
begin
definition $\text{Sup } (X::{\text{int }}{\text{ set}}) = (\text{THE } x. x \in X \land (\forall y \in X. y \leq x))$

definition $\text{Inf } (X::{\text{int }}{\text{ set}}) = - (\text{Sup } (\text{uminus } X))$

instance
proof
{ fix $x :: {\text{int }}$ and $X :: {\text{int }}{\text{ set}}$ assume $X \neq \{\}$ bdd-above $X$
  then obtain $x y$ where $X \subseteq \{..y\} x \in X$
  by (auto simp: bdd-above-def)
  then have $*: \text{finite } (X \cap \{x..y\}) X \cap \{x..y\} \neq \{\}$ and $x \leq y$
  by (auto simp: subset-eq)
  have $\exists! x \in X. (\forall y \in X. y \leq x)$
  proof
  { fix $z$
    assume $z \in X$
    have $z \leq \text{Max } (X \cap \{x..y\})$
    proof cases
      assume $x \leq z$ with $z \in X \cap \{x..y\}$ *1 show $?\text{thesis}$
      by (auto intro!: Max-ge)
    next
      assume $\neg x \leq z$
      then have $z < x$ by simp
      also have $x \leq \text{Max } (X \cap \{x..y\})$
      using $x \in X$ *1 (1) $x \leq y$ by (intro Max-ge) auto
      finally show $?\text{thesis}$ by simp
  qed 
  note le = this
  with $\text{Max-in[OF } *\text{]}$ show $\exists! x \in X. (\forall y \in X. y \leq x)$
  proof cases
    assume $x \leq z$ with $z \in X \cap \{x..y\}$ *1 show $?\text{thesis}$
    by (auto intro!: Max-ge)
    next
      assume $\neg x \leq z$
      then have $z < x$ by simp
      also have $x \leq \text{Max } (X \cap \{x..y\})$
      using *1 $\exists! x \in X. (\forall y \in X. y \leq x)$ by (intro Max-ge) auto
      finally show $?\text{thesis}$ by simp
  qed
  note le-Sup = this
  { fix $x :: {\text{int }}$ and $X :: {\text{int }}{\text{ set}}$ assume $x \in X \land (\forall z \in X. z \leq \text{Max } (X \cap \{x..y\}))$
    by auto
    fix $z$
    assume $*: z \in X \land (\forall y \in X. y \leq z)$
    with $\text{le}$ have $z \leq \text{Max } (X \cap \{x..y\})$
    by auto
    moreover have $\text{Max } (X \cap \{x..y\}) \leq z$
    using $* \text{ex}$ by auto
    ultimately show $z = \text{Max } (X \cap \{x..y\})$
    by auto
    qed
  then have $\text{Sup } X \in X \land (\forall y \in X. y \leq \text{Sup } X)$
  unfolding $\text{Sup-int-def}$ by (rule theI' )
  note Sup-int = this
}

{ fix $x :: {\text{int }}$ and $X :: {\text{int }}{\text{ set}}$ assume $x \in X \land (\forall z \in X. z \leq \text{Sup } X)$
  using $\text{Sup-int[of } X\text{]}$ by auto }
note le-Sup = this
{ fix $x :: {\text{int }}$ and $X :: {\text{int }}{\text{ set}}$ assume $X \neq \{\}$ and $y \in X \Rightarrow y \leq x$
  then show $\text{Sup } X \leq x$
    using $\text{Sup-int[of } X\text{]}$ by (auto simp: bdd-above-def) 
  note Sup-le = this
THEORY "Conditionally-Complete-Lattices"

{ fix x :: int and X :: int set assume x ∈ X bdd-below X then show Inf X ≤ x
  using le-Sup[of -x uminus ' X] by (auto simp: Inf-int-def) }
{ fix x :: int and X :: int set assume X ≠ {} ∧ y ∈ X → x ≤ y then show x ≤ Inf X
  using Sup-le[of uminus ' X -x] by (force simp: Inf-int-def) }

qed

end

lemma interval-cases:
 fixes S :: 'a :: conditionally-complete-linorder set
 assumes ivl: ∃a b x. a ∈ S → b ∈ S → a ≤ x → x ≤ b → x ∈ S
 shows ∃a b. S = {} ∨ S = UNIV ∨ S = {..<b} ∨ S = {..b} ∨ S = {a<..} ∨ S = {..<a} ∨ S = {a..<b} ∨ S = {a<..<b} ∨ S = {a..<b} ∨ S = {a..b}
 proof
  define lower upper where lower = {x. ∃s∈S. s ≤ x} and upper = {x. ∃s∈S. x ≤ s}
  with ivl have S = lower ∩ upper
    by auto
  moreover have ∃a. upper = UNIV ∨ upper = {} ∨ upper = {..<a} ∨ upper = {..a}
    by auto
  next
  assume *: bdd-above S ∧ S ≠ {} from * have upper ⊆ {..< Sup S}
    by (auto simp: upper-def intro: cSup-upper2)
  moreover from * have {..< Sup S} ⊆ upper
    by (force simp add: less-cSup-iff upper-def subset-eq Ball-def)
  ultimately have upper = {..< Sup S} ∨ upper = {..< Sup S}
    unfolding ieq-disj-un(2)[symmetric] by auto
  then show ?thesis by auto
 next
  assume ¬(bdd-above S ∧ S ≠ {})
  then have upper = UNIV ∨ upper = {} by (auto simp: upper-def bdd-above-def not-le dest: less-imp-le)
  then show ?thesis by auto
 qed

moreover
  have ∃b. lower = UNIV ∨ lower = {} ∨ lower = {b ..} ∨ lower = {b <..}
  proof
  case
THEORY "Conditionally-Complete-Lattices"

assume *: bdd-below S ∧ S ≠ { }
from * have lower ⊆ {Inf S ..}
  by (auto simp: lower-def intro: cInf-lower2)
moreover from * have {Inf S <..} ⊆ lower
  by (force simp add: cInf-less-iff lower-def subset-eq Ball-def)
ultimately have lower = {Inf S ..} ∨ lower = {Inf S <..}
  unfolding inf-disj-un[symmetric] by auto
then show ?thesis by auto
next
assume ¬ (bdd-below S ∧ S ≠ { })
then have lower = UNIV ∨ lower = { }
  by (auto simp: lower-def bdd-below-def not-le dest: less-imp-le)
then show ?thesis by auto
qed ultimately show ?thesis
  unfolding greaterThanAtMost-def greaterThanLessThan-def atLeastAtMost-def
  atLeastLessThan-def
  by (metis inf-bot-left inf-bot-right inf-top-left-neutral inf-top.right-neutral)
qed

lemma cSUP-eq-cINF-D:
  fixes f :: 'b::conditionally-complete-lattice
  assumes eq: (∨ x∈A. f x) = (∨ x∈A. f x)
  and bdd: bdd-above (f ' A) bdd-below (f ' A)
  and a: a ∈ A
  shows f a = (∨ x∈A. f x)
apply (rule antisym)
using a bdd
apply (auto simp: cINF-lower)
apply (metis eq cSUP-upper)
done

lemma cSUP-UNION:
  fixes f :: 'b::conditionally-complete-lattice
  assumes ne: A ≠ { } ∧ x ∈ A ⇒ B(x) ≠ { }
  and bdd-UN: bdd-above (⋃ x∈A. f ' B x)
  shows (⋃ z ∈ ⋃ x∈A. B x. f z) = (⋃ x∈A. ⋃ z∈B x. f z)
proof −
  have bdd: ∀ x. x ∈ A ⇒ bdd-above (f ' B x)
    using bdd-UN by (meson UN-upper bdd-above mono)
  obtain M where ∨ x y. x ∈ A ⇒ y ∈ B(x) ⇒ f y ≤ M
    using bdd-UN by (auto simp: bdd-above-def)
  then have bdd2: bdd-above ((λx. ⋃ z∈B x. f z) ' A)
    unfolding bdd-above-def by (force simp: bdd cSUP-le-iff ne(2))
  have (∨ z ∈ ⋃ x∈A. B x. f z) ≤ (⋃ x∈A. ⋃ z∈B x. f z)
    using assms by (fastforce simp add: intro: cSUP-least intro: cSUP-upper2 simp: bdd2 bdd)
  moreover have (⋃ x∈A. ⋃ z∈B x. f z) ≤ (⋃ z ∈ ⋃ x∈A. B x. f z)

using assms by (fastforce simp add: intro!: cSUP-least intro: cSUP-upper simp: image-UN bdd-UN)
ultimately show ?thesis
  by (rule order-antisym)
qed

lemma cINF-UNION:
  fixes f :: 'a::conditionally-complete-lattice
assumes ne: A ≠ {} \( \forall x \in A \Rightarrow B(x) \neq {} \)
  and bdd-UN: bdd-below \( (\bigcup x \in A. f \cdot B x) \)
shows \( (\prod z \in \bigcup x \in A. B x. f z) = (\prod x \in A. \prod z \in B x. f z) \)
proof –
  have bdd: \( \forall x \in A \Rightarrow bdd-below (f \cdot B x) \)
  using bdd-UN by (meson UN-upper bdd-below-mono)
  obtain M where \( \forall x y. x \in A \Rightarrow y \in B(x) \Rightarrow f y \geq M \)
  using bdd-UN by (auto simp: bdd-below-def)
  then have bdd2: bdd-below \( ((\lambda x. \prod z \in B x. f z) \cdot A) \)\)
  unfolding bdd-below-def by (force simp: bdd le-cINF-iff ne(2))
  have \( (\prod z \in \bigcup x \in A. B x. f z) \leq (\prod x \in A. \prod z \in B x. f z) \)
  using assms by (fastforce simp add: intro!: cINF-greatest intro: cINF-lower simp: bdd bdd-UN bdd2)
  moreover have \( (\prod x \in A. \prod z \in B x. f z) \leq (\prod z \in \bigcup x \in A. B x. f z) \)
  using assms by (fastforce simp add: intro!: cINF-greatest intro: cINF-lower2 simp: bdd bdd-UN bdd2)
  ultimately show ?thesis
    by (rule order-antisym)
qed

lemma cSup-abs-le:
  fixes S :: ('a::linordered-idom,conditionally-complete-linorder) set
shows \( S \neq {} \Rightarrow (\forall x. x \in S \Rightarrow |x| \leq a) \Rightarrow |\text{Sup } S| \leq a \)
apply (auto simp add: abs-le-iff intro: cSup-least)
by (metis bdd-aboveI cSup-upper neg-le-iff-le order-trans)

90 Factorial Function, Rising Factorials

theory Factorial
  imports Groups-List
begin

90.1 Factorial Function

context semiring-char-0
begin

definition fact :: nat ⇒ 'a
  where fact-prod: fact n = of-nat (\prod \{1..n\})
lemmad fact-prod-Suc: fact n = of-nat (prod Suc {0..<n})
  unfolding fact-prod using atLeast0LessThan prod.atLeast1-atMost-eq by auto

lemma fact-prod-rev: fact n = of-nat (\prod i = 0..<n. n - i)
proof
  have prod Suc {0..<n} = \prod {1..n}
    by (simp add: atLeast0LessThan prod.atLeast1-atMost-eq)
  then have prod Suc {0..<n} = prod ((\-) (n + 1)) {1..n}
    using prod.atLeastAtMost-rev [of \lambda i. i 1 n] by presburger
  then show ?thesis
    unfolding fact-prod-Suc by (simp add: atLeast0LessThan prod.atLeast1-atMost-eq)
qed

lemma fact-0 [simp]: fact 0 = 1
  by (simp add: fact-prod)

lemma fact-1 [simp]: fact 1 = 1
  by (simp add: fact-prod)

lemma fact-Suc-0 [simp]: fact (Suc 0) = 1
  by (simp add: fact-prod)

lemma fact-Suc [simp]: fact (Suc n) = of-nat (Suc n) * fact n
  by (simp add: fact-prod atLeastAtMostSuc-cone algebra-simps)

lemma fact-2 [simp]: fact 2 = 2
  by (simp add: numeral-2-eq-2)

lemma fact-split: k \leq n \implies fact n = of-nat (prod Suc {n - k..<n}) * fact (n - k)
  by (simp add: fact-prod-Suc prod.union-disjoint [symmetric]
    iel-disj-an ac-simps of-nat-mult [symmetric])
end

lemma of-nat-fact [simp]: of-nat (fact n) = fact n
  by (simp add: fact-prod)

lemma of-int-fact [simp]: of-int (fact n) = fact n
  by (simp only: fact-prod of-int-of-nat-eq)

lemma fact-reduce: n > 0 \implies fact n = of-nat n * fact (n - 1)
  by (cases n) auto

lemma fact-nonzero [simp]: fact n \neq (0::'a::{semiring-char-0,semiring-no-zero-divisors})
  apply (induct n)
  apply auto
  using of-nat-eq-0-iff
apply fastforce
done

lemma fact-mono-nat: \( m \leq n \implies \text{fact} m \leq (\text{fact} n :: \mathbb{N}) \)
by (induct n) (auto simp: le-Suc-eq)

lemma fact-in-Nats: \( \text{fact} n \in \mathbb{N} \)
by (induct n) auto

lemma fact-in-Ints: \( \text{fact} n \in \mathbb{Z} \)
by (induct n) auto

context assumes SORT-CONSTRAINT('a::linordered-semidom)
begin

lemma fact-mono: \( m \leq n \implies \text{fact} m \leq (\text{fact} n :: 'a) \)
by (metis of-nat-fact of-nat-le-iff fact-mono-nat)

lemma fact-ge-1 [simp]: \( \text{fact} n \geq (1 :: 'a) \)
by (metis le0 fact-0 fact-mono)

lemma fact-gt-zero [simp]: \( \text{fact} n > (0 :: 'a) \)
using fact-ge-1 less-le-trans zero-less-one by blast

lemma fact-ge-zero [simp]: \( \text{fact} n \geq (0 :: 'a) \)
by (simp add: less-imp-le)

lemma fact-not-neg [simp]: \( \neg \text{fact} n < (0 :: 'a) \)
by (simp add: not-less-iff-gr-or-eq)

lemma fact-le-power: \( \text{fact} n \leq (\text{of-nat} (n^n) :: 'a) \)
proof (induct n)
  case 0
  then show \(?case\) by simp
next
  case (Suc n)
  then have \(*\): \( \text{fact} n \leq (\text{of-nat} (Suc n \cdot n) :: 'a) \)
    by (rule order-trans) (simp add: power-mono del: of-nat-power)
  have fact (Suc n) = (of-nat (Suc n) * fact n :: 'a)
    by (simp add: algebra-simps)
  also have \(...\) \leq of-nat (Suc n) * of-nat (Suc n \cdot n)
    by (simp add: * ordered-comm-semiring-class.comm-mult-left-mono del: of-nat-power)
  also have \(...\) \leq of-nat (Suc n \cdot Suc n)
    by (metis of-nat-mul refl power-Suc)
  finally show \(?case\).
qed

end
lemma fact-less-mono-nat: \(0 < m \implies m < n \implies \text{fact } m < (\text{fact } n :: \text{nat})\)
by (induct n) (auto simp: less-Suc-eq)

lemma fact-less-mono: \(0 < m \implies m < n \implies \text{fact } m < (\text{fact } n :: 'a::linordered-semidom)\)
by (metis of-nat-fact of-nat-less-iff fact-less-mono-nat)

lemma fact-ge-Suc-0-nat [simp]: \(\text{fact } n \geq \text{Suc } 0\)
by (metis One-nat-def fact-ge-1)

lemma dvd-fact: \(1 \leq m \implies m \leq n \implies m \text{ dvd fact } n\)
by (induct n) (auto simp: dvdI le-Suc-eq)

lemma fact-ge-self: \(\text{fact } n \geq n\)
by (cases n = 0) (simp-all add: dvd-imp-le dvd-fact)

lemma fact-dvd: \(n \leq m \implies \text{fact } n \text{ dvd } (\text{fact } m :: 'a::linordered-semidom)\)
by (induct m) (auto simp: le-Suc-eq)

lemma fact-mod: \(m \leq n \implies \text{fact } n \mod (\text{fact } m :: 'a::\{semidom-modulo, linordered-semidom\}) = 0\)
by (simp add: mod-eq-0-iff-dvd fact-dvd)

lemma fact-div-fact:
assumes \(m \geq n\)
shows \(\text{fact } m \div \text{fact } n = \prod \{n + 1..m\}\)
proof –
obtain d where \(d = m - n\)
by auto
with assms have \(m = n + d\)
by auto
have \(\text{fact } (n + d) \div (\text{fact } n) = \prod \{n + 1..n + d\}\)
proof (induct d)
case 0
show ?case by simp
next
case (Suc d')
have \(\text{fact } (n + \text{Suc } d') \div \text{fact } n = \text{Suc } (n + d') \ast \text{fact } (n + d') \div \text{fact } n\)
by simp
also from Suc.hyps have \(\ldots = \text{Suc } (n + d') \ast \prod \{n + 1..n + d'\}\)
unfolding div-mult1-eq[of \cdot fact (n + d')][of \cdot fact (n + d')]
by (simp add: fact-mod)
also have \(\ldots = \prod \{n + 1..n + \text{Suc } d'\}\)
by (simp add: atLeastAtMostSuc-conv)
finally show ?case .
qed
with \(m = n + d\) show \(?thesis\) by simp
qed

lemma fact-num-eq-if: \(\text{fact } m = (\text{if } m = 0 \text{ then } 1 \text{ else } \text{of-nat } m \ast \text{fact } (m - 1))\)
by (cases m) auto

lemma fact-div-fact-le-pow:
  assumes r ≤ n
  shows fact n div fact (n - r) ≤ n ^ r
proof
  have r ≤ n ⇒ \prod \{ n - r .. n \} = (n - r) * \prod \{ Suc (n - r) .. n \} for r
  with assms show ?thesis
  by (induct r rule: nat.induct) (auto simp: atLeastAtMost-insertL)
qed

lemma prod-Suc-fact: prod Suc {0 ..< n} = fact n
by (simp add: fact-prod-Suc)

lemma prod-Suc-Suc-fact: prod Suc {Suc 0 ..< n} = fact n
proof (cases n = 0)
  case True
  then show ?thesis by simp
  next
  case False
  have prod Suc {Suc 0 ..< n} = Suc 0 * prod Suc {Suc 0 ..< n}
  by simp
  also have \ldots = prod Suc (insert 0 {Suc 0 ..< n})
  by simp
  also have insert 0 {Suc 0 ..< n} = {0 ..< n}
  using False by auto
  finally show ?thesis
  by (simp add: fact-prod-Suc)
qed

lemma fact-numeral: fact (numeral k) = numeral k * fact (pred-numeral k)
  — Evaluation for specific numerals
  by (metis fact-Suc numeral-eq-Suc of-nat-numeral)

90.2 Pochhammer’s symbol: generalized rising factorial

See https://en.wikipedia.org/wiki/Pochhammer_symbol.

color-style
context comm-semiring-1
begin

definition pochhammer :: 'a ⇒ nat ⇒ 'a
  where pochhammer-prod: pochhammer a n = prod (λi. a + of-nat i) {0..<n}

lemma pochhammer-prod-rev: pochhammer a n = prod (λi. a + of-nat (n - i)) {1..n}
  using prod.atLeastLessThan-rev-at-least-atMost [of λi. a + of-nat i 0 n]
  by (simp add: pochhammer-prod)
lemma pochhammer-Suc-prod: pochhammer a (Suc n) = prod (λi. a + of-nat i) {0..n}
  by (simp add: pochhammer-prod atLeastLessThanSuc-atLeastAtMost)

lemma pochhammer-Suc-prod-rev: pochhammer a (Suc n) = prod (λi. a + of-nat (n - i)) {0..n}
  using prod.atLeast-Suc-atMost-Suc-shift
  by (simp add: pochhammer-prod-rev prod.atLeast-Suc-atMost-Suc-shift del: prod.cl-ivl-Suc)

lemma pochhammer-0 [simp]: pochhammer a 0 = 1
  by (simp add: pochhammer-prod)

lemma pochhammer-1 [simp]: pochhammer a 1 = a
  by (simp add: pochhammer-prod lessThan-Suc)

lemma pochhammer-Suc0 [simp]: pochhammer a (Suc 0) = a
  by (simp add: pochhammer-prod lessThan-Suc)

lemma pochhammer-Suc: pochhammer a (Suc n) = pochhammer a n * (a + of-nat n)
  by (simp add: pochhammer-prod atLeast0-lessThan-Suc ac-simps)

end

lemma pochhammer-nonneg:
  fixes x :: 'a :: linordered-semidom
  shows x > 0 ⇒ pochhammer x n ≥ 0
  by (induction n) (auto simp: pochhammer-Suc intro!: mult-nonneg-nonneg add-nonneg-nonneg)

lemma pochhammer-pos:
  fixes x :: 'a :: linordered-semidom
  shows x > 0 ⇒ pochhammer x n > 0
  by (induction n) (auto simp: pochhammer-Suc intro!: mult-pos-pos add-pos-nonneg)

context comm-semiring-1
begin

lemma pochhammer-of-nat: pochhammer (of-nat x) n = of-nat (pochhammer x n)
  by (simp add: pochhammer-prod Factorial.pochhammer-prod)

end

context comm-ring-1
begin

lemma pochhammer-of-int: pochhammer (of-int x) n = of-int (pochhammer x n)
  by (simp add: pochhammer-prod Factorial.pochhammer-prod)

end
lemma pochhammer-rec: pochhammer \(a\) \((\text{Suc} \ n)\) = \(a \ast\) pochhammer \((a + 1)\) \(n\)
by (simp add: pochhammer-prod prod.atLeast0-lessThan-Suc-shift ac-simps del: prod.op-ivl-Suc)

lemma pochhammer-rec': pochhammer \(z\) \((\text{Suc} \ n)\) = \((z + \text{of-nat} \ n) \ast\) pochhammer \(z\) \(n\)
by (simp add: pochhammer-prod prod.atLeast0-lessThan-Suc ac-simps)

lemma pochhammer-fact: \(\text{fact} \ n\) = pochhammer \(1\) \(n\)
by (simp add: pochhammer-prod fact-prod-Suc)

lemma pochhammer-of-nat-eq-0-lemma: \(k > n\) \(\implies\) pochhammer \((− (\text{of-nat} \ n :: 'a:: idom))\) \(k\) = \(0\)
by (auto simp add: pochhammer-prod)

lemma pochhammer-of-nat-eq-0-lemma':
assumes \(\text{kn}\): \(k \leq n\)
shows pochhammer \((- (\text{of-nat} \ n :: 'a:: idom,ring-char-0))\) \(k\) \(\neq 0\)
proof (cases \(k\))
case \(0\)
then show \(?\text{thesis}\) by simp
next
case \((\text{Suc} \ h)\)
then show \(?\text{thesis}\)
  apply (simp add: pochhammer-Suc-prod)
  using \(\text{Suc kn}\)
  apply (auto simp add: algebra-simps)
  done
qed

lemma pochhammer-of-nat-eq-0-iff:
pochhammer \((- (\text{of-nat} \ n :: 'a:: idom,ring-char-0))\) \(k\) = \(0\) \(\iff\) \(k > n\)
(is \(?l = ?r\))
using pochhammer-of-nat-eq-0-lemma[of \(n\) \(k\), where \(?a=\text{'a}\)]
pochhammer-of-nat-eq-0-lemma[of \(k\) \(n\), where \(?a=\text{'a}\)]
by (auto simp add: not-le[symmetric])

lemma pochhammer-0-left:
pochhammer \(0\) \(n\) = (\(\text{if} \ n = 0 \text{ then } 1 \text{ else } 0\))
by (induction \(n\)) (simp-all add: pochhammer-rec)

lemma pochhammer-eq-0-iff: pochhammer \(a\) \(n\) = \((0::'a::field-char-0)\) \(\iff\) \((\exists \ k < n. \ a = − \text{of-nat} \ k)\)
by (auto simp add: pochhammer-prod eq-neg-iff-add-eq-0)

lemma pochhammer-eq-0-mono:
pochhammer \(a\) \(n\) = \((0::'a::field-char-0)\) \(\implies\) \(m \geq n \implies\) pochhammer \(a\) \(m\) = \(0\)
unfolding pochhammer-eq-0-iff by auto
lemma pochhammer-neq-0-mono: pochhammer a m ≠ (0::'a::field-char-0) ⇒ m ≥ n ⇒ pochhammer a n ≠ 0
unfolding pochhammer-eq-0-iff by auto

lemma pochhammer-minus: pochhammer (− b) k = ((− 1) ^ k :: 'a::comm-ring-1) * pochhammer (b − of-nat k + 1) k
proof (cases k)
case 0
then show ?thesis by simp
next
case (Suc h)
have eq: ((− 1) ^ Suc h :: 'a) = ([∏ i = 0..h. − 1)
  using prod-constant [where A={0..h} and y=− 1 :: 'a]
by auto
with Suc show ?thesis
  using pochhammer-Suc-prod-rev [of b − of-nat k + 1]
  by (auto simp add: pochhammer-Suc-prod prod.distrib [symmetric] eq of-nat-diff simp del: prod-constant)
qed

lemma pochhammer-minus': pochhammer (b − of-nat k + 1) k = ((− 1) ^ k :: 'a::comm-ring-1) * pochhammer (− b) k
by (simp add: pochhammer-minus)

lemma pochhammer-same: pochhammer (− of-nat n) n = ((− 1) ^ n :: 'a::{semiring-char-0,comm-ring-1,semiring-no-zero-divisors}) * fact n
unfolding pochhammer-minus
by (simp add: of-nat-diff pochhammer-fact)

lemma pochhammer-product': pochhammer z (n + m) = pochhammer z n * pochhammer z (z + of-nat n) m
proof (induct n arbitrary: z)
case 0
then show ?case by simp
next
case (Suc n z)
have pochhammer z (Suc n) * pochhammer (z + of-nat (Suc n)) m = z * (pochhammer (z + 1) n * pochhammer (z + 1 + of-nat n) m)
  by (simp add: pochhammer-rec ac-simps)
also note Suc[symmetric]
also have z * pochhammer (z + 1) (n + m) = pochhammer z (Suc (n + m))
  by (subst pochhammer-rec) simp
finally show ?case
  by simp
qed
lemma pochhammer-product:
\[ m \leq n \implies \text{pochhammer } z \ n = \text{pochhammer } z \ m \ * \ \text{pochhammer } (z + \text{of-nat } m) \ (n - m) \]
using pochhammer-product[of \( z \ n - m \)] by simp

lemma pochhammer-times-pochhammer-half:
fixes \( z :: 'a :: field_char_0 \)
shows \( \text{pochhammer } z \ (Suc \ n) \ * \ \text{pochhammer} \ (z + 1/2) \ (Suc \ n) = (\prod_{k=0..2*n+1} z + \text{of-nat } k / 2) \)
proof (induct \( n \))
case 0
then show ?case by (simp add: atLeast0-atMost-Suc)
next
case (Suc \( n \))
define \( n' \) where \( n' = Suc \ n \)
have \( \text{pochhammer } z \ (Suc \ n') \ * \ \text{pochhammer} \ (z + 1/2) \ n' = \)
\( (\text{pochhammer } z \ n' \ * \ \text{pochhammer} \ (z + 1/2) \ n') \ * \ ((z + \text{of-nat } n') \ * \ (z + 1/2 \ + \text{of-nat } n')) \)
(is - = - * ?A)
by (simp-all add: pochhammer-rec' mult-ac)
also have ?A = \( (z + \text{of-nat } (Suc \ (2 \ * \ n + 1)) / 2) \ * \ (z + \text{of-nat } (Suc \ (Suc \ (2 \ * \ n + 1)))) / 2) \)
(is - = ?B)
by (simp-all add: field-simps n'-def)
also note Suc[folded n'-def]
also have \( (\prod_{k=0..2 \ * \ n + 1} z + \text{of-nat } k / 2) \ * \ ?B = (\prod_{k=0..2 \ * \ Suc \ n + 1. \ z + \text{of-nat } k / 2}) \)
by (simp-all add: atLeast0-atMost-Suc)
finally show ?case
by (simp-all add: n'-def)
qed

lemma pochhammer-double:
fixes \( z :: 'a :: field_char_0 \)
shows \( \text{pochhammer } (2 \ * \ z) \ (2 \ * \ n) = \text{of-nat} \ (2^*(2*n)) \ * \ \text{pochhammer } z \ n \ * \ \text{pochhammer} \ (z+1/2) \ n \)
proof (induct \( n \))
case 0
then show ?case by simp
next
case (Suc \( n \))
have \( \text{pochhammer } (2 \ * \ z) \ (2 \ * \ (Suc \ n)) = \text{pochhammer} \ (2 \ * \ z) \ (2 \ * \ n) \ * \)
\( (2 \ * \ (z + \text{of-nat } n)) \ * \ (2 \ * \ (z + \text{of-nat } n) + 1) \)
by (simp-all add: pochhammer-rec' ac-simps)
also note Suc
also have \( \text{of-nat} \ (2^*(2*n)) \ * \ \text{pochhammer } z \ n \ * \ \text{pochhammer} \ (z+1/2) \ n \ * \)
\( (2 \ * \ (z + \text{of-nat } n)) \ * \ (2 \ * \ (z + \text{of-nat } n) + 1) = \)
lemma fact-double:
fact (2 * n) = (2 ^ (2 * n)) * pochhammer (1 / 2) n * fact n :: 'a::field-char-0
using pochhammer-double[of 1/2::'a n] by (simp add: pochhammer-fact)

lemma pochhammer-absorb-comp: (r - of-nat k) * pochhammer (-r) k = r *
pochhammer (-r + 1) k
(is ?lhs = ?rhs)
for r :: 'a::comm-ring-1
proof
  have ?lhs = - pochhammer (-r) (Suc k)
    by (subst pochhammer-rec) (simp add: algebra-simps)
  also have ... = ?rhs
    by (subst pochhammer-rec) simp
finally show ?thesis .
qed

90.3 Misc

lemma fact-code [code]:
fact n = (of-nat (fold-atLeastAtMost-nat ((*)) 2 n 1) :: 'a::semiring-char-0)
proof
  have fact n = (of-nat (Π {1..n}) :: 'a)
    by (simp add: fact-prod)
  also have Π {1..n} = Π {2..n}
    by (intro prod.mono-neutral-right) auto
  also have ... = fold-atLeastAtMost-nat ((*)) 2 n 1
    by (simp add: prod-atLeastAtMost-code)
finally show ?thesis .
qed

lemma pochhammer-code [code]:
pochhammer a n =
  (if n = 0 then 1
   else fold-atLeastAtMost-nat (λn acc. (a + of-nat n) * acc) 0 (n - 1) 1)
by (cases n)
by (simp-all add: pochhammer-prod prod-atLeastAtMost-code [symmetric]
atLeastLessThanSuc-atLeastAtMost)

end

91 Binomial Coefficients and Binomial Theorem

theory Binomial
imports Presburger Factorial

begin

91.1 Binomial coefficients

This development is based on the work of Andy Gordon and Florian Kam-mueller.

Combinatorial definition

definition binomial :: nat ⇒ nat ⇒ nat (infixl choose 65)
where n choose k = card {K ∈ Pow {0..<n}. card K = k}

theorem n-subsets:
assumes finite A
shows card {B. B ⊆ A ∧ card B = k} = card A choose k
proof
− from assms obtain f where bij: bij-betw f {0..<card A} A
  by (blast dest: ex-bij-betw-nat-finite)
then have [simp]: card (image f ' C) = card C if C ⊆ {0..<card A} for C
  by (meson bij-betw-imp-inj-on bij-betw-subset card-image that)
from bij have bij-betw (image f) (Pow {0..<card A}) (Pow A)
  by (rule bij-betw-Pow)
then have inj-on (image f) (Pow {0..<card A})
  by (rule bij-betw-imp-inj-on)
moreover have {K. K ⊆ {0..<card A} ∧ card K = k} ⊆ Pow {0..<card A}
  by auto
ultimately have inj-on (image f) {K. K ⊆ {0..<card A} ∧ card K = k}
  by (rule inj-on-subset)
then have card (image f ' {K. K ⊆ {0..<card A} ∧ card K = k}) (is - = card ?C)
  by (simp add: card-image)
also have ?C = {K. K ⊆ f ' {0..<card A} ∧ card K = k}
  by (auto elim: subset-imageE)
also have f ' {0..<card A} = A
  by (meson bij bij-betw-def)
finally show ?thesis
  by (simp add: binomial-def)
qed

Recursive characterization

lemma binomial-n-0 [simp]: n choose 0 = 1
proof −
  have {K ∈ Pow {0..<n}. card K = 0} = {{}}
    by (auto dest: finite-subset)
  then show ?thesis
    by (simp add: binomial-def)
qed

lemma binomial-0-Suc [simp]: 0 choose Suc k = 0
THEORY “Binomial”

by (simp add: binomial-def)

lemma binomial-Suc-Suc [simp]: Suc n choose Suc k = (n choose k) + (n choose Suc k)
proof –
  let ?P = λn k. {K. K ⊆ {0..<n} ∧ card K = k}
  let ?Q = ?P (Suc n) (Suc k)
  have inj: inj-on (insert n) (?P n k)
    by rule (auto; metis atLeastLessThan-iff insert-iff less-irrefl subsetCE)
  have disjoint: insert n ⋂ ?P n k ∩ ?P n (Suc k) = {} by auto
  have ?Q = \{K ∈ ?Q. n ∈ K\} ∪ \{K ∈ ?Q. n \∉ K\}
    by auto
  also have \{K ∈ ?Q. n ∈ K\} = insert n ⋂ ?P n k (is ?A = ?B)
    proof (rule set-eqI)
      fix K
      have K-finite: finite K if K ⊆ insert n {0..<n}
        using that by (rule finite-subset) simp-all
      have Suc-card-K: Suc (card K − Suc 0) = card K if n ∈ K
        and finite K
      proof –
        from \{n ∈ K\} obtain L where K = insert n L and n \∉ L
          by (blast elim: Set.set-insert)
        with that show ?thesis by (simp add: card-insert)
      qed
      show K ∈ ?A ↔ K ∈ ?B
        by (subst in-image-insert-iff)
        (auto simp add: card-insert subset-eq-atLeast0-lessThan-finite
          Diff-subset-conv K-finite Suc-card-K)
    qed
  also have \{K ∈ ?Q. n \∉ K\} = ?P n (Suc k)
    by (auto simp add: atLeast0-lessThan-Suc)
  finally show ?thesis using inj disjoint
    by (simp add: binomial-def card-Un-disjoint card-image)
  qed

lemma binomial-eq-0: n < k ⇒ n choose k = 0
by (auto simp add: binomial-def dest: subset-eq-atLeast0-lessThan-card)

lemma zero-less-binomial: k ≤ n ⇒ n choose k > 0
by (induct n k rule: diff-induct) simp-all

lemma binomial-eq-0-iff [simp]: n choose k = 0 ⇐ n < k
by (metis binomial-eq-0 iff less-numeral-extra(3) not-less zero-less-binomial)

lemma zero-less-binomial-iff [simp]: n choose k > 0 ⇐ k ≤ n
by (metis binomial-eq-0-iff not-less0 not-less zero-less-binomial)

lemma binomial-n-n [simp]: n choose n = 1
by (induct n) (simp-all add: binomial-eq-0)

lemma binomial-Suc-n [simp]: Suc n choose n = Suc n
  by (induct n) simp-all

lemma binomial-1 [simp]: n choose Suc 0 = n
  by (induct n) simp-all

lemma choose-reduce-nat:
  \( \theta < n \implies 0 < k \implies \binom{n}{k} = (\binom{n - 1}{k - 1}) + (\binom{n - 1}{k}) \)
  using binomial-Suc-Suc [of n - 1 k - 1] by simp

lemma Suc-times-binomial-eq: Suc n \ast (n choose k) = (Suc n choose Suc k) \ast Suc k
  apply (induct n arbitrary: k)
  apply simp
  apply arith
  apply (case-tac k)
  apply (auto simp add: add-mult-distrib add-mult-distrib2 le-Suc-eq binomial-eq-0)
  done

lemma binomial-le-pow2: n choose k \le 2^n
  apply (induct n arbitrary: k)
  apply (case-tac k)
  apply simp-all
  apply (auto simp add: add-le-mono mult-2)
  done

The absorption property.

lemma Suc-times-binomial: Suc k \ast (Suc n choose Suc k) = Suc n \ast (n choose k)
  using Suc-times-binomial-eq by auto

This is the well-known version of absorption, but it’s harder to use because of the need to reason about division.

lemma binomial-Suc-Suc-eq-times: (Suc n choose Suc k) = (Suc n \ast (n choose k))
  div Suc k
  by (simp add: Suc-times-binomial-eq del: mult-Suc mult-Suc-right)

Another version of absorption, with \(-1\) instead of \(Suc\).

lemma times-binomial-minus1-eq: \( \theta < k \implies k \ast (n choose k) = n \ast ((n - 1) choose (k - 1)) \)
  using Suc-times-binomial-eq [where n = n - 1 and k = k - 1]
  by (auto split: nat-diff-split)
91.2 The binomial theorem (courtesy of Tobias Nipkow):

Avigad's version, generalized to any commutative ring

\textbf{theorem} binomial-ring: \((a + b :: 'a::comm-semiring-1) ^ n =
\sum_{k \leq n. \text{of-nat} \ (n \ choose \ k)} a \cdot k \cdot b \cdot (n-k)\)

\textbf{proof} (induct \(n\))

\texttt{case 0}

\texttt{then show ?case by simp}

\texttt{next}

\texttt{case (Suc \(n\))}

\texttt{have \(\text{decomp}: \{0..n+1\} = \{0\} \cup \{n + 1\} \cup \{1..n\}\)}

\texttt{by auto}

\texttt{have \(\text{decomp2}: \{0..n\} = \{0\} \cup \{1..n\}\)}

\texttt{by auto}

\texttt{have \((a + b) ^ (n+1) = (a + b) \cdot (\sum_{k \leq n. \text{of-nat} \ (n \ choose \ k)} a \cdot k \cdot b \cdot (n-k))\)}

\texttt{using Suc.hyps by simp}

\texttt{also have \(\ldots = a \cdot (\sum_{k \leq n. \text{of-nat} \ (n \ choose \ k)} a \cdot k \cdot b \cdot (n-k)) +
\sum_{k \leq n. \text{of-nat} \ (n \ choose \ k)} a \cdot k \cdot b \cdot (n-k)\)}

\texttt{by (rule distrib-right)}

\texttt{also have \(\ldots = (\sum_{k \leq n. \text{of-nat} \ (n \ choose \ k)} a \cdot (k+1) \cdot b \cdot (n-k)) +
(\sum_{k \leq n. \text{of-nat} \ (n \ choose \ k)} a \cdot k \cdot b \cdot (n-k+1))\)}

\texttt{by (auto simp add: sum-distrib-left ac-simps)}

\texttt{also have \(\ldots = (\sum_{k \leq n. \text{of-nat} \ (n \ choose \ k)} a \cdot k \cdot b \cdot (n + 1 - k)) +
(\sum_{k=1..n+1. \text{of-nat} \ (n \ choose \ (k-1))} a \cdot k \cdot b \cdot (n + 1 - k))\)}

\texttt{by (simp add: atMost-atLeast0 sum.shift-bounds-cl-Suc-ivl Suc-diff-le field-simps del: \(\text{sum.cl-ivl-Suc}\))}

\texttt{also have \(\ldots = b \cdot (n + 1) +
(\sum_{k=1..n. \text{of-nat} \ (n \ choose \ k)} a \cdot k \cdot b \cdot (n + 1 - k)) + (a \cdot (n + 1) +
(\sum_{k=1..n. \text{of-nat} \ (n \ choose \ (k-1))} a \cdot k \cdot b \cdot (n + 1 - k)))\)}

\texttt{using sum.nat-ivl-Suc' [of 1 n \(\ldots\)] of-nat \(\text{of-nat} \ (n \ choose \ (k-1)) \cdot a \cdot k \cdot b \cdot (n + 1 - k)\)}

\texttt{by (simp add: sum.Atleast-Suc-atMost atMost-atLeast0)\)}

\texttt{also have \(\ldots = a \cdot (n + 1) + b \cdot (n + 1) +
(\sum_{k=1..n. \text{of-nat} \ (n + 1 \ choose \ k)} a \cdot k \cdot b \cdot (n + 1 - k))\)}

\texttt{by (auto simp add: field-simps sum.distrib [symmetric] choose-reduce-nat)}

\texttt{also have \(\ldots = (\sum_{k \leq n+1. \text{of-nat} \ (n + 1 \ choose \ k)} a \cdot k \cdot b \cdot (n + 1 - k))\)}

\texttt{using decomp by (simp add: atMost-atLeast0 field-simps)\)}

\texttt{finally show ?case}

\texttt{by simp}

\texttt{qed}

Original version for the naturals.

\textbf{corollary} binomial: \((a + b :: nat) ^ n = (\sum_{k \leq n. \text{of-nat} \ (n \ choose \ k)} a \cdot k \cdot b \cdot (n-k))\)

\texttt{using binomial-ring [of int a int b n]}\)

lemma binomial-fact-lemma: \( k \leq n \implies \text{fact } k \times \text{fact } (n - k) \times (n \choose k) = \text{fact } n \)

proof (induct \( n \) arbitrary: \( k \) rule: nat-less-induct)

fix \( n \) \( k \)
assume \( H: \forall m < n. \forall x \leq m. \text{fact } x \times \text{fact } (m - x) \times (m \choose x) = \text{fact } m \)
assume \( kn: k \leq n \)
let \( \text{ths} = \text{fact } k \times \text{fact } (n - k) \times (n \choose k) = \text{fact } n \)
consider \( n = 0 \lor k = 0 \lor n = k \mid m \ h \) where \( n = \text{Suc } m \ k = \text{Suc } h \ h < m \)
using \( kn \) by atomize-elim presburger
then show \( \text{fact } k \times \text{fact } (n - k) \times (n \choose k) = \text{fact } n \)
proof cases
  case 1
  with \( kn \) show \( \text{thesis} \) by auto
next
case 2
  note \( n = \langle n = \text{Suc } m \rangle \)
  note \( k = \langle k = \text{Suc } h \rangle \)
  note \( hm = \langle h < m \rangle \)
  have \( mn: m < n \)
    using \( n \) by arith
  have \( hm': h \leq m \)
    using \( hm \) by arith
  have \( km: k \leq m \)
    using \( km \ k \ n \ k n \) by arith
  have \( m - h = \text{Suc } (m - \text{Suc } h) \)
    using \( k \ km \ hm \) by arith
  with \( km \ k \) have \( \text{fact } (m - h) = (m - h) \times \text{fact } (m - k) \)
    by simp
  with \( n \ k \) have \( \text{fact } k \times \text{fact } (n - k) \times (n \choose k) = \)
    \( k \times (\text{fact } h \times \text{fact } (m - h) \times (m \choose h)) + \)
    \( (m - h) \times (\text{fact } k \times \text{fact } (m - k) \times (m \choose k)) \)
    by (simp add: field-simps)
  also have \( \ldots = \text{fact } m \)
    using \( H[\text{rule-format}, \text{OF } mn \ hm'] H[\text{rule-format}, \text{OF } mn \ km] \)
    by (simp add: field-simps)
finally show \( \text{thesis} \)
  using \( k \ n \ km \) by simp
qed
qed

lemma binomial-fact':
  assumes \( k \leq n \)
  shows \( n \choose k = \text{fact } n \div (\text{fact } k \times \text{fact } (n - k)) \)
  using binomial-fact-lemma \[OF \ assms\]
  by (metis fact-nonzero mult-eq-0-iff nonzero-mult-div-cancel-left)

lemma binomial-fact:
  assumes \( kn: k \leq n \)
THEORY “Binomial”

shows (of-nat (n choose k :: 'a::field-char-0) = fact n / (fact k * fact (n - k)))
using binomial-fact-lemma[OF kn]
apply (simp add: field-simps)
apply (metis mult.commute of-nat-fact of-nat-mult)
done

lemma fact-binomial:
  assumes k ≤ n
  shows fact k * of-nat (n choose k :: 'a::field-char-0) = (fact n / fact (n - k) :: 'a::field-char-0)
unfolding binomial-fact [OF assms] by (simp add: field-simps)

lemma choose-two: n choose 2 = n * (n - 1) div 2
proof (cases n ≥ 2)
  case False
  then have n = 0 ∨ n = 1
    by auto
  then showthesis by auto
next
  case True
  define m where m = n - 2
  with True have fact n = n * (n - 1) * fact (n - 2)
    by (simp add: fact-prod-Suc atLeast0-lessThan-Suc algebra-simps)
  with True show thesis by ( simp add: binomial-fact )
qed

lemma choose-row-sum: (∑ k≤n. n choose k) = 2^n
using binomial [of 1 1 n] by (simp add: numeral-2-eq-2)

lemma sum-choose-lower: (∑ k≤n. (r+k) choose k) = Suc (r+n) choose n
  by (induct n) auto

lemma sum-choose-upper: (∑ k≤n. k choose m) = Suc n choose Suc m
  by (induct n) auto

lemma choose-alternating-sum:
  n > 0 ⇒ (∑ i≤n. (-1)^i * of-nat (n choose i)) = (0 :: 'a::comm-ring-1)
using binomial-ring[of -1 :: 'a 1 n]
by (simp add: atLeast0AtMost mult-of-nat-commute zero-power)

lemma choose-even-sum:
  assumes n > 0
  shows 2 * (∑ i≤n. if even i then of-nat (n choose i) else 0) = (2 ^ n :: 'a::comm-ring-1)
proof
  have 2 ^ n = (∑ i≤n. of-nat (n choose i)) + (∑ i≤n. (-1) ^ i * of-nat (n choose i) :: 'a)
  moreover
using choose-row-sum[of n]
by (simp add: choose-alternating-sum assms atLeast0AtMost of-nat-sum[symmetric])
also have \ldots = (\sum i\leq n. of-nat (n choose i)) + (-1) ^ i * of-nat (n choose i))
  by (simp add: sum-distrib)
also have \ldots = 2 \ast (\sum i\leq n. if even i then of-nat (n choose i) else 0)
by (subth sum-distrib-left, intro sum.cong) simp-all
finally show \?thesis ..
qed

lemma choose-odd-sum:
  assumes n > 0
  shows 2 \ast (\sum i\leq n. if odd i then of-nat (n choose i) else 0) = (2 ^ n :: 'a::comm-ring-1)
proof –
  have 2 ^ n = (\sum i\leq n. of-nat (n choose i)) − (\sum i\leq n. (-1) ^ i * of-nat (n choose i)):: 'a
    using choose-row-sum[of n]
  by simp add: sum-alternating-sum assms atLeast0AtMost of-nat-sum[symmetric]
  also have \ldots = \ldots by (simp add: sum-subtractf)
  also have \ldots = 2 \ast (\sum i\leq n. if odd i then of-nat (n choose i) else 0)
  by (subst sum-distrib-left, intro sum.cong) simp-all
finally show \?thesis ..
qed

NW diagonal sum property

lemma sum-choose-diagonal:
  assumes m \leq n
  shows (\sum k\leq m. (n − k) choose (m − k)) = Suc n choose m
proof –
  have (\sum k\leq m. (n − k) choose (m − k)) = (\sum k\leq m. (n − m + k) choose k)
    using sum.atLeastAtMost-rev [of \lambda k. (n − k) choose (m − k) 0 m] assms
  by (simp add: sum-atLeastAtMost0)
  also have \ldots = Suc (n − m + m) choose m
  by (rule sum-choose-lower)
  also have \ldots = Suc n choose m
  using assms by simp
finally show \?thesis .
qed

91.3 Generalized binomial coefficients

definition gbinomial :: 'a::{semidom-divide,semiring-char-0} ⇒ nat ⇒ 'a (infixl gchoose 65)
  where gbinomial-prod-rev: a gchoose k = prod (\lambda i. a − of-nat i) \{0..<k\} div fact k

lemma gbinomial-0 [simp]:
a gchoose 0 = 1
\begin{verbatim}
0 \text{gchoose (Suc } k) = 0 
by (simp-all add: gbinomial-prod-rev prod.atLeast0-lessThan-Suc-shift del: prod.op-ivel-Suc)

lemma gbinomial-Suc: a \text{gchoose (Suc } k) = prod (\lambda i. a - of-nat i) \{0..k\} \text{ div fact (Suc } k)
by (simp add: gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost)

lemma gbinomial-1 [simp]: a \text{gchoose 1} = a
by (simp add: gbinomial-prod-rev lessThan-Suc)

lemma gbinomial-Suc0 [simp]: a \text{gchoose Suc } 0 = a
by (simp add: gbinomial-prod-rev lessThan-Suc)

lemma gbinomial-mult-fact: fact k * (a \text{gchoose k}) = (\prod_{i=0<..k} a - of-nat i)
for a :: 'a::field-char-0
by (simp-all add: gbinomial-prod-rev field-simps)

lemma gbinomial-mult-fact': (a \text{gchoose k}) * fact k = (\prod_{i=0<..k} a - of-nat i)
for a :: 'a::field-char-0
using gbinomial-mult-fact [of k a] by (simp add: ac-simps)

lemma gbinomial-pochhammer: a \text{gchoose k} = (\text{−1)}^k * \text{pochhammer (− a)} k / \text{fact k}
for a :: 'a::field-char-0
proof (cases k)
  case (Suc k')
  then show ?thesis
  apply (simp add: pochhammer-minus)
  done
qed auto

lemma gbinomial-pochhammer': a \text{gchoose k} = \text{pochhammer (a - of-nat k + 1)} k / \text{fact k}
for a :: 'a::field-char-0
proof
  have a \text{gchoose k} = ((\text{−1)}^k * (\text{−1)}^k) * \text{pochhammer (a - of-nat k + 1)} k / \text{fact k}
  by (simp add: gbinomial-pochhammer pochhammer-minus mult-ac)
  also have (\text{−1)}^k * (\text{−1)}^k = 1
  by (subst power-add [symmetric]) simp
  finally show ?thesis
  by simp
qed

lemma gbinomial-binomial: n \text{gchoose k} = n \text{ choose k}
\end{verbatim}
proof (cases k ≤ n)
  case False
  then have n < k
    by (simp add: not-le)
  then have 0 ∈ ((−) n) · {0..<k}
    by auto
  then have prod ((−) n) {0..<k} = 0
    by (auto intro: prod-zero)
  with (n < k) show ?thesis
    by (simp add: binomial-eq-0 gbinomial-prod-rev prod-zero)
  next
  case True
  from True have *: prod ((−) n) {0..<k} = ∏ {Suc (n − k)..<n}
    by (intro prod.reindex_bij_witness[of − λi. n − i λi. n − i]) auto
  from True have \( \binom{n}{k} = \frac{\text{fact } n}{\text{fact } k \cdot \binom{n - k}{n - k}} \)
    by (rule binomial-fact')
  with * show ?thesis
    by (simp add: gbinomial-prod-rev mult.commute[of fact k] div_mult2_eq fact_div_fact)
qed

lemma of-nat-gbinomial: of-nat (n\nchoose k) = (of-nat n \nchoose k :: 'a::field-char-0)
proof (cases k ≤ n)
  case False
  then show ?thesis
    by (simp add: not-le gbinomial-binomial binomial-eq-0 gbinomial-prod-rev)
  next
  case True
  define m where m = n − k
  with True have n: n = m + k
    by arith
  from n have fact n = (\(\prod i = 0..<m + k. \text{of-nat } (m + k − i)\) :: 'a)
    by (simp add: fact-prod-rev)
  also have \(\cdots = (\prod i\in\{0..<k\} \cup \{k..<m + k\}. \text{of-nat } (m + k − i)) :: 'a\)
    by (simp add: ivl_disj_un)
  finally have fact n = (fact m * (\(\prod i = 0..<k. \text{of-nat } m + \text{of-nat } k − of-nat i\) :: 'a)
  using prod.shift_bounds_nat_ivl [of λi. of-nat (m + k − i) :: 'a 0 k m]
    by (simp add: fact-prod-rev[of m] prod.union_disjoint of_nat_diff)
  then have fact n / fact (n − k) = (\(\prod i = 0..<k. \text{of-nat } n − \text{of-nat } i\) :: 'a)
    by (simp add: n)
  with True have fact k * of-nat (n\nchoose k) = (fact k * (of-nat n\nchoose k) :: 'a)
  by (simp only: gbinomial-mult-fact[of k of-nat n] gbinomial-binomial[of n k] fact-binomial)
  then show ?thesis
    by simp
qed

lemma binomial-gbinomial: of-nat (n\nchoose k) = (of-nat n\nchoose k :: 'a::field-char-0)
setup (simp add: gbinomial-binomial [symmetric] of-nat-gbinomial)

lemma gbinomial-mult-1:
  fixes a :: 'a::field-char-0
  shows a * (a choose k) = of-nat k * (a choose k) + of-nat (Suc k) * (a choose (Suc k))
  (is \l = \r)
proof -
  have \r = ((- 1) \choose k * pochhammer (- a) k / fact k) * (of-nat k - (- a + of-nat k))
    apply (simp only: gbinomial-pochhammer pochhammer-Suc right-diff-distrib power-Suc)
    apply (simp del: of-nat-Suc fact-Suc)
    apply (auto simp add: field-simps simp del: of-nat-Suc)
    done
  also have \ldots = \l
    by (simp add: field-simps gbinomial-pochhammer)
  finally show \thesis ..
qed

lemma gbinomial-mult-1':
  (a choose k) * a = of-nat k * (a choose k) + of-nat (Suc k) * (a choose (Suc k))
  for a :: 'a::field-char-0
  by (simp add: mult.commute gbinomial-mult-1)

lemma gbinomial-Suc-Suc: (a + 1) choose (Suc k) = a choose k + (a choose (Suc k))
  for a :: 'a::field-char-0
proof (cases k)
  case 0
  then show \thesis by simp
next
  case (Suc h)
  have eq0: (\prod i\in{1..k}. (a + 1) - of-nat i) = (\prod i\in{0..h}. a - of-nat i)
    apply (rule prod.reindex-cong [where l = Suc])
    using Suc
    apply (auto simp add: image-Suc-atMost)
    done
  have fact (Suc k) * (a choose k + (a choose (Suc k))) =
    (a choose Suc h) * (fact (Suc (Suc h))) +
    (a choose Suc (Suc h)) * (fact (Suc (Suc h)))
    by (simp add: Suc field-simps del: fact-Suc)
  also have \ldots =
    (a choose Suc h) * of-nat (Suc (Suc h) * fact (Suc h)) + (\prod i=0..Suc h. a -
of-nat i)
  apply (simp del: fact-Suc prod.op-ivl-Suc add: gbinomial-mult-fact field-simps
          mult.left-commute [of - 2])
  apply (simp del: fact-Suc add: fact-Suc [of Suc h] field-simps gbinomial-mult-fact
          mult.left-commute [of - 2] atLeastLessThanSuc-atLeastAtMost)
  done
also have ... =
  (fact (Suc h) * (a gchoose Suc h)) * of-nat (Suc (Suc h)) + (Π i=0..Suc h. a
  - of-nat i)
  by (simp only: fact-Suc mult.commute mult.left-commute of-nat-fact of-nat-id
       of-nat-mult)
also have ... =
  of-nat (Suc (Suc h)) * (Π i=0..Suc h. a - of-nat i) + (Π i=0..Suc h. a - of-nat
  i)
  unfolding gbinomial-mult-fact atLeastLessThanSuc-atLeastAtMost by auto
also have ... =
  (Π i=0..Suc h. a - of-nat i) + (of-nat h * (Π i=0..h. a - of-nat i)) + 2 *
  (Π i=0..h. a - of-nat i))
  by (simp add: field-simps)
also have ... =
  ((a gchoose Suc h) * (fact (Suc h)) * of-nat (Suc k)) + (Π i∈{0..Suc h}. a -
  of-nat i)
  unfolding gbinomial-mult-fact’
  by (simp add: comm-semiring-class.distrib field-simps Suc atLeastLessThanSuc-atLeastAtMost)
also have ... = (Π i∈{0..h}. a - of-nat i) * (a + 1)
  unfolding gbinomial-mult-fact’ atLeast0-atMost-Suc
  by (simp add: field-simps Suc atLeastLessThanSuc-atLeastAtMost)
also have ... = (Π i∈{0..k}. (a + 1) - of-nat i)
  using eq0
  by (simp add: Suc prod.atLeast0-atMost-Suc-shift del: prod.cl-ivl-Suc)
also have ... = (fact (Suc k)) * ((a + 1) gchoose (Suc k))
  by (simp only: gbinomial-mult-fact atLeastLessThanSuc-atLeastAtMost)
finally show ?thesis
  using fact-nonzero [of Suc k] by auto
qed

lemma gbinomial-reduce-nat: 0 < k ==> a gchoose k = (a - 1) gchoose (k - 1)
+ ((a - 1) gchoose k)
for a :: 'a::field-char-0
by (metis Suc-pred' diff-add-cancel gbinomial-Suc-Suc)

lemma gchoose-row-sum-weighted:
  (∑ k = 0..m. (r gchoose k) * (r/2 - of-nat k)) = of-nat(Suc m) / 2 * (r gchoose
(Suc m))
for r :: 'a::field-char-0
by (induct m) (simp-all add: field-simps distrib gbinomial-mult-1)

lemma binomial-symmetric:
  assumes kn: k ≤ n
shows \( n \choose k = n \choose (n - k) \)

proof –

have \( kn' \colon n - k \leq n \)
  using \( kn \) by arith

from binomial-fact-lemma[OF \( kn \)] binomial-fact-lemma[OF \( kn' \)]

have \( \text{fact } k \times \text{fact } (n - k) \times (n \choose k) = \text{fact } (n - k) \times \text{fact } (n - (n - k)) \)
  by simp

then show \(?thesis\)
  using \( kn \) by simp

qed

lemma choose-rising-sum:

\[
\sum_{j \leq m} ((n + j) \choose n) = ((n + m + 1) \choose (n + 1)) \\
\sum_{j \leq m} ((n + j) \choose n) = ((n + m + 1) \choose m)
\]

proof –

show \( \sum_{j \leq m} ((n + j) \choose n) = ((n + m + 1) \choose (n + 1)) \)
  by (induct \( m \)) simp-all

also have \( \ldots = (n + m + 1) \choose m \)
  by (subst binomial-symmetric) simp-all

finally show \( \sum_{j \leq m} ((n + j) \choose n) = (n + m + 1) \choose m \).

qed

lemma choose-linear-sum: \( \sum_{i \leq n} i \times (n \choose i) = n \times 2^{n - 1} \)

proof (cases \( n \))

  case 0
  then show \(?thesis\) by simp

next

  case (Suc \( m \))

  have \( \sum_{i \leq n} i \times (n \choose i) = \sum_{i \leq Suc \( m \)} i \times (Suc \( m \) \choose i) \)
    by (simp add: Suc)

  also have \( \ldots = Suc \( m \) \times 2^{\cdot \cdot} \)
    unfolding sum.atMost-Suc-shift Suc-times-binomial sum-distrib-left[symmetric]
    by (simp add: choose-row-sum)

  finally show \(?thesis\)
    using Suc by simp

qed

lemma choose-alternating-linear-sum:

  assumes \( n \neq 1 \)

  shows \( \sum_{i \leq n} (-1)^i \times \text{of-nat } i \times \text{of-nat } (n \choose i) :: 'a::comm-ring-1 = 0 \)

proof (cases \( n \))

  case 0
  then show \(?thesis\) by simp

next

  case (Suc \( m \))

  with assms have \( m > 0 \)
    by simp

  have \( \sum_{i \leq n} (-1)^i \times \text{of-nat } i \times \text{of-nat } (n \choose i) :: 'a) = \)

THEORY “Binomial” 1665

\[
\sum_{i \leq \text{Suc } m} (-1)^i \cdot \text{of-nat } i \cdot \text{of-nat } (\text{Suc } m \text{ choose } i)
\]
by (simp add: Suc)

also have \ldots \(= \sum_{i \leq m} (-1)^i \cdot \text{of-nat } (\text{Suc } i \text{ choose } \text{Suc } i))
by (simp only: sum.atMost-Suc-shift sum-distrib-left[symmetric] mult-ac of-nat-mult)

also have \ldots \(= - \text{of-nat } (\text{Suc } m) \cdot (\sum_{i \leq m} (-1)^i \cdot \text{of-nat } (\text{Suc } m \text{ choose } i))) = 0
using choose-alternating-sum[OF \langle m \rangle]
by simp

finally show \(?thesis
by simp
qed

lemma vandermonde: \(\sum_{k \leq r} (\text{Suc } k \text{ choose } k) \cdot (n \text{ choose } (r - k))) = (m + n) \text{ choose } r
proof (induct n arbitrary: r)
case 0
have \(= \sum_{k \leq r} \text{if } k = r \text{ then } (m \text{ choose } k) \text{ else } 0)
by (intro sum.cong) simp-all

also have \ldots \(= n \text{ choose } r
by simp

finally show \(?case
by simp
qed

next
case (Suc n r)
show \(?case
by (cases r) (simp-all add: Suc [symmetric] algebra-simps sum.distrib Suc-diff-le)
qed

lemma choose-square-sum: \(\sum_{k \leq n} (n \text{ choose } k) \cdot 2) = (2 \cdot n) \text{ choose } n
using vandermonde[of n n n]
by (simp add: power2-eq-square mult-2 binomial-symmetric [symmetric])

lemma pochhammer-binomial-sum:
fixes a b :: 'a::comm-ring-1
shows \(\sum_{k \leq \text{Suc } n} \text{of-nat } (\text{Suc } n \text{ choose } k) \cdot \text{pochhammer } a k \cdot \text{pochhammer } b (n - k)
proof (induction n arbitrary: a b)
case 0
then show \(?case by simp
next
case (Suc n a b)
have \(= \sum_{i \leq \text{Suc } n} \text{of-nat } (\text{Suc } n \text{ choose } i) \cdot \text{pochhammer } a (\text{Suc } i) \cdot \text{pochhammer } b (n - i)) +
\( ((\sum_{i \leq n. \text{of-nat}} (n \choose \text{Suc } i) \ast \text{pochhammer } a (\text{Suc } i) \ast \text{pochhammer } b (n - i)) + \text{pochhammer } b (\text{Suc } n)) \)

by (subst sum.atMost-Suc-shift) (simp add: ring-distrib sum.distrib)

also have \( (\sum_{i \leq n. \text{of-nat}} (n \choose i) \ast \text{pochhammer } a (\text{Suc } i) \ast \text{pochhammer } b (n - i)) = a \ast \text{pochhammer } ((a + 1) + b) n \)

by (subst Suc) (simp add: sum-distrib-left pochhammer-rec mult-ac)

also have \( (\sum_{i \leq n. \text{of-nat}} (n \choose i) \ast \text{pochhammer } a i \ast \text{pochhammer } b (n - i)) + \text{pochhammer } b (\text{Suc } n) = (\sum_{i=0..\text{Suc } n. \text{of-nat}} (n \choose i) \ast \text{pochhammer } a i \ast \text{pochhammer } b (\text{Suc } n - i)) \)

apply (subst sum.atLeast-Suc-atMost)
apply simp
apply (subst sum.shift-bounds-cl-Suc-ivl)
done

also have \( (\sum_{i \leq n. \text{of-nat}} (n \choose i) \ast \text{pochhammer } a i \ast \text{pochhammer } b (n - i)) = (\sum_{i=0..\text{Suc } n. \text{of-nat}} (n \choose i) \ast \text{pochhammer } a i \ast \text{pochhammer } b (\text{Suc } (n - i))) \)

by (intro sum.cong) (simp-all add: Suc-diff-le)
also have \( (\sum_{i \leq n. \text{of-nat}} (n \choose i) \ast \text{pochhammer } a i \ast \text{pochhammer } b (n - i)) = a \ast \text{pochhammer } ((a + 1) + b) n + b \ast \text{pochhammer } (a + (b + 1)) n = \text{pochhammer } (a + b) (\text{Suc } n) \)

by (simp add: pochhammer-rec algebra-simps)
finally show ?case ..

qed

Contributed by Manuel Eberl, generalised by LCP. Alternative definition of the binomial coefficient as \( \prod_{i<k. \text{of-nat}} (n - i) / (k - i) \).

lemma gbinomial-altdef-of-nat: 
\[ a \text{ gchoose } k = (\prod_{i = 0..<k. \text{of-nat}} (a - i) / (k - i)) \] for \( k :: \text{nat} \) and \( a :: 'a::field-char-0 \)

by (simp add: prod-dividef gbinomial-prod-rev fact-prod-rev)

lemma gbinomial-ge-n-over-k-pow-k: 
fixes \( k :: \text{nat} \)
and \( a :: 'a::linear-order \)
assumes \( \text{of-nat } k \leq a \)
shows \( (a / \text{of-nat } k :: 'a) ^ k \leq a \text{ gchoose } k \)

proof –
have \( x: 0 \leq a \)
using assms of-nat-0-le_iff order-trans by blast
have \( (a / \text{of-nat } k :: 'a) ^ k = (\prod_{i = 0..<k. \text{of-nat}} a / \text{of-nat } k :: 'a) \)
by simp
also have ... ≤ a \choose k
unfolding gbinomial-altdef-of-nat
apply (safe intro!: prod-mono)
apply simp-all
prefer 2
subgoal premises for i
proof –
  from assms have a \cdot of-nat i ≥ of-nat (i \cdot k)
    by (metis mult.commute mult-cancel-right of-nat-less-0_iff of-nat-mult)
  then have a \cdot of-nat k - a \cdot of-nat i ≤ a \cdot of-nat k - of-nat (i \cdot k)
    by arith
  then have a \cdot of-nat k - a \cdot of-nat i ≤ of-nat k
    by (simp only: of-nat-mult [symmetric] of-nat-le_iff)
  finally show ?thesis
qed

lemma gbinomial-negated-upper: (a \choose k) = (-1) ^ k \cdot ((of-nat k - a - 1) \choose k)
  by (simp add: gbinomial-pochhammer pochhammer-minus algebra-simps)

lemma gbinomial-minus: ((-a) \choose k) = (-1) ^ k \cdot ((a + of-nat k - 1) \choose k)
  by (subst gbinomial-negated-upper) (simp add: add-ac)

lemma Suc-times-gbinomial: of-nat (Suc k) \cdot ((a + 1) \choose (Suc k)) = (a + 1) \cdot (a \choose k)
proof (cases k)
  case 0
  then show ?thesis by simp
next
  case (Suc b)
  then have ((a + 1) \choose (Suc (Suc b))) = (\prod i = 0..Suc b. a + (1 - of-nat i)) / fact (b + 2)
    by (simp add: field-simps gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost)
  also have (\prod i = 0..Suc b. a + (1 - of-nat i)) = (a + 1) \cdot (\prod i = 0..b. a - of-nat i)
    by (simp add: prod.atLeast0-atMost-Suc-shift del: prod.cl_ivl_Suc)
  also have ... / fact (b + 2) = (a + 1) / of-nat (Suc (Suc b)) \cdot (a \choose Suc b)
    by (simp-all add: gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost)
  finally show ?thesis by (simp add: Suc field-simps del: of-nat-Suc)
qed

lemma gbinomial-factors: ((a + 1) gchoose (Suc k)) = (a + 1) / of-nat (Suc k) * (a gchoose k)
proof (cases k)
  case 0
  then show ?thesis by simp
next
case (Suc b)
  then have ((a + 1) gchoose (Suc (Suc b))) = (\Pi_{i = 0}^{Suc b}. a + (1 - of-nat i)) / fact (b + 2)
    by (simp add: field-simps gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost)
  also have (\Pi_{i = 0}^{Suc b}. a + (1 - of-nat i)) = (a + 1) * (\Pi_{i = 0..b}. a - of-nat i)
    by (simp add: prod.atLeast0-atMost-Suc-shift del prod.cl-ivl-Suc)
  also have ... / fact (b + 2) = (a + 1) / of-nat (Suc (Suc b)) * (a gchoose Suc b)
    by (simp-all add: gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost atLeast0AtMost)
  finally show ?thesis by (simp add: Suc)
qed

lemma gbinomial-rec: ((a + 1) gchoose (Suc k)) = (a gchoose k) * ((a + 1) / of-nat (Suc k))
  using gbinomial-mult-1[of a k]
  by (subst gbinomial-Suc-Suc) (simp add: field-simps del of-nat-Suc, simp add: algebra-simps)

lemma gbinomial-of-nat-symmetric: k ≤ n \implies (of-nat n) gchoose k = (of-nat n) gchoose (n - k)
  using binomial-symmetric[of k n] by (simp add: binomial-gbinomial [symmetric])

The absorption identity (equation 5.5 [3, p. 157]):
\[ \binom{r}{k} = \frac{r}{k} \binom{r-1}{k-1}, \text{ integer } k \neq 0. \]

lemma gbinomial-absorption': k > 0 \implies a gchoose k = (a / of-nat k) * (a - 1 gchoose (k - 1))
  using gbinomial-rec[of a - 1 k - 1]
  by (simp-all add: gbinomial-rec field-simps del: of-nat-Suc)

The absorption identity is written in the following form to avoid division by k (the lower index) and therefore remove the k \neq 0 restriction [3, p. 157]:
\[ k \binom{r}{k} = r \binom{r-1}{k-1}, \text{ integer } k. \]

lemma gbinomial-absorption: of-nat (Suc k) * (a gchoose Suc k) = a * ((a - 1) gchoose k)
using gbinomial-absorption[of Suc k a] by (simp add: field-simps del: of-nat-Suc)

The absorption identity for natural number binomial coefficients:

lemma binomial-absorption: Suc k * (n choose Suc k) = n * ((n - 1) choose k)
by (cases n) (simp-all add: binomial-eq-0 Suc-times-binomial del: binomial-Suc-Suc mult-Suc)

The absorption companion identity for natural number coefficients, following the proof by GKP [3, p. 157]:

lemma binomial-absorb-comp: (n - k) * (n choose k) = n * ((n - 1) choose k)
(is ?lhs = ?rhs)
proof (cases n ≤ k)
case True
then show ?thesis by auto
next
case False
then have ?rhs = Suc ((n - 1) - k) * (n choose ((n - 1) - k))
using binomial-symmetric[of k n - 1] binomial-absorption[of (n - 1) - k n]
by simp
also have Suc ((n - 1) - k) = n - k
using False by simp
also have n choose ... = n choose k
using False by (intro binomial-symmetric [symmetric]) simp-all
finally show ?thesis ..
qed

The generalised absorption companion identity:

lemma gbinomial-absorb-comp: (a - of-nat k) * (a gchoose k) = a * ((a - 1) gchoose k)
using pochhammer-absorb-comp[of a k] by (simp add: gbinomial-pochhammer)

lemma gbinomial-addition-formula:
a gchoose (Suc k) = ((a - 1) gchoose (Suc k)) + ((a - 1) gchoose k)
using gbinomial-Suc-Suc[of a - 1 k] by (simp add: algebra-simps)

lemma binomial-addition-formula:
0 < n ⇒ n choose (Suc k) = ((n - 1) choose (Suc k)) + ((n - 1) choose k)
by (subst choose-reduce-nat) simp-all

Equation 5.9 of the reference material [3, p. 159] is a useful summation formula, operating on both indices:

\[ \sum_{k \leq n} \binom{r + k}{k} = \binom{r + n + 1}{n}, \text{ integer } n. \]

lemma gbinomial-parallel-sum: \( \sum_{k \leq n} (a + \text{ of-nat } k) \text{ gchoose } k = (a + \text{ of-nat } n + 1) \text{ gchoose } n \)
proof (induct n)
THEORY "Binomial"

| case 0 |
| then show | ?case by simp |
| next |
| case (Suc m) |
| then show | ?case |
| using | gbinomial-Suc-Suc[of (a + of-nat m + 1) m] |
| by (simp add: add-ac) |

qed

91.3.1 Summation on the upper index

Another summation formula is equation 5.10 of the reference material [3, p. 160], aptly named summation on the upper index:

$$\sum_{0 \leq k \leq n} \binom{k}{m} = \binom{n+1}{m+1}, \text{ integers } m, n \geq 0.$$

lemma gbinomial-sum-up-index:

$$\sum_{j = 0\ldots.n.} (of-nat j \text{ gchoose } k) = (of-nat n + 1) \text{ gchoose } (k + 1)$$

proof (induct n)

| case 0 |
| show | ?case |
| using | gbinomial-Suc-Suc[of 0 k] |
| by (cases k) auto |

next

| case (Suc n) |
| then show | ?case |
| using | gbinomial-Suc-Suc[of of-nat (Suc n) :: 'a k] |
| by (simp add: add-ac) |

qed

lemma gbinomial-index-swap:

$$((-1)^k) * ((- (of-nat n) - 1) \text{ gchoose } k) = ((-1)^n) * ((- (of-nat k) - 1) \text{ gchoose } n)$$

(is ?lhs = ?rhs)

proof |

have | ?lhs = (of-nat (k + n) \text{ gchoose } k) |
| by (subst gbinomial-negated-upper) (simp add: power-mult-distrib [symmetric]) |
also have | ... = (of-nat (k + n) \text{ gchoose } n) |
| by (subst gbinomial-of-nat-symmetric) simp-all |
also have | ... = ?rhs |
| by (subst gbinomial-negated-upper) simp |
finally show | ?thesis |

qed

lemma gbinomial-sum-lower-neg:

$$\sum_{k \leq m.} (a \text{ gchoose } k) * (-1)^k = (-1)^m * (a - 1 \text{ gchoose } m)$$

(is ?lhs = ?rhs)
proof  
  have ?lhs = (\( \sum_{k \leq m} -(a + 1) + \text{of-nat } k \; gchoose \; k \)  
by (intro sum.cong[OF refl]) (subst gbinomial-negated-upper, simp add: power-mult-distrib)  
also have \ldots = - a + \text{of-nat } m \; gchoose \; m  
by (subst gbinomial-parallel-sum) simp  
also have \ldots = ?rhs  
by (subst gbinomial-negated-upper) (simp add: power-mult-distrib)  
finally show ?thesis .  
qed

lemma gbinomial-partial-row-sum:  
(\( \sum_{k \leq m} (a \; gchoose \; k) \ast ((a / 2) - \text{of-nat } k) \) = ((\text{of-nat } m + 1)/2) \ast (a \; gchoose \; (m + 1))  
proof (induct m)  
  case 0  
  then show ?case by simp  
next  
  case (Suc mm)  
  then have (\( \sum_{k \leq \text{Suc mm}} (a \; gchoose \; k) \ast ((a / 2) - \text{of-nat } k) \) =  
(a - \text{of-nat } (\text{Suc mm})) \ast (a \; gchoose \; \text{Suc mm}) / 2  
by (simp add: field-simps)  
also have \ldots = a \ast (a - 1 \; gchoose \; \text{Suc mm}) / 2  
by (subst gbinomial-absorb-comp) (rule refl)  
also have \ldots = (\text{of-nat } (\text{Suc mm} + 1)) / 2 \ast (a \; gchoose \; (\text{Suc mm} + 1))  
by (subst gbinomial-absorption [symmetric]) simp  
finally show ?case .  
qed

lemma sum-bounds-lt-plus1: (\( \sum_{k < \text{mm}} f \; (\text{Suc } k) \) = (\( \sum_{k=1..\text{mm}} f \; k \)  
by (induct mm) simp-all

lemma gbinomial-partial-sum-poly:  
(\( \sum_{k \leq m} (\text{of-nat } m + a \; gchoose \; k) \ast x^k \ast y^{(m-k)} \) =  
(\( \sum_{k \leq m} (-a \; gchoose \; k) \ast (-x)^k \ast (x + y)^{(m-k)} \)  
is ?lhs m = ?rhs m)  
proof (induction m)  
  case 0  
  then show ?case by simp  
next  
  case (Suc mm)  
  define G where G i k = (\text{of-nat } i + a \; gchoose \; k) \ast x^k \ast y^{(i - k)} \text{ for } i \; k  
  define S where S = ?lhs  
  have SG-def: S = (\( \lambda i. (\sum_{k \leq i} (G \; i \; k)) \)  
  unfolding S-def G-def ..  
  have S (Suc mm) = G (Suc mm) \theta + (\( \sum_{k=Suc \; 0..\text{Suc mm}} G \; (\text{Suc mm}) \; k \)  
using SG-def by (simp add: sum.atLeast-Suc-atMost atLeast0AtMost [symmetric])  
also have (\( \sum_{k=Suc \; 0..\text{Suc mm}} G \; (\text{Suc mm}) \; k \) = (\( \sum_{k=0..\text{mm}} G \; (\text{Suc mm}) \) (Suc k))
by (subst sum.shift-bounds-cl-Suc-ivl) simp
also have \ldots = (\sum_{k=0}^{\text{mm}}. (\text{of-nat mm} + a \text{ gchoose (Suc k)}) +
\text{of-nat mm} + a \text{ gchoose k}) * x^\prime(\text{Suc k}) \ast y^\prime(\text{mm} - k)) +
(\sum_{k=0}^{\text{mm}}. (\text{of-nat mm} + a \text{ gchoose k}) \ast x^\prime(\text{Suc k}) \ast y^\prime(\text{mm} - k))
unfolding G-def by (subst gbinomial-addition-formula) simp
also have \ldots = (\sum_{k=0}^{\text{mm}}. (\text{of-nat mm} + a \text{ gchoose (Suc k)}) \ast x^\prime(\text{Suc k}) \ast y^\prime(\text{mm} - k)) +
(\sum_{k=0}^{\text{mm}}. (\text{of-nat mm} + a \text{ gchoose k}) \ast x^\prime(\text{Suc k}) \ast y^\prime(\text{mm} - k))
also have (\sum_{k=0}^{\text{mm}}. (\text{of-nat mm} + a \text{ gchoose (Suc k)}) \ast x^\prime(\text{Suc k}) \ast y^\prime(\text{mm} - k)) =
(\sum_{k<\text{Suc mm}}. (\text{of-nat mm} + a \text{ gchoose (Suc k)}) \ast x^\prime(\text{Suc k}) \ast y^\prime(\text{mm} - k))
by (simp only: atLeast0AtMost lessThan-Suc-atMost)
also have \ldots = (\sum_{k<\text{mm}}. (\text{of-nat mm} + a \text{ gchoose Suc k}) \ast x^\prime(\text{Suc k}) \ast y^\prime(\text{mm} - k)) +
(\sum_{k<\text{mm}}. (\text{of-nat mm} + a \text{ gchoose (Suc mm)}) \ast x^\prime(\text{Suc mm})
(is - = ?A + ?B)
by (subst sum.lessThan-Suc) simp
also have ?A = (\sum_{k=1}^{\text{mm}}. (\text{of-nat mm} + a \text{ gchoose k}) \ast x^\prime k \ast y^\prime(\text{mm} - k + 1))
proof (subst sum.bounds-it-plus1 [symmetric], intro sum.cong[OF refl], clarify)
fix k
assume k < mm
then have mm - k = mm - Suc k + 1
by linarith
then show (\text{of-nat mm} + a \text{ gchoose Suc k}) \ast x^\prime Suc k \ast y^\prime (\text{mm} - k) =
(\text{of-nat mm} + a \text{ gchoose Suc k}) \ast x^\prime Suc k \ast y^\prime (\text{mm} - Suc k + 1)
by (simp only:)
qed
also have \ldots + ?B = y \ast (\sum_{k=1}^{\text{mm}}. (G \text{ mm k}) + \text{of-nat mm} + a \text{ gchoose (Suc mm)}) \ast x^\prime(Suc mm)
unfolding G-def by (subst sum.distrib-left) (simp add: algebra-simps)
also have (\sum_{k=0}^{\text{mm}}. (\text{of-nat mm} + a \text{ gchoose k}) \ast x^\prime(\text{Suc k}) \ast y^\prime(\text{mm} - k)) = x \ast (S \text{ mm})
unfolding S-def by (subst sum.distrib-left) (simp add: atLeast0AtMost algebra-simps)
also have (G (Suc mm) 0) = y \ast (G mm 0)
by (simp add: G-def)
finally have S (Suc mm) =
y \ast (G mm 0 + (\sum_{k=1}^{\text{mm}}. (G mm k))) + (\text{of-nat mm} + a \text{ gchoose (Suc mm)}) \ast x^\prime(Suc mm) + x \ast (S \text{ mm})
by (simp add: ring-distribs)
also have G mm 0 + (\sum_{k=1}^{\text{mm}}. (G mm k)) = S mm
by (simp add: sum.atLeast-Suc-atMost[symmetric] S-def atLeast0AtMost)
finally have S (Suc mm) = (x + y) \ast (S \text{ mm}) + (\text{of-nat mm} + a \text{ gchoose (Suc mm)}) \ast x^\prime(Suc mm)
by (simp add: algebra-simps)
also have (\text{of-nat mm} + a \text{ gchoose (Suc mm)}) = (-1) ^\prime (Suc mm) \ast (- a \text{ gchoose (Suc mm))}
by (subst gbinomial-negated-upper) simp
also have \((-1) \cdot \text{Suc } mm \ast (\text{Suc } mm) \ast x \ast (\text{Suc } mm) =
\((-1) \ast \text{Suc } mm \ast \text{Suc } mm\) by (simp add: power-minus[of \(x\)])
also have \((x + y) \ast S mm + \ldots = (x + y) \ast \text{rhs } mm + (\text{Suc } mm) \ast (\text{-x}) \ast (\text{Suc } mm)\) unfolding \(S\)-def by (subst Suc.IH) simp
also have \((x + y) \ast \text{rhs } mm = (\sum \ll n \leq \text{mm.} ((\text{-a} \text{ gchoose } n) \ast (\text{-x}) \ast n) \ast (x + y) \ast (\text{Suc } mm - n))\) by (simp add: Suc-diff-le)
also have \(\ldots + (\text{-a} \text{ gchoose } (\text{Suc } mm)) \ast (\text{-x}) \ast \text{Suc } mm =
(\sum \ll n \leq \text{Suc } mm. (\text{-a} \text{ gchoose } n) \ast (\text{-x}) \ast n) \ast (x + y) \ast (\text{Suc } mm - n))\) by simp
finally show \(?case\) by (simp only: \(S\)-def)

qed

lemma \(\text{binomial-partial-sum-poly-xpos:}\)
\((\sum \ll k \leq \text{m.} \text{(of-nat } m + \text{ a} \text{ gchoose } k) \ast x \ast k \ast y \ast (m-k)) =
(\sum \ll k \leq \text{m.} \text{(of-nat } k + a - 1 \text{ gchoose } k) \ast x \ast k \ast (x + y) \ast (m-k))\)
apply (subst \(\text{binomial-partial-sum-poly}\))
apply (subst \(\text{binomial-negated-upper}\))
apply (intro sum.cong, rule refl)
apply (simp add: power-mult-distrib [symmetric])
done

lemma \(\text{binomial-r-part-sum:} (\sum k \leq m. (2 \ast m + 1 \text{ choose } k)) = 2 ^ (2 \ast m)\)
proof -
  have \(2 \ast 2 ^ (2 \ast m) = (\sum k = 0..(2 \ast m + 1). (2 \ast m + 1 \text{ choose } k))\)
  using choose-row-sum[where \(n=2 \ast m + 1\)] by (simp add: atLeast-atLeast0)
  also have \((\sum k = 0..(2 \ast m + 1). (2 \ast m + 1 \text{ choose } k)) =
(\sum k = 0..m. (2 \ast m + 1 \text{ choose } k)) +
(\sum k = m+1..2*m+1. (2 \ast m + 1 \text{ choose } k))\)
  using sum.ub-add-nat[of \(0 \text{ m} \lambda k. 2 \ast m + 1 \text{ choose } k + 1\)]
  by (simp add: mult-2)
  also have \((\sum k = m+1..2*m+1. (2 \ast m + 1 \text{ choose } k)) =
(\sum k = 0..m. (2 \ast m + 1 \text{ choose } (k + (m + 1)))\)
  by (subst sum.shift-bounds-cl-nat [symmetric]) (simp add: mult-2)
  also have \(\ldots = (\sum k = 0..m. (2 \ast m + 1 \text{ choose } (m - k)))\)
  by (intro sum.cong[OF refl], subst binomial-symmetric) simp-all
  also have \(\ldots = (\sum k = 0..m. (2 \ast m + 1 \text{ choose } k))\)
  using sum.atLeastAtMost-rev[of \(\lambda k. 2 \ast m + 1 \text{ choose } (m - k) 0 m\]
  by simp
  also have \(\ldots + \ldots = 2 \ast \ldots\)
  by simp
  finally show \(?thesis\)
  by (subst (asm) mult-cancel1) (simp add: atLeast0AtMost)
qed

lemma \(\text{binomial-r-part-sum:} (\sum k \leq m. (2 \ast (\text{of-nat } m + 1 \text{ gchoose } k)) = 2 ^ (2 \ast (}
THEORY "Binomial"

* m)
(is ?lhs = ?rhs)

proof –

have ?lhs = of-nat (∑ k≤m. (2 * m + 1) choose k)
  by (simp add: binomial-gbinomial add-ac)
also have ... = of-nat (2 ^ (2 * m))
  by (subst binomial-r-part-sum) (rule refl)
finally show ?thesis by simp
qed

lemma gbinomial-sum-nat-pow2:
(∑ k≤m. (of-nat (m + k) gchoose :: 'a::field-char-0) / 2 ^ k) = 2 ^ m
(is ?lhs = ?rhs)

proof –

have 2 ^ m * 2 ^ m = (2 ^ (2 * m)) :: 'a
  by (induct m) simp-all
also have ... = (∑ k≤m. (2 * (of-nat m) + 1 gchoose k))
  using gbinomial-r-part-sum ..
also have ... = (∑ k≤m. (of-nat (m + k) gchoose k) * 2 ^ (m - k))
  using gbinomial-partial-sum-poly-xpos[where x=1 and y=1 and a=of-nat m + 1 and m=m]
  by (simp add: add-ac)
also have ... = 2 ^ m * (∑ k≤m. (of-nat (m + k) gchoose k) / 2 ^ k)
  by (subst sum-distrib-left) (simp add: algebra-simps power-diff)
finally show ?thesis
  by (subst (asm) mult-left-cancel) simp-all
qed

lemma gbinomial-trinomial-revision:
assumes k ≤ m
shows (a gchoose m) * (of-nat m gchoose k :: 'a::field-char-0) = (a gchoose k) * (a - of-nat k gchoose (m - k))

proof –

have (a gchoose m) * (of-nat m gchoose k) = (a gchoose m) * fact m / (fact k * fact (m - k))
  using assms by (simp add: binomial-gbinomial [symmetric] binomial-fact)
also have ... = (a gchoose k) * (a - of-nat k gchoose (m - k))
  using assms by (simp add: gbinomial-pochhammer power-diff pochhammer-product)
finally show ?thesis .
qed

Versions of the theorems above for the natural-number version of "choose"

lemma binomial-altdef-of-nat:
k ≤ n ⇒ of-nat (n choose k) = (∏ i = 0..<k. of-nat (n - i) / of-nat (k - i)) :: 'a
  for n k :: nat and x :: 'a::field-char-0
  by (simp add: binomial-altdef-of-nat binomial-gbinomial of-nat-diff)

lemma binomial-ge-n-over-k-pow-k: k ≤ n ⇒ (of-nat n / of-nat k :: 'a) ^ k ≤
of-nat (n choose k)
  for k n :: nat and x :: 'a::linordered-field
  by (simp add: gbinomial-ge-n-over-k-pow-k binomial-gbinomial of-nat-diff)

lemma binomial-le-pow:
  assumes r ≤ n
  shows \( \text{choose } n \leq n \div r \)
proof –
  have n choose r ≤ fact n div fact (n - r)
  using assms by (subst binomial-fact-lemma [symmetric]) auto
  with fact-div-fact-le-pow [OF assms] show ?thesis
  by auto
qed

lemma binomial-altdef-nat: k ≤ n ⇒ n choose k = fact n div (fact k * fact (n - k))
  for k n :: nat
  by (subst binomial-fact-lemma [symmetric]) auto

lemma choose-dvd:
  k ≤ n ⇒ fact k * fact (n - k) dvd (fact n :: 'a::linordered-semidom)
  unfolding dvd_def
  apply (rule exI [where x=of-nat (n choose k)])
  using binomial-fact-lemma [of k n, THEN arg-cong [where f = of-nat and 'b'=a]]
  apply auto
  done

lemma fact-fact-dvd-fact:
  fact k * fact n dvd (fact (k + n) :: 'a::linordered-semidom)
  by (metis add.commute add-diff-cancel-left' choose-dvd le-add2)

lemma choose-mult-lemma:
  \( ((m + r + k) \choose (m + k)) * ((m + k) \choose k) = ((m + r + k) \choose k) \)
  * ((m + r) \choose m)\)
  (is ?lhs = -)
proof –
  have ?lhs =
  fact (m + r + k) div (fact (m + k) * fact (m + r - m)) * (fact (m + k)
  div (fact k * fact m))
  by (simp add: binomial-altdef-nat)
  also have... = fact (m + r + k) div (fact r * (fact k * fact m))
  apply (subst div-mult-div-if-dvd)
  apply (auto simp: algebra-simps fact-fact-dvd-fact)
  done
  also have... = (fact (m + r + k) * fact (m + r)) div (fact r * (fact k * fact m) * fact (m + r))
  apply (subst div-mult-div-if-dvd [symmetric])
apply (auto simp add: algebra-simps)
done

also have . . . =
(fact (m + r + k) div (fact k * fact (m + r)) * (fact (m + r) div (fact r * fact m)))
apply (auto simp: fact-fact-dvd-fact dvd-trans nat-mult-cancel-disj)
done

finally show ?thesis
by (simp add: binomial-altdef-nat mult.commute)

qed

The "Subset of a Subset" identity.

lemma choose-mult:
  \( k \leq m \implies m \leq n \implies (n \text{ choose } m) \times (m \text{ choose } k) = (n \text{ choose } k) \times ((n - k) \text{ choose } (m - k)) \)
using choose-mult-lemma [of m-k n-m k] by simp

91.4 More on Binomial Coefficients

lemma choose-one: \( n \text{ choose } I = n \) for \( n :: nat \)
by simp

lemma card-UNION:
  assumes finite A
  and \( \forall k \in A \). finite k
  shows \( \text{card} (\bigcup A) = \text{nat} (\sum I \mid I \subseteq A \land I \neq \{} \). \((-1)^{-} \times (\text{card} I + 1) \times \text{int} (\text{card} (\bigcap I))) \)
(is ?lhs = ?rhs)
proof
  have . . . = \( \text{nat} (\sum I \mid I \subseteq A \land I \neq \{} \). \((-1)^{-} \times (\text{card} I + 1) \times (\sum - \in \bigcap I. 1) \)
  by simp
  also have . . . = \( \text{nat} (\sum I \mid I \subseteq A \land I \neq \{} \). \((-1)^{-} \times (\text{card} I + 1)) \)
  by (subst sum-distrib-left) simp
  also have . . . = \( \text{nat} (\sum (I,) \in \Sigma \{I. I \subseteq A \land I \neq \{\}) \) \text{Inter}. \((-1)^{-} \times (\text{card} I + 1)) \)
  using assms by (subst sum.Sigma) auto
  also have . . . = \( \sum (x, I) \in (\Sigma x: \text{UNIV}. \{I. I \subseteq A \land I \neq \{\} \land x \in I \}). \(-1)^{-} \times (\text{card} I + 1)) \)
  by (rule sum.reindex-cong [where \( l = \lambda(x, y). (y, x) \)]) (auto intro: inj-onI)
  also have . . . = \( \sum (x, I) \in (\Sigma x: \bigcup A. \{I. I \subseteq A \land I \neq \{\} \land x \in I \}). \(-1)^{-} \times (\text{card} I + 1)) \)
  using assms
  by (auto intro!: summono-neutral-cong-right finite-SigmaI2 intro: finite-subset[where \( B = \bigcup A \)])
  also have . . . = \( \sum x \in \bigcup A. (\sum I | I \subseteq A \land I \neq \{\} \land x \in I \). \(-1)^{-} \times (\text{card} I + 1)) \)
using assms by (subst sum.Sigma) auto
also have \( \ldots = (\sum_{i\in A} I) \) (is \( \text{sum } \text{?lhs } - = - \))
proof (rule sum.cong[OF refl])
  fix \( x \)
  assume \( x: x \in \bigcup A \)
  define \( K \) where \( K = \{X \in A. x \in X\} \)
  with \( \text{finite } A \) have \( K: \text{finite } K \)
  by auto
  let \( \lambda I = \lambda i. \{I. I \subseteq A \land \text{card } I = i \land x \in \bigcap I\} \)
  have inj-on snd (SIGMA i:{1..card A}. \?I i )
  using assms by (auto intro!: inj-on)
  moreover have \([\text{symmetric}]: \text{snd } (\text{SIGMA } i:{1..card } A). \?I i ) = \{I. I \subseteq A \land I \neq \emptyset \}\land x \in \bigcap I\)
  using assms
  by (auto intro!: rev-image-eqI[where \( x=(\text{card } a, a) \) for a]
      simp add: card-at-0-iff[folded Suc-le-eq]
      dest: finite-subset intro: card-mono)
  ultimately have \( \text{?lhs } x = (\sum_{i, I\in (\text{SIGMA } i:{1..card } A). \?I i ). (-1) \cdot (i + 1)) \)
  by (rule sum.reindex-cong [where \( l = \text{snd} \) ] fastforce

  also have \( \ldots = (\sum_{i=1..\text{card } A. (\sum I|I \subseteq A \land \text{card } I = i \land x \in \bigcap I. (-1) \cdot (i + 1)) } \)
  using assms by (auto intro!: card mono)
  also have \( \ldots = (\sum_{i=1..\text{card } A. (-1) \cdot (i + 1) \cdot (\sum I|I \subseteq A \land \text{card } I = i \land x \in \bigcap I. 1)) ) \)
  by (auto simp add: \( \text{finite } A \) by (auto simp add: \( K \)-def intro: card mono)
next
  fix \( i \)
  assume \( i \in \{1..\text{card } A\} - \{1..\text{card } K\} \)
  then have \( \exists i: i < \text{card } A \land \text{card } K < i \)
  by auto
  have \( \{I. I \subseteq A \land \text{card } I = i \land x \in \bigcap I\} = \{I. I \subseteq K \land \text{card } I = i\} \)
  by (auto simp add: \( K \)-def)
  also have \( \ldots = \emptyset \)
  using \( \text{finite } A \) by (auto simp add: \( K \)-def dest: card mono[rotated 1])
  finally show \( (-1) \cdot (i + 1) \cdot (\sum I|I \subseteq A \land \text{card } I = i \land x \in \bigcap I. 1 = \text{int}) = 0 \)
  by (simp only:) simp
next
  fix \( i \)
  have \( (\sum I|I \subseteq A \land \text{card } I = i \land x \in \bigcap I. 1) = (\sum I|I \subseteq K \land \text{card } I = \)
THEORY "Binomial" 1678

i. 1 :: int

(is ?lhs = ?rhs)
  by (rule sum.cong) (auto simp add: K-def)
then show \((-1)^m \cdot (i + 1) * ?lhs = (-1)^m \cdot (i + 1) * ?rhs
by simp
qed
also have \(\{I. I \subseteq K \land \text{card } I = 0\} = \{\}\)
using assms by (auto simp add: card-eq-0-iff K-def dest: finite-subset)
then have ?rhs = \((\sum i = 0.. \text{card } K. (-1)^m \cdot (i + 1) * (\sum I | I \subseteq K \land \text{card } I = i. I :: int)) + 1
by (subth (2) sum.atLeast-Suc-atMost) simp-all
also have \(\cdots = (\sum i = 0.. \text{card } K. (-1)^m \cdot (i + 1) * \text{int } (\text{card } K \text{ choose } i)) + 1
using K by (subth n-subsets[symmetric]) simp-all
also have \(\cdots = - (\sum i = 0.. \text{card } K. (-1)^m \cdot i * \text{int } (\text{card } K \text{ choose } i)) + 1
by (subth sum-distrib-left[symmetric]) simp
also have \(\cdots = - ((-1 + 1)^m \cdot \text{card } K) + 1
by (subth binomial-ring) (simp add: ac-simps atMost-atLeast0)
also have \(\cdots = 1
using x K by (auto simp add: K-def card-gt-0-iff)
finally show ?lhs x = 1.
qed
also have nat \(\cdots = \text{card } (\bigcup A)
by simp
finally show ?thesis ..
qed

The number of nat lists of length \(m\) summing to \(N\) is \(N + m - 1\) choose \(N\):

lemma card-length-sum-list-rec:
assumes \(m \geq 1\)
shows \(\text{card } \{l :: \text{nat list. length } l = m \land \text{sum-list } l = N\} = \)
  \(\text{card } \{l. \text{length } l = (m - 1) \land \text{sum-list } l = N\} + \)
  \(\text{card } \{l. \text{length } l = m \land \text{sum-list } l + 1 = N\}
(is \text{card } ?C = \text{card } ?A + \text{card } ?B)

proof -
let \(?A' = \{l. \text{length } l = m \land \text{sum-list } l = N \land \text{hd } l = 0\}
let \(?B' = \{l. \text{length } l = m \land \text{sum-list } l = N \land \text{hd } l \neq 0\}
let \(\text{if } = \lambda l. 0 \# l
let \(\text{if } = \lambda l. \text{hd } l + 1 \# \text{tl } l
have 1: \text{x} \neq [] \Longrightarrow x = \text{hd } xs \Longrightarrow x \# \text{tl } xs = xs \text{ for } x :: \text{nat and } xs
by simp
have 2: \text{x} \neq [] \Longrightarrow \text{sum-list}(\text{tl } xs) = \text{sum-list } xs - \text{hd } xs \text{ for } xs :: \text{nat list}
by (auto simp add: neq-Nil-conv)
have \(\text{f} :: \text{bij-betw } ?f \ ?A \ ?A'
apply (rule bij-betw-byWitness[where \(f' = \text{tl}\])
using assms
apply (auto simp: 2 length-0-conv[symmetric] 1 simp del: length-0-conv)
done
have 3: xs ≠ [] ⇒ hd xs + (sum-list xs − hd xs) = sum-list xs for xs :: nat list
   by (metis 1 sum-list-simps(2) 2)

have g: bij_betw ?g ?B ?B'
   apply (rule bij_betw_byWitness[where f' = λl. (hd l - 1) # tl l])
   using assms
   by (auto simp: 2 length-0-cone[symmetric] intro: 3
       simp del: length-greater-0-cone length-0-cone)

have fin: finite {xs. size xs = M ∧ set xs ⊆ {0..<N}} for M N :: nat
   using finite-lists-length-eq[OF finite-atLeastLessThan]
   by (auto)

have fin-A: finite ?A
   using fin[of - N + 1]
   by (intro finite-subset[where ?A = ?B and ?B = {xs. size xs = m - 1 ∧ set xs ⊆ {0..<N+1}}])

have fin-B: finite ?B
   by (intro finite-subset[where ?A = ?A and ?B = {xs. size xs = m ∧ set xs ⊆ {0..<N}}])

have uni: ?C = ?A' ∪ ?B'
   by auto

have disj: ?A' ∩ ?B' = {} by blast

have card ?C = card(?A' ∪ ?B')
   using uni by simp

also have ... = card ?A + card ?B
   using card-Un-disjoint[OF - - disj] bij_betw-finite[OF f] bij_betw-finite[OF g]
   bij_betw-same-card[OF f] bij_betw-same-card[OF g] fin-A fin-B
   by presburger

finally show ?thesis.

qed

lemma card-length-sum-list: card {l::nat list. size l = m ∧ sum-list l = N} = (N + m - 1) choose N
   — by Holden Lee, tidied by Tobias Nipkow

proof (cases m)
  case 0
  then show ?thesis
    by (cases N) (auto cong: conj-cong)

next
  case (Suc m)
  have m: m ≥ 1
    by (simp add: Suc)
  then show ?thesis
    proof (induct N + m - 1 arbitrary: N m)
      case 0 — In the base case, the only solution is [0].
      then show ?thesis
        proof (simpl)
          have [simp]: {l::nat list. length l = Suc 0 ∧ (∀n∈set l. n = 0)} = {[0]}
            by (auto simp: length-Suc-cone)
          have m = 1 ∧ N = 0
            using 0 by linarith
          then show ?case
            by simp
        qed
    qed
next
  case (Suc k)
  have c1: card \{l::nat list. size l = (m - 1) \land \text{sum-list } l = N\} = (N + (m - 1) - 1) choose N
  proof (cases m = 1)
    case True
    with Suc.hyps have N \geq 1
    by auto
    with True show ?thesis
    by (simp add: binomial-eq-0)
  next
    case False
    then show ?thesis
    using Suc by fastforce
  qed
from Suc have c2: card \{l::nat list. size l = m \land \text{sum-list } l + 1 = N\} =
(if N > 0 then ((N - 1) + m - 1) choose (N - 1) else 0)
proof
  have \*: n > 0 \implies Suc m = n \iff m = n - 1 for m n
  by arith
  from Suc have N > 0 \implies
card \{l::nat list. size l = m \land \text{sum-list } l + 1 = N\} =
((N - 1) + m - 1) choose (N - 1)
by (simp add: *)
then show ?thesis
by auto
qed
from Suc.prems have (card \{l::nat list. size l = (m - 1) \land \text{sum-list } l = N\} +
card \{l::nat list. size l = m \land \text{sum-list } l + 1 = N\}) = (N + m - 1)
choose N
by (auto simp: c1 c2 choose-reduce-nat[of N + m - 1 N] simp del: One-nat-def)
then show ?thesis
using card-length-sum-list-rec[OF Suc.prems] by auto
qed

lemma card-disjoint-shuffles:
assumes set xs \cap set ys = {}
shows card (shuffles xs ys) = (length xs + length ys) choose length xs
using assms
proof (induction xs ys rule: shuffles.induct)
  case (3 x xs y ys)
  have shuffles (x \# xs) (y \# ys) = (#) x \cdot shuffles xs (y \# ys) \cup (#) y \cdot shuffles
(x \# xs) ys
  by (rule shuffles.simps)
  also have card … = card ((#) x \cdot shuffles xs (y \# ys)) + card ((#) y \cdot shuffles
(x \# xs) ys)
  by (rule card-Un-disjoint) (insert 3.prems, auto)
also have card ((#) x \cdot shuffles xs (y \# ys)) = card (shuffles xs (y \# ys))
by (rule card-image) auto
also have \( \ldots = (\text{length } xs + \text{length } (y \# ys)) \) choose length xs
using 3.prems by (intro 3.IH) auto
also have \( \text{card } ((\#) y \# \text{shuffles } (x \# xs) \# ys) = \text{card } (\text{shuffles } (x \# xs) \# ys) \)
by (rule card-image) auto
also have \( \ldots = (\text{length } (x \# xs) + \text{length } ys) \) choose length \( (x \# xs) \)
using 3.prems by (intro 3.IH) auto
also have \( \text{length } xs + \text{length } (y \# ys) \) choose length \( xs + \ldots = \)
\( (\text{length } (x \# xs) + \text{length } (y \# ys)) \) choose length \( (x \# xs) \) by simp
finally show \( \text{?case} \).
qed auto

lemma Suc-times-binomial-add
Suc a \(*\) (Suc (Suc \((a + b)\) choose Suc a)) = Suc b \(*\) (Suc (Suc \((a + b)\) choose a))
— by Lukas Bulwahn
proof –
have dvd: Suc a \(*\) (Suc a \(*\) fact b) dvd fact (Suc (Suc \((a + b)\) choose Suc a)) for a b
using fact-fact-dvd-exists[of Suc a b, where \( \text{\textquoteleft} a = \text{nat} \text{\textquoteleft} \)]
by (simp only: fact-Suc add-Suc[of-nat-id mult.assoc])
have Suc a \(*\) (Suc a \(*\) fact b) \text{div} (Suc a \(*\) fact a \(*\) fact b) =
Suc a \(*\) fact (Suc \((a + b)\) choose Suc a \(*\) fact a \(*\) fact b))
by (subst div-mul-swap[symmetric]; simp only: mul.assoc dvd)
also have \( \ldots = \text{Suc b \(*\) fact (Suc (a + b) choose Suc b)} \text{div} (\text{Suc b \(*\) fact a \(*\) fact b}) \)
by (simp only: div-mul1)
also have \( \ldots = \text{Suc b \(*\) fact (Suc (a + b) choose Suc b)} \text{div} (\text{Suc b \(*\) fact a \(*\) fact b}) \)
using dvd(a \(*\) b \(*\) a) by (subst div-mul-swap[symmetric]; simp only: ac-simps dvd)
finally show \( \text{?thesis} \)
by (subst \((1 \ 2) \text{ binomial-altdef-nat}\)
(simp-all only: ac-simps diff-Suc-Suc Suc-diff-le diff-add-inverse fact-Suc of-nat-id))
qed

91.5 Executable code

lemma gbinomial-code [code]:
a gchoose k =
(if k = 0 then 1
else fold-atLeastAtMost-nat (\( \lambda \text{acc. } \text{(a - of-nat k) \* acc} \) 0 (k - 1) 1 / fact k)
by (cases k)
(by simp-all add: gbinomial-prod-rev prod-atLeastAtMost-code [symmetric]
atLeastLessThanSuc-atLeastAtMost)

lemma binomial-code [code]:
n choose k =
(if k > n then 0
else if 2 \(*\) k > n then n choose (n - k)
else (fold-atLeastAtMost-nat (*\) (n - k) 1 n 1 div fact k))
proof –
\{ 
  \text{assume } k \leq n \\
  \text{then have } \{1..n\} = \{1..n-k\} \cup \{n-k+1..n\} \text{ by auto} \\
  \text{then have } (\text{fact } n :: \text{nat}) = \text{fact } (n-k) \ast \prod \{n-k+1..n\} \\
  \text{by } (\text{simp add: prod.union-disjoint fact-prod}) \\
\}

\text{then show } ?\text{thesis by } (\text{auto simp: binomial-altdef-nat mult-ac prod-atLeastAtMost-code})

\text{qed}

end

92 \text{Main HOL}

Classical Higher-order Logic – only “Main”, excluding real and complex numbers etc.

theory \text{Main}
  imports
    Predicate-Compile
    Quickcheck-Narrowing
    Extraction
    Nunchaku
    BNF-Greatest-Fixpoint
    Filter
    Conditionally-Complete-Lattices
    Binomial
    GCD
begin

Namespace cleanup

hide-const (open)
  czero cinfinite cfinite csun cone ctwo Csum cprod cexp image2 image2p vimage2p
Gr Grp collect
  fsts snds setl setr convol pick-middlep fstOp sndOp csquare relImage relInvImage
  Succ Shift
  shift proj id-bnf

hide-fact (open) id-bnf-def type-definition-id-bnf-UNIV

Syntax cleanup

no-notation
  bot (⊥) and
  top (⊤) and
  inf (infixl \& 70) and
  sup (infixl ⊔ 65) and
  Inf (infixl \bigsqcup) and
  Sup (infixl \bigsqcap) and
  ordLeq2 (infix \leq o 50) and
ordLeq3 (infix ≤ o 50) and
ordLess2 (infix < o 50) and
ordIso2 (infix = o 50) and
card-of ([⊥]) and
BNF-Cardinal-Arithmetic.csum (infixr + e 65) and
BNF-Cardinal-Arithmetic.cprod (infixr * e 80) and
BNF-Cardinal-Arithmetic.cexp (infixr ^ c 90) and
BNF-Def.convol (⟨⟨-,-⟩⟩)

bundle cardinal-syntax begin
notation
ordLeq2 (infix <= o 50) and
ordLeq3 (infix <= o 50) and
ordLess2 (infix < o 50) and
ordIso2 (infix = o 50) and
card-of ([⊥]) and
BNF-Cardinal-Arithmetic.csum (infixr + e 65) and
BNF-Cardinal-Arithmetic.cprod (infixr * e 80) and
BNF-Cardinal-Arithmetic.cexp (infixr ^ c 90)

alias cinfinite = BNF-Cardinal-Arithmetic.cinfinite
alias czero = BNF-Cardinal-Arithmetic.czero
alias cone = BNF-Cardinal-Arithmetic.cone
alias ctwo = BNF-Cardinal-Arithmetic.ctwo
end

no-syntax
-INFI :: pttrns ⇒ 'b ⇒ 'b (|×|-/: -) [0, 10] 10)
-INF :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b (|×|-/: -) [0, 0, 10] 10)
-SUPI :: pttrns ⇒ 'b ⇒ 'b (|×|-/: -) [0, 10] 10)
-SUP :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b (|×|-/: -) [0, 0, 10] 10)

end

93 Archimedean Fields, Floor and Ceiling Functions

theory Archimedean-Field
imports Main
begin

lemma cInf-abs-ge:
  fixes S :: 'a::{linordered-idom,conditionally-complete-linorder} set
  assumes S ≠ {} and bdd: ∀x. x∈S ⇒ |x| ≤ a
  shows |Inf S| ≤ a
proof –
  have Sup (uminus ' S) = − (Inf S)
proof (rule antisym)
  show − (Inf S) ≤ Sup (uminus ' S)
    apply (subst minus-le-iff)
    apply (rule cInf-greatest [OF ⟨S ≠ {⟩])
    apply (subst minus-le-iff)
    apply (rule cSup-upper)
    apply force
    using bdd
    apply (force simp: abs-le-iff bdd-above-def)
  done
next
  have ∗: ∀x. x ∈ S =⇒ Inf S ≤ x
    by (meson abs-le-iff bdd bdd-below-def cInf-lower minus-le-iff)
  show Sup (uminus ' S) ≤ − Inf S
    using ⟨S ≠ {⟩ by (force intro: ∗ cSup-least)
  qed
with cSup-abs-le [of uminus ' S] assms show ?thesis
  by fastforce
qed

lemma cSup-asclose:
  fixes S :: 'a::{linordered-idom,conditionally-complete-linorder} set
  assumes S: S ≠ {} and b: ∀x∈S. |x − l| ≤ e
  shows |Sup S − l| ≤ e
proof −
  have ∗: |x − l| ≤ e ↔ l − e ≤ x ∧ x ≤ l + e for x l e :: 'a
    by arith
  have bdd-above S
    using b by (auto intro: bdd-aboveI [of - l + e])
  with S b show ?thesis
    unfolding ∗ by (auto intro!: cSup-upper2 cSup-least)
  qed

lemma cInf-asclose:
  fixes S :: 'a::{linordered-idom,conditionally-complete-linorder} set
  assumes S: S ≠ {} and b: ∀x∈S. |x − l| ≤ e
  shows |Inf S − l| ≤ e
proof −
  have ∗: |x − l| ≤ e ↔ l − e ≤ x ∧ x ≤ l + e for x l e :: 'a
    by arith
  have bdd-below S
    using b by (auto intro!: bdd-belowI [of - l − e])
  with S b show ?thesis
    unfolding ∗ by (auto intro!: cInf-lower2 cInf-greatest)
  qed
93.1 Class of Archimedean fields

Archimedean fields have no infinite elements.

```plaintext
class archimedean-field = linordered-field +
  assumes ex-le-of-int: \exists z. x \leq of-int z

lemma ex-less-of-int: \exists z. x < of-int z
  for x :: 'a::archimedean-field
proof -
  from ex-le-of-int obtain z where x \leq of-int z ..
  then have x < of-int (z + 1) by simp
  then show ?thesis ..
qed

lemma ex-of-int-less: \exists z. of-int z < x
  for x :: 'a::archimedean-field
proof -
  from ex-less-of-int obtain z where -x < of-int z ..
  then have of-int (-z) < x by simp
  then show ?thesis ..
qed

lemma reals-Archimedean2: \exists n. x < of-nat n
  for x :: 'a::archimedean-field
proof -
  obtain z where x < of-int z
    using ex-less-of-int ..
  also have \ldots \leq of-int (int (nat z))
    by simp
  also have \ldots = of-nat (nat z)
    by (simp only: of-int-of-nat-eq)
  finally show ?thesis ..
qed

lemma real-arch-simple: \exists n. x \leq of-nat n
  for x :: 'a::archimedean-field
proof -
  obtain n where x < of-nat n
    using reals-Archimedean2 ..
  then have x \leq of-nat n
    by simp
  then show ?thesis ..
qed
```

Archimedean fields have no infinitesimal elements.

```plaintext
lemma reals-Archimedean:
  fixes x :: 'a::archimedean-field
  assumes \theta < x
  shows \exists n. inverse (of-nat (Suc n)) < x
```
proof –
from \( 0 < x \) have \( 0 < \text{inverse } x \)
  by (rule positive-imp-inverse-positive)
obtain \( n \) where \( \text{inverse } x < \text{of-nat } n \)
  using reals-Archimedean2 ..
then obtain \( m \) where \( \text{inverse } x < \text{of-nat } (\text{Suc } m) \)
  using \( (0 < \text{inverse } x) \) by (cases \( n \)) (simp-all del: of-nat-Suc)
then have \( \text{inverse } (\text{of-nat } (\text{Suc } m)) < \text{inverse } x \)
  using \( (0 < \text{inverse } x) \) by (rule less-imp-inverse-less)
then have \( \text{inverse } (\text{of-nat } (\text{Suc } m)) < x \)
  using \( (0 < x) \) by (simp add: nonzero-inverse-inverse-eq)
then show \( ?\text{thesis} \).
qed

lemma ex-inverse-of-nat-less:
fixes \( x \) :: 'a::archimedean-field
assumes \( 0 < x \)
shows \( \exists n > 0. \text{inverse } (\text{of-nat } n) < x \)
using reals-Archimedean [OF \( (0 < x) \)] by auto

lemma ex-less-of-nat-mult:
fixes \( x \) :: 'a::archimedean-field
assumes \( 0 < x \)
shows \( \exists n. y < \text{of-nat } n * x \)
proof –
obtain \( n \) where \( y / x < \text{of-nat } n \)
  using reals-Archimedean2 ..
with \( (0 < x) \) have \( y < \text{of-nat } n * x \)
  by (simp add: pos-divide-less-eq)
then show \( ?\text{thesis} \).
qed

93.2 Existence and uniqueness of floor function

lemma exists-least-lemma:
assumes \( \neg P \ 0 \) and \( \exists n. \ P \ n \)
shows \( \exists n. \neg P \ n \land P (\text{Suc } n) \)
proof –
from \( \exists n. \ P \ n \) have \( P (\text{Least } P) \)
  by (rule LeastI-ex)
with \( (\neg P \ 0) \) obtain \( n \) where \( \text{Least } P = \text{Suc } n \)
  by (cases Least \( P \)) auto
then have \( n < \text{Least } P \)
  by simp
then have \( \neg P \ n \)
  by (rule not-less-Least)
then have \( \neg P \ n \land P (\text{Suc } n) \)
  using \( (P (\text{Least } P)) \ (\text{Least } P = \text{Suc } n) \) by simp
then show \( ?\text{thesis} \).
lemma floor-exists:
fixes \( x :: 'a::archimedean-field \)
shows \( \exists z. \text{of-int } z \leq x \land x < \text{of-int} \ (z + 1) \)
proof (cases \( 0 \leq x \))
case True
then have \( \neg x < \text{of-nat} \ 0 \)
by simp
then have \( \exists n. \neg x < \text{of-nat} \ n \land x < \text{of-nat} \ (Suc \ n) \)
using real-Archimedean2 by (rule exists-least-lemma)
then obtain \( n \) where \( \neg x < \text{of-nat} \ n \land x < \text{of-nat} \ (Suc \ n) \)
then have \( \text{of-int} \ (\text{int} \ n) \leq x \land x < \text{of-int} \ (\text{int} \ n + 1) \)
by simp
then show \( \text{thesis} \).
next
case False
then have \( \neg x \leq \text{of-nat} \ 0 \)
by simp
then have \( \exists n. \neg x \leq \text{of-nat} \ n \land x \leq \text{of-nat} \ (Suc \ n) \)
using real-arch-simple by (rule exists-least-lemma)
then obtain \( n \) where \( \neg x \leq \text{of-nat} \ n \land x \leq \text{of-nat} \ (Suc \ n) \)
then have \( \text{of-int} \ (\text{int} \ n - 1) \leq x \land x < \text{of-int} \ (\text{int} \ n - 1 + 1) \)
by simp
then show \( \text{thesis} \).
qed

93.3 Floor function

class floor-ceiling = archimedean-field +
fixes floor :: 'a => int \([\cdot]\)
assumes floor-correct: of-int \( \lfloor x \rfloor \leq x \land x < \text{of-int} \ (\lfloor x \rfloor + 1) \)

lemma floor-unique: of-int \( z \leq x \implies x < \text{of-int} \ z + 1 \implies \lfloor x \rfloor = z \)
using floor-correct \( \lfloor x \rfloor \) floor-exists1 \( \lfloor x \rfloor \) by auto
lemma floor-eq-iff: \(|x| = a \iff \text{of-int } a \leq x \wedge x < \text{of-int } a + 1\)
using floor-correct floor-unique by auto

lemma of-int-floor-le [simp]: \(\text{of-int } |x| \leq x\)
using floor-correct ..

lemma le-floor-iff: \(z \leq |x| \iff \text{of-int } z \leq x\)
proof
  assume \(z \leq |x|\)
  then have \((\text{of-int } z :\text{ of-int } |x|) \leq \text{of-int } |x|\) by simp
  also have \(\text{of-int } |x| \leq x\) by (rule of-int-floor-le)
  finally show \(\text{of-int } z \leq x\).
next
  assume \(\text{of-int } z \leq x\)
  also have \(x < \text{of-int } (|x| + 1)\) using floor-correct ..
  finally show \(z \leq |x|\) by (simp del: of-int-add)
qed

lemma floor-less-iff: \(|x| < z \iff x < \text{of-int } z\)
by (simp add: not-le [symmetric] le-floor-iff)

lemma less-floor-iff: \(z < |x| \iff \text{of-int } z + 1 \leq x\)
using le-floor-iff [of z + 1 x] by auto

lemma floor-le-iff: \(|x| \leq z \iff x < \text{of-int } z + 1\)
by (simp add: not-less [symmetric] less-floor-iff)

lemma floor-split[arith-split]: \(P |t| \iff (\forall i. \text{of-int } i \leq t \wedge t < \text{of-int } i + 1 \implies P i)\)
by (metis floor-correct floor-unique less-floor-iff not-le order_refl)

lemma floor-mono:
  assumes \(x \leq y\)
  shows \(|x| \leq |y|\)
proof –
  have \(\text{of-int } |x| \leq x\) by (rule of-int-floor-le)
  also note \(x \leq y\)
  finally show \(?thesis\) by (simp add: le-floor-iff)
qed

lemma floor-less-cancel: \(|x| < |y| \implies x < y\)
by (auto simp add: not-le [symmetric] floor-mono)

lemma floor-of-int [simp]: \(|\text{of-int } z| = z\)
by (rule floor-unique) simp-all

lemma floor-of-nat [simp]: \(|\text{of-nat } n| = \text{int } n\)
using floor-of-int [of \text{of-nat } n] by simp
THEORY "Archimedean-Field"

lemma le-floor-add: \[|x| + |y| \leq |x + y|\]
  by (simp only: le-floor-iff of-int-add add-mono of-int-floor-le)

Floor with numerals.

lemma floor-zero [simp]: \(|0| = 0\)
  using floor-of-int [of 0] by simp

lemma floor-one [simp]: \(|1| = 1\)
  using floor-of-int [of 1] by simp

lemma floor-numeral [simp]: \([\text{numeral } v] = \text{numeral } v\)
  using floor-of-int [of numeral v] by simp

lemma floor-neg-numeral [simp]: \([-\text{numeral } v] = -\text{numeral } v\)
  using floor-of-int [of \(-\text{numeral } v\)] by simp

lemma zero-le-floor [simp]: \(0 \leq |x| \iff 0 \leq x\)
  by (simp add: le-floor-iff)

lemma one-le-floor [simp]: \(1 \leq |x| \iff 1 \leq x\)
  by (simp add: le-floor-iff)

lemma numeral-le-floor [simp]: \(\text{numeral } v \leq |x| \iff \text{numeral } v \leq x\)
  by (simp add: le-floor-iff)

lemma zero-less-floor [simp]: \(0 < |x| \iff 1 \leq x\)
  by (simp add: less-floor-iff)

lemma one-less-floor [simp]: \(1 < |x| \iff 2 \leq x\)
  by (simp add: less-floor-iff)

lemma numeral-less-floor [simp]: \(\text{numeral } v < |x| \iff \text{numeral } v + 1 \leq x\)
  by (simp add: less-floor-iff)

lemma zero-less-numeral-floor [simp]: \(-\text{numeral } v < |x| \iff -\text{numeral } v \leq x\)
  by (simp add: less-floor-iff)

lemma one-less-numeral-floor [simp]: \(1 < |x| \iff 2 \leq x\)
  by (simp add: less-floor-iff)

lemma le-floor-zero [simp]: \(|x| \leq 0 \iff x < 1\)
  by (simp add: floor-le-iff)

lemma le-floor-one [simp]: \(|x| \leq 1 \iff x < 2\)
  by (simp add: floor-le-iff)

lemma le-floor-numeral [simp]: \(|x| \leq \text{numeral } v \iff x < \text{numeral } v + 1\)
THEORY "Archimedean-Field"

by (simp add: floor-le-iff)

lemma floor-le-neg-numeral [simp]: \([x] \leq - \text{numeral } v \leftrightarrow x < - \text{numeral } v + 1\)
  by (simp add: floor-le-iff)

lemma floor-less-zero [simp]: \([x] < 0 \leftrightarrow x < 0\)
  by (simp add: floor-less-iff)

lemma floor-less-one [simp]: \([x] < 1 \leftrightarrow x < 1\)
  by (simp add: floor-less-iff)

lemma floor-less-numeral [simp]: \([x] < \text{numeral } v \leftrightarrow x < \text{numeral } v\)
  by (simp add: floor-less-iff)

lemma floor-less-neg-numeral [simp]: \([x] < - \text{numeral } v \leftrightarrow x < - \text{numeral } v\)
  by (simp add: floor-less-iff)

lemma le-mult-floor-Ints:
  assumes \(0 \leq a \in \text{Ints}\)
  shows \((\text{of-int } \lfloor a \rfloor \ast \lfloor b \rfloor) \leq (\text{of-int } \lfloor a \ast b \rfloor) : 'a :: linorder-idom\)
  by (metis Ints-cases assms floor-less-iff floor-of-int linorder-not-less mult-left-mono
    of-int-floor-le of-int-less-iff of-int-mult)

Addition and subtraction of integers.

lemma floor-add-int: \([x] + z = \lfloor x + \text{of-int } z \rfloor\)
  using floor-correct \([\text{of } x]\) by (simp add: floor-unique[symmetric])

lemma int-add-floor: \(z + [x] = \lfloor \text{of-int } z + x \rfloor\)
  using floor-correct \([\text{of } x]\) by (simp add: floor-unique[symmetric])

lemma one-add-floor: \([x] + 1 = \lfloor x + 1 \rfloor\)
  using floor-add-int \([\text{of } x 1]\) by simp

lemma floor-diff-of-int [simp]: \([x] - \text{of-int } z = \lfloor x \rfloor - z\)
  using floor-add-int \([\text{of } x - z]\) by (simp add: algebra-simps)

lemma floor-uminus-of-int [simp]: \([- (\text{of-int } z)] = - z\)
  by (metis floor-diff-of-int \([\text{of } 0]\) diff-0 floor-zero)

lemma floor-diff-numeral [simp]: \([x] - \text{numeral } v = \lfloor x \rfloor - \text{numeral } v\)
  using floor-diff-of-int \([\text{of } x \text{ numeral } v]\) by simp

lemma floor-diff-one [simp]: \([x] - 1 = \lfloor x \rfloor - 1\)
  using floor-diff-of-int \([\text{of } x 1]\) by simp

lemma le-mult-floor:
  assumes \(0 \leq a \text{ and } 0 \leq b\)
  shows \([a] \ast [b] \leq [a \ast b]\)
proof

have of-int ⌊a⌋ ≤ a and of-int ⌈b⌉ ≤ b
  by (auto intro: of-int-floor-le)
then have of-int ⌊(a + b)⌋ ≤ a * b
  using assms by (auto intro!: mult-mono)
also have a * b < of-int ⌈(a * b) + 1⌉
  using floor-correct[of a * b] by auto
finally show ?thesis
  unfolding of-int-less_iff by simp
qed

lemma floor-divide-of-int-eq: ⌊of-int k / of-int l⌋ = k div l
for k l :: int
proof (cases l = 0)
  case True
  then show ?thesis by simp
next
  case False
  have *: ⌊of-int (k mod l) / of-int l :: 'a⌋ = 0
    proof (cases l > 0)
      case True
      then show ?thesis
        by (auto intro: floor-unique)
    next
      case False
      obtain r where r = - l
        by blast
      then have l: l = - r
        by simp
      with ⟨l ≠ 0⟩ False have r > 0
        by simp
      with l show ?thesis
        using pos-mod-bound[of r]
        by (auto simp add: zmod-zminus2-eq-if less-le field-simps intro: floor-unique)
    qed
    have (of-int k :: 'a) = of-int (k div l * l + k mod l)
      by simp
    also have ... = (of-int (k div l) + of-int (k mod l) / of-int l) * of-int l
      using False by (simp only: of-int-add) (simp add: field-simps)
    finally have (of-int k / of-int l :: 'a) = ... / of-int l
      by simp
    then have (of-int k / of-int l :: 'a) = of-int (k div l) + of-int (k mod l) / of-int l
      using False by (simp only:) (simp add: field-simps)
    then have ⌊of-int k / of-int l :: 'a⌋ = ⌊of-int (k div l) + of-int (k mod l) / of-int l :: 'a⌋
      by simp
    then have ⌊of-int k / of-int l :: 'a⌋ = ⌊of-int (k mod l) / of-int l + of-int (k div l) :: 'a⌋
THEORY "Archimedean-Field"

by (simp add: ac-simps)
then have \([\text{of-int } k / \text{of-int } l :: 'a] = [\text{of-int } (k \mod l) / \text{of-int } l :: 'a] + k \div l\)
by (simp add: floor-add-int)
with \(*\) show \(?\text{thesis}\)
by simp

qed

lemma \(\text{floor-divide-of-nat-eq: } [\text{of-nat } m / \text{of-nat } n] = \text{of-nat } (m \div n)\)
for \(m n :: \text{nat}\)
proof (cases \(n = 0\))
case True
then show \(?\text{thesis}\) by simp
next
case False
then have \(*::[\text{of-nat } (m \mod n) / \text{of-nat } n :: 'a] = 0\)
by (auto intro: floor-unique)
have \(\text{of-nat } m :: 'a] = \text{of-nat } (m \div n \ast n + m \mod n)\)
by simp
also have \(\ldots = (\text{of-nat } (m \div n) + \text{of-nat } (m \mod n) / \text{of-nat } n) \ast \text{of-nat } n\)
using False by (simp only: of-nat-add) (simp add: field-simps)
finally have \(\text{of-nat } m / \text{of-nat } n :: 'a] = \ldots / \text{of-nat } n\)
by simp
then have \((\text{of-nat } m / \text{of-nat } n :: 'a) = \text{of-nat } (m \div n) + \text{of-nat } (m \mod n) / \text{of-nat } n\)
using False by (simp only:) simp
then have \([\text{of-nat } m / \text{of-nat } n :: 'a] = [\text{of-nat } (m \div n) + \text{of-nat } (m \mod n) / \text{of-nat } n :: 'a]\)
by simp
then have \([\text{of-nat } m / \text{of-nat } n :: 'a] = [\text{of-nat } (m \mod n) / \text{of-nat } n :: 'a] + \text{of-nat } (m \div n)\)
by (simp add: ac-simps)
moreover have \(\text{of-nat } (m \div n) :: 'a] = \text{of-int } (\text{of-nat } (m \div n))\)
by simp
ultimately have \([\text{of-nat } m / \text{of-nat } n :: 'a] = \text{of-int } (\text{of-nat } (m \div n)) + \text{of-nat } (m \div n)\)
by (simp only: floor-add-int)
with \(*\) show \(?\text{thesis}\)
by simp

qed

lemma \(\text{floor-divide-lower}:\)
fixes \(q :: 'a::floor-ceiling\)
says \(q > 0 \implies \text{of-int } [p / q] \ast q \leq p\)
using of-int-floor-le pos-le-divide-eq by blast

lemma \(\text{floor-divide-upper}:\)
fixes \(q :: 'a::floor-ceiling\)
says \(q > 0 \implies p < (\text{of-int } [p / q] + 1) \ast q\)
by (meson floor-eq-iff pos-divide-less-eq)
93.4 Ceiling function

definition ceiling :: 'a::floor-ceiling ⇒ int ([\cdot])
  where \([x] = - \lfloor -x \rfloor\)

lemma ceiling-correct: of-int \([x] - 1 < x \land x \leq \text{of-int} \ [x]\)
  unfolding ceiling-def using floor-correct [of - x]
  by (simp add: le-minus-iff)

lemma ceiling-unique: of-int \(z - 1 < x \implies z \leq \text{of-int} \ z \implies \ [x] = z\)
  unfolding ceiling-def using floor-unique [of - z - x] by simp

lemma ceiling-eq-iff: \([x] = a \iff \text{of-int} \ a - 1 < x \land x \leq \text{of-int} \ a\)
  using ceiling-correct ceiling-unique by auto

lemma ceiling-add-le: \(\lceil x + y \rceil \leq \lceil x \rceil + \lceil y \rceil\)
  by (simp only: ceiling-le-iff of-int-add add-mono le-of-int-ceiling)

lemma mult-ceiling-le:
  assumes \(0 \leq a \land 0 \leq b\)
  shows \([a \cdot b] \leq [a] \cdot [b]\)
  by (metis assms ceiling-le-iff ceiling-mono le-of-int-ceiling mult-mono of-int-mult)
lemma mult-ceiling-le-Ints:
  assumes \(0 \leq a\ a \in \text{Ints}\)
  shows \((\text{of-int} \lceil a \times b \rceil) :: a :: \text{linordered-idom} \leq \text{of-int}(\lfloor a \rfloor \times \lfloor b \rfloor)\)
  by (metis \text{Ints-cases} \text{assms ceiling-le-iff ceiling-of-int le-of-int-ceiling mult-left-mono of-int-le-iff of-int-mult})

lemma finite-int-segment:
  fixes \(a::\lceil a \rceil::\text{floor-ceiling}\)
  shows finite \(\{ x \in \mathbb{Z}. a \leq x \land x \leq b \}\)
proof
  have finite \(\{ \text{ceiling } a .. \text{floor } b \}\)
    by simp
  moreover have \(\{ x \in \mathbb{Z}. a \leq x \land x \leq b \} = \text{of-int} \{ \text{ceiling } a .. \text{floor } b \}\)
    by (auto simp add: \text{le-floor-iff ceiling-le-iff elim: \text{Ints-cases}})
  ultimately show \(\text{thesis}\)
    by simp
qed

corollary finite-abs-int-segment:
  fixes \(a::\lceil a \rceil::\text{floor-ceiling}\)
  shows finite \(\{ k \in \mathbb{Z}. |k| \leq a \}\)
  using finite-int-segment \(\text{of } - a a\)
    by (auto simp add: \text{abs-le-iff conj-commute minus-le-iff})

93.4.1 Ceiling with numerals.

lemma ceiling-zero [simp]: \(\lceil 0 \rceil = 0\)
  using ceiling-of-int \(\lfloor 0 \rfloor\) by simp

lemma ceiling-one [simp]: \(\lceil 1 \rceil = 1\)
  using ceiling-of-int \(\lfloor 1 \rfloor\) by simp

lemma ceiling-numeral [simp]: \(\lceil \text{numeral } v \rceil = \text{numeral } v\)
  using ceiling-of-int \(\lfloor \text{numeral } v \rfloor\) by simp

lemma ceiling-neg-numeral [simp]: \(\lceil -\text{numeral } v \rceil = -\text{numeral } v\)
  using ceiling-of-int \(\lfloor -\text{numeral } v \rfloor\) by simp

lemma ceiling-le-zero [simp]: \(\lceil x \rceil \leq 0 \iff x \leq 0\)
  by (simp add: ceiling-le-iff)

lemma ceiling-le-one [simp]: \(\lceil x \rceil \leq 1 \iff x \leq 1\)
  by (simp add: ceiling-le-iff)

lemma ceiling-le-numeral [simp]: \(\lceil x \rceil \leq \text{numeral } v \iff x \leq \text{numeral } v\)
  by (simp add: ceiling-le-iff)

lemma ceiling-le-neg-numeral [simp]: \(\lceil x \rceil \leq -\text{numeral } v \iff x \leq -\text{numeral } v\)
  by (simp add: ceiling-le-iff)
lemma ceiling-less-zero [simp]: \[ |x| < 0 \iff x \leq -1 \]
  by (simp add: ceiling-less-iff)

lemma ceiling-less-one [simp]: \[ |x| < 1 \iff x \leq 0 \]
  by (simp add: ceiling-less-iff)

lemma ceiling-less-numeral [simp]: \[ |x| < \text{numeral } v \iff x \leq \text{numeral } v - 1 \]
  by (simp add: ceiling-less-iff)

lemma ceiling-less-neg-numeral [simp]: \[ |x| < -\text{numeral } v \iff x \leq -\text{numeral } v - 1 \]
  by (simp add: ceiling-less-iff)

lemma zero-le-ceiling [simp]: \[ 0 \leq |x| \iff -1 < x \]
  by (simp add: le-ceiling-iff)

lemma one-le-ceiling [simp]: \[ 1 \leq |x| \iff 0 < x \]
  by (simp add: le-ceiling-iff)

lemma numeral-le-ceiling [simp]: \[ \text{numeral } v \leq |x| \iff \text{numeral } v - 1 < x \]
  by (simp add: le-ceiling-iff)

lemma neg-numeral-le-ceiling [simp]: \[ -\text{numeral } v \leq |x| \iff -\text{numeral } v - 1 < x \]
  by (simp add: le-ceiling-iff)

lemma ceiling-altdef: \[ |x| = (\text{if } x = \text{of-int } \lfloor x \rfloor \text{ then } \lfloor x \rfloor \text{ else } \lfloor x \rfloor + 1) \]
  by (intro ceiling-unique; simp, linarith?)

lemma floor-le-ceiling [simp]: \[ \lfloor x \rfloor \leq |x| \]
  by (simp add: ceiling-altdef)

93.4.2 Addition and subtraction of integers.

lemma ceiling-add-of-int [simp]: \[ x + \text{of-int } z = |x| + z \]
  using ceiling-correct [of x] by (simp add: ceiling-def)
lemma ceiling-add-numeral [simp]: \([x + \text{numeral } v] = [x] + \text{numeral } v\)
using ceiling-add-of-int [of x numeral v] by simp

lemma ceiling-add-one [simp]: \([x + 1] = [x] + 1\)
using ceiling-add-of-int [of x 1] by simp

lemma ceiling-diff-of-int [simp]: \([x - \text{of-int } z] = [x] - z\)
using ceiling-add-of-int [of \(x\) \(\text{numeral } v\)] by (simp add: algebra-simps)

lemma ceiling-diff-numeral [simp]: \([x - \text{numeral } v] = [x] - \text{numeral } v\)
using ceiling-diff-of-int [of \(x\) \(\text{numeral } v\)] by simp

lemma ceiling-diff-one [simp]: \([x - 1] = [x] - 1\)
using ceiling-diff-of-int [of \(x\) 1] by simp

lemma ceiling-split [arith-split]: \(P (\lceil t \rceil) \iff (\forall i. \\text{of-int } i - 1 < t \land t \leq \text{of-int } i) \longrightarrow P i)\)
by (auto simp add: ceiling-unique ceiling-correct)

lemma ceiling-diff-floor-le-1:
\([x] - \lfloor x \rfloor \leq 1\)
proof
  have \(\text{of-int } [x] - 1 < x\)
  using ceiling-correct [of \(x\)] by simp
  also have \(x < \text{of-int } [x] + 1\)
  using floor-correct [of \(x\)] by simp-all
  finally have \(\text{of-int } ([x] - [x]) < (\text{of-int } 2::'a)\)
  by simp
  then show \(?thesis\)
  unfolding of-int-less-iff by simp
qed

lemma nat-approx-posE:
fixes e :: 'a::{archimedean-field,floor-integer}
assumes 0 < e
obtains n :: nat where \(1 / \text{of-nat} (\text{Suc } n) < e\)
proof
  have \(1::'a) / \text{of-nat} (\text{Suc } \text{nat } [1/e]) < 1 / \text{of-int } ([1/e])\)
  proof (rule divide-strict-left-monotonic)
    show \((\text{of-int } [1/e])::'a\) < \(\text{of-nat} (\text{Suc } \text{nat } [1/e])\)
    using assms by (simp add: field-simps)
    show \((\theta::'a < \text{of-nat} (\text{Suc } \text{nat } [1/e]) * \text{of-int } [1/e])\)
    using assms by (auto simp: zero-less-mult-iff pos-add-strict)
  qed auto
  also have \(1 / \text{of-int } ([1/e]) \leq 1 / (1/e)\)
  by (rule divide-strict-left-monotonic) (auto simp: \(\theta < e\) ceiling-correct)
  also have \(\ldots = e \text{ by simp}\)
  finally show \(1 / \text{of-nat} (\text{Suc } \text{nat } [1/e]) < e\)
  by metis
lemma \textit{ceiling-divide-upper}:
fixes $q :: 'a::floor-ceiling$
shows $q > 0 \implies \lfloor p / q \rfloor \leq \lfloor (p / q) \rfloor * q$
by (meson divide-le-eq le-of-int-ceiling)

lemma \textit{ceiling-divide-lower}:
fixes $q :: 'a::floor-ceiling$
shows $q > 0 \implies (\lfloor p / q \rfloor - 1) * q < p$
by (meson ceiling-eq-iff pos-less-divide-eq)

93.5 Negation

lemma \textit{floor-minus} $\lfloor -x \rfloor = -\lceil x \rceil$
\textbf{unfolding ceiling-def} \textbf{by simp}

lemma \textit{ceiling-minus} $\lceil -x \rceil = -\lfloor x \rfloor$
\textbf{unfolding ceiling-def} \textbf{by simp}

93.6 Natural numbers

lemma \textit{of-nat-floor} $r \geq 0 \implies of-nat (nat \lfloor r \rfloor) \leq r$
\textbf{by simp}

lemma \textit{of-nat-ceiling} $of-nat (nat \lceil r \rceil) \geq r$
\textbf{by (cases r\geq0) auto}

93.7 Frac Function

definition \textit{frac} :: 'a \Rightarrow 'a::floor-ceiling

where \textit{frac} $x \equiv x - \lfloor x \rfloor$

lemma \textit{frac-lt-1} $\textit{frac} x < 1$
\textbf{by (simp add: frac-def) linarith}

lemma \textit{frac-eq-0-iff} \textbf{[simp]}: $\textit{frac} x = 0 \iff x \in \mathbb{Z}$
\textbf{by (simp add: frac-def) (metis Ints-cases Ints-of-int floor-of-int )}

lemma \textit{frac-ge-0} \textbf{[simp]}: $\textit{frac} x \geq 0$
\textbf{unfolding frac-def by linarith}

lemma \textit{frac-gt-0-iff} \textbf{[simp]}: $\textit{frac} x > 0 \iff x \notin \mathbb{Z}$
\textbf{by (metis frac-eq-0-iff frac-ge-0 le-less_less_irrefl)}

lemma \textit{frac-of-int} \textbf{[simp]}: $\textit{frac} (of-int z) = 0$
\textbf{by (simp add: frac-def)}

lemma \textit{frac-frac} \textbf{[simp]}: $\textit{frac} (\textit{frac} x) = \textit{frac} x$
\textbf{by (simp add: frac-def)}
**THEORY “Archimedean-Field”**

1698

**lemma** floor-add: \([x + y] = (if \frac{x} + \frac{y} < 1 then [x] + [y] else ([x] + [y]) + 1)\)

**proof**
- **have** \(x + y < 1 \rightarrow (of\_int \ [x] + of\_int \ [y]) \Rightarrow [x + y] = [x] + [y]\)
  - **by** (metis add.commute floor-unique le-floor-add le-floor-iff of-int-add)
- **moreover**
  - **have** \(\neg x + y < 1 \rightarrow (of\_int \ [x] + of\_int \ [y]) \Rightarrow [x + y] = 1 + ([x] + [y])\)
  - **apply** (simp add: floor-eq-iff)
  - **apply** (auto simp add: algebra-simps)
  - **apply** linarith
  - **done**

**ultimately show** thesis **by** (auto simp add: frac-def algebra-simps)

**qed**

**lemma** floor-add2[simp]: \(x \in \mathbb{Z} \lor y \in \mathbb{Z} \Rightarrow [x + y] = [x] + [y]\)

**by** (metis add.commute add.left-neutral frac-lt-1 floor-add frac-eq-0-iff)

**lemma** frac-add: \(\frac{x + y} = (if \frac{x} + \frac{y} < 1 then \frac{x} + \frac{y} else (\frac{x} + \frac{y} - 1))\)

**by** (simp add: frac-def floor-add)

**lemma** frac-unique-iff: \(\frac{x} = a \longleftrightarrow x - a \in \mathbb{Z} \land 0 \leq a \land a < 1\)

**for** \(x :: 'a:\text{floor-ceiling}\)

**apply** (auto simp: Ints-def frac-def algebra-simps floor-unique)

**apply** linarith

**done**

**lemma** frac-eq: \(\frac{x} = x \longleftrightarrow 0 \leq x \land x < 1\)

**by** (simp add: frac-unique-iff)

**lemma** frac-neg: \(\frac{(- x)} = (if x \in \mathbb{Z} then 0 else 1 - \frac{x})\)

**for** \(x :: 'a:\text{floor-ceiling}\)

**apply** (auto simp add: frac-unique-iff)

**apply** (simp add: frac-def)

**apply** (meson frac-lt-1 less-iff-diff-less-0 not-le not-less_iff_gr_or_eq)

**done**

**lemma** frac-in-Ints-iff [simp]: \(\frac{x} \in \mathbb{Z} \longleftrightarrow x \in \mathbb{Z}\)

**proof** safe
- **assume** \(\frac{x} \in \mathbb{Z}\)
- **hence** \(of\_int \ [x] + \frac{x} \in \mathbb{Z}\) **by** auto
- **also have** \(of\_int \ [x] + \frac{x} = x\) **by** (simp add: frac-def)
- **finally show** \(x \in \mathbb{Z}\) .

**qed** (auto simp: frac-def)

**lemma** frac-1-eq: \(\frac{x+1} = \frac{x}\)

**by** (simp add: frac-def)
93.8 Rounding to the nearest integer

definition round :: 'a::floor-ceiling ⇒ int
  where round x = ⌊x + 1/2⌋

lemma of-int-round-gc: of-int (round x) ≥ x − 1/2
  and of-int-round-le: of-int (round x) ≤ x + 1/2
  and of-int-round-abs-le: |of-int (round x) − x| ≤ 1/2
  and of-int-round-gt: of-int (round x) > x − 1/2
proof −
  from floor-correct[of x + 1/2] have x + 1/2 < of-int (round x) + 1
  by (simp add: round-def)
  from add-strict-right-mono[OF this, of −1] show A: of-int (round x) > x − 1/2
  by simp
then show of-int (round x) ≥ x − 1/2
  by simp
from floor-correct[of x + 1/2] show of-int (round x) ≤ x + 1/2
  by (simp add: round-def)
with A show |of-int (round x) − x| ≤ 1/2
  by linarith
qed

lemma round-of-int [simp]: round (of-int n) = n
unfolding round-def by (subst floor-eq-iff) force

lemma round-0 [simp]: round 0 = 0
using round-of-int[of 0] by simp

lemma round-1 [simp]: round 1 = 1
using round-of-int[of 1] by simp

lemma round-numeral [simp]: round (numeral n) = numeral n
using round-of-int[of numeral n] by simp

lemma round-neg-numeral [simp]: round (−numeral n) = −numeral n
using round-of-int[of −numeral n] by simp

lemma round-of-nat [simp]: round (of-nat n) = of-nat n
using round-of-int[of int n] by simp

lemma round-mono: x ≤ y ⇒ round x ≤ round y
unfolding round-def by (intro floor-mono) simp

lemma round-unique: of-int y > x − 1/2 ⇒ of-int y ≤ x + 1/2 ⇒ round x = y
unfolding round-def
proof (rule floor-unique)
  assume x − 1 / 2 < of-int y
from add-strict-left-mono[OF this, of 1] show x + 1 / 2 < of-int y + 1
by simp

qed

lemma round-unique': |x - of-int n| < 1/2 \implies round x = n
by (subst (asm) abs-less_iff; rule round-unique) (simp-all add: field-simps)

lemma round-altdef: round x = (if frac x \geq 1/2 then \lceil x \rceil else \lfloor x \rfloor)
by (cases frac x \geq 1/2)
(rule round-unique, ((simp add: frac-def field-simps ceiling-altdef; linarith)+)[2])+

lemma floor-le-round: \lfloor x \rfloor \leq round x
unfolding round-def by (intro floor-mono) simp

lemma ceiling-ge-round: \lceil x \rceil \geq round x
unfolding round-altdef by simp

lemma round-diff-minimal: |z - of-int (round z)| \leq |z - of-int m|
for z :: 'a::floor-ceiling
proof (cases of-int m \geq z)
case True
then have |z - of-int (round z)| \leq |of-int \lceil z \rceil - z|
unfolding round-altdef by (simp add: field-simps ceiling-altdef frac-def) linarith
also have of-int \lceil z \rceil - z \geq 0
by linarith
with True have |of-int \lceil z \rceil - z| \leq |z - of-int m|
by (simp add: ceiling-le_iff)
finally show ?thesis .
next
case False
then have |z - of-int (round z)| \leq |of-int \lfloor z \rfloor - z|
unfolding round-altdef by (simp add: field-simps ceiling-altdef frac-def) linarith
also have z - of-int \lfloor z \rfloor \geq 0
by linarith
with False have |of-int \lfloor z \rfloor - z| \leq |z - of-int m|
by (simp add: le-floor_iff)
finally show ?thesis .
qed

end

94 Rational numbers

theory Rat
import Archimedean-Field
begin
94.1 Rational numbers as quotient

94.1.1 Construction of the type of rational numbers

definition ratrel :: (int × int) ⇒ (int × int) ⇒ bool
  where ratrel = (λx y. snd x ≠ 0 ∧ snd y ≠ 0 ∧ fst x * snd y = fst y * snd x)

lemma ratrel-iff [simp]: ratrel x y ←→ snd x ≠ 0 ∧ snd y ≠ 0 ∧ fst x * snd y = fst y * snd x
  by (simp add: ratrel-def)

lemma exists-ratrel-refl: ∃x. ratrel x x
  by (auto intro!: one-neq-zero)

lemma symp-ratrel: symp ratrel
  by (simp add: ratrel-def symp-def)

lemma transp-ratrel: transp ratrel
proof (rule transpI, unfold split-paired-all)
  fix a b a' b' a'' b'' :: int
  assume *: ratrel (a, b) (a', b')
  assume **: ratrel (a', b') (a'', b'')
  have b' * (a * b'') = b'' * (a * b') by simp
  also from * have a * b' = a' * b by auto
  also have b'' * (a' * b) = b * (a' * b'') by simp
  also from ** have a' * b'' = a'' * b' by auto
  also have b * (a'' * b') = b' * (a'' * b) by simp
  finally have b' * (a * b'') = b' * (a'' * b).
  moreover from ** have b' ≠ 0 by auto
  ultimately have a * b'' = a'' * b by simp
  with * ** show ratrel (a, b) (a'', b'') by auto
qed

lemma part-equivp-ratrel: part-equivp ratrel
  by (rule part-equivpI [OF exists-ratrel-refl symp-ratrel transp-ratrel])

quotient-type rat = int × int / partial: ratrel

morphisms Rep-Rat Abs-Rat
  by (rule part-equivp-ratrel)

lemma Domainp-cr-rat [transfer-domain-rule]: Domainp per-rat = (λx. snd x ≠ 0)
  by (simp add: rat.domain-eq)

94.1.2 Representation and basic operations

lift-definition Fract :: int ⇒ int ⇒ rat
  is λa b. if b = 0 then (0, 1) else (a, b)
  by simp
lemma eq-rat:
\( \forall a\ b\ c\ d.\ b \neq 0 \implies d \neq 0 \implies \text{Fract } a\ b = \text{Fract } c\ d \iff a * d = c * b \)
\( \forall a.\ \text{Fract } a\ 0 = \text{Fract } 0\ 1 \)
\( \forall a.\ \text{Fract } 0\ a = \text{Fract } 0\ c \)
by (transfer, simp)+

lemma Rat-cases [case-names Fract, cases type: rat]:
assumes that:
\( \forall a\ b.\ q = \text{Fract } a\ b \implies b > 0 \implies \text{coprime } a\ b \implies C \)
shows C
proof –
obtain a b :: int where q: q = Fract a b and b: b \neq 0
  by transfer simp
let ?a = a div gcd a b
let ?b = b div gcd a b
from b have ?b \neq 0
  by simp
with b have ?b \neq 0
  by fastforce
with q b have q2: q = Fract ?a ?b
  by (simp add: eq-rat dvd-div-mult mult.commute [of a])
from b have coprime: coprime ?a ?b
  by (auto intro: div-gcd-coprime)
show C
proof (cases b > 0)
case True
  then have ?b > 0
    by (simp add: nonneg1-imp-zdiv-pos-iff)
  from q2 this coprime show C by (rule that)
next
case False
  have q = Fract (- ?a) (- ?b)
    unfolding q2 by transfer simp
  moreover from False b have - ?b > 0
    by (simp add: pos-imp-zdiv-neg-iff)
  moreover from coprime have coprime (- ?a) (- ?b)
    by simp
  ultimately show C
    by (rule that)
qed

lemma Rat-induct [case-names Fract, induct type: rat]:
assumes \( \forall a\ b.\ b > 0 \implies \text{coprime } a\ b \implies P \ (\text{Fract } a\ b) \)
shows P q
using assms by (cases q) simp

instantiation rat :: field
begin
lift-definition zero-rat :: rat is (0, 1)
  by simp

lift-definition one-rat :: rat is (1, 1)
  by simp

lemma Zero-rat-def: 0 = Fract 0 1
  by transfer simp

lemma One-rat-def: 1 = Fract 1 1
  by transfer simp

lift-definition plus-rat :: rat ⇒ rat ⇒ rat
  is λ x y. (fst x * snd y + fst y * snd x, snd x * snd y)
  by (auto simp: distrib-right) (simp add: ac-simps)

lemma add-rat [simp]:
  assumes b ≠ 0 and d ≠ 0
  shows Fract a b + Fract c d = Fract (a * d + c * b) (b * d)
  using assms by transfer simp

lift-definition uminus-rat :: rat ⇒ rat is λx. (− fst x, snd x)
  by simp

lemma minus-rat [simp]: − Fract a b = Fract (− a) b
  by transfer simp

lemma minus-rat-cancel [simp]: Fract (− a) (− b) = Fract a b
  by (cases b = 0) (simp-all add: eq-rat)

definition diff-rat-def: q − r = q + − r for q r :: rat

lemma diff-rat [simp]:
  b ≠ 0 ⇒ d ≠ 0 ⇒ Fract a b − Fract c d = Fract (a * d − c * b) (b * d)
  by (simp add: diff-rat-def)

lift-definition times-rat :: rat ⇒ rat ⇒ rat
  is λx y. (fst x * fst y, snd x * snd y)
  by (simp add: ac-simps)

lemma mult-rat [simp]: Fract a b * Fract c d = Fract (a * c) (b * d)
  by transfer simp

lemma mult-rat-cancel: c ≠ 0 ⇒ Fract (c * a) (c * b) = Fract a b
  by transfer simp

lift-definition inverse-rat :: rat ⇒ rat
  is λx. if fst x = 0 then (0, 1) else (snd x, fst x)
  by (auto simp add: mult.commute)
lemma inverse-rat [simp]: \( \text{inverse} \left( \text{Fract} \ a \ b \right) = \text{Fract} \ b \ a \)
by transfer simp

definition divide-rat-def: \( \text{q div r} = \text{q * inverse} \ r \) for \( q \ r :: \text{rat} \)

lemma divide-rat [simp]: \( \text{Fract} \ a \ b \text{ div} \ \text{Fract} \ c \ d = \text{Fract} \ (a * d) \ (b * c) \)
by (simp add: divide-rat-def)

instance
proof
fix \( q \ r \ s :: \text{rat} \)
show \( (q * r) * s = q * (r * s) \)
by transfer simp
show \( q * r = r * q \)
by transfer simp
show \( 1 * q = q \)
by transfer simp
show \( (q + r) + s = q + (r + s) \)
by transfer (simp add: algebra-simps)
show \( q + r = r + q \)
by transfer simp
show \( 0 + q = q \)
by transfer simp
show \( - q + q = 0 \)
by transfer simp
show \( q - r = q + - r \)
by (fact diff-rat-def)
show \( (q + r) * s = q * s + r * s \)
by transfer (simp add: algebra-simps)
show \( 0 :: \text{rat} \) \( \neq \ 1 \)
by transfer simp
show \( \text{inverse} \ q \ * \ q = 1 \) if \( q \neq 0 \)
using that by transfer simp
show \( q \text{ div} \ r = q * \text{inverse} \ r \)
by (fact divide-rat-def)
show \( \text{inverse} \ 0 = (0 :: \text{rat}) \)
by transfer simp
qed

end

lemma div-add-self1-no-field [simp]:
assumes NO-MATCH \( (x :: 'b :: \text{field}) \ b \ (b :: 'a :: \text{euclidean-semiring-cancel}) \neq 0 \)
shows \( (b + a) \text{ div} \ b = a \text{ div} \ b + 1 \)
using assms(2) by (fact div-add-self1)
lemma div-add-self2-no-field [simp]:
  assumes NO-MATCH (x :: 'b :: field) b (b :: 'a :: euclidean-semiring-cancel) ≠ 0
  shows (a + b) div b = a div b + 1
  using assms(2) by (fact div-add-self2)

lemma of-nat-rat: of-nat k = Fract (of-nat k) 1
  by (induct k) (simp-all add: Zero-rat-def One-rat-def)

lemma of-int-rat: of-int k = Fract k 1
  by (cases k rule: int-diff-cases) (simp add: of-nat-rat)

lemma Fract-of-nat-eq: Fract (of-nat k) 1 = of-nat k
  by (rule of-nat-rat [symmetric])

lemma Fract-of-int-eq: Fract k 1 = of-int k
  by (rule of-int-rat [symmetric])

lemma rat-number-collapse:
  Fract 0 k = 0
  Fract 1 1 = 1
  Fract (numeral w) 1 = numeral w
  Fract (− numeral w) 1 = − numeral w
  Fract (− 1) 1 = − 1
  Fract k 0 = 0
  using Fract-of-int-eq [of numeral w]
  and Fract-of-int-eq [of − numeral w]
  by (simp-all add: Zero-rat-def One-rat-def eq-rat)

lemma rat-number-expand:
  0 = Fract 0 1
  1 = Fract 1 1
  numeral k = Fract (numeral k) 1
  − 1 = Fract (− 1) 1
  − numeral k = Fract (− numeral k) 1
  by (simp-all add: rat-number-collapse)

lemma Rat-cases-nonzero [case-names Fract 0]:
  assumes Fract: ∃a b. q = Fract a b ⇒ b > 0 ⇒ a ≠ 0 ⇒ coprime a b ⇒ C
  and 0: q = 0 ⇒ C
  shows C
proof (cases q = 0)
  case True
  then show C using 0 by auto
next
  case False
  then obtain a b where #: q = Fract a b > 0 coprime a b
  by (cases q) auto
with False have \(0 \neq \text{Fract} \ a \ b\)
  by simp
with \((b > 0)\) have \(a \neq 0\)
  by (simp add: Zero-rat-def eq-rat)
with \(\text{Fract} *\) show \(\text{C}\) by blast
qed

94.1.3 Function normalize

lemma Fract-coprime: \(\text{Fract} \ (a \div \gcd \ a \ b) \ (b \div \gcd \ a \ b) = \text{Fract} \ a \ b\)
proof (cases \(b = 0\))
  case True
  then show \(?\)thesis
  by (simp add: eq-rat)
next
  case False
  moreover have \(b \div \gcd \ a \ b \ast \gcd \ a \ b = b\)
  by (rule dvd-div-mult-self) simp
  ultimately have \(b \div \gcd \ a \ b \ast \gcd \ a \ b \neq 0\)
  by simp
  then have \(b \div \gcd \ a \ b \neq 0\)
  by fastforce
  with False show \(?\)thesis
  by [simp add: eq-rat dvd-div-mult mult.commute [of a]]
qed

definition normalize :: \(\text{int} \times \text{int} \Rightarrow \text{int} \times \text{int}\)
where normalize \(p\) =
  (if \(\text{snd} \ p > 0\) then (let \(a = \gcd \ (\text{fst} \ p) \ (\text{snd} \ p)\) in \((\text{fst} \ p \ \text{div} \ a, \ \text{snd} \ p \ \text{div} \ a)\))
  else if \(\text{snd} \ p = 0\) then \((0, 1)\)
  else (let \(a = -\ \gcd \ (\text{fst} \ p) \ (\text{snd} \ p)\) in \((\text{fst} \ p \ \text{div} \ a, \ \text{snd} \ p \ \text{div} \ a)\))

lemma normalize-crossproduct:
  assumes \(q \neq 0 \ \text{s} \neq 0\)
  assumes normalize \((p, \ q) = \text{normalize} \ (r, \ s)\)
  shows \(p \ast \ s = r \ast \ q\)
proof
  have \(\ast\): \(p \ast \ s = q \ast \ r\)
    if \(p \ast \gcd \ r \ s = \text{sgn} \ (q \ast \ s) \ast r \ast \gcd \ p \ q\) \and \(q \ast \gcd \ r \ s = \text{sgn} \ (q \ast \ s) \ast s \ast \gcd \ p \ q\)
  proof
    from that have \((p \ast \gcd \ r \ s) \ast (\text{sgn} \ (q \ast \ s) \ast s \ast \gcd \ p \ q) =\)
      \((q \ast \gcd \ r \ s) \ast (\text{sgn} \ (q \ast \ s) \ast r \ast \gcd \ p \ q)\)
    by simp
    with assms show \(?\)thesis
    by (auto simp add: ac-simps sgn-mult sgn-0-0)
  qed
  from assms show \(?\)thesis
  by (auto simp: normalize-def Let-def dvd-div-div-eq-mult mult.commute sgn-mult)
split: if-splits intro: *)

qed

lemma normalize-eq: normalize (a, b) = (p, q) \implies Fract p q = Fract a b
by (auto simp: normalize-def Let-def Fract-coprime dvd-div-neg rat-number-collapse
split: if-split-asm)

lemma normalize-denom-pos: normalize r = (p, q) \implies q > 0
by (auto simp: normalize-def Let-def dvd-div-neg pos-imp-zdiv-neg-iff nonneg1-imp-zdiv-pos-iff
split: if-split-asm)

lemma normalize-coprime: normalize r = (p, q) \implies coprime p q
by (auto simp: normalize-def Let-def dvd-div-neg div-gcd-coprime split: if-split-asm)

lemma normalize-stable [simp]: q > 0 \implies coprime p q \implies normalize (p, q) = (p, q)
by (simp add: normalize-def)

lemma normalize-denom-zero [simp]: normalize (p, 0) = (0, 1)
by (simp add: normalize-def

lemma normalize-negative [simp]: q < 0 \implies normalize (p, q) = normalize (−p, −q)
by (simp add: normalize-def Let-def dvd-div-neg dvd-neg-div)

Decompose a fraction into normalized, i.e. coprime numerator and denominator:

definition quotient-of :: rat \Rightarrow int \times int
where quotient-of x = (THE pair. x = Fract (fst pair) (snd pair) \land snd pair > 0 \land coprime (fst pair) (snd pair))

lemma quotient-of-unique: \exists! p. r = Fract (fst p) (snd p) \land snd p > 0 \land coprime (fst p) (snd p)
proof (cases r)
case (Fract a b)
then have r = Fract (fst (a, b)) (snd (a, b)) \land
snd (a, b) > 0 \land coprime (fst (a, b)) (snd (a, b))
by auto
then show \?thesis
proof (rule ex1I)
fix p
assume r: r = Fract (fst p) (snd p) \land snd p > 0 \land coprime (fst p) (snd p)
obtain c d where p = (c, d) by (cases p)
with r have Fract': r = Fract c d > 0 \ coprime c d
by simp-all
have (c, d) = (a, b)
proof (cases a = 0)
case True
with Fract Fract' show ?thesis 
  by (simp add: eq-rat)
next
  case False
  with Fract Fract' have *: c * b = a * d and c ≠ 0
    by (auto simp add: eq-rat)
  then have c * b > 0 ⟷ a * d > 0
    by auto
  with ⟨b > 0⟩ ⟨d > 0⟩ have a > 0 ⟷ c > 0
    by (simp add: zero-less-mult-iff)
  with ⟨a ≠ 0⟩ ⟨c ≠ 0⟩ have sgn: sgn a = sgn c
    by (auto simp add: not-less)
  then have c * b > 0 ←→ a * d > 0
    by auto
  with ⟨b > 0⟩ ⟨d > 0⟩ have a > 0 ←→ c > 0
    by (simp add: zero-less-mult-iff)
  with ⟨a ≠ 0⟩ ⟨c ≠ 0⟩ have c * b > 0 ←→ a * d > 0
    by auto
  from ⟨coprime a b⟩ ⟨coprime c d⟩ have
    |a| * |d| = |c| * |b| ⟷ |a| = |c| ∧ |d| = |b|
    by (simp add: coprime-crossproduct-int)
  with ⟨b > 0⟩ ⟨d > 0⟩ have a > 0 ←→ c > 0
    by simp
  then have a * d = c * b ←→ a * c = b
    by (simp add: abs-sgn)
  with sgn * show ?thesis
    by (auto simp add: sgn-0-0)
qed
with p show p = (a, b)
  by simp
qed

lemma quotient-of-Fract [code]: quotient-of (Fract a b) = normalize (a, b)
proof
  have Fract a b = Fract (fst (normalize (a, b))) (snd (normalize (a, b))) (is ?Fract)
    by (rule sym) (auto intro: normalize-eq)
  moreover have 0 < snd (normalize (a, b)) is ?denom-pos
    by (cases normalize (a, b)) (rule normalize-denom-pos, simp)
  moreover have coprime (fst (normalize (a, b))) (snd (normalize (a, b))) (is ?coprime)
    by (rule normalize-coprime) simp
  ultimately have ?Fract ∧ ?denom-pos ∧ ?coprime by blast
  then have (THE p. Fract a b = Fract (fst p) (snd p) ∧ 0 < snd p ∧
    coprime (fst p) (snd p)) = normalize (a, b)
    by (rule the1-equality [OF quotient-of-unique])
  then show ?thesis by (simp add: quotient-of-def)
qed

lemma quotient-of-number [simp]:
  quotient-of 0 = (0, 1)
  quotient-of 1 = (1, 1)
  quotient-of (numeral k) = (numeral k, 1)
  quotient-of (−1) = (−1, 1)
quotient-of \((-\text{numeral } k)\) = \((-\text{numeral } k, 1)\)
by (simp-all add: rat-number-expand quotient-of-Fract)

lemma quotient-of-eq: quotient-of \((\text{Fract } a \ b)\) = \((p, q)\) \implies \text{Fract } p \ q = \text{Fract } a \ b
by (simp add: quotient-of-Fract normalize-eq)

lemma quotient-of-denom-pos: quotient-of \(r\) = \((p, q)\) \implies q > 0
by (cases \text{Fract } r) (simp add: quotient-of-Fract normalize-denom-pos)

lemma quotient-of-denom-pos': snd (quotient-of \(r\)) > 0
using quotient-of-denom-pos [of \(r\)] by (simp add: prod-eq-iff)

lemma quotient-of-coprime: quotient-of \(r\) = \((p, q)\) \implies \text{coprime } p \ q
by (cases \text{Fract } r) (simp add: quotient-of-Fract normalize-coprime)

lemma quotient-of-inject:
assumes quotient-of \(a\) = quotient-of \(b\)
shows \(a\) = \(b\)
proof
obtain \(p\) \(q\) \(r\) \(s\) where \(a\): \(\text{Fract } p \ q\) and \(b\): \(\text{Fract } r \ s\) and \(q > 0\) and \(s > 0\)
by (cases \(a\), cases \(b\))
with assms show \(\?\text{thesis}\)
by (simp add: eq-rat quotient-of-Fract normalize-crossproduct)
qed

lemma quotient-of-inject-eq: quotient-of \(a\) = quotient-of \(b\) \iff \(a\) = \(b\)
by (auto simp add: quotient-of-inject)

94.1.4 Various

lemma Fract-of-int-quotient: \(\text{Fract } k \ l\) = \(\text{of-int } k \ / \ \text{of-int } l\)
by (simp add: Fract-of-int-eq [symmetric])

lemma Fract-add-one: \(n \neq 0\) \implies \(\text{Fract } (m + n)\) \(n\) = \(\text{Fract } m \ n + 1\)
by (simp add: rat-number-expand)

lemma quotient-of-div:
assumes \(r\): quotient-of \(r\) = \((n,d)\)
shows \(r\) = \(\text{of-int } n \ / \ \text{of-int } d\)
proof
from theI \([OF \ \text{quotient-of-unique}[of \ r], \text{unfolded } r][\text{unfolded quotient-of-def}]\)
have \(r\) = \(\text{Fract } n \ d\) by simp
then show \(\?\text{thesis}\) using Fract-of-int-quotient
by simp
qed

94.1.5 The ordered field of rational numbers

lift-definition positive :: \(\text{rat} \Rightarrow \text{bool}\)
is $\lambda x. \, 0 < \text{fst } x \ast \text{snd } x$

proof clarsimp
  fix $a \, b \, c \, d :: \text{int}$
  assume $b \neq 0$ and $d \neq 0$ and $a \ast d = c \ast b$
  then have $a \ast d \ast b \ast d = c \ast b \ast b \ast d$
    by simp
  then have $a \ast b \ast d^2 = c \ast d \ast b^2$
    unfolding power2-eq-square by (simp add: ac-simps)
  then have $0 < a \ast b \ast d^2 \iff 0 < c \ast d \ast b^2$
    by simp
  then show $0 < a \ast b \iff 0 < c \ast d$
    using $\langle b \neq 0 \rangle$ and $\langle d \neq 0 \rangle$
    by (simp add: zero-less-mult-iff)
qed

lemma positive-zero: $\neg$ positive 0
  by transfer simp

lemma positive-add: positive $x \implies$ positive $y \implies$ positive ($x + y$)
  apply transfer
  apply (simp add: zero-less-mult-iff)
  apply (elim disjE)
  apply (simp-all add: add-pos-pos add-neg-neg mult-pos-neg mult-neg-pos mult-neg-neg)
  done

lemma positive-mult: positive $x \implies$ positive $y \implies$ positive ($x \ast y$)
  apply transfer
  apply (drule (1) mult-pos-pos)
  apply (simp add: ac-simps)
  done

lemma positive-minus: $\neg$ positive $x \implies x \neq 0 \implies$ positive ($-x$)
  by transfer (auto simp: neq-iff zero-less-mult-iff mult-less-0-iff)

instantiation rat :: linordered-field
begin

definition $x < y \iff$ positive ($y - x$)

definition $x \leq y \iff x < y \lor x = y$ for $x \, y :: \text{rat}$

definition $|a| = (\text{if } a < 0 \text{ then } -a \text{ else } a)$ for $a :: \text{rat}$

definition $\text{sgn } a = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } 1 \text{ else } -1)$ for $a :: \text{rat}$

instance
proof
  fix $a \, b \, c :: \text{rat}$
  show $|a| = (\text{if } a < 0 \text{ then } -a \text{ else } a)$
by (rule abs-rat-def)
show \( a < b \iff a \leq b \land \neg b \leq a \)
  unfolding less-eq-rat-def less-rat-def
  apply auto
  apply (drule (1) positive-add)
  apply (simp-all add: positive-zero)
  done
show \( a \leq a \)
  unfolding less-eq-rat-def by simp
show \( a \leq b \implies b \leq c \implies a \leq c \)
  unfolding less-eq-rat-def less-rat-def
  apply auto
  apply (drule (1) positive-add)
  apply (simp add: algebra-simps)
  done
show \( a \leq b \implies b \leq a \implies a = b \)
  unfolding less-eq-rat-def less-rat-def
  apply auto
  apply (drule (1) positive-add)
  apply (simp add: positive-zero)
  done
show \( a \leq b \implies c + a \leq c + b \)
  unfolding less-eq-rat-def less-rat-def by auto
show sgn \( a \) = (if \( a = 0 \) then 0 else if \( 0 < a \) then 1 else \( -1 \))
  by (rule sgn-rat-def)
show \( a \leq b \lor b \leq a \)
  unfolding less-eq-rat-def less-rat-def
  by (auto dest!: positive-minus)
show \( a < b \implies \theta < c \implies c \cdot a < c \cdot b \)
  unfolding less-rat-def
  apply (drule (1) positive-mul)
  apply (simp add: algebra-simps)
  done
qed

end

instantiation rat :: distrib-lattice
begin

definition (inf :: rat \Rightarrow rat \Rightarrow rat) = min

definition (sup :: rat \Rightarrow rat \Rightarrow rat) = max

instance
  by standard (auto simp add: inf-rat-def sup-rat-def max-min-distrib2)
end
lemma positive-rat: positive (Fract a b) \iff 0 < a * b
  by transfer simp

lemma less-rat [simp]:
  b \neq 0 \implies d \neq 0 \implies Fract a b < Fract c d \iff (a * d) * (b * d) < (c * b) * (b * d)
  by (simp add: less-rat-def positive-rat algebra-simps)

lemma le-rat [simp]:
  b \neq 0 \implies d \neq 0 \implies Fract a b \leq Fract c d \iff (a * d) * (b * d) \leq (c * b) * (b * d)
  by (simp add: le-less eq-rat)

lemma abs-rat [simp, code]: |Fract a b| = Fract |a| |b|
  by (auto simp add: abs-rat-def zabs_def Zero-rat-def not-less le-less zero-less-mult-iff)

lemma sgn-rat [simp, code]: sgn (Fract a b) = of-int (sgn a * sgn b)
  unfolding Fract-of-int-eq
  by (auto simp add: zsgn_def sgn-rat-def Zero-rat-def eq-rat)
    (auto simp: rat-number-collapse not-less le-less zero-less-mult-iff)

lemma Rat-induct-pos [case-names Fract, induct type: rat]:
  assumes step: \\( \forall a b. 0 < b \implies P (Fract a b) \) shows P q
  proof (cases q)
    case (Fract a b)
    have step': P (Fract a b) if b: b < 0 for a b :: int
      proof
        from b have 0 < ~ b
          by simp
        then have P (Fract (~ a) (~ b))
          by (rule step)
        then show P (Fract a b)
          by (simp add: order-less-imp-not-eq [OF b])
      qed
    from Fract show P q
      by (auto simp add: linorder-neq_iff step step')
  qed

lemma zero-less-Fract-iff: 0 < b \implies 0 < Fract a b \iff 0 < a
  by (simp add: Zero-rat-def zero-less-mult-iff)

lemma Fract-less-zero-iff: 0 < b \implies Fract a b < 0 \iff a < 0
  by (simp add: Zero-rat-def mult-less-0-iff)

lemma zero-le-Fract-iff: 0 < b \implies 0 \leq Fract a b \iff 0 \leq a
  by (simp add: Zero-rat-def zero-le-mult-iff)

lemma Fract-le-zero-iff: 0 < b \implies Fract a b \leq 0 \iff a \leq 0
by (simp add: Zero-rat-def mult-le-0-iff)

lemma one-less-Fract: \(0 < b \iff 1 < \text{Fract} \ a \ b \iff b < a\)
by (simp add: One-rat-def mult-less-cancel-right-disj)

lemma Fract-less-one-iff: \(0 < b \iff \text{Fract} \ a \ b < 1 \iff a < b\)
by (simp add: One-rat-def mult-less-cancel-right-disj)

lemma one-le-Fract-iff: \(0 < b \iff 1 \leq \text{Fract} \ a \ b \iff b \leq a\)
by (simp add: One-rat-def mult-le-cancel-right)

lemma Fract-le-one-iff: \(0 < b \iff \text{Fract} \ a \ b \leq 1 \iff a \leq b\)
by (simp add: One-rat-def mult-le-cancel-right)

94.1.6 Rationals are an Archimedean field

lemma rat-floor-lemma: \(\text{of-int} \ (a \div b) \leq \text{Fract} \ a \ b \wedge \text{Fract} \ a \ b < \text{of-int} \ (a \div b + 1)\)
proof –
have \(\text{Fract} \ a \ b = \text{of-int} \ (a \div b) + \text{Fract} \ (a \mod b) \ b\)
by (cases \(b = 0\)) (simp, simp add: of-int-rat)
moreover have \(0 \leq \text{Fract} \ (a \mod b) \ b \wedge \text{Fract} \ (a \mod b) \ b < 1\)
unfolding Fract-of-int-quotient
by (rule linorder-cases [of \(b \ 0\)]) (simp-all add: divide-nonpos-neg)
ultimately show \(?thesis\) by simp
qed

instance rat :: archimedean-field
proof
show \(\exists \ z. \ r \leq \text{of-int} \ z \text{ for } r :: \text{rat}\)
proof (induct \(r\))
case (Fract \(a \ b\))
have \(\text{Fract} \ a \ b \leq \text{of-int} \ (a \div b + 1)\)
using rat-floor-lemma \(\text{[of} \ a \ b\] \text{ by simp}\)
then show \(\exists \ z. \text{Fract} \ a \ b \leq \text{of-int} \ z \text{ ..}\)
qed
qed

instantiation rat :: floor-ceiling
begin

definition floor-rat :: \(\text{rat} \Rightarrow \text{int}\)
where\([x] = (\text{THE} \ z. \ \text{of-int} \ z \leq x \wedge x < \text{of-int} \ (z + 1))\) \(\text{for } x :: \text{rat}\)

instance
proof
show \(\text{of-int} \ \lfloor x \rfloor \leq x \wedge x < \text{of-int} \ (\lfloor x \rfloor + 1)\) \(\text{for } x :: \text{rat}\)
unfolding floor-rat-def using floor-existsI by (rule theI')
qed
end

lemma floor-Fract [simp]: ⌊Fract a b⌋ = a div b
  by (simp add: Fract-of-int-quotient floor-divide-of-int-eq)

94.2 Linear arithmetic setup

declaration
K (Lin-Arith.add-inj-thms @{thms of-int-le-iff [THEN iffD2] of-int-eq-iff [THEN iffD2]})
  (* not needed because x < (y::int) can be rewritten as x + 1 <= y: of-int-less-iff
   RS iffD2 *)
  #> Lin-Arith.add-inj-const (const-name of-rat, typ nat ⇒ rat)
  #> Lin-Arith.add-inj-const (const-name of-int, typ int ⇒ rat))

94.3 Embedding from Rationals to other Fields

category field-char-0

begin

lift-definition of-rat :: rat ⇒ ′a
  is λx. of-int (fst x) / of-int (snd x)
  by (auto simp: nonzero-divide-eq-eq nonzero-eq-divide-eq) (simp only: of-int-mult [symmetric])

end

lemma of-rat-rat: b ≠ 0 → of-rat (Fract a b) = of-int a / of-int b
  by transfer simp

lemma of-rat-0 [simp]: of-rat 0 = 0
  by transfer simp

lemma of-rat-1 [simp]: of-rat 1 = 1
  by transfer simp

lemma of-rat-add: of-rat (a + b) = of-rat a + of-rat b
  by transfer (simp add: add-frac-eq)

lemma of-rat-minus: of-rat (− a) = − of-rat a
  by transfer simp

lemma of-rat-neg-one [simp]: of-rat (− 1) = − 1
  by (simp add: of-rat-minus)

lemma of-rat-diff: of-rat (a − b) = of-rat a − of-rat b
  using of-rat-add [of a − b] by (simp add: of-rat-minus)
lemma of-rat-mult: \( \text{of-rat} \ (a \times b) = \text{of-rat} \ a \times \text{of-rat} \ b \)
by transfer \(\text{(simp add: divide-inverse nonzero-inverse-mult-distrib ac-simps)}\)

lemma of-rat-sum: \( \text{of-rat} \ \left( \sum a \in A. \ f a \right) = \left( \sum a \in A. \ \text{of-rat} \ (f a) \right) \)
by \(\text{(induct rule: infinite-finite-induct)}\) \(\text{(auto simp: of-rat-add)}\)

lemma of-rat-prod: \( \text{of-rat} \ \left( \prod a \in A. \ f a \right) = \left( \prod a \in A. \ \text{of-rat} \ (f a) \right) \)
by \(\text{(induct rule: infinite-finite-induct)}\) \(\text{(auto simp: of-rat-mult)}\)

lemma nonzero-of-rat-inverse: \( a \neq 0 \implies \text{of-rat} \ \left( \text{inverse} \ a \right) = \text{inverse} \ \left( \text{of-rat} \ a \right) \)
by \(\text{(rule inverse-unique [symmetric])}\) \(\text{(simp add: of-rat-mult [symmetric])}\)

lemma of-rat-inverse: \( \text{of-rat} \ \left( \text{inverse} \ a \right) \) :: \( \text{\'a::field-char-0} \) = \( \text{inverse} \ \left( \text{of-rat} \ a \right) \)
by \(\text{(cases \ a = 0)}\) \(\text{(simp-all add: nonzero-of-rat-inverse)}\)

lemma nonzero-of-rat-divide: \( b \neq 0 \implies \text{of-rat} \ \left( \frac{a}{b} \right) = \frac{\text{of-rat} \ a}{\text{of-rat} \ b} \)
by \(\text{(simp add: divide-inverse of-rat-mult nonzero-of-rat-inverse)}\)

lemma of-rat-divide: \( \text{of-rat} \ \left( \frac{a}{b} \right) \) :: \( \text{\'a::field-char-0} \) = \( \frac{\text{of-rat} \ a}{\text{of-rat} \ b} \)
by \(\text{(cases \ b = 0)}\) \(\text{(simp-all add: nonzero-of-rat-divide)}\)

lemma of-rat-power: \( \text{of-rat} \ \left( a \ ^{n} \right) \) :: \( \text{\'a::field-char-0} \) = \( a \ ^{n} \)
by \(\text{(induct \ n)}\) \(\text{(simp-all add: of-rat-mult)}\)

lemma of-rat-eq-iff \([\text{simp}]\): \( \text{of-rat} \ a = \text{of-rat} \ b \) \(\iff\) \( a = b \)
apply transfer
apply \(\text{(simp add: nonzero-divide-eq-eq nonzero-eq-divide-eq)}\)
apply \(\text{(simp only: of-int-mult [symmetric] of-int-eq-iff)}\)
done

lemma of-rat-eq-0-iff \([\text{simp}]\): \( \text{of-rat} \ a = 0 \) \(\iff\) \( a = 0 \)
using \(\text{of-rat-eq-iff [of - 0]}\) by simp

lemma zero-eq-of-rat-iff \([\text{simp}]\): \( 0 = \text{of-rat} \ a \) \(\iff\) \( 0 = a \)
by simp

lemma of-rat-eq-1-iff \([\text{simp}]\): \( \text{of-rat} \ a = 1 \) \(\iff\) \( a = 1 \)
using \(\text{of-rat-eq-iff [of - 1]}\) by simp

lemma one-eq-of-rat-iff \([\text{simp}]\): \( 1 = \text{of-rat} \ a \) \(\iff\) \( 1 = a \)
by simp

lemma of-rat-less: \( \text{of-rat} \ r :: \text{\'a::linordered-field} < \text{of-rat} \ s \) \(\iff\) \( r < s \)
proof \(\text{(induct \ r, induct \ s)}\)
fix \( a \ b \ c \ d :: \text{int} \)
assume \text{not-zero}: \( b > 0 \ d > 0 \)
then have \( b \ast d > 0 \) by simp
have \(\text{of-int-divide-less-eq}: \)
\( \text{(of-int} \ a :: \text{\'a)} / \ \text{of-int} \ b < \ \text{of-int} \ c / \ \text{of-int} \ d \) \(\iff\)
THEORY "Rat"

(of-int a :: 'a) * of-int d < of-int c * of-int b
using not-zero by (simp add: pos-less-divide-eq pos-divide-less-eq)
show (of-rat (Fract a b) :: 'a::linordered-field) < of-rat (Fract c d) \iff
Fract a b < Fract c d
using not-zero (b * d > 0)
qed

lemma of-rat-less-eq: (of-rat r :: 'a::linordered-field) ≤ of-rat s \iff r ≤ s
unfolding le-less by (auto simp add: of-rat-less)
lemma of-rat-le-0-iff [simp]: (of-rat r :: 'a::linordered-field) ≤ 0 \iff r ≤ 0
using of-rat-less-eq [of r 0, where 'a = 'a] by simp
lemma of-rat-le-1-iff [simp]: (of-rat r :: 'a::linordered-field) ≤ 1 \iff r ≤ 1
using of-rat-less-eq [of r 1] by simp
lemma one-le-of-rat-iff [simp]: 1 ≤ (of-rat r :: 'a::linordered-field) \iff 1 ≤ r
using of-rat-less-eq [of 1 r] by simp
lemma of-rat-less-0-iff [simp]: (of-rat r :: 'a::linordered-field) < 0 \iff r < 0
using of-rat-less [of r 0, where 'a = 'a] by simp
lemma zero-less-of-rat-iff [simp]: 0 < (of-rat r :: 'a::linordered-field) \iff 0 < r
using of-rat-less [of 0 r, where 'a = 'a] by simp
lemma of-rat-less-1-iff [simp]: (of-rat r :: 'a::linordered-field) < 1 \iff r < 1
using of-rat-less [of 1 r] by simp
lemma one-less-of-rat-iff [simp]: 1 < (of-rat r :: 'a::linordered-field) \iff 1 < r
using of-rat-less [of 1 r] by simp
lemma of-rat-eq-id [simp]: of-rat = id
proof
  show of-rat a = id a for a
    by (induct a) (simp add: of-rat-rat Fract-of-int-eq [symmetric])
qed
lemma abs-of-rat [simp]:
|of-rat r| = (of-rat |r| :: 'a::linordered-field)
by (cases r ≥ 0) (simp-all add: not-le of-rat-minus)

Collapse nested embeddings.
lemma of-rat-of-nat-eq [simp]: of-rat (of-nat n) = of-nat n
by (induct n) (simp-all add: of-rat-add)
lemmas of-rat-of-int-eq [simp]: of-rat (of-int z) = of-int z
  by (cases z rule: int-diff-cases) (simp add: of-rat-diff)

lemma of-rat-numeral-eq [simp]: of-rat (numeral w) = numeral w
  using of-rat-of-int-eq [of numeral w] by simp

lemma of-rat-neg-numeral-eq [simp]: of-rat (− numeral w) = − numeral w
  using of-rat-of-int-eq [of − numeral w] by simp

lemma of-rat-floor [simp]:
  ⌊of-rat r⌋ = ⌊r⌋
  by (cases r) (simp add: Fract-of-int-quotient of-rat-divide floor-divide-of-int-eq)

lemma of-rat-ceiling [simp]:
  ⌈of-rat r⌉ = ⌈r⌉
  using of-rat-floor [of − r] by (simp add: of-rat-minus ceiling-def)

lemmas zero-rat = Zero-rat-def
lemmas one-rat = One-rat-def

abbreviation rat-of-nat :: nat ⇒ rat
  where rat-of-nat ≡ of-nat

abbreviation rat-of-int :: int ⇒ rat
  where rat-of-int ≡ of-int

94.4 The Set of Rational Numbers

context field-char-0
begin

definition Rats :: 'a set (Q)
  where Q = range of-rat
end

lemma Rats-cases [cases set: Rats]:
  assumes q ∈ Q
  obtains (of-rat) r where q = of-rat r
  proof
    from (q ∈ Q) have q ∈ range of-rat
      by (simp only: Rats-def)
    then obtain r where q = of-rat r ..
    then show thesis ..
  qed

lemma Rats-of-rat [simp]: of-rat r ∈ Q
  by (simp add: Rats-def)
lemma Rats-of-int [simp]: of-int z ∈ Q
   by (subst of-rat-of-int-eq [symmetric]) (rule Rats-of-rat)

lemma Ints-subset-Rats: ℤ ⊆ Q
   using Ints-cases Rats-of-int by blast

lemma Rats-of-nat [simp]: of-nat n ∈ Q
   by (subst of-rat-of-nat-eq [symmetric]) (rule Rats-of-rat)

lemma Nats-subset-Rats: ℤ ⊆ Q
   using Ints-subset-Rats Nats-subset-Ints by blast

lemma Rats-number-of [simp]: numeral w ∈ Q
   by (subst of-rat-numeral-eq [symmetric]) (rule Rats-of-rat)

lemma Rats-0 [simp]: 0 ∈ Q
   unfolding Rats-def by (rule range-eqI) (rule of-rat-0 [symmetric])

lemma Rats-1 [simp]: 1 ∈ Q
   unfolding Rats-def by (rule range-eqI) (rule of-rat-1 [symmetric])

lemma Rats-add [simp]: a ∈ Q ⇒ b ∈ Q ⇒ a + b ∈ Q
   apply (auto simp add: Rats-def)
   apply (rule range-eqI)
   apply (rule of-rat-add [symmetric])
   done

lemma Rats-minus-iff [simp]: − a ∈ Q ↔ a ∈ Q
   by (metis Rats-cases Rats-of-rat add.inverse-inverse of-rat-minus)

lemma Rats-diff [simp]: a ∈ Q ⇒ b ∈ Q ⇒ a − b ∈ Q
   apply (auto simp add: Rats-def)
   apply (rule range-eqI)
   apply (rule of-rat-diff [symmetric])
   done

lemma Rats-mult [simp]: a ∈ Q ⇒ b ∈ Q ⇒ a ∗ b ∈ Q
   apply (auto simp add: Rats-def)
   apply (rule range-eqI)
   apply (rule of-rat-mult [symmetric])
   done

lemma Rats-inverse [simp]: a ∈ Q ⇒ inverse a ∈ Q
   for a :: ′a::field-char-0
   apply (auto simp add: Rats-def)
   apply (rule range-eqI)
   apply (rule of-rat-inverse [symmetric])
   done
lemma Rats-divide [simp]: \( a \in \mathbb{Q} \implies b \in \mathbb{Q} \implies a/b \in \mathbb{Q} \)

for \( a, b :: 'a::field-char-0 \)
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (rule of-rat-divide [symmetric])
done

lemma Rats-power [simp]: \( a \in \mathbb{Q} \implies a^n \in \mathbb{Q} \)

for \( a :: 'a::field-char-0 \)
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (rule of-rat-power [symmetric])
done

lemma Rats-induct [case-names of-rat, induct set: Rats]: \( q \in \mathbb{Q} \implies (\forall r. P \, (of-rat \, r)) \implies P \, q \)
by (rule Rats-cases) auto

lemma Rats-infinite: \( \neg \text{finite } \mathbb{Q} \)
by (auto dest: finite-imageD simp: inj-on-def infinite-UNIV-char-0 Rats-def)

94.5 Implementation of rational numbers as pairs of integers

Formal constructor

definition Frct :: int \times int \Rightarrow rat
where [simp]: Frct \, p = Fract \, (fst \, p) \, (snd \, p)

lemma [code abstype]: Frct \, (quotient-of q) = q
by (cases q) (auto intro: quotient-of_eq)

Numerals
declare quotient-of-Fract [code abstract]

definition of-int :: int \Rightarrow rat
where [code-abbrev]: of-int = Int.of-int

hide-const (open) of-int

lemma quotient-of-int [code abstract]: quotient-of \, (Rat.of-int \, a) = (a, 1)
by (simp add: of-int-def of-int-rat quotient-of-Fract)

lemma [code-unfold]: numeral k = Rat.of-int \, (numeral k)
by (simp add: Rat.of-int-def)

lemma [code-unfold]: \(-\, numeral \, k = Rat.of-int \, (-\, numeral \, k)
by (simp add: Rat.of-int-def)

lemma Frct-code-post [code-post]:
Frct \, (0, a) = 0
THEORY “Rat”

Frct (a, 0) = 0
Frct (1, 1) = 1
Frct (numeral k, 1) = numeral k
Frct (1, numeral k) = 1 / numeral k
Frct (numeral k, numeral l) = numeral k / numeral l
Frct (−a, b) = −Frct (a, b)
Frct (a, −b) = −Frct (a, b)
−(−Frct q) = Frct q
by (simp-all add: Fract-of-int-quotient)

Operations

lemma rat-zero-code [code abstract]: quotient-of 0 = (0, 1)
  by (simp add: Zero-rat-def quotient-of-Fract normalize-def)
lemma rat-one-code [code abstract]: quotient-of 1 = (1, 1)
  by (simp add: One-rat-def quotient-of-Fract normalize-def)
lemma rat-plus-code [code abstract]:
  quotient-of (p + q) = (let (a, c) = quotient-of p; (b, d) = quotient-of q
  in normalize (a * d + b * c, c * d))
  by (cases p, cases q) (simp add: quotient-of-Fract)
lemma rat-uminus-code [code abstract]:
  quotient-of (−p) = (let (a, b) = quotient-of p in (−a, b))
  by (cases p) (simp add: quotient-of-Fract)
lemma rat-minus-code [code abstract]:
  quotient-of (p − q) =
  (let (a, c) = quotient-of p; (b, d) = quotient-of q
  in normalize (a * d − b * c, c * d))
  by (cases p, cases q) (simp add: quotient-of-Fract)
lemma rat-times-code [code abstract]:
  quotient-of (p * q) =
  (let (a, c) = quotient-of p; (b, d) = quotient-of q
  in normalize (a * b, c * d))
  by (cases p, cases q) (simp add: quotient-of-Fract)
lemma rat-inverse-code [code abstract]:
  quotient-of (inverse p) =
  (let (a, b) = quotient-of p
  in if a = 0 then (0, 1) else (sgn a * b, |a|))
proof (cases p)
case (Fract a b)
then show ?thesis
  by (cases 0::int a rule: linorder-cases) (simp add: quotient-of-Fract ac-simps)
qed

lemma rat-divide-code [code abstract]:
quotient-of \( (p / q) \) =
\[
\begin{align*}
& \text{(let } \(a, c\) = \text{quotient-of } p; \(b, d\) = \text{quotient-of } q \\
& \text{in normalize } (a * d, c * b)) \end{align*}
\]
by (cases \(p, q\)) (simp add: quotient-of-Fract)

**lemma** rat-abs-code [code abstract]:
\[
\text{quotient-of } |p| = \text{(let } \(a, b\) = \text{quotient-of } p \text{ in } (|a|, b))
\]
by (cases \(p\)) (simp add: quotient-of-Fract)

**proof** (cases \(p\))
\[
\text{case } \text{(Fract } a b) \\
\text{then show } ?\text{thesis}
\]
by (cases 0::int \(a\) rule: linorder-cases) (simp-all add: quotient-of-Fract)
qed

**lemma** rat-floor-code [code]: \(\lfloor p \rfloor\) = \text{(let } \(a, b\) = \text{quotient-of } p \text{ in } a \div b)
by (cases \(p\)) (simp add: quotient-of-Fract floor-Fract)

**instantiation** rat :: equal
begin

**definition** [code]: \(\text{HOL.equal } a b \iff \text{quotient-of } a = \text{quotient-of } b\)

**instance**
by standard (simp add: equal-rat-def quotient-of-inject-eq)

**lemma** rat-eq-refl [code nbe]: \(\text{HOL.equal } (\text{rat } r) \iff \text{True}\)
by (rule equal-refl)
end

**lemma** rat-less-eq-code [code]:
\(p \leq q \iff \text{(let } \(a, c\) = \text{quotient-of } p; \(b, d\) = \text{quotient-of } q \text{ in } a * d \leq c * b)\)
by (cases \(p, q\)) (simp add: quotient-of-Fract mult.commute)

**lemma** rat-less-code [code]:
\(p < q \iff \text{(let } \(a, c\) = \text{quotient-of } p; \(b, d\) = \text{quotient-of } q \text{ in } a * d < c * b)\)
by (cases \(p, q\)) (simp add: quotient-of-Fract mult.commute)

**lemma** [code]: \(\text{of-rat } p = \text{(let } \(a, b\) = \text{quotient-of } p \text{ in } \text{of-int } a / \text{of-int } b)\)
by (cases \(p\)) (simp add: quotient-of-Fract of-rat-rat)

**Quickcheck**

**definition** (in term-syntax)
\[
\begin{align*}
\text{valterm-fract :: } & \text{int } \times \text{ (unit } \Rightarrow \text{Code-Evaluation.term}) \Rightarrow \\
& \text{int } \times \text{ (unit } \Rightarrow \text{Code-Evaluation.term}) \Rightarrow \\
& \text{rat } \times \text{ (unit } \Rightarrow \text{Code-Evaluation.term})
\end{align*}
\]
where [code-unfold]: valterm-fract k l = Code-Evaluation.valtermify Fract {·} k {·} l

notation fcomp (infixl ◦> 60)
notation scomp (infixl ◦→ 60)

instantiation rat :: random
begin
definition
  Quickcheck-Random.random i =
  Quickcheck-Random.random i ◦> (\num. Random.range i ◦→ (\denom. Pair
  (let j = int-of-integer (integer-of-natural (denom + 1))
   in valterm-fract num (j, \u. Code-Evaluation.term-of j))))

instance ..
end

no-notation fcomp (infixl ◦> 60)
no-notation scomp (infixl ◦→ 60)

instantiation rat :: exhaustive
begin
definition
  exhaustive-rat f d =
  Quickcheck-Exhaustive.exhaustive
  (\l. Quickcheck-Exhaustive.exhaustive
   (\k. f (Fract k (int-of-integer (integer-of-natural l) + 1))) d) d

instance ..
end

instantiation rat :: full-exhaustive
begin
definition
  full-exhaustive-rat f d =
  Quickcheck-Exhaustive.full-exhaustive
  (\l. -). Quickcheck-Exhaustive.full-exhaustive
  (\k. f
   (let j = int-of-integer (integer-of-natural l) + 1
    in valterm-fract k (j, \-. Code-Evaluation.term-of j)) d) d

instance ..
end
instance  rat :: partial-term-of ..

lemma [code]:
  partial-term-of (ty :: rat itself) (Quickcheck-Narrowing.Narrowing-variable p tt) ≡
  Code-Evaluation.Free (STR "\(\cdot\)") (Typerep.Typerep (STR "Rat.rat") [])
  partial-term-of (ty :: rat itself) (Quickcheck-Narrowing.Narrowing-constructor 0 [l, k]) ≡
  Code-Evaluation.App
  (Code-Evaluation.Const (STR "Rat.Frac"))
  (Typerep.Typerep (STR "\(\cdot\)")
   [Typerep.Typerep (STR "Rat.rat") [], Typerep.Typerep (STR "Int.int") []],
   Typerep.Typerep (STR "Product-Type.prod")
   []),
  Typerep.Typerep (STR "Rat.rat") []))
  (Code-Evaluation.App
   (Code-Evaluation.App
    (Code-Evaluation.Const (STR "Product-Type.Pair")
     (Typerep.Typerep (STR "\(\cdot\)")
      [Typerep.Typerep (STR "Int.int") [],
       Typerep.Typerep (STR "\(\cdot\)")[]],
      Typerep.Typerep (STR "Product-Type.prod")
      []),
     Typerep.Typerep (STR "Int.int") [],
     Typerep.Typerep (STR "Product-Type.prod")
     [])))
  (partial-term-of (TYPE(int)) l)) (partial-term-of (TYPE(int)) k))
  by (rule partial-term-of-anything)+

instantiation  rat :: narrowing begin

definition
  narrowing =
  Quickcheck-Narrowing.apply
  (Quickcheck-Narrowing.apply
   (Quickcheck-Narrowing.cons (\(\lambda\)nom denom. Fract nom denom) narrowing)
  narrowing)

instance ..

end

94.6 Setup for Nitpick
declaration

  Nitpick-HOL.register-frac-type type-name (rat)
  [|\(\text{const-name}\) (Abs-Rat), \(\text{const-name}\) (Nitpick.Abs-Frac),
    \(\text{const-name}\) (zero-rat-inst.zero-rat), \(\text{const-name}\) (Nitpick.zero-frac)|]


THEORY "Rat"

1724

\( \text{const-name} \langle \text{one-rat-inst}. \text{one-rat} \rangle, \ \text{const-name} \langle \text{Nitpick}. \text{one-frac} \rangle, \) \\
\( \text{const-name} \langle \text{plus-rat-inst}. \text{plus-rat} \rangle, \ \text{const-name} \langle \text{Nitpick}. \text{plus-frac} \rangle, \) \\
\( \text{const-name} \langle \text{times-rat-inst}. \text{times-rat} \rangle, \ \text{const-name} \langle \text{Nitpick}. \text{times-frac} \rangle, \) \\
\( \text{const-name} \langle \text{uminus-rat-inst}. \text{uminus-rat} \rangle, \ \text{const-name} \langle \text{Nitpick}. \text{uminus-frac} \rangle, \) \\
\( \text{const-name} \langle \text{inverse-rat-inst}. \text{inverse-rat} \rangle, \ \text{const-name} \langle \text{Nitpick}. \text{inverse-frac} \rangle, \) \\
\( \text{const-name} \langle \text{ord-rat-inst}. \text{ord-rat} \rangle, \ \text{const-name} \langle \text{Nitpick}. \text{less-frac} \rangle, \) \\
\( \text{const-name} \langle \text{ord-rat-inst}. \text{less-eq-rat} \rangle, \ \text{const-name} \langle \text{Nitpick}. \text{less-eq-frac} \rangle, \) \\
\( \text{const-name} \langle \text{field-char-0-class}. \text{of-rat} \rangle, \ \text{const-name} \langle \text{Nitpick}. \text{of-frac} \rangle \) \\
\rangle

\lemmas [\text{nitpick-unfold}]= \\
\text{inverse-rat-inst}. \text{inverse-rat} \\
\text{one-rat-inst}. \text{one-rat} \ \text{ord-rat-inst}. \text{less-rat} \\
\text{ord-rat-inst}. \text{less-eq-rat} \ \text{plus-rat-inst}. \text{plus-rat} \ \text{times-rat-inst}. \text{times-rat} \\
\text{uminus-rat-inst}. \text{uminus-rat} \ \text{zero-rat-inst}. \text{zero-rat}

\section{94.7 Float syntax}

\textbf{syntax} -Float :: float-const ⇒ 'a  (-)

\textbf{parse-translation} < \let \\
\text{fun mk-frac str =} \\
\let \val \{mant = i, \ \exp = n\} = \text{Lexicon}.\text{read-float str}; \\
\val \exp = \text{Syntax}.\text{const} \langle \text{Power}. \text{power} \rangle; \\
\val ten = \text{Numeral}.\text{mk-number-syntax} 10; \\
\val \exp10 = \text{if} \ n = 1 \text{then} \ \text{ten} \ \text{else} \ \text{exp} \ \text{$ten$ \ Numeral}.\text{mk-number-syntax} \\
\text{n;} \\
\in \text{Syntax}.\text{const} \langle \text{Fields}. \text{inverse-divide} \rangle \ \text{$Numeral}.\text{mk-number-syntax} \\
\text{i $\exp10$ end}; \\
\text{fun float-tr [(c as Const}\ \langle \text{syntax-const} (-\text{constrain}), -\rangle) \ \text{$t$ $u$} = c \ \text{$float-tr$} \ [\text{t}] \ \text{$u$} \\
| \text{float-tr} \ [\text{t as Const}\ \langle \text{str}, -\rangle] = \text{mk-frac str} \\
| \text{float-tr} \ \text{ts} = \text{raise TERM} \ (\text{float-tr}, \ \text{ts}); \\
\text{in} \ [(\text{syntax-const} (-\text{Float}), \ K \text{float-tr}] \ \text{end}
>

Test:

\textbf{lemma} \ 123.456 = -111.111 + 200 + 30 + 4 + 5/10 + 6/100 + (7/1000::rat) \\
\text{by simp}

\section{94.8 Hiding implementation details}

\textbf{hide-const} (open) \textbf{normalize} positive

\textbf{lifting-update} \textbf{rat}.\textbf{lifting} \\
\textbf{lifting-forget} \textbf{rat}.\textbf{lifting}
Development of the Reals using Cauchy Sequences

95.1 Preliminary lemmas

Useful in convergence arguments

lemma inverse-of-nat-le:
  fixes n :: nat shows \[ n \leq m; n\neq 0 \] \implies 1 / of-nat m \leq (1::'a::linordered-field)
  by (simp add: frac-le)

lemma add-diff-add: \( (a + c) - (b + d) = (a - b) + (c - d) \)
  for a b c d :: 'a::ab-group-add
  by simp

lemma minus-diff-minus: \( -a - -b = - (a - b) \)
  for a b :: 'a::ab-group-add
  by simp

lemma mult-diff-mult: \( (x \cdot y - a \cdot b) = x \cdot (y - b) + (x - a) \cdot b \)
  for x y a b :: 'a::ring
  by (simp add: algebra-simps)

lemma inverse-diff-inverse:
  fixes a b :: 'a::division-ring
  assumes a \neq 0 and b \neq 0
  shows inverse a - inverse b = - (inverse a \cdot (a - b) \cdot inverse b)
  using assms by (simp add: algebra-simps)

lemma obtain-pos-sum:
  fixes r :: rat assumes r: 0 < r
  obtains s t where 0 < s and 0 < t and r = s + t
  proof
    from r show 0 < r/2 by simp
    from r show 0 < r/2 by simp
    show r = r/2 + r/2 by simp
95.2 Sequences that converge to zero

**Definition** vanishes :: (nat ⇒ rat) ⇒ bool

where vanishes X 🆸 → (∀ r>0. ∃ k. ∀ n≥k. |X n| < r)

**Lemma** vanishesI: (∀ r. 0 < r → ∃ k. ∀ n≥k. |X n| < r) → vanishes X

unfolding vanishes-def by simp

**Lemma** vanishesD: vanishes X → 0 < r → ∃ k. ∀ n≥k. |X n| < r

unfolding vanishes-def by simp

**Lemma** vanishes-const [simp]: vanishes (λn. c) ←→ c = 0

proof (cases c = 0)

case True

then show ?thesis

by (simp add: vanishesI)

next

case False

then show ?thesis

unfolding vanishes-def

using zero-less-abs-iff by blast

qed

**Lemma** vanishes-minus: vanishes X → vanishes (λn. − X n)

unfolding vanishes-def by simp

**Lemma** vanishes-add:

assumes X: vanishes X

and Y: vanishes Y

shows vanishes (λn. X n + Y n)

proof (rule vanishesI)

fix r :: rat

assume 0 < r

then obtain s t where s: 0 < s and t: 0 < t and r: r = s + t

by (rule obtain-pos-sum)

obtain i where i: ∀ n≥i. |X n| < s

using vanishesD [OF X s]..

obtain j where j: ∀ n≥j. |Y n| < t

using vanishesD [OF Y t]..

have ∀ n≥max i j. |X n + Y n| < r

proof clarsimp

fix n

assume n: i ≤ n j ≤ n

have |X n + Y n| ≤ |X n| + |Y n|

by (rule abs-triangle-ineq)

also have ... < s + t

by (simp add: add-strict-mono i j n)
finally show \(|X n + Y n| < r\)
by (simp only: r)
qed
then show \(\exists k. \forall n \geq k. |X n + Y n| < r\) ..
qed

lemma \textit{vanishes-diff}:
assumes \textit{vanishes} \(X\) \textit{vanishes} \(Y\)
shows \textit{vanishes} \((\lambda n. X n - Y n)\)
unfolding \textit{diff-conv-add-uminus} by (intro \textit{vanishes-add} \textit{vanishes-minus} \textit{assms})

lemma \textit{vanishes-mult-bounded}:
assumes \(X: \exists a > 0. \forall n. |X n| < a\)
assumes \(Y: \textit{vanishes} (\lambda n. Y n)\)
shows \textit{vanishes} \((\lambda n. X n \ast Y n)\)
proof (rule \textit{vanishesI})
fix \(r :: \text{rat}\)
assume \(r: 0 < r\)
obtain \(a\) where \(a: 0 < a \forall n. |X n| < a\)
using \(X\) by blast
obtain \(b\) where \(b: 0 < b \ast a = b\)
proof
  show \(0 < r / a\) using \(r a\) by simp
  show \(r = a \ast (r / a)\) using \(a\) by simp
qed
obtain \(k\) where \(k: \forall n \geq k. |Y n| < b\)
using \textit{vanishesD [OF Y b]} ..
have \(\forall n \geq k. |X n \ast Y n| < r\)
  by (simp add: b(2) \textit{abs-mult} \textit{mult-strict-mono'} \(a \ k\))
then show \(\exists k. \forall n \geq k. |X n \ast Y n| < r\) ..
qed

95.3 \textbf{Cauchy sequences}

\textbf{definition} \textit{cauchy} :: \((\text{nat} \Rightarrow \text{rat}) \Rightarrow \text{bool}\)
where \textit{cauchy} \(X \iff (\forall r > 0. \exists k. \forall m \geq k. \forall n \geq k. |X m - X n| < r)\)

\textbf{lemma} \textit{cauchyI}: \((\forall r. 0 < r \Longrightarrow \exists k. \forall m \geq k. \forall n \geq k. |X m - X n| < r) \Longrightarrow \textit{cauchy} X\)
unfolding \textit{cauchy-def} by simp

\textbf{lemma} \textit{cauchyD}: \textit{cauchy} \(X \Longrightarrow 0 < r \Longrightarrow \exists k. \forall m \geq k. \forall n \geq k. |X m - X n| < r\)
unfolding \textit{cauchy-def} by simp

\textbf{lemma} \textit{cauchy-const [simp]}: \textit{cauchy} \((\lambda n. x)\)
unfolding \textit{cauchy-def} by simp

\textbf{lemma} \textit{cauchy-add [simp]}:
assumes \(X: \textit{cauchy} X\) \textbf{and} \(Y: \textit{cauchy} Y\)
shows cauchy $(\lambda n. X n + Y n)$
proof (rule cauchyI)
  fix $r :: \text{rat}$
  assume $0 < r$
then obtain $s t$ where $s$: $0 < s$ and $t$: $0 < t$ and $r: r = s + t$
  by (rule obtain-pos-sum)
obtain $i$ where $i$: $\forall m \geq i. \forall n \geq i. \vert X m - X n \vert < s$
  using cauchyD [OF X s] ..
obtain $j$ where $j$: $\forall m \geq j. \forall n \geq j. \vert Y m - Y n \vert < t$
  using cauchyD [OF Y t] ..
have $\forall m \geq \max i j. \forall n \geq \max i j. \vert (X m + Y m) - (X n + Y n) \vert < r$
proof clarsimp
  fix $m n$
  assume $*: i \leq m j \leq m i \leq n j \leq n$
have $\vert (X m + Y m) - (X n + Y n) \vert \leq \vert X m - X n \vert + \vert Y m - Y n \vert$
  unfolding add-diff-add by (rule abs-triangle-ineq)
also have $\ldots < s + t$
  by (rule add-strict-mono) (simp-all add: $i j *$)
finally show $\vert (X m + Y m) - (X n + Y n) \vert < r$
  by (simp only: $r$)
qed
then show $\exists k. \forall m \geq k. \forall n \geq k. \vert (X m + Y m) - (X n + Y n) \vert < r$ ..
qed

lemma cauchy-minus [simp]:
  assumes $X$: cauchy $X$
shows cauchy $(\lambda n. - X n)$
  using assms unfolding cauchy-def
unfolding minus-diff-minus abs-minus-cancel .

lemma cauchy-diff [simp]:
  assumes cauchy $X$ cauchy $Y$
shows cauchy $(\lambda n. X n - Y n)$
  using assms unfolding diff-conv-add-uminus by (simp del: add-uminus-conv-diff)

lemma cauchy-imp-bounded:
  assumes cauchy $X$
shows $\exists b > 0. \forall n. \vert X n \vert < b$
proof
  obtain $k$ where $k$: $\forall m \geq k. \forall n \geq k. \vert X m - X n \vert < 1$
    using cauchyD [OF assms zero-less-one] ..
show $\exists b > 0. \forall n. \vert X n \vert < b$
  proof (intro exI conjI allI)
    have $0 \leq \vert X 0 \vert$ by simp
also have $\vert X 0 \vert \leq \max (\vert X \cdot \{..k\})$ by simp
finally have $0 \leq \max (\vert X \cdot \{..k\})$ .
then show $0 < \max (\vert X \cdot \{..k\}) + 1$ by simp
next
  fix $n :: \text{nat}$
show $\vert X n \vert < \max (\vert X \cdot \{..k\}) + 1$
proof (rule linorder-le-cases)
  assume n ≤ k
  then have |X n| ≤ Max (abs ' X ' [..k]) by simp
  then show |X n| < Max (abs ' X ' [..k]) + 1 by simp
next
  assume k ≤ n
  have |X n| = |X k + (X n - X k)| by simp
  also have |X k + (X n - X k)| ≤ |X k| + |X n - X k|
    by (rule abs-triangle-ineq)
  also have ... < Max (abs ' X ' [..k]) + 1
    by (rule add-le-less-mono) (simp-all add: k (k ≤ n))
  finally show |X n| < Max (abs ' X ' [..k]) + 1.
qed
qed

lemma cauchy-mult [simp]:
  assumes X: cauchy X and Y: cauchy Y
  shows cauchy (λn. X n * Y n)
proof (rule cauchyl)
  fix r :: rat assume θ < r
  then obtain u v where u: θ < u and v: θ < v and r = u + v
    by (rule obtain-pos-sum)
  obtain a where a: θ < a ∀ n. |X n| < a
    using cauchy-imp-bounded [OF X] by blast
  obtain b where b: θ < b ∀ n. |Y n| < b
    using cauchy-imp-bounded [OF Y] by blast
  obtain s t where s: θ < s and t: θ < t and r = a * t + s * b
    proof
      show 0 < v/b using v b(1) by simp
      show 0 < u/a using u a(1) by simp
      show r = a * (u/a) + (v/b) * b
        using a(1) b(1) r = a + v by simp
    qed
  obtain i where i: ∀ m≥i. ∀ n≥i. |X m − X n| < s
    using cauchyD [OF X s] ..
  obtain j where j: ∀ m≥j. ∀ n≥j. |Y m − Y n| < t
    using cauchyD [OF Y t] ..
  have ∀ m≥max i j. ∀ n≥max i j. |X m * Y m − X n * Y n| < r
    proof clarsimp
    fix m n
    assume *: i ≤ m j ≤ m i ≤ n j ≤ n
    have |X m * Y m − X n * Y n| = |X m * (Y m − Y n) + (X m − X n) * Y n|
    unfolding mult-diff-mult ..
    also have ... ≤ |X m * (Y m − Y n)| + |(X m − X n) * Y n|
      by (rule abs-triangle-ineq)
    also have ... = |X m| * |Y m − Y n| + |X m − X n| * |Y n|
    unfolding abs-mult ..
also have \( \ldots < a \cdot t + s \cdot b \)
by \((\text{simp-all add: add-strict-mono mult-strict-mono' a b i j}*)\)
finally show \(|X m \cdot Y m - X n \cdot Y n| < r\)
by \((\text{simp only: r})\)
qed
then show \(\exists k. \forall m \geq k. \forall n \geq k. |X m \cdot Y m - X n \cdot Y n| < r \ldots\)
qed

lemma \textit{cauchy-not-vanishes-cases}:
assumes \(X: \text{cauchy X}\)
assumes \(\text{nz}: \neg \text{vanishes X}\)
shows \(\exists b > 0. \exists k. (\forall n \geq k. b < -X n) \lor (\forall n \geq k. b < X n)\)
proof –
obtain \(r\) where \(0 < r\) and \(r\): \(\forall k. \exists n \geq k. r \leq |X n|\)
using \(\text{nz}\) unfolding \text{vanishes-def} by \((\text{auto simp add: not-less})\)
obtain \(s t\) where \(s: 0 < s\) and \(t: 0 < t\) and \(r = s + t\)
using \(0 < r\) by \((\text{rule obtain-pos-sum})\)
obtain \(i 0\) where \(i: \forall m \geq i. \forall n \geq i. |X m - X n| < s\)
using \text{cauchyD [OF X s]} ..
obtain \(k 0\) where \(i \leq k\) and \(r \leq |X k|\)
using \(r\) by \(\text{blast}\)
have \(\exists n \geq k. |X n - X k| < s\)
using \(i 0\) \(\forall i \leq k\) by \(\text{auto}\)
have \(X k \leq -r \lor r \leq X k\)
using \(|r \leq |X k||\) by \(\text{auto}\)
then have \(\forall n \geq k. t < -X n) \lor (\forall n \geq k. t < X n)\)
unfolding \(r = s + t\) using \(k\) by \(\text{auto}\)
then have \(\exists k. (\forall n \geq k. t < -X n) \lor (\forall n \geq k. t < X n)\) ..
then show \(\exists t > 0. \exists k. (\forall n \geq k. t < -X n) \lor (\forall n \geq k. t < X n)\)
using \(t\) by \(\text{auto}\)
qed

lemma \textit{cauchy-not-vanishes}:
assumes \(X: \text{cauchy X}\)
and \(\text{nz}: \neg \text{vanishes X}\)
shows \(\exists b > 0. \exists k. \forall n \geq k. b < |X n|\)
using \text{cauchy-not-vanishes-cases [OF assms]}\]
by \((\text{elim ex-forward conj-forward asm-rl})\) \(\text{auto}\)

lemma \textit{cauchy-inverse [simp]}:
assumes \(X: \text{cauchy X}\)
and \(\text{nz}: \neg \text{vanishes X}\)
shows \(\text{cauchy} (\lambda n. \text{inverse} (X n))\)
proof \((\text{rule cauchyl})\)
fix \(r :: \text{rat}\)
assume \(0 < r\)
obtain \(b i\) where \(b: 0 < b\) and \(i: \forall n \geq i. b < |X n|\)
using \text{cauchy-not-vanishes [OF X nz]} by \(\text{blast}\)
from \(b i\) have \(\text{nz}: \forall n \geq i. X n \neq 0\) by \(\text{auto}\)
obtain \( s \) where \( s: 0 < s \) and \( r = \text{inverse } b * s * \text{inverse } b \)
proof
  show \( 0 < b * r * b \) by (simp add: \( \langle 0 < r \rangle \) \( b \))
  show \( r = \text{inverse } b * (b * r * b) * \text{inverse } b \)
    using \( b \) by simp
qed

obtain \( j \) where \( j: \forall m \geq j \cdot \forall n \geq j \cdot |X m - X n| < s \)
using cauchyD [OF \( X \) \( s \)] ..
have \( \forall m \geq \max i \cdot \forall n \geq \max j \cdot |\text{inverse } (X m) - \text{inverse } (X n)| < r \)
proof clarsimp
  fix \( m \) \( n \)
  assume \( \ast: i \leq m \cdot j \leq m \cdot i \leq n \cdot j \leq n \)
  have \( |\text{inverse } (X m) - \text{inverse } (X n)| = \text{inverse } |X m| * |X m - X n| * \text{inverse } |X n| \)
    by (simp add: inverse-diff-inverse nz * abs-mult)
  also have \( \ldots < \text{inverse } b * s * \text{inverse } b \)
    by (simp add: mult-strict-mono less-imp-inverse-less \( i \cdot j \cdot b \cdot s \))
  finally show \( |\text{inverse } (X m) - \text{inverse } (X n)| < r \) by (simp only: \( r \))
qed

then show \( \exists k. \forall m \geq k \cdot \forall n \geq k. |\text{inverse } (X m) - \text{inverse } (X n)| < r \) ..
qed

lemma vanishes-diff-inverse:
  assumes \( X: \text{cauchy } X \sim \text{vanishes } X \)
  and \( Y: \text{cauchy } Y \sim \text{vanishes } Y \)
  and \( XY: \text{vanishes } (\lambda n. X n - Y n) \)
shows \( \text{vanishes } (\lambda n. \text{inverse } (X n) - \text{inverse } (Y n)) \)
proof (rule vanishesI)
  fix \( r :: \text{rat} \)
  assume \( r: 0 < r \)
obtain \( a \) \( i \) where \( a: 0 < a \) and \( i: \forall n \geq i. a < |X n| \)
    using cauchy-not-vanishes [OF \( X \)] by blast
obtain \( b \) \( j \) where \( b: 0 < b \) and \( j: \forall n \geq j. b < |Y n| \)
    using cauchy-not-vanishes [OF \( Y \)] by blast
obtain \( s \) where \( s: 0 < s \) and \( \text{inverse } a * s * \text{inverse } b = r \)
proof
  show \( 0 < a * r * b \)
    using \( a \cdot r \cdot b \) by simp
  show \( \text{inverse } a * (a * r * b) * \text{inverse } b = r \)
    using \( a \cdot r \cdot b \) by simp
qed

obtain \( k \) where \( k: \forall n \geq k. |X n - Y n| < s \)
using vanishesD [OF \( XY \cdot \) \( s \)] ..
have \( \forall n \geq \max (\max i \cdot j \cdot k). |\text{inverse } (X n) - \text{inverse } (Y n)| < r \)
proof clarsimp
  fix \( n \)
  assume \( n: i \leq n \cdot j \leq n \cdot k \leq n \)
  with \( i \cdot j \cdot a \cdot b \) have \( X n \neq 0 \) and \( Y n \neq 0 \)
    by auto
then have \(|\text{inverse} (X n) - \text{inverse} (Y n)| = \text{inverse} |X n| * |X n - Y n| * \text{inverse} |Y n|
\)
by (simp add: inverse-diff-inverse abs-mult)
also have \(\ldots < \text{inverse} a * s * \text{inverse} b\)
by (intro mult-strict-mono' less-imp-inverse-less) (simp-all add: a b i j k n)
also note \(\text{inverse} a * s * \text{inverse} b = r\)
finally show \(|\text{inverse} (X n) - \text{inverse} (Y n)| < r\).
qed
then show \(\exists k. \forall n \geq k. |\text{inverse} (X n) - \text{inverse} (Y n)| < r\).
qed

95.4 Equivalence relation on Cauchy sequences

definition realrel :: \((\text{nat} \Rightarrow \text{rat}) \Rightarrow \text{nat} \Rightarrow \text{rat} \Rightarrow \text{bool}\)
where realrel = \((\lambda X Y. \text{cauchy} X \land \text{cauchy} Y \land \text{vanishes} (\lambda n. X n - Y n))\)

lemma realrelI [intro?]: cauchy X \implies cauchy Y \implies \text{vanishes} (\lambda n. X n - Y n) \implies realrel X Y
by (simp add: realrel-def)

lemma realrel-refl: cauchy X \implies realrel X X
by (simp add: realrel-def)

lemma symp-realrel: symp realrel
by (simp add: abs-minus-commute realrel-def symp-def vanishes-def)

lemma transp-realrel: transp realrel
unfolding realrel-def
by (rule transpI) (force simp add: dest: vanishes-add)

lemma part-equivp-realrel: part-equivp realrel
by (blast intro: part-equivpI symp-realrel transp-realrel realrel-refl cauchy-const)

95.5 The field of real numbers

quotient-type real = nat \Rightarrow rat / partial: realrel
morphisms rep-real Real
by (rule part-equivp-realrel)

lemma cr-real-eq: pcr-real = \((\lambda x y. \text{cauchy} x \land \text{Real} x = y)\)
unfolding real_per_cr_eq cr-real-def realrel-def by auto

lemma Real-induct [induct type: real]:
assumes \(\forall X. \text{cauchy} X \implies P (\text{Real} X)\)
shows P X
proof (induct x)
case (1 X)
then have cauchy X by (simp add: realrel-def)
then show P (Real X) by (rule assms)
qed
lemma eq-Real: cauchy X \implies cauchy Y \implies Real X = Real Y \iff vanishes (\lambda n. X n - Y n)
  using real.rel-eq-transfer
  unfolding real.rel-eq transfer cr-real-def rel-fun-def realrel-def by simp

lemma Domainp-pcr-real [transfer-domain-rule]: Domainp pcr-real = cauchy
  by (simp add: real.domain-eq realrel-def)

instantiation real :: field
begin

lift-definition zero-real :: real is \lambda n. 0
  by (simp add: realrel-refl)

lift-definition one-real :: real is \lambda n. 1
  by (simp add: realrel-refl)

lift-definition plus-real :: real \Rightarrow real \Rightarrow real is \lambda X Y n. X n + Y n
  unfolding realrel-def add-diff-add
  by (simp only: cauchy-add vanishes-add simp-thms)

lift-definition uminus-real :: real \Rightarrow real is \lambda X n. - X n
  unfolding realrel-def minus-diff-minus
  by (simp only: cauchy-minus vanishes-minus simp-thms)

lift-definition times-real :: real \Rightarrow real \Rightarrow real is \lambda X Y n. X n \ast Y n
  proof -
    fix f1 f2 f3 f4
    have [cauchy f1; cauchy f4; vanishes (\lambda n. f1 n - f2 n); vanishes (\lambda n. f3 n - f4 n)]
      \implies vanishes (\lambda n. f1 n \ast (f3 n - f4 n) + f4 n \ast (f1 n - f2 n))
      by (simp add: vanishes-add vanishes-mult-bounded cauchy-imp-bounded)
    then show [realrel f1 f2; realrel f3 f4] \implies realrel (\lambda n. f1 n \ast f3 n) (\lambda n. f2 n \ast f4 n)
      by (simp add: mult.commute realrel-def mult-diff-mult)
  qed

lift-definition inverse-real :: real \Rightarrow real
  is \lambda X. if vanishes X then (\lambda n. 0) else (\lambda n. inverse (X n))
  proof -
    fix X Y
    assume realrel X Y
    then have X: cauchy X and Y: cauchy Y and XY: vanishes (\lambda n. X n - Y n)
      by (simp-all add: realrel-def)
    have vanishes X \iff vanishes Y
      proof
        assume vanishes X
        from vanishes-diff [OF this XY] show vanishes Y
by simp
next
  assume \textit{vanishes} Y
  from \textit{vanishes-add} \[ OF \ this \ XY \] \textit{show} \textit{vanishes} X
  by simp
qed
then \textit{show} \ ?thesis \ X \ Y
  by (simp add: \textit{vanishes-diff-inverse} \ X \ Y \ XY \ \textit{realrel-def})
qed

\begin{definition}
x - y = x + - y \ for \ x \ y :: \textit{real}
\end{definition}

\begin{definition}
x div y = x * \textit{inverse} y \ for \ x \ y :: \textit{real}
\end{definition}

\begin{lemma}
\textit{add-Real}:\ \textit{cauchy} X \Longrightarrow \textit{cauchy} Y \Longrightarrow \textit{Real} \ X + \textit{Real} \ Y = \textit{Real} (\lambda n. \ X n + Y n)
using \textit{plus-real.transfer} \ \textit{by} (simp add: \textit{cr-real-eq rel-fun-def})
\end{lemma}

\begin{lemma}
\textit{minus-Real}:\ \textit{cauchy} X \Longrightarrow - \textit{Real} \ X = \textit{Real} (\lambda n. - X n)
using \textit{uminus-real.transfer} \ \textit{by} (simp add: \textit{cr-real-eq rel-fun-def})
\end{lemma}

\begin{lemma}
\textit{diff-Real}:\ \textit{cauchy} X \Longrightarrow \textit{cauchy} Y \Longrightarrow \textit{Real} \ X - \textit{Real} \ Y = \textit{Real} (\lambda n. \ X n - Y n)
by (simp add: \textit{minus-Real add-Real} \textit{minus-real-def})
\end{lemma}

\begin{lemma}
\textit{mult-Real}:\ \textit{cauchy} X \Longrightarrow \textit{cauchy} Y \Longrightarrow \textit{Real} \ X * \textit{Real} \ Y = \textit{Real} (\lambda n. \ X n * Y n)
using \textit{times-real.transfer} \ \textit{by} (simp add: \textit{cr-real-eq rel-fun-def})
\end{lemma}

\begin{lemma}
\textit{inverse-Real}:
\textit{cauchy} X \Longrightarrow \textit{inverse} (\textit{Real} \ X) = (\textit{if} \ \textit{vanishes} \ X \ \textit{then} \ 0 \ \textit{else} \ \textit{Real} (\lambda n. \ \textit{inverse} (X n)))
using \textit{inverse-real.transfer} \textit{zero-real.transfer}
unfolding \textit{cr-real-eq rel-fun-def} \ \textit{by} (simp split: if-split-asm, \textit{metis})
\end{lemma}

instance
proof
  fix a b c :: \textit{real}
  show a + b = b + a
    by transfer (simp add: \textit{ac-simps realrel-def})
  show (a + b) + c = a + (b + c)
    by transfer (simp add: \textit{ac-simps realrel-def})
  show 0 + a = a
    by transfer (simp add: realrel-def)
  show - a + a = 0
    by transfer (simp add: realrel-def)
  show a - b = a + - b
    by (rule \textit{minus-real-def})
  show (a * b) * c = a * (b * c)
by transfer (simp add: ac-simps realrel-def)
show \( a \cdot b = b \cdot a \)
  by transfer (simp add: ac-simps realrel-def)
show \( 1 \cdot a = a \)
  by transfer (simp add: ac-simps realrel-def)
show \( (a + b) \cdot c = a \cdot c + b \cdot c \)
  by transfer (simp add: distrib-right realrel-def)
show \((0::real) \neq (1::real)\)
  by transfer (simp add: realrel-def)
have vanishes \((\lambda n. \inverse (X \cdot n) \cdot X \cdot n - 1)\)
if \( X \): cauchy \( X \) \neg vanishes \( X \)
  for \( X \)
proof (rule vanishesI)
  fix \( r :: rat \)
  assume \( 0 < r \)
  obtain \( b \cdot k \) where \( b > 0 \forall n \geq k. b < |X \cdot n| \)
    using \( X \) cauchy-not-vanishes
  by blast
  then show \( \exists k. \forall n \geq k. \\inverse (X \cdot n) \cdot X \cdot n - 1 < r \)
    using \( \langle 0 < r \rangle \)
    by force
  qed
  then show \( a \neq 0 \Longrightarrow \inverse a \cdot a = 1 \)
    by transfer (simp add: realrel-def)
  show \( a \div b = a \cdot \inverse b \)
    by (rule divide-real-def)
  show \( \inverse (0::real) = 0 \)
    by transfer (simp add: realrel-def)
  qed

end

95.6 Positive reals

lift-definition positive :: real \Rightarrow bool
  is \( \lambda X. \exists r > 0. \exists k. \forall n \geq k. r < X \cdot n \)
proof –
  have \( \exists r > 0. \exists k. \forall n \geq k. r < Y \cdot n \)
    if \( * : \ realrel X \cdot Y \) and \( ** : \exists r > 0. \exists k. \forall n \geq k. r < X \cdot n \) for \( X \cdot Y \)
proof –
  from \( * \) have \( XY : \) vanishes \((\lambda n. X \cdot n - Y \cdot n)\)
    by (simp-all add: realrel-def)
  from \( ** \) obtain \( r \cdot i \) where \( 0 < r \) and \( i : \forall n \geq i. r < X \cdot n \)
    by blast
  obtain \( s \cdot t \) where \( s : 0 < s \) and \( t : 0 < t \) and \( r : r = s + t \)
    using \( \langle 0 < r \rangle \) by (rule obtain-pos-sum)
  obtain \( j \) where \( j : \forall n \geq j. |X \cdot n - Y \cdot n| < s \)
    using \( \) vanishesD \( [OF XY \cdot s] \) ..
  have \( \forall n \geq \max i \cdot j. l < Y \cdot n \)
  proof clarsimp
    fix \( n \)
    assume \( n: i \leq n \cdot j \leq n \)
have $|X n - Y n| < s$ and $r < X n$
  using $i j n$ by simp-all
then show $t < Y n$ by (simp add: $r$)
qed
then show $?thesis$ using $t$ by blast
qed
fix $X Y$ assume realrel $X Y$
then have realrel $X Y$ and realrel $Y X$
  using symp-realrel by (auto simp: symp-def)
then show $?thesis$ $X Y$
  by (safe elim!: 1)
qed

lemma positive-Real: cauchy $X$ $\Longrightarrow$ positive ($\text{Real } X$) $\Longleftrightarrow$ ($\exists r > 0. \ \exists k. \ \forall n \geq k. \ r < X n$)
  using positive.transfer by (simp add: cr-real-eq rel-fun-def)

lemma positive-zero: $\neg$ positive $0$
  by transfer auto

lemma positive-add:
  assumes positive $x$ positive $y$
  shows positive ($x + y$)
proof
  have $\ast$: $[\forall n \geq i. \ a < x n; \ \forall n \geq j. \ b < y n; \ 0 < a; \ 0 < b; \ n \geq \max i j]$
    $\Longrightarrow$ $a + b < x n + y n$ for $x$ $y$ and $a$ $b$::rat and $i$ $j$::nat
  by (simp add: add-strict-mono)
  show $?thesis$
    using assms
  by transfer (blast intro: $\ast$ pos-add-strict)
qed

lemma positive-mult:
  assumes positive $x$ positive $y$
  shows positive ($x * y$)
proof
  have $\ast$: $[\forall n \geq i. \ a < x n; \ \forall n \geq j. \ b < y n; \ 0 < a; \ 0 < b; \ n \geq \max i j]$
    $\Longrightarrow$ $a * b < x n * y n$ for $x$ $y$ and $a$ $b$::rat and $i$ $j$::nat
  by (simp add: mult-strict-mono $\ast$)
  show $?thesis$
    using assms
  by transfer (blast intro: $\ast$ mult-pos-pos)
qed

lemma positive-minus: $\neg$ positive $x$ $\Longrightarrow$ $x \neq 0$ $\Longrightarrow$ positive ($- x$)
apply transfer
apply (simp add: realrel-def)
apply (blast dest: cauchy-not-vanishes-cases)
done

instantiation real :: linordered-field
begin

definition $x < y \iff \text{positive} (y - x)$

definition $x \leq y \iff x < y \vee x = y$ for $x, y :: \text{real}$

definition $|a| = (\text{if } a < 0 \text{ then } -a \text{ else } a)$ for $a :: \text{real}$

definition $\sgn a = (\text{if } a = 0 \text{ then } 0 \text{ else } \text{if } 0 < a \text{ then } 1 \text{ else } -1)$ for $a :: \text{real}$

instance
proof
  fix $a, b, c :: \text{real}$
  show $|a| = (\text{if } a < 0 \text{ then } -a \text{ else } a)$
    by (rule abs-real-def)
  show $a < b \iff a \leq b \land \neg b \leq a$
    a $\leq b \implies b \leq c \implies a \leq c$ a $\leq a$
    a $\leq b \implies b \leq a \implies a = b$
    a $\leq b \implies c + a \leq c + b$
    unfolding less-eq-real-def less-real-def
    by (force simp add: positive-zero dest: positive-add+)
  show $\sgn a = (\text{if } a = 0 \text{ then } 0 \text{ else } \text{if } 0 < a \text{ then } 1 \text{ else } -1)$
    by (rule sgn-real-def)
  show $a \leq b \lor b \leq a$
    by (auto dest!: positive-minus simp: less-eq-real-def less-real-def)
  show $a < b \implies 0 < c \implies c * a < c * b$
    unfolding less-real-def
    by (force simp add: algebra-simps dest: positive-mult)
qed

end

instantiation $\text{real} :: \text{distrib-lattice}$
begin

definition $(\text{inf} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real}) = \text{min}$

definition $(\text{sup} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real}) = \text{max}$

instance
  by standard (auto simp add: inf-real-def sup-real-def max-min-distrib2)

end

lemma $\text{of-nat-Real}: \text{of-nat} x = \text{Real} (\lambda n. \text{of-nat} x)$
  by (induct x) (simp-all add: zero-real-def one-real-def add-Real)

lemma $\text{of-int-Real}: \text{of-int} x = \text{Real} (\lambda n. \text{of-int} x)$
  by (cases x rule: int-diff-cases) (simp add: of-nat-Real diff-Real)
lemma of-rat-Real: of-rat $x = \text{Real} \ (\lambda n \cdot x)$
proof (induct $x$)
  case (Fract $a \ b$)
  then show ?case
  apply (simp add: Fract-of-int-quotient of-rat-divide)
  apply (simp add: of-int-Real divide-inverse inverse-Real mult-Real)
done
qed

instance real :: archimedean-field
proof
show $\exists z. \ x \leq \text{of-int} \ z$ for $x :: \text{real}$
proof (induct $x$)
  case (1 $X$)
  then obtain $b$ where $0 < b$ and $b: \forall n. |X n| < b$
  by (blast dest: cauchy-imp-bounded)
  then have $\text{Real} \ X < \text{of-int} \ (\lceil b \rceil + 1)$
  using 1
  apply (simp add: of-int-Real less-real-def diff-Real positive-Real)
  apply (rule-tac $x=1$ in exI)
  apply (simp add: algebra-simps)
  by (metis abs-ge-self le-less-trans le-of-int-ceiling less-le)
  then show ?case
  using less-eq-real-def by blast
qed
qed

95.7 Completeness

lemma not-positive-Real:
assumes cauchy $X$ shows $\neg \text{positive} \ (\text{Real} \ X) \iff (\forall r > 0. \ \exists k. \ \forall n \geq k. \ X n \leq r)$
(is $\forall \text{lhs} = \forall \text{rhs}$)
unfolding positive-Real [OF assms]
proof (intro iffI allI notI impI)
show $\exists k. \forall n \geq k. \ X n \leq r \ \text{if} \ r: \neg (\exists r > 0. \ \exists k. \ \forall n \geq k. \ r < X n)$ and $0 < r$ for
proof –
  obtain \( s \) \( t \) where \( s > 0 \) \( t > 0 \) \( r = s + t \)
  using \((r > 0)\) obtain-pos-sum by blast
  obtain \( k \) where \( k: \forall m n. [m \geq k; n \geq k] \implies |X m - X n| < t \)
  using cauchyD \((\text{OF assms})\) \((t > 0)\) by blast
  obtain \( n \) where \( n \geq k \) \( \forall X n \leq s \)
  by (meson \( r < s \) not-less)
  then have \( X l \leq r \) if \( l \geq n \) for \( l \)
  using \( k \) \((\text{OF } \forall n \geq k)\), of \( l \) that \( r = s + t \) by linarith
  then show \( \text{thesis} \)
  by blast
qed

lemma \( \text{le-Real} \):
  assumes \( \text{cauchy X cauchy Y} \)
  shows \( \text{Real X} \leq \text{Real Y} = (\forall r > 0. \exists k. \forall n \geq k. X n \leq Y n + r) \)
  unfolding not-less \([\text{symmetric, where 'a=real}]\) \( \text{less-real-def} \)
  apply \((\text{simp add: diff-Real not-positive-Real assms})\)
  apply \((\text{simp add: diff-le-eq ac-simps})\)
  done

lemma \( \text{le-RealI} \):
  assumes \( Y: \text{cauchy Y} \)
  shows \( \forall n. x \leq \text{of-rat} (Y n) \implies x \leq \text{Real Y} \)
  proof \((\text{induct } x)\)
    fix \( X \)
    assume \( X: \text{cauchy X and } \forall n. \text{Real X} \leq \text{of-rat} (Y n) \)
    then have \( \text{le}: \forall m r. 0 < r \implies \exists k. \forall n \geq k. X n \leq Y m + r \)
    by \((\text{simp add: of-rat-Real le-Real})\)
    then have \( \exists k. \forall n \geq k. X n \leq Y n + r \) if \( 0 < r \) for \( r :: \text{rat} \)
    proof –
      from that obtain \( s \) \( t \) where \( s > 0 \) and \( t > 0 \) and \( r = s + t \)
      by \((\text{rule obtain-pos-sum})\)
      obtain \( i \) where \( i: \forall m \geq i. \forall n \geq i. |Y m - Y n| < s \)
      using cauchyD \((\text{OF Y s})\) ..
      obtain \( j \) where \( j: \forall n \geq j. X n \leq Y i + t \)
      using \( \text{le} [\text{OF } l] \) ..
      have \( \forall n \geq \max i j. X n \leq Y n + r \)
      proof clarsimp
        fix \( n \)
        assume \( n: i \leq n j \leq n \)
        have \( X n \leq Y i + t \)
        using \( n j \) by simp
        moreover have \( |Y i - Y n| < s \)
        using \( n i \) by simp
        ultimately show \( X n \leq Y n + r \)
        unfolding \( r \) by simp
      qed
qed
then show thesis ..
qed
then show Real X ≤ Real Y
  by (simp add: of-rat-Real le-Real X Y)
qed

lemma Real-leI:
  assumes X: cauchy X
  assumes le: ∀ n. of-rat (X n) ≤ y
  shows Real X ≤ y
proof
  have −y ≤ −Real X
    by (simp add: minus-Real X le-RealI of-rat-minus le)
  then show thesis by simp
qed

lemma less-RealD:
  assumes cauchy Y
  shows x < Real Y ⇒ ∃ n. x < of-rat (Y n)
apply (erule contrapos-pp)
apply (simp add: not-less)
apply (erule Real-leI [OF assms])
done

lemma of-nat-less-two-power [simp]: of-nat n < (2::_:linordered-idom) ^ n
apply (induct n)
apply simp
apply (metis add-le-less-mono mult-2 of-nat-Suc one-le-numeral one-le-power power-Suc)
done

lemma complete-real:
fixes S :: real set
assumes ∃ x. x ∈ S and ∃ z. ∀ x∈S. x ≤ z
shows ∃ y. (∀ x∈S. x ≤ y) ∧ (∀ z. (∀ x∈S. x ≤ z) → y ≤ z)
proof
  obtain x where x: x ∈ S using assms(1) ..
obtain z where z: ∀ x∈S. x ≤ z using assms(2) ..

define P where P x ↔ (∀ y∈S. y ≤ of-rat x) for x
obtain a where a: ¬ P a
proof
  have of-int |x − 1| ≤ x − 1 by (rule of-int-floor-le)
  also have x − 1 < x by simp
  finally have of-int |x − 1| < x .
  then have ¬ x ≤ of-int |x − 1| by (simp only: not-le)
  then show ¬ P (of-int |x − 1|)
    unfolding P-def of-rat-of-int-eq using x by blast
qed
obtain \( b \) where \( b \colon P \, b \)
proof
  show \( P \, (\text{of-int } \lceil z \rceil) \)
  unfolding \( P\)-def of-rat-of-int-eq
  proof
    fix \( y \) assume \( y \in S \)
    then have \( y \leq z \) using \( z \) by simp
    also have \( z \leq \text{of-int } \lceil z \rceil \) by (rule le-of-int-ceiling)
    finally show \( y \leq \text{of-int } \lceil z \rceil \).
qed

define \( \text{avg} \) where \( \text{avg} \, x \, y = \frac{x}{2} + \frac{y}{2} \)
for \( x \, y :: \text{rat} \)

define \( \text{bisect} \) where \( \text{bisect} = \lambda (x, \, y). \) if \( P \, (\text{avg} \, x \, y) \) then \( (x, \, \text{avg} \, x \, y) \) else \( (\text{avg} \, x \, y, \, y) \)

define \( A \) where \( A \, n = \text{fst} \, ((\text{bisect} \, ^{\sim} \, n) \, (a, \, b)) \) for \( n \)

define \( B \) where \( B \, n = \text{snd} \, ((\text{bisect} \, ^{\sim} \, n) \, (a, \, b)) \) for \( n \)

define \( C \) where \( C \, n = \text{avg} \, (A \, n, \, B \, n) \) for \( n \)

have \( A\text{-}0 \) [simp]: \( A \, 0 = a \) unfolding \( A\)-def by simp

have \( B\text{-}0 \) [simp]: \( B \, 0 = b \) unfolding \( B\)-def by simp

have \( A\text{-}Suc \) [simp]: \( \forall n. \, A \, (\text{Suc} \, n) = (\text{if } P \, (C \, n) \, \text{then } A \, n \, \text{else } C \, n) \)

unfolding \( A\)-def \( B\)-def \( C\)-def \( \text{bisect-def} \) \( \text{split-def} \) by simp

have \( B\text{-}Suc \) [simp]: \( \forall n. \, B \, (\text{Suc} \, n) = (\text{if } P \, (C \, n) \, \text{then } C \, n \, \text{else } B \, n) \)

unfolding \( A\)-def \( B\)-def \( C\)-def \( \text{bisect-def} \) \( \text{split-def} \) by simp

have width: \( B \, n - A \, n = (b - a) / 2^{\sim} n \) for \( n \)
proof (induct \( n \))
  case (Suc \( n \))
  then show \( ?case \) by (simp add: \( C\)-def \( eq\)-divide-eq \( avg\)-def \( algebra\)-simps)
qed simp

have twoos: \( \exists n. \, y / 2^{\sim} n < r \) if \( 0 < r \) for \( y \, r :: \text{rat} \)
proof –
  obtain \( n \) where \( y / r < \text{rat-of-nat} \, n \)
  using \( 0 < r \) reals-Archimedean2 by blast
  then have \( \exists n. \, y < r * 2^{\sim} n \)
  by (metis divide-less-eq less-trans mult.commute of-nat-less-two-power that)
  then show \( ?thesis \) by (simp add: field-split-simps)
qed

have \( PA \colon \neg P \, (A \, n) \) for \( n \)
  by (induct \( n \)) (simp-all add: \( a \))

have \( PB \colon P \, (B \, n) \) for \( n \)
  by (induct \( n \)) (simp-all add: \( b \))

have \( a \, b \) unfolding \( P\)-def
  by (meson leI less-le-trans of-rat-less)
have \( AB \colon A \, n < B \, n \) for \( n \)
by (induct n) (simp-all add: ab C-def avg-def)

have \( A \ i \leq A \ j \land B \ j \leq B \ i \) if \( i < j \) for \( i \ j \)
using that

proof (induction rule: less-Suc-induct)

case (1 i)
then show \(?case\)
apply (clarsimp simp add: ab C-def avg-def add-divide-distrib [symmetric])
done

qed simp

then have \( A \text{-mono}: A \ i \leq A \ j \) and \( B \text{-mono}: B \ j \leq B \ i \) if \( i \leq j \) for \( i \ j \)
by (metis eq-refl le-neq-implies-less that)

have cauchy-lemma: cauchy \( X \) if \( \forall n \ i. \ i \geq n \ \Rightarrow \ A \ n \leq X \ i \land X \ i \leq B \ n \) for \( X \)

proof (rule cauchyI)
  fix \( r :: \text{rat} \)
  assume \( 0 < r \)
  then obtain \( k \) where \( k \): \( \frac{(b - a)}{2^k} \ < \ r \)
  using twos by blast
  have \( |X \ m - X \ n| < r \) if \( m \geq k \ n \geq k \) for \( m \ n \)
  proof -
    have \( |X \ m - X \ n| \leq B \ k - A \ k \)
    by (simp add: * abs-rat-def diff-mono that)
    also have \( \ldots < r \)
    by (simp add: k width)
    finally show \(?thesis\).
  qed
  then show \( \exists k. \forall m \geq k. \forall n \geq k. |X \ m - X \ n| < r \)
  by blast

  qed

  have cauchy A
  by (rule cauchy-lemma) (meson AB A-mono B-mono dual-order.strict-implies-order less-le-trans)

  have cauchy B
  by (rule cauchy-lemma) (meson AB A-mono B-mono dual-order.strict-implies-order le-less-trans)

  have \( \forall x \in S. \ x \leq \text{Real} \ B \)

  proof
    fix \( x \)
    assume \( x \in S \)
    then show \( x \leq \text{Real} \ B \)
    using PB [unfolded P-def] (cauchy B)
    by (simp add: le-RealI)

  qed

moreover have \( \forall z. (\forall x \in S. \ x \leq z) \ \rightarrow \ \text{Real} \ A \leq z \)
by (meson PA Real-leI P-def \( \langle\text{cauchy A}\rangle\) le-cases order.trans)
moreover have \( \forall n. (b - a) / 2^n \)

proof (rule vanishesI)
  fix \( r :: \text{rat} \)
assume $0 < r$
then obtain $k$ where $k: |b - a| / 2 ^ k < r$
using twos by blast
have $\forall n \geq k. \| (b - a) / 2 ^ n \| < r$
proof clarify
fix $n$
assume $n: k \leq n$
have $| (b - a) / 2 ^ n | = | b - a | / 2 ^ n$
by simp
also have $\ldots \leq | b - a | / 2 ^ k$
using $n$ by (simp add: divide-left-mono)
also note $k$
finally show $| (b - a) / 2 ^ n | < r$
qed
then show $\exists k. \forall n \geq k. \| (b - a) / 2 ^ n \| < r$ ..
qed
then have $\text{Real B} = \text{Real A}$
by (simp add: eq-Real \langle cauchy A \rangle \langle cauchy B \rangle width)
ultimately show $\exists y. (\forall x \in S. x \leq y) \land (\forall z. (\forall x \in S. x \leq z) \rightarrow y \leq z)$
by force
qed

instantiation real :: linear-continuum
begin

95.8 Supremum of a set of reals

definition $\text{Sup X} = (\text{LEAST} z::\text{real}. \forall x \in X. x \leq z)$
definition $\text{Inf X} = - \text{Sup (uminus } ^ \langle X \rangle)$ for $X :: \text{real set}$
instantiation
proof
show $\text{Sup-upper}: x \leq \text{Sup X}$
if $x \in X \text{ bdd-above} X$
for $x :: \text{real and} X :: \text{real set}$
proof
from that obtain $s$ where $s: \forall y \in X. y \leq s \land \forall z. \forall y \in X. y \leq z \rightarrow s \leq z$
using complete-real \langle of $X$ \rangle unfolding bdd-above-def by blast
then show $\text{thesis}$
unfolding $\text{Sup-real-def}$ by (rule LeastI2-order) (auto simp: that)
qed
show $\text{Sup-least}: \text{Sup X} \leq z$
if $X \neq \{\}$ and $z: \sqcap x. x \in X \rightarrow x \leq z$
for $z :: \text{real and} X :: \text{real set}$
proof
from that obtain $s$ where $s: \forall y \in X. y \leq s \land \forall z. \forall y \in X. y \leq z \rightarrow s \leq z$
using complete-real \langle of $X$ \rangle by blast
then have $\text{Sup X} = s$
unfolding $\text{Sup-real-def}$ by (best intro: Least-equality)
also from \( s \) \( z \) have \( \ldots \leq z \) 
by blast
finally show \(?thesis\).
qed
show \( \text{Inf } X \leq x \) if \( x \in X \) \text{bdd-below } X
for \( x :: \) real \text{ and } \( X :: \) real set
using \( \text{Sup-upper } \{ \text{of } \text{uminus} ' X \} \) by (auto simp: \text{Inf-real-def that})
show \( z \leq \text{Inf } X \) if \( X \neq \{ \} \) \text{ and } \( \forall x. x \in X \implies z \leq x \)
for \( z :: \) real \text{ and } \( X :: \) real set
using \( \text{Sup-least } \{ \text{of } \text{uminus} ' X - z \} \) by (force simp: \text{Inf-real-def that})
show \( \exists a \ b :: \) real. \( a \neq b \)
using \text{zero-neq-one} by blast
qed
end

95.9 Hiding implementation details
hide-const \( \text{(open)} \) \text{ vanishes cauchy positive Real}
declare \( \text{Real-induct } \{ \text{induct del} \} \)
declare \( \text{Abs-real-induct } \{ \text{induct del} \} \)
declare \( \text{Abs-real-cases } \{ \text{cases del} \} \)
lifting-update real.lifting
lifting-forget real.lifting

95.10 More Lemmas
BH: These lemmas should not be necessary; they should be covered by existing simp rules and simplification procedures.
lemma \( \text{real-mult-less-iff1 } \{ \text{simp} \} : 0 < z \implies x * z < y * z \iff x < y \)
for \( x \ y \ z :: \) real
by simp
lemma \( \text{real-mult-le-cancel-iff1 } \{ \text{simp} \} : 0 < z \implies x * z \leq y * z \iff x \leq y \)
for \( x \ y \ z :: \) real
by simp
lemma \( \text{real-mult-le-cancel-iff2 } \{ \text{simp} \} : 0 < z \implies z * x \leq z * y \iff x \leq y \)
for \( x \ y \ z :: \) real
by simp

95.11 Embedding numbers into the Reals
abbreviation \( \text{real-of-nat } :: \) nat \( \Rightarrow \) real
where \( \text{real-of-nat } \equiv \text{ of-nat} \)
abbreviation \( \text{real } :: \) nat \( \Rightarrow \) real
where $\text{real} \equiv \text{of-nat}$

abbreviation $\text{real-of-int} :: \text{int} \Rightarrow \text{real}$
  where $\text{real-of-int} \equiv \text{of-int}$

abbreviation $\text{real-of-rat} :: \text{rat} \Rightarrow \text{real}$
  where $\text{real-of-rat} \equiv \text{of-rat}$

declare $\text{coercion-enabled}$

declare $\text{coercion of-nat} :: \text{nat} \Rightarrow \text{int}$

declare $\text{coercion of-nat} :: \text{nat} \Rightarrow \text{real}$

declare $\text{coercion of-int} :: \text{int} \Rightarrow \text{real}$

declare $\text{coercion-map $\lambda$ f g h x. g (h (f x))}$

declare $\text{coercion-map $\lambda$ f g (x,y). (f x, g y)}$

declare $\text{of-int-eq-0-iff}$ [algebra, presburger]

declare $\text{of-int-eq-1-iff}$ [algebra, presburger]

declare $\text{of-int-eq-iff}$ [algebra, presburger]

declare $\text{of-int-less-0-iff}$ [algebra, presburger]

declare $\text{of-int-less-1-iff}$ [algebra, presburger]

declare $\text{of-int-less-iff}$ [algebra, presburger]

declare $\text{of-int-le-0-iff}$ [algebra, presburger]

declare $\text{of-int-le-1-iff}$ [algebra, presburger]

declare $\text{of-int-le-iff}$ [algebra, presburger]

declare $\text{of-int-0-less-iff}$ [algebra, presburger]

declare $\text{of-int-0-le-iff}$ [algebra, presburger]

declare $\text{of-int-1-less-iff}$ [algebra, presburger]

declare $\text{of-int-1-le-iff}$ [algebra, presburger]

lemma $\text{int-less-real-le}: n < m \iff \text{real-of-int } n + 1 \leq \text{real-of-int } m$

proof -
  have $(0 :: \text{real}) \leq 1$
    by (metis less-eq-real-def zero-less-one)
  then show ?thesis
    by (metis floor-of-int less-floor-iff)
qed

lemma $\text{int-le-real-less}: n \leq m \iff \text{real-of-int } n < \text{real-of-int } m + 1$
  by (meson int-less-real-le not-le)

lemma $\text{real-of-int-div-aux}$:
  $(\text{real-of-int } x) \div (\text{real-of-int } d) =$
  \text{real-of-int } (x \div d) + (\text{real-of-int } (x \mod d)) \div (\text{real-of-int } d)$

proof -
have \(x = (x \div d) \ast d + x \mod d\)
  by auto
then have \(\text{real-of-int } x = \text{real-of-int } (x \div d) \ast \text{real-of-int } d + \text{real-of-int } (x \mod d)\)
  by (metis of-int-add of-int-mult)
then have \(\text{real-of-int } x \div \text{real-of-int } d = \ldots \div \text{real-of-int } d\)
  by simp
then show \(?\text{thesis}\)
  by (auto simp add: add-divide-distrib algebra-simps)
qed

lemma real-of-int-div:
\(d \mid n \Longrightarrow \text{real-of-int } (n \div d) = \text{real-of-int } n \div \text{real-of-int } d\) for \(d \cdot n \vdash \text{int}\)
by (simp add: real-of-int-div-aux)

lemma real-of-int-div2: \(0 \leq \text{real-of-int } n \div \text{real-of-int } x = \text{real-of-int } (n \div x)\)
proof (cases \(x = 0\))
case False
then show \(?\text{thesis}\)
  by (metis diff-ge-0-iff-ge floor-divide-of-int-eq of-int-floor-le)
qed simp

lemma real-of-int-div3: \(\text{real-of-int } n \div \text{real-of-int } x - \text{real-of-int } (n \div x) \leq 1\)
apply (simp add: algebra-simps)
by (metis add.commute floor-correct floor-divide-of-int-eq less-eq-real-def of-int-1 of-int-add)

lemma real-of-int-div4: \(\text{real-of-int } (n \div x) \leq \text{real-of-int } n \div \text{real-of-int } x\)
using real-of-int-div2 \([of n x]\) by simp

95.12 Embedding the Naturals into the Reals

lemma real-of-card: \(\text{real } (\text{card } A) = \text{sum } (\lambda x. 1) A\)
  by simp

lemma nat-less-real-le: \(n < m \iff \text{real } n + 1 \leq \text{real } m\)
  by (meson discrete of-nat-1 of-nat-add of-nat-le-iff)

lemma nat-le-real-less: \(n \leq m \iff \text{real } n < \text{real } m + 1\)
  for \(m \cdot n \vdash \text{nat}\)
  by (meson nat-less-real-le not-le)

lemma real-of-nat-div-aux: \(\text{real } x \div \text{real } d = \text{real } (x \div d) + \text{real } (x \mod d) \div \text{real } d\)
proof -
  have \(x = (x \div d) \ast d + x \mod d\)
    by auto
  then have \(\text{real } x = \text{real } (x \div d) \ast \text{real } d + \text{real } (x \mod d)\)
    by (metis of-nat-add of-nat-mult)
then have real x / real d = ... / real d
  by simp
then show ?thesis
  by (auto simp add: add-divide-distrib algebra-simps)
qed

lemma real-of-nat-div: d dvd n ⇒ real(n div d) = real n / real d
  by (subst real-of-nat-div-aux) (auto simp add: dvd-eq-mod-eq-0 [symmetric])

lemma real-of-nat-div2: 0 ≤ real n / real x = real (n div x) for n x :: nat
  apply (simp add: algebra-simps)
  by (metis floor-divide-of-nat-eq of-int-floor-le of-int-of-nat-eq)

lemma real-of-nat-div3: real n / real x − real (n div x) ≤ 1 for n x :: nat
  proof (cases x = 0)
    case False
    then show ?thesis
      by (metis of-int-of-nat-eq real-of-int-div3 zdiv-int)
  qed auto

lemma real-of-nat-div4: real (n div x) ≤ real n / real x for n x :: nat
  using real-of-nat-div2 [of n x]
  by simp

95.13 The Archimedean Property of the Reals

lemma real-arch-inverse: 0 < e ⇔ (∃ n::nat. n ≠ 0 ∧ 0 < inverse (real n) ∧
  inverse (real n) < e)
  using reals-Archimedean[of e] less-trans[of 0 1 / real e for n::nat]
  by (auto simp add: field-simps cong: conj-cong simp del: of-nat-Suc)

lemma reals-Archimedean3: 0 < x ⇒ ∀ y. ∃ n. y < real n * x
  by (auto intro: ex-less-of-nat-mult)

lemma real-archimedian-rdiv-eq-0:
  assumes x0: x ≥ 0
  and c: c ≥ 0
  and xc: ∀ m::nat. m > 0 ⇒ real m * x ≤ c
  shows x = 0
  by (metis reals-Archimedean3 dual-order.order-iff-strict le0 le-less-trans not-le x0 xc)

95.14 Rationals

lemma Rats-abs-iff[simp]:
  |(x::real)| ∈ Q ⇔ x ∈ Q
by(simp add: abs-real-def split: if-splits)

lemma Rats-eq-int-div-int: Q = {real-of-int i / real-of-int j | i j ≠ 0} (is - = ?S)
proof
show $Q \subseteq \mathcal{S}$
proof
fix $x :: \text{real}$
assume $x \in Q$
then obtain $r$ where $x = \text{of-rat } r$
unfolding Rats-def ..
have $\text{of-rat } r \in \mathcal{S}$
by (cases $r$) (auto simp add: of-rat-rat)
then show $x \in \mathcal{S}$
using $\langle x = \text{of-rat } r \rangle$ by simp
qed
next
show $\mathcal{S} \subseteq Q$
proof (auto simp: Rats-def)
fix $i \ j :: \text{int}$
assume $j \neq 0$
then have $\text{real-of-int } i \ / \ \text{real-of-int } j = \text{of-rat } (\text{Fract } i \ j)$
by (simp add: of-rat-rat)
then show $\text{real-of-int } i \ / \ \text{real-of-int } j \in \text{range of-rat}$
by blast
qed
qed

lemma Rats-eq-int-div-nat: $Q = \{ \text{real-of-int } i \ / \ \text{real } n \mid i \ n. \ n \neq 0 \}$
proof (auto simp: Rats-eq-int-div-int)
fix $i \ j :: \text{int}$
assume $j \neq 0$
show $\exists (i' :: \text{int}) (n :: \text{nat}). \ \text{real-of-int } i \ / \ \text{real-of-int } j = \text{real-of-int } i' \ / \ \text{real } n \land 0 < n$
proof (cases $j > 0$)
  case True
  then have $\text{real-of-int } i \ / \ \text{real-of-int } j = \text{real-of-int } i \ / \ \text{real } (\text{nat } j) \land 0 < \text{nat } j$
  by simp
  then show $?thesis$ by blast
next
  case False
  with $j \neq 0$
  have $\text{real-of-int } i \ / \ \text{real-of-int } j = \text{real-of-int } (\neg \ i) \ / \ \text{real } (\text{nat } (\neg \ j)) \land 0 < \text{nat } (\neg \ j)$
  by simp
  then show $?thesis$ by blast
qed

next
fix $i :: \text{int}$ and $n :: \text{nat}$
assume $0 < n$
then have $\text{real-of-int } i \ / \ \text{real } n = \text{real-of-int } i \ / \ \text{real-of-int}(\text{int } n) \land \text{int } n \neq 0$
by simp
then show $\exists i' \ j. \ \text{real-of-int } i \ / \ \text{real } n = \text{real-of-int } i' \ / \ \text{real-of-int } j \land j \neq 0$
by blast
lemma Rats-abs-nat-div-natE:
assumes x ∈ Q
obtains m n :: nat where n ≠ 0 and |x| = real m / real n and coprime m n
proof -
  from ⟨x ∈ Q⟩ obtain i :: int and n :: nat where n ≠ 0 and x = real-of-int i / real n
  by (auto simp add: Rats-eq-int-div-nat)
  then have |x| = real (nat |i|) / real n by simp
  then obtain m :: nat where x-rat: |x| = real m / real n by blast
  let ?gcd = gcd m n
  from ⟨n ≠ 0⟩ have gcd: ?gcd ≠ 0 by simp
  let ?k = m div ?gcd
  let ?l = n div ?gcd
  let ?gcd' = gcd ?k ?l
  have ?gcd dvd m ..
    by (rule dvd-mult-div-cancel)
  have ?gcd dvd n ..
    by (rule dvd-mult-div-cancel)
  from ⟨n ≠ 0⟩ and gcd-l have ?gcd * ?l ≠ 0 by simp
  then have ?l ≠ 0 by (blast dest!: mult-not-zero)
  moreover have |x| = real ?k / real ?l
  proof -
    from gcd have real ?k / real ?l = real (?gcd * ?k) / real (?gcd * ?l)
      by (simp add: real-of-nat-div)
    also from gcd-k and gcd-l have ... = real m / real n by simp
    also from x-rat have ... = |x| ..
    finally show ?thesis ..
  qed
  moreover have ?gcd' = 1
  proof -
    have ?gcd * ?gcd' = gcd (?gcd * ?k) (?gcd * ?l)
      by (rule gcd-mult-distrib-nat)
    with gcd-k gcd-l have ?gcd * ?gcd' = ?gcd by simp
    with gcd show ?thesis by auto
  qed
  then have coprime ?k ?l
    by (simp only: coprime_iff_gcd_eq_1)
  ultimately show ?thesis ..
  qed
95.15 Density of the Rational Reals in the Reals

This density proof is due to Stefan Richter and was ported by TN. The original source is *Real Analysis* by H.L. Royden. It employs the Archimedean property of the reals.

**Lemma Rats-dense-in-real:**

- **Fixes** \( x :: \text{real} \)
- **Assumes** \( x < y \)
- **Shows** \( \exists r \in \mathbb{Q}. \ x < r \land r < y \)

**Proof**

- From \( x < y \) have \( 0 < y - x \) by simp
- With \( \text{reals-Archimedean} \) obtain \( q :: \text{nat} \) where \( q: \text{inverse} (\text{real } q) < y - x \) and \( 0 < q \)
  
  - by blast

  - Define \( p \) where \( p = \lfloor y \times \text{real } q \rfloor - 1 \)
  - Define \( r \) where \( r = \text{of-int } p / \text{real } q \)
  - From \( q \) have \( x < y - \text{inverse} (\text{real } q) \)
    
    - By simp
  - Also from \( 0 < q \) have \( y - \text{inverse} (\text{real } q) \leq r \)
    
    - By (simp add: r-def p-def le-divide-eq left-diff-distrib)
  - Finally have \( x < r \).
  - Moreover from \( 0 < q \) have \( r < y \)
    
    - By (simp add: r-def p-def divide-less-eq diff-less-eq less-ceiling-iff [symmetric])
  - Moreover have \( r \in \mathbb{Q} \)
    
    - By (simp add: r-def)
  - Ultimately show \( \text{thesis} \) by blast

**QED**

**Lemma of-rat-dense:**

- **Fixes** \( x y :: \text{real} \)
- **Assumes** \( x < y \)
- **Shows** \( \exists q :: \text{rat}. \ x < \text{of-rat } q \land \text{of-rat } q < y \)

  - Using \( \text{Rats-dense-in-real [OF } (x < y) \] \)
  - By (auto elim: Rats-cases)

95.16 Numerals and Arithmetic

**Declaration**

\[ K \langle \text{Lin-Arith.add-inj-const } (\text{const-name} (\text{of-nat}), \text{typ} (\text{nat } \Rightarrow \text{real})) \rangle \]

\[ > \text{Lin-Arith.add-inj-const } (\text{const-name} (\text{of-int}), \text{typ} (\text{int } \Rightarrow \text{real})) \]

95.17 Simprules combining \( x + y \) and \( 0 \)

**Lemma real-add-minus-iff** [simp]: \( x + - a = 0 \iff x = a \)

- For \( x a :: \text{real} \)
  
  - By arith

**Lemma real-add-less-0-iff**: \( x + y < 0 \iff y < - x \)
for \( x \) \( y :: \text{real} \)
by auto

lemma \text{real-0-less-add-iff}: \( 0 < x + y \iff -x < y \)
for \( x \) \( y :: \text{real} \)
by auto

lemma \text{real-add-le-0-iff}: \( x + y \leq 0 \iff y \leq -x \)
for \( x \) \( y :: \text{real} \)
by auto

lemma \text{real-0-le-add-iff}: \( 0 \leq x + y \iff -x \leq y \)
for \( x \) \( y :: \text{real} \)
by auto

95.18 Lemmas about powers

lemma \text{two-realpow-ge-one}: \( (1 :: \text{real}) \leq 2 ^ n \)
by simp

declare \text{sum-squares-eq-zero-iff} simp \text{sum-power2-eq-zero-iff} simp

lemma \text{real-minus-mult-self-le} simp: \( - (u * u) \leq x * x \)
for \( u \) \( x :: \text{real} \)
by (rule order-trans [where \( y = 0 \)]) auto

lemma \text{realpow-square-minus-le} simp: \( - u ^ 2 \leq x ^ 2 \)
for \( u \) \( x :: \text{real} \)
by (auto simp add: power2-eq-square)

95.19 Density of the Reals

lemma \text{field-lbound-gt-zero}: \( 0 < d1 \Rightarrow 0 < d2 \Rightarrow \exists e. 0 < e \wedge e < d1 \wedge e < d2 \)
for \( d1 \) \( d2 :: 'a::linordered-field \)
by (rule exI [where \( x = \min d1 \, d2 / 2 \)]) (simp add: min-def)

lemma \text{field-less-half-sum}: \( x < y \Rightarrow x < (x + y) / 2 \)
for \( x \) \( y :: 'a::linordered-field \)
by auto

lemma \text{field-sum-of-halves}: \( x / 2 + x / 2 = x \)
for \( x :: 'a::linordered-field \)
by simp

95.20 Archimedean properties and useful consequences

Bernoulli’s inequality
proposition Bernoulli-inequality:
fixes \( x \) :: real
assumes \(-1 \leq x\)
shows \( 1 + n \cdot x \leq (1 + x)^n \)
proof (induct \( n \))
case 0
then show \(?case\) by simp
next
case (Suc \( n \))
have \( 1 + \text{Suc } n \cdot x \leq 1 + (\text{Suc } n) \cdot x + n \cdot x^2 \)
  by (simp add: algebra-simps)
also have \( \ldots = (1 + x) \cdot (1 + n \cdot x) \)
  by (auto simp: power2-eq-square algebra-simps)
also have \( \ldots \leq (1 + x)^{\text{Suc } n} \)
  using Suc.hyps assms mult-left-mono
by fastforce
finally show \(?case\).
qed

corollary Bernoulli-inequality-even:
fixes \( x \) :: real
assumes even \( n \)
shows \( 1 + n \cdot x \leq (1 + x)^n \)
proof (cases \(-1 \leq x \lor n=0\))
case True
then show \(?thesis\)
  by (auto simp: Bernoulli-inequality)
next
case False
then have real \( n \geq 1 \)
  by simp
with False have \( n \cdot x \leq -1 \)
  by (metis linear minus-zero mult.commute mult.left-neutral mult.left-mono-neg
     neg-le-iff-le order-trans zero-le-one)
then have \( 1 + n \cdot x \leq 0 \)
  by auto
also have \( \ldots \leq (1 + x)^n \)
  using assms
  using zero-le-even-power
  by blast
finally show \(?thesis\).
qed

corollary real-arch-pow:
fixes \( x \) :: real
assumes \( x: 1 < x \)
shows \( \exists n. \ y < x^n \)
proof
from \( x \) have \( x^0: x - 1 > 0 \)
  by arith
from reals-Archimedean3[OF \( x^0 \), rule-format, of \( y \)....]
obtain \( n : \text{nat} \) where \( n : y < \text{real} \ n \ast (x - 1) \) by metis
from \( x0 \) have \( x00 : x - 1 \geq -1 \) by arith
from Bernoulli-inequality[OF \( x00 \), of \( n \)] have \( y < x \cdot n \) by auto
then show \( \text{thesis} \) by metis
qed

corollary real-arch-pow-inv:
fixes \( x \) \( y :: \text{real} \)
assumes \( y : y > 0 \)
and \( x1 : x < 1 \)
shows \( \exists n. \ x^n < y \)
proof (cases \( x > 0 \))
  case True
  with \( x1 \) have \( ix : 1 < 1/x \) by (simp add: field-simps)
  from real-arch-pow[OF \( ix \), of \( 1/y \)] obtain \( n \) where \( n : 1/y < (1/x)^n \) by blast
  then show \( \text{thesis} \) using \( y \langle x > 0 \rangle \)
    by (auto simp add: field-simps)
next
  case False
  with \( y \) \( x1 \) show \( \text{thesis} \)
    by (metis less-le-trans not-less power-one-right)
qed

lemma forall-pos-mono:
(\( \forall d e :: \text{real}. \ d < e \mapsto P \ d \mapsto P \ e \) \mapsto
  (\( \forall n :: \text{nat}. \ n \neq 0 \mapsto P \ (\text{inverse} \ (\text{real} \ n)) \) \mapsto (\( \forall e. \ 0 < e \mapsto P \ e \)
by (metis real-arch-inverse)

lemma forall-pos-mono-1:
(\( \forall d e :: \text{real}. \ d < e \mapsto P \ d \mapsto P \ e \) \mapsto
  (\( \forall n. \ P \ (\text{inverse} \ (\text{real} \ (\text{Suc} \ n))) \) \mapsto 0 < e \mapsto P \ e
apply (rule forall-pos-mono)
apply auto
apply (metis Suc-pred of-nat-Suc)
done

95.21 Floor and Ceiling Functions from the Reals to the Integers

lemma real-of-nat-less-numeral-iff [simp]: \( \text{real} \ n < \text{numeral} \ w \longleftrightarrow n < \text{numeral} \ w \)
  for \( n :: \text{nat} \)
  by (metis of-nat-less-iff of-nat-numeral)

lemma numeral-less-real-of-nat-iff [simp]: \( \text{numeral} \ w < \text{real} \ n \longleftrightarrow \text{numeral} \ w < n \)
  for \( n :: \text{nat} \)
by (metis of-nat-less-iff of-nat-numeral)

lemma numeral-le-real-of-nat-iff [simp]: numeral n ≤ real m if-then numeral n ≤ m
  for m :: nat
  by (metis not-le real-of-nat-less-numeral-iff)

lemma of-int-floor-cancel [simp]: of-int ⌊x⌋ = x if-then (∃ n::int. x = of-int n)
  by (metis floor-of-int)

lemma floor-eq: real-of-int n < x if-then x < real-of-int n + 1 if-then ⌊x⌋ = n
  by linarith

lemma floor-eq2: real-of-int n ≤ x if-then x < real-of-int n + 1 if-then ⌊x⌋ = n
  by (fact floor-unique)

lemma floor-eq3: real n < x if-then x < real (Suc n) if-then ⌊x⌋ = n
  by linarith

lemma floor-eq4: real n ≤ x if-then x < real (Suc n) if-then ⌊x⌋ = n
  by linarith

lemma real-of-int-floor-ge-diff-one [simp]: r − 1 ≤ real-of-int ⌊r⌋
  by linarith

lemma real-of-int-floor-gt-diff-one [simp]: r − 1 < real-of-int ⌊r⌋
  by linarith

lemma real-of-int-floor-add-one-ge [simp]: r ≤ real-of-int ⌊r⌋ + 1
  by linarith

lemma real-of-int-floor-add-one-gt [simp]: r < real-of-int ⌊r⌋ + 1
  by linarith

lemma floor-divide-real-eq-div:
  assumes 0 ≤ b
  shows ⌊a / real-of-int b⌋ = ⌊a⌋ div b
proof (cases b = 0)
  case True
  then show ?thesis by simp
next
  case False
  with assms have b: b > 0 by simp
  have j = i div b
    if real-of-int i ≤ a a < 1 + real-of-int i
      real-of-int j * real-of-int b ≤ a a < real-of-int b + real-of-int j * real-of-int b
      for i j :: int
    proof
      from that have i < b + j * b
        by (metis le-less-trans of-int-add of-int-less-iff of-int-mult)
moreover have \( j \cdot b < 1 + i \)

proof

- have real-of-int \((j \cdot b) < \text{real-of-int } i + 1\)
  - using \((a < 1 + \text{real-of-int } i) \cdot \text{real-of-int } j \cdot \text{real-of-int } b \leq a\) by force
  then show \( j \cdot b < 1 + i \) by linarith

qed

ultimately have \((j - i \div b) \cdot b \leq i \mod b < ((j - i \div b) + 1) \cdot b\)

then have \((j - i \div b) \cdot b < 1 \cdot b \cdot 0 < ((j - i \div b) + 1) \cdot b\)

using pos-mod-bound \([OF \ b, \ of \ i]\) pos-mod-sign \([OF \ b, \ of \ i]\)

by linarith+

then show \(?\text{thesis}\) using \(b\) unfolding \text{mult-less-cancel-right} by auto

qed with \(b\) show \(?\text{thesis}\) by (auto split: floor-split simp: field-simps)

qed

lemma floor-one-divide-eq-div-numeral [simp]:
\[
\lfloor \frac{1}{\text{numeral } b} \rfloor = \text{numeral } b \div \text{numeral } b
\]

by (metis floor-divide-of-int-eq of-int-1 of-int-numeral)

lemma floor-minus-one-divide-eq-div-numeral [simp]:
\[
\lfloor -\left(\frac{1}{\text{numeral } b}\right) \rfloor = -1 \div \text{numeral } b
\]

by (metis (mono-tags, hide-lams) div-minus-right minus-divide-right

floor-divide-of-int-eq of-int-neg-numeral of-int-1)

lemma floor-divide-eq-div-numeral [simp]:
\[
\lfloor \frac{\text{numeral } a}{\text{numeral } b} \rfloor = \text{numeral } a \div \text{numeral } b
\]

by (metis floor-divide-of-int-eq of-int-numeral)

lemma floor-minus-divide-eq-div-numeral [simp]:
\[
\lfloor -\left(\frac{\text{numeral } a}{\text{numeral } b}\right) \rfloor = -\text{numeral } a \div \text{numeral } b
\]

by (metis divide-minus-left floor-divide-of-int-eq of-int-neg-numeral of-int-numeral)

lemma of-int-ceiling-cancel [simp]: of-int \([x] = x \iff (\exists n::\text{int}. \ x = \text{of-int } n)\)

using ceiling-of-int by metis

lemma ceiling-eq: of-int \(n < x \implies x \leq \text{of-int } n + 1 \implies [x] = n + 1\)

by (simp add: ceiling-unique)

lemma of-int-ceiling-diff-one-le [simp]: of-int \([r] - 1 \leq r\)

by linarith

lemma of-int-ceiling-le-add-one [simp]: of-int \([r] \leq r + 1\)

by linarith

lemma ceiling-le: \(x \leq \text{of-int } a \implies [x] \leq a\)

by (simp add: ceiling-le-iff)

lemma ceiling-divide-eq-div: \([\text{of-int } a] / \text{of-int } b\) = \(-a \div b\)
by (metis ceiling-def floor-divide-of-int-eq minus-divide-left of-int-minus)

lemma ceiling-divide-eq-div-numeral [simp]:
\[\lceil \text{numeral } a / \text{numeral } b \rceil = - (\text{numeral } a \div \text{numeral } b)\]
using ceiling-divide-eq-div[of numeral a numeral b] by simp

lemma ceiling-minus-divide-eq-div-numeral [simp]:
\[\lceil - (\text{numeral } a / \text{numeral } b) \rceil = - (\text{numeral } a \div \text{numeral } b)\]
using ceiling-divide-eq-div[of - numeral a numeral b] by simp

The following lemmas are remnants of the erstwhile functions natfloor and natceiling.

lemma nat-floor-neg: \(x \leq 0 \implies \lceil x \rceil = 0\)
for \(x :: \text{real}\)
by linarith

lemma le-nat-floor: \(x \leq a \implies x \leq \lceil a \rceil\)
by linarith

lemma le-mult-nat-floor: \(\lceil a \rceil \ast \lceil b \rceil \leq \lceil a \ast b \rceil\)
by (cases \(0 \leq a \land 0 \leq b\))
(auto simp add: nat-mult-distrib[symmetric] nat-mono le-mult-floor)

lemma nat-ceiling-le-eq [simp]: \(\lceil x \rceil \leq a \iff x \leq \text{real } a\)
by linarith

lemma real-nat-ceiling-ge: \(x \leq \text{real } (\lceil x \rceil)\)
by linarith

lemma Rats-no-top-le: \(\exists q \in \mathbb{Q}. x \leq q\)
for \(x :: \text{real}\)
by (auto intro!: bexI[of - of-nat (\lceil x \rceil)]) linarith

lemma Rats-no-bot-less: \(\exists q \in \mathbb{Q}. q < x\) for \(x :: \text{real}\)
by (auto intro!: bexI[of - of-int (\lfloor x \rfloor - 1)]) linarith

95.22 Exponentiation with floor

lemma floor-power:
assumes \(x = \text{of-int } \lfloor x \rfloor\)
shows \(\lfloor x \ast n \rfloor = \lfloor x \rfloor \ast n\)
proof -
have \(x \ast n = \text{of-int } (\lfloor x \rfloor \ast n)\)
using assms by (induct n arbitrary: x) simp-all
then show ?thesis by (metis floor-of-int)
qed

lemma floor-numeral-power [simp]: \(\lfloor \text{numeral } x \ast n \rfloor = \text{numeral } x \ast n\)
by (metis floor-of-int of-int-numeral of-int-power)
lemma ceiling-numeral-power [simp]: [numeral x ^ n] = numeral x ^ n
by (metis ceiling-of-int of-int-numeral of-int-power)

95.23 Implementation of rational real numbers

Formal constructor

definition Ratreal :: rat ⇒ real
  where [code-abbrev, simp]: Ratreal = real-of-rat

code-datatype Ratreal

Quasi-Numerals

lemma [code-abbrev]:
  real-of-rat (numeral k) = numeral k
  real-of-rat (− numeral k) = − numeral k
  real-of-rat (rat-of-int a) = real-of-int a
  by simp-all

lemma [code-post]:
  real-of-rat 0 = 0
  real-of-rat 1 = 1
  real-of-rat (− 1) = − 1
  real-of-rat (1 / numeral k) = 1 / numeral k
  real-of-rat (numeral k / numeral l) = numeral k / numeral l
  real-of-rat (− (1 / numeral k)) = − (1 / numeral k)
  real-of-rat (− (numeral k / numeral l)) = − (numeral k / numeral l)
  by (simp-all add: of-rat-divide of-rat-minus)

Operations

lemma zero-real-code [code]: 0 = Ratreal 0
  by simp

lemma one-real-code [code]: 1 = Ratreal 1
  by simp

instantiation real :: equal
begin

definition HOL.equal x y ⟷ x − y = 0 for x :: real
instance by standard (simp add: equal-real-def)

lemma real-equal-code [code]: HOL.equal (Ratreal x) (Ratreal y) ⟷ HOL.equal x y
  by (simp add: equal-real-def equal)

lemma [code nbe]: HOL.equal x x ⟷ True
for $x :: \text{real}$
by (rule equal-refl)

end

lemma real-less-eq-code [code]: $\text{Ratreal } x \leq \text{Ratreal } y \iff x \leq y$
by (simp add: of-rat-less-eq)

lemma real-less-code [code]: $\text{Ratreal } x < \text{Ratreal } y \iff x < y$
by (simp add: of-rat-less)

lemma real-plus-code [code]: $\text{Ratreal } x + \text{Ratreal } y = \text{Ratreal } (x + y)$
by (simp add: of-rat-add)

lemma real-times-code [code]: $\text{Ratreal } x \ast \text{Ratreal } y = \text{Ratreal } (x \ast y)$
by (simp add: of-rat-mult)

lemma real-uminus-code [code]: $- \text{Ratreal } x = \text{Ratreal } (- x)$
by (simp add: of-rat-minus)

lemma real-minus-code [code]: $\text{Ratreal } x - \text{Ratreal } y = \text{Ratreal } (x - y)$
by (simp add: of-rat-diff)

lemma real-inverse-code [code]: $\text{inverse } (\text{Ratreal } x) = \text{Ratreal } (\text{inverse } x)$
by (simp add: of-rat-inverse)

lemma real-divide-code [code]: $\text{Ratreal } x / \text{Ratreal } y = \text{Ratreal } (x / y)$
by (simp add: of-rat-divide)

lemma real-floor-code [code]: $\lfloor \text{Ratreal } x \rfloor = \lfloor x \rfloor$
by (metis Ratreal-def floor-le-iff floor-unique le-floor-iff of-int-floor-le of-rat-of-int-eq real-less-eq-code)

Quickcheck

definition (in term-syntax)
valterm-ratreal :: rat $\times$ (unit $\Rightarrow$ Code-Evaluation.term) $\Rightarrow$ real $\times$ (unit $\Rightarrow$ Code-Evaluation.term)
where [code-unfold]: valterm-ratreal $k = \text{Code-Evaluation.valtermify } \text{Ratreal} \{ \cdot \} k$

notation fcomp (infixl $\triangleright$ 60)
notation scomp (infixl $\triangleright\triangleright$ 60)

instantiation real :: random
begin

definition Quickcheck-Random.random $i = \text{Quickcheck-Random.random } i \triangleright\triangleright (\lambda r. \text{Pair} (\text{valterm-ratreal } r))$
instance ..
end

no-notation fcomp (infixl o > 60)
no-notation scomp (infixl o → 60)

instantiation real :: exhaustive
begin

definition
exhaustive-real f d = Quickcheck-Exhaustive.exhaustive (λ r. f (Ratreal r)) d

instance ..
end

instantiation real :: full-exhaustive
begin

definition
full-exhaustive-real f d = Quickcheck-Exhaustive.full-exhaustive (λ r. f (valterm-ratreal r)) d

instance ..
end

instantiation real :: narrowing
begin

definition
narrowing-real = Quickcheck-Narrowing.apply (Quickcheck-Narrowing.cons Ratreal) narrowing

instance ..
end

95.24 Setup for Nitpick

declaration :
Nitpick-HOL.register-frac-type type-name : (real):
[ (const-name : zero-real-inst.zero-real), const-name : (Nitpick.zero-frac),
 (const-name : one-real-inst.one-real), const-name : (Nitpick.one-frac),
 (const-name : plus-real-inst.plus-real), const-name : (Nitpick.plus-frac),
 (const-name : times-real-inst.times-real), const-name : (Nitpick.times-frac),
 (const-name : uminus-real-inst.uminus-real), const-name : (Nitpick.uminus-frac),
]
one-real ord-real-inst.
less-real ord-real-inst.
less-eq-real plus-real-inst.
plus-real times-real-inst.
times-real uminus-real-inst.
uminus-real zero-real-inst.
zero-real

95.25 Setup for SMT

ML-file ⟨Tools/SMT/smt-real.ML⟩
ML-file ⟨Tools/SMT/z3-real.ML⟩

lemma [z3-rule]:
  0 + x = x
  x + 0 = x
  0 * x = 0
  1 * x = x
  -x = -1 * x
  x + y = y + x
  for x y :: real
  by auto

95.26 Setup for Argo

ML-file ⟨Tools/Argo/argo-real.ML⟩

end

96 Topological Spaces

theory Topological-Spaces
  imports Main
begin

named-theorems continuous-intros structural introduction rules for continuity

96.1 Topological space

class open =
  fixes open :: 'a set ⇒ bool

class topological-space = open +
  assumes open-UNIV [simp, intro]: open UNIV
  assumes open-Int [intro]: open S ⇒ open T ⇒ open (S ∩ T)
  assumes open-Union [intro]: ∀S∈K. open S ⇒ open (∪ K)
begin
definition closed :: 'a set ⇒ bool
where closed S ⟷ open (¬ S)

lemma open-empty [continuous-intros, intro, simp]: open {}
using open-Union [of {}] by simp

lemma open-Un [continuous-intros, intro]: open S ⟹ open T ⟹ open (S ∪ T)
using open-Union [of {S, T}] by simp

lemma open-UN [continuous-intros, intro]: ∀ x ∈ A. open (B x) ⟹ open (∪ x ∈ A. B x)
using open-Union [of B ' A] by simp

lemma open-Inter [continuous-intros, intro]: finite S ⟹ ∀ T ∈ S. open T ⟹ open (∩ S)
by (induct set: finite) auto

lemma open-INT [continuous-intros, intro]: finite A ⟹ ∀ x ∈ A. open (B x) ⟹
open (∩ x ∈ A. B x)
using open-Inter [of B ' A] by simp

lemma openI:
assumes ∃ T. open T ∧ x ∈ T ∧ T ⊆ S
shows open S
proof
have open (⋃ {T. open T ∧ T ⊆ S}) by auto
moreover have ∪ {T. open T ∧ T ⊆ S} = S by (auto dest!: assms)
ultimately show open S by simp
qed

lemma open-subopen: open S ⟷ (∀ x ∈ S. ∃ T. open T ∧ x ∈ T ∧ T ⊆ S)
by (auto intro: openI)

lemma closed-empty [continuous-intros, intro, simp]: closed {}
unfolding closed-def by simp

lemma closed-Un [continuous-intros, intro]: closed S ⟹ closed T ⟹ closed (S ∪ T)
unfolding closed-def by auto

lemma closed-UNIV [continuous-intros, intro, simp]: closed UNIV
unfolding closed-def by simp

lemma closed-Int [continuous-intros, intro]: closed S ⟹ closed T ⟹ closed (S ∩ T)
unfolding closed-def by auto

lemma closed-INT [continuous-intros, intro]: ∀ x ∈ A. closed (B x) ⟹ closed (∩ x ∈ A.
B x)

unfolding closed-def by auto

lemma closed-Inter [continuous-intros, intro]: \( \forall S \in K. \) closed \( S \) \( \implies \) closed \( \bigcap K \)
unfolding closed-def aminus-Inf by auto

lemma closed-Union [continuous-intros, intro]: finite \( S \) \( \implies \) \( \forall T \in S. \) closed \( T \) \( \implies \) closed \( \bigcup S \)
by (induct set: finite) auto

lemma closed-UN [continuous-intros, intro]:
finite \( A \) \( \implies \) \( \forall x \in A. \) closed \( (B x) \) \( \implies \) closed \( \bigcup x \in A. B x \)
using closed-Union [of B ' A] by simp

lemma open-closed: open \( S \) \( \iff \) closed \( \neg S \)
by (simp add: closed-def)

lemma open-Diff [continuous-intros, intro]: open \( S \) \( \implies \) closed \( T \) \( \implies \) open \( S - T \)
by (simp add: closed-open Diff-eq open-Int)

lemma closed-Diff [continuous-intros, intro]: closed \( S \) \( \implies \) open \( T \) \( \implies \) closed \( S - T \)
by (simp add: open-closed Diff-eq closed-Int)

lemma open-Compl [continuous-intros, intro]: closed \( S \) \( \implies \) open \( \neg S \)
by (simp add: closed-open)

lemma closed-Compl [continuous-intros, intro]: open \( S \) \( \implies \) closed \( \neg S \)
by (simp add: open-closed)

lemma open-Collect-neg: closed \( \{ x. \ P x \} \) \( \implies \) open \( \{ x. \neg P x \} \)
unfolding Collect-neg-eq by (rule open-Compl)

lemma open-Collect-conj:
assumes open \( \{ x. \ P x \} \) open \( \{ x. \ Q x \} \)
shows open \( \{ x. \ P x \wedge Q x \} \)
using open-Int[OF assms] by (simp add: Int-def)

lemma open-Collect-disj:
assumes open \( \{ x. \ P x \} \) open \( \{ x. \ Q x \} \)
shows open \( \{ x. \ P x \lor Q x \} \)
using open-Un[OF assms] by (simp add: Un-def)

lemma open-Collect-ex: \( (\forall i. \ ) open \( \{ x. \ P i x \} \) \) \( \implies \) open \( \{ x. \exists i. P i x \} \)
using open-UN[of UNIV \( \lambda i. \ ) \{ x. P i x \}] unfolding Collect-ex-eq by simp
lemma open-Collect-imp: closed \{ x. P x \} \implies open \{ x. Q x \} \implies open \{ x. P x \ \rightarrow Q x \}
unfolding imp-cone-disj by (intro open-Collect-disj open-Collect-neg)

lemma open-Collect-const: open \{ x. P \}
by (cases P) auto

lemma closed-Collect-neg: open \{ x. P x \} \implies closed \{ x. \neg P x \}
unfolding Collect-neg-eq by (rule closed-Compl)

lemma closed-Collect-conj:
assumes closed \{ x. P x \} closed \{ x. Q x \}
shows closed \{ x. P x \land Q x \}
using closed-Int[OF assms] by (simp add: Int-def)

lemma closed-Collect-disj:
assumes closed \{ x. P x \} closed \{ x. Q x \}
shows closed \{ x. P x \lor Q x \}
using closed-Un[OF assms] by (simp add: Un-def)

lemma closed-Collect-all:
\((\forall i. \text{closed } \{ x. P i x \}) \implies \text{closed } \{ x. \forall i. P i x \}\)
using closed-INT[of UNIV \lambda i. \{ x. P i x \}] by (simp add: Collect-all-eq)

lemma closed-Collect-imp: open \{ x. P x \} \implies closed \{ x. Q x \} \implies closed \{ x. P x \ \rightarrow Q x \}
unfolding imp-cone-disj by (intro closed-Collect-disj closed-Collect-neg)

lemma closed-Collect-const: closed \{ x. P \}
by (cases P) auto

end

96.2 Hausdorff and other separation properties

class t0-space = topological-space +
assumes t0-space: \( x \neq y \implies \exists U. \text{open } U \land \neg (x \in U \leftrightarrow y \in U) \)

class t1-space = topological-space +
assumes t1-space: \( x \neq y \implies \exists U. \text{open } U \land x \in U \land y \notin U \)

instance t1-space \subseteq t0-space
by standard (fast dest: t1-space)

context t1-space begin

lemma separation-t1: \( x \neq y \leftrightarrow (\exists U. \text{open } U \land x \in U \land y \notin U) \)
using t1-space[of x y] by blast
lemma closed-singleton [iff]: closed \{a\}
proof −
  let \(?T = \bigcup\{S. \text{open } S \land a \notin S\}\)
  have open \(?T\)
    by (simp add: open-Union)
  also have \(?T = -\{a\}\)
    by (auto simp add: set-eq-iff separation-t1)
  finally show closed \{a\}
    by (simp only: closed-def)
qed

lemma closed-insert [continuous-intros, simp]:
  assumes closed \(S\)
  shows closed (insert \(a\) \(S\))
proof −
  from closed-singleton assms have closed (\{a\} \(\cup\) \(S\))
    by (rule closed-Un)
  then show closed (insert \(a\) \(S\))
    by simp
qed

lemma finite-imp-closed: finite \(S\) \(\Rightarrow\) closed \(S\)
  by (induct pred: finite) simp-all
end

T2 spaces are also known as Hausdorff spaces.

class t2-space = topological-space +
  assumes hausdorff: \(x \neq y \implies \exists U \ V. \text{open } U \land \text{open } V \land x \in U \land y \in V \land U \cap V = \emptyset\)

instance t2-space \(\subseteq\) t1-space
  by standard (fast dest: hausdorff)

lemma (in t2-space) separation-t2: \(x \neq y \iff (\exists U \ V. \text{open } U \land \text{open } V \land x \in U \land y \in V \land U \cap V = \emptyset)\)
  using hausdorff [of \(x\) \(y\)] by blast

lemma (in t0-space) separation-t0: \(x \neq y \iff (\exists U. \text{open } U \land \neg (x \in U \iff y \in U))\)
  using t0-space [of \(x\) \(y\)] by blast

A classical separation axiom for topological space, the T3 axiom – also called regularity: if a point is not in a closed set, then there are open sets separating them.

class t3-space = t2-space +
  assumes t3-space: closed \(S\) \(\Rightarrow\) \(y \notin S \implies \exists U \ V. \text{open } U \land \text{open } V \land y \in U \land S \subseteq V \land U \cap V = \emptyset\)
A classical separation axiom for topological space, the T4 axiom – also called normality: if two closed sets are disjoint, then there are open sets separating them.

\[
\text{class } t4\text{-space } = t2\text{-space } + \\
\text{assumes } t4\text{-space: } \text{closed } S \implies \text{closed } T \implies S \cap T = \{\} \implies \exists U \ V. \text{open } U \\
\text{open } V \wedge S \subseteq U \wedge T \subseteq V \wedge U \cap V = \{\}
\]

T4 is stronger than T3, and weaker than metric.

\[
\text{instance } t4\text{-space } \subseteq t3\text{-space}
\]

\[
\text{proof } \\
\text{fix } S \text{ and } y::'a \text{ assume closed } S \ y \notin S \text{ then show } \exists U \ V. \text{open } U \wedge \text{open } V \wedge y \in U \wedge S \subseteq V \wedge U \cap V = \{\} \\
\text{using } t4\text{-space}\{\{y\}\} S \text{ by auto }
\]

qed

A perfect space is a topological space with no isolated points.

\[
\text{class } \text{perfect-space } = \text{topological-space } + \\
\text{assumes } \text{not-open-singleton: } \neg \text{open } \{x\}
\]

\[
\text{lemma (in perfect-space) } \text{UNIV\text{-}not\text{-}singleton: } \text{UNIV } \neq \{x\}
\]

\[
\text{for } x::'a \text{ by (metis (no-types) open\text{-}UNIV not\text{-}open\text{-}singleton)}
\]

96.3 Generators for topologies

\[
\text{inductive } \text{generate\text{-}topology } :: \ 'a \text{ set set } \Rightarrow \ 'a \text{ set } \Rightarrow \text{ bool } \text{ for } S :: \ 'a \text{ set set } \\
\text{where } \\
\text{UNIV: } \text{generate\text{-}topology } S \text{ UNIV } \\
| \text{Int: } \text{generate\text{-}topology } S (a \cap b) \text{ if } \text{generate\text{-}topology } S a \text{ and } \text{generate\text{-}topology } S b \\
| \text{UN: } \text{generate\text{-}topology } S (\bigcup K) \text{ if } (\\text{\forall } k. \ k \in K \implies \text{generate\text{-}topology } S k) \\
| \text{Basis: } \text{generate\text{-}topology } S s \text{ if } s \in S
\]

\[
\text{hide\text{-}fact (open) } \text{UNIV Int UN Basis}
\]

\[
\text{lemma } \text{generate\text{-}topology\text{-}Union: } \\
(\text{\forall } k. \ k \in I \implies \text{generate\text{-}topology } S (K k)) \implies \text{generate\text{-}topology } S (\bigcup k \in I. \ K k) \\
\text{using } \text{generate\text{-}topology.\text{UN } [of K \ I]} \text{ by auto}
\]

\[
\text{lemma } \text{topological\text{-}space\text{-}generate\text{-}topology: } \text{class.topological\text{-}space } (\text{generate\text{-}topology } S) \\
\text{by standard (auto intro: generate\text{-}topology.intros)}
\]

96.4 Order topologies

\[
\text{class } \text{order\text{-}topology } = \text{order } + \text{ open } + \\
\text{assumes } \text{open\text{-}generated\text{-}order: } \text{open } = \text{generate\text{-}topology } (\lambda a. \{..< a\}) \\
\cup \text{range } (\lambda a. \{a <..\}))
\]
begin

subclass topological-space
  unfolding open-generated-order
  by (rule topological-space-generate-topology)

lemma open-greaterThan [continuous-intros, simp]: open \{ a <.. \}
  unfolding open-generated-order by (auto intro: generate-topology,Basis)

lemma open-lessThan [continuous-intros, simp]: open \{ ..< a \}
  unfolding open-generated-order by (auto intro: generate-topology,Basis)

lemma open-greaterThanLessThan [continuous-intros, simp]: open \{ a <..< b \}
  unfolding greaterThanLessThan-eq by (simp add: open-Int)

end

class linorder-topology = linorder + order-topology

lemma closed-atMost [continuous-intros, simp]: closed \{ ..a \}
  for a :: 'a::linorder-topology
  by (simp add: closed-open)

lemma closed-atLeast [continuous-intros, simp]: closed \{ a.. \}
  for a :: 'a::linorder-topology
  by (simp add: closed-open)

lemma closed-atLeastAtMost [continuous-intros, simp]: closed \{ a..b \}
  for a b :: 'a::linorder-topology
proof –
  have \{ a .. b \} = \{ a .. \} \cap \{ .. b \}
    by auto
  then show \?thesis
    by (simp add: closed-Int)
qed

lemma (in order) less-separate:
  assumes x < y
  shows \exists a b. x \in \{..<a\} \land y \in \{b<..\} \land \{..<a\} \cap \{b<..\} = \{}
proof (cases \exists z. x < z \land z < y )
  case True
  then obtain z where x < z \land z < y ..
  then have x \in \{..<z\} \land y \in \{z<..\} \land \{z<..\} \cap \{..<z\} = \{}
    by auto
  then show \?thesis by blast
next
  case False
  with (x < y) have x \in \{..<y\} \land y \in \{x<..\} \land \{..<y\} = \{}
    by auto
then show \(?thesis\) by blast
qed

instance linorder-topology \(\subseteq\) t2-space
proof
  fix \(x\ y\ ::\ 'a\)
  show \(x \neq y \implies \exists U\ V.\ \text{open}\ U \land \text{open}\ V \land x \in U \land y \in V \land U \cap V = \{\}\)
  using less-separate [of \(x\ y\)] less-separate [of \(y\ x\)]
  by (elim neqE\; metis open-lessThan open-greaterThan Int-commute)
qed

lemma (in linorder-topology) open-right:
  assumes open \(S\) \(x\ \in\ S\)
  and gt-ex: \(x < y\)
  shows \(\exists b > x.\ \{x ..< b\} \subseteq S\)
  using assms unfolding open-generated-order
proof induct
  case UNIV
  then show \(?case\) by blast
next
  case (Int \(A\ B\))
  then obtain \(a\ b\) where \(a > x\ \{x ..< a\} \subseteq A\) \(b > x\ \{x ..< b\} \subseteq B\)
  by auto
  then show \(?case\)
  by (auto intro: exI[of - min \(a\ b\)])
next
  case UN
  then show \(?case\) by blast
next
  case Basis
  then show \(?case\)
  by (fastforce intro: exI[of - \(y\)] gt-ex)
qed

lemma (in linorder-topology) open-left:
  assumes open \(S\) \(x\ \in\ S\)
  and lt-ex: \(y < x\)
  shows \(\exists b < x.\ \{b ..< x\} \subseteq S\)
  using assms unfolding open-generated-order
proof induction
  case UNIV
  then show \(?case\) by blast
next
  case (Int \(A\ B\))
  then obtain \(a\ b\) where \(a < x\ \{a ..< x\} \subseteq A\) \(b < x\ \{b ..< x\} \subseteq B\)
  by auto
  then show \(?case\)
  by (auto intro: exI[of - max \(a\ b\)])
next
case UN
  then show ?case by blast
next
  case Basis
  then show ?case
    by (fastforce intro: exI[of - y] lt-ex)
qed

96.5 Setup some topologies

96.5.1 Boolean is an order topology

class discrete-topology = topological-space +
  assumes open-discrete: \( \forall A. \text{open } A \)

instance discrete-topology < t2-space
proof
  fix x y :: 'a
  assume x ≠ y
  then show \( \exists U V. \text{open } U \land \text{open } V \land x \in U \land y \in V \land U \cap V = {} \)
    by (intro exI[of - {}]) (auto intro!: open-discrete)
qed

instantiation bool :: linorder-topology
begin

definition open-bool :: bool set ⇒ bool
  where open-bool = generate-topology (range (λ a. {..< a})) \( \cup \) range (λ a. {a <..})

instance
  by standard (rule open-bool-def)
end

instance bool :: discrete-topology
proof
  fix A :: bool set
  have *: {False <..} = {True} {..< True} = {False}
    by auto
  have A = UNIV ∨ A = {} ∨ A = {False <..} ∨ A = {..< True}
    using subset-UNIV[of A] unfolding UNIV-bool * by blast
  then show open A
    by auto
qed

instantiation nat :: linorder-topology
begin

definition open-nat :: nat set ⇒ bool
  where open-nat = generate-topology (range (λ a. {..< a})) \( \cup \) range (λ a. {a <..})
instance
by standard (rule open-nat-def)
end

instance nat :: discrete-topology
proof
fix A :: nat set
have open \{n\} for n :: nat
proof (cases n)
case 0
moreover have \{0\} = \{..<1::nat\}
by auto
ultimately show \?thesis
by auto
next
case (Suc n)
then have \{n\} = \{..<Suc n\} ∩ \{n’..<\}
by auto
with Suc show \?thesis
by (auto intro: open-lessThan open-greaterThan)
qed
then have open (⋃ a∈A. \{a\})
by (intro open-UN) auto
then show open A
by simp
qed

instantiation int :: linorder-topology
begin

definition open-int :: int set ⇒ bool
where open-int = generate-topology (range (λa. \{..< a\}) ∪ range (λa. \{a <..<\}))

instance
by standard (rule open-int-def)
end

instance int :: discrete-topology
proof
fix A :: int set
have \{..<i + 1\} ∩ \{i-1 <..<\} = \{i\} for i :: int
by auto
then have open \{i\} for i :: int
using open-Int[of open-lessThan[of i + 1] open-greaterThan[of i - 1]] by auto
then have open (⋃ a∈A. \{a\})
by (intro open-UN) auto
then show open A
by simp
qed

96.5.2 Topological filters

definition (in topological-space) nhds :: 'a ⇒ 'a filter
where nhds a = (INF S∈{S. open S ∧ a ∈ S}. principal S)

definition (in topological-space) at-within :: 'a ⇒ 'a set ⇒ 'a filter
(at (-)/ within (-) [1000, 60] 60)
where at a within s = inf (nhds a) (principal (s - {a}))

abbreviation (in topological-space) at :: 'a ⇒ 'a filter
where at x ≡ at x within (CONST UNIV)

abbreviation (in order-topology) at-right :: 'a ⇒ 'a filter
where at-right x ≡ at x within {x<..}

abbreviation (in order-topology) at-left :: 'a ⇒ 'a filter
where at-left x ≡ at x within {..<x}

lemma (in topological-space) nhds-generated-topology:
open = generate-topology T ⇒ nhds x = (INF S∈{S∈T. x ∈ S}. principal S)
unfolding nhds-def
proof (safe intro!: antisym INF-greatest)
  fix S
  assume generate-topology T S x ∈ S
  then show (INF S∈{S∈T. x ∈ S}. principal S) ≤ principal S
  by induct
qed (auto intro!: INF-lower intro: generate-topology.intros)

lemma (in topological-space) eventually-nhds:
eventually P (nhds a) ⟷ (∃S. open S ∧ a ∈ S ∧ (∀x∈S. P x))
unfolding nhds-def by (subst eventually-INF-base) (auto simp: eventually-principal)

lemma eventually-eventually:
eventually (λy. eventually P (nhds y)) (nhds x) = eventually P (nhds x)
by (auto simp: eventually-nhds)

lemma (in topological-space) eventually-nhds-in-open:
open s ⇒ x ∈ s ⇒ eventually (λy. y ∈ s) (nhds x)
by (subst eventually-nhds) blast

lemma (in topological-space) eventually-nhds-x-imp-x: eventually P (nhds x) ⇒ P x
THEORY “Topological-Spaces”

by (subst (asm) eventually-nhds) blast

lemma (in topological-space) nhds-neq-bot [simp]: nhds a ≠ bot
  by (simp add: trivial-limit-def eventually-nhds)

lemma (in t1-space) t1-space-nhds: x ≠ y⟹ (∀ F in nhds x, x ≠ y)
  by (drule t1-space) (auto simp: eventual-nhds)

lemma (in topological-space) nhds-discrete-open: open {x}⟹ nhds x = principal {x}
  by (auto simp: nhds-def intro antisym INF-greatest INF-lower2[of {x}])

lemma (in discrete-topology) nhds-discrete: nhds x = principal {x}
  by (simp add: nhds-discrete-open open-discrete)

lemma (in discrete-topology) at-discrete: at x within S = bot
  unfolding at-within-def nhds-discrete by simp

lemma (in discrete-topology) tendsto-discrete:
  filterlim (f :: 'b ⇒ 'a) (nhds y) F ↔ eventually (λx. f x = y) F
  by (auto simp: nhds-discrete filterlim-principal)

lemma (in topological-space) at-within-eq:
  at x within s = (INF S∈{S. open S ∧ x ∈ S}. principal (S ∩ s − {x}))
  unfolding nhds-def at-within-def
  by (subst INF-inf-const2[symmetric]) (auto simp: Diff-Int-distrib)

lemma (in topological-space) eventually-at-filter:
  eventually P (at a within s) ↔ eventually (λx. x ≠ a → x ∈ s → P x) (nhds a)
  by (simp add: at-within-def eventually-inf-principal imp-conjL[symmetric] conj-commute)

lemma (in topological-space) at-le: s ⊆ t ⟹ at x within s ≤ at x within t
  unfolding at-within-def by (intro inf-mono) auto

lemma (in topological-space) eventually-at-topological:
  eventually P (at a within s) ↔ (∃S. open S ∧ a ∈ S ∧ (∀x∈S. x ≠ a → x ∈ s → P x))
  by (simp add: eventually-nhds eventually-at-filter)

lemma (in topological-space) at-within-open: a ∈ S ⟹ open S ⟹ at a within S = at a
  unfolding filter-eq-iff eventually-at-topological by (metis open-Int Int-iff UNIV-I)

lemma (in topological-space) at-within-open-NO-MATCH:
  a ∈ s ⟹ open s ⟹ NO-MATCH UNIV s ⟹ at a within s = at a
  by (simp only: at-within-open)

lemma (in topological-space) at-within-open-subset:
\(a \in S \implies \text{open } S \implies S \subseteq T \implies \text{at } a \text{ within } T = a\)

by (metis at-le at-within-open dual-order.antisym subset-UNIV).

**Lemma (in topological-space) at-within-nhd:**

assumes \(x \in S \text{ open } S \cap T - \{x\} = U \cap S - \{x\}\)

shows \(\text{at } x \text{ within } T = \text{at } x \text{ within } U\)

**Proof** (intro allI eventually-subst)

have \(\text{eventually } (\lambda x. x \in S) \text{ (nhds } x)\)

using \((x \in S) \text{ (open } S)\) by (auto simp: eventually-nhds)

then show \(\forall F \in \text{nhds } x. (n \neq x \implies n \in T \implies P \ n) = (n \neq x \implies n \in U \implies P \ n)\) for \(P\)

by eventually-elim (insert \((T \cap S - \{x\} = U \cap S - \{x\}),\)) blast

qed

**Lemma (in topological-space) at-within-empty [simp]:\(a\) at within \(\{}\) = bot

**Unfolding** at-within-def by simp

**Lemma (in topological-space) at-within-union:**

\(at \ x \text{ within } (S \cup T) = \sup (at \ x \text{ within } S) \ (at \ x \text{ within } T)\)

**Unfolding** filter-eq-iff eventually-sup eventually-at-filter

by (auto elim!: eventually-rev-mp)

**Lemma (in topological-space) at-eq-bot-iff:** \(a = \bot \iff \text{open } \{a\}\)

**Unfolding** trivial-limit-def eventually-at-topological

apply safe

apply (case-tac \(S = \{a\}\))

apply simp

apply fast

apply fast

done

**Lemma (in perfect-space) at-neq-bot [simp]:\(a\) \(\neq\) bot

by (simp add: at-eq-bot-iff not-open-singleton)

**Lemma (in order-topology) nhds-order:**

\(\text{nhds } x = \inf \ (\text{INF } a \in \{x <..\}. \text{principal } \{..< a\}) \ (\text{INF } a \in \{..< x\}. \text{principal } \{a <..\})\)

**Proof**

- have 1: \(\{S \in \text{range } \text{lessThan } \cup \text{range } \text{greaterThan}. \ x \in S\} = \)

  \((\lambda a. \ [..< a]) \cdot \{x <..\} \cup (\lambda a. \{a <..\}) \cdot \{..< x\}\)

  by auto

  show ?thesis

  by (simp only: nhds-generated-topology[of open-generated-order] INF-union 1 INF-image comp-def)

qed

**Lemma (in topological-space) filterlim-at-within-If:**

assumes \(\text{filterlim } f \ G \ (at \ x \text{ within } (A \cap \{x. \ P \ x\}))\)
and filterlim \( g \) \( (at \ x \ within \ (A \cap \{x. \neg P \ x\})) \)
shows \( \text{filterlim} \ (\lambda x. \text{if } P \ x \text{ then } f \ x \text{ else } g \ x) \ G \ (at \ x \ within \ A) \)
proof (rule filterlim-If)

note \( \text{assms} \ 1 \)
also have \( at \ x \ within \ (A \cap \{x. P \ x\}) = \text{inf} \ (\text{nhds} \ x) \) \( (\text{principal} \ (A \cap \text{Collect } P - \{x\})) \)
by (simp add: at-within-def)
also have \( A \cap \text{Collect } P - \{x\} = (A - \{x\}) \cap \text{Collect } P \)
by blast
also have \( \text{inf} \ (\text{nhds} \ x) \) \( (\text{principal} \ldots) = \text{inf} \ (at \ x \ within \ A) \) \( (\text{principal} \ \text{Collect } P) \)
by (simp add: at-within-def inf-assoc)
finally show \( \text{filterlim} \ f \ G \ (\text{inf} \ (at \ x \ within \ A) \) \( (\text{principal} \ \text{Collect } P)) \).

next

note \( \text{assms} \ 2 \)
also have \( at \ x \ within \ (A \cap \{x. \neg P \ x\}) = \text{inf} \ (\text{nhds} \ x) \) \( (\text{principal} \ {x. \neg P \ x}) \)
by blast
also have \( \text{inf} \ (\text{nhds} \ x) \) \( (\text{principal} \ldots) = \text{inf} \ (at \ x \ within \ A) \) \( (\text{principal} \ x. \neg P \ x) \)
by (simp add: at-within-def inf-assoc)
finally show \( \text{filterlim} \ g \ G \ (\text{inf} \ (at \ x \ within \ A) \) \( (\text{principal} \ x. \neg P \ x)) \).

qed

lemma (in topological-space) filterlim-at-If:
assumes \( \text{filterlim} \ f \ G \ (at \ x \ within \ x. P \ x) \)
and \( \text{filterlim} \ g \ G \ (at \ x \ within \ x. \neg P \ x) \)
shows \( \text{filterlim} \ (\lambda x. \text{if } P \ x \text{ then } f \ x \text{ else } g \ x) \ G \ (at \ x) \)
using \( \text{assms} \) by (intro filterlim-at-within-If) simp-all

lemma (in linorder-topology) at-within-order:
assumes \( \text{UNIV} \neq \{x\} \)
shows \( at \ x \ within \ s = \) \( \text{inf} \ (\text{INF } a \in \{x < \ldots\}. \text{principal} \ (\{x < a\} \cap s - \{x\})) \)
\( (\text{INF } a \in \{x < \ldots\}. \text{principal} \ (\{a < x\} \cap s - \{x\})) \)
proof (cases \( \{x < \ldots\} = \{\} \) \( \{x < \ldots\} = \{\} \) rule: case-split [case-product case-split])
case \( \text{True-True} \)
have \( \text{UNIV} = \{x \} \cup \{x\} \cup \{x < \ldots\} \)
by auto
with \( \text{assms} \) \( \text{True-True} \) show \( ?\text{thesis} \)
by auto
qed (auto simp del: \text{inf-principal simp: at-within-def nhds-order Int-Diff inf-principal}[symmetric] \text{INF-inf-const2 inf-sup-aci}[\text{where } a = \text{a filter}] \text{where } a = \text{a filter})

lemma (in linorder-topology) at-left-eq:
\( y < x \implies at-left \ x = (\text{INF } a \in \{x < \ldots\}. \text{principal} \{a < x\}) \)
by (subst at-within-order)
\( (\text{auto simp: greaterThan-Int-greaterThan greaterThanLessThan-eq}[\text{symmetric}] \text{symmetric}) \)
THEORY "Topological-Spaces"

```
min.absorb2 INF-constant
intro!: INF-lower2 inf-absorb2)

lemma (in linorder-topology) eventually-at-left:
y < x \implies eventually P (at-left x) \iff (\exists b < x. \forall y > b. y < x \implies P y)
unfolding at-left-eq
by (subst eventually-INF-base) (auto simp: eventually-principal Ball-def)

lemma (in linorder-topology) at-right-eq:
x < y \implies at-right x = (INF a \in \{x <..\}. principal \{x <..< a\})
by (subst at-within-order)
(auto simp: lessThan-Int-lessThan greaterThanLessThan-eq[symmetric] max.absorb2
INF-constant Int-commute
intro!: INF-lower2 inf-absorb1)

lemma (in linorder-topology) eventually-at-right:
x < y \implies eventually P (at-right x) \iff (\exists b > x. \forall y > x. y < b \implies P y)
unfolding at-right-eq
by (subst eventually-INF-base) (auto simp: eventually-principal Ball-def)

lemma eventually-at-right-less: \forall F y \in at-right (x::'a::linorder-topology, no-top).
x < y
using gt-ex[of x] eventually-at-right[of x] by auto

lemma trivial-limit-at-right-top: at-right (top::'a::order-top,linorder-topology) = bot
by (auto simp: filter-eq-iff eventually-at-topological)

lemma trivial-limit-at-left-bot: at-left (bot::'a::order-bot,linorder-topology) = bot
by (auto simp: filter-eq-iff eventually-at-topological)

lemma trivial-limit-at-left-real [simp]: \neg trivial-limit (at-left x)
for x :: 'a::no-bot,dense-order,linorder-topology
using lt-ex[of x]
by safe (auto simp add: trivial-limit-def eventually-at-left dest: dense)

lemma trivial-limit-at-right-real [simp]: \neg trivial-limit (at-right x)
for x :: 'a::no-top,dense-order,linorder-topology
using gt-ex[of x]
by safe (auto simp add: trivial-limit-def eventually-at-right dest: dense)

lemma (in linorder-topology) at-eq-sup-left-right: at x = sup (at-left x) (at-right x)
by (auto simp: eventually-at-filter filter-eq-iff eventually-sup elim: eventually-eq-sup elim2 eventually-mono)

lemma (in linorder-topology) eventually-at-split:
eventually P (at x) \iff eventually P (at-left x) \land eventually P (at-right x)
by (subst at-eq-sup-left-right) (simp add: eventually-sup)
```
lemma (in order-topology) eventually-at-leftI:
  assumes \( \forall x. x \in \{a..b\} \Rightarrow P \ x \ a < b \)
  shows \( \text{eventually } P \ (\text{at-left } b) \)
  using assms unfolding eventually-at-topological by (intro exI[of - \{a..<\}]) auto

lemma (in order-topology) eventually-at-rightI:
  assumes \( \forall x. x \in \{a..b\} \Rightarrow P \ x \ a < b \)
  shows \( \text{eventually } P \ (\text{at-right } a) \)
  using assms unfolding eventually-at-topological by (intro exI[of - \{..<b\}]) auto

lemma eventually-filtercomap-nhds:
  eventually \( P \ (\text{filtercomap } f \ (\text{nhds } x)) \) \( \iff \) \( \exists S. \text{open } S \land x \in S \land (\forall x. f x \in S \Rightarrow P x) \)
  unfolding eventually-filtercomap eventually-nhds by auto

lemma eventually-filtercomap-at-topological:
  eventually \( P \ (\text{filtercomap } f \ (\text{at A within B})) \) \( \iff \) \( \exists S. \text{open } S \land A \in S \land (\forall x. f x \in S \cap B - \{A\} \Rightarrow P x) \) (is \$lhs = ?rhs)
  unfolding at-within-def filtercomap-inf eventually-inf-principal filtercomap-principal
  by (blast)

lemma eventually-at-right-field:
  eventually \( P \ (\text{at-right } x) \) \( \iff \) \( \exists b > x. \forall y > x. y < b \Rightarrow P y \)
  for \( x :: 'a::{\text{linordered-field, linorder-topology}} \)
  using linordered-field-no-ub[rule-format, of x]
  by (auto simp: eventually-at-right)

lemma eventually-at-left-field:
  eventually \( P \ (\text{at-left } x) \) \( \iff \) \( \exists b < x. \forall y > b. y < x \Rightarrow P y \)
  for \( x :: 'a::{\text{linordered-field, linorder-topology}} \)
  using linordered-field-no-lb[rule-format, of x]
  by (auto simp: eventually-at-left)

96.5.3 Tendsto

abbreviation (in topological-space)
  tendsto :: (\'b \Rightarrow \'a) \Rightarrow \'a \Rightarrow \'b filter \Rightarrow bool (infixr \( \longrightarrow \) 55)
  where \( (f \longrightarrow l) \ F \equiv \text{filterlim } f \ (\text{nhds } l) \ F \)

definition (in t2-space)
  Lim :: \'f filter \Rightarrow (\'f \Rightarrow \'a) \Rightarrow \'a
  where \( \text{Lim } A \ f = (\text{THE } l. (f \longrightarrow l) A) \)

lemma (in topological-space) tendsto-eq-rhs: \( (f \longrightarrow x) \ F \Rightarrow x = y \Rightarrow (f \longrightarrow y) \ F \)
  by simp

named-theorems tendsto-intros introduction rules for tendsto
setup:
Global-Theory.add-thms-dynamic (binding tendsto-eq-intros),
  fn context =>
    Named-Theorems.get (Context.proof-of context) named-theorems (tendsto-intros)
  |> map-filter (try fn thm => @{thm tendsto-eq-rhs} OF [thm]))

context topological-space begin

lemma tendsto-def:
(f −→ l) F ←→ (∀S. open S → l ∈ S → eventually (λx. f x ∈ S) F)
unfolding nhds-def filterlim-INF filterlim-principal by auto

lemma tendsto-cong: (f −→ c) F ←→ (g −→ c) F if eventually (λx. f x = g)
unfolding tendsto-def le-filter-def by fast

lemma tendsto-mono: F ≤ F' ⇒ (f −→ l) F' ⇒ (f −→ l) F
unfolding tendsto-def le-filter-def by fast

lemma tendsto-ident-at [tendsto-intros, simp, intro]: ((λx. x) −→ a) (at a within s)
by (auto simp: tendsto-def eventually-at-topological)

lemma tendsto-const [tendsto-intros, simp, intro]: ((λx. k) −→ k) F
by (simp add: tendsto-def)

lemma filterlim-at: (LIM x F. f x :> at b within s) ←→ eventually (λx. f x ∈ s ∧ f x ≠ b) F ∧ (f
  −→ b) F
by (simp add: at-within-def filterlim-inf filterlim-principal conj-commute)

lemma (in −)
assumes filterlim f (nhds L) F
shows tendsto-imp-filterlim-at-right:
  eventually (λx. f x > L) F ⇒ filterlim f (at-right L) F
and tendsto-imp-filterlim-at-left:
  eventually (λx. f x < L) F ⇒ filterlim f (at-left L) F
using assms by (auto simp: filterlim-at elim: eventually-mono)

lemma filterlim-at-withinI:
assumes filterlim f (nhds c) F
assumes eventually (λx. f x ∈ A − {c}) F
shows filterlim f (at c within A) F
using assms by (simp add: filterlim-at)

lemma filterlim-atI:
assumes filterlim f (nhds c) F
assumes eventually (λx. f x ≠ c) F
"Topological-Spaces"

shows \( \lim_{\to c} f \) \( F \)
using assms by (intro filterlim-at-withinI) simp-all

lemma topological-tendstoI:
\[
(\forall S. \text{open } S \implies l \in S \implies \text{eventually } (\lambda x. f x \in S) \implies (f \to l) \implies (f \to l) F
\]
by (auto simp: tendsto-def)

lemma topological-tendstoD:
\[
(f \to l) F \to \text{open } S \implies l \in S \implies \text{eventually } (\lambda x. f x \in S) \implies (f \to l) F
\]
by (auto simp: tendsto-def)

lemma tendsto-bot [simp]: \( (f \to a) \) \( \bot \)
by (simp add: tendsto-def)

lemma tendsto-eventually:
\[
(\lambda x. f x = l) \text{ net} \to ((\lambda x. f x \to l) \text{ net}) = \to ((\lambda x. f x \to l) \text{ net})
\]
by (rule topological-tendstoI) (auto elim: eventually-mono)

end

lemma (in topological-space) filterlim-within-subset:
\[
\lim_{\to x} f l \text{ (at } x \text{ within } S) = T \subseteq S \implies \lim_{\to x} f l \text{ (at } x \text{ within } T) = T \subseteq S \implies \lim_{\to x} f l \text{ (at } x \text{ within } T)
\]
by (blast intro: filterlim-mono at-le)

lemmas tendsto-within-subset = filterlim-within-subset

lemma (in order-topology) order-tendsto-iff:
\[
(f \to x) F \iff (\forall l < x. \text{eventually } (\lambda x. l < f x) \implies (\forall u > x. \text{eventually } (\lambda x. f x < u) \implies (f \to y) F)
\]
by (auto simp: nhds-order filterlim-inf filterlim-INF filterlim-principal)

lemma (in order-topology) order-tendstoI:
\[
(\forall a. a < y \implies \text{eventually } (\lambda x. a < f x) \implies (\lambda x. f x < a) \implies (f \to y) F)
\]
by (auto simp: order-tendsto-iff)

lemma (in order-topology) order-tendstoD:
\[
\text{assumes } (f \to y) F
\]
\[
\text{shows } a < y \implies \text{eventually } (\lambda x. a < f x) F
\]
\[
\text{and } y < a \implies \text{eventually } (\lambda x. f x < a) F
\]
using assms by (auto simp: order-tendsto-iff)

lemma (in linorder-topology) tendsto-max [tendsto-intros]:
\[
\text{assumes } X: (X \to x) \text{ net}
\]
\[
\text{and } Y: (Y \to y) \text{ net}
\]
\[
\text{shows } ((\lambda x. \max (X x) (Y x)) \to \max x y) \text{ net}
\]
proof (rule order-tendstoI)
\[
\text{fix } a
\]
\[
\text{assume } a < \max x y
\]
then show eventually (λx. a < max (X x) (Y x)) net
using order-tendstoD(1)(OF X, of a) order-tendstoD(1)(OF Y, of a)
by (auto simp: less-max-iff-disj elim: eventually-mono)

next
fix a
assume max x y < a
then show eventually (λx. max (X x) (Y x) < a) net
using order-tendstoD(2)(OF X, of a) order-tendstoD(2)(OF Y, of a)
by (auto simp: eventually-conj-iff)

qed

lemma (in linorder-topology) tendsto-min[tendsto-intros]:
assumes X: (X ----> x) net
and Y: (Y ----> y) net
shows ((λx. min (X x) (Y x)) ----> min x y) net

proof (rule order-tendstoI)
fix a
assume a < min x y
then show eventually (λx. min (X x) (Y x) < a) net
using order-tendstoD(1)(OF X, of a) order-tendstoD(1)(OF Y, of a)
by (auto simp: eventually-conj-iff)

next
fix a
assume min x y < a
then show eventually (λx. min (X x) (Y x) < a) net
using order-tendstoD(2)(OF X, of a) order-tendstoD(2)(OF Y, of a)
by (auto simp: min-less-iff-disj elim: eventually-mono)

qed

lemma (in order-topology)
assumes a < b
shows at-within-Icc-at-right: at a within {a..<b} = at-right a
and at-within-Icc-at-left: at b within {a..b} = at-left b
using order-tendstoD(1)(OF tendsto-ident-at assms, of {a..<})
using order-tendstoD(1)(OF tendsto-ident-at assms, of {..<b})
by (auto intro!: order-class.antisym filter-leI
simp: eventually-at-filter less-le
elim: eventually-elim2)

lemma (in order-topology) at-within-Icc-at: a < x ----> x < b ----> at x within {a..b} = at x
by (rule at-within-open-subset[where S={a..<b}]) auto

lemma (in t2-space) tendsto-unique:
assumes F ≠ bot
and (f ----> a) F
and (f ----> b) F
shows a = b

proof (rule ccontr)
**THEORY “Topological-Spaces”** 1779

**assume** \( a \neq b \)

**obtain** \( U V \) where \( \text{open} \ U \) \( \text{open} \ V \) \( a \in U \) \( b \in V \) \( U \cap V = {} \)

**using** \( \text{hausdorff} \ [OF \ \langle a \neq b \rangle] \) by fast

**have** eventually \( (\lambda x. \, f \ x \in U) \ F \)

**using** \( (f \longrightarrow a) \ F \) \( \langle \text{open} \ U \rangle \ (a \in U) \) by (rule topological-tendstoD)

**moreover**

**have** eventually \( (\lambda x. \, f \ x \in V) \ F \)

**using** \( (f \longrightarrow b) \ F \) \( \langle \text{open} \ V \rangle \ (b \in V) \) by (rule topological-tendstoD)

**ultimately**

**have** eventually \( (\lambda x. \, \text{False}) \ F \)

**proof** eventually-elim

**case** \( (\text{elim} \ x) \)

**then** have \( f \ x \in U \cap V \) by simp

**with** \( \langle U \cap V = {} \rangle \) **show** ?case by simp

**qed**

**with** \( \langle \neg \text{trivial-limit} \ F \rangle \) **show** \text{False} by (simp add: trivial-limit-def)

**qed**

**lemma** (in t2-space) tendsto-const-iff:

**fixes** \( a \) \( b \) :: 'a

**assumes** \( \neg \text{trivial-limit} \ F \)

**shows** \( ((\lambda x. \, a) \longrightarrow b) \ F \longleftrightarrow a = b \)

**by** (auto intro!: tendsto-unique [OF assms tendsto-const])

**lemma** Lim-in-closed-set:

**assumes** closed \( S \) eventually \( (\lambda x. \, f(x) \in S) \ F \) \( F \neq \text{bot} \) \( (f \longrightarrow l) \ F \)

**shows** \( l \in S \)

**proof** (rule ccontr)

**assume** \( l \notin S \)

**with** \( \langle \text{closed} \ S \rangle \) **have** open \( (- S) \) \( l \in - S \)

**by** (simp-all add: open-Compl)

**with** \( \text{assms}(4) \) **have** eventually \( (\lambda x. \, f \ x \in - S) \ F \)

**by** (rule topological-tendstoD)

**with** \( \text{assms}(2) \) **have** eventually \( (\lambda x. \, \text{False}) \ F \)

**by** (rule eventually-elim2) simp

**with** \( \text{assms}(3) \) **show** \text{False} by (simp add: eventually-False)

**qed**

**lemma** (in t3-space) nhds-closed:

**assumes** \( x \in A \) and \( \text{open} \ A \)

**shows** \( \exists A'. \ x \in A' \cap \text{closed} \ A' \cap A' \subseteq A \cap \text{eventually} \ (\lambda y. \ y \in A') \ (\text{nhds} \ x) \)

**proof** –

**from** \( \text{assms} \) **have** \( \exists U \ V. \ \text{open} \ U \cap \text{open} \ V \cap x \in U \cap - A \subseteq V \cap U \cap V = {} \)

**by** (intro t3-space) auto

**then** **obtain** \( U V \) where \( UV: \ \text{open} \ U \text{ open} \ V \ x \in U \ - A \subseteq V \ U \cap V = {} \)

**by** auto
have eventually \((\lambda y. \ y \in U) (nhds x)\)
   using \((\text{open } U) \quad \text{and} \quad (x \in U)\) by \((\text{intro eventually-nhds-in-open})\)

hence eventually \((\lambda y. \ y \in -V) (nhds x)\)
   by \((\text{eventually-elim} \quad \text{use } UV \text{ in auto})\)

with \(UV\) show \(?\text{thesis} \quad \text{by} \quad (\text{intro exI}[\text{of } -V])\) auto

qed

lemma \((\text{in order-topology})\) increasing-tendsto:
   assumes \(\text{bdd:} \quad \text{eventually } (\lambda n. \ f n \leq l) \ F\)
   and \(\text{en:} \quad \forall x. \ x < l \implies \text{eventually } (\lambda n. \ x < f n) \ F\)

   shows \((f \longrightarrow l) \ F\)
   using \(\text{assms by} \quad (\text{intro order-tendstoI}) \quad (\text{auto elim!: eventually-mono})\)

lemma \((\text{in order-topology})\) decreasing-tendsto:
   assumes \(\text{bdd:} \quad \text{eventually } (\lambda n. \ l \leq f n) \ F\)
   and \(\text{en:} \quad \forall x. \ l < x \implies \text{eventually } (\lambda n. \ f n < x) \ F\)

   shows \((f \longrightarrow l) \ F\)
   using \(\text{assms by} \quad (\text{intro order-tendstoI}) \quad (\text{auto elim!: eventually-mono})\)

lemma \((\text{in order-topology})\) tendsto-sandwich:
   assumes \(\text{ev:} \quad \text{eventually } (\lambda n. \ f n \leq g n) \ \text{net} \ \text{eventually } (\lambda n. \ g n \leq h n) \ \text{net}\)
   assumes \(\text{lim:} \quad (f \longrightarrow c) \ \text{net} \ (h \longrightarrow c) \ \text{net}\)

   shows \((g \longrightarrow c) \ \text{net}\)
   proof \((\text{rule order-tendstoI})\)
      fix \(a\)
      show \(a < c \implies \text{eventually } (\lambda x. \ a < g x) \ \text{net}\)
         using \(\text{order-tendstoD}[\text{OF lim}(1), \ \text{of } a] \ \text{ev by} \quad (\text{auto elim: eventually-elim2})\)

next

   fix \(a\)
   show \(c < a \implies \text{eventually } (\lambda x. \ g x < a) \ \text{net}\)
      using \(\text{order-tendstoD}[\text{OF lim}(2), \ \text{of } a] \ \text{ev by} \quad (\text{auto elim: eventually-elim2})\)

qed

lemma \((\text{in t1-space})\) limit-frequently-eq:
   assumes \(F \neq \text{bot}\)
   and \(\text{frequently } (\lambda x. \ f x = c) \ F\)
   and \((f \longrightarrow d) \ F\)

   shows \(d = c\)
   proof \((\text{rule ccontr})\)
      assume \(d \neq c\)
      from \(\text{t1-space}[\text{OF this}] \ \text{obtain } U \ \text{where} \ \text{open } U \ d \in U \ c \notin U\)
         by \text{blast}
      with \text{assms have} \ \text{eventually } (\lambda x. \ f x \in U) \ F\)

      unfolding tendsto-def by \text{blast}

      then \text{have} \ \text{eventually } (\lambda x. \ f x \neq c) \ F\)
         by \text{eventually-elim} \quad (\text{insert } c \notin U, \ \text{blast})
      with \text{assms(2) show False}

      unfolding frequently-def by \text{contradiction}

qed
lemma (in t1-space) tendsto-imp-eventually-ne:
  assumes \((f \rightarrow c) \, F \, c \neq c'\)
  shows eventually \((\lambda z. f z \neq c') \, F\)
proof (cases \(F=\text{bot}\))
  case True
  thus \(\text{thesis by auto}\)
next
  case False
  show \(\text{thesis}\)
  proof (rule ccontr)
    assume \(\neg\) eventually \((\lambda z. f z \neq c') \, F\)
    then have frequently \((\lambda z. f z = c') \, F\)
    by (simp add: frequently-def)
    from limit-frequently-eq[OF False this \((f \rightarrow c) \, F\)] and \((c \neq c')\) show False
    by contradiction
  qed
qed

lemma (in linorder-topology) tendsto-le:
  assumes \(F; \neg\) trivial-limit \(F\)
  and \(x: (f \rightarrow x) \, F\)
  and \(y: (g \rightarrow y) \, F\)
  and ev: eventually \((\lambda x. g x \leq f x) \, F\)
  shows \(y \leq x\)
proof (rule ccontr)
  assume \(\neg\) \(y \leq x\)
  with less-separate[of \(x \, y\)] obtain \(a \, b\) where \(xy: x < a \, b < y \{..a\} \cap \{b<..\}\) = {} 
  by (auto simp: not-le)
  then have eventually \((\lambda x. f x < a) \, F\) eventually \((\lambda x. b < g x) \, F\)
  using \(x \, y\) by (auto intro: order-tendstoD)
  with ev have eventually \((\lambda x. \text{False}) \, F\)
  by eventually-elim (insert \(xy\), fastforce)
  with \(F\) show False
  by (simp add: eventually-False)
qed

lemma (in linorder-topology) tendsto-lowerbound:
  assumes \(x: (f \rightarrow x) \, F\)
  and ev: eventually \((\lambda i. a \leq f i) \, F\)
  and \(F; \neg\) trivial-limit \(F\)
  shows \(a \leq x\)
using \(F \, x\) tendsto-const ev by (rule tendsto-le)

lemma (in linorder-topology) tendsto-upperbound:
  assumes \(x: (f \rightarrow x) \, F\)
  and ev: eventually \((\lambda i. a \geq f i) \, F\)
  and \(F; \neg\) trivial-limit \(F\)
THEORY “Topological-Spaces”

shows $a \geq x$
by (rule tendsto-le [OF F tendsto-const x ev])

lemma filterlim-at-within-not-equal:
fixes $f :: 'a \Rightarrow 'b::t2-space$
assumes filterlim $f$ (at $a$ within $s$) $F$
shows eventually $(\lambda w. f w \in s \land f w \neq b) F$
proof (cases $a=b$)
case True
then show ?thesis using assms by (simp add: filterlim-at)
next
case False
from hausdorff[OF this] obtain $U V$ where $UV$:
open $U$ open $V$
$a \in U$ $b \in V$
$U \cap V = \{\}$
by auto
have $(f \longrightarrow a) F$ using assms filterlim-at by auto
then have $\forall F. x \in F. f x \in s \land f x \neq a$ using assms filterlim-at by auto
ultimately show ?thesis
apply eventually-elim using $UV$ by auto
qed

96.5.4 Rules about Lim

lemma tendsto-Lim: $\neg$ trivial-limit net $\longrightarrow$ $(f \longrightarrow l)$ net $\longrightarrow$ Lim net $f = l$
unfolding Lim-def using tendsto-unique [of net $f$] by auto

lemma Lim-ident-at: $\neg$ trivial-limit (at $x$ within $s$) $\longrightarrow$ Lim (at $x$ within $s$) $(\lambda x. x) = x$
by (rule tendsto-Lim[OF - tendsto-ident-at]) auto

lemma eventually-Lim-ident-at:
$(\forall F \ y \ in \ at \ x \ within \ X. \ P \ (\text{Lim} \ (at \ x \ within \ X) \ (\lambda x. x)) \ y) \longleftrightarrow$
$(\forall F \ y \ in \ at \ x \ within \ X. \ P \ x \ y) \ for \ x::'a::t2-space$
by (cases at $x$ within $X = \text{bot}$) (auto simp: Lim-ident-at)

lemma filterlim-at-bot-at-right:
fixes $f :: 'a::linorder-topology \Rightarrow 'b::linorder$
assumes mono: $\forall x \ y. \ Q \ x \Longrightarrow \ Q \ y \Longrightarrow \ x \leq y \Longrightarrow f \ x \leq f \ y$
and bij: $\forall x. \ P \ x \Longrightarrow f \ (g \ x) = x \ \wedge x. \ P \ x \Longrightarrow Q \ (g \ x)$
and $Q$: eventually $Q$ (at-right $a$)
and bound: $\forall b. \ Q \ b \Longrightarrow a < b$
and $P$: eventually $P$ at-bot
shows filterlim $f$ at-bot (at-right $a$)
proof -
from $P$ obtain $x$ where $x: \forall y. \ y \leq x \Longrightarrow P \ y$
unfolding eventually-at-bot-linorder by auto
show ?thesis
proof (intro filterlim-at-bot-le[THEN iffD2] allI impI)
  fix z
  assume z ≤ x
  with x have P z by auto
  have eventually (λx. x ≤ g z) (at-right a)
    using bound[OF bij(2)[OF P z]]
    unfolding eventually-at-right[OF bound[OF bij(2)[OF P z]]]
    by (auto intro!: exI[of - g z])
  with Q show eventually (λx. f x ≤ z) (at-right a)
    by eventually-elim (metis bij ⟨P z⟩ mono)
qed

lemma filterlim-at-top-at-left:
  fixes f :: 'a::linorder-topology ⇒ 'b::linorder
  assumes mono: ∀x y. Q x ⇒ Q y ⇒ x ≤ y ⇒ f x ≤ f y
    and bij: ∀x. P x ⇒ f (g x) = x
    and Q: eventually Q (at-left a)
    and bound: ∀b. Q b ⇒ b < a
    and P: eventually P at-top
  shows filterlim f at-top (at-left a)
proof −
  from P obtain x where x: ∀y. x ≤ y ⇒ P y
    unfolding eventually-at-top-linorder by auto
  show ?thesis
    proof (intro filterlim-at-top-ge[THEN iffD2] allI impI)
      fix z
      assume x ≤ z
      with x have P z by auto
      have eventually (λx. g z ≤ x) (at-left a)
        using bound[OF bij(2)[OF P z]]
        unfolding eventually-at-left[OF bound[OF bij(2)[OF P z]]]
        by (auto intro!: exI[of - g z])
      with Q show eventually (λx. z ≤ f x) (at-left a)
        by eventually-elim (metis bij ⟨P z⟩ mono)
    qed
qed

lemma filterlim-split-at:
  filterlim f F (at-left x) ⟹ filterlim f F (at-right x) ⟹
  filterlim f F (at x)
  for x :: 'a::linorder-topology
  by (subst at-eq-sup-left-right) (rule filterlim-sup)

lemma filterlim-at-split:
  filterlim f F (at x) ⟷ filterlim f F (at-left x) ∧ filterlim f F (at-right x)
  for x :: 'a::linorder-topology
  by (subst at-eq-sup-left-right) (simp add: filterlim-def filtermap-sup)
lemma eventually-nhds-top:

fixes \( P : \text{bool} \) and \( b : 'a \)

assumes \( b < \text{top} \)

shows eventually \( P \) \(\iff\) \( (\exists b < \text{top}. (\forall z. b < z \rightarrow P z)) \)

unfolding eventually-nhds

proof safe

fix \( S : 'a \set \)

assume open \( S \) \(\in\) \( S \)

note open-left[OF this \langle b < \text{top} \rangle]

moreover assume \( \forall s \in S. P s \)

ultimately show \( \exists \ b < \text{top}. \forall z > b. P z \)

by (auto simp: subset-eq Ball-def)

next

fix \( b \)

assume \( b < \text{top} \) \(\forall z > b. P z \)

then show \( \exists S. \text{open } S \land \text{top} \land (\forall xa \in S. P xa) \)

by (intro exI[of - \{b <..\}]) auto

qed

lemma tendsto-at-within-iff-tendsto-nhds:

\((g \to g l) (at \ l \ within \ S) \iff (g \to g l) (\inf (nhds \ l) (\principal S))\)

unfolding tendsto-def eventually-at-filter eventually-inf-principal

by (intro ext all-cong imp-cong) (auto elim!: eventually-mono)

96.6 Limits on sequences

abbreviation (in topological-space)

\( LIMSEQ : [nat \Rightarrow \text{bool}] \)

where \( \xrightarrow{\text{sequentially}} L \equiv (X \to L) \)

abbreviation (in t2-space) \( \text{lim} : [nat \Rightarrow \text{a}] \Rightarrow \text{a} \)

where \( \text{lim} X \equiv \text{Lim sequentially } X \)

definition (in topological-space) \( \text{convergent} : [nat \Rightarrow \text{a}] \Rightarrow \text{bool} \)

where \( \text{convergent } X = (\exists L. X \to L) \)

lemma lim-def: \( \text{lim } X = (\text{THE } L. X \to L) \)

unfolding Lim-def ..

lemma lim-explicit:

\( f \to f0 \iff (\forall S. \text{open } S \to f0 \in S \to (\exists N. \forall n \geq N. f n \in S))\)

unfolding tendsto-def eventually-sequentially by auto

96.7 Monotone sequences and subsequences

Definition of monotonicity. The use of disjunction here complicates proofs considerably. One alternative is to add a Boolean argument to indicate the direction. Another is to develop the notions of increasing and decreasing
first.

**Definition** monoseq :: (nat ⇒ 'a::order) ⇒ bool

where monoseq X ←→ (∀ m. ∀ n≥m. X m ≤ X n) ∨ (∀ m. ∀ n≥m. X n ≤ X m)

**Abbreviation** incseq :: (nat ⇒ 'a::order) ⇒ bool

where incseq X ≡ mono X

**Lemma** incseq-def: incseq X ←→ (∀ m. ∀ n≥m. X n ≥ X m)

**Unfolding** mono-def ..

**Abbreviation** decseq :: (nat ⇒ 'a::order) ⇒ bool

where decseq X ≡ antimono X

**Lemma** decseq-def: decseq X ←→ (∀ m. ∀ n≥m. X n ≤ X m)

**Unfolding** antimono-def ..

96.7.1 Definition of subsequence.

**Lemma** strict-mono-leD: strict-mono r =⇒ m ≤ n =⇒ r m ≤ r n

by (erule (1) monoD [OF strict-mono-mono])

**Lemma** strict-mono-id: strict-mono id

by (simp add: strict-mono-def)

**Lemma** incseq-SucI: (∀ n. X n ≤ X (Suc n)) =⇒ incseq X

using lift-Suc-mono-le[of X] by (auto simp: incseq-def)

**Lemma** incseqD: incseq f =⇒ i ≤ j =⇒ f i ≤ f j

by (auto simp: incseq-def)

**Lemma** incseq-SucD: incseq A =⇒ A i ≤ A (Suc i)

using incseqD[of A i Suc i] by auto

**Lemma** incseq-Suc-iff: incseq f =⇒ (∀ n. f n ≤ f (Suc n))

by (auto intro: incseq-SucI dest: incseq-SucD)

**Lemma** incseq-const[simp, intro]: incseq (λx. k)

unfolding incseq-def by auto

**Lemma** decseq-SucI: (∀ n. X (Suc n) ≤ X n) =⇒ decseq X

using order.lift-Suc-mono-le[of X] by (auto simp: decseq-def)

**Lemma** decseqD: decseq f =⇒ i ≤ j =⇒ f j ≤ f i

by (auto simp: decseq-def)

**Lemma** decseq-SucD: decseq A =⇒ A (Suc i) ≤ A i

using decseqD[of A i Suc i] by auto
lemma \textit{decseq-Suc-iff}: \textit{decseq} f \longleftrightarrow (\forall n. f (\text{Suc} n) \leq f n) 
by (auto intro: decseq-SucI dest: decseq-SucD)

lemma \textit{decseq-const}: \textit{decseq} (\lambda x. k)
unfolding \textit{decseq-def} by auto

lemma \textit{monoseq-iff}: \textit{monoseq} X \longleftrightarrow \textit{incseq} X \lor \textit{decseq} X
unfolding \textit{monoseq-def incseq-def decseq-def} ..

lemma \textit{monoseq-Suc}: \textit{monoseq} X \longleftrightarrow (\forall n. X n \leq X (\text{Suc} n)) \lor (\forall n. X (\text{Suc} n) \leq X n)
unfolding \textit{monoseq-iff incseq-Suc-iff decseq-Suc-iff} ..

lemma \textit{monoseq-minus}:
fixes a :: \texttt{nat} \Rightarrow 'a::ordered-ab-group-add
assumes \textit{monoseq} a
shows \textit{monoseq} (\lambda n. - a n)
proof (cases \forall m. \forall n \geq m. a m \leq a n)
case True
then have \forall m. \forall n \geq m. - a n \leq - a m by auto
then show ?thesis by (rule monoI2)
next
case False
then have \forall m. \forall n \geq m. - a m \leq - a n
using (\textit{monoseq} a) [unfolded \textit{monoseq-def}] by auto
then show ?thesis by (rule monoI1)
qed

96.7.2 \textbf{Subsequence (alternative definition, (e.g. Hoskins)}

lemma \textit{strict-mono-Suc-iff}: \textit{strict-mono} f \longleftrightarrow (\forall n. f n < f (\text{Suc} n))
proof (intro iffI \textit{strict-mono1})
assume *: \forall n. f n < f (\text{Suc} n)
fix m n :: \texttt{nat} assume m < n
thus \textit{f} m < \textit{f} n
by (induction rule: less-Suc-induct) (use * in auto)
qed (auto simp: \textit{strict-mono-def})
lemma strict-mono-add: strict-mono \((\lambda n::'a::linordered-semidom. n + k)\) by (auto simp: strict-mono-def)

For any sequence, there is a monotonic subsequence.

lemma seq-monosub:
fixes s :: nat ⇒ 'a::linorder
shows \(\exists f. \text{strict-mono } f \land \text{monoseq } (\lambda n. \ (s \ (f \ n)))\)
proof (cases \(\forall n. \exists p>n. \forall m \geq p. \ s m \leq s p\))
  case True
  then have \(\exists f. \forall n. \ (\forall m \geq f n. \ s m \leq s (f n)) \land f n < f (Suc n)\)
    by (intro dependent-nat-choice) (auto simp: conj-commute)
  then obtain f :: nat ⇒ nat
    where f: strict-mono f and mono:
      \(\forall n m. \ f n \leq m \Longrightarrow s m \leq s (f n)\)
    by (auto simp: strict-mono-Suc-iff)
  then have incseq f
    unfolding strict-mono-Suc-iff incseq-Suc-iff by (auto intro: less-imp-le)
  then have monoseq (\(\lambda n. s (f n)\))
    by (auto simp add: incseq-def intro!: mono monoI2)
  with f show ?thesis
    by auto
next
  case False
  then obtain N where N: p > N ⇒ \exists m>p. s p < s m for p
    by (force simp: not-le le-less)
  have \(\exists f. \forall n. \ N < f n \land f n < f (Suc n) \land s (f n) \leq s (f (Suc n))\)
    (intro dependent-nat-choice)
  fix x
  assume N < x with N[of x]
  show \(\exists y>N. x < y \land s x \leq s y\)
    by (auto intro: less-trans)
  qed auto
then show ?thesis
  by (auto simp: monoseq-iff incseq-Suc-iff strict-mono-Suc-iff)
qed

lemma seq-suble:
assumes sf: strict-mono (f :: nat ⇒ nat)
shows n ≤ f n
proof (induct n)
  case 0
  show ?case by simp
next
  case (Suc n)
  with sf [unfolded strict-mono-Suc-iff, rule-format, of n] have n < f (Suc n)
    by arith
  then show ?case by arith
qed
lemma eventually-subseq:
strict-mono $r \implies \text{eventually } P$ sequentially $\implies \text{eventually } (\lambda n. P (r n))$ sequentially
unfolding eventually-sequentially by (metis seq-suble le-trans)

lemma not-eventually-sequentiallyD:
assumes $\neg \text{eventually } P$ sequentially
shows $\exists r :: \text{nat} \Rightarrow \text{nat}. \text{strict-mono } r \land (\forall n. r P (r n))$
proof
from assms have $\forall n. \exists m \geq n. \neg m$
unfolding eventually-sequentially by (simp add: not-less)
then obtain $r$ where $\forall n. r n \geq n \land n. \neg P (r n)$
by (auto simp: choice-iff)
then show $?thesis$
by (auto intro: exI [of $\lambda n. r ((Suc \circ r) ^ \omega Suc n) 0)]$
simp: less-eq-Suc-le strict-mono-Suc-iff)
qed

lemma sequentially-offset:
assumes eventually $\lambda i. P i$ sequentially
shows eventually $\lambda i. P (i + k)$ sequentially
using assms by (rule eventually-sequentially-seg [THEN iffD2])

lemma seq-offset-neg:
$(f \longrightarrow l)$ sequentially $\implies ((\lambda i. f (i - k)) \longrightarrow l)$ sequentially
apply (erule filterlim-compose)
apply (simp add: filterlim-def le-sequentially eventually-filtermap eventually-sequentially, arith)
done

lemma filterlim-subseq: strict-mono $f \implies \text{filterlim } f$ sequentially sequentially
unfolding filterlim-iff by (metis eventually-subseq)

lemma strict-mono-o: strict-mono $r \implies \text{strict-mono } s \implies \text{strict-mono } (r \circ s)$
unfolding strict-mono-def by simp

lemma strict-mono-compose: strict-mono $r \implies \text{strict-mono } s \implies \text{strict-mono}$
$(\lambda x. r (s x))$
using strict-mono-o[of $r$ $s$] by (simp add: o-def)

lemma incseq-imp-monoseq: incseq $X \implies \text{monoseq } X$
by (simp add: incseq-def monoseq-def)

lemma decseq-imp-monoseq: decseq $X \implies \text{monoseq } X$
by (simp add: decseq-def monoseq-def)

lemma decseq-eq-incseq: decseq $X = \text{incseq } (\lambda n. - X n)$
for $X :: \text{nat} \Rightarrow \text{a::ordered-ab-group-add}$
by (simp add: decseq-def incseq-def)
lemma INT-decseq-offset:
  assumes decseq F
  shows \((\bigcap i. F i) = (\bigcap i\in\{n..\}. F i)\)
proof safe
  fix x i
  assume x: x \in (\bigcap i\in\{n..\}. F i)
  show x \in F i
  proof cases
    from x have x \in F n by auto
    also assume i \leq n
    with \langle decseq F \rangle have F n \subseteq F i
    unfolding decseq-def by simp
    finally show \?thesis .
  qed (insert x, simp)
qed auto

lemma LIMSEQ-const-iff: \((\lambda n. k) \rightarrow l \iff k = l\)
for k l :: 'a::t2-space
using trivial-limit-sequentially by (rule tendsto-const-iff)

lemma LIMSEQ-SUP: incseq X \Longrightarrow X \rightarrow (SUP i. X i :: 'a::complete-linorder,linorder-topology)
by (intro increasing-tendsto)
  (auto simp: SUP-upper less-SUP-iff incseq-def eventually-sequentially intro: less-le-trans)

lemma LIMSEQ-INF: decseq X \Longrightarrow X \rightarrow (INF i. X i :: 'a::complete-linorder,linorder-topology)
by (intro decreasing-tendsto)
  (auto simp: INF-lower INF-less-iff decseq-def eventually-sequentially intro: le-less-trans)

lemma LIMSEQ-ignore-initial-segment: f \rightarrow l \Longrightarrow (\lambda n. f (n + k)) \rightarrow a
  unfolding tendsto-def by (subst eventually-sequentially-seg[where k=k])

lemma LIMSEQ-offset: (\lambda n. f (n + k)) \rightarrow a \Longrightarrow f \rightarrow a
unfolding tendsto-def
by (subst (asm) eventually-sequentially-seg[where k=k])

lemma LIMSEQ-Suc: f \rightarrow l \Longrightarrow (\lambda n. f (Suc n)) \rightarrow l
by (drule LIMSEQ-ignore-initial-segment [where k=Suc 0]) simp

lemma LIMSEQ-imp-Suc: (\lambda n. f (Suc n)) \rightarrow l \Longrightarrow f \rightarrow l
by (rule LIMSEQ-offset [where k=Suc 0]) simp

lemma LIMSEQ-Suc-iff: (\lambda n. f (Suc n)) \rightarrow l \iff f \rightarrow l
by (rule filterlim-sequentially-Suc)

lemma LIMSEQ-lessThan-iff-atMost:
shows (\lambda n. f \{..<n\}) \rightarrow x \iff (\lambda n. f \{..n\}) \rightarrow x
apply (subst LIMSEQ-Suc-iff [symmetric])
apply (simp only: lessThan-Suc-atMost)
done

lemma LIMSEQ-unique: X ----> a ==> X ----> b ==> a = b
for a b :: 'a::t2-space
using trivial-limit-sequentially by (rule tendsto-unique)

lemma LIMSEQ-le-const: X ----> x =
⇒ \exists N. \forall n\geq N. X n \leq Y n
⇒ x \leq y
for x y :: 'a::linorder-topology
using tendsto-le[of sequentially Y y X x] by (simp add: eventually-sequentially)

lemma LIMSEQ-le-const2: X ----> x =
⇒ \exists N. \forall n\geq N. X n \leq a
⇒ x \leq a
for a x :: 'a::linorder-topology
by (rule LIMSEQ-le-const2)

lemma Lim-bounded: f ----> l =
⇒ \forall n\geq M. f n \leq C
⇒ l \leq C
for l :: 'a::linorder-topology
by (intro LIMSEQ-le-const2)

lemma Lim-bounded2:
fixes f :: nat \Rightarrow 'a::linorder-topology
assumes lim:f ----> l and ge: \forall n\geq N. f n \geq C
shows l \geq C
using ge
by (intro tendsto-le[OF trivial-limit-sequentially lim tendsto-const])
(auto simp: eventually-sequentially)

lemma lim-mono:
fixes X Y :: nat \Rightarrow 'a::linorder-topology
assumes \forall n\leq n. N \leq n \Longrightarrow X n \leq Y n
and X ----> x
and Y ----> y
shows x \leq y
using assms(1) by (intro LIMSEQ-le[of assms(2,3)])
(auto)

lemma Sup-lim:
fixes a :: 'a::{complete-linorder,linorder-topology}
assumes \forall n. b n \in s
and b ----> a
shows a \leq Sup s
by (metis Lim-bounded assms complete-lattice-class.Sup-upper)

lemma Inf-lim:
fixes a :: 'a::{complete-linorder,linorder-topology}
assumes $\forall n. \ b \ n \in s$
and $b \longrightarrow a$

shows $\Inf s \leq a$

by (metis Lim-bounded2 assms complete-lattice-class.Inf_lower)

**lemma** $\text{SUP-Lim}$:
fixes $X :: \text{nat} \Rightarrow 'a::\{\text{complete-linorder},\text{linorder-topology}\}$$\text{inc} X$
and $l: X \longrightarrow l$
shows $(\SUP n \cdot X n) = l$
using $\text{LIMSEQ-SUP}[\text{OF inc}] \ tendsto\text{-unique}[\text{OF trivial-limit-sequentially l}]$
by simp

**lemma** $\text{INF-Lim}$:
fixes $X :: \text{nat} \Rightarrow 'a::\{\text{complete-linorder},\text{linorder-topology}\}$$\text{dec} X$
and $l: X \longrightarrow l$
shows $(\INF n \cdot X n) = l$
using $\text{LIMSEQ-INF}[\text{OF dec}] \ tendsto\text{-unique}[\text{OF trivial-limit-sequentially l}]$
by simp

**lemma** $\text{convergentD}$: convergent $X \Longrightarrow \exists L. X \longrightarrow L$
by (simp add: convergent-def)

**lemma** $\text{convergentI}$: $X \longrightarrow L \Longrightarrow$ convergent $X$
by (auto simp add: convergent-def)

**lemma** $\text{convergent-LIMSEQ-iff}$: convergent $X \iff X \longrightarrow \lim X$
by (auto intro: theI LIMSEQ-unique simp add: convergent-def lim-def)

**lemma** $\text{convergent-const}$: convergent $(\lambda n. c)$
by (rule convergentI) (rule tendsto-const)

**lemma** $\text{monoseq-le}$:
monoseq $a \Longrightarrow a \longrightarrow x \Longrightarrow$
$(\forall n. \ a \ n \leq x) \land (\forall m. \ \forall n \geq m. \ a \ m \leq a \ n) \lor$
$(\forall n. \ x \leq a \ n) \land (\forall m. \ \forall n \geq m. \ a \ n \leq a \ m)$
for $x :: 'a::\text{linorder-topology}$
by (metis LIMSEQ-le-const LIMSEQ-le-const2 decseq-def incseq-def monoseq-iff)

**lemma** $\text{LIMSEQ-subseq-LIMSEQ}$: $X \longrightarrow L \Longrightarrow$ strict-mono $f \Longrightarrow (X \circ f) \longrightarrow L$
unfolding comp-def by (rule filterlim-compose [of X, OF - filterlim-subseq])

**lemma** $\text{convergent-subseq-convergent}$: convergent $X \Longrightarrow$ strict-mono $f \Longrightarrow$ convergent $(X \circ f)$
by (auto simp: convergent-def intro: LIMSEQ-subseq-LIMSEQ)

**lemma** $\text{limI}$: $X \longrightarrow L \Longrightarrow \lim X = L$
by (rule tendsto-Lim) (rule trivial-limit-sequentially)

lemma lim-le: convergent f \implies (\forall n. f n \leq x) \implies \lim f \leq x
  for x :: 'a:linorder-topology
  using LIMSEQ-le-const2[of f lim f x] by (simp add: convergent-LIMSEQ-iff)

lemma lim-const [simp]: \lim (\lambda m. a) = a
  by (simp add: limI)

96.7.3 Increasing and Decreasing Series

lemma incseq-le: incseq X \implies X \longrightarrow L \implies X n \leq L
  for L :: 'a:linorder-topology
  by (metis incseq-def LIMSEQ-le-const)

lemma decseq-ge: decseq X \implies X \longrightarrow L \implies L \leq X n
  for L :: 'a:linorder-topology
  by (metis decseq-def LIMSEQ-le-const2)

96.8 First countable topologies

class first-countable-topology = topological-space +
  assumes first-countable-basis:
    \exists A :: nat \Rightarrow 'a set. (\forall i. x \in A i \land open (A i)) \land (\forall S. open S \land x \in S \longrightarrow (\exists i. A i \subseteq S))

lemma (in first-countable-topology) countable-basis-at-decseq:
  obtains A :: nat \Rightarrow 'a set where
    \forall i. open (A i) \land i. x \in (A i)
    \forall S. open S \longrightarrow x \in S \longrightarrow eventually (\lambda i. A i \subseteq S) sequentially
  proof atomize-elim
    from first-countable-basis[of x] obtain A :: nat \Rightarrow 'a set where
      nhds: \forall i. open (A i) \land i. x \in A i
      and incl: \forall S. open S \longrightarrow x \in S \longrightarrow (\exists i. A i \subseteq S)
      by auto
    define F where F n = (\bigcap_{i\leq n} A i) for n
    show \exists A. (\forall i. open (A i)) \land (\forall i. x \in A i) \land
      (\forall S. open S \longrightarrow x \in S \longrightarrow eventually (\lambda i. A i \subseteq S) sequentially)
    proof (safe intro!: exI[of - F])
      fix i
      show open (F i)
        using nhds(1) by (auto simp: F-def)
      show x \in F i
        using nhds(2) by (auto simp: F-def)
    next
    fix S
    assume open S x \in S
    from incl[OF this] obtain i where F i \subseteq S
      unfolding F-def by auto
    moreover have \bigcap_{i \leq j} F j \subseteq F i
ultimately show eventually ($\lambda i. F i \subseteq S$) sequentially
by (auto simp: eventually-sequentially)

lemma (in first-countable-topology) nhds-countable:
obtains $X :: nat \Rightarrow 'a set$
where decseq $X \cap n. open (X n) \cap n. x \in X n \text{ nhds } x = (\text{INF } n. \text{ principal } (X n))$
proof –
from first-countable-basis obtain $A :: nat \Rightarrow 'a set$
where $\ast: \lambda n. x \in A n \cap n. open (A n) \cap S. open S \Longrightarrow x \in S \Longrightarrow \exists i. A i \subseteq S$
by metis
show thesis
proof
show decseq ($\lambda n. (\cap i \leq n. A i)$
by (simp add: antimono-iff-le-Suc atMost-Suc)
show $x \in (\cap i \leq n. A i) \cap n. open (\cap i \leq n. A i)$ for $n$
using $\ast$ by auto
show $\text{nhds } x = (\text{INF } n. \text{ principal } (\cap i \leq n. A i))$
using $\ast$
unfolding nhds-def
apply –
apply (rule INF-eq)
apply simp-all
apply fastforce
apply (intro exI [of - \cap i \leq n. A i for n] conjI open-INT)
apply auto
done
qed

lemma (in first-countable-topology) countable-basis:
obtains $A :: nat \Rightarrow 'a set$ where
$\lambda i. open (A i) \lambda i. x \in A i$
$\lambda F. (\forall n. F n \in A n) \Longrightarrow F \longrightarrow x$
proof atomize-elim
obtain $A :: nat \Rightarrow 'a set$ where $\ast:$
$\lambda i. open (A i)$
$\lambda i. x \in A i$
$\lambda S. open S \Longrightarrow x \in S \Longrightarrow \text{eventually } (\lambda i. A i \subseteq S) \text{ sequentially}$
by (rule countable-basis-at-decseq) blast
have eventually ($\lambda n. F n \in S$) sequentially
if $\forall n. F n \in A n \text{ open } S x \in S \text{ for } F S$
using $\ast(3)[of S]$ that by (auto elim: eventually-mono simp: subset-eq)
with $\ast$ show $\exists A. (\forall i. open (A i)) \land (\forall i. x \in A i) \land (\forall F. (\forall n. F n \in A n) 
\Longrightarrow F \longrightarrow x)$
by (intro exI[of - A]) (auto simp: tendsto_def)

qed

lemma (in first-countable-topology) sequentially-imp-eventually-nhds-within:
assumes \( \forall f. (\forall n. f n \in s) \land f \longrightarrow a \longrightarrow (\lambda n. P (f n)) \) sequentially
shows eventually \( P \ (\inf \ (nhds \ a) \ (principal \ s)) \)

proof (rule ccontr)
obtain \( A :: nat \Rightarrow 'a \) set where *:
\( \land i. \ \text{open} \ (A i) \)
\( \land i. \ a \in A i \)
\( \land F. \ \forall n. F n \in A n \Longrightarrow F \longrightarrow a \)
by (rule countable-basis) blast
assume \( \neg \ ?thesis \)
with * have \( \exists F. \ \forall n. F n \in s \land F n \in A n \land \neg P (F n) \)
unfolding eventually-inf-principal eventually-nhds
by (intro choice) fastforce
then obtain \( F \) where \( \forall n. F n \in s \) and \( \forall n. F n \in A n \) and \( F' ; \ \forall n. \neg P (F n) \)
by blast
with * have \( F \longrightarrow a \)
by auto
then have eventually \( (\lambda n. P (F n)) \) sequentially
using assms \( F \) by simp
then show False
by (simp add: \( F' \))

qed

lemma (in first-countable-topology) eventually-nhds-iff-sequentially:
\( \text{eventually} \ P \ (nhds \ a) \longleftrightarrow \forall x \in S. x \in s \longrightarrow P x \)
using eventually-nhds-within-iff-sequentially[of \( P \ a \ \text{UNIV} \)] by simp

lemma Inf-as-limit:
fixes \( A ::'a::\{\text{linorder-topology, first-countable-topology, complete-linorder}\} \) set
assumes \( A \neq \{\} \)
shows \( \exists u. (\forall n. u < n \in A) \land u \longrightarrow \inf A \)
proof (cases Inf A \in A)
  case True
  show \(?thesis \)
    by (rule \( \exists ! u. \text{of - } \lambda n. \text{Inf A} \), auto simp add: True)
next
case False
obtain \( y \) where \( y \in A \) using assms by auto
then have \( \text{Inf A} < y \) using False Inf-lower less-le by auto
obtain \( F \) where \( \forall i. \text{open} (F i) \land i. \text{Inf A} \in F i \)
  \( \land u. (\forall n. u < n \in F n) \longrightarrow u \longrightarrow \text{Inf A} \)
  by (metis first-countable-topology-class.countable-basis)
define \( u \) where \( u = (\lambda n. \text{SOME} z. z \in F n \land z \in A) \)
have \( \exists z. z \in U \land z \in A \) if \( \text{Inf A} \in U \) open \( U \) for \( U \)
proof ¬
obtain \( b \) where \( b > \text{Inf A} \) \( \{ \text{Inf A} ..<b \} \subseteq U \)
  using open-right[\( OF \) open \( U \)] \( \lambda n. \text{Inf A} \in U \) \( \{ \text{Inf A} \in \text{Inf} \} \)
  by auto
obtain \( z \) where \( z < b \) \( z \in A \)
  using \( \text{Inf A} < b \) Inf-less-iff by auto
then have \( \exists \{ \text{Inf A} ..<b \} \)
  by (simp add: Inf-lower)
then show \(?thesis \) using \( \{ \text{Inf A} ..<b \} \subseteq U \) by auto
qed
then have \( \ast \) \( \ast : u n \in F n \land u n \in A \) for \( n \)
  using \( \text{Inf A} \in F n \) \( \text{open} (F n) \) unfolding u-def by (metis (no-types, lifting) some-ex)
then have \( u \longrightarrow \text{Inf A} \) using \( F(\ast) \) by simp
then show \(?thesis \) using \( \ast \) by auto
qed

lemma tendsto-at-iff-sequentially:
\( (f \longrightarrow a) \) \((at \ x \ within \ s) \longleftrightarrow (\forall X. (\forall i. X i \in s - \{ x \} ) \longrightarrow X \longrightarrow x \longrightarrow ((f \circ X) \longrightarrow a)) \)
for \( f : 'a::\text{first-countable-topology} \Rightarrow - \)
unfolding filterlim-def[of - nhds a] le-filter-def eventually-filtermap
at-within-def eventually-nhds-within-iff-sequentially comp-def
by metis

lemma approx-from-above-dense-linorder:
fixes \( x :: 'a::\{\text{dense-linorder}, \text{linorder-topology}, \text{first-countable-topology}\} \)
assumes \( x < y \)
shows \( \exists u. (\forall n. u > n \land x) \land (u \longrightarrow x) \)
proof ¬
obtain \( A :: \text{nat} \Rightarrow \lambda s. \{ \forall i. \text{open} (A i) \land i. x \in A i \}
  \land F. (\forall n. F n \in A n) \longrightarrow F \longrightarrow x \)
  by (metis first-countable-topology-class.countable-basis)
define \( u \) where \( u = (\lambda n. \text{SOME} z. z \in A n \land z > x) \)
have \( \exists z. z \in U \land x < z \) if \( x \in U \) open \( U \) for \( U \)
using open-right[OF open U ⟨x ∈ U⟩ ⟨x < y⟩]
by (meson atLeastLessThan-iff dense less-imp-le subset-eq)
then have *: u ∈ A n ∧ x < u n for n
using ⟨x ∈ A n⟩ ⟨open (A n)⟩ unfolding u-def by (metis (no-types, lifting) someI-ex)
then have u ───→ x using A(3) by simp
then show ?thesis using * by auto
qed

lemma approx-from-below-dense-linorder:
fixes x::′a::{dense-linorder, linorder-topology, first-countable-topology}
assumes x > y
shows ∃ u. (∀ n. u n < x) ∧ (u ───→ x)
proof –
obtain A :: nat ⇒ ′a set where A: i. open (A i) ∧ i. x ∈ A i ∧ F. (∀ n. F n ∈ A n) ⇒ F ───→ x
by (metis first-countable-topology-class.countable-basis)
define u where u = (λ n. SOME z. z ∈ A n ∧ z < x)
have ∃ z. z ∈ U ∧ z < x if x ∈ U open U for U
using open-left[OF open U ⟨x ∈ U⟩ ⟨x > y⟩]
by (meson dense greaterThanAtMost-iff less-imp-le subset-eq)
then have *: u ∈ A n ∧ u n < x for n
using ⟨x ∈ A n⟩ ⟨open (A n)⟩ unfolding u-def by (metis (no-types, lifting) someI-ex)
then have u ┼ x using A(3) by simp
then show ?thesis using * by auto
qed

96.9 Function limit at a point
abbreviation LIM :: (′a::topological-space ⇒ ′b::topological-space) ⇒ ′a ⇒ ′b ⇒ bool
((λ x. L) / (λ x. L) / (− x. L)[60, 0, 60] 60)
where f − a→ L ≡ (f ┼ l)(at a)

lemma tendsto-within-open: a ∈ S ⇒ open S ⇒ (f ┼ l)(at a within S)
←− (f − a→ l)
by (simp add: tendsto-def at-within-open[where S = S])

lemma tendsto-within-open-NO-MATCH:
a ∈ S ⇒ NO-MATCH UNIV S ⇒ open S ⇒ (f ┼ l)(at a within S) ←− (f − a→ l)(at a)
for f :: ′a::topological-space ⇒ ′b::topological-space
using tendsto-within-open by blast

lemma LIM-const-not-eq[tendsto-intras]: k ≠ L ⇒ (λ x. k) − a→ L
for a :: ′a::perfect-space and k L :: ′b::t2-space
by (simp add: tendsto-const-iff)
lemmas LIM-not-zero = LIM-const-not-eq [where \( L = 0 \)]

lemma LIM-const-eq: \( (\lambda x. k) \to L \) \( \to k = L \)

for a :: 'a::perfect-space and k L :: 'b::t2-space
by (simp add: tendsto-const-iff)

lemma LIM-unique: \( f \to L = M \) \( \to \) \( f \to L = M \)

for a :: 'a::perfect-space and L M :: 'b::t2-space
using at-neq-bot by (rule tendsto-unique)

Limits are equal for functions equal except at limit point.

lemma LIM-equal: \( \forall x. x \neq a \to f x = g x \) \( \to \) \( f \to l = M \)
by (simp add: tendsto-def eventually-at-topological)

lemma tendsto-cong: \( a = b \) \( \to \) \( \lambda x. x \neq b \to f x = g x \) \( \to l = m \)
by (simp add: LIM-unique)

lemma tendsto-compose: \( g \to l \to m \) \( \to \) \( (f \to l) \to g l \to m \)
by (rule filterlim-compose[of g])

lemma tendsto-compose-eventually:
assumes \( f \to y \) \( \to \) \( g \to z \) \( \to \) \( \lambda x. f x \neq b \) \( \to \) \( c \)
shows \( \lambda x. g(fx) \) \( \to c \)
using assms(2,1,3) by (rule tendsto-compose-eventually)

lemma tendsto-compose-filtermap: \( (g \circ f) \to T \) \( \to \) \( g \to T \) \( (filtermap f F) \)
by (simp add: filterlim-def filtermap-filtermap comp-def)

lemma tendsto-compose-at:
assumes \( f \to y \) \( \to \) \( g \to z \) \( \to \) \( w = y \to g y = z \)
shows \( (g \circ f) \to z \)

96.9.1 Relation of $LIM$ and $LIMSEQ$

**Lemma (in first-countable-topology) sequentially-imp-eventually-within:**

$$(\forall f. (\forall n. f n \in S \land f n \neq a) \land f \longrightarrow a \longrightarrow \text{eventually} (\lambda n. P (f n)) \text{ sequentially}) \Longrightarrow$$

eventually $P$ (at $a$ within $s$)

**Unfolding at-within-def**

by (intro sequentially-imp-eventually-within-nhds-within) auto

**Lemma (in first-countable-topology) sequentially-imp-eventually-at:**

$$(\forall f. (\forall n. f n \neq a) \land f \longrightarrow a \longrightarrow \text{eventually} (\lambda n. P (f n)) \text{ sequentially}) \Longrightarrow$$

eventually $P$ (at $a$)

using sequentially-imp-eventually-within [where $s=\text{UNIV}$] by simp

**Lemma $\text{LIMSEQ-SEQ}{}$-conv1:**

fixes $f : 'a::\text{topological-space} \Rightarrow 'b::\text{topological-space}$

assumes $f : f \longrightarrow a \longrightarrow l$

shows $\forall S. (\forall n. S n \neq a) \land S \longrightarrow a \longrightarrow (\lambda n. f (S n)) \longrightarrow l$

using tendsto-compose-eventually [OF $f$, where $F=\text{sequentially}$] by simp

**Lemma $\text{LIMSEQ-SEQ}{}$-conv2:**

fixes $f : 'a::\text{first-countable-topology} \Rightarrow 'b::\text{topological-space}$

assumes $\forall S. (\forall n. S n \neq a) \land S \longrightarrow a \longrightarrow (\lambda n. f (S n)) \longrightarrow l$

shows $f \longrightarrow a \longrightarrow l$

using assms unfolding tendsto-def [where $l=\text{l}$] by (simp add: sequentially-imp-eventually-at)

**Lemma $\text{LIMSEQ-SEQ}{}$-conv: $$(\forall S. (\forall n. S n \neq a) \land S \longrightarrow a \longrightarrow (\lambda n. X (S n)) \longrightarrow L) \longleftrightarrow X -a \longrightarrow L$$**

for $a : 'a::\text{first-countable-topology}$ and $L : 'b::\text{topological-space}$

using $\text{LIMSEQ-SEQ}{}$-conv2 $\text{LIMSEQ-SEQ}{}$-conv1 ..

**Lemma sequentially-imp-eventually-at-left:**

fixes $a : 'a::\{\text{linorder-topology, first-countable-topology}\}$

assumes $b \text{ simp}; b < a$

and $*: \forall f. (\forall n. b < f n) \Longrightarrow (\forall n. f n < a) \Longrightarrow \text{incseq} f \Longrightarrow f \longrightarrow a \Longrightarrow$

eventually $(\lambda n. P (f n))$ sequentially

shows eventually $P$ (at-left $a$)

**Proof** (safe intro! sequentially-imp-eventually-within)

fix $X$
assumption $X: \forall n \cdot X n \in \{..< a\} \land X n \neq a \longrightarrow a$

show eventually $(\lambda n. P (X n))$

proof (rule econtr)

assumption neg: $\neg$thesis

have $\exists s. \forall n. (\neg P (X (s n)) \land b < X (s n)) \land (X (s n) \leq X (s (Suc n)) \land Suc (s n) \leq s (Suc n))$

(is $\exists s. (P s)$)

proof (rule dependent-nat-choice)

have $\neg$ eventually $(\lambda n. b < X n \longrightarrow P (X n))$

by (intro not-eventually-impl neg order-tendstoD) [OF $X (2)$ $b$]

then show $\exists x. b < X x$ by (auto dest: not-eventuallyD)

next

fix $x n$

have $\neg$ eventually $(\lambda n. Suc x \leq n \longrightarrow b < X n \longrightarrow X x < X n \longrightarrow P (X n))$

using $X$

by (intro not-eventually-impl order-tendstoD) [OF $X (2)$] eventually-ge-at-top auto

then show $\exists n. (\neg P (X n) \land b < X n) \land (X x \leq X n \land Suc x \leq n)$

by (auto dest: not-eventuallyD)

qed

then obtain $s$ where $P s ..$

with $X$ have $b < X (s n)$

and $X (s n) < a$

and incseq $(\lambda n. X (s n))$

and $(\lambda n. X (s n)) \longrightarrow a$

and $\neg P (X (s n))$

for $n$

by (auto simp: strict-mono-Suc-iff Suc-le-eq incseq-Suc-iff intro!: LIMSEQ-subseq-LIMSEQ [OF $X \longrightarrow a$, unfolded comp_def])

from $\{OF this(1,2,3,4)\}$ this(5) show False

by auto

qed

lemma tendsto-at-left-sequentially:

fixes $a \ b :: \langle \text{linorder-topology,first-countable-topology} \rangle$

assumes $b < a$

assumes *: $(\forall n. S n < a) \implies (\forall n. b < S n) \implies \text{incseq } S \implies S \longrightarrow a$

$(\forall n. X (S n)) \longrightarrow L$

shows $(X \longrightarrow L) (\text{at-left } a)$

using assms by (simp add: tendsto_def [where l=L] sequentially-imp-eventually-at-left)

lemma sequentially-imp-eventually-at-right:

fixes $a \ b :: \langle \text{linorder-topology,first-countable-topology} \rangle$

assumes $b[simp]: a < b$

assumes *: $(\forall n. a < f n) \implies (\forall n. f n < b) \implies \text{decseq } f \implies f \longrightarrow a$

qed
THEORY "Topological-Spaces"

eventually (λn. P (f n)) sequentially
shows eventually P (at-right a)
proof (safe intro!: sequentially-imp-eventually-within)
  fix X
  assume X: ∀n. X n ∈ {a <..} ∧ X n ≠ a X ⟹ a
  show eventually (λn. P (X n)) sequentially
proof (rule ccontr)
  assume neg: ¬thesis
  have ∃s. ∀n. (¬ P (X (s n)) ∧ X (s n) < b) ∧ (X (s (Suc n)) ≤ X (s n) ∧ Suc (s n) ≤ s (Suc n))
  (is ∃s. ?thesis)
  proof (rule dependent-nat-choice)
    have ¬ eventually (λn. X n < b ⟹ P (X n)) sequentially
    by (intro not-eventually-impl neg order-tendstoD (OF X (2) b])
    then show ∃x. ¬ P (X x) ∧ X x < b
      by (auto dest!: not-eventuallyD)
  next
  fix x n
  have ¬ eventually (λn. Suc x ≤ n ⟹ X n < b ⟹ X n < X x ⟹ P (X n)) sequentially
  using X
  by (intro not-eventually-impl order-tendstoD (OF X (2) b] eventually-ge-at-top)
  qed
  then obtain s where ?thesis ..
  with X have a < X (s n)
  and X (s n) < b
  and decseq (λn. X (s n))
  and (λn. X (s n)) ⟹ a
  and ¬ P (X (s n))
  for n
  by (auto simp: strict-mono-Suc_iff Suc-le-iff decseq-Suc_iff)
  qed

lemma tendsto-at-right-sequentially:
  fixes a :: - :: {linorder-topology, first-countable-topology}
  assumes a < b
  and *: ∃S. (∀n. a < S n) ⟹ (∀n. S n < b) ⟹ decseq S ⟹ S ⟹ a
  shows (X ⟹ L) (at-right a)
using assms by (simp add: tendsto-def [where l=L] sequentially-imp-eventually-at-right)
96.10 Continuity

96.10.1 Continuity on a set

definition continuous-on :: 'a set ⇒ ('a::topological-space ⇒ 'b::topological-space) ⇒ bool
  where continuous-on s f ≡ (∀x∈s. (f −→ f x) (at x within s))

lemma continuous-on-cong [cong]:
  s = t ⇒ (∀x∈t. f x = g x) ⇒ continuous-on s f ←→ continuous-on t g
  unfolding continuous-on-def
  by (intro ball-cong filterlim-cong) (auto simp: eventually-at-filter)

lemma continuous-on-cong-simp:
  s = t ⇒ (∀x∈t. f x = g x) ⇒ continuous-on s f ←→ continuous-on t g
  unfolding simp-implies-def by (rule continuous-on-cong)

lemma continuous-on-topological:
  continuous-on s f ←→ (∀x∈s. ∀B. open B −→ f x ∈ B −→ (∃A. open A ∧ x ∈ A ∧ (∀y∈s. y ∈ A −→ f y ∈ B)))
  unfolding continuous-on-def tendsto-def eventually-at-topological by metis

lemma continuous-on-open-invariant:
  continuous-on s f ←→ (∀B. open B −→ (∃A. open A ∧ A ∩ s = f −' B ∩ s))
proof safe
  fix B :: 'b set
  assume continuous-on s f open B
  then have ∀x∈f −' B ∩ s. (∃A. open A ∧ x ∈ A ∧ s ∩ A ⊆ f −' B)
    by (auto simp: continuous-on-topological subset-eq Ball-def imp-conjL)
  then obtain A where ∀x∈f −' B ∩ s. open (A x) ∧ x ∈ A x ∧ s ∩ A x ⊆ f −' B
    unfolding bchoice-iff ..
  then show ∃A. open A ∧ A ∩ s = f −' B ∩ s
    by (intro exI[of _ ∪ x∈f −' B ∩ s. A x]) auto
next
  assume B: ∀B. open B −→ (∃A. open A ∧ A ∩ s = f −' B ∩ s)
  show continuous-on s f
    unfolding continuous-on-topological
  proof safe
    fix x B
    assume x ∈ s open B f x ∈ B
    with B obtain A where A: open A A ∩ s = f −' B ∩ s
      by auto
    with ⟨x ∈ s. f x ∈ B⟩ show ∃A. open A ∧ x ∈ A ∧ (∀y∈s. y ∈ A −→ f y ∈ B)
      by (intro exI[of _ A]) auto
  qed
qed

lemma continuous-on-open-vimage:
open s \Rightarrow continuous-on s f \longleftrightarrow (\forall B. \text{open } B \longrightarrow \text{open } f \setminus B \cap s))
unfolding continuous-on-open-invariant
by (metis open-Int Int-absorb Int-commute[of s] Int-assoc[of - s])

corollary continuous-imp-open-vimage:
assumes continuous-on s f open s open B f \setminus B \subseteq s
shows open (f \setminus B)
by (metis assms continuous-on-open-vimage le-iff-inf)

corollary open-vimage[continuous-intros]:
assumes open s
and continuous-on UNIV f
shows open (f \setminus s)
using assms by (simp add: continuous-on-open-vimage[of open-UNIV])

lemma continuous-on-closed-vimage:
continuous-on s f \longleftrightarrow (\forall B. \text{closed } B \longrightarrow (\exists A. \text{closed } A \cap A \cap s = f \setminus B \cap s))

proof -
have *: (\forall A. P A \longleftrightarrow Q (\neg A)) \Longrightarrow (\forall A. P A) \longleftrightarrow (\forall A. Q A)
for P Q :: 'b set \Rightarrow bool
by (metis double-compl)
show ?thesis
unfolding continuous-on-open-invariant
by (intro *) (auto simp: open-closed[symmetric])

qed

lemma continuous-on-closed-vimage:
closed s \Rightarrow continuous-on s f \longleftrightarrow (\forall B. \text{closed } B \longrightarrow \text{closed } (f \setminus B \cap s))
unfolding continuous-on-closed-invariant
by (metis closed-Int Int-absorb Int-commute[of s] Int-assoc[of - - s])

corollary closed-vimage-Int[continuous-intros]:
assumes closed s
and continuous-on t f
and t: closed t
shows closed (f \setminus s \cap t)
using assms by (simp add: continuous-on-closed-vimage[of t])

corollary closed-vimage[continuous-intros]:
assumes closed s
and continuous-on UNIV f
shows closed (f \setminus s)
using closed-vimage-Int[of assms] by simp

lemma continuous-on-empty [simp]: continuous-on {} f
by (simp add: continuous-on-def)

lemma continuous-on-sing [simp]: continuous-on \{x\} f
  by (simp add: continuous-on-def at-within-def)

lemma continuous-on-open-Union:
  \((\forall s. s \in S \Rightarrow open s) \Rightarrow (\forall s. s \in S \Rightarrow continuous-on s f) \Rightarrow continuous-on (\bigcup S) f\)
  unfolding continuous-on-def
  by safe (metis open-Union at-within-open UnionI)

lemma continuous-on-open-UN:
  \((\forall s. s \in S \Rightarrow open (A s)) \Rightarrow (\forall s. s \in S \Rightarrow continuous-on (A s) f) \Rightarrow continuous-on (\bigcup s \in S. A s) f\)
  by (rule continuous-on-open-Union) auto

lemma continuous-on-open-Un:
  \(open s \Rightarrow open t \Rightarrow continuous-on s f \Rightarrow continuous-on t f \Rightarrow continuous-on (s \cup t) f\)
  using continuous-on-open-Union [of \{s,t\}] by auto

lemma continuous-on-closed-Un:
  \(closed s \Rightarrow closed t \Rightarrow continuous-on s f \Rightarrow continuous-on t f \Rightarrow continuous-on (s \cup t) f\)
  by (auto simp add: continuous-on-closed-vimage closed-Un Int-Un-distrib)

lemma continuous-on-closed-Union:
  assumes finite I
  \(\forall i. i \in I \Rightarrow closed (U i)\)
  \(\forall i. i \in I \Rightarrow continuous-on (U i) f\)
  shows continuous-on (\bigcup i \in I. U i) f
  using assms
  by (induction I) (auto intro!: continuous-on-closed-Un)

lemma continuous-on-If:
  assumes closed: closed s closed t
    and cont: continuous-on s f continuous-on t g
    and P: \(\forall x. x \in s \Rightarrow \neg P x \Rightarrow f x = g x \\\\\\\forall x. x \in t \Rightarrow P x \Rightarrow f x = g x\)
  shows continuous-on (s \cup t) (\(\lambda x. if P x then f x else g x\))
    (is continuous-on - ?h)
  proof
    from P have \(\forall x\in s. f x = \ ?h x \\forall x\in t. g x = \ ?h x\)
      by auto
    with cont have continuous-on s \(\ ?h\) continuous-on t \(\ ?h\)
      by simp-all
    with closed show \(\ ?thesis\)
      by (rule continuous-on-closed-Un)
  qed
lemma continuous-on-cases:
closed s ⇒ closed t ⇒ continuous-on s f ⇒ continuous-on t g ⇒
∀x. (x ∈ s ∧ ¬ P x) ∨ (x ∈ t ∧ P x) ⇒ f x = g x
by (rule continuous-on-If) auto

lemma continuous-on-id[continuous-intros,simp]: continuous-on s (λx. x)
unfolding continuous-on-def by fast

lemma continuous-on-id'[continuous-intros,simp]: continuous-on s id
unfolding continuous-on-def id-def by fast

lemma continuous-on-const[continuous-intros,simp]: continuous-on s (λx. c)
unfolding continuous-on-def by auto

lemma continuous-on-subset: continuous-on s f ⇒ t ⊆ s ⇒ continuous-on t f
unfolding continuous-on-def
by (metis subset-eq tendsto-within-subset)

lemma continuous-on-compose[continuous-intros]:
continuous-on s f ⇒ continuous-on (f ' s) g ⇒ continuous-on s (g ◦ f)
unfolding continuous-on-topological by simp metis

lemma continuous-on-compose2:
continuous-on t g ⇒ continuous-on s f ⇒ f ' s ⊆ t ⇒ continuous-on s (λx. g (f x))
using continuous-on-compose[of s f g] continuous-on-subset by (force simp add: comp-def)

lemma continuous-on-generate-topology:
assumes *: open = generate-topology X
and **: ∀B. B ∈ X ⇒ ∃C. open C ∧ C ∩ A = f −' B ∩ A
shows continuous-on A f
unfolding continuous-on-open-invariant
proof safe
fix B :: 'a set
assume open B
then show ∃C. open C ∧ C ∩ A = f −' B ∩ A
unfolding *
proof induct
  case (UN K)
  then obtain C where ∀k. k ∈ K ⇒ open (C k) ∩ k. k ∈ K ⇒ C k ∩ A
  = f −' k ∩ A
  by metis
  then show ?case
  by (intro exI[of - (UN k∈K. C k)]) blast
qed (auto intro: **)
lemma continuous-on-IccI:

fixes f :: 'a::linorder_topology ⇒ 'b::{dense-order,linorder_topology}
assumes open (f' A)
and mono: ∀x y. x ∈ A ⇒ y ∈ A ⇒ x ≤ y ⇒ f x ≤ f y
shows continuous-on A f

proof (rule continuous-on-generate_topology[OF open_generated_order], safe)
have monoD: ∀x y. x ∈ A ⇒ y ∈ A ⇒ f x < f y ⇒ x < y
  by (auto simp: not_le[symmetric] mono)
have ∃x. x ∈ A ∧ f x < b ∧ a < x if a: a ∈ A and fa: f a < b for a b
proof –
  obtain y where f a < y {f a..<y} ⊆ f' A
    using open-right[OF OF 'open (f' A), of f a b] a fa
    by auto
  obtain z where z: f a < z z < min b y
    using dense[of f a min b y] (f a < y) (f a < b) by auto
  then obtain c where z = f c c ∈ A
    using ({f a..<y} ⊆ f' A)[THEN subsetD, of z] by (auto simp: less_imp_le)
  with a z show ?thesis
    by (auto intro!: exI[of - c] simp: monoD)
qed

then show ∃C. open C ∧ C ∩ A = f − {..<b} ∩ A for b
  by (intro exI[of - (⋃x∈{x∈A. f x < b}. {..<x})])
    (auto intro: le_less_trans[OF mono] less_imp_le)

have ∃x. x ∈ A ∧ b < f x ∧ x < a if a: a ∈ A and fa: b < f a for a b
proof –
  note a fa
  moreover
  obtain y where y < f a {y..<f a} ⊆ f' A
    using open-left[OF OF 'open (f' A), of f a b] a fa
    by auto
  then obtain z where z: max b y < z z < f a
    using dense[of max b y f a] (y < f a) (b < f a) by auto
  then obtain c where z = f c c ∈ A
    using ({y..<f a} ⊆ f' A)[THEN subsetD, of z] by (auto simp: less_imp_le)
  with a z show ?thesis
    by (auto intro!: exI[of - c] simp: monoD)
qed

then show ∃C. open C ∧ C ∩ A = f − {b..<} ∩ A for b
  by (intro exI[of - (⋃x∈{x∈A. b < f x}. {x..<})])
    (auto intro: less_le_trans[OF mono] less_imp_le)
qed

lemma continuous-on-IccI:

[[f →→ f a] (at-right a);
 (f →→ f b) (at-left b);
 (∀x. a < x ⇒ x < b ⇒ f −→ f x); a < b] ⇒
continuous-on {a .. b} f
for a::'a::linorder_topology
using at-within-open[of - {a<..<b}]
by (auto simp: continuous-on-def at-within-Icc-at-right at-within-Icc-at-left le-less at-within-Icc-at)

lemma
  fixes a b :: 'a::linorder-topology
  assumes continuous-on {a .. b} f a < b
  shows continuous-on-Icc-at-rightD: (f ----> f a) (at-right a)
    and continuous-on-Icc-at-leftD: (f ----> f b) (at-left b)
  using assms
  by (auto simp: at-within-Icc-at-right at-within-Icc-at-left continuous-on-def dest: bspec[where x=a] bspec[where x=b])

lemma continuous-on-discrete [simp]:
  continuous-on A (f :: 'a :: discrete-topology ⇒ -)  
  by (auto simp: continuous-on-def at-discrete)

96.10.2 Continuity at a point

definition continuous :: 'a::t2-space filter ⇒ (′a ⇒ ′b::topological-space) ⇒ bool
  where continuous F f ≡ (f −−−→ f (Lim F (λx. x)))) F

lemma continuous-bot[continuous-intros, simp]: continuous bot f
  unfolding continuous-def by auto

lemma continuous-trivial-limit: trivial-limit net ⇒ continuous net f
  by simp

lemma continuous-within: continuous (at x within s) f ←→ (f −−−→ f x) (at x within s)
  by (cases trivial-limit (at x within s)) (auto simp add: Lim-ident-at continuous-def)

lemma continuous-within-topological:
  continuous (at x within s) f ←→
    (∀ B. open B −→ f x ∈ B −→ (∃ A. open A ∧ x ∈ A ∧ (∀ y∈s. y ∈ A −→ f y ∈ B.present)))
  unfolding continuous-within tendsto-def eventually-at-topological by metis

lemma continuous-within-compose[continuous-intros]:
  continuous (at x within s) f ⇒ continuous (at (f x) within f ' s) g ⇒
    continuous (at x within s) (g ◦ f)
  by (simp add: continuous-within-topological) metis

lemma continuous-within-compose2:
  continuous (at x within s) f ⇒ continuous (at (f x) within f ' s) g ⇒
    continuous (at x within s) (λ x. g (f x))
  using continuous-within-compose[of x s f g] by (simp add: comp-def)

lemma continuous-at: continuous (at x) f ←→ f −x→ f x
using continuous-within[of x UNIV f] by simp

lemma continuous-ident[continuous-intros, simp]: continuous (at x within S) (λx. x)
  unfolding continuous-within by (rule tendsto-ident-at)

lemma continuous-id[continuous-intros, simp]: continuous (at x within S) id
  by (simp add: id-def)

lemma continuous-const[continuous-intros, simp]: continuous F (λx. c)
  unfolding continuous-def by (rule tendsto-const)

lemma continuous-on-eq-continuous-within:
  continuous-on s f ←→ (∀x∈s. continuous (at x within s) f)
  unfolding continuous-on-def continuous-within ..

lemma continuous-discrete [simp]:
  continuous (at x within A) (f :: 'a :: discrete-topology ⇒ -)
  by (auto simp: continuous-def at-discrete)

abbreviation isCont :: (′a :: t2-space ⇒ ′b :: topological-space) ⇒ ′a ⇒ bool
  where isCont f a ≡ continuous (at a) f

lemma isCont-def: isCont f a ←→ f −a→ f a
  by (rule continuous-at)

lemma isContD: isCont f x =⇒ f −x→ f x
  by (simp add: isCont-def)

lemma isCont-cong:
  assumes eventually (λx. f x = g x) (nhds x)
  shows isCont f x ←→ isCont g x
proof –
  from assms have [simp]: f x = g x
    by (rule eventually-nhds-x-imp-x)
  from assms have eventually (λx. f x = g x) (at x)
    by (auto simp: eventually-at-filter elim!: eventually-mono)
  with assms have isCont f x ←→ isCont g x unfolding isCont-def
    by (intro filterlim-cong) (auto elim!: eventually-mono)
  with assms show ?thesis by simp
qed

lemma continuous-at-imp-continuous-at-within: isCont f x =⇒ continuous (at x within s) f
  by (auto intro: tendsto-mono at-le simp: continuous-at continuous-within)

lemma continuous-on-eq-continuous-at: open s =⇒ continuous-on s f ←→ (∀x∈s. isCont f x)
  by (simp add: continuous-on-def continuous-at at-within-open[of - s])
lemma continuous-within-open: \( a \in A \implies \text{open } A \implies \text{continuous } (\text{at } a \text{ within } A) f \iff \text{isCont } f a \)
by (simp add: at-within-open-NO-MATCH)

lemma continuous-at-imp-continuous-on: \( \forall x \in s. \text{isCont } f x \implies \text{continuous-on } s f \)
by (auto intro: continuous-at-imp-continuous-at-within simp: continuous-on-eq-continuous-within)

lemma isCont-o2: \( \text{isCont } f a \implies \text{isCont } g (f a) \implies \text{isCont } (\lambda x. g (f x)) a \)
unfolding isCont-def by (rule tendsto-compose)

lemma continuous-at-compose: \( \text{isCont } f a \implies \text{continuous-on } s f \implies \text{continuous-on } ((\lambda x. g (f x)) a) \)
unfolding o-def by (rule isCont-o2)

lemma isCont-tendsto-compose: \( \text{isCont } g l \implies f \longrightarrow l F \implies ((\lambda x. g (f x)) \longrightarrow g l) F \)
unfolding isCont-def by (rule tendsto-compose)

lemma continuous-on-tendsto-compose:
assumes f-cont: \( \text{continuous-on } s f \)
and g: \( (g \longrightarrow l) F \)
and l: \( l \in s \)
and ev: \( \forall x \in F. g x \in s \)
shows \( ((\lambda x. f (g x)) \longrightarrow f l) F \)
proof –
from f-cont l have f: \( (f \longrightarrow f l) \text{ (at } l \text{ within } s) \)
  by (simp add: continuous-on-def)
have i: \( ((\lambda x. \text{if } g x = l \text{ then } f l \text{ else } f (g x)) \longrightarrow f l) F \)
  by (rule filterlim-If)
  (auto intro!: filterlim-compose[OF f] eventually-conj tendsto-mono[OF - g]
    simp: filterlim-at eventually-inf-principal eventually-mono[OF ev])
show \( ?\text{thesis} \)
  by (rule filterlim-cong[THEN iffD1[OF - i]]) auto
qed

lemma continuous-within-compose2: \( \text{isCont } g (f x) \implies \text{continuous } (\text{at } x \text{ within } s) f \implies \text{continuous } (\text{at } x \text{ within } s) (\lambda x. g (f x)) \)
using continuous-at-imp-continuous-at-within continuous-within-compose2 by blast

lemma filtermap-nhds-open-map:
assumes cont: \( \text{isCont } f a \)
and open-map: \( \forall S. \text{open } S \implies \text{open } (f S) \)
shows \( \text{filtermap } f (\text{nhds } a) = \text{nhds } (f a) \)
unfolding filter-eq-iff
proof safe
fix \( P \)
assume eventually $P$ (filtermap $f$ (nhds $a$))
then obtain $S$ where open $S$ a ∈ $S$ ∀ $x$ ∈ $S$. $P$ ($f$ $x$)
  by (auto simp: eventually-filtermap eventually-nhds)
then show eventually $P$ (nhds ($f$ $a$))
  unfolding eventually-nhds by (intro exI [of - $f$'$S$]) (auto intro!: open-map)
qed (metis filterlim-iff tendsto-at-iff-tendsto-nhds isCont-def eventually-filtermap cont)

lemma continuous-at-split:
  continuous (at $x$) $f$ ←→ continuous (at-left $x$) $f$ ∧ continuous (at-right $x$) $f$
for $x$ :: 'a::linorder-topology
by (simp add: continuous-within filterlim-at-split)

lemma continuous-on-max [continuous-intros]:
  fixes $f$ $g$ :: 'a::topological-space ⇒ 'b::linorder-topology
  shows continuous-on $A$ $f$ ⇒ continuous-on $A$ $g$ ⇒ continuous-on $A$ ($λ$x. max ($f$ $x$) ($g$ $x$))
  by (auto simp: continuous-on-def intro!: tendsto-max)

lemma continuous-on-min [continuous-intros]:
  fixes $f$ $g$ :: 'a::topological-space ⇒ 'b::linorder-topology
  shows continuous-on $A$ $f$ ⇒ continuous-on $A$ $g$ ⇒ continuous-on $A$ ($λ$x. min ($f$ $x$) ($g$ $x$))
  by (auto simp: continuous-on-def intro!: tendsto-min)

lemma continuous-max [continuous-intros]:
  fixes $f$ :: 'a::t2-space ⇒ 'b::linorder-topology
  shows [continuous $F$ $f$; continuous $F$ $g$] ⇒ continuous $F$ ($λ$x. (max ($f$ $x$) ($g$ $x$)))
  by (simp add: tendsto-max continuous-def)

lemma continuous-min [continuous-intros]:
  fixes $f$ :: 'a::t2-space ⇒ 'b::linorder-topology
  shows [continuous $F$ $f$; continuous $F$ $g$] ⇒ continuous $F$ ($λ$x. (min ($f$ $x$) ($g$ $x$)))
  by (simp add: tendsto-min continuous-def)

The following open/closed Collect lemmas are ported from Sébastien Gouëzel’s Ergodic-Theory.

lemma open-Collect-neq:
  fixes $f$ $g$ :: 'a::topological-space ⇒ 'b::t2-space
  assumes $f$: continuous-on UNIV $f$ and $g$: continuous-on UNIV $g$
  shows open {x. $f$ $x$ ≠ $g$ $x$}
proof (rule openI)
fix $t$
assume $t$ ∈ {x. $f$ $x$ ≠ $g$ $x$}
then obtain $U$ $V$ where *: open $U$ open $V$ $f$ $t$ ∈ $U$ $g$ $t$ ∈ $V$ $U$ ∩ $V$ = {}
  by (auto simp add: separation-t2)
with open-vimage[OF open $U$] $f$ open-vimage[OF open $V$] $g$
show $\exists T. \text{open } T \wedge t \in T \wedge T \subseteq \{x. f x \neq g x\}$
by (intro exI[of - f - \text{U} \cap - g - \text{V}]) auto
qed

lemma closed-Collect-eq:
fixes $f \cdot g :: 'a \Rightarrow 'b \Rightarrow 'c$
assumes $f$: continuous-on UNIV $f$ and $g$: continuous-on UNIV $g$
shows closed $\{x. f x = g x\}$
using open-Collect-neq[OF $f \cdot g$] by (simp add: closed-def Collect-neg-eq)

lemma open-Collect-less:
fixes $f \cdot g :: 'a \Rightarrow 'b \Rightarrow 'c$
assumes $f$: continuous-on UNIV $f$ and $g$: continuous-on UNIV $g$
shows open $\{x. f x < g x\}$
proof (rule openI)
fix $t$
assume $t$: $t \in \{x. f x < g x\}$
show $\exists T. \text{open } T \wedge t \in T \wedge T \subseteq \{x. f x < g x\}$
proof (cases $\exists z. f t < z \wedge z < g t$)
case True
demonstrate $z$ where $f t < z \wedge z < g t$ by blast
demonstrate $\text{thesis}$
using open-vimage[OF $f$ of $\{..<z\}$] open-vimage[OF $g$ of $\{z<..\}$]
by (intro exI[of - $f$ - \{ ..<z \} \cap - g - \{ z<.. \}]) auto
next
case False
demonstrate $\ast$: $\{g t ..\} = \{f t <..\} \{..< g t\} = \{.. f t\}$
demonstrate $\ast$ by (auto intro: leI)
demonstrate $\text{thesis}$
using open-vimage[OF $f$ of $\{..< g t\}$] open-vimage[OF $g$ of $\{f t <..\}$] $t$
apply (intro exI[of - $f$ - \{ ..< g t \} \cap - g - \{ f t<.. \}])
apply (simp add: open-Int)
apply (auto simp add: $\ast$)
done
qed

lemma closed-Collect-le:
fixes $f \cdot g :: 'a \Rightarrow 'b \Rightarrow 'c$
assumes $f$: continuous-on UNIV $f$
and $g$: continuous-on UNIV $g$
shows closed $\{x. f x \leq g x\}$
using open-Collect-less $[OF g f]$ by (simp add: closed-def Collect-neg-eq[symmetric] not-le)

96.10.3 Open-cover compactness

context topological-space
begin
definition compact :: 'a set ⇒ bool where
compact-eq-Heine-Borel:
  compact S ⇔ (∀ C. (∀ c∈C. open c) ∧ S ⊆ ∪ C → (∃ D⊆C. finite D ∧ S ⊆ ∪ D))

lemma compactI:
  assumes ∃ C. ∀ t∈C. open t =⇒ s ⊆ ∪ C =⇒ ∃ C'. C' ⊆ C ∧ finite C' ∧ s ⊆ ∪ C'
  shows compact s
  unfolding compact-eq-Heine-Borel using assms by metis

lemma compact-empty[simp]: compact {}
  by (auto intro!: compactI)

lemma compactE:
  assumes compact S S ⊆ ∪ T ∧ B ∈ T =⇒ open B
  obtains T' where T' ⊆ T finite T' s ⊆ ∪ T'
  by (meson assms compact-eq-Heine-Borel)

lemma compactE-image:
  assumes compact S
  and opn: ∃ T. T ⊆ C =⇒ open (f T)
  and S: S ⊆ (∪ c∈C. f c)
  obtains C' where C' ⊆ C and finite C' and S ⊆ (∪ c∈C'. f c)
  apply (rule compactE[OF ‹compact S› S])
  using opn apply force
  by (metis finite-subset-image)

lemma compact-Int-closed [intro]:
  assumes compact S
  and closed T
  shows compact (S ∩ T)
proof (rule compactI)
  fix C
  assume C: ∀ c∈C. open c
  assume cover: S ∩ T ⊆ ∪ C
  from cover have ∀ c∈C ∪ {− T}. open c
  by auto
  moreover from cover have S ⊆ ∪ (C ∪ {− T})
  by auto
  ultimately have ∃ D⊆C ∪ {− T}. finite D ∧ S ⊆ ∪ D
    using ‹compact S› unfolding compact-eq-Heine-Borel by auto
  then obtain D where D ⊆ C ∪ {− T} ∧ finite D ∧ S ⊆ ∪ D ..
  then show ∃ D⊆C. finite D ∧ S ∩ T ⊆ ∪ D
    by (intro exI[of ‹− D − {− T}›]) auto
  qed

lemma compact-diff: compact S; open T → compact(S − T)
by (simp add: Diff-eq compact-Int-closed open-closed)

lemma inj-setminus: inj-on uminus \( A::\text{a set set} \)
by (auto simp: inj-on-def)

96.11 Finite intersection property

lemma compact-fip:
  compact \( U \iff \)
  \( (\forall A. (\forall a\in A. \text{closed } a) \rightarrow (\forall B \subseteq A. \text{finite } B \rightarrow U \cap \bigcap B \neq \{\})) \rightarrow U \cap \bigcap A \neq \{\} \)
(is - \( \iff \) ?R)

proof (safe intro!: compact-eq-Heine-Borel[THEN iffD2])
  fix \( A \)
  assume compact \( U \)
  assume \( \forall a\in A. \text{closed } a U \cap \bigcap A = \{\} \)
  from \( A \) have \( (\forall a\in \text{uminus'}A. \text{open } a) \land U \subseteq \bigcup(\text{uminus'}A) \)
  by auto
  with \( \text{compact } U \) obtain \( B \) where \( B \subseteq A \text{ finite } \text{uminus' } B \subseteq \bigcup(\text{uminus'}B) \)
  unfolding compact-eq-Heine-Borel by (metis subset-image-iff)
  with fin[THEN spec. of \( B \)] show False
  by (auto dest: finite-imageD intro: inj-setminus)

next
  fix \( A \)
  assume \( ?R \)
  assume \( \forall a\in A. \text{open } a U \subseteq \bigcup A \)
  then have \( U \cap \bigcap (\text{uminus'}A) = \{\} \forall a\in \text{uminus'}A. \text{closed } a \)
  by auto
  with \( ?R \) obtain \( B \) where \( B \subseteq A \text{ finite } \text{uminus' } B \subseteq \bigcup(\text{uminus'}B) = \{\} \)
  by (metis subset-image-iff)
  then show \( \exists T\subseteq A. \text{finite } T \land U \subseteq \bigcup T \)
  by (auto intro!: exI[of - B] inj-setminus dest: finite-imageD)
qed

lemma compact-imp-fip:
  assumes compact \( S \)
  and \( \bigwedge T, T \in F \rightarrow \text{closed } T \)
  and \( \bigwedge F', \text{finite } F' \rightarrow F' \subseteq F \rightarrow S \cap (\bigcap F') \neq \{\} \)
  shows \( S \cap (\bigcap F) \neq \{\} \)
  using assms unfolding compact-fip by auto

lemma compact-imp-fip-image:
  assumes compact \( s \)
  and \( P: \bigwedge i, i \in I \rightarrow \text{closed } (f i) \)
  and \( Q: \bigwedge I', \text{finite } I' \rightarrow I' \subseteq I \rightarrow (s \cap (\bigcap i\in I'. f i) \neq \{\}) \)
  shows \( s \cap (\bigcap i\in I. f i) \neq \{\} \)

proof
  note compact \( s \)
moreover from $P$ have $\forall i \in f \cdot I. \text{closed } i$
by blast
moreover have $\forall A. \text{finite } A \land A \subseteq f \cdot I \rightarrow (s \cap (\bigcap A) \neq \{\})$
apply rule
apply rule
apply (erule conjE)
proof
- fix $A :: 'a \text{ set set}$
  assume finite $A$ and $A \subseteq f \cdot I$
  then obtain $B$ where $B \subseteq I$ and finite $B$ and $A = f \cdot B$
  using finite-subset-image [of $A f I$] by blast
  with $Q$ [of $B$] show $s \cap \bigcap A \neq \{\}$
  by simp
  ultimately have $s \cap (\bigcap (f \cdot I)) \neq \{\}$
  by (metis compact-imp-fip)
  then show $\exists \text{thesis by simp}$
  qed
end

lemma (in t2-space) compact-imp-closed: assumes compact $s$
shows closed $s$
proof (rule openI)
fix $y$
assume $y \in - s$
let $\forall C = \bigcup x \in s. \{u. \text{open } u \land x \in u \land \text{eventually } (\lambda y. y \notin u) (\text{nhds } y)\}$
have $s \subseteq \bigcup \forall C$
proof
- fix $x$
  assume $x \in s$
  with $(y \in - s)$ have $x \neq y$ by clarsimp
  then have $\exists u v. \text{open } u \land \text{open } v \land x \in u \land y \in v \land u \cap v = \{\}$
  by (rule hausdorff)
  with $(x \in s)$ show $x \in \bigcup \forall C$
  unfolding eventually-nhds by auto
  qed
then obtain $D$ where $D \subseteq \forall C$ and finite $D$ and $s \subseteq \bigcup D$
by (rule compactE [OF 'compact s]) auto
from $(D \subseteq \forall C) \land \forall x \in D. \text{eventually } (\lambda g. y \notin x) (\text{nhds } y)$
by auto
with $(\text{finite } D)$ have eventually $(\lambda g. y \notin \bigcup D) (\text{nhds } y)$
by (simp add: eventually-ball-finite)
with $(s \subseteq \bigcup D)$ have eventually $(\lambda g. y \notin s) (\text{nhds } y)$
by (auto elim!: eventually-mono)
then show $\exists t. \text{open } t \land y \in t \land t \subseteq - s$
by (simp add: eventually-nhds subset-eq)
qed

lemma compact-continuous-image:

assumes \( f\colon \text{continuous-on } s \ f \) and \( s\colon \text{compact } s \)

shows \( \text{compact } (f' s) \)

proof (rule compactI)

fix \( C \)

assume \( \forall c\in C. \text{ open } c \) and \( \text{cover: } f's \subseteq \bigcup C \)

with \( f\) have \( \forall c\in C. \exists A. \text{ open } A \land A \cap s = f'c \cap s \)

unfolding continuous-on-open-invariant by blast

then obtain \( A \) where \( A \colon \forall c\in C. \text{ open } (A c) \)

with \( A \) have \( \forall c\in C. (A c \cap s) = f'c \cap s \)

unfolding bchoice-iff ..

with \( \text{cover} \) have \( \forall c\in C. \exists A. \text{ open } A \land A \cap s = f'c \cap s \)

by (fastforce simp add: subset-eq set-eq-iff)

from compactE-image[OF \( s \) this]

obtain \( D \) where \( D \subseteq C, \text{ finite } D \subseteq f's \subseteq \bigcup c\in C. A c \)

with \( D \) show \( \exists D \subseteq C. \text{ finite } D \land f's \subseteq \bigcup D \)

by (intro exI[of - \( D \)]) (fastforce simp add: subset-eq set-eq-iff)

qed

lemma continuous-on-inv:

fixes \( f\colon 'a::topological-space \Rightarrow 'b::t2-space \) assumes \( \text{continuous-on } s \ f \) and \( \text{compact } s \)

and \( \forall x\in s. g(f x) = x \)

shows \( \text{continuous-on } (f' s) g \)

unfolding continuous-on-topological proof (clarsimp simp add: assms(3))

fix \( x\colon 'a \) and \( B\colon 'a \) set

assume \( x\in s \) and \( \text{open } B \) and \( x\in B \)

have \( 1\colon \forall x\in s. f x \in f' (s - B) \leftrightarrow x \in s - B \)

using assms(3) by (auto, metis)

have \( \text{continuous-on } (s - B) f \)

using (continuous-on s f) Diff-subset by (rule continuous-on-subset)

moreover have \( \text{compact } (s - B) \)

using (open B) and (compact s) unfolding Diff-eq by (intro compact-Int-closed closed-Compl)

ultimately have \( \text{compact } (f' (s - B)) \)

by (rule compact-continuous-image)

then have \( \text{closed } (f' (s - B)) \)

by (rule compact-imp-closed)

then have \( \text{open } (- f' (s - B)) \)

by (rule open-Compl)

moreover have \( f x \in - f' (s - B) \)

using \( \forall x\in s. f x \in - f' (s - B) \)

by (simp add: 1)

moreover have \( \forall y\in s. f y \in - f' (s - B) \)

by (simp add: 1)
ultimately show $\exists A. \text{open } A \land f x \in A \land (\forall y \in s. f y \in A \implies y \in B)$

by fast

qed

lemma continuous-on-inv-into:
fixes $f :: 'a::topological-space \Rightarrow 'b::t2-space$
assumes $s: \text{continuous-on } s f$ $\text{compact } s$
and $f: \text{inj-on } f s$
shows $\text{continuous-on } (f \circ s)$ $(\text{the-inv-into } s f)$
by (rule continuous-on-inv [OF $s$]) (auto simp: the-inv-into-f-f)

lemma (in linorder-topology) compact-attains-sup:
assumes $\text{compact } S S \neq \{\}$
shows $\exists s \in S. \forall t \in S. t \leq s$
proof (rule classical)
assume $\neg (\exists s \in S. \forall t \in S. t \leq s)$
then obtain $t$ where $t: \forall s \in S. t s \in S$ and $\forall s \in S. s \leq t s$
by (metis not-le)
then have $\forall s. s \in S \implies \text{open } \{..< t s\} S \subseteq (\bigcup s \in S. \{..< t s\})$
by auto
with $\langle \text{compact } S \rangle$ obtain $C$ where $C \subseteq S$ finite $C$ and $C: S \subseteq (\bigcup s \in C. \{..< t s\})$
by (metis compactE-image)
with $\langle S \neq \{\} \rangle$ have $\text{Max}: \text{Max } (t'C) \in t'C$ and $\forall s \in t'C. s \leq \text{Max } (t'C)$
by (auto intro!: Max-in)
with $C$ have $S \subseteq \{..< \text{Max } (t'C)\}$
by (auto intro: less-le-trans simp: subset-eq)
with $t$ $\text{Max } (C \subseteq S)$ show $?\text{thesis}$
by fastforce
qed

lemma (in linorder-topology) compact-attains-inf:
assumes $\text{compact } S S \neq \{\}$
shows $\exists s \in S. \forall t \in S. s \leq t$
proof (rule classical)
assume $\neg (\exists s \in S. \forall t \in S. s \leq t)$
then obtain $t$ where $t: \forall s \in S. t s \in S$ and $\forall s \in S. t s \leq s$
by (metis not-le)
then have $\forall s. s \in S \implies \text{open } \{t s <..\} S \subseteq (\bigcup s \in S. \{t s <..\})$
by auto
with $\langle \text{compact } S \rangle$ obtain $C$ where $C \subseteq S$ finite $C$ and $C: S \subseteq (\bigcup s \in C. \{t s <..\})$
by (metis compactE-image)
with $\langle S \neq \{\} \rangle$ have $\text{Min}: \text{Min } (t'C) \in t'C$ and $\forall s \in t'C. \text{Min } (t'C) \leq s$
by (auto intro!: Min-in)
with $C$ have $S \subseteq \{\text{Min } (t'C) <..\}$
by (auto intro: le-less-trans simp: subset-eq)
with $t$ $\text{Min } (C \subseteq S)$ show $?\text{thesis}$
by fastforce
qed

lemma continuous-attains-sup:
  fixes f :: 'a::topological-space ⇒ 'b::linorder-topology
  shows compact s =⇒ s ≠ {} =⇒ continuous-on s f =⇒ (∃x∈s. ∀y∈s. f y ≤ f x)
  using compact-attains-sup[of f ' s] compact-continuous-image[of s f] by auto

lemma continuous-attains-inf:
  fixes f :: 'a::topological-space ⇒ 'b::linorder-topology
  shows compact s =⇒ s ≠ {} =⇒ continuous-on s f =⇒ (∃x∈s. ∀y∈s. f y ≤ f x)
  using compact-attains-inf[of f ' s] compact-continuous-image[of s f] by auto

96.12 Connectedness

context topological-space
begin

definition connected S ←→ ¬ (∃A B. open A ∧ open B ∧ S ⊆ A ∪ B ∧ A ∩ B ∩ S = {} ∧ A ∩ S ≠ {} ∧ B ∩ S ≠ {})

lemma connectedI:
  (∀A B. open A ⇒ open B ⇒ A ∩ U ≠ {} ⇒ B ∩ U ≠ {} ⇒ A ∩ B ∩ U = {} ⇒ U ⊆ A ∪ B ⇒ False)
  =⇒ connected U
  by (auto simp: connected-def)

lemma connected-empty [simp]: connected {}
  by (auto intro!: connectedI)

lemma connected-sing [simp]: connected {x}
  by (auto intro!: connectedI)

lemma connectedD:
  connected A ⇒ open U ⇒ open V ⇒ U ∩ V ∩ A = {} ⇒ A ⊆ U ∪ V
  =⇒ U ∩ A = {} ∨ V ∩ A = {}
  by (auto simp: connected-def)

end

lemma connected-closed:
  connected s =⇒ ¬ (∃A B. closed A ∧ closed B ∧ s ⊆ A ∪ B ∧ A ∩ B ∩ s = {} ∧ A ∩ s ≠ {} ∧ B ∩ s ≠ {})
  apply (simp add: connected-def del: ex-simps, safe)
  apply (dral_tac x=−A in spec)
  apply (dral_tac x=−B in spec)
apply (fastforce simp add: closed-def [symmetric])
apply (drule-tac x=−A in spec)
apply (drule-tac x=−B in spec)
apply (fastforce simp add: open-closed [symmetric])
done

lemma connected-closedD:
[ connected s; A ∩ B ∩ s = {}; s ⊆ A ∪ B; closed A; closed B ] ⇒ A ∩ s = {}
v B ∩ s = {} by (simp add: connected-closed)

lemma connected-Union:
assumes cs: ⋀s. s ∈ S ⇒ connected s
and ne: ∩S ≠ {}
shows connected(∪S)
proof (rule connectedI)
  fix A B
  assume A: open A and B: open B and Alap: A ∩ ∪S ≠ {} and Blap: B ∩ ∪S ≠ {}
  and disj: A ∩ B ∩ ∪S = {} and cover: ∪S ⊆ A ∪ B
  have disjs: ⋀s. s ∈ S ⇒ A ∩ B ∩ s = {} using disj by auto
  obtain sa where sa: sa ∈ S A ∩ sa ≠ {} using Alap by auto
  obtain sb where sb: sb ∈ S B ∩ sb ≠ {} using Blap by auto
  obtain x where x: ⋀s. s ∈ S ⇒ x ∈ s using ne by auto
  then have x ∈ ∪S using ⟨sa ∈ S⟩ by blast
  then have x ∈ A ∨ x ∈ B using cover by auto
  then show False using cs [unfolded connected-def]
    by (metis A B IntI Sup-upper sa sb disjs x cover empty-iff subset-trans)
qed

lemma connected-Un: connected s ⇒ connected t ⇒ s ∩ t ≠ {} ⇒ connected (s ∪ t)
  using connected-Union [of {s,t}] by auto

lemma connected-diff-open-from-closed:
assumes st: s ⊆ t
and tu: t ⊆ u
and s: open s
and t: closed t
and u: connected u
and ts: connected (t − s)
shows connected(u − s)
proof (rule connectedI)
fix A B
assume AB: open A open B A ∩ (u − s) ≠ {} B ∩ (u − s) ≠ {} and disj: A ∩ B ∩ (u − s) = {}
and cover: u − s ⊆ A ∪ B
then consider A ∩ (t − s) = {} B ∩ (t − s) = {}
using st ts tu connectedD [of t−s A B] by auto
then show False
proof cases
  case 1
  then have (A − t) ∩ (B ∪ s) ∩ u = {} using disj st by auto
  moreover have u ⊆ (A − t) ∪ (B ∪ s) using 1 cover by auto
  ultimately show False using connectedD [of u A − t B ∪ s] AB s t 1 u by auto
next
  case 2
  then have (A ∪ s) ∩ (B − t) ∩ u = {} using disj st by auto
  moreover have u ⊆ (A ∪ s) ∪ (B − t) using 2 cover by auto
  ultimately show False using connectedD [of u A ∪ s B − t] AB s t 2 u by auto
qed

lemma connected-iff-const:
fixes S :: 'a::topological-space set
shows connected S ←→ (∀ P::'a ⇒ bool. continuous-on S P → (∃ c. ∀ s∈S. P s = c))
proof safe
fix P :: 'a ⇒ bool
assume connected S continuous-on S P
then have ∀ b. ∃ A. open A ∧ A ∩ S = P − {b} ∩ S unfolding continuous-on-open-invariant by (simp add: open-discrete)
from this[of True] this[of False]
obtain t f where open t open f and *: f ∩ S = P − {True} ∩ S t ∩ S = P − {False} ∩ S t ∩ S = P
  by meson
then have t ∩ S = {} ∨ f ∩ S = {} by (intro connectedD[of connected S]) auto
then show ∃ c. ∀ s∈S. P s = c by (rule disjE)
assume t ∩ S = {}
then show ?thesis unfolding * by (intro exI[of False]) auto
next
assume f ∩ S = {}
then show \( \text{thesis} \)

  unfolding * by (intro exI[of - True]) auto

qed

next

assume \( P : \forall P : 'a \Rightarrow \text{bool} \). \text{continuous-on} \ S \ P \longrightarrow (\exists c. \forall s \in S. P \ s = c) \)

show \( \text{connected} \ S \)

proof (rule connectedI)

  fix \( A. B \)

  assume *: \( \text{open} \ A. \text{open} \ B \). \( A \cap S \neq \{\} \). \( B \cap S \neq \{\} \). \( A \cap B \cap S = \{\} \). \( S \subseteq A \cup B \)

  have \( \text{continuous-on} \ S \ (\lambda x. x \in A) \)

  unfolding \( \text{continuous-on-open-invariant} \)

  proof safe

    fix \( C :: \text{bool set} \)

    have \( C = \text{UNIV} \lor C = \{\text{True}\} \lor C = \{\text{False}\} \lor C = \{\} \)

    using \( \text{subset-UNIV[of } C\text{]} \)

    unfolding \( \text{UNIV-bool} \)

    by (auto)

    with * show \( \exists T. \text{open} \ T \land T \cap S = (\lambda x. x \in A) - ' C \cap S \)

    by (intro exI[of - (if True \in C then A else \{\}) \cup (if False \in C then B else \{\})]) auto

  qed

  from \( P \)

  [rule-format, OF this]

  obtain \( c \) where \( \forall s \in S. \ s \in A \Longrightarrow (s \in A) = c \)

  by blast

  with * show \( \text{False} \)

  by (cases \( c \)) auto

qed

lemma \( \text{connectedD-const}: \text{connected} \ S \Longrightarrow \text{continuous-on} \ S \ P \Longrightarrow \exists c. \forall s \in S. P \ s = c \)

for \( P :: 'a :: \text{topological-space} \Rightarrow \text{bool} \)

by (auto simp: connected-iff-const)

lemma \( \text{connectedI-const}: (\forall P :: 'a :: \text{topological-space} \Rightarrow \text{bool} . \text{continuous-on} \ S \ P \Longrightarrow \exists c. \forall s \in S. P \ s = c) \Longrightarrow \text{connected} \ S \)

by (auto simp: connected-iff-const)

lemma \( \text{connected-local-const}: \)

assumes \( \text{connected} \ A \ a \in A \ b \in A \)

and *: \( \forall a \in A. \ \text{eventually} \ (\lambda b. f \ a = f \ b) \ (a \ \text{at} \ a \ \text{within} \ A) \)

shows \( f \ a = f \ b \)

proof

  obtain \( S \) where \( S : \forall a. \ a \in A \Longrightarrow a \in S \) a \( \forall a. \ a \in A \Longrightarrow \text{open} \ (S \ a) \)

  \( \forall a \ x. \ a \in A \Longrightarrow x \in S \) a \( \Longrightarrow x \in A \Longrightarrow f \ a = f \ x \)

  using * unfolding \( \text{eventually-at-topological by metis} \)

  let \( ?P = \bigcup \{b \in A. f \ a = f \ b\}. \ S \ b \) and \( ?N = \bigcup \{b \in A. f \ a \neq f \ b\}. \ S \ b \)

  have \( ?P \cap A = \{\} \lor ?N \cap A = \{\} \)

  using \( \text{connected} \ A \) \( S \ (a \in A) \)

  by (intro connectedD) (auto, metis)
then show \( f \, a = f \, b \)
proof
  assume \(?N \cap A = \{\}\)
  then have \(\forall x \in A. \, f \, a = f \, x\)
    using \(S(1)\) by auto
  with \(\{b \in A\}\) show \(?\text{thesis}\) by auto
next
  assume \(?P \cap A = \{\}\) then show \(?\text{thesis}\)
    using \(\{a \in A; \, S(1)\}\) by auto
qed

Lemma (in linorder-topology) connectedD-interval:
assumes connected \(U\)
and \(xy: \, x \in U \, y \in U\)
and \(x \leq z \, z \leq y\)
shows \(z \in U\)
proof –
  have eq: \(\{..<z\} \cup \{z<..\} = \{z\}\)
    by auto
  have \(\neg \) connected \(U\) if \(z \notin U \, x < z \, z < y\)
    using \(xy\) that
    apply (simp only: connected-def simp-thms)
    apply (rule-tac exI [of - {..< z}])
    apply (rule-tac exI [of - \{z <..\}])
    apply (auto simp add: eq)
    done
  with assms show \(z \in U\)
    by (metis less-le)
qed

Lemma (in linorder-topology) not-in-connected-cases:
assumes conn: connected \(S\)
assumes nbdd: \(x \notin S\)
assumes ne: \(S \neq \{\}\)
obtains bdd-above \(S \land y. \, y \in S \Longrightarrow x \geq y\) | bdd-below \(S \land y. \, y \in S \Longrightarrow x \leq y\)
proof –
  obtain \(s\) where \(s \in S\) using ne by blast
  
  assume \(s \leq x\)
  have \(\text{False}\) if \(x \leq y \, y \in S\) for \(y\)
    using connectedD-interval[OF conn \(s \in S\) \(y \in S\) \(s \leq x\) \(x \leq y\) \(x \notin S\)]
    by simp
  then have wit: \(y \in S \Longrightarrow x \geq y\) for \(y\)
    using le-cases by blast
  then have bdd-above \(S\)
    by (rule local.bdd-aboveI)
  note this wit
  } moreover {
**THEORY “Topological-Spaces”**

```
assume x ≤ s
have False if x ≥ y y ∈ S for y
  using connectedD-interval[OF conn ↯ y ∈ S ↯ s ∈ S ↯ x ≥ y ↯ s ≥ x ↯ x ∉ S]
  by simp
then have wit: y ∈ S ⟹ x ≤ y for y
  using le-cases by blast
then have bdd-below S
  by (rule bdd-belowI)
  note this wit
} ultimately show ?thesis
  by (meson le-cases that)
qed

lemma connected-continuous-image:
  assumes *: continuous-on s f
  and connected s
  shows connected (f ' s)
proof (rule connectedI-const)
  fix P :: 'b ⇒ bool
  assume continuous-on (f ' s) P
  then have continuous-on s (P ◦ f)
    by (rule continuous-on-compose[OF *])
  from connectedD-const[OF (connected s) this] show ∃ c. ∀ s ∈ f ' s. P s = c
    by auto
qed

97 Linear Continuum Topologies

class linear-continuum-topology = linorder-topology + linear-continuum
begin

lemma Inf-notin-open:
  assumes A: open A
  and bnd: ∀ a ∈ A. x < a
  shows Inf A /∈ A
proof
  assume Inf A ∈ A
  then obtain b where b < Inf A {b <.. Inf A} ⊆ A
    using open-left[of A Inf A x] assms by auto
  with dense[of b Inf A] obtain c where c < Inf A c ∈ A
    by (auto simp: subset-eq)
  then show False
    using cInf-lower[OF c ∈ A] bnd
    by (metis not-le less-imp-le bdd-belowI)
qed

lemma Sup-notin-open:
  assumes A: open A
  and bnd: ∀ a ∈ A. a < x
```
shows \( \sup A \notin A \)
proof
  assume \( \sup A \in A \)
  with assms obtain \( b \) where \( \sup A < b \) \( \{ \sup A \ldots < b \} \subseteq A \)
    using open-right[of \( A \sup A x \)] by auto
  with dense[of \( A b \)] obtain \( c \) where \( \sup A < c \) \( c \in A \)
    by (auto simp: subset-eq)
  then show \( \text{False} \)
    using \( c \supupper \{ c \in A \} \)
    by (metis less-imp-le not-le bdd-aboveI)
qed

end

instance linear-continuum-topology \( \subseteq \) perfect-space
proof
  fix \( x :: 'a \)
  obtain \( y \) where \( x < y \lor y < x \)
    using ex-gt-or-lt[of \( x \) ..]
  with Inf-notin-open[of \( \{ x \} y \)] Sup-notin-open[of \( \{ x \} y \)] show \( \neg \text{open} \{ x \} \)
    by auto
qed

lemma connectedI-interval:
  fixes \( U :: 'a :: linear-continuum-topology \) set
  assumes \( \star : \forall x y z. x \in U \Rightarrow y \in U \Rightarrow x \leq z \Rightarrow z \leq y \Rightarrow z \in U \)
  shows \( \text{connected} \ U \)
proof (rule connectedI)
  { fix \( A B \)
    assume \( \text{open} A \) \( \text{open} B \) \( A \cap B \cap U = \{ \} \) \( U \subseteq A \cup B \)
    fix \( x y \)
    assume \( x < y \) \( x \in A \) \( y \in B \) \( x \in U \) \( y \in U \)
    let \( ?z = \inf (B \cap \{ x \ldots \}) \)
    have \( x \leq ?z \) \( ?z \leq y \)
      using \( (y \in B) \ (x < y) \) by (auto intro: cInf-lower cInf-greatest)
    with \( (x \in U) \ (y \in U) \) have \( ?z \in U \)
      by (rule \( \star \))
    moreover have \( ?z \notin B \cap \{ x \ldots \} \)
      using \( (open B) \) (intro Inf-notin-open) auto
    ultimately have \( ?z \in A \)
      using \( (x \leq ?z) \ (A \cap B \cap U = \{ \}) \ (x \in A) \ (U \subseteq A \cup B) \) by auto
    have \( \exists b \in B. b \in A \land b \in U \) if \( ?z < y \)
    proof
      obtain \( a \) where \( ?z < a \) \( \{ ?z \ldots < a \} \subseteq A \)
        using open-right[of \( A \) \( ?z \) \( ?z \in A \) \( ?z < y \)] by auto
      moreover obtain \( b \) where \( b \in B \) \( x < b \) \( b < \text{min} a y \)
  qed

THEORY "Topological-Spaces"

using cInf-less-iff[of B ∩ {x <..} min a y] (?z < a) (?z < y) (x < y) (y ∈ B)
by auto
moreover have ?z ≤ b
using (b ∈ B) (x < b)
by (intro cInf-lower) auto
moreover have b ∈ U
using (x ≤ ?z) (?z ≤ b) (b < min a y)
by (intro *(OF [x ∈ U; y ∈ U])) (auto simp: less-imp-le)
ultimately show ?thesis
by (intro bexI[of - b]) auto
qed
then have False
using (?z ≤ y) (?z ∈ A) (y ∈ U) (A ∩ B ∩ U = {})
unfolding le-less by blast

} note not-disjoint = this

fix A B assume AB: open A open B U ⊆ A ∪ B A ∩ B ∩ U = {}
moreover assume A ∩ U ≠ {} then obtain x where x: x ∈ U x ∈ A by auto
moreover assume B ∩ U ≠ {} then obtain y where y: y ∈ U y ∈ B by auto
moreover note not-disjoint[of B A y x] not-disjoint[of A B x y]
ultimately show False
by (cases x y rule: linorder-cases) auto
qed

lemma connected-iff-interval: connected U ←→ (∀ x∈U. ∀ y∈U. ∀ z. x ≤ z → z ≤ y → z ∈ U)
for U :: 'a::linear-continuum-topology set
by (auto intro: connectedI-interval dest: connectedD-interval)

lemma connected-UNIV[simp]: connected (UNIV::'a::linear-continuum-topology set)
by (simp add: connected-iff-interval)

lemma connected-Ioi[simp]: connected {a<..}
for a :: 'a::linear-continuum-topology
by (auto simp: connected-iff-interval)

lemma connected-Ici[simp]: connected {a..}
for a :: 'a::linear-continuum-topology
by (auto simp: connected-iff-interval)

lemma connected-Iio[simp]: connected {..<a}
for a :: 'a::linear-continuum-topology
by (auto simp: connected-iff-interval)

lemma connected-Iic[simp]: connected {..a}
for a :: 'a::linear-continuum-topology
by (auto simp: connected-iff-interval)
lemma connected-Ioo[simp]: connected \{a<..<b\}
for a b :: 'a::linear-continuum-topology
unfolding connected-iff-interval by auto

lemma connected-Ioc[simp]: connected \{a<..b\}
for a b :: 'a::linear-continuum-topology
by (auto simp: connected-iff-interval)

lemma connected-Ico[simp]: connected \{a..<b\}
for a b :: 'a::linear-continuum-topology
by (auto simp: connected-iff-interval)

lemma connected-Icc[simp]: connected \{a..b\}
for a b :: 'a::linear-continuum-topology
by (auto simp: connected-iff-interval)

lemma connected-contains-Ioo:
fixes A :: 'a :: linorder-topology set
assumes connected A a \in A b \in A shows \{a<..<b\} \subseteq A
using connectedD-interval[OF assms] by (simp add: subset_eq Ball_def less_imp_le)

lemma connected-contains-Icc:
fixes A :: 'a::linorder-topology set
assumes connected A a \in A b \in A
shows \{a..b\} \subseteq A
proof
  fix x assume x \in \{a..b\}
  then have x = a \lor x = b \lor x \in \{a<..<b\}
    by auto
  then show x \in A
    using assms connected-contains-Ioo[of A a b] by auto
qed

97.1 Intermediate Value Theorem

lemma IVT':
fixes f :: 'a::linear-continuum-topology \Rightarrow 'b::linorder-topology
assumes y: f a \leq y \leq f b a \leq b
and *: continuous-on \{a .. b\} f
shows \exists x. a \leq x \land x \leq b \land f x = y
proof
  have connected \{a..b\}
    unfolding connected-iff-interval by auto
  from connected-continuous-image[OF * this, THEN connectedD-interval, of f a f b y] y
  show ?thesis
    by (auto simp add: atLeastAtMost_def atLeast_def atMost_def)
lemma IVT2':
  fixes f :: 'a :: linear-continuum-topology ⇒ 'b :: linorder-topology
  assumes y: f b ≤ y y ≤ f a a ≤ b
  and *: continuous-on {a .. b} f
  shows ∃ x. a ≤ x ∧ x ≤ b ∧ f x = y
proof –
  have connected {a..b}
    unfolding connected-iff-interval by (auto)
  from connected-continuous-image[OF * this, THEN connectedD-interval, of f b f a y] y
  show ?thesis
  by (auto simp add: atLeastAtMost-def atLeast-def atMost-def)
qed

lemma IVT:
  fixes f :: 'a::linear-continuum-topology ⇒ 'b::linorder-topology
  shows f a ≤ y =⇒ y ≤ f b =⇒ a ≤ b =⇒ (∀ x. a ≤ x ∧ x ≤ b −→ isCont f x)
proof –
  note I = inj-on-eq-iff[OF inj]
  {assume f x < f a a < f x < f b
   then obtain s t where x ≤ s s ≤ b a ≤ t t ≤ x f s = f t f x < f s
   using IVT[of f x min (f a) (f b) b] IVT2[of f x min (f a) (f b) a] x
   by (auto simp: continuous-on-subset[OF cont] less-imp-le)
   with x I have False by auto}
  moreover
  {assume f a < f x f b < f x
   then obtain s t where x ≤ s s ≤ b a ≤ t t ≤ x f s = f t f x < f x
   using IVT[of f a max (f a) (f b) x] IVT2[of f b max (f a) (f b) x] x
by (auto simp: continuous-on-subset[OF cont] less-imp-le)
with x I have False by auto
\}
ultimately show \(?thesis
  using I[of a x] I[of x b] x less-trans[OF x]
  by (auto simp add: le-less less-imp-neq neq-iff)
qed

lemma continuous-at-Sup-mono:
  fixes f :: 'a::{linorder-topology,conditionally-complete-linorder} ⇒ 'b::{linorder-topology,conditionally-complete-linorder}
  assumes mono f
  and cont: continuous (at-left (Sup S)) f
  and S: S ≠ {} bdd-above S
  shows f (Sup S) = (SUP s∈S. f s)
proof (rule antisym)
  have f: (f −−−→ f (Sup S)) (at-left (Sup S))
    using cont unfolding continuous-within .
  show f (Sup S) ≤ (SUP s∈S. f s)
proof cases
  assume Sup S ∈ S
  then show \(?thesis
    by (rule cSUP-upper) (auto intro: bdd-above-image-mono S ⟨mono f⟩)
  next
  assume Sup S \∉ S
  from ⟨S ≠ {}⟩ obtain s where s ∈ S
  by auto
  with ⟨Sup S \∉ S. S have s < Sup S
    unfolding less-le by (blast intro: cSup-upper)
  show \(?thesis
proof (rule ccontr)
  assume ~ \(?thesis
  with order-tendstoD(1)[OF f, of SUP s∈S. f s] obtain b where b < Sup S
  and y: (y, b < y ⇒ y < Sup S ⇒ (SUP s∈S. f s) < f y
    by (auto simp: not-le eventually-at-left[OF ⟨s < Sup S⟩])
  with ⟨S ≠ {}⟩ obtain c where c ∈ S b < c
  using less-cSupD[of S b] by auto
  with ⟨Sup S \∉ S. S have c < Sup S
    unfolding less-le by (blast intro: cSup-upper)
  from ⟨OF ⟨b < c⟩. ⟨c < Sup S⟩⟩ cSUP-upper[OF ⟨c ∈ S⟩ bdd-above-image-mono[of f]]
  show False
  by (auto simp: assms)
  qed
  qed
qed

lemma continuous-at-Sup-antimono:
  fixes f :: 'a::{linorder-topology,conditionally-complete-linorder} ⇒
assumes antimono f
and cont: continuous (at-left \(\operatorname{Sup} S\)) f
and \(S; S \neq \{\}\) bdd-above \(S\)
shows \(f (\operatorname{Sup} S) = (\operatorname{INF} s \in S. \ f s)\)

proof (rule antisym)

have \(f: (f \rightarrow f (\operatorname{Sup} S))\) (at-left \(\operatorname{Sup} S\))
using cont unfolding continuous-within .
show \((\operatorname{INF} s \in S. \ f s) \leq f (\operatorname{Sup} S)\)

proof cases
assume \(\operatorname{Sup} S \in S\)
then show \(?\)thesis
by (intro cINF-lower) (auto intro: bdd-below-image-antimono \(S \langle\operatorname{antimono f}\rangle\))

next
assume \(\operatorname{Sup} S \notin S\)
from \(\langle S \neq \{\}\rangle\) obtain \(s\) where \(s \in S\)
by auto
with \(\langle \operatorname{Sup} S \notin S; S \rangle\) have \(s < \operatorname{Sup} S\)
using less-cSupD \(\langle\operatorname{OF} S \notin S\rangle\)
show \(?\)thesis
proof (rule ccontr)
assume \(\neg ?\)thesis
with order-tends toD \(2\langle\operatorname{OF} f, \langle\operatorname{OF} s \in S. \ f s\rangle\rangle\) obtain \(b\) where \(b < \operatorname{Sup} S\)
and \(\langle\forall y. \ b < y \implies y < \operatorname{Sup} S \implies f y < (\operatorname{INF} s \in S. \ f s)\rangle\)
by (auto simp: not-le eventually-at-left \langle\operatorname{OF} \langle s < \operatorname{Sup} S\rangle\rangle)
with \(\langle S \neq \{\}\rangle\) obtain \(c\) where \(c \in S\) \(b < c\)
using less-cSupD \(\langle\operatorname{OF} S \notin S\rangle\)
by auto
with \(\langle \operatorname{Sup} S \notin S; S \rangle\) have \(c < \operatorname{Sup} S\)
unfolding less-le by (blast intro: cSup-upper)
from \(\langle\operatorname{OF} \langle b < c\rangle; \langle c < \operatorname{Sup} S\rangle\rangle\) cINF-lower \langle\operatorname{OF} \langle\operatorname{bdd-below-image-antimono}, of f S c\rangle; \langle c \in S\rangle\rangle
show False
by (auto simp: assms)
qed

qed (intro cINF-greatest \(\langle\operatorname{antimono f}\rangle\langle\operatorname{THEN antimonoD}\rangle\) cSup-upper \(S\))
assume \( \inf S \in S \)
then show \(?thesis\)
  by (rule c\text{INF}-lower[rotated]) (auto intro: bdd-below-image-mono S \{\text{mono } f\})

next
assume \( \inf S \notin S \)
from \( S \neq \{} \) obtain \( s \) where \( s \in S \)
  by auto
with \( \inf S \notin S \) \( \mathrm{S} \) have \( \inf S < s \)
  unfolding less-le by (blast intro: c\text{Inf}-lower)
show \(?thesis\)

proof (rule c\text{contr})
assume \( \neg \ ?thesis \)
  with \( \text{order-tendstoD}() \) \{OF \( f \) of \( \inf S \in S. f s \)\} obtain \( b \) where \( \inf S < b \)
and \( s \) \( \lambda y. \inf S < y \implies y < b \implies f y < (\inf S \in S. f s) \)
by (auto simp: not-le eventually-at-right (OF \( \inf S < s \)))
with \( S \neq \{} \) obtain \( c \) where \( c \in S \ c < b \)
  using c\text{Inf-lessD}() \{\text{of } S b\} by auto
with \( \inf S \notin S \) \( \mathrm{S} \) have \( \inf S < c \)
  unfolding less-le by (blast intro: c\text{Inf}-lower)
from \( \text{OF } \inf S < c \) \( \{c < b\} \) c\text{INF}-lower \{OF bdd-below-image-mono[of ]f\}
\( c \in S \)
  show False
  by (auto simp: assms)
qed

qed (intro c\text{INF}-greatest \{\text{mono } f\}[THEN monoD] c\text{Inf}-lower \{bdd-below S \}:\{S \neq \{}\})

lemma continuous-at-\text{Inf-antimono}:
fixes \( f :: 'a::{\text{linorder-topology,conditionally-complete-linorder}} \Rightarrow 'b::{\text{linorder-topology,conditionally-complete-linorder}} \)
assumes antimono \( f \)
  and cont: continuous (at-right \( (\inf S) \)) \( f \)
  and \( S :: S \neq \{} \) bdd-below \( S \)
shows \( f \ (\inf S) \) = (SUP \( s \in S. f s \))

proof (rule antisym)
  have \( f :: (f \longrightarrow f \ (\inf S)) \) (at-right \( (\inf S) \))
    using cont unfolding continuous-within .
  show \( f \ (\inf S) \) \( \leq \) (SUP \( s \in S. f s \))
  proof cases
    assume \( \inf S \in S \)
    then show \(?thesis\)
      by (rule cSUP-upper) (auto intro: bdd-above-image-antimono S \antimono{ f\})
  next
    assume \( \inf S \notin S \)
    from \( S \neq \{} \) obtain \( s \) where \( s \in S \)
      by auto
    with \( \inf S \notin S \) \( \mathrm{S} \) have \( \inf S < s \)
      unfolding less-le by (blast intro: c\text{Inf}-lower)
show thesis

proof (rule ccontr)

assume ¬thesis

with order-tendstoD[1](f, of SUP s∈S. f s] obtain b where Inf S < b

and *: ∀y. Inf S < y → y < b → (SUP s∈S. f s) < f y

by (auto simp; not-le eventually-at-right[OF (Inf S < s)])

with S ≠ {} obtain c where c ∈ S c < b

using cInf-lessD[of S b] by auto

with (Inf S ∉ S) S have Inf S < c

unfolding less-le by (blast intro: cInf-lower)

from *|OF (Inf S < c) (c < b) cSUP-upper|OF (c ∈ S) bdd-above-image-antimono[of f]

show False

by (auto simp: assms)

qed

qed (intro cSUP-least ⟨antimono f⟩[THEN antimonoD] cInf-lower S)

97.2 Uniform spaces

class uniformity =

  fixes uniformity :: ('a × 'a) filter

begin

abbreviation uniformity-on :: 'a set ⇒ ('a × 'a) filter

  where uniformity-on s ≡ inf uniformity (principal (s×s))

end

lemma uniformity-Abort:

  uniformity =

    Filter.abstract-filter (λu. Code.abort (STR "uniformity is not executable") (λu. uniformity))

  by simp

class open-uniformity = open + uniformity +

assumes open-uniformity:

  ∀U. open U ←→ (∀x∈U. eventually (λ(x', y), x' = x → y ∈ U) uniformity)

begin

subclass topological-space

  by standard (force elim: eventually-mono eventually-elim2 simp: split-beta' open-uniformity)+

end

class uniform-space = open-uniformity +

assumes uniformity-refl: eventually E uniformity ⇒ E (x, x)

  and uniformity-sym: eventually E uniformity ⇒ eventually (λ(x, y), E (y, x)) uniformity
and uniformity-trans:

\[ \text{eventually } E \text{ uniformity} \implies \exists D. \text{ eventually } D \text{ uniformity} \land (\forall x y z. D(x, y) \implies D(y, z) \implies E(x, z)) \]

begin

lemma uniformity-bot: uniformity \neq bot
using uniformity-refl by auto

lemma uniformity-trans':

\[ \text{eventually } E \text{ uniformity} \implies \text{ eventually } (\lambda((x, y), (y', z)). y = y' \implies E(x, z)) \text{ (uniformity } \times F \text{ uniformity}) \]

by (drule uniformity-trans) (auto simp add: eventually-prod-same)

lemma uniformity-transE:

assumes eventually E uniformity
obtains D where eventually D uniformity \land (\forall x y z. D(x, y) = \implies D(y, z) = \implies E(x, z))

using uniformity-trans \[ OF \text{ assms} \] by auto

lemma eventually-nhds-uniformity:

\[ \text{eventually } P(\text{nhds } x) \iff \text{ eventually } (\lambda(x', y). x' = x \implies P y) \text{ uniformity} \]

(is - \iff ?N P x)

unfolding eventually-nhds

proof safe
assume *: ?N P x
have ?N (?N P) x if ?N P x for x
proof -
from that obtain D where ev: eventually D uniformity
and D: D(a, b) \implies D(b, c) \implies case (a, c) of (x', y) \implies x' = x \implies P y
for a b c
by (rule uniformity-transE) simp
from ev show ?thesis
by eventually-elim (insert ev D, force elim: eventually-mono split: prod.split)
qed

then have open \{ x. ?N P x \}
by (simp add: open-uniformity)
then show \exists S. open S \land x \in S \land (\forall x \in S. P x)
by (intro exI[of - \{ x. ?N P x \}]) (auto dest: uniformity-refl simp: *)
qed (force simp add: open-uniformity elim: eventually-mono)

97.2.1 Totally bounded sets

definition totally-bounded :: 'a set \Rightarrow bool
where totally-bounded S \iff
(\forall E. \text{ eventually } E \text{ uniformity} \implies (\exists X. \text{ finite } X \land (\forall s \in S. \exists x \in X. E(x, s))))

lemma totally-bounded-empty[iff]: totally-bounded {}
by (auto simp add: totally-bounded-def)
lemma totally-bounded-subset: totally-bounded $S \implies T \subseteq S \implies$ totally-bounded $T$
  by (fastforce simp add: totally-bounded-def)

lemma totally-bounded-Union[intro]:
  assumes $M$: finite $M \land S. S \in M \implies$ totally-bounded $S$
  shows totally-bounded $(\bigcup M)$
  unfolding totally-bounded-def
proof safe
  fix $E$
  assume eventually $E$ uniformity
  with $M$ obtain $X$ where \begin{itemize}
    \item $\forall S \in M. \text{finite} (X S) \land (\forall s \in S. \exists x \in X S. E (x, s))$
    \item (metis totally-bounded-def)
  \end{itemize}
  with \begin{itemize}
    \item finite $M$
    \item show $\exists X. \text{finite} X \land (\forall s \in \bigcup M. \exists x \in X S. E (x, s))$
  \end{itemize}
  by (intro exI[of - $\bigcup S \in M. X S$]) force
qed

97.2.2 Cauchy filter

definition cauchy-filter :: 'a filter $\Rightarrow$ bool
  where cauchy-filter $F \iff F \times F F \leq$ uniformity

definition Cauchy :: ('nat $\Rightarrow$ 'a) $\Rightarrow$ bool
  where Cauchy-uniform: Cauchy $X \iff$ cauchy-filter ($\text{filtermap} X \text{ sequentially}$)

lemma Cauchy-uniform-iff:
  Cauchy $X \iff (\forall P. \text{eventually} P \text{ uniformity} \implies (\exists N. \forall n \geq N. \forall m \geq N. P (X n, X m)))$
  unfolding Cauchy-uniform cauchy-filter-def le-filter-def eventually-prod-same eventually-filtermap eventually-sequentially
proof safe
  let $?U = \lambda P. \text{eventually} P \text{ uniformity} \\{ 
  \begin{align*}
    \text{fix } P \\
    \text{assume } ?U P \forall P. ?U P \implies (\exists Q. (\exists N. \forall n \geq N. Q (X n)) \land (\forall x y. Q x \implies Q y \implies P (x, y)))
  \end{align*}
  \text{then obtain } Q N \text{ where } \land n. n \geq N \implies Q (X n) \land x y. Q x \implies Q y \implies P (x, y) \text{ by metis} \\
  \text{then show } \exists N. \forall n \geq N. \forall m \geq N. P (X n, X m) \text{ by blast}
  \text{next}
  \text{fix } P \\
  \text{assume } ?U P \text{ and } P; \forall P. ?U P \implies (\exists N. \forall n \geq N. \forall m \geq N. P (X n, X m))
  \text{then obtain } Q \text{ where } ?U Q \text{ and } Q; \land x y z. Q (x, y) \implies Q (y, z) \implies P (x, z) \text{ by (auto elim: uniformity-transE)}
  \text{then have } ?U (\lambda x. Q x \land (\lambda(x, y). Q (y, x)) x)
THEORY "Topological-Spaces"

lemma Cauchy-subseq-Cauchy:
  assumes Cauchy X strict-mono f
  shows Cauchy (X o f)
  unfolding Cauchy-uniform comp-def filtermap-filtermap[symmetric] cauchy-filter-def
  by (rule order-trans[OF - Cauchy X[unfolded Cauchy-uniform cauchy-filter-def]])
      (intro prod-filter-mono filtermap-mono filterlim-subseq[OF strict-mono f]),
lemma convergent-Cauchy: convergent X \implies Cauchy X

unfolding convergent-def by (erule exE, erule LIMSEQ-imp-Cauchy)

definition complete :: 'a set \Rightarrow bool
where complete-uniform: complete S \iff
(\forall F \leq principal S. F \neq bot \implies cauchy-filter F \implies (\exists x \in S. F \leq nhds x))

end

97.2.3 Uniformly continuous functions

definition uniformly-continuous-on :: 'a set \Rightarrow ('a::uniform-space \Rightarrow 'b::uniform-space) \Rightarrow bool
where uniformly-continuous-on-uniformity: uniformly-continuous-on s f \iff
(LIM (x, y) (uniformity-on s). (f x, f y) :> uniformity)

lemma uniformly-continuous-onD:
uniformly-continuous-on s f \implies eventually E uniformity \implies
eventually (\lambda (x, y). x \in s \rightarrow y \in s \rightarrow E (f x, f y)) uniformity
by (simp add: uniformly-continuous-on-uniformity filterlim-iff
eventually-inf-principal split-beta' mem-Times-iff imp-conjL)

lemma uniformly-continuous-on-const[continuous-intros]: uniformly-continuous-on s (\lambda x. c)
by (auto simp: uniformly-continuous-on-uniformity filterlim-iff uniformity-refl)

lemma uniformly-continuous-on-id[continuous-intros]: uniformly-continuous-on s (\lambda x. x)
by (auto simp: uniformly-continuous-on-uniformity filterlim-def)

lemma uniformly-continuous-on-compose[continuous-intros]:
uniformly-continuous-on s g \implies uniformly-continuous-on (g's) f \implies
uniformly-continuous-on s (\lambda x. f (g x))
using filterlim-compose[of \lambda (x, y). (f x, f y) uniformity
uniformity-on (g's) \lambda (x, y). (g x, g y) uniformity-on s]
by (simp add: split-beta' uniformly-continuous-on-uniformity
filterlim-inf filterlim-principal eventually-inf-principal mem-Times-iff)

lemma uniformly-continuous-imp-continuous:
assumes f: uniformly-continuous-on s f
shows continuous-on s f
by (auto simp: filterlim-iff eventually-at-filter eventually-nhds-uniformity continuous-on-def
elim: eventually-mono dest!: uniformly-continuous-onD[OF f])
98 Product Topology

98.1 Product is a topological space

instantiation prod :: (topological-space, topological-space) topological-space
begin

definition open-prod-def[code def]:
  open (S :: ('a × 'b) set) ─→
  (∀ x∈S. ∃ A B. open A ∧ open B ∧ x ∈ A × B ∧ A × B ⊆ S)

lemma open-prod-elim:
  assumes open S and x ∈ S
  obtains A B where open A and open B and x ∈ A × B and A × B ⊆ S
  using assms unfolding open-prod-def by fast

lemma open-prod-intro:
  assumes ∃x. x ∈ S =⇒ ∃ A B. open A ∧ open B ∧ x ∈ A × B ∧ A × B ⊆ S
  shows open S
  using assms unfolding open-prod-def by fast

instance
proof
  show open (UNIV :: ('a × 'b) set)
    unfolding open-prod-def by auto
next
fix S T :: ('a × 'b) set
assume open S open T
show open (S ∩ T)
proof (rule open-prod-intro)
  fix x
  assume x: x ∈ S ∩ T
  from x have x ∈ S by simp
  obtain Sa Sb where A: open Sa open Sb x ∈ Sa × Sb Sa × Sb ⊆ S
    using (open S) and (x ∈ S) by (rule open-prod-elim)
  from x have x ∈ T by simp
  obtain Ta Tb where B: open Ta open Tb x ∈ Ta × Tb Ta × Tb ⊆ T
    using (open T) and (x ∈ T) by (rule open-prod-elim)
  let ?A = Sa ∩ Ta and ?B = Sb ∩ Tb
    using A B by (auto simp add: open-Int)
  then show ∃ A B. open A ∧ open B ∧ x ∈ A × B ∧ A × B ⊆ S ∩ T
    by fast
qed
next
fix K :: ('a × 'b) set set
assume ∀S∈K. open S
then show open (⋃K)
  unfolding open-prod-def by fast
qed
declare [[code abort: open :: ('a::topological-space × 'b::topological-space) set ⇒ bool]]

lemma open-Times: open S ⇒ open T ⇒ open (S × T)
  unfolding open-prod-def by auto

lemma fst-vimage-eq-Times: fst −¹ S = S × UNIV
  by auto

lemma snd-vimage-eq-Times: snd −¹ S = UNIV × S
  by auto

lemma open-vimage-fst: open S ⇒ open (fst −¹ S)
  by (simp add: fst-vimage-eq-Times open-Times)

lemma open-vimage-snd: open S ⇒ open (snd −¹ S)
  by (simp add: snd-vimage-eq-Times open-Times)

lemma closed-vimage-fst: closed S ⇒ closed (fst −¹ S)
  unfolding closed-open vimage-Compl [symmetric]
  by (rule open-vimage-fst)

lemma closed-vimage-snd: closed S ⇒ closed (snd −¹ S)
  unfolding closed-open vimage-Compl [symmetric]
  by (rule open-vimage-snd)

lemma closed-Times: closed S ⇒ closed T ⇒ closed (S × T)
proof –
  have S × T = (fst −¹ S) ∩ (snd −¹ T)
    by auto
  then show closed S ⇒ closed T ⇒ closed (S × T)
    by (simp add: closed-vimage-fst closed-vimage-snd closed-Int)
qed

lemma subset-fst-imageI: A × B ⊆ S ⇒ y ∈ B ⇒ A ⊆ fst −¹ S
  unfolding image-def subset-eq by force

lemma subset-snd-imageI: A × B ⊆ S ⇒ x ∈ A ⇒ B ⊆ snd −¹ S
  unfolding image-def subset-eq by force

lemma open-image-fst:
  assumes open S
  shows open (fst −¹ S)
proof (rule openI)
  fix x
  assume x ∈ fst −¹ S
then obtain \( y \) where \((x, y) \in S\)
  by auto
then obtain \( A B \) where open \( A \) open \( B \) \( x \in A \ y \in B \) \( A \times B \subseteq S \)
  using (\( \text{open S} \)) unfolding open-prod-def by auto
from \( (A \times B \subseteq S) \) \( y \in B \) have \( A \subseteq \text{fst} \ ' \ S \)
  by (rule subset-fst-imageI)
with (\( \text{open A} \)) \( \langle x \in A \rangle \) have \( A \land x \in A \land A \subseteq \text{fst} \ ' \ S \)
  by simp
then show \( \exists \ T. \) \( \text{open} \ T \land x \in T \land T \subseteq \text{fst} \ ' \ S \) ..
q.e.d.

lemma open-image-snd:
  assumes open \( S \)
  shows open (\( \text{snd} \ ' \ S \))
proof (rule openI)
  fix \( y \)
  assume \( y \in \text{snd} \ ' \ S \)
  then obtain \( x \) where \((x, y) \in S\)
    by auto
  then obtain \( A B \) where open \( A \) open \( B \) \( x \in A \ y \in B \) \( A \times B \subseteq S \)
    using (\( \text{open S} \)) unfolding open-prod-def by auto
from (\( A \times B \subseteq S \)) \( x \in A \) have \( B \subseteq \text{snd} \ ' \ S \)
  by (rule subset-snd-imageI)
with (\( \text{open B} \)) \( \langle y \in B \rangle \) have \( B \land y \in B \land B \subseteq \text{snd} \ ' \ S \)
  by simp
then show \( \exists \ T. \) \( \text{open} \ T \land y \in T \land T \subseteq \text{snd} \ ' \ S \) ..
q.e.d.

lemma nhds-prod: nhds \((a, b)\) = nhds \(a \times F\) nhds \(b\)
unfolding nhds-def
proof (subst prod-filter-INF, auto intro!: antisym INF-greatest simp: principal-prod-principal)
  fix \( S \ T \)
  assume open \( S \) \( a \in S \) open \( T \) \( b \in T \)
  then show (\( \text{INF} \ x \in \{ S. \ \text{open} \ S \land (a, b) \in S \} \). \( \text{principal} \ x \) \( \leq \) principal \( S \times T \))
    by (intro INF-lower) (auto intro!: open-Times)
next
  fix \( S' \)
  assume open \( S' \) \( (a, b) \in S' \)
  then obtain \( S \ T \) where open \( S \) \( a \in S \) open \( T \) \( b \in T \) \( S \times T \subseteq S' \)
    by (auto elim: open-prod-elim)
  then show (\( \text{INF} \ x \in \{ S. \ \text{open} \ S \land a \in S \} \). \( \text{INF} \ y \in \{ S. \ \text{open} \ S \land b \in S \} \). \( \text{principal} \ (x \times y) \) \( \leq \) principal \( S' \))
    by (auto intro!: INF-lower2)
q.e.d.

98.1.1 Continuity of operations

lemma tendsto-fst [tendsto-intros]:
assumes \((f \to a)\) \(F\)
shows \(((\lambda x. \text{fst } (f \, x)) \to \text{fst } a)\) \(F\)
proof (rule topological-tendstoI)
fix \(S\)
assume open \(S\) and \(\text{fst } a \in S\)
then have open \((\text{fst } - ' S)\) and \(a \in \text{fst } - ' S\)
by (simp-all add: open-vimage-fst)
with assms have eventually \((\lambda x \, f \, x \in \text{fst } - ' S)\) \(F\)
by (rule topological-tendstoD)
then show eventually \((\lambda x \, \text{fst } (f \, x) \in S)\) \(F\)
by simp
qed

lemma tendsto-snd [tendsto-intros]:
assumes \((f \to a)\) \(F\)
shows \(((\lambda x. \text{snd } (f \, x)) \to \text{snd } a)\) \(F\)
proof (rule topological-tendstoI)
fix \(S\)
assume open \(S\) and \(\text{snd } a \in S\)
then have open \((\text{snd } - ' S)\) and \(a \in \text{snd } - ' S\)
by (simp-all add: open-vimage-snd)
with assms have eventually \((\lambda x \, f \, x \in \text{snd } - ' S)\) \(F\)
by (rule topological-tendstoD)
then show eventually \((\lambda x \, \text{snd } (f \, x) \in S)\) \(F\)
by simp
qed

lemma tendsto-Pair [tendsto-intros]:
assumes \((f \to a)\) \(F\) and \((g \to b)\) \(F\)
shows \(((\lambda x \, (f \, x, g \, x)) \to (a, b))\) \(F\)
unfolding nlds-prod using assms by (rule filterlim-Pair)

lemma continuous-fst [continuous-intros]: continuous \(F\) \(f\) \(\Rightarrow\) continuous \(F\) \((\lambda x. \text{fst } \, (f \, x))\)
unfolding continuous-def by (rule tendsto-fst)

lemma continuous-snd [continuous-intros]: continuous \(F\) \(f\) \(\Rightarrow\) continuous \(F\) \((\lambda x. \text{snd } \, (f \, x))\)
unfolding continuous-def by (rule tendsto-snd)

lemma continuous-Pair [continuous-intros]:
continuous \(F\) \(f\) \(\Rightarrow\) continuous \(F\) \(g\) \(\Rightarrow\) continuous \(F\) \((\lambda x. \, (f \, x, g \, x))\)
unfolding continuous-def by (rule tendsto-Pair)

lemma continuous-on-fst [continuous-intros]:
continuous-on \(s\) \(f\) \(\Rightarrow\) continuous-on \(s\) \((\lambda x. \, \text{fst } \, (f \, x))\)
unfolding continuous-on-def by (auto intro: tendsto-fst)

lemma continuous-on-snd [continuous-intros]:
THEORY "Topological-Spaces"

continuous-on s f \implies \text{continuous-on } s \, (\lambda x. \, \text{snd} \,(f \, x))

\textbf{unfolding} \text{continuous-on-def by (auto intro: tendsto-snd)}

\textbf{lemma} \text{continuous-on-Pair[continuous-intros]}:

\text{continuous-on } s \, f \implies \text{continuous-on } s \, g \implies \text{continuous-on } s \, (\lambda x. \, (f \, x, \, g \, x))

\textbf{unfolding} \text{continuous-on-def by (auto intro: tendsto-Pair)}

\textbf{lemma} \text{continuous-on-swap[continuous-intros]}: \text{continuous-on } A \, \text{prod.swap}

by (simp add: \text{prod.swap-def} \text{continuous-on-fst} \text{continuous-on-snd} \\
\text{continuous-on-Pair} \text{ continuous-on-id})

\textbf{lemma} \text{continuous-on-swap-args:}

\textbf{assumes} \text{continuous-on } (A \times B) \, (\lambda (x,y). \, d \, x \, y)

\textbf{shows} \text{continuous-on } (B \times A) \, (\lambda (x,y). \, d \, y \, x)

\textbf{proof} –

have \((\lambda (x,y). \, d \, y \, x) = (\lambda (x,y). \, d \, x \, y) \circ \text{prod.swap}

by force

\textbf{then show} ?thesis

by (metis \text{assms} \text{continuous-on-compose} \text{continuous-on-swap} \text{product-swap})

\textbf{qed}

\textbf{lemma} \text{isCont-fst [simp]}: \text{isCont } f \, a \implies \text{isCont } (\lambda x. \, \text{fst} \,(f \, x)) \, a

by (fact \text{continuous-fst})

\textbf{lemma} \text{isCont-snd [simp]}: \text{isCont } f \, a \implies \text{isCont } (\lambda x. \, \text{snd} \,(f \, x)) \, a

by (fact \text{continuous-snd})

\textbf{lemma} \text{isCont-Pair [simp]}: \,[\text{isCont } f \, a; \text{isCont } g \, a] \implies \text{isCont } (\lambda x. \,(f \, x, \, g \, x)) \, a

by (fact \text{continuous-Pair})

\textbf{lemma} \text{continuous-on-compose-Pair:}

\textbf{assumes} \, f: \text{continuous-on } (\Sigma A \, B) \, (\lambda (a,b). \, f \, a \, b)

\textbf{assumes} \, g: \text{continuous-on } C \, g

\textbf{assumes} \, h: \text{continuous-on } C \, h

\textbf{assumes} \, \text{subset:} \, \forall c. \, c \in C \implies g \, c \in A \wedge (c. \, c \in C \implies h \, c \in B \,(g \, c))

\textbf{shows} \text{continuous-on } C \, (\lambda c. \, f \,(g \, c) \,(h \, c))

\textbf{using} \text{continuous-on-compose2[\text{OF } f \text{ continuous-on-Pair}][\text{OF } g \, h]} \, \text{subset}

by auto

\subsection{Connectedness of products}

\textbf{proposition} \text{connected-Times:}

\textbf{assumes} \, S: \text{connected } S \text{ and } T: \text{connected } T

\textbf{shows} \text{connected } (S \times T)

\textbf{proof} (rule \text{connectedI-const})

\textbf{fix} \, P::'a \times 'b \Rightarrow \text{bool}

\textbf{assume} \, P[\text{THEN } \text{continuous-on-compose2}, \text{continuous-intros}]: \text{continuous-on } (S \times T) \, P

\textbf{have} \text{continuous-on } S \,(\lambda s. \, P \,(s, \, t)) \text{ if } t \in T \text{ for } t
by (auto intro!: continuous-intros that)
from connectedD-const[OF S this]
obtain c1 where c1: \(\forall s\ t. t \in T \implies s \in S \implies P (s, t) = c1\ t\)
  by metis
moreover
have continuous-on T (\(\lambda t. P (s, t)\)) if \(s \in S\) for \(s\)
  by (auto intro!: continuous-intros that)
from connectedD-const[OF T this]
obtain c2 where \(\forall s\ t. t \in T \implies s \in S \implies P (s, t) = c2\ s\)
  by metis
ultimately show \(\exists c. \forall s \in S \times T. P s = c\)
  by auto
qed

**corollary** connected-Times-eq [simp]:
  \(\text{connected} (S \times T) \iff S = \{\} \lor T = \{\} \lor \text{connected} S \land \text{connected} T\) (is ?lhs = ?rhs)
proof
  assume L: ?lhs
  show ?rhs
  proof cases
    assume \(S \neq \{\} \land T \neq \{\}\)
    moreover
    have connected (fst ' (S \times T)) connected (snd ' (S \times T))
      using continuous-on-fst continuous-on-snd continuous-on-id
      by (blast intro: connected-continuous-image [OF - L])+
    ultimately show ?thesis
      by auto
  qed auto
qed (auto simp: connected-Times)

### 98.1.3 Separation axioms

**instance** prod :: \((t0-space, t0-space)\) t0-space
proof
  fix \(x\ y::'a \times 'b\)
  assume \(x \neq y\)
  then have \(\text{fst} x \neq \text{fst} y \lor \text{snd} x \neq \text{snd} y\)
    by (simp add: prod_eq_iff)
  then show \(\exists U. \text{open} U \land (x \in U) \neq (y \in U)\)
    by (fast dest: t0-space elim: open_vimage_fst open_vimage_snd)
qed

**instance** prod :: \((t1-space, t1-space)\) t1-space
proof
  fix \(x\ y::'a \times 'b\)
  assume \(x \neq y\)
  then have \(\text{fst} x \neq \text{fst} y \lor \text{snd} x \neq \text{snd} y\)
    by (simp add: prod_eq_iff)
then show $\exists U. \text{open } U \land x \in U \land y \notin U$
by { (fast dest: t1-space elim: open-vimage-fst open-vimage-snd) }
qed

instance prod :: (t2-space, t2-space) t2-space
proof
fix $x \cdot y :: 'a \times 'b$
assume $x \neq y$
then have $\text{fst } x \neq \text{fst } y \lor \text{snd } x \neq \text{snd } y$
by { simp add: prod-eq-iff }
then show $\exists U \cdot V. \text{open } U \land \text{open } V \land x \in U \land y \in V \land U \cap V = \{\}$
by { (fast dest: hausdorff elim: open-vimage-fst open-vimage-snd) }
qed

lemma isCont-swap[continuous-intros]: isCont prod.swap a
using continuous-on-eq-continuous-within continuous-on-swap by blast

lemma open-diagonal-complement:
open $\{(x,y) \mid x \cdot y. x \neq (y::'a::t2-space)\}$
proof
have $\text{open } \{(x,y). x \neq (y::'a)\}$
unfolding split-def by { intro open-Collect-neq continuous-intros }
also have $\{(x,y). x \neq (y::'a)\} = \{(x,y) \mid x \cdot y. x \neq (y::'a)\}$
by auto
finally show $?thesis$.
qed

lemma closed-diagonal:
closed $\{y. \exists x::(a::t2-space). y = (x,x)\}$
proof
have $\{y. \exists x::'a. y = (x,x)\} = \text{UNIV} - \{(x,y) \mid x \cdot y. x \neq y\}$ by auto
then show $?thesis$ using open-diagonal-complement closed-Diff by auto
qed

lemma open-superdiagonal:
open $\{(x,y) \mid x \cdot y. x > (y::'a::linorder-topology)\}$
proof
have $\{x,y. x > (y::'a)\}$
unfolding split-def by { intro open-Collect-less continuous-intros }
also have $\{(x,y). x > (y::'a)\} = \{(x,y) \mid x \cdot y. x > (y::'a)\}$
by auto
finally show $?thesis$.
qed

lemma closed-subdiagonal:
closed $\{(x,y) \mid x \cdot y. x \leq (y::'a::linorder-topology)\}$
proof
have $\{(x,y) \mid x \cdot y. x \leq (y::'a)\} = \text{UNIV} - \{(x,y) \mid x \cdot y. x > (y::'a)\}$ by auto
then show $?thesis$ using open-superdiagonal closed-Diff by auto
theory "Hull"

import Main

begin

98.2  A generic notion of the convex, affine, conic hull, or closed "hull".

definition hull :: ('a set ⇒ bool) ⇒ 'a set ⇒ 'a set  (infixl hull 75)  
where  S hull s = ⋂ { t. S t ∧ s ⊆ t }

lemma hull-same: S s ⇒ S hull s = s
  unfolding hull-def by auto

lemma hull-in: (∧ T. Ball T S ⇒ S (∩ T)) ⇒ S (S hull s)
  unfolding hull-def Ball-def by auto

lemma hull-eq: (∧ T. Ball T S ⇒ S (∩ T)) ⇒ (S hull s) = s ⇔ S s
  using hull-same[of S s] hull-in[of S s] by metis

lemma hull-hull [simp]: S hull (S hull s) = S hull s
  unfolding hull-def by blast

lemma hull-subset[intro]: s ⊆ (S hull s)
  unfolding hull-def by blast

lemma hull-mono: s ⊆ t ⇒ (S hull s) ⊆ (S hull t)
  unfolding hull-def by blast

end

THEORY "Hull"
theory "Hull"

lemma hull-antimono: \( \forall x. S x \rightarrow T x \rightarrow (T \text{ hull } s) \subseteq (S \text{ hull } s) \)
unfolding hull-def by blast

lemma hull-minimal: \( s \subseteq t \rightarrow S t \rightarrow (S \text{ hull } s) \subseteq t \)
unfolding hull-def by blast

lemma subset-hull: \( S t \rightarrow S \text{ hull } s \subseteq t \leftarrow s \subseteq t \)
unfolding hull-def by blast

lemma hull-UNIV [simp]: \( S \text{ hull } \text{UNIV} = \text{UNIV} \)
unfolding hull-def by auto

lemma hull-unique: \( s \subseteq t \rightarrow S t \rightarrow ( \forall t'. s \subseteq t' \rightarrow S t' \rightarrow t \subseteq t') \rightarrow (S \text{ hull } s = t) \)
unfolding hull-def by auto

lemma hull-induct: \[ \[ a \in Q \text{ hull } S; \forall x. x \in S \rightarrow P x; Q \{ x. P x \} \] \] \rightarrow P a
using hull-minimal[of S \{ x. P x \} Q]
by (auto simp add: subset-eq)

lemma hull-inc: \( x \in S \rightarrow x \in P \text{ hull } S \)
by (metis hull-subset subset-eq)

lemma hull-Un-subset: \( (S \text{ hull } s) \cup (S \text{ hull } t) \subseteq (S \text{ hull } (s \cup t)) \)
unfolding Un-subset-iff by (metis hull-mono Un-upper1 Un-upper2)

lemma hull-Un:\[ a \in Q \text{ hull } S; \forall T. \text{ Ball } T S \rightarrow S (\cap T) \]
shows \( S \text{ hull } (s \cup t) = S \text{ hull } (S \text{ hull } s \cup S \text{ hull } t) \)
apply (rule equalityI)
apply (meson hull-mono hull-subset sup mono)
by (metis hull-Un-subset hull-hull hull-mono)

lemma hull-Un-left: \( P \text{ hull } (S \cup T) = P \text{ hull } (P \text{ hull } S \cup T) \)
apply (rule equalityI)
apply (simp add: Un-commute hull-mono hull-subset sup.coboundedI2)
by (metis Un-subset-iff hull-hull hull-mono hull-subset)

lemma hull-Un-right: \( P \text{ hull } (S \cup T) = P \text{ hull } (S \cup P \text{ hull } T) \)
by (metis hull-Un-left sup.commute)

lemma hull-insert: \( P \text{ hull } (\text{insert } a S) = P \text{ hull } (\text{insert } a (P \text{ hull } S)) \)
by (metis hull-Un-right insert-is-Un)

lemma hull-redundant-eq: \( a \in (S \text{ hull } s) \leftarrow S \text{ hull } (\text{insert } a s) = S \text{ hull } s \)
unfolding hull-def by blast
99 Modules

Bases of a linear algebra based on modules (i.e. vector spaces of rings).

locale additive = 
  fixes f :: 'a::ab-group-add ⇒ 'b::ab-group-add 
  assumes add: f (x + y) = f x + f y
begin

lemma zero: f 0 = 0
proof -
  have f 0 = f (0 + 0) by simp
  also have ... = f 0 + f 0 by (rule add)
  finally show f 0 = 0 by simp
qed

lemma minus: f (− x) = − f x
proof -
  have f (− x) + f x = f (− x + x) by (rule add [symmetric])
  also have ... = − f x + f x by (simp add: zero)
  finally show f (− x) = − f x by (rule add-right-imp-eq)
qed

lemma diff: f (x − y) = f x − f y
  using add [of x − y] by (simp add: minus)

lemma sum: f (sum g A) = (∑ x∈A. f (g x))
  by (induct A rule: infinite-finite-induct) (simp-all add: zero add)
end

Modules form the central spaces in linear algebra. They are a generalization from vector spaces by replacing the scalar field by a scalar ring.

locale module =
  fixes scale :: 'a::comm-ring-1 ⇒ 'b::ab-group-add ⇒ 'b (infixr "∗")
  assumes scale-right-distrib [algebra-simps, algebra-split-simps]:
    a ∗ s (x + y) = a ∗ s x + a ∗ s y
  and scale-left-distrib [algebra-simps, algebra-split-simps]:
THEORY "Modules"

\[(a + b) \ast s x = a \ast s x + b \ast s x\]

and \textit{scale-scale} [simp]: \(a \ast s (b \ast s x) = (a \ast b) \ast s x\)

and \textit{scale-one} [simp]: \(1 \ast s x = x\)

begin

\textbf{lemma} \textit{scale-left-commute}: \(a \ast s (b \ast s x) = b \ast s (a \ast s x)\)

\textit{by} (simp add: mult.commute)

\textbf{lemma} \textit{scale-zero-left} [simp]: \(0 \ast s x = 0\)

and \textit{scale-minus-left} [simp]: \((- a) \ast s x = -(a \ast s x)\)

and \textit{scale-left-diff-distrib} [algebra-simps, algebra-split-simps]:

\(\sum a \ast s x = a \ast s x - b \ast s x\)

and \textit{scale-sum-left} [sum f A]: \(a \ast s x = (\sum a \ast s x - b \ast s x)\)

\textbf{proof} -

interpret s: additive \(\lambda a. a \ast s x\)

by standard (rule scale-left-distrib)

show \(0 \ast s x = 0\) \textit{by} (rule s.zero)

show \((- a) \ast s x = -(a \ast s x)\) \textit{by} (rule s.minus)

show \((a - b) \ast s x = a \ast s x - b \ast s x\) \textit{by} (rule s.diff)

\textbf{lemma} \textit{scale-zero-right} [simp]: \(a \ast s 0 = 0\)

and \textit{scale-minus-right} [simp]: \(a \ast s (- x) = -(a \ast s x)\)

and \textit{scale-right-diff-distrib} [algebra-simps, algebra-split-simps]:

\(a \ast s (x - y) = a \ast s x - a \ast s y\)

and \textit{scale-sum-right} [sum f A]: \(a \ast s (\sum f A) = (\sum f A) \ast s (f x)\)

\textbf{proof} -

interpret s: additive \(\lambda x. a \ast s x\)

by standard (rule scale-right-distrib)

show \(a \ast s 0 = 0\) \textit{by} (rule s.zero)

show \(a \ast s (- x) = -(a \ast s x)\) \textit{by} (rule s.minus)

show \(a \ast s (x - y) = a \ast s x - a \ast s y\) \textit{by} (rule s.diff)

\textbf{lemma} \textit{sum-constant-scale}: \(\sum x \in A. y = \text{scale (of-nat (card A))} y\)

\textit{by} (induct A rule: infinite-finite-induct) (simp-all add: algebra-simps)

end

\textbf{setup} (Sign.add-const-constraint (\texttt{const-name} divide), SOME \texttt{typ} 'a \Rightarrow 'a \Rightarrow 'a))

\textbf{context module}

begin

\textbf{lemma} [field-simps, field-split-simps]:

\textit{shows} \textit{scale-left-distrib-NO-MATCH} [NO-MATCH]: \((x \div y) c \Rightarrow (a + b) \ast s x\)
\[ a \ast s x + b \ast s y = (a \div y) \ast (x + y) = a \ast s x + a \ast s y \]

and \textbf{scale-right-distrib-NO-MATCH}: \textbf{NO-MATCH} \hspace{1em} (x \div y) \ast (x + y) \hspace{1em} \Rightarrow \hspace{1em} a \ast s x + a \ast s y \]

and \textbf{scale-left-diff-distrib-NO-MATCH}: \textbf{NO-MATCH} \hspace{1em} (x \div y) \ast (c \div (a - b)) \hspace{1em} \Rightarrow \hspace{1em} c \ast s (x - y) \hspace{1em} = \hspace{1em} a \ast s x - a \ast s y \]

by \textbf{(rule scale-left-distrib scale-right-distrib scale-left-diff-distrib scale-right-diff-distrib)}+ end

setup \langle \textbf{Sign.add-const-constraint (const-name divide SOME typ 'a::divide ⇒ 'a ⇒ 'a)} \rangle

\begin{section}{100 Subspace}

\textbf{context module begin}

\textbf{definition} \texttt{subspace :: 'b set ⇒ bool}

\textbf{where} \texttt{subspace S} \hspace{1em} \iff \hspace{1em} 0 \in S \land (\forall x \in S . \forall y \in S . x + y \in S) \land (\forall c . \forall x \in S . c \ast s x \in S) \hspace{1em} x \in S \hspace{1em}

\textbf{lemma} \texttt{subspaceI}:

\[ 0 \in S \Rightarrow (\forall x \in S \Rightarrow y \in S \Rightarrow x + y \in S) \Rightarrow (\forall c . x \in S \Rightarrow c \ast s x \in S) \Rightarrow \text{subspace S} \]

by \textbf{(auto simp: subspace-def)}

\textbf{lemma} \texttt{subspace-UNIV[simp]:} \texttt{subspace UNIV}

by \textbf{(simp add: subspace-def)}

\textbf{lemma} \texttt{subspace-single-0[simp]:} \texttt{subspace \{0\}}

by \textbf{(simp add: subspace-def)}

\textbf{lemma} \texttt{subspace-0:} \texttt{subspace S ⇒ 0 ∈ S}

by \textbf{(metis subspace-def)}

\textbf{lemma} \texttt{subspace-add:} \texttt{subspace S ⇒ x ∈ S ⇒ y ∈ S ⇒ x + y ∈ S}

by \textbf{(metis subspace-def)}

\textbf{lemma} \texttt{subspace-scale:} \texttt{subspace S ⇒ x ∈ S ⇒ c \ast s x ∈ S}

by \textbf{(metis subspace-def)}

\textbf{lemma} \texttt{subspace-neg:} \texttt{subspace S ⇒ x ∈ S ⇒ - x ∈ S}

by \textbf{(metis scale-minus-left scale-one subspace-scale)}

\textbf{lemma} \texttt{subspace-diff:} \texttt{subspace S ⇒ x ∈ S ⇒ y ∈ S ⇒ x - y ∈ S}

by \textbf{(metis diff-conv-add-uminus subspace-add subspace-neg)}
THEORY "Modules"

lemma subspace-sum: subspace A \implies (\forall x. x \in B \implies f x \in A) \implies \text{sum} f B \in A
by (induct B rule: infinite-finite-induct) (auto simp: subspace-add subspace-0)

lemma subspace-Int: (\forall i. i \in I \implies \text{subspace} (s i)) \implies \text{subspace} (\bigcap i \in I. s i)
by (auto simp: subspace-def)

lemma subspace-Inter: \forall s \in f. \text{subspace} s \implies \text{subspace} (\bigcap f)
unfolding subspace-def by auto

lemma subspace-inter: subspace A \implies subspace B \implies \text{subspace} (A \cap B)
by (simp add: subspace-def)

101  Span: subspace generated by a set

definition span :: 'b set \Rightarrow 'b set
where span-explicit: span b = \{(\sum a \in t. r a \ast s a) \mid t r. \text{finite} t \land t \subseteq b\}

lemma span-explicit':
span b = \{(\sum v \mid f v \neq 0. f v \ast s v) \mid f. \text{finite} \{v. f v \neq 0\} \land (\forall v. f v \neq 0 \implies v \in b)\}
unfolding span-explicit
proof safe
fix t r assume finite t t \subseteq b
then show \exists f. (\sum a \in t. r a \ast s a) = (\sum v \mid f v \neq 0. f v \ast s v) \land \text{finite} \{v. f v \neq 0\} \land (\forall v. f v \neq 0 \implies v \in b)
  by (intro exI[of - \lambda v. if v \in t then r v else 0]) (auto intro!: sum_mono_neutral_cong_right)

next
fix f :: 'b \Rightarrow 'a assume finite \{v. f v \neq 0\} (\forall v. f v \neq 0 \implies v \in b)
then show \exists t r. (\sum v \mid f v \neq 0. f v \ast s v) = (\sum a \in t. r a \ast s a) \land \text{finite} t \land t \subseteq b
  by (intro exI[of - \{v. f v \neq 0\}] exI[of - f]) auto
qed

lemma span-alt:
span B = \{(\sum x \mid f x \neq 0. f x \ast s x) \mid f. \{x. f x \neq 0\} \subseteq B \land \text{finite} \{x. f x \neq 0\}\}
unfolding span-explicit' by auto

lemma span-finite:
assumes fS: \text{finite} S
shows span S = \text{range} (\lambda u. \sum v \in S. u v \ast s v)
unfolding span-explicit
proof safe
fix t r assume t \subseteq S then show (\sum a \in t. r a \ast s a) \in \text{range} (\lambda u. \sum v \in S. u v \ast s v)
  by (intro image_eqI[of - - \lambda a. if a \in t then r a else 0])
(auto simp: if_distrib[of \lambda r. r a \ast s a for a] sum.If_cases fS Int-absorb1)

next
show \exists t r. (\sum v \in S. u v \ast s v) = (\sum a \in t. r a \ast s a) \land \text{finite} t \land t \subseteq S for u
  by (intro exI[of - u] exI[of - S]) (auto intro: fS)
qed

lemma span-induct-alt [consumes 1, case-names base step, induct set: span]:
assumes x: x ∈ span S
assumes h0: h 0 and hS: \( \text{\( \forall \)} x y. x \in S \implies h y \implies h (c \cdot s x + y) \)
shows h x
using x unfolding span-explicit
proof safe
fix t r assume finite t t ⊆ S then show h \( (\sum a \in t. r a \cdot s a) \)
  by (induction t) (auto intro!: hS h0)
qed

lemma span-mono: \( A \subseteq B \implies \text{span } A \subseteq \text{span } B \)
by (auto simp: span-explicit)

lemma span-base: a ∈ S ⇒ a ∈ span S
by (auto simp: span-explicit intro: exI [of - {a}])

lemma span-superset: S ⊆ span S
by (auto simp: span-base)

lemma span-zero: 0 ∈ span S
by (auto simp: span-explicit)

lemma span-UNIV [simp]: span UNIV = UNIV
by (auto intro: span-base)

lemma span-add: x ∈ span S ⇒ y ∈ span S ⇒ x + y ∈ span S
unfolding span-explicit
proof safe
fix tx ty rz ry assume *: finite tx finite ty tx ⊆ S ty ⊆ S
have [simp]: \( tx \cup ty \) ⊆ \( tx \cup ty \) \( tx \cup ty \) ⊆ ty
  by auto
show \( \exists t r. (\sum a \in tx. r x a \cdot s a) + (\sum a \in ty. r y a \cdot s a) = (\sum a \in t. r a \cdot s a) \land \text{finite } t \land t \subseteq S \)
  apply (intro exI [of - tx ∪ ty])
  apply (intro exI [of - λa. (if a ∈ tx then rz a else 0) + (if a ∈ ty then ry a else 0)] [of - λa. (if a ∈ tx then rz a else 0) + (if a ∈ ty then ry a else 0)])
  apply (auto simp: * scale-left-distrib sum.distrib if-distrib [of λr. r * s a for a] sum.If-cases)
done

qed

lemma span-scale: x ∈ span S ⇒ c * s x ∈ span S
unfolding span-explicit
proof safe
fix t r assume *: finite t t ⊆ S
show \( \exists t' r'. c \cdot s (\sum a \in t. r a \cdot s a) = (\sum a \in t'. r' a \cdot s a) \land \text{finite } t' \land t' \subseteq S \)
  by (intro exI [of - t] exI [of - λa. c * r a]) (auto simp: * scale-sum-right)
qed

**lemma** subspace-span [iff]: subspace (span S)
  **by** (auto simp: subspace-def span-zero span-add span-scale)

**lemma** span-neg: \( x \in \text{span} S \implies -x \in \text{span} S \)
  **by** (metis subspace-neg subspace-span)

**lemma** span-diff: \( x \in \text{span} S \implies y \in \text{span} S \implies x - y \in \text{span} S \)
  **by** (metis subspace-span subspace-diff)

**lemma** span-sum: (\( \forall x. x \in A \implies f x \in \text{span} S \)) \( \implies \text{sum} f A \in \text{span} S \)
  **by** (rule subspace-sum, rule subspace-span)

**lemma** span-minimal: \( S \subseteq T \implies \text{subspace} T \implies \text{span} S \subseteq T \)
  **by** (auto simp: span-explicit intro \( \vdash \): subspace-sum subspace-scale)

**lemma** span-def: \( \text{span} S = \text{subspace \ hull} S \)
  **by** (intro hull-unique [symmetric] span-superset subspace-span span-minimal)

**lemma** span-unique:
  \( S \subseteq T \implies \text{subspace} T \implies (\forall T'. S \subseteq T' \implies \text{subspace} T' \implies T \subseteq T') \implies \text{span} S = T \)
  **unfolding** span-def **by** (rule hull-unique)

**lemma** span-subspace-induct[consumes 2]:
  **assumes** \( x \in \text{span} S \)
  and \( P : \text{subspace} (\text{Collect} P) \)
  and \( SP : \forall x. x \in S \implies x \in P \)
  **shows** \( x \in P \)
  **proof** –
  from \( SP \) have \( SP' : S \subseteq P \)
    **by** (simp add: subset-eq)
  from \( x \text{ hull-minimal}[\text{where} S=\text{subspace}, \text{OF} SP' P, \text{unfolded} \text{span-def}[\text{symmetric}]] \)
  show \( x \in P \)
    **by** (metis subset-eq)
  qed

**lemma** (in module) span-induct[consumes 1, case-names base step, induct set: \( \text{span} \)]:
  **assumes** \( x \in \text{span} S \)
  and \( P : \text{subspace} (\text{Collect} P) \)
  and \( SP : \forall x. x \in S \implies P x \)
  **shows** \( P x \)
  **using** \( P \text{ SP} \text{ span-subspace-induct} x \) **by** fastforce

**lemma** span-empty[simp]: \( \text{span} \{\} = \{0\} \)
  **by** (rule span-unique) (auto simp add: subspace-def)
lemma span-subspace: \( A \subseteq B \implies B \subseteq \text{span} A \implies \text{subspace} B \implies \text{span} A = B \)
by (metis order-antisym span-def hull-minimal)

lemma span-span: \( \text{span} (\text{span} A) = \text{span} A \)
unfolding span-def hull-hull ..

lemma span-add-eq: assumes \( x \in \text{span} S \) shows \( x + y \in \text{span} S \iff y \in \text{span} S \)
proof assume \( *: x + y \in \text{span} S \)
have \( (x + y) - x \in \text{span} S \) using \(* x \) by (rule span-diff)
then show \( y \in \text{span} S \) by simp
qed (intro span-add x)

lemma span-add-eq2: assumes \( y \in \text{span} S \) shows \( x + y \in \text{span} S \iff x \in \text{span} S \)
using span-add-eq[of y S x]
by (auto simp: ac-simps)

lemma span-singleton: \( \text{span} \{x\} = \text{range} (\lambda k. k * s x) \)
by (auto simp: span-finite)

lemma span-Un: \( \text{span} (S \cup T) = \{x + y \mid x, y \in \text{span} S \land y \in \text{span} T\} \)
proof safe
fix \( x \) assume \( x \in \text{span} (S \cup T) \)
then obtain \( t r \) where \( t: \text{finite} t \subseteq S \cup T \) and \( x = (\sum a \in t. r a * s a) \)
by (auto simp: span-explicit)
moreover have \( t \cap S \cup (t - S) = t \) by auto
ultimately show \( \exists xa y. x = xa + y \land xa \in \text{span} S \land y \in \text{span} T \)
unfolding \( x \)
apply (rule-tac exI[of - \sum a \in t \cap S. r a * s a])
apply (rule-tac exI[of - \sum a \in t - S. r a * s a])
apply (subst sum.union-inter-neutral[symmetric])
apply (auto intro!: span-sum span-scale intro: span-base)
done

next
fix \( x y \) assume \( x \in \text{span} S \land y \in \text{span} T \) then show \( x + y \in \text{span} (S \cup T) \)
using span-mono[of S S \cup T] span-mono[of T S \cup T]
by (auto intro!: span-add)
qed

lemma span-insert: \( \text{span} (\text{insert} a S) = \{x. \exists k. (x - k * s a) \in \text{span} S\} \)
proof
have \( \text{span} \{(a) \cup S\} = \{x. \exists k. (x - k * s a) \in \text{span} S\} \)
unfolding span-Un span-singleton
apply (auto simp add: set-eq_iff)
subgoal for \( y k \) by (auto intro!: exI[of - k])
subgoal for \( y k \) by (rule exI[of - k * s a], rule exI[of - y - k * s a]) auto
done
then show ?thesis by simp
qed

**lemma** span-breakdown:
assumes bS: \( b \in S \)
and aS: \( a \in \text{span} \ S \)
shows \( \exists k. a - k \cdot s \ b \in \text{span} \ (S - \{ b \}) \)
using assms span-insert \([of \ b \ S - \{ b \}]\)
by (simp add: insert-absorb)

**lemma** span-breakdown-eq: \( x \in \text{span} \ (\text{insert} \ a \ S) \iff (\exists k. x - k \cdot s \ a \in \text{span} \ S) \)
by (simp add: span-insert)

**lemmas** span-clauses = span-base span-zero span-add span-scale

**lemma** span-eq-iff [simp]: \( \text{span} \ s = s \iff \text{subspace} \ s \)
unfolding span-def by (rule hull-eq) (rule subspace-Inter)

**lemma** span-eq-insert-eq:
assumes \((x - y) \in \text{span} \ S\)
shows \(\text{span} \ (\text{insert} \ x \ S) = \text{span} \ (\text{insert} \ y \ S)\)
proof

have *: \(\text{span} \ (\text{insert} \ x \ S) \subseteq \text{span} \ (\text{insert} \ y \ S)\) if \((x - y) \in \text{span} \ S\) for \(x\ y\)
proof

have 1: \((r \cdot s \ x - r \cdot s \ y) \in \text{span} \ S\) for \(r\)
by (metis scale-right-diff-distrib span-scale that)

have 2: \((z - k \cdot s \ y) - k \cdot s \ (x - y) = z - k \cdot s \ x\) for \(z \ k\)
by (simp add: scale-right-diff-distrib)
show ?thesis

apply (clarsimp simp add: span-breakdown-eq)
by (metis 1 2 diff-add-cancel scale-right-diff-distrib span-add-eq)
qed

show ?thesis

apply (intro subset-antisym * assms)
using assms subspace-neg subspace-span minus-diff-eq by force
qed

102 Dependent and independent sets

definition dependent :: 'b set \Rightarrow \text{bool}
where dependent-explicit: dependent s \iff (\exists t \ u. \text{finite} \ t \wedge t \subseteq s \wedge (\sum v \in t. \ u v \cdot s v) = 0 \wedge (\exists v \in t. \ u v \neq 0))

abbreviation independent s \equiv \neg \text{dependent} s

**lemma** dependent-mono: dependent B \Rightarrow B \subseteq A \Rightarrow dependent A
THEORY "Modules"

theory

lemma independent-monofinite: independent A ⇒ B ⊆ A ⇒ independent B
by (auto intro: dependent-mono)

lemma dependent-zero: 0 ∈ A ⇒ dependent A
by (auto simp: dependent-explicit intro: dependent-mono)

lemma independent-empty[intro]: independent {}
by (simp add: dependent-explicit)

lemma independent-explicit-module:
independent s ←→ (∀ t u v. finite t → t ⊆ s → (∑ v∈t. u v * s v) = 0 → v ∈ t → u v = 0)

lemma independentD: independent s ⇒ finite t ⇒ t ⊆ s ⇒ (∑ v∈t. u v * s v) = 0 =⇒ v ∈ t =⇒ u v = 0
by (simp add: independent-explicit-module)

lemma independent-Union-directed:
assumes directed: ∀ c d. c ∈ C =⇒ d ∈ C =⇒ c ⊆ d ∨ d ⊆ c
assumes indep: ∀ c. c ∈ C =⇒ independent c
shows independent (∪ C)
proof
  assume dependent (∪ C)
  then obtain u v S where S: finite S S ⊆ ∪ C v ∈ S u v ≠ 0 (∑ v∈S. u v * s v) = 0
  by (auto simp: dependent-explicit)

  have S ≠ {}
  using (v ∈ S) by auto

  have ∃ c∈C. S ⊆ c
  using (finite S) (S ≠ {}) (S ⊆ C)
  proof (induction rule: finite-ne-induct)
    case (insert i I)
    then obtain c d where cd: c ∈ C d ∈ C and iI: I ⊆ c i ∈ d
    by blast
    from directed[OF cd] cd have c ∪ d ∈ C
    by (auto simp: sup.absorb1 sup.absorb2)
    with iI show ?case
    by (intro bexI[of - c ∪ d]) auto
  qed auto

  then obtain c where c ∈ C S ⊆ c
  by auto

  have dependent c
  unfolding dependent-explicit
  by (intro exI[of - S] exI[of - u] bexI[of - v] conjI) fact+
  with indep[OF (c ∈ C)] show False
lemma dependent-finite:
assumes finite S
shows dependent S ←→ (∃ u. (∃ v ∈ S. u v ≠ 0) ∧ (∑ v∈S. u v * s v) = 0)
(is ?lhs = ?rhs)
proof
assume ?lhs then obtain T u v
where finite T T ⊆ S u v ≠ 0 (∑ v∈T. u v * s v) = 0
by (force simp: dependent-explicit)
with assms show ?rhs
apply (rule-tac x = λ v. if v ∈ T then u v else 0 in exI)
apply (auto simp: sum mono-neutral-right)
done
next
assume ?rhs with assms show ?lhs
by (fastforce simp add: dependent-explicit)
qed

lemma dependent-alt:
dependent B ←→ (∃ X. finite {x. X x ≠ 0} ∧ {x. X x ≠ 0} ⊆ B ∧ (∑ x∈B. X x * s x) = 0 ∧ (∃ x. X x ≠ 0))
unfolding dependent-explicit
apply safe
subgoal for S u v
apply (intro exI[of _ : λx. if x ∈ S then u x else 0])
apply (subst sum mono-neutral-cong-left[where T=S])
apply (auto intro!: sum mono-neutral-cong-right cong: rev-conj-cong)
done
apply auto
done

lemma independent-alt:
independent B ←→ (∀ X. finite {x. X x ≠ 0} → {x. X x ≠ 0} ⊆ B → (∑ x∈B. X x * s x) = 0 → (∀ x. X x = 0))
unfolding dependent-alt by auto

lemma independentD-alt:
independent B → finite {x. X x ≠ 0} → {x. X x ≠ 0} ⊆ B → (∑ x∈B. X x * s x) = 0 → X x = 0
unfolding independent-alt by blast

lemma independentD-unique:
assumes B: independent B
and X: finite {x. X x ≠ 0} {x. X x ≠ 0} ⊆ B
and \( Y : \text{finite} \{ x. \ Y \ x \neq 0 \} \ \{ x. \ Y \ x \neq 0 \} \subseteq B \)

and \((\sum x \mid X x \neq 0. \ X x \ * s \ x) = (\sum x \mid Y x \neq 0. \ Y x \ * s \ x)\)

shows \( X = Y \)

**proof**

have \( X x - Y x = 0 \) for \( x \)

using \( B \)

**proof** (rule independentD-alt)

have \( \{ x. \ X x - Y x \neq 0 \} \subseteq \{ x. \ X x \neq 0 \} \cup \{ x. \ Y x \neq 0 \} \)

by `auto`

then show finite \( \{ x. \ X x - Y x \neq 0 \} \ \{ x. \ X x - Y x \neq 0 \} \subseteq B \)

using \( X Y \) by (auto dest: finite-subset)

then have \((\sum x \mid X x - Y x \neq 0. \ (X x - Y x) \ * s \ x) = (\sum v \in \{ S. \ X S \neq 0 \} \cup \{ S. \ Y S \neq 0 \}. \ (X v - Y v) \ * s \ v)\)

using \( X Y \) by (intro sum_mono-neut-cong-left) auto

also have \( \ldots = (\sum v \in \{ S. \ X S \neq 0 \} \cup \{ S. \ Y S \neq 0 \}. \ X v \ * s \ v) - (\sum v \in \{ S. \ X S \neq 0 \} \cup \{ S. \ Y S \neq 0 \}. \ Y v \ * s \ v)\)

by (`simp add: scale-left-diff-distrib sum-subtractf assms`)

also have \((\sum v \in \{ S. \ X S \neq 0 \} \cup \{ S. \ Y S \neq 0 \}. \ X v \ * s \ v) = (\sum v \in \{ S. \ X S \neq 0 \}. \ X v \ * s \ v)\)

using \( X Y \) by (intro sum_mono-neut-cong-right) auto

also have \((\sum v \in \{ S. \ X S \neq 0 \} \cup \{ S. \ Y S \neq 0 \}. \ Y v \ * s \ v) = (\sum v \in \{ S. \ Y S \neq 0 \}. \ Y v \ * s \ v)\)

using \( X Y \) by (intro sum_mono-neut-cong-right) auto

finally show \((\sum x \mid X x - Y x \neq 0. \ (X x - Y x) \ * s \ x) = 0\)

using `assms` by `simp`

qed

then show `?thesis`

by `auto`

qed

### 103 Representation of a vector on a specific basis

**definition** `representation` :: `'b set ⇒ 'b ⇒ 'b ⇒ 'a`

**where** `representation` basis \( v = \)

(if independent basis ∧ \( v \) ∈ span basis then

SOME \( f. (\forall v. \ f v \neq 0 \ → v \in \text{basis}) ∧ \text{finite} \ \{ v. \ f v \neq 0 \} \ ∧ (\sum v \in \{ v. \ f v \neq 0 \}. \ f v \ * s \ v) = v\)

else (λb. 0))

**lemma** `unique-representation`:

**assumes** `basis`: independent basis

and `in-basis`: \(\forall v. \ f v \neq 0 \ → v \in \text{basis}\) \∧ \(\forall v. \ g v \neq 0 \ → v \in \text{basis}\)

and `[simp]`: finite \(\{ v. \ f v \neq 0 \}\) finite \(\{ v. \ g v \neq 0 \}\)

and `eq`: \((\sum v \in \{ v. \ f v \neq 0 \}. \ f v \ * s \ v) = (\sum v \in \{ v. \ g v \neq 0 \}. \ g v \ * s \ v)\)

shows \( f = g \)

**proof** (rule `ext`, rule `cccontr`)

fix \( v \) assume `ne`: \( f v \neq g v \)

have dependent basis

unfolding dependent-explicit
proof (intro exI congI)
  have *: \{v. f v - g v \neq 0\} \subseteq \{v. f v \neq 0\} \cup \{v. g v \neq 0\}
  by auto
  show finite \{v. f v - g v \neq 0\}
  by (rule finite-subset[OF *]) simp
  show \exists v\in\{v. f v - g v \neq 0\}. f v - g v \neq 0
  by (rule bexI[of - v]) (auto simp: ne)
  have (\sum v | f v - g v \neq 0). (f v - g v) *s v =
    (\sum v\in\{v. f v \neq 0\} \cup \{v. g v \neq 0\}. (f v - g v) *s v)
  by (intro sum.mono-neutral-cong-left) auto
  also have \ldots =
    (\sum v\in\{v. f v \neq 0\} \cup \{v. g v \neq 0\}. f v *s v) - (\sum v\in\{v. f v \neq 0\} \cup \{v. g v \neq 0\}. g v *s v)
  by (simp add: algebra-simps sum-subtractf)
  also have \ldots = (\sum v | f v \neq 0, f v *s v) - (\sum v | g v \neq 0, g v *s v)
  by (intro arg-cong2[where f= (-)]) simp add: span-explicit sum-cong
  finally show (\sum v | f v - g v \neq 0. (f v - g v) *s v) = 0
  by (simp add: eq)
  show \{v. f v - g v \neq 0\} \subseteq basis
  using \{v. f v \neq 0\}. f v *s v = v
  proof
  { assume basis: independent basis and v: v \in span basis
    define p where p f \longleftrightarrow
      (\forall v. f v \neq 0 \longrightarrow v \in basis) \land finite \{v. f v \neq 0\} \land (\sum v\in\{v. f v \neq 0\}. f v *s v) = v
    for f
    obtain t r where *: finite t t \subseteq basis (\sum b\in t. r b *s b) = v
      using (v \in span basis) by (auto simp: span-explicit)
    define f where f b = (if b \in t then r b else 0) for b
    have p f
      using * by (auto simp: p-def f-def intro!: sum.mono-neutral-cong-left)
    have *: representation basis v = Eps p by (simp add: p-def[abs-def] representation-def basis v)
      from someI[of p f, OF \{p f\}] have p (representation basis v)
      unfolding * . }
  note * = this

  show representation basis v b \neq 0 \Longrightarrow b \in basis for b
  using * by (cases independent basis \land v \in span basis) (auto simp: representation-def)
show finite \{ b. \text{representation basis} v b \neq 0 \}
using * by (cases independent basis \land v \in \text{span basis}) (auto simp: representation-def)

show independent basis \implies v \in \text{span basis} \implies (\sum b \mid \text{representation basis} v b \neq 0. \text{representation basis} v b \ast s b) = v
using * by auto
qed

lemma sum-representation-eq:
(\sum b \in B. \text{representation basis} v b \ast s b) = v
if independent basis \land v \in \text{span basis} \land \text{finite} B \subseteq B

proof
  have (\sum b \in B. \text{representation basis} v b \ast s b) =
  (\sum b \mid \text{representation basis} v b \neq 0. \text{representation basis} v b \ast s b)
  apply (rule sum.mono-neutral-cong)
  apply (rule finite-representation)
  apply fact

subgoal for b
  using (that representation-ne-zero[of basis v b])
  by auto
subgoal by auto
subgoal by simp
done
also have \ldots = v
  by (rule sum-nonzero-representation-eq; fact)
finally show \?thesis .
qed

lemma representation-eqI:
assumes basis: independent basis and b: v \in \text{span basis}
and ne-zero: \land b. f b \neq 0 \implies b \in \text{basis}
and finite: finite \{ b. f b \neq 0 \}
and eq: (\sum b \mid f b \neq 0. f b \ast s b) = v
shows representation basis v = f
by (rule unique-representation[of basis])
  (auto simp: representation-ne-zero finite-representation
  sum-nonzero-representation-eq[of basis b] ne-zero finite eq)

lemma representation-basis:
assumes basis: independent basis and b: b \in \text{basis}
sends representation basis b = (\lambda v. if v = b then 1 else 0)
proof (rule unique-representation[of basis])
show representation basis b v \neq 0 \implies v \in \text{basis} for v
  using representation-ne-zero .
send finite \{ v. \text{representation basis} b v \neq 0 \}
  using finite-representation .
send (if v = b then 1 else 0) \neq 0 \implies v \in \text{basis} for v
by (cases v = b) (auto simp: b)
have*: \{ v. (if v = b then 1 else 0 :: 'a) \neq 0 \} = \{ b \}
by auto

show finite \{ v. (if v = b then 1 else 0) ≠ 0 \} unfolding * by auto

show (\sum v \mid \text{representation basis} \ b \ v \neq 0, \text{representation basis} \ b \ v \ast s \ v) =
(\sum v \mid (if v = b then 1 else 0):\ a) \neq 0, (if v = b then 1 else 0) \ast s \ v)

unfolding * sum-nonzero-representation-eq[OF basis span-base[OF b]] by auto

qed

lemma representation-zero: representation basis 0 = (λb. 0)

proof cases

assume basis: independent basis show ?thesis

by (rule representation-eqI[OF basis span-zero]) auto

qed (simp add: representation-def)

lemma representation-diff:

assumes basis: independent basis and v: v ∈ span basis and u: u ∈ span basis

shows representation basis (u - v) = (λb. representation basis u b - representation basis v b)

proof (rule representation-eqI[OF basis span-diff[OF u v]])

let ?R = representation basis

note finite-representation[simp] u[simp] v[simp]

have *: \{ b. ?R u b - ?R v b \neq 0 \} ⊆ \{ b. ?R u b \neq 0 \} ∪ \{ b. ?R v b \neq 0 \}

by auto

then show ?R u b - ?R v b \neq 0 ⇒ b ∈ basis for b

by (auto dest: representation-ne-zero)

show finite \{ b. ?R u b - ?R v b \neq 0 \}

by (intro finite-subset[OF *]) simp-all

have (\sum b \mid ?R u b - ?R v b \neq 0. (?R u b - ?R v b) \ast s \ b) =
(\sum b \in \{ b. ?R u b \neq 0 \} ∪ \{ b. ?R v b \neq 0 \}. (?R u b - ?R v b) \ast s \ b)

by (intro sum mono-neutral-cong-left *) auto

also have * =
(\sum b \in \{ b. ?R u b \neq 0 \} ∪ \{ b. ?R v b \neq 0 \}. ?R u b \ast s \ b) - (\sum b \in \{ b. ?R u b \neq 0 \} ∪ \{ b. ?R v b \neq 0 \}. ?R v b \ast s \ b)

by (simp add: algebra-simps sum-subtractf)

also have * = (\sum b \mid ?R u b \neq 0. ?R u b \ast s \ b) - (\sum b \mid ?R v b \neq 0. ?R v b \ast s \ b)

by (intro arg cong2[where f = (−)] sum mono-neutral-cong-right) auto

finally show (\sum b \mid ?R u b - ?R v b \neq 0. (?R u b - ?R v b) \ast s \ b) = u - v

by (simp add: sum nonzero-representation-eq[OF basis])

qed

lemma representation-neq:

independent basis ⇒ v ∈ span basis ⇒ representation basis (− v) = (λb. − representation basis v b)

using representation-diff[of basis v 0] by (simp add: representation-zero span-zero)

lemma representation-add:

independent basis ⇒ v ∈ span basis ⇒ u ∈ span basis ⇒
representation basis (u + v) = (λb. representation basis u b + representation basis v b)
using representation-diff[of basis − v u] by (simp add: representation-neg representation-diff span-neg)

lemma representation-sum:
  assumes basis: independent basis and v: v ∈ span basis
  shows representation basis (r *s v) = (λb. r * representation basis v b)
proof (rule representation-eq[OF basis span-scale[OF v]])
  let ?R = representation basis
  have *: {b. r * ?R v b ≠ 0} ⊆ {b. ?R v b ≠ 0} by auto
  then show r * representation basis v b ≠ 0 ⟹ b ∈ basis for b
    using representation-ne-zero by auto
  show finite {b. r * ?R v b ≠ 0}
    by (intro finite-subset[OF *]) simp-all
  have (⨁ b | r * ?R v b ≠ 0. (r * ?R v b) *s b) = (∑ b∈{b. ?R v b ≠ 0}. (r * ?R v b) *s b)
    by (intro summono-neutral-cong-left *) auto
  also have ... = r *s (∑ b | ?R v b ≠ 0. ?R v b *s b)
    by (simp add: scale-scale[symmetric] scale-sum-right del: scale-scale)
  finally show (∑ b | r * ?R v b ≠ 0. (r * ?R v b) *s b) = r *s v
    by (simp add: sum-nonzero-representation-eq[OF basis])

qed

lemma representation-extend:
  assumes basis: independent basis and v: v ∈ span basis' and basis': basis' ⊆ basis
  shows representation basis v = representation basis' v
proof (rule representation-eq[OF basis'])
  show v': v ∈ span basis using span-mono[OF basis'] v by auto
  have *: independent basis' using basis' basis by (auto intro: dependent-mono)
  show representation basis' v b ≠ 0 ⟹ b ∈ basis for b
    using representation-ne-zero basis' by auto
  show finite {b. representation basis' v b ≠ 0}
    using finite-representation .
  show (∑ b | representation basis' v b ≠ 0. representation basis' v b *s b) = v
    using sum-nonzero-representation-eq[OF * v] .

qed

The set B is the maximal independent set for span B, or A is the minimal spanning set

lemma spanning-subset-independent:
  assumes BA: B ⊆ A
and $iA$: independent $A$
and $AsB$: $A \subseteq \text{span } B$
shows $A = B$

**proof** (intro antisym[OF - BA] subsetI)

have $iB$: independent $B$ using independent-mono [OF $iA$ $BA$].
fix $v$ assume $v \in A$
with $AsB$ have $v \in \text{span } B$ and $?RA = \text{representation } A v$

have $?RB v = 1$
  unfolding representation-extend[OF $iA$ $\langle v \in \text{span } B \rangle$] $BA$,
symmetric [OF $iA$ $\langle v \in A \rangle$] by simp
then show $v \in B$
  using representation-ne-zero[of $B$ $v$ $v$] by auto

qed

end

A linear function is a mapping between two modules over the same ring.

locale module-hom = $m1$: module $s1$ + $m2$: module $s2$
  for $s1 :: 'a::comm-ring-1 \Rightarrow 'b::ab-group-add \Rightarrow 'b$ (infixr $* a 75$)
  and $s2 :: 'a::comm-ring-1 \Rightarrow 'c::ab-group-add \Rightarrow 'c$ (infixr $* b 75$) +
fixes $f :: 'b \Rightarrow 'c$
assumes add: $f (b1 + b2) = f b1 + f b2$
and scale: $f (r * a b) = r * b f b$

begin

lemma zero[simp]: $f 0 = 0$
using scale[of $0 0$] by simp

lemma neg: $f (\neg x) = \neg f x$
using scale [where $r=\neg 1$] by (metis add add-eq-0-iff zero)

lemma diff: $f (x - y) = f x - f y$
by (metis diff-conv-add-uminus add neg)

lemma sum: $f (\sum g S) = (\sum a \in S. f (g a))$
proof (induct $S$ rule: infinite-finite-induct)
case (insert $x F$)
have $f (\sum g (\text{insert } x F)) = f (g x + \sum g F)$
  using insert.hyps by simp
also have \ldots $= f (g x) + f (\sum g F)$
  using add by simp
also have \ldots $= (\sum a \in \text{insert } x F. f (g a))$
  using insert.hyps by simp
finally show $?case$.
qed simp-all

lemma inj-on-iff-eq-0:
assumes $s$: $m1$.subspace $s$
shows inj-on $f$ $s$ $\iff (\forall x \in s. f x = 0 \imp x = 0)$
proof
  
  have inj-on $f$ $s$ $\iff (\forall x \in s. \forall y \in s. f x \neq f y \imp x \neq y)$
    by (simp add: inj-on-def)
  also have $\ldots$ $\iff (\forall x \in s. \forall y \in s. f (x - y) = 0 \imp x \neq y)$
    by (simp add: diff)
  also have $\ldots$ $\iff (\forall x \in s. f x = 0 \imp x = 0)$ (is $?l = ?r$)
  proof safe
    fix $x$ assume $?l$ assume $x \in s \ f x = 0$
    with $\langle ?l \rangle$ [rule-format, of $x 0$]
    show $x = 0$
      by (auto simp: m1.subspace-0)
  next
    fix $x \ y$ assume $?r$ assume $x \in s \ y \in s \ f (x - y) = 0$
    with $\langle ?r \rangle$[rule-format, of $x - y$] $s$
    show $x - y = 0$
      by (auto simp: m1.subspace-diff)
  qed
  finally show $?\thesis$
    by auto
qed

lemma inj-iff-eq-0: $\text{inj } f = (\forall x. f x = 0 \imp x = 0)$
  by (rule inj-on-iff-eq-0[OF m1.subspace-UNIV, unfolded ball-UNIV])

lemma subspace-image: assumes $S$: m1.subspace $S$ shows m2.subspace $(f ^{'} S)$
  unfolding m2.subspace-def
proof safe
  
  show $0 \in f ^{'} S$
    by (rule image-eqI[of _ _ 0]) (auto simp: m1.subspace-0)
  show $x \in S \imp y \in S \imp f x + f y \in f ^{'} S$ for $x \ y$
    by (rule image-eqI[of _ _ x + y]) (auto simp: m1.subspace-add add)
  show $x \in S \imp r * b f x \in f ^{'} S$ for $r x$
    by (rule image-eqI[of _ _ r * a x]) (auto simp: m1.subspace-scale scale)
  qed

lemma subspace-vimage: m2.subspace $S$ $\imp$ m1.subspace $(f ^{'} S)$
  by (simp add: vimage-def add scale m1.subspace-def m2.subspace-0 m2.subspace-add m2.subspace-scale)

lemma subspace-kernel: m1.subspace \{ $x. f x = 0$\}
  using subspace-vimage[OF m2.subspace-single-0] by (simp add: vimage-def)

lemma span-image: m2.span $(f ^{'} S) = f ^{'} (m1.span S)$
  (rule m2.span-unique)
  
  show $f ^{'} S \subseteq f ^{'} m1.span S$
    by (rule image-mono, rule m1.span-superset)
  show $m2.span (f ^{'} m1.span S)$
    using m1.subspace-span by (rule subspace-image)
next
fix $T$ assume $f \cdot s \subseteq T$ and $m_2$ subspace $T$ then show $f \cdot m_1$ span $S \subseteq T$
unfolding image-subset-iff-subset-vimage by (metis subspace-vimage m1 span-minimal)
qed

lemma dependent-inj-imageD:
assumes $d$: $m_2$ independent $(f \cdot s)$ and $i$: inj-on $f$ $(m_1$ span $s$)
shows $m_1$ dependent $s$
proof
have [intro]: inj-on $f$ $s$
  using (inj-on $f$ $(m_1$ span $s)$) $m_1$ span-superset by (rule inj-on-subset)
from $d$ obtain $s' r v$ where $*$: finite $s' s' \subseteq s$ $(\sum v \in f \cdot s' \cdot r v \ast b v) = 0 v \in s' r f (v v) \neq 0$
  by (auto simp: $m_2$ dependent-explicit subset-image-iff dest!: finite-imageD intro: inj-on-subset)
  have $f (\sum v \in s' \cdot r (f v) \ast a v) = (\sum v \in s' \cdot r (f v) \ast b f v)$
    by (simp add: sum scale)
  also have ... = $(\sum v \in f \cdot s' \cdot r v \ast b v)$
    using ($s' \subseteq s$) by (subst sum.reindex) (auto dest!: finite-imageD intro: inj-on-subset)
finally have $f (\sum v \in s' \cdot r (f v) \ast a v) = 0$
  by (simp add: *)
with ($s' \subseteq s$) have $(\sum v \in s' \cdot r (f v) \ast a v) = 0$
  by (intro inj-onD[of $i$] $m_1$ span-zero $m_1$ span-sum $m_1$ span-scale) (auto intro: $m_1$ span-base)
  then show $m_1$ dependent $s$
    using (finite $s'$) ($s' \subseteq s$) ($v \in s' \cdot r (f v) \neq 0$) by (force simp add: $m_1$ dependent-explicit)
qed

lemma eq-0-on-span:
assumes $f0$: $\forall x. x \in b \Longrightarrow f x = 0$ and $x$: $x \in m_1$ span $b$ shows $f x = 0$
using $m_1$ span-induct[of $x$ subspace-kernel] $f0$ by simp

lemma independent-injective-image: $m_1$ independent $s$ $\Longrightarrow$ inj-on $f$ $(m_1$ span $s)$
$\Longrightarrow$ $m_2$ independent $(f \cdot s)$
using dependent-inj-imageD[of $s$] by auto

lemma inj-on-span-independent-image:
assumes $fB$: $m_2$ independent $(f \cdot B)$ and $f$: inj-on $f B$ shows inj-on $f$ $(m_1$ span $B)$
unfolding inj-on-iff-eq-0[of $m_1$ subspace-span] unfolding $m_1$ span-explicit'
proof safe
fix $r$ assume $fr$: finite $\{ v, r v \neq 0 \}$ and $r$: $\forall v. r v \neq 0 \longrightarrow v \in B$
and $eq0$: $f (\sum v \mid r v \neq 0. r v \ast a v) = 0$
have $0 = (\sum v \mid r v \neq 0. r v \ast b f v)$
  using $eq0$ by (simp add: sum scale)
also have ... = $(\sum v \in f \cdot \{ v, r v \neq 0 \}. r (\text{the-inv-into} B f v) \ast b v)$
  using $r$ by (subst sum.reindex) (auto simp: the-inv-into-f[OF $f$] intro!: inj-on-subset[of $f$] sum.cong)
finally have $r v \neq 0 \Longrightarrow r (\text{the-inv-into} B f (f v)) = 0$ for $v$
  using $fr$ $fB$ unfolded $m_2$ independent-explicit-module, rule-format,
of \( f \cdot \{ v, r v \neq 0 \} \lambda v. r (\text{the-inv-into } B f v) \]

by auto

then have \( r v = 0 \) for \( v \)

using \( \text{the-inv-into-f} f \{ \text{OF f} \} \) \( r \) by auto

then show \( (\sum v | r v \neq 0. r v \cdot a v) = 0 \) by auto

qed

**lemma** \( \text{inj-on-span-iff-independent-image} \):

\( \text{m2.independent } (f \cdot B) \Longrightarrow \text{inj-on } f \\
(\text{m1.span } B) \Longleftrightarrow \text{inj-on } f B \)

using \( \text{inj-on-span-independent-image}[\text{of B}] \) \( \text{inj-on-subset} \) \( \text{OF - m1.span-superset} \), \( \text{OF f B} \) by auto

**lemma** \( \text{subspace-linear-preimage} \):

\( \text{m2.subspace } S \Longrightarrow \text{m1.subspace } \{ x. f x \in S \} \\
by (\text{simp add: add scale m1.subspace-def m2.subspace-def})

**lemma** \( \text{spans-image} \):

\( V \subseteq \text{m1.span } B \Longrightarrow f \cdot V \subseteq \text{m2.span } (f \cdot B) \)

by (metis image-mono span-image)

Relation between bases and injectivity/surjectivity of map.

**lemma** \( \text{spanning-surjective-image} \):

assumes us: \( \text{UNIV} \subseteq \text{m1.span } S \)
and sf: \( \text{surj } f \)
shows \( \text{UNIV} \subseteq \text{m2.span } (f \cdot S) \)

proof –

have \( \text{UNIV} \subseteq f \cdot \text{UNIV} \)
using sf by (auto simp add: surj-def)
also have \( \ldots \subseteq \text{m2.span } (f \cdot S) \)
using \( \text{spans-image}[\text{OF us}] \).
finally show \( \text{thesis} \).

qed

**lemmas** \( \text{independent-inj-on-image} = \text{independent-injective-image} \)

**lemma** \( \text{independent-inj-image} \):

\( \text{m1.independent } S \Longrightarrow \text{inj } f \Longrightarrow \text{m2.independent } (f \cdot S) \)

using \( \text{independent-inj-on-image}[\text{of S}] \) by (auto simp: subset-inj-on)

end

**lemma** \( \text{module-hom-iff} \):

\( \text{module-hom s1 s2 } f \Longleftrightarrow \\
\text{module s1 } \land \text{module s2 } \land \\
(\forall x y. f (x + y) = f x + f y) \land (\forall c x. f (s1 c x) = s2 c (f x)) \\
by (\text{simp add: module-hom-def module-hom-axioms-def})

locale \( \text{module-pair} = \text{m1: module s1 + m2: module s2} \)
for s1 :: 'a :: \text{comm-ring-1} \Rightarrow 'b :: \text{ring-1}
and s2 :: 'a :: \text{comm-ring-1} \Rightarrow 'c :: \text{ab-group-add}

begin
lemma module-hom-zero: module-hom s1 s2 (λx. 0)
  by (simp add: module-hom-iff m1.module-axioms m2.module-axioms)

lemma module-hom-add: module-hom s1 s2 f ⇒ module-hom s1 s2 g ⇒ module-hom s1 s2 (λx. f x + g x)
  by (simp add: module-hom-iff module-scale-right-distrib)

lemma module-hom-sub: module-hom s1 s2 f ⇒ module-hom s1 s2 g ⇒ module-hom s1 s2 (λx. f x - g x)
  by (simp add: module-hom-iff module-scale-right-diff-distrib)

lemma module-hom-neg: module-hom s1 s2 f ⇒ module-hom s1 s2 (λx. -f x)
  by (simp add: module-hom-iff module-scale-minus-right)

lemma module-hom-compose-scale:
  module-hom s1 s2 (λx. s2 (f x) (c))
  if module-hom s1 (λx. f x)
proof –
  interpret mh: module-hom s1 (λx. f x) by fact
  show ?thesis
    by unfold-locales (simp-all add: mh.add mh.scale m2.scale-left-distrib)
qed

lemma bij-module-hom-imp-inv-module-hom: module-hom s1 s2 scale1 scale2 f ⇒ bij f
  ⇒ module-hom s1 s2 (λx. scale2 f (scale1 x))
by (auto simp: module-hom-iff bij-is-surj bij-is-inj surj-f-inv-f
  intro!: Hilbert-Choice.inv-f-eq)

lemma module-hom-sum: (⋀i. i ∈ I ⇒ module-hom s1 s2 (f i)) ⇒ (I = {} ⇒ module s1 ∧ module s2) ⇒ module-hom s1 s2 (λx. ∑i∈I. f i x)
  apply (induction I rule: infinite-finite-induct)
  apply (auto intro!: module-hom-zero module-hom-add)
  using m1.module-axioms m2.module-axioms by blast

lemma module-hom-eq-on-span: f x = g x
  if module-hom s1 s2 f module-hom s1 s2 g
  and (λx. x ∈ B ⇒ f x = g x) x ∈ m1.span B
proof –
  interpret module-hom s1 s2 (λx. f x - g x)
    by (rule module-hom-sub that)+
  from eq-0-on-span[OF - that(4)] that(3) show ?thesis by auto
qed
end

context module begin

lemma module-hom-scale-self[simp]:
  module-hom scale scale (λx. scale c x)
  using module-axioms module-hom-iff scale-left-commute scale-right-distrib by blast

lemma module-hom-scale-left[simp]:
  module-hom (∗) scale (λr x)
  by unfold-locales (auto simp: algebra-simps)

lemma module-hom-id: module-hom scale scale id
  by (simp add: module-hom-iff module-axioms)

lemma module-hom-ident: module-hom scale scale (λx. x)
  by (simp add: module-hom-iff module-axioms)

lemma module-hom-uminus: module-hom scale scale uminus
  by (simp add: module-hom-iff module-axioms)

end

lemma module-hom-compose: module-hom s1 s2 f =⇒ module-hom s2 s3 g =⇒ module-hom s1 s3 (g o f)
  by (auto simp: module-hom-iff)

end

104 Vector Spaces

theory Vector-Spaces imports Modules begin

lemma isomorphism-expand:
  f o g = id ∧ g o f = id =⇒ (∀x. f (g x) = x) ∧ (∀x. g (f x) = x)
  by (simp add: fun-eq-iff o-def id-def)

lemma left-right-inverse-eq:
  assumes fg: f o g = id
  and gh: g o h = id
  shows f = h
  proof -
  have f = f o (g o h)
    unfolding gh by simp
  also have ... = (f o g) o h
    by (simp add: o-assoc)
  
end
finally show \( f = h \)
unfolding \( f \) by simp
qed

lemma ordLeq3-finite-infinite: assumes \( A: \) finite \( A \) and \( B: \) infinite \( B \) shows \( \text{ordLeq3} \ (\text{card-of} \ A) \ (\text{card-of} \ B) \)
proof --
have \( \text{ordLeq3} \ (\text{card-of} \ A) \ (\text{card-of} \ B) \lor \text{ordLeq3} \ (\text{card-of} \ B) \ (\text{card-of} \ A) \)
by (intro ordLeq-total card-of-Well-order)
moreover have \( \neg \text{ordLeq3} \ (\text{card-of} \ B) \ (\text{card-of} \ A) \)
using \( B \) \( A \) card-of-ordLeq-finite \[\text{of } B \ A\]
by auto
ultimately show \(?\text{thesis}\) by auto
qed

locale vector-space =
fixes \( \text{scale} :: \) \( 'a::\text{field} \Rightarrow 'b::\text{ab-group-add} \Rightarrow 'b\) \( (\text{infixr } \ast 75) \)
assumes \( \text{vector-space-assms}:\) re-stating the assumptions of module instead of extending module allows us to rewrite in the sublocale.
\[
\begin{align*}
  a \ast s (x + y) &= a \ast s x + a \ast s y \\
  (a + b) \ast s x &= a \ast s x + b \ast s x \\
  a \ast s (b \ast s x) &= (a \ast b) \ast s x \\
  1 \ast s x &= x
\end{align*}
\]
lemma module-iff-vector-space: \( \text{module } s \rightleftharpoons \text{vector-space } s \)
unfolding module-def vector-space-def ..

locale linear = \( \text{vs1: } \text{vector-space } s1 + \text{vs2: } \text{vector-space } s2 + \text{module-hom } s1 \ s2 \ f \)
for \( s1 :: 'a::\text{field} \Rightarrow 'b::\text{ab-group-add} \Rightarrow 'b\) \( (\text{infixr } \ast a 75) \)
and \( s2 :: 'a::\text{field} \Rightarrow 'c::\text{ab-group-add} \Rightarrow 'c\) \( (\text{infixr } \ast b 75) \)
and \( f :: 'b \Rightarrow 'c\)
lemma module-hom-iff-linear: \( \text{module-hom } s1 \ s2 \ f \rightleftharpoons \text{linear } s1 \ s2 \ f \)
unfolding module-hom-def linear-def module-iff-vector-space by auto
lemmas module-hom-eq-linear = module-hom-iff-linear[abs-def, THEN meta-eq-to-obj-eq]
lemmas linear-iff-module-hom = module-hom-iff-linear[symmetric]
lemmas linear-module-hom1 = module-hom-iff-linear[THEN iffD1]
and module-hom-linear1 = module-hom-iff-linear[THEN iffD2]

class vector-space

sublocale module scale rewrites module-hom = linear
by unfold-locales (fact vector-space-assms module-hom-eq-linear)+

lemmas -- from module
linear-id = module-hom-id
and linear-ident = module-hom-ident
and linear-scale-self = module-hom-scale-self
and linear-scale-left = module-hom-scale-left
and linear-uminus = module-hom-uminus
lemma linear-imp-scale:
  fixes D::'a ⇒ 'b
  assumes linear (∗) scale D
  obtains d where D = (λx. scale x d)
proof –
  interpret linear (∗) scale D by fact
  show ?thesis
    by (metis mult.commute mult.left-neutral scale that)
qed

lemma scale-eq-0-iff [simp]: scale a x = 0 ←→ a = 0 ∨ x = 0
  by (metis scale-left-commute right-inverse scale-one scale-scale scale-zero-left)

lemma scale-left-imp-eq:
  assumes nonzero: a ≠ 0
      and scale: scale a x = scale a y
  shows x = y
proof –
  from scale have scale a (x − y) = 0
    by (simp add: scale-right-diff-distrib)
  with nonzero have x − y = 0 by simp
  then show x = y by (simp only: right-minus-eq)
qed

lemma scale-right-imp-eq:
  assumes nonzero: x ≠ 0
      and scale: scale a x = scale b x
  shows a = b
proof –
  from scale have scale (a − b) x = 0
    by (simp add: scale-left-diff-distrib)
  with nonzero have a − b = 0 by simp
  then show a = b by (simp only: right-minus-eq)
qed

lemma scale-cancel-left [simp]: scale a x = scale a y ←→ x = y ∨ a = 0
  by (auto intro: scale-left-imp-eq)

lemma scale-cancel-right [simp]: scale a x = scale b x ←→ a = b ∨ x = 0
  by (auto intro: scale-right-imp-eq)

lemma injective-scale: c ≠ 0 ⇒ inj (λx. scale c x)
  by (simp add: inj-on-def)

lemma dependent-def: dependent P ←→ (∃ a ∈ P. a ∈ span (P − {a}))
  unfolding dependent-explicit
proof safe
  fix a assume aP: a ∈ P and a ∈ span (P − {a})
then obtain \(a S u\)

where \(aP\): \(a \in P\) and \(fS\): finite \(S\) and \(SP\): \(S \subseteq P\) \(a \notin S\) and \(ua\): \((\sum_{v \in S} u v) = a\)

unfolding span-explicit by blast

let \(?S = \text{insert } a S\)

let \(?u = \lambda y. \text{if } y = a \text{ then } -1 \text{ else } u y\)

from \(fS SP\) have \((\sum_{v \in ?S} ?u v * v) = 0\)

by (simp add: if-distrib[of \(\lambda r. r * s a\) for \(a\)] sum.If-cases field-simps Diff-eq[ symmetric] 

moreover have finite \(?S \subseteq P\) \(a \in ?S\) \(?u a \neq 0\)

using \(fS SP aP\) by auto

ultimately show \(\exists t u. \text{finite } t \land t \subseteq P \land (\sum_{v \in t} u v * v) = 0 \land (\exists v \in t. u v \neq 0)\) by fast

next

fix \(S u v\)

assume \(fS\): finite \(S\) and \(SP\): \(S \subseteq P\) and \(vS\): \(v \in S\)

and \(uv\): \(uv \neq 0\) and \(u\): \((\sum_{v \in S} u v * v) = 0\)

let \(?a = v\)

let \(?S = S - \{v\}\)

let \(?u = \lambda i. (- u i) / u v\)

have \(th0\): \(?a \in P\) finite \(?S \subseteq P\)

using \(fS SP vS\) by auto

have \((\sum_{v \in ?S} ?u v * v) = (\sum_{v \in S} (- (\text{inverse} (u ?a))) * (u v * v)) - ?u v * v\)

using \(fS vS uw\) by (simp add: sum-diff1 field-simps)

also have \(. . . = \?a\)

unfolding scale-sum-right[ symmetric] \(u\) using \(uw\) by simp

finally have \((\sum_{v \in ?S} ?u v * v) = \?a\)

with \(th0\) show \(\exists a \in P. a \in \text{span} (P - \{a\})\)

unfolding span-explicit by (auto intro! : bexI[ where \(x = ?a\] exI[ where \(x = ?S\] exI[ where \(x = ?u\])

qed

lemma dependent-single\([\text{ simp}]\): dependent \(\{x\} \leftrightarrow x = 0\)

unfolding dependent-def by auto

lemma in-span-insert:

assumes \(a: a \in \text{span} (\text{insert } b S)\)

and \(na: a \notin \text{span } S\)

shows \(b \in \text{span } (\text{insert } a S)\)

proof –

from span-breakdown[of \(b\) insert \(b S a\), OF insertI1 \(a\)]

obtain \(k\) where \(k: a - k * b \in \text{span} (S - \{b\})\) by auto

have \(k \neq 0\)

proof

assume \(k = 0\)

with \(k\) span-mono[of \(S - \{b\}\) \(S\)] have \(a \in \text{span } S\) by auto

with \(na\) show \(False\) by blast

qed
then have eq: \[ b = (1/k) * s a - (1/k) * s (a - k * s b) \]
  by (simp add: algebra-simps)

from k have \[(1/k) * s (a - k * s b) \in \text{span} (S - \{b\})\]
  by (rule span-scale)
also have ... \[\subseteq \text{span} (\text{insert} \ a \ S)\]
  by (rule span-mono) auto
finally show \[?thesis\]
  using k by (subst eq) (blast intro: span-diff span-scale span-base)
qed

lemma dependent-insertD: assumes a: a \notin \text{span} \ S and S: dependent (insert a S)
shows dependent S
proof
  have a \notin \ S using a by (auto dest: span-base)
  obtain b where b: b = a \lor b \in \ S \ b \in \text{span} (\text{insert} \ a \ S - \{b\})
    using S unfolding dependent-def by blast
  have b \neq a b \in \ S
    using b : (a \notin \ S) a by auto
  with b have *: b \in \text{span} (\text{insert} \ a \ (S - \{b\}))
    by (auto simp: insert-Diff-if)
  show dependent S
proof cases
  assume b \in \text{span} (S - \{b\}) with \langle b \in S \rangle show ?thesis
    by (auto simp add: dependent-def)
next
  assume b \notin \text{span} (S - \{b\})
  with \langle a \notin \ S \rangle a \in \text{span} (S - \{b\}) by (rule in-span-insert)
  with a show ?thesis
    using \langle b \in S \rangle by (auto simp: insert-absorb)
qed
qed

lemma independent-insertI: a \notin \text{span} \ S \Longrightarrow independent S \Longrightarrow independent (insert a S)
by (auto dest: dependent-insertD)

lemma independent-insert:
  independent (insert a S) \longleftrightarrow (if a \in S then independent S else independent S \land a \notin \text{span} S)
proof
  have a \notin S \Longrightarrow a \in \text{span} S \Longrightarrow dependent (insert a S)
    by (auto simp: dependent-def)
  then show ?thesis
    by (auto intro: dependent-mono simp: independent-insertI)
qed

lemma maximal-independent-subset-extend:
  assumes S \subseteq V independent S
obtains $B$ where $S \subseteq B \subseteq V$ independent $B \subseteq V \subseteq \text{span } B$

proof –
let $\mathcal{C} = \{ B, S \subseteq B \land \text{independent } B \land B \subseteq V \}$

have $\exists M \in \mathcal{C}. \forall X \in \mathcal{C}. M \subseteq X \rightarrow X = M$

proof (rule subset-Zorn)

fix $C : \text{'b set set}$ assume subset.chain $\mathcal{C} C$
then have $\exists M \in \mathcal{C}. \forall X \in \mathcal{C}. M \subseteq X \rightarrow X = M$

proof (rule subset-Zorn)

fix $C ::'b set set$
assume $\subseteq$.chain-def
unfolding subset.chain-def by blast+

show $\exists U \in \mathcal{C}. \forall X \in C. X \subseteq U$

proof cases
assume $C = \{ \}$ with assms show $\exists U \in \mathcal{C}. \forall X \in C. X \subseteq U$
proof cases
assume $C \neq \{ \}$
with $\subseteq$.chain-def $\Rightarrow$ $\subseteq$.independent-dir $\Rightarrow$ $\subseteq$.independent-dir $\Rightarrow$

next

assume $\neq \{ \}$

moreover have independent $\bigcup C$
proof (intro independent-dir)

moreover have $\bigcup C \subseteq V$
proof (intro $\subseteq$.chain-def)

ultimately show $\exists U \in \mathcal{C}. \forall X \in C. X \subseteq U$
proof (meson $\subseteq$.chain-def)

qed

qed

then obtain $B$ where $B$, independent $B \subseteq V \subseteq B$
and $\subseteq$.max $\Rightarrow$ $\subseteq$.independent-dir $\Rightarrow$ $\subseteq$.independent-dir $\Rightarrow$

by $\subseteq$.chain-def

moreover

assume $\not\subseteq V \subseteq \text{span } B$
then obtain $v$ where $v \in V v \not\in \text{span } B$
proof (meson $\subseteq$.chain-def)

with $B$ have independent $\langle \text{insert } v B \rangle$
proof (meson $\subseteq$.chain-def)

ultimately show $\exists U \in \mathcal{C}. \forall X \in C. X \subseteq U$
proof (meson $\subseteq$.chain-def)

qed

lemma maximal-independent-subset:

obtains $B$ where $B \subseteq V$ independent $B \subseteq V \subseteq \text{span } B$
proof (meson maximal-independent-subset-extend [of $\{ \}$] independent-dir [of $\{ \}$])

Extends a basis from $B$ to a basis of the entire space.
definition extend-basis :: 'b set ⇒ 'b set
where extend-basis B = (SOME B'. B ⊆ B' ∧ independent B' ∧ span B' = UNIV)

lemma assumes B: independent B
shows extend-basis-superset: B ⊆ extend-basis B
  and independent-extend-basis: independent (extend-basis B)
  and span-extend-basis[simp]: span (extend-basis B) = UNIV
proof –
define p where p B' ≡ B ⊆ B' ∧ independent B' ∧ span B' = UNIV for B'
obtain B' where p B'
  using maximal-independent-subset-extend[OF subset-UNIV B]
  by (metis top.extremum-uniqueI p-def)
then have p (extend-basis B)
  unfolding extend-basis-def p-def [symmetric] by (rule someI)
then show B ⊆ extend-basis B independent (extend-basis B) span (extend-basis B) = UNIV
  by (auto simp: p-def)
qed

lemma in-span-delete:
  assumes a: a ∈ span S and na: a /∈ span (S - {b})
  shows y ∈ span (insert a (S - {b}))
  by (metis Diff-empty Diff-insert0 a in-span-insert insert-Diff na)

lemma span-redundant: x ∈ span S ⇒ span (insert x S) = span S
unfolding span-def by (rule hull-redundant)

lemma span-trans: x ∈ span S ⇒ y ∈ span (insert x S) ⇒ y ∈ span S
by (simp only: span-redundant)

lemma span-insert-0[simp]: span (insert 0 S) = span S
by (metis span-zero span-redundant)

lemma span-delete-0[simp]: span(S - {0}) = span S
proof
  show span (S - {0}) ⊆ span S
    by (blast intro!: span-mono)
next
  have span S ⊆ span(insert 0 (S - {0}))
    by (blast intro!: span-mono)
  also have ... ⊆ span(S - {0})
    using span-insert-0 by blast
  finally show span S ⊆ span (S - {0})
qed

lemma span-image-scale:
  assumes finite S and nz: ∀x. x ∈ S ⇒ c x ≠ 0
shows \( \text{span} \left( (\lambda x. c \times s \times) \cdot S \right) = \text{span} S \)

using \text{assms}

proof (induction \( S \) arbitrary: \( c \))

\text{case} (\text{empty} \( c \)) \text{ show} \ ?\text{case} \ by \ \text{simp}

next

\text{case} (\text{insert} \ x \ F \ c)

\text{show} \ ?\text{case}

proof (intro \text{set-eqI} \ \text{iffI})

\text{fix} \ y

\text{assume} \ y \in \text{span} \left( (\lambda x. c \times s \times) \cdot \text{insert} \ x \ F \right)

\text{then show} \ y \in \text{span} (\text{insert} \ x \ F)

using \text{insert} \ by \ (\text{force simp: span-breakdown-eq})

next

\text{fix} \ y

\text{assume} \ y \in \text{span} (\text{insert} \ x \ F)

\text{then show} \ y \in \text{span} (\lambda x. c \times s \times) \cdot \text{insert} \ x \ F)

using \text{insert}

apply (\text{clarsimp simp: span-breakdown-eq})

apply (\text{rule-tac} x=k / c \ x \ \text{in exI})

by \ \text{simp}

qed

qed

\text{lemma} exchange-lemma:

\text{assumes} \ f: \ \text{finite} \ T

\text{and} \ i: \ \text{independent} \ S

\text{and} \ sp: \ S \subseteq \text{span} \ T

\text{shows} \ \exists t'. \ \text{card} \ t' = \text{card} \ T \land \text{finite} \ t' \land S \subseteq t' \land t' \subseteq S \cup T \land S \subseteq \text{span} t'

using \text{f i sp}

proof (induct \text{card} (T - S) \text{ arbitrary:} \ S \ T \text{ rule: less-induct})

\text{case} less

\text{note} \ ft = (\text{finite} \ T) \ \text{and} \ S = (\text{independent} \ S) \ \text{and} \ sp = (S \subseteq \text{span} \ T)

\text{let} \ ?P = \lambda t'. \ \text{card} \ t' = \text{card} \ T \land \text{finite} \ t' \land S \subseteq t' \land t' \subseteq S \cup T \land S \subseteq \text{span} t'

\text{show} \ ?\text{case}

proof (cases \ S \subseteq T \lor T \subseteq S)

\text{case} True

\text{then show} \ ?\text{thesis}

proof

\text{assume} \ S \subseteq T \text{ then show} \ ?\text{thesis}

by (\text{metis ft Un-commute sp sup-le1})

next

\text{assume} \ T \subseteq S \text{ then show} \ ?\text{thesis}

by (\text{metis Un-absorb sp spanning-subset-independent[OF - S sp] ft})

qed

next

\text{case} False

\text{then have} \ st: \neg S \subseteq T \neg T \subseteq S

by \ auto

from \text{st}(2) \text{ obtain} \ b \ \text{where} \ b: \ b \in T \ b \notin S
by blast
from b have T - {b} - S ⊆ T - S
  by blast
then have cardlt: card (T - {b} - S) < card (T - S)
  using ft by (auto intro: psubset-card-mono)
from b ft have ct0: card T ≠ 0
  by auto
show ?thesis
proof (cases S ⊆ span (T - {b}))
  case True
  from ft have ftb: finite (T - {b})
    by auto
  from less(1)[OF cardlt ftb S True]
  obtain U where U:
    card U = card (T - {b})
    S ⊆ U
    U ⊆ S ∪ (T - {b})
    S ⊆ span U
    and fu: finite U by blast
  let ?w = insert b U
  have th0: S ⊆ insert b U
    using U by blast
  have th1: insert b U ⊆ S ∪ T
    using U b by blast
  have bu: b /∈ U
    using b U by blast
  from U(1) ft b have card U = (card T - 1)
    by auto
  then have th2: card (insert b U) = card T
    using card-insert-disjoint[OF fu bu] ct0 by auto
  from U(4) have S ⊆ span U
  .
  also have ... ⊆ span (insert b U)
  by (rule span-mono)
  blast
  finally have th3: S ⊆ span (insert b U)
  .
  from th0 th1 th2 th3 fu have th: ?P ?w
  .
    by blast
  from th show ?thesis by blast
next
  case False
  then obtain a where a: a ∈ S a /∈ span (T - {b})
    by blast
  have ab: a ≠ b
    using a b by blast
  have at: a /∈ T
    using a ab span-base[of a T - {b}] by auto
  have mlt: card ((insert a (T - {b})) - S) < card (T - S)
    using cardlt ft a b by auto
  have ft': finite (insert a (T - {b}))
    using ft by auto
  have sp': S ⊆ span (insert a (T - {b}))
  proof
    fix x
assumes \( \mathbf{xs} : x \in S \)
have \( T : T \subseteq \text{insert } b \ (\text{insert } a \ (T - \{b\})) \)
using \( b \) by auto
have \( \mathbf{bs} : b \in \text{span} \ (\text{insert } a \ (T - \{b\})) \)
by (rule \text{in-span-delete}) (use \( a \) \( \text{sp in} \) auto)
from \( \mathbf{xs} \) \( \text{sp} \) have \( x \in \text{span } T \)
by blast
with \( \text{span-mono}[OF T] \) have \( x : x \in \text{span} \ (\text{insert } b \ (\text{insert } a \ (T - \{b\}))) \).
from \( \text{span-trans}[OF bs x] \) show \( x \in \text{span} \ (\text{insert } a \ (T - \{b\})) \).
qed
from \( \text{less}(1)[OF \mathbf{mlt ft'} S \mathbf{sp'}] \) obtain \( \mathbf{U} \) where
\( \mathbf{U} : \text{card } \mathbf{U} = \text{card} \ (\text{insert } a \ (T - \{b\})) \)
finite \( \mathbf{U} \) \( S \subseteq \mathbf{U} \) \( \mathbf{U} \subseteq S \cup \text{insert } a \ (T - \{b\}) \)
\( S \subseteq \text{span } \mathbf{U} \) by blast
from \( \mathbf{U} \) \( a \) \( ft \) \( at \) \( ct0 \) have \( ?P \) \( \mathbf{U} \)
by auto
then show \( ?\text{thesis} \) by blast
qed
qed
lemma \( \text{independent-span-bound} \):
assumes \( \mathbf{f} : \text{finite } T \)
and \( \mathbf{i} : \text{independent } S \)
and \( \mathbf{sp} : S \subseteq \text{span } T \)
shows \( \text{finite } S \land \text{card } S \leq \text{card } T \)
by (metis \text{exchange-lemma}[OF \mathbf{f i sp}] \text{finite-subset card-mono})
lemma \( \text{independent-explicit-finite-subsets} \):
\( \text{independent } \mathbf{A} \leftrightarrow (\forall S \subseteq \mathbf{A}. \text{finite } S \rightarrow (\forall u. (\sum v \in S. u \ v * s \ v) = 0 \rightarrow (\forall v \in S. u \ v = 0))) \)
unfolding \( \text{dependent-explicit} [\text{of } \mathbf{A}] \) by (simp add: disj-not2)
lemma \( \text{independent-if-scalars-zero} \):
assumes \( \mathbf{fin-A} : \text{finite } \mathbf{A} \)
and \( \mathbf{sum} : (\forall x. (\sum x \in \mathbf{A}. f \ x * s \ x) = 0 \rightarrow x \in \mathbf{A} \rightarrow f \ x = 0) \)
shows \( \text{independent } \mathbf{A} \)
proof (unfold independent-explicit-finite-subsets, clarify)
fix \( S \ v \) and \( u :: 'b \Rightarrow 'a \)
assume \( S : S \subseteq \mathbf{A} \) \( \text{and } v : v \in S \)
let \( ?g = \lambda x. \text{if } x \in S \text{ then } u \ x \text{ else } 0 \)
have \( (\sum v \in \mathbf{A}. ?g \ v * s \ v) = (\sum v \in S. u \ v * s \ v) \)
using \( \mathbf{fin-A} \) by (auto intro!: \text{sum.mono-neutral-cong-right})
also assume \( (\sum v \in S. u \ v * s \ v) = 0 \)
finally have \( ?g \ v = 0 \) using \( v \mathbf{S} \) \( \text{sum by force} \)
thus \( u \ v = 0 \) unfolding \( \text{if-}[OF \mathbf{v}] \).
qed
lemma \( \text{bij-if-span-eq-span-bases} \):
assumes $B$: independent $B$ and $C$: independent $C$
and $eq$: span $B = \text{span} C$
shows $\exists f$: bij-betw $f B C$

proof cases
assume finite $B \lor$ finite $C$
then have finite $B \land$ finite $C \land \text{card } C = \text{card } B$
using independent-span-bound[of $B$ $C$] independent-span-bound[of $C$ $B$]
assms span-superset[of $B$] span-superset[of $C$]
by auto
then show $?thesis$
by (auto intro!: finite_same_card_bij)

next
assume $\neg$ (finite $B \lor$ finite $C$)
then have infinite $B$ infinite $C$ by auto
{ fix $B$ $C$ assume $B$: independent $B$ and $C$: independent $C$ and infinite $B$
and $eq$: span $B = \text{span} C$
let $?R = \text{representation } B$ and $?R' = \text{representation } C$
let $?U = ?\lambda c. \{ v. ?R c v \neq 0 \}$

have in-span-$C$ [simp, intro]: $\forall b \in B \implies b \in \text{span } C$; for $b$ unfolding eq[symmetric]
by (rule span-base)

have in-span-$B$ [simp, intro]: $\forall c \in C \implies c \in \text{span } B$; for $c$ unfolding eq by (rule span-base)

have $(B \subseteq (\bigcup C \cap U ))$

proof
fix $b$
assume $\neg (b \in B)$

have $(b \in \text{span } C)$
using $(b \in B)$ unfolding eq[symmetric]
by (rule span-base)

have $(\sum v \mid ?R' b v \neq 0, \sum w \mid ?R v w \neq 0. ( ?R' b v \ast ?R v w ) \ast s w ) =$
$(\sum v \mid ?R' b v \neq 0. ?R' b v \ast s ( \sum w \mid ?R v w \neq 0. ?R v w \ast s w ))$
by (simp add: scale-sum-right)

also have $(\ldots = \{ v. ?R' b v \neq 0. ?R' b v \ast s v \})$
by (auto simp: sum-nonzero-representation-eq $B$ eq span-base_representation-ne-zero)

also have $(\ldots = b)$
by (rule sum-nonzero-representation-eq $[OF C \ (b \in \text{span } C)]$)

finally have $?R b b = ?R ( \sum v \mid ?R' b v \neq 0. \sum w \mid ?R v w \neq 0. ( ?R' b v\ast ?R v w ) \ast s w) b$
by simp

also have $(\ldots = ( \sum i \in \{ v. ?R' b v \neq 0 \}. ?R ( \sum w \mid ?R i w \neq 0. ( ?R' b i \ast ?R i w ) \ast s w )) b)$
by (subst representation-sum[OF $B$])
(auto intro: span-sum span-scale span-base_representation-ne-zero)

also have $(\ldots = ( \sum j \in \{ w. ?R i w \neq 0 \}. ?R ( ( ?R' b i \ast ?R i j ) \ast s j ) b)$
by (subst representation-sum[OF $B$])
(auto simp add: span-sum span-scale span-base_representation-ne-zero)

also have $(\ldots = ( \sum v \mid ?R' b v \neq 0. \sum w \mid ?R v w \neq 0. ?R' b v \ast ?R v w \ast ?R w b))$

using $(b \in B)$ by (simp add: representation-scale[OF $B$] span-base_representation-ne-zero)
finally have \((\sum v \mid \not \exists R' \ b \ v \neq 0. \sum w \mid \not \exists R \ v \ w \neq 0. \ \not \exists R' \ b \ v \not \exists R \ v \ w \not \exists R \ w \ b) \neq 0\)
using representation-basis[of B \of b \in B] by auto
then obtain \(v \ w\) where \(\not \exists R' \ b \ v \neq 0\) and \(\not \exists R \ v \ w \neq 0\) and \(\not \exists R' \ b \ v \not \exists R \ v \ w \not \exists R \ w \ b\)
\(\neq 0\)
by (blast elim: sum.not-neutral-contains-not-neutral)
with representation-basis[of B]
then have \(\langle \exists R' \ b \ v \neq 0 \rangle \langle \exists R \ v \ b \neq 0 \rangle\)
by (auto split: if-splits)
then show \(\langle b \in (\bigcup c \in C. \ ?U c) \rangle\)
by (auto intro: span-base representation-ne-zero eq)
qed

from this[of B C] this[of C B] B C eq \(\langle \exists R \ v \ b \neq 0 \rangle \langle \exists R \ v \ b \neq 0 \rangle\)
then have \(\langle b \in (\bigcup c \in C. \ ?U c) \rangle\)
by (intro bij-betw-same-card)
then show \(\langle b \in (\bigcup c \in C. \ ?U c) \rangle\)
by (auto intro: span-base representation-ne-zero eq)
qed

definition dim :: \('b set \Rightarrow \nat\)
where \(\dim V = (if \ \exists \ b. \ independent \ b \ \& \ \& \ span \ b = \span V \ then \ \card (\some b. \ independent \ b \ \& \ \& \ span \ b = \span V) \ else 0)\)

lemma dim-eq-card:
assumes \(BV: \ span B = \span V \ and \ B: \ independent B\)
shows \(\dim V = \card B\)
proof –
define \(p\) where \(p \ b \equiv \ independent \ b \ \& \ \& \ span \ b = \span V \ for \ b\)
have \(p \ (\some B. \ p B)\)
using assms by (intro someI[of p B]) (auto simp: p-def)
then have \(\exists f. \ bij-betw f B \ (\some B. \ p B)\)
by (subst (asm) p-def, intro bij-if-span-eq-span-bases[of B]) (simp-all add: BV)
then have \(\card B = \card (\some B. \ p B)\)
by (auto intro: bij-betw-same-card)
then show \(?thesis\)
using \(BV\) \(B\)
by (auto simp add: dim-def p-def)
qed

lemma basis-card-eq-dim:
\(B \subseteq V \implies V \subseteq \span B \implies independent B \implies \card B = \dim V\)
THEORY “Vector-Spaces”

lemma basis-exists:
  obtains B where B ⊆ V independent B V ⊆ span B card B = dim V
by (meson basis-card-eq-dim empty-subsetI independent-empty maximal-independent-subset-extend)

lemma dim-eq-card-independent: independent B ⇒ dim B = card B
by (rule dim-eq-card[OF refl])

lemma dim-span[simp]: dim (span S) = dim S
by (auto simp add: dim-def span-span)

lemma dim-span-eq-card-independent: independent B ⇒ dim (span B) = card B
by (simp add: dim-eq-card)

lemma dim-le-card: assumes V ⊆ span W finite W shows dim V ≤ card W
proof –
  obtain A where independent A A ⊆ V V ⊆ span A
  using maximal-independent-subset[of V] by force
  with assms independent-span-bound[of W A] basis-card-eq-dim[of A V]
  show ?thesis by auto
qed

lemma span-eq-dim: span S = span T ⇒ dim S = dim T
by (metis dim-span)

corollary dim-le-card':
  finite s ⇒ dim s ≤ card s
by (metis basis-exists card-mono)

lemma span-card-ge-dim: B ⊆ V ⇒ V ⊆ span B ⇒ finite B ⇒ dim V ≤ card B
by (simp add: dim-le-card)

lemma dim-unique:
  B ⊆ V ⇒ V ⊆ span B ⇒ independent B ⇒ card B = n ⇒ dim V = n
by (metis basis-card-eq-dim)

lemma subspace-sums: [subspace S; subspace T] ⇒ subspace {x + y | x y. x ∈ S ∧ y ∈ T}
apply (simp add: subspace-def)
apply (intro conjI impI allI; clasimp simp: algebra-simps)
using add.left-neutral apply blast
apply (metis add.assoc)
using scale-right-distrib by blast

end

lemma linear-iff: linear s1 s2 f ←→
(vector-space s1 ∧ vector-space s2 ∧ (∀ x y. f (x + y) = f x + f y) ∧ (∀ c x. f

THEORY "Vector-Spaces"

(s1 c x) = s2 c (f x))

unfolding linear-def module-hom-iff vector-space-def module-def by auto

context begin
qualified lemma linear-compose: linear s1 s2 f ⇒ linear s2 s3 g ⇒ linear s1 s3 (g o f)

unfolding module-hom-iff-linear[symmetric] by (rule module-hom-compose)
end

locale vector-space-pair = vs1: vector-space s1 + vs2: vector-space s2
for s1 :: 'a::field ⇒ 'b::ab-group-add ⇒ 'b (infixr * 75)
and s2 :: 'a::field ⇒ 'c::ab-group-add ⇒ 'c (infixr * 75)
begin
context fixes f assumes linear s1 s2 f begin
interpretation linear s1 s2 f by fact
lemmas — from locale module-hom
        linear-0 = zero
and linear-add = add
and linear-scale = scale
and linear-neg = neg
and linear-diff = diff
and linear-sum = sum
and linear-inj-on-iff-eq-0 = inj-on-iff-eq-0
and linear-inj-iff-eq-0 = inj-iff-eq-0
and linear-subspace-image = subspace-image
and linear-subspace-vimage = subspace-vimage
and linear-subspace-kernel = subspace-kernel
and linear-span-image = span-image
and linear-dependent-inj-imageD = dependent-inj-imageD
and linear-eq-0-on-span = eq-0-on-span
and linear-independent-injective-image = independent-injective-image
and linear-inj-on-span-independent-image = inj-on-span-independent-image
and linear-inj-on-span-iff-independent-image = inj-on-span-iff-independent-image
and linear-subspace-linear-preimage = subspace-linear-preimage
and linear-spans-image = spans-image
and linear-spanning-surjective-image = spanning-surjective-image
end

sublocale module-pair
rewrites module-hom = linear
by unfold-locales (fact module-hom-eq-linear)

lemmas — from locale module-pair
        linear-eq-0-on-span = module-hom-eq-on-span
and linear-compose-scale-right = module-hom-scale
and linear-compose-add = module-hom-add
and linear-zero = module-hom-zero
THEORY "Vector-Spaces"

and linear-compose-sub = module-hom-sub
and linear-compose-neg = module-hom-neg
and linear-compose-scale = module-hom-compose-scale

lemma linear-indep-image-lemma:
assumes lf: linear s1 s2 f
and fB: finite B
and ifB: vs2.independent (f ' B)
and fi: inj-on f B
and xsB: x ∈ vs1.span B
and fx: f x = 0
shows x = 0
using fB ifB fi xsB fx
proof (induction B arbitrary: x rule: finite-induct)
  case empty
  then show ?case by auto
next
  case (insert a b x)
  have th0: f ' b ⊆ f ' (insert a b)
    by (simp add: subset-insertI)
  have ifb: vs2.independent (f ' b)
    using vs2.independent-mono insert.prems(1) th0 by blast
  have fB: inj-on f b
    using insert.prems(2) by blast
  from vs1.span-breakdown[of a insert a b, simplified, OF insert.prems(3)]
  obtain k where k: x - k *a a ∈ vs1.span (b - {a})
    by blast
  have f (x - k *a a) ∈ vs2.span (f ' b)
    unfolding linear-span-image[OF lf]
    using insert.prems(2) k by auto
  then have f x - k *b f a ∈ vs2.span (f ' b)
    by (simp add: linear-diff linear-scale lf)
  then have th': -k *b f a ∈ vs2.span (f ' b)
    using insert.prems(4) by simp
  have xsB: x ∈ vs1.span b
  proof (cases k = 0)
    case True
    with k have x ∈ vs1.span (b - {a}) by simp
    then show ?thesis using vs1.span-mono[of b - {a} b]
      by blast
  next
    case False
    from inj-on-image-set-diff[OF insert.prems(2), of insert a b {a}, symmetric]
    have f' insert a b = f' {a} = f' (insert a b - {a}) by blast
    then have f a ≠ vs2.span (f' b)
      using vs2.dependent-def insert.prems(2) insert.prems(1) by fastforce
    moreover have f a ∈ vs2.span (f' b)
      using False vs2.span-scale[OF th', of - 1 / k] by auto
    ultimately have False
THEORY “Vector-Spaces”

by blast
then show ?thesis by blast
qed

show x = 0
using ifb fib xsb insert.IH insert.prems(4) by blast
qed

lemma linear-eq-on:
assumes l: linear s1 s2 f linear s1 s2 g
assumes x: x ∈ vs1.span B and eq: ∀b. b ∈ B ⇒ f b = g b
shows f x = g x
proof
interpret d: linear s1 s2 λ x. f x - g x
using l by (intro linear-compose-sub) (auto simp: module-hom-iff-linear)
have f x - g x = 0
by (rule d.eq-0-on-span[OF - x]) (auto simp: eq)
then show ?thesis by auto
qed

definition construct :: 'b set ⇒ ('b ⇒ 'c) ⇒ ('b ⇒ 'c)
where construct B g v = (∑b | vs1.representation (vs1.extend-basis B) v b ≠ 0.
    vs1.representation (vs1.extend-basis B) v b * b (if b ∈ B then g b else 0))

lemma construct-cong: (∀b. b ∈ B ⇒ f b = g b) ⇒ construct B f = construct B g
  unfolding construct-def by (rule ext, auto intro!: sum.cong)

lemma linear-construct:
  assumes B[simp]: vs1.independent B
  shows linear s1 s2 (construct B f)
  unfolding module-hom-iff-linear iff
proof safe
  have cB[simp]: vs1.independent (vs1.extend-basis B)
    using vs1.independent-extend-basis[OF B].
  let ?R = vs1.representation (vs1.extend-basis B)
  fix c x y
  have construct B f (x + y) =
    (∑b∈{b. ?R x b ≠ 0} ∪ {b. ?R y b ≠ 0}. ?R (x + y) b * b (if b ∈ B then f b else 0))
    by (auto intro!: sum.mono-neutral-cong-left simp: vs1.finite-representation
      vs1.representation-add construct-def)
  also have ... = construct B f x + construct B f y
    by (auto simp: construct-def vs1.representation-add vs2.scale-left-distrib sum.distrib
      intro!: arg-cong2[where f=+] sum.mono-neutral-cong-right vs1.finite-representation)
  finally show construct B f (x + y) = construct B f x + construct B f y .

show construct B f (c * a x) = c * b construct B f x
by (auto simp del: vs2.scale-scale intro!: sum.mono-neutral-cong-left vs1.finite-representation)
theory "Vector-Spaces"

simp add: construct-def vs2.scale-scale[symmetric] vs1.representation-scale vs2.scale-sum-right)

qed intro-locales

lemma construct-basis:
assumes B[simp]: vs1.independent B and b: b \in B
shows construct B f b = f b
proof
  have *: vs1.representation (vs1.extend-basis B) b = (\lambda v. if v = b then 1 else 0)
  using vs1.extend-basis-superset[OF B b]
  by (intro vs1.representation-basis vs1.independent-extend-basis) auto
  then have \{ v. vs1.representation (vs1.extend-basis B) b v \neq 0 \} = \{ b \}
  by auto
  then show ?thesis
    unfolding construct-def by (simp add: * b)
qed

lemma construct-outside:
assumes B: vs1.independent B and v: v \in vs1.span (vs1.extend-basis B - B)
shows construct B f v = 0
unfolding construct-def
proof (clarsimp intro!: sum.neutral simp del: vs2.scale-eq-0-iff)
  fix b assume b \in B
  then have vs1.representation (vs1.extend-basis B - B) v b = 0
  using vs1.representation-ne-zero[of vs1.extend-basis B - B v b] by auto
  moreover have vs1.representation (vs1.extend-basis B) v = vs1.representation (vs1.extend-basis B - B) v
  using vs1.representation-extend[OF vs1.independent-extend-basis[of B v] by auto
  ultimately show vs1.representation (vs1.extend-basis B) v b \ast b f b = 0
    by simp
qed

lemma construct-add:
assumes B[simp]: vs1.independent B
shows construct B (\lambda x. f x + g x) v = construct B f v + construct B g v
proof (rule linear-eq-on)
  show v \in vs1.span (vs1.extend-basis B) by simp
  show b \in vs1.extend-basis B \implies construct B (\lambda x. f x + g x) b = construct B f b + construct B g b for b
  using construct-outside[of B vs1.span-base, of b] by (cases b \in B) (auto simp: construct-basis)
qed (intro linear-compose-add linear-construct B)+

lemma construct-scale:
assumes B[simp]: vs1.independent B
shows construct B (\lambda x. c \ast b f x) v = c \ast b construct B f v
proof (rule linear-eq-on)
  show v \in vs1.span (vs1.extend-basis B) by simp
show $b \in \text{vs1.extend-basis } B \implies \text{construct } B (\lambda x. \ c*b\ f\ x)\ b = c*b \text{ construct } B \ f \ b \text{ for } b$

using construct-outside[OF $B$ vs1.span-base, of $b$] by (cases $b \in B$) (auto simp: construct-basis)

qed (intro linear-construct module-hom-scale $B$)+

lemma construct-in-span:
assumes $B$[simp]: vs1.independent $B$
shows $\text{construct } B \ f \ v \in \text{vs2.span } (f ' B)$
proof (cases finite $S$)
  case True then show ?thesis using $lS$ by induct (simp-all add: linear-zero linear-compose-add)
next
  case False then show ?thesis by (simp add: linear-zero)
qed

lemma linear-compose-sum:
assumes $lS$: $\forall a \in S. \ \text{linear } s1\ s2\ (f\ a)$
shows $\text{linear } s1\ s2\ (\lambda x. \ \sum (\lambda a. \ f a\ x)\ S)$
proof (cases finite $S$)
  case True
  then show ?thesis using $lS$ by induct (simp-all add: linear-zero linear-compose-add)
next
  case False
  then show ?thesis by (simp add: linear-zero)
qed

lemma in-span-in-range-construct:
$x \in \text{range } (\text{construct } B \ f) \text{ if } i: \ \text{vs1.independent } B \ \text{and } x: \ x \in \text{vs2.span } (f ' B)$
proof (interpret linear ($*a$) ($*b$) construct $B$)
  using $i$ by (rule linear-construct)
  obtain $bb :: \ ('b \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'b\ \text{set} \Rightarrow 'b\ \text{where}$
    $\forall x0\ x1\ x2. \ (\exists v4. \ v4 \in x2 \land x1 \land v4 \neq x0\ v4) = (bb\ x0\ x1\ x2 \in x2 \land x1 \land (bb\ x0\ x1\ x2) \neq x0\ (bb\ x0\ x1\ x2))$
    by moura
  then have $f2: \ \forall B\ Ba\ f\ fa. \ (B \neq Ba \lor \ \exists v4. \ v4 \in x2 \land x1 \land v4 \neq x0\ v4) = (bb\ x0\ x1\ x2 \neq x0\ (bb\ x0\ x1\ x2))$
by (meson image-cong)
have vs1.span B ⊆ vs1.span (vs1.extend-basis B)
  by (simp add: vs1.extend-basis-superset[OF i] vs1.span-mono)
then show x ∈ range (construct B f)
  using f2 x by (metis (no-types) construct-basis[OF i, of - f]
  vs1.span-extend-basis[OF i] subsetD span-image spans-image)
qed

lemma range-construct-eq-span:
  range (construct B f) = vs2.span (f' B)
if vs1.independent B
by (auto simp: that construct-in-span in-span-in-range-construct)

lemma linear-independent-extend-subspace:
  — legacy: use construct instead
assumes vs1.independent B
shows ∃ g. linear s1 s2 g ∧ (∀ x ∈ B. g x = f x) ∧ range g = vs2.span (f'B)
by (rule exI[where x = construct B f])
(auto simp: linear-construct assms construct-basis range-construct-eq-span)

lemma linear-independent-extend:
  vs1.independent B =⇒ ∃ g. linear s1 s2 g ∧ (∀ x ∈ B. g x = f x)
using linear-independent-extend-subspace[of B f] by auto

lemma linear-exists-left-inverse-on:
assumes lf: linear s1 s2 f
assumes V: vs1.subspace V and f: inj-on f V
shows ∃ g. g' UNIV ⊆ V ∧ linear s2 s1 g ∧ (∀ v ∈ V. g (f v) = v)
proof —
interpret linear s1 s2 f by fact
obtain B where V-eq: V = vs1.span B and B: vs1.independent B
  using vs1.maximal-independent-subset[of V] vs1.span-minimal[OF - ⟨vs1.subspace V⟩]
  by (metis antisym-conv)
have f: inj-on f (vs1.span B)
  using f unfolding V-eq .
show ?thesis
proof (intro exI ballI conjI)
interpret p: vector-space-pair s2 s1 by unfold-locales
have fB: vs2.independent (f' B)
  using independent-injective-image[OF f fB] .
let ?g = p.construct (f' B) (the-inv-into B f)
show linear (?g) (?g a) ?g
  by (rule p.linear-construct[OF fB])
have ?g b ∈ vs1.span (the-inv-into B f' f B) for b
  by (intro p.construct-in-span fB)
moreover have the-inv-into B f' f B = B
  by (auto simp: image-comp comp-def the-inv-into-f f inj-on-subset[OF f]
  vs1.span-superset)
ultimately show \( ?g : \text{UNIV} \subseteq V \)
by (auto simp: V-eq)
have \((?g \circ f) v = \text{id} v\) if \(v \in \text{vs1.span} B\) for \(v\)
proof (rule vector-space-pair.linear-eq-on[where \(x=v\)])
  show vector-space-pair \((\ast a) (\ast a)\) by unfold-locales
  show linear \((\ast a) (\ast a)\) \((?g \circ f)\)
  proof (rule Vector-Spaces.linear-compose[of - (\ast b)])
    show linear \((\ast a) (\ast b) f\)
    by unfold-locales
  qed
  fact
  show linear \((\ast a) (\ast a)\) \(\text{id}\) by (rule vs1.linear-id)
  show \(v \in \text{vs1.span} B\) by fact
  have \(b \in B \implies (p.construct (f \circ B) \text{(the-inv-into } B f) \circ f) b = \text{id} b\) for \(b\)
  by (simp add: p.construct-boundary B the-inv-into-f-f inj-on-subset[OF f vs1.span-superset])
  qed
  then show \(v \in V \implies ?g (f v) = v\) for \(v\) by (auto simp: comp-def id-def V-eq)
qed

lemma linear-exists-right-inverse-on:
assumes \(lf: \text{linear } s1 s2 f\)
assumes \(\text{vs1.subspace } V\)
shows \(\exists g. \ g' \text{UNIV} \subseteq V \land \text{linear } s2 s1 g \land (\forall v \in f \circ V. f (g v) = v)\)
proof
  obtain \(B\) where \(V\text{-eq}: V = \text{vs1.span } B\) and \(B: \text{vs1.independent } B\)
  using \text{vs1.maximal-independent-subset[of } V\text{]} \text{vs1.minimal[of } f \text{vs1.span-superset]}\[OF f vs1.span-superset]\)
  qed
  then show \(v \in V \implies ?g (f v) = v\) for \(v\) by (auto simp: comp-def id-def V-eq)
qed

lemma linear-exists-right-inverse-on:
assumes \(lf: \text{linear } s1 s2 f\)
assumes \(\text{vs1.subspace } V\)
shows \(\exists g. \ g' \text{UNIV} \subseteq V \land \text{linear } s2 s1 g \land (\forall v \in f \circ V. f (g v) = v)\)
proof
  obtain \(B\) where \(V\text{-eq}: V = \text{vs1.span } B\) and \(B: \text{vs1.independent } B\)
  using \text{vs1.maximal-independent-subset[of } f \text{vs1.span-superset]}\)
  by (metis antisym-conv)
  obtain \(C\) where \(C: \text{vs2.independent } C\) and \(fB-C: f \circ B \subseteq \text{vs2.span } C C \subseteq f \circ B\)
  using \text{vs2.maximal-independent-subset[of } f \text{vs1.span-superset]}\)
  by metis
  then have \(\forall v \in C. \exists b \in B. v = f b\) by auto
  then obtain \(g\) where \(g: \forall v . v \in C \implies g v \in B \land v. v \in C \implies f (g v) = v\)
  by metis
  show \(?\thesis\)
  proof (intro exI ballI conjI)
    interpret \(p: \text{vector-space-pair } s2 s1\) by unfold-locales
    let \(?g = p.construct C g\)
    show linear \((\ast b) (\ast a) \ ?g\)
    by (rule p.linear-construct[of C])
    have \(?g v \in \text{vs1.span (f C) for } v\)
    by (rule p.construct-in-span[of C])
    also have \(\ldots \subseteq V\) unfolding \(V\text{-eq}\) using \(g\) by (intro vs1.span-mono) auto
    finally show \(?g' \text{UNIV} \subseteq V\) by auto
    have \((f \circ \ ?g) v = \text{id} v\) if \(v: v \in f \circ V\) for \(v\)
    proof (rule vector-space-pair.linear-eq-on[where \(x=v\)])
      show vector-space-pair \((\ast b) (\ast b)\) by unfold-locales
show linear (*b) (*b) (f ° ?g)
  by (rule Vector-Spaces.linear-compose[of - (*a)]) fact+
show linear (*b) (*b) id by (rule vs2.linear-id)
have vs2.span (f ° B) = vs2.span C
  using fB-C vs2.span mono[of C f ° B] vs2.span minimal[of fB vs2.span C]
  by auto
then show v ∈ vs2.span C
  using v linear-span image[of fB C vs2.span mono[of C f ° B]
  vs2.span minimal[of fB vs2.span C]
  by (auto simp add: V-eq)
qed
then show v ∈ f ° V ⇒ f (?g v) = v for v by (auto simp: comp-def id-def)
qed

lemma linear-inj-on-left-inverse:
  assumes lf: linear s1 s2 f
  assumes fi: inj-on f (vs1.span S)
  shows ∃ g. range g ⊆ vs1.span S ∧ linear s2 s1 g ∧ (∀ x∈vs1.span S. g (f x) = x)
  using linear-exists-left-inverse-on[of f vs1.span vs1.span fi]
  by (auto simp: linear-iff-module-hom)

lemma linear-injective-left-inverse: linear s1 s2 f ⇒ inj f ⇒ ∃ g. linear s2 s1 g ∧ g ° f = id
  using linear-inj-on-left-inverse[of f UNIV]
  by force

lemma linear-surj-right-inverse:
  assumes lf: linear s1 s2 f
  assumes sf: vs2.span T ⊆ f vs1.span S
  shows ∃ g. range g ⊆ vs1.span S ∧ linear s2 s1 g ∧ (∀ x∈vs2.span T. f (g x) = x)
  using linear-exists-right-inverse-on[of f vs1.span vs2.span, of S] sf
  by (force simp: linear-iff-module-hom)

lemma linear-surjective-right-inverse: linear s1 s2 f ⇒ surj f ⇒ ∃ g. linear s2 s1 g ∧ f ° g = id
  using linear-surj-right-inverse[of f UNIV UNIV]
  by (auto simp: fun-eq iff)

lemma finite-basis-to-basis-subspace-isomorphism:
  assumes s: vs1.span S
  and t: vs2.span T
  and d: vs1.dim S = vs2.dim T
  and fB: finite B
  and B: B ⊆ S vs1.independent B S ⊆ vs1.span B card B = vs1.dim S
  and fC: finite C
  and C: C ⊆ T vs2.independent C T ⊆ vs2.span C card C = vs2.dim T
shows $\exists f. \text{linear } s_1 s_2 f \wedge f : B \rightarrow C \wedge f : S \rightarrow T \wedge \text{inj-on } f S$

proof

- from $B(4) C(4)$ card-le-inj[of $B$ $C$] d obtain $f$ where
  $f : f : B \subseteq C$ inj-on $f$ $B$ using (finite $B$: finite $C$) by auto

- from linear-independent-extend[OF $B(2)$] obtain $g$ where
  $g : \text{linear } s_1 s_2 g \forall x \in B. \ g x = f x$ by blast

- interpret $g : \text{linear } s_1 s_2 g$

- have $\forall x \in B. \ g x = f x$ by blast

- note $g0 = \text{linear-indep-image-lemma}[OF $g(1)$ $fB$, unfolded $gBC$, OF $C(2)$ $gi$]

  \{ fix $x$ $y$

  \assumes $x : x \in S$ and $y : y \in S$ and $gxy : g x = g y$

  from $B(3) x y$ have $x' : x \in vs1.span B$ and $y' : y \in vs1.span B$

  by blast

  from $gxy$ have $th0 : g (x - y) = 0$

  by (simp add: g.diff)

  have $th1 : x - y \in vs1.span B$ using $x' y'$

  by (metis vs1.span-diff)

  have $x = y$ using $g0[OF th1 th0]$ by simp

  \}\n
- then have $giS : \text{inj-on } g S$ unfolding inj-on-def by blast

- have $g : g : S = vs2.span (g : B)$

  by (simp add: g.span-image)

- also have $\ldots = vs2.span C$

  unfolding $gBC$ ..

- also have $\ldots = T$

  using vs2.span-subspace[OF $C(1,3)$ $t$]

- finally have $gS : g : S = T$.

- from $g(1)$ $gS giS gBC$ show ?thesis

  by blast

qed
and \( \text{span-Basis} \) \( \text{span} \) \( \text{Basis} = \text{UNIV} \)

begin

definition dimension = card Basis

lemma finitelI-independent: independent \( B \) \( \longrightarrow \) finite \( B \)
using independent-span-bound[\( OF \) finite-Basis, of \( B \)] by (auto simp: span-Basis)

lemma dim-empty [simp]: \( \text{dim} \) \{ \} = 0
by (rule dim-unique[\( OF \) order-refl]) (auto simp: dependent-def)

lemma dim-insert:
\( \text{dim} \) (insert \( x \) \( S \)) = (if \( x \in \text{span} \) \( S \) then \( \text{dim} \) \( S \) else \( \text{dim} \) \( S + 1 \))
proof -
show \( \text{thesis} \)
proof (cases \( x \in \text{span} \) \( S \))
  case True then show \( \text{thesis} \)
  by (metis dim-span span-redundant)
next
case False
obtain \( B \) where \( B \subseteq \text{span} \) \( S \) independent \( B \) \( \text{span} \) \( S \) \( \subseteq \text{span} \) \( B \) card \( B = \text{dim} \) \( \text{span} \) \( S \)
using basis-exists [of \( \text{span} \) \( S \)] by blast
have \( \text{dim} \) \( (\text{span} \ (\text{insert} \ x \ S)) = \text{Suc} \ (\text{dim} \ S) \)
proof (rule dim-unique)
  show \( \text{insert} \ x \ B \subseteq \text{span} \ (\text{insert} \ x \ S) \)
  by (meson \( B \subseteq \text{span} \ S \) insertI1 insert-subset order-trans span-base span-mono subset-insertI)
  show \( \text{span} \ (\text{insert} \ x \ S) \subseteq \text{span} \ (\text{insert} \ x \ B) \)
  by (metis \( B \subseteq \text{span} \ S \) span-breakdown-eq span-subspace subsetI subset-span)
  show independent \( (\text{insert} \ x \ B) \)
  by (metis B(1-3) independent-insert span-subspace span-breakdown-eq span-subspace False)
  show \( \text{card} \ (\text{insert} \ x \ B) = \text{Suc} \ (\text{dim} \ S) \)
  using \( B \) False finitelI-independent by force
qed
then show \( \text{thesis} \)
by (metis False Suc-eq-plus1 dim-span)
qed

lemma dim-singleton [simp]: \( \text{dim} \) \{ \( x \) \} = (if \( x = 0 \) then \( 0 \) else \( 1 \))
by (simp add: dim-insert)

proposition choose-subspace-of-subspace:
assumes \( n \leq \text{dim} \ S \)
obtains \( T \) where \( \text{subspace} \ T \ T \subseteq \text{span} \ S \ \text{dim} \ T = n \)
proof -
  have \( \exists T. \text{subspace} \ T \land T \subseteq \text{span} \ S \land \text{dim} \ T = n \)
using assms
proof (induction n)
  case 0 then show ?case by (auto intro: exI[where x={0}] span-zero)
next
  case (Suc n)
  then obtain T where subspace T T ⊆ span S dim T = n
  by force
  then show ?case
  proof (cases span S ⊆ span T)
    case True
    have span T ⊆ span S
    by (simp add: T ⊆ span S) span-minimal
    then have dim S = dim T
    by (rule span-eq-dim [OF subset-antisym [OF True]])
    then show ?thesis
    using Suc.prems (dim T = n) by linarith
  next
    case False
    then obtain y where y: y ∈ S y /∈ T
    by (meson span-mono subsetI)
    then have span (insert y T) ⊆ span S
    by (metis (no-types) T ⊆ span S) subsetD insert-subset span-superset
    with (dim T = n) (subspace T) y show ?thesis
    apply (rule-tac x=span (insert y T) in exI)
    using span-eq-iff by (fastforce simp: dim-insert)
  qed
  qed
  with that show ?thesis by blast
qed

lemma basis-subspace-exists:
  assumes subspace S
  obtains B where finite B B ⊆ S independent B span B = S card B = dim S
  by (metis assms span-subspace basis-exists finite1-independent)

lemma dim-mono: assumes V ⊆ span W shows dim V ≤ dim W
proof -
  obtain B where independent B B ⊆ W W ⊆ span B
  using maximal-independent-subset[of W] by force
  span-mono[of B W] span-minimal[of - subspace-span, of W B]
  show ?thesis
  by (auto simp: finite-Basis span-Basis)
qed

lemma dim-subset: S ⊆ T ==> dim S ≤ dim T
using dim-mono[of S T] by (auto intro: span-base)
lemma dim-eq-0 [simp]:
\[ \dim S = 0 \iff S \subseteq \{0\} \]
by (metis basis-exists card-eq-0-iff dim-span finiteI-independent span-empty subset-empty subset-singletonD)

lemma dim-UNIV [simp]: \[ \dim \text{UNIV} = \text{card Basis} \]
using dim-eq-card [of Basis UNIV] by (simp add: independent-Basis span-Basis)

lemma independent-card-le-dim: assumes \( B \subseteq V \) and independent \( B \) shows card \( B \leq \dim V \)
by (subst dim-eq-card [symmetric, OF refl \( \langle \text{independent } B \rangle \)]) (rule dim-subset [OF \( \langle B \subseteq V \rangle \)])

lemma dim-subset-UNIV: \[ \dim S \leq \text{dimension} \]
by (metis dim-subset subset-UNIV dim-UNIV dimension-def)

lemma card-ge-dim-independent:
assumes \( BV: B \subseteq V \)
and \( iB: \text{independent } B \)
and \( dVB: \dim V \leq \text{card } B \)
shows \( V \subseteq \text{span } B \)
proof
fix \( a \)
assume \( aV: a \in V \)
{ 
assume \( aB: a \notin \text{span } B \)
then have \( iaB: \text{independent } (\text{insert } a \text{ B}) \)
using \( iB aV BV \) by (simp add: independent-insert)
from \( aV BV \) have \( \text{th0: insert } a \text{ B } \subseteq V \)
by blast
from \( aB \) have \( a \notin B \)
by (auto simp add: span-base)
with independent-card-le-dim [OF \( \text{th0 } iaB \)] \( dVB \) finiteI-independent [OF \( iB \)]
have False by auto
}
then show \( a \in \text{span } B \) by blast
qed

lemma card-le-dim-spanning:
assumes \( BV: B \subseteq V \)
and \( VB: V \subseteq \text{span } B \)
and \( fB: \text{finite } B \)
and \( dVB: \dim V \geq \text{card } B \)
shows independent \( B \)
proof –
{ 
fix \( a \)
assume \( a: a \in B \ a \in \text{span } (B - \{a\}) \)
from a fB have c0: card B ≠ 0
by auto
from a fB have cb: card (B − {a}) = card B − 1
by auto

{ fix x
assume x: x ∈ V
from a have eq: insert a (B − {a}) = B
by blast
from x VB have x↓: x ∈ span B
by blast
from span-trans[OF a(2), unfolded eq, OF x↓] have x ∈ span (B − {a}) .
} then have th1: V ⊆ span (B − {a})
by blast
have th2: finite (B − {a})
using fB by auto
from dim-le-card[OF th1 th2] have c: dim V ≤ card (B − {a}) .
from c c0 dVB cb have False by simp
} then show ?thesis
unfolding dependent-def by blast
qed

lemma card-eq-dim: B ⊆ V → card B = dim V → finite B → independent B ←→ V ⊆ span B
by (metis order-eq-iff card-le-dim-spanning card-ge-dim-independent)

lemma subspace-dim-equal:
assumes subspace S
and subspace T
and S ⊆ T
and dim S ≥ dim T
shows S = T
proof –
obtain B where B: B ≤ S independent B ∧ S ⊆ span B card B = dim S
using basis-exists[of S] by metis
then have span B ⊆ S
using span-mono[of B S] span-eq-iff[of S] assms by metis
then have span B = S
using B by auto
have dim S = dim T
using assms dim-subset[of S T] by auto
then have T ⊆ span B
using card-eq-dim[of B T] B finiteI-independent assms by auto
then show ?thesis
using assms ⟨span B = S⟩ by auto


**corollary** dim-eq-span:

shows $[S \subseteq T; \dim T \leq \dim S] \implies \span S = \span T$

by (simp add: span-mono subspace-dim-equal)

**lemma** dim-psubset:

span $S \subset \span T \implies \dim S < \dim T$

by (metis (no-types, hide-lams) dim-span less-le not-le subspace-dim-equal subspace-span)

**lemma** dim-eq-full:

shows $\dim S = \dimension \iff \span S = \UNIV$

by (metis dim-eq-span dim-subset-UNIV span-Basis span-span subset-UNIV dim-UNIV dim-span dimension-def)

**lemma** indep-card-eq-dim-span:

assumes independent $B$

shows $\finite B \land \card B = \dim (\span B)$

using dim-span-eq-card-independent[OF assms]

by auto

More general size bound lemmas.

**lemma** independent-bound-general:

independent $S \implies \finite S \land \card S \leq \dim S$

by (simp add: dim-eq-card-independent finiteI-independent)

**lemma** independent-explicit:

shows independent $B \iff \finite B \land (\forall c. (\sum_{v \in B} c v \cdot s v) = 0 \implies (\forall v \in B. c v = 0))$

using independent-bound-general

by (fastforce simp: dependent-finite)

**proposition** dim-sums-Int:

assumes subspace $S \subseteq T$

shows $\dim \{x + y | x, y \in S \land y \in T\} + \dim(S \cap T) = \dim S + \dim T$ (is $\dim ?ST + \cdot - = -$)

proof -

obtain $B$ where $B: B \subseteq S \cap T \subseteq \span B$

and indB: independent $B$

and cardB: card $B = \dim (S \cap T)$

using basis-exists by metis

then obtain $C D$ where $B \subseteq C C \subseteq S$ independent $C S \subseteq \span C$

and $B \subseteq D D \subseteq T$ independent $D T \subseteq \span D$

using maximal-independent-subset-extend

by (metis Int-subset-iff \{B \subseteq S \cap T\} indB)

then have finite $B$ finite $C$ finite $D$

by (simp-all add: finite-independent indB independent-bound-general)

have $B: B = C \cap D$

proof (rule spanning-subset-independent [symmetric])

show independent (C ∩ D) 
    by (meson independent C; independent-monono inf.coboundedI)
qed (use B; (C ⊆ S; (D ⊆ T); (B ⊆ C; (B ⊆ D); in auto)
then have Deq: D = B ∪ (D − C)
    by blast
have C∪D: C ∪ D ⊆ ?ST 
proof (simp, intro conjI)
    show C ⊆ ?ST
        using span-zero span-minimal [OF − (subspace T)]; (C ⊆ S) by force
show D ⊆ ?ST
    using span-zero span-minimal [OF − (subspace S)]; (D ⊆ T) by force
qed 
have a v = 0 if 0; (∑ v∈C. a v ∗ s v) + (∑ v∈D − C. a v ∗ s v) = 0
    and v: v ∈ C ∪ (D − C) for a v
proof − 
    have CsS: ∃x. x ∈ C − (C ⊆ S) (subspace S) subspace-scale by auto
    have eq: (∑ v∈D − C. a v ∗ s v) = − (∑ v∈C. a v ∗ s v)
    using that add-eq-0-iff by blast
    have (∑ v∈D − C. a v ∗ s v) ∈ S
        by (simp add: eq CsS (subspace S) subspace-neg subspace-sum)
    moreover have (∑ v∈D − C. a v ∗ s v) ∈ T
        apply (rule subspace-sum [OF (subspace T)]);
        by (meson DiffD1 ; D ⊆ T; (subspace T); subset-eq subspace-def)
    ultimately have (∑ v∈D − C. a v ∗ s v) ∈ span B
    using B by blast
then obtain e where e: (∑ v∈B. e v ∗ s v) = (∑ v∈D − C. a v ∗ s v)
    using span-finite [OF (finite B)] by force
have (∑ v∈C. c v ∗ s v) = 0; v ∈ C − (C ⊆ S) (independent C) independentD by blast
define cc where cc x = (if x ∈ B then a x + e x else a x) for x
have (simp): C ∩ B = B D ∩ B = B C ∩ − B = C − D B ∩ (D − C) = {} 
    using (B ⊆ C); (B ⊆ D); Beq by blast+
    have f2: (∑ v∈C ∩ D. e v ∗ s v) = (∑ v∈D − C. a v ∗ s v)
    using Beq e by presburger
    have f3: (∑ v∈C ∪ D. a v ∗ s v) = (∑ v∈C − D. a v ∗ s v) + (∑ v∈D − C. 
        a v ∗ s v) + (∑ v∈C ∩ D. a v ∗ s v)
    using (finite C) (finite D) dunion-diff2 by blast
    have f4: (∑ v∈C ∪ (D − C). a v ∗ s v) = (∑ v∈C. a v ∗ s v) + (∑ v∈D − C. a v ∗ s v) 
    by (meson Diff-disjoint (finite C) (finite D) finite-Diff dunion-disjoint)
    have (∑ v∈C. c c v ∗ s v) = 0
        using 0 f2 f3 f4
    apply (simp add: cc-def Beq (finite C) sum.If-cases algebra-simps sum.distrib 
        if-distrib if-distribR)
    apply (simp add: add.commute add.left-commute diff-eq)
    done 
then have ∃v. v ∈ C − (C ⊆ S) (independent C) (finite C) by blast
then have \( C_0: \forall v. v \in C - B \implies a \cdot v = 0 \)

by \( \text{simp add: cc-def Beq} \) meson

then have \( \text{simp}: (\sum x \in C - B. a \cdot x \ast s \cdot x) = 0 \)

by simp

have \( (\sum x \in C. a \cdot x \ast s \cdot x) = (\sum x \in B. a \cdot x \ast s \cdot x) \)

proof –

have \( C - D = C - B \)

using \( \text{Beq} \) by blast

then show \( \text{thesis} \)

using \( \text{Beq} \) by \( \text{auto} \)

qed

with \( \emptyset \) have \( D_0: (\sum v \in D. a \cdot v \ast s \cdot v) = 0 \)

by \( \text{subst Deq} \) \( \text{simp add: \langle \text{finite B} \rangle \langle \text{finite D} \rangle \text{sum-Un}} \)

then have \( D_0: \forall v. v \in D \implies a \cdot v = 0 \)

using \( \text{independent-explicit} \) \( \langle \text{independent D} \rangle \langle \text{finite D} \rangle \) by \( \text{blast} \)

show \( \text{thesis} \)

using \( v \ C_0 \ D_0 \ \text{Beq} \) by \( \text{blast} \)

qed

then have \( \text{independent} \ (C \cup (D - C)) \)

unfolding \( \text{independent-explicit} \)

using \( \text{independent-explicit} \)

by \( \text{simp add: independent-explicit \langle \text{finite C} \rangle \langle \text{finite D} \rangle \text{sum-Un del: Un-Diff-cancel}} \)

then have \( \text{indCUD: independent} \ (C \cup D) \) by simp

have \( \text{dim} \ (S \cap T) = \text{card} B \)

by \( \text{rule dim-unique [OF B indB refl]} \)

moreover have \( \text{dim} S = \text{card} C \)

by \( \text{metis \langle C \subseteq S \rangle \langle \text{independent C} \rangle \langle S \subseteq \text{span C} \rangle \text{basis-card-eq-dim} \) \)

moreover have \( \text{dim} T = \text{card} D \)

by \( \text{metis \langle D \subseteq T \rangle \langle \text{independent D} \rangle \langle T \subseteq \text{span D} \rangle \text{basis-card-eq-dim} \) \)

moreover have \( \text{dim} \ ?ST = \text{card}(C \cup D) \)

proof –

have \( \ast: \forall x \ y. [x \in S; y \in T] \implies x + y \in \text{span} \ (C \cup D) \)

by \( \text{meson \langle S \subseteq \text{span C} \rangle \langle T \subseteq \text{span D} \rangle \text{span-add span-mono subsetCE}} \ sup.\cobounded1 sup.\cobounded2 \) \text{sup.\cobounded1 sup.\cobounded2} \)

show \( \text{thesis} \)

by \( \text{auto intro: \ast \ dim-unique [OF CUD - indCUD refl]} \)

qed

ultimately show \( \text{thesis} \)

using \( \langle B = C \cap D \rangle \) \[\text{symmetric} \]

by \( \text{simp add: \langle independent C \rangle \langle independent D \rangle \text{card-Un-Int finiteI-independent} \) \)

qed

lemma \( \text{dependent-biggerset-general:} \)

\( \text{finite S \implies card S > dim S \implies dependent S} \)

using \( \text{independent-bound-general[of S]} \) by \( \text{metis linorder-not-le} \)

lemma \( \text{subset-le-dim:} \)

\( S \subseteq \text{span} T \implies \text{dim} S \leq \text{dim} T \)

by \( \text{metis dim-span dim-subset} \)
**THEORY “Vector-Spaces”**

```plaintext
lemma linear-inj-imp-surj:
  assumes lf: linear scale scale f
    and fi: inj f
  shows surj f
proof –
  interpret lf: linear scale scale f by fact
  where B: B ⊆ UNIV independent B UNIV ⊆ span B card B = dim UNIV
    by blast
  from B(4) have d: dim UNIV = card B
    by simp
  have UNIV ⊆ span (f ` B)
    proof (rule card-ge-dim-independent)
      show independent (f ` B)
        by (simp add: B(2) fi lf.independent-inj-image)
      have card (f ` B) = dim UNIV
        by (metis B(1) card-image d fi inj-on-subset)
      then show dim UNIV ≤ card (f ` B)
        by simp
    qed blast
  then show ?thesis
    unfolding lf.span-image surj_def
    using B(3) by blast
  qed

end

locale finite-dimensional-vector-space-pair-1 =
  vs1: finite-dimensional-vector-space s1 B1 + vs2: vector-space s2
  for vs1 :: 'a::field ⇒ 'b::ab-group-add ⇒ 'b (infixr *a 75)
    and B1 :: 'b set
  and vs2 :: 'a::field ⇒ 'c::ab-group-add ⇒ 'c (infixr *b 75)
begin

sublocale vector-space-pair s1 s2 by unfold-locales

lemma dim-image-eq:
  assumes lf: linear s1 s2 f
    and fi: inj-on f (vs1.span S)
  shows vs2.dim (f ` S) = vs1.dim S
proof –
  interpret lf: linear by fact
  obtain B where B: B ⊆ S vs1.independent B S ⊆ vs1.span B card B = vs1.dim S
    using vs1.basis-exists[of S] by auto
  then have vs1.span S = vs1.span B
```
moreover have \( \text{card} \ (f \setminus B) = \text{card} \ B \)
by auto
moreover have \( (f \setminus B) \subseteq (f \setminus S) \)
using B by auto
ultimately show \(?thesis\)
by (metis B(2) B(4) fi if:dependent-inj-imageD if.span-image vs2.dim-eq-card-independent vs2.dim-span)
qed

lemma \( \text{dim-image-le} \):
assumes \( \text{lf} : \text{linear s1 s2 f} \)
shows \( \text{vs2.dim} \ (f \setminus S) \leq \text{vs1.dim} \ (S) \)
proof
−
from vs1.basis-exists[of S] obtain B where
\( B : B \subseteq S \text{ vs1.independent B S} \subseteq \text{vs1.span B card B = vs1.dim S} \) by blast
from B have \( fB : \text{finite B card B = vs1.dim S} \)
using vs1.independent-bound-general by blast+
have \( \text{vs2.dim} \ (f \setminus S) \leq \text{card} \ (f \setminus B) \)
apply (rule vs2.span-card-ge-dim)
using \( \text{lf B fB} \)
apply (auto simp add: module-hom.span-image module-hom.spans-image subset-image-iff
linear-iff-module-hom)
done
also have \( \ldots \leq \text{vs1.dim S} \)
using card-image-le[of fB(1)] fB by simp
finally show \(?thesis\).
qed

end

locale finite-dimensional-vector-space-pair =
\( \text{vs1} : \text{finite-dimensional-vector-space s1 B1} + \text{vs2} : \text{finite-dimensional-vector-space s2 B2} \)
for s1 :: 'a::field ⇒ 'b::ab-group-add ⇒ 'b (infixr *a 75)
and B1 :: 'b set
and s2 :: 'a::field ⇒ 'c::ab-group-add ⇒ 'c (infixr *b 75)
and B2 :: 'c set
begin

sublocale finite-dimensional-vector-space-pair-1 ..

lemma linear-surjective-imp-injective:
assumes \( \text{lf} : \text{linear s1 s2 f and sf} : \text{surj f and eq : vs2.dim UNIV = vs1.dim UNIV} \)
shows \( \text{inj f} \)
proof −
interpret linear s1 s2 f by fact
have \( \ast \) \( \colon \) \( \text{card} \ (f : B_1) \leq \text{vs2.dim \ UNIV} \)
using \( \text{vs1.finite-Basis vs1.dim-eq-card[of B1 UNIV]} \) sf
by \{ auto simp: \( \text{vs1.span-Basis vs1.independent-Basis eq \ simp del: vs2.dim-UNIV intro!} \) card-image-le \}

have indep-fB: \( \text{vs2.independent (f \ B_1)} \)
using \( \text{vs1.finite-Basis vs1.dim-eq-card[of B1 UNIV]} \) sf *
by \{ intro vs2.card-le-dim-spanning[of \( f : B_1 \) UNIV] \} \( \text{(auto simp: span-image vs1.span-Basis \ simp del: vs2)} \)

have \( \text{vs2.dim \ UNIV} \leq \text{card} \ (f : B_1) \)
unfolding eq sf \[ \text{[symmetric]} \] \( \text{vs2.dim-span-eq-independent \[ \text{[symmetric, OF indep-fB]} \] \text{symmetric, OF \ indep-fB] \ simp del: vs2}} \)

vs2.dim-span
by \{ intro vs2.dim-mono \} \( \text{(auto simp: span-image vs1.span-Basis, \ simp del: vs2}} \)

with \( \ast \) have \( \text{card} \ (f : B_1) = \text{vs2.dim \ UNIV} \) by auto
also have \( \ldots = \text{card} \ B_1 \)
unfolding eq vs1.dim-UNIV ..

finally have inj-on f B1
by \{ subst inj-on-iff-eq-card[of \( f : B_1 \) \text{finite-Basis}] \}

then show \( \text{inj} \ f \)
using \( \text{inj-on-span-iff-independent-image[OF indep-fB]} \) \( \text{vs1.span-Basis auto \ simp del: vs2}} \)

qed

lemma \( \text{linear-injective-imp-surjective:} \)
assumes \( \text{lf: linear s1 s2 f and sf: inj f and eq: vs2.dim \ UNIV = vs1.dim \ UNIV} \)
shows surj f
proof –
interpret \( \text{linear s1 s2 f by fact} \)
have \( \ast \colon \text{False} \) if \( b : b \notin \text{vs2.span (f \ B_1)} \) for \( b \)
proof –
have \( \ast \colon \text{vs2.independent (f \ B_1)} \)
using \( \text{vs1.independent-Basis by \{ intro independent-injective-image inj-on-subset[of \( f : B_1 \)] \}}} \)

sf \}\) auto

have \( \ast \colon \text{vs2.independent (insert b (f \ B_1))} \)
using \( \text{b \ast by \{ rule vs2.independent-insert1 \}} \)

have \( b \notin f \ B_1 \) \( \text{using vs2.span-base[of b \ f \ B1]} \) b by auto
then have \( \text{Suc (card B1) = card (insert b (f \ B1))} \)
using \( \text{sf[THEN inj-on-subset, of B1] by \{ subst card-insert \} \ (auto intro: vs1.finite-Basis simp: \ text{card-image})} \)
also have \( \ldots = \text{vs2.dim (insert b (f \ B1))} \)
using \( \text{vs2.dim-eq-card-independent[of \[ \text{OF \ \ast \]} \} \text{by \ simp}} \)
also have \( \text{vs2.dim (insert b (f \ B1)) \leq vs2.dim B2} \)
by \{ rule vs2.dim-mono \} \( \text{(auto simp: vs2.span-Basis \ simp del: vs2}} \)
also have \( \ldots = \text{card B1} \)

finally show \( \text{False by \ simp} \)
have \( f \cdot \text{UNIV} = f \cdot \text{vs1.span } B1 \) unfolding vs1.span-Basis ..
also have \( \ldots = \text{vs2.span} (f \cdot B1) \) unfolding span-image ..
also have \( \ldots = \text{UNIV} \) using *
finally show \(?thesis\).

qed

lemma linear-injective-isomorphism:
assumes lf: linear s1 s2 f
and fi: inj f
and dims: vs2.dim UNIV = vs1.dim UNIV
shows \( \exists f'. \text{linear } s2 s1 f' \land (\forall x. f' (f x) = x) \land (\forall x. f (f' x) = x) \)
unfolding isomorphism-expand
[ symmetric ]
using linear-injective-imp-surjective[OF lf fi dims]
using fi left-right-inverse-eq lf linear-injective-left-inverse linear-surjective-right-inverse
by blast

lemma linear-surjective-isomorphism:
assumes lf: linear s1 s2 f
and sf: surj f
and dims: vs2.dim UNIV = vs1.dim UNIV
shows \( \exists f'. \text{linear } s2 s1 f' \land (\forall x. f' (f x) = x) \land (\forall x. f (f' x) = x) \)
using linear-surjective-imp-injective[OF lf sf dims] sf
linear-exists-right-inverse-on[OF lf vs1.subspace-UNIV]
linear-exists-left-inverse-on[OF lf vs1.subspace-UNIV]
dims lf linear-injective-isomorphism by auto

lemma basis-to-basis-subspace-isomorphism:
assumes s: vs1.subspace S
and t: vs2.subspace T
and d: vs1.dim S = vs2.dim T
and B: B \subseteq S vs1.independent B S \subseteq vs1.span B card B = vs1.dim S
and C: C \subseteq T vs2.independent C T \subseteq vs2.span C card C = vs2.dim T
shows \( \exists f. \text{linear } s1 s2 f \land f \cdot B = C \land f \cdot S = T \land \text{inj-on } f S \)
proof –
from B have fB: finite B
by (simp add: vs1.finiteI-independent)
from C have fC: finite C
by (simp add: vs2.finiteI-independent)
from finite-basis-to-basis-subspace-isomorphism[OF s t d fB B fC C] show \(?thesis\).

qed

end

context finite-dimensional-vector-space begin

lemma linear-surj-imp-inj:
assumes lf: linear scale scale f and sf: surj f
shows inj f

proof –

interpret finite-dimensional-vector-space-pair scale Basis scale Basis by unfold-locales

let ?U = UNIV :: 'b set

from basis-exists[of ?U] obtain B


  by blast

{ 
  fix x
  assume x: x ∈ span B and fx: f x = 0
  from B(2) have fB: finite B
    using finitel-independent by auto

  have Uspan: UNIV ⊆ span (f ' B)
    by (simp add: B(3) if linear-spanning-surjective-image sf)

  have fBi: independent (f ' B)

  proof (rule card-le-dim-spanning)

    show card (f ' B) ≤ dim ?U
      using card-image-le d fB by fastforce
  qed (use fB Uspan in auto)

  have th0: dim ?U ≤ card (f ' B)
    by (rule span-card-ge-dim) (use Uspan fB in auto)

  moreover have card (f ' B) ≤ card B
    by (rule card-image-le, rule fB)

  ultimately have th1: card B = card (f ' B)

    unfolding d by arith

  have fB: inj-on f B
    by (simp add: eq-card-imp-inj-on fB th1)

  from linear-indep-image-lemma[OF lf fB fBi x] fx

  have x = 0 by blast
}

then show ?thesis

  unfolding linear-inj-iff-eq-0[OF lf] using B(3) by blast

qed

lemma linear-inverse-left:

assumes lf: linear scale scale f

and lf': linear scale scale f'

shows f ◦ f' = id ↔ f' ◦ f = id

proof –

{ 
  fix f f': 'b ⇒ 'b
  assume lf: linear scale scale f linear scale scale f'
  assume f: f ◦ f' = id

  from f have sf: surj f
    by (auto simp add: o-def id-def surj-def) metis

  interpret finite-dimensional-vector-space-pair scale Basis scale Basis by unfold-locales

  from linear-surjective-isomorphism[OF lf(1) sf] lf f

  have f' ◦ f = id

    unfolding fun-eq-iff o-def id-def by metis
}
then show \(?thesis\)
  using \(lf \ l'\) by metis
qed

lemma left-inverse-linear:
  assumes \(lf: \text{linear scale scale } f\)
  and \(gf: g \circ f = \text{id}\)
  shows \(\text{linear scale scale } g\)
proof –
  from \(gf\) have \(fi: \text{inj } f\)
    by (auto simp add: inj-on-def o-def id-def fun-eq-iff) metis
interpret finite-dimensional-vector-space-pair scale Basis scale Basis by unfold-locales
from linear-injective-isomorphism[OF \(lf \ fi\)]
obtain \(h: \ 'b \Rightarrow 'b\) where \(\text{linear scale scale } h\) and \(h: \forall x. h (f x) = x \forall x. f (h x) = x\)
  by blast
have \(h = g\)
  by (metis \(gf\) \(h\) isomorphism-expand left-right-inverse-eq)
with \(\langle \text{linear scale scale } h\rangle\) show \(?thesis\) by blast
qed

lemma inj-linear-imp-inv-linear:
  assumes \(\text{linear scale scale } f \text{ inj } f\)
  shows \(\text{linear scale scale } (\text{inv } f)\)
using \(assms \text{ inj-iff left-inverse-linear by blast}\)

lemma right-inverse-linear:
  assumes \(lf: \text{linear scale scale } f\)
  and \(gf: f \circ g = \text{id}\)
  shows \(\text{linear scale scale } g\)
proof –
  from \(gf\) have \(fi: \text{surj } f\)
    by (auto simp add: surj-def o-def id-def) metis
interpret finite-dimensional-vector-space-pair scale Basis scale Basis by unfold-locales
from linear-surjective-isomorphism[OF \(lf \ fi\)]
obtain \(h:: \ 'b \Rightarrow 'b\) where \(h: \text{linear scale scale } h\) and \(h (f x) = x \forall x. f (h x) = x\)
  by blast
then have \(h = g\)
  by (metis \(gf\) \(h\) isomorphism-expand left-right-inverse-eq)
with \(h(1)\) show \(?thesis\) by blast
qed

end

context finite-dimensional-vector-space-pair begin

lemma subspace-isomorphism:
  assumes \(s: \text{vs1.subspace } S\)
and \( t \): \texttt{vs2.subspace} \( T \)
and \( d \): \texttt{vs1.dim} \( S = \texttt{vs2.dim} \) \( T \)
shows \( \exists f\. \text{linear} \ s1 \ s2 \ f \land f' \ S = T \land \text{inj-on} \ f \ S \)
proof –
from \texttt{vs1.basis-exists[of S]} \texttt{vs1.finiteI-independent}
obtain \( B \) where \( B: B \subseteq S \) \texttt{vs1.independent} \( B S \subseteq \texttt{vs1.span} \) \( B \) \( \subseteq \texttt{vs1.span} \) \( B \) \( \text{card} \ B = \texttt{vs1.dim} \) \( S \) and \( \beta B: \text{finite} \ B \)
\hspace{1cm} \text{by} \texttt{metis}
from \texttt{vs2.basis-exists[of T]} \texttt{vs2.finiteI-independent}
obtain \( C \) where \( C: C \subseteq T \) \texttt{vs2.independent} \( C T \subseteq \texttt{vs2.span} \) \( C \) \( \text{card} \ C = \texttt{vs2.dim} \) \( T \) and \( f\beta C: \text{finite} \ C \)
\hspace{1cm} \text{by} \texttt{metis}
from \texttt{B(4)} \( C(4) \) \texttt{card-le-inj[of B C]} \( d \)
obtain \( f \) where \( f: f' B \subseteq C \) \texttt{inj-on} \( f \) \( B \) \texttt{using} \( \texttt{finite} B \) \( \texttt{finite} C \)
\hspace{1cm} \text{by} \texttt{auto}
from \texttt{linear-independent-extend[OF B(2)]}
obtain \( g \) where \( g: \text{linear} \ s1 \ s2 \ g \ \forall x \in B \). \( g \ x = f \ x \)
\hspace{1cm} \text{by} \texttt{blast}
interpret \( g: \text{linear} \ s1 \ s2 \ g \) \texttt{by fact}
from \texttt{inj-on-iff-eq-card[OF fB, of f]} \( f(2) \) \texttt{have} \( \text{card} \ (f' B) = \text{card} B \)
\hspace{1cm} \text{by} \texttt{simp}
with \texttt{B(4)} \( C(4) \) \texttt{have} \( \text{ceq} \): \( \text{card} \ (f' B) = \text{card} C \)
\hspace{1cm} \text{using} \( d \) \texttt{by simp}
\hspace{1cm} \text{have} \( g' B = f' B \)
\hspace{1cm} \text{using} \( g(2) \) \texttt{by} \( \texttt{auto simp add: image-iff} \)
also \texttt{have} \ldots = \( C \) \texttt{using} \( \text{card-subset-eq[OF fC f(1) ceq]} \).
finally \texttt{have} \( g\beta C: g' B = C \).
\hspace{1cm} \texttt{have} \( g: \text{inj-on} \ g \ B \)
\hspace{1cm} \texttt{using} \( f(2) \) \( g(2) \) \texttt{by} \( \texttt{auto simp add: inj-on-def} \)
\texttt{note} \( g0 = \text{linear-indep-image-lemma[OF g(1) fB, unfolded g\beta C, OF C(2) g]} \)
\hspace{1cm} \{ \)
\hspace{1cm} \texttt{fix} \( x \ y \)
\hspace{1cm} \texttt{assume} \( x: x \in S \) and \( y: y \in S \) and \( g\beta y: g \ x = g \ y \)
\hspace{1cm} \texttt{from} \texttt{B(3)} \( x \ y \) \texttt{have} \( \exists x': x \in \texttt{vs1.span} \) \( B \) \texttt{and} \( y': y \in \texttt{vs1.span} \) \( B \)
\hspace{1cm} \texttt{by} \texttt{blast+}
\hspace{1cm} \texttt{from} \( g\beta y \) \texttt{have} \( \texttt{th0}: g \ (x - y) = 0 \)
\hspace{1cm} \texttt{by} \( \texttt{(simp add: linear-diff g)} \)
\hspace{1cm} \texttt{have} \( \texttt{th1}: x - y \in \texttt{vs1.span} \) \( B \)
\hspace{1cm} \texttt{using} \( x' y' \) \texttt{by} \( \texttt{(metis vs1.span-diff)} \)
\hspace{1cm} \texttt{have} \( x = y \)
\hspace{1cm} \texttt{using} \( \texttt{g0[OF th1 th0]} \) \texttt{by simp}
\hspace{1cm} \}\)
then \texttt{have} \( g\beta S: \text{inj-on} \ g \ S \)
\hspace{1cm} \texttt{unfolding} \( \texttt{inj-on-def} \) \texttt{by} \texttt{blast}
from \texttt{vs1.span-subspace[OF B(1,3) s]} \texttt{have} \( g' S = \texttt{vs2.span} \) \( g' B \)
\hspace{1cm} \texttt{by} \( \texttt{(simp add: module-hom.span-image[OF g(1)[unfolded linear-iff-module-hom]])} \)
also \texttt{have} \ldots = \( \texttt{vs2.span} \) \( C \) \texttt{unfolding} \( g\beta C \).
also \texttt{have} \ldots = \( T \) \texttt{using} \( \texttt{vs2.span-subspace[OF C(1,3) t]} \).
finally \texttt{have} \( g\beta S: g' S = T \).

from g(1) gS giS show ?thesis
  by blast
qed

end

hide-const (open) linear

end

105 Vector Spaces and Algebras over the Reals

theory Real-Vector-Spaces
imports Real Topological-Spaces Vector-Spaces
begin

105.1 Real vector spaces

class scaleR =
  fixes scaleR :: real ⇒ 'a ⇒ 'a (infixr "*" 75)
begin

abbreviation divideR :: 'a ⇒ real ⇒ 'a (infixl "/" 70)
  where x /R r ≡ inverse r *R x

end

class real-vector = scaleR + ab-group-add +
  assumes scaleR-add-right: a *R (x + y) = a *R x + a *R y
  and scaleR-add-left: (a + b) *R x = a *R x + b *R x
  and scaleR-scaleR: a *R b *R x = (a * b) *R x
  and scaleR-one: 1 *R x = x

class real-algebra = real-vector + ring +
  assumes mult-scaleR-left [simp]: a *R x * y = a *R (x * y)
  and mult-scaleR-right [simp]: x * a *R y = a *R (x * y)

class real-algebra-1 = real-algebra + ring-1

class real-div-algebra = real-algebra-1 + division-ring

class real-field = real-div-algebra + field

instantiation real :: real-field
begin

definition real-scaleR-def [simp]: scaleR a x = a * x

instance
by standard (simp-all add: algebra-simps)

end

locale linear = Vector-Spaces.linear scaleR::⇒'a::real-vector scaleR::⇒'b::real-vector
begin

lemmas scaleR = scale

end

global-interpretation real-vector?: vector-space scaleR :: real ⇒ 'a ⇒ 'a :: real-vector
rewrites Vector-Spaces.linear (∗R) (∗R) = linear
and Vector-Spaces.linear (∗) (∗R) = linear
defines dependent-raw-def: dependent = real-vector.dependent
and representation-raw-def: representation = real-vector.representation
and subspace-raw-def: subspace = real-vector.subspace
and span-raw-def: span = real-vector.span
and extend-basis-raw-def: extend-basis = real-vector.extend-basis
and dim-raw-def: dim = real-vector.dim
apply unfold-locales
  apply (rule scaleR-add-right)
  apply (rule scaleR-add-left)
  apply (rule scaleR-scaleR)
  apply (rule scaleR-one)
  apply (force simp: linear-def)
apply (force simp: linear-def real-scaleR-def[abs-def])
done

hide-const (open)— locale constants
real-vector.dependent
real-vector.independent
real-vector.representation
real-vector.subspace
real-vector.span
real-vector.extend-basis
real-vector.dim

abbreviation independent x ≡ ¬ dependent x

global-interpretation real-vector?: vector-space-pair scaleR::⇒'a::real-vector scaleR::⇒'b::real-vector
rewrites Vector-Spaces.linear (∗R) (∗R) = linear
and Vector-Spaces.linear (∗) (∗R) = linear
defines construct-raw-def: construct = real-vector.construct
apply unfold-locales
unfolding linear-def real-scaleR-def
by (rule refl)+
hide-const (open) — locale constants

real-vector.construct

lemma linear-compose: linear f ⇒ linear g ⇒ linear (g ◦ f)
  unfolding linear-def by (rule Vector-Spaces.linear-compose)

Recover original theorem names

lemmas scaleR-left-commute = real-vector.scale-left-commute
lemmas scaleR-zero-left = real-vector.scale-zero-left
lemmas scaleR-minus-left = real-vector.scale-minus-left
lemmas scaleR-sum-left = real-vector.scale-sum-left
lemmas scaleR-zero-right = real-vector.scale-zero-right
lemmas scaleR-minus-right = real-vector.scale-minus-right
lemmas scaleR-diff-right = real-vector.scale-right-diff-distrib
lemmas scaleR-sum-right = real-vector.scale-sum-right
lemmas scaleR-eq-0-iff = real-vector.scale-eq-0-iff
lemmas scaleR-left-imp-eq = real-vector.scale-left-imp-eq
lemmas scaleR-right-imp-eq = real-vector.scale-right-imp-eq
lemmas scaleR-cancel-left = real-vector.scale-cancel-left
lemmas scaleR-cancel-right = real-vector.scale-cancel-right

lemma [field-simps]:
c ≠ 0 ⇒ a = b / R c ←→ c *R a = b
   c ≠ 0 ⇒ b / R c = a ←→ b = c *R a
   c ≠ 0 ⇒ a + b / R c = (c *R a + b) / R c
   c ≠ 0 ⇒ a / R c + b = (a + c *R b) / R c
   c ≠ 0 ⇒ a − b / R c = (c *R a − b) / R c
   c ≠ 0 ⇒ a / R c − b = (a − c *R b) / R c
   c ≠ 0 ⇒ − (a / R c) + b = (− a + c *R b) / R c
   c ≠ 0 ⇒ − (a / R c) − b = (− a − c *R b) / R c
for a b :: 'a :: real-vector
by (auto simp add: scaleR-add-right scaleR-add-left scaleR-diff-right scaleR-diff-left)

Legacy names

lemmas scaleR-left-distrib = scaleR-add-left
lemmas scaleR-right-distrib = scaleR-add-right
lemmas scaleR-left-diff-distrib = scaleR-diff-left
lemmas scaleR-right-diff-distrib = scaleR-diff-right

lemmas linear-injective-0 = linear-inj-iff-eq-0
and linear-injective-on-subspace-0 = linear-inj-on-iff-eq-0
and linear-cmul = linear-scale
and linear-scaleR = linear-scale-self
and subspace-mul = subspace-scale
and span-linear-image = linear-span-image
and span-0 = span-zero
and span-mul = span-scale
and injective-scaleR = injective-scale
lemma scaleR-minus-left [simp]: \( \text{scaleR} (-1) \, x = -x \)
for \( x :: 'a::real-vector \)
using scaleR-minus-left [of 1 x] by simp

lemma scaleR-2:
fixes \( x :: 'a::real-vector \)
shows \( \text{scaleR} \, 2 \, x = x + x \)
unfolding one-add-one [symmetric] scaleR-left-distrib by simp

lemma scaleR-half-double [simp]:
fixes \( a :: 'a::real-vector \)
shows \( (1 / 2) *_R (a + a) = a \)
proof –
  have \( \forall r. \, r *_R (a + a) = (r * 2) *_R a \)
    by (metis scaleR-2 scaleR-scaleR)
  then show ?thesis
    by simp
qed

lemma linear-scale-real:
fixes \( r :: \text{real} \)
shows \( \text{linear} \, f \implies f (r * b) = r * f b \)
using linear-scale by fastforce

interpretation scaleR-left: additive \((\lambda a. \text{scaleR} \, a \, x :: 'a::real-vector)\)
by standard (rule scaleR-left-distrib)

interpretation scaleR-right: additive \((\lambda x. \text{scaleR} \, a \, x :: 'a::real-vector)\)
by standard (rule scaleR-right-distrib)

lemma nonzero-inverse-scaleR-distrib:
\( a \neq 0 \implies x \neq 0 \implies \text{inverse} \,(\text{scaleR} \, a \, x) = \text{scaleR} \,(\text{inverse} \, a) \,(\text{inverse} \, x) \)
for \( x :: 'a::real-div-algebra \)
by (rule inverse-unique) simp

lemma inverse-scaleR-distrib: inverse \((\text{scaleR} \, a \, x) = \text{scaleR} \,(\text{inverse} \, a) \,(\text{inverse} \, x) \)
for \( x :: 'a::\{\text{real-div-algebra, division-ring}\} \)
by (metis inverse-zero nonzero-inverse-scaleR-distrib scale-eq-0-iff)

lemmas sum-constant-scaleR = real-vector.sum-constant-scale—legacy name

named-theorems vector-add-divide-simps to simplify sums of scaled vectors

lemma [vector-add-divide-simps]:
\( v + (b / z) *_R w = (if z = 0 then v else (z *_R v + b *_R w) /_R z) \)
\( a *_R v + (b / z) *_R w = (if z = 0 then a *_R v else ((a * z) *_R v + b *_R w) /_R z) \)
\( (a / z) *_R v + w = (if z = 0 then w else (a *_R v + z *_R w) /_R z) \)
(a / z) *₅ v + b *₅ w = (if z = 0 then b *₅ w else (a *₅ v + (b * z) *₅ w) /₅ z)

v - (b / z) *₅ w = (if z = 0 then v else (z *₅ v - b *₅ w) /₅ z)
a *₅ v - (b / z) *₅ w = (if z = 0 then a *₅ v else ((a * z) *₅ v - b *₅ w) /₅ z)

(a / z) *₅ v - w = (if z = 0 then -w else (a *₅ v - z *₅ w) /₅ z)

(a / z) *₅ v - b *₅ w = (if z = 0 then -b *₅ w else (a *₅ v - (b * z) *₅ w) /₅ z)

for (v :: 'a :: real-vector)
by (simp-all add: divide-inverse-commute scaleR-add-right scaleR-diff-right)

lemma eq-vector-fraction-iff [vector-add-divide-simps]:
fixes x :: 'a :: real-vector
shows (x = (u / v) *₅ a) ↔ (if v = 0 then x = 0 else v *₅ x = u *₅ a)
by auto (metis (no-types) divide-eq-1-iff divide-inverse-commute scaleR-one scaleR-scaleR)

lemma vector-fraction-eq-iff [vector-add-divide-simps]:
fixes x :: 'a :: real-vector
shows ((u / v) *₅ a = x) ↔ (if v = 0 then x = 0 else u *₅ a = v *₅ x)
by (metis eq-vector-fraction-iff)

lemma real-vector-affinity-eq:
fixes x :: 'a :: real-vector
assumes m0: m ≠ 0
shows m *₅ x + c = y ↔ x = inverse m *₅ y - (inverse m *₅ c)

proof
  assume ?lhs
  then have m *₅ x = y - c by (simp add: field-simps)
  then have inverse m *₅ (m *₅ x) = inverse m *₅ (y - c) by simp
  then show x = inverse m *₅ y - (inverse m *₅ c)
    using m0
    by (simp add: scaleR-diff-right)
next
  assume ?rhs
  with m0 show m *₅ x + c = y
    by (simp add: scaleR-diff-right)
qed

lemma real-vector-eq-affinity: m ≠ 0 ⇒ y = m *₅ x + c ↔ inverse m *₅ y
  - (inverse m *₅ c) = x
  for x :: 'a :: real-vector
  using real-vector-affinity-eq[where m = m and x = x and y = y and c = c]
  by metis

lemma scaleR-eq-iff [simp]: b + u *₅ a = a + u *₅ b ↔ a = b ∨ u = 1
  for a :: 'a :: real-vector
proof (cases u = 1)
case True
then show ?thesis by auto

next
case False
have a = b if b + u *R a = a + u *R b
proof -
  from that have (u - 1) *R a = (u - 1) *R b
  by (simp add: algebra-simps)
  with False show ?thesis
  by auto
qed
then show ?thesis by auto
qed

lemma scaleR-collapse [simp]: (1 - u) *R a + u *R a = a
for a :: 'a::real-vector
by (simp add: algebra-simps)

105.2 Embedding of the Reals into any real-algebra-1: of-real
definition of-real :: real ⇒ 'a::real-algebra-1
  where of-real r = scaleR r 1

lemma scaleR-conv-of-real: scaleR r x = of-real r * x
  by (simp add: of-real-def)

lemma of-real-0 [simp]: of-real 0 = 0
  by (simp add: of-real-def)

lemma of-real-1 [simp]: of-real 1 = 1
  by (simp add: of-real-def)

lemma of-real-add [simp]: of-real (x + y) = of-real x + of-real y
  by (simp add: of-real-def scaleR-left-distrib)

lemma of-real-minus [simp]: of-real (− x) = − of-real x
  by (simp add: of-real-def)

lemma of-real-diff [simp]: of-real (x − y) = of-real x − of-real y
  by (simp add: of-real-def scaleR-left-diff-distrib)

lemma of-real-mult [simp]: of-real (x * y) = of-real x * of-real y
  by (simp add: of-real-def)

lemma of-real-sum[simp]: of-real (sum f s) = (∑ x∈s. of-real (f x))
  by (induct s rule: infinite-finite-induct) auto

lemma of-real-prod[simp]: of-real (prod f s) = (∏ x∈s. of-real (f x))
  by (induct s rule: infinite-finite-induct) auto
lemma nonzero-of-real-inverse:
\[ x \neq 0 \Longrightarrow \text{of-real} \ (\text{inverse} \ x) = \text{inverse} \ (\text{of-real} :: 'a::real-div-algebra) \]
by (simp add: of-real-def nonzero-inverse-scaleR-distrib)

lemma of-real-inverse [simp]:
\[ \text{of-real} \ (\text{inverse} \ x) = \text{inverse} \ (\text{of-real} :: 'a::real-div-algebra,division-ring}) \]
by (simp add: of-real-def inverse-scaleR-distrib)

lemma nonzero-of-real-divide:
\[ y \neq 0 \Longrightarrow \text{of-real} \ (x / y) = (\text{of-real} x / \text{of-real} y :: 'a::real-field) \]
by (simp add: divide-inverse nonzero-of-real-inverse)

lemma of-real-divide [simp]:
\[ \text{of-real} \ (x / y) = (\text{of-real} x / \text{of-real} y :: 'a::real-div-algebra) \]
by (simp add: divide-inverse)

lemma of-real-power [simp]:
\[ \text{of-real} \ (x ^ n) = (\text{of-real} x :: 'a::real-algebra-1}) ^ n \]
by (induct n) simp-all

lemma of-real-eq-iff [simp]: of-real x = of-real y \longleftrightarrow x = y 
by (simp add: of-real-def)

lemma inj-of-real: inj of-real 
by (auto intro: injI)

lemmas of-real-eq-0-iff [simp] = of-real-eq-iff [of 0, simplified]
lemmas of-real-eq-1-iff [simp] = of-real-eq-iff [of 1, simplified]

lemma minus-of-real-eq-of-real-iff [simp]: \(-\text{of-real} x = \text{of-real} y \longleftrightarrow -x = y \)
using of-real-eq-iff[of \(-x y\)] by (simp only: of-real-minus)

lemma of-real-eq-minus-of-real-iff [simp]: of-real x = \(-\text{of-real} y \longleftrightarrow x = -y \)
using of-real-eq-iff[of \(x -y\)] by (simp only: of-real-minus)

lemma of-real-eq-id [simp]: of-real = (id :: real \Rightarrow real) 
by (rule ext) (simp add: of-real-def)

Collapse nested embeddings.

lemma of-real-of-nat-eq [simp]: of-real (of-nat n) = of-nat n 
by (induct n) auto

lemma of-real-of-int-eq [simp]: of-real (of-int z) = of-int z 
by (cases z rule: int-diff-cases) simp

lemma of-real-numeral [simp]: of-real (numeral w) = numeral w 
using of-real-of-int-eq [of numeral w] by simp
lemma of-real-neg-numeral [simp]: of-real (− numeral w) = − numeral w
  using of-real-of-int-eq [of − numeral w] by simp

Every real algebra has characteristic zero.

instance real-algebra-1 < ring-char-0
proof
  from inj-of-real inj-of-nat have inj (of-real o of-nat)
  by (rule inj-compose)
  then show inj (of-nat :: nat ⇒ 'a)
  by (simp add: comp-def)
qed

lemma fraction-scaleR-times [simp]:
  fixes a :: 'a::real-algebra-1
  shows (numeral u / numeral v) *R (numeral w * a) = (numeral u * numeral w) / numeral v *R a
  by (metis (no-types, lifting) of-real-numeral scaleR-conv-of-real scaleR-scaleR times-divide-eq-left)

lemma inverse-scaleR-times [simp]:
  fixes a :: 'a::real-algebra-1
  shows (1 / numeral v) *R (numeral w * a) = (numeral w / numeral v) *R a
  by (metis divide-inverse-commute inverse-eq-divide of-real-numeral scaleR-conv-of-real scaleR-scaleR)

lemma scaleR-times [simp]:
  fixes a :: 'a::real-algebra-1
  shows (numeral u) *R (numeral w * a) = (numeral u * numeral w) *R a
  by (simp add: scaleR-conv-of-real)

instance real-field < field-char-0 ..

105.3 The Set of Real Numbers

definition Reals :: 'a::real-algebra-1 set (R)
  where R = range of-real

lemma Reals-of-real [simp]: of-real r ∈ R
  by (simp add: Reals-def)

lemma Reals-of-int [simp]: of-int z ∈ R
  by (subst of-real-of-int-eq [symmetric], rule Reals-of-real)

lemma Reals-of-nat [simp]: of-nat n ∈ R
  by (subst of-real-of-nat-eq [symmetric], rule Reals-of-real)

lemma Reals-numeral [simp]: numeral w ∈ R
  by (subst of-real-numeral [symmetric], rule Reals-of-real)

lemma Reals-0 [simp]: 0 ∈ R and Reals-1 [simp]: 1 ∈ R
lemma Reals-add [simp]: \( a \in \mathbb{R} \implies b \in \mathbb{R} \implies a + b \in \mathbb{R} \)
  by (metis (no-types, hide-lams) Reals-def Reals-of-real imageE of-real-add)

lemma Reals-minus [simp]: \( a \in \mathbb{R} \implies -a \in \mathbb{R} \)
  by (auto simp: Reals-def)

lemma Reals-minus-iff [simp]: \(-a \in \mathbb{R} \iff a \in \mathbb{R}\)
  apply (auto simp: Reals-def)
  by (metis Reals-add Reals-minus-iff add-uminus-conv-diff)

lemma Reals-mult [simp]: \( a \in \mathbb{R} \implies b \in \mathbb{R} \implies a \cdot b \in \mathbb{R} \)
  by (metis (no-types, lifting) Reals-def Reals-of-real imageE of-real-mult)

lemma nonzero-Reals-inverse: \( a \in \mathbb{R} \implies a \neq 0 \implies \text{inverse} a \in \mathbb{R} \)
  for \( a :: 'a::real-div-algebra \)
  by (metis Reals-def Reals-of-real imageE of-real-inverse)

lemma Reals-inverse [simp]: \( a \in \mathbb{R} \implies \text{inverse} a \in \mathbb{R} \)
  for \( a :: 'a::{real-field,division-ring} \)
  using nonzero-Reals-inverse by fastforce

lemma Reals-inverse-iff [simp]: \( \text{inverse} x \in \mathbb{R} \iff x \in \mathbb{R} \)
  for \( x :: 'a::{real-div-algebra,division-ring} \)
  by (metis Reals-inverse inverse-inverse-eq)

lemma nonzero-Reals-divide: \( a \in \mathbb{R} \implies b \in \mathbb{R} \implies b \neq 0 \implies a / b \in \mathbb{R} \)
  for \( a b :: 'a::{real-field,field} \)
  using nonzero-Reals-divide by fastforce

lemma Reals-power [simp]: \( a \in \mathbb{R} \implies a ^ n \in \mathbb{R} \)
  for \( a :: 'a::real-algebra-1 \)
  by (metis Reals-def Reals-of-real imageE of-real-power)

lemma Reals-cases [cases set: Reals]:
  assumes \( q \in \mathbb{R} \)
  obtains (of-real) \( r \) where \( q = \text{of-real} r \)
  unfolding Reals-def
  proof -
    from \( q \in \mathbb{R} \) have \( q \in \text{range of-real} \) unfolding Reals-def .
  then obtain \( r \) where \( q = \text{of-real} r \) .
then show thesis ..

qed

lemma sum-in-Reals [intro,simp]: \((\forall i. i \in s \implies f i \in \mathbb{R}) \implies \sum f s \in \mathbb{R}\)
proof (induct s rule: infinite-finite-induct)
  case infinite
  then show ?case by (metis Reals-0 sum.infinite)
qed simp-all

lemma prod-in-Reals [intro,simp]: \((\forall i. i \in s \implies f i \in \mathbb{R}) \implies \prod f s \in \mathbb{R}\)
proof (induct s rule: infinite-finite-induct)
  case infinite
  then show ?case by (metis Reals-1 prod.infinite)
qed simp-all

lemma Reals-induct [case-names of-real, induct set: Reals]:
  \(q \in \mathbb{R} \implies (\forall r. P (\text{of-real} r)) \implies P q\)
by (rule Reals-cases) auto

105.4 Ordered real vector spaces

class ordered-real-vector = real-vector + ordered-ab-group-add +
  assumes scaleR-left-mono: \(x \leq y \implies 0 \leq a \implies a * R x \leq a * R y\)
  and scaleR-right-mono: \(a \leq b \implies 0 \leq x \implies a * R x \leq b * R x\)
begin

lemma scaleR-mono:
  \(a \leq b \implies x \leq y \implies 0 \leq b \implies 0 \leq x \implies a * R x \leq b * R y\)
by (meson order-trans scaleR-left-mono scaleR-right-mono)

lemma scaleR-mono':
  \(a \leq b \implies c \leq d \implies 0 \leq a \implies 0 \leq c \implies a * R c \leq b * R d\)
by (rule scaleR-mono) (auto intro: order.trans)

lemma pos-le-divideR-eq [field-simps]:
  \(a \leq b / R c \iff c * R a \leq b \) if \(0 < c\)
proof
  assume ?P
  with scaleR-left-mono that have \(c * R a \leq c * R (b / R c)\)
  by simp
  with that show ?Q
  by (simp add: scaleR-one scaleR-scaleR inverse-eq-divide)
next
  assume ?Q
  with scaleR-left-mono that have \(c * R a / R c \leq b / R c\)
  by simp
  with that show ?P
  by (simp add: scaleR-one scaleR-scaleR inverse-eq-divide)
qed
lemma pos-less-divideR-eq [field-simps]:
\[ a < \frac{b}{R} c \leftrightarrow c *_{R} a < b \text{ if } c > 0 \]
using that pos-le-divideR-eq [of c a b]
by (auto simp add: le-less scaleR-scaleR scaleR-one)

lemma pos-divideR-le-eq [field-simps]:
\[ \frac{b}{R} c \leq a \leftrightarrow b \leq c *_{R} a \text{ if } c > 0 \]
using that pos-le-divideR-eq [of inverse c b a]
by simp

lemma pos-divideR-less-eq [field-simps]:
\[ \frac{b}{R} c < a \leftrightarrow b < c *_{R} a \text{ if } c > 0 \]
using that pos-less-divideR-eq [of inverse c b a]
by simp

lemma pos-minus-divideR-le-eq [field-simps]:
\[ -(\frac{b}{R} c) \leq a \leftrightarrow \frac{b}{R} c \leq a \text{ if } c > 0 \]
using that by (metis pos-divideR-le-eq pos-le-minus-divideR-eq that
inverse-positive-iff-positive le-imp-neg-le minus-minus)

end

lemma scaleR-image-atLeastAtMost: \[ c > 0 \Rightarrow \text{scaleR } c \{x..y\} = \{c *_{R} x..c *_{R} y\} \]
apply (auto intro!: scaleR-left-mono)
apply (rule-tac x = inverse c *_{R} xa in image-eqI)
apply (simp-all add: pos-le-divideR-eq[symmetric] scaleR-scaleR scaleR-one)
done

lemma neg-le-divideR-eq [field-simps]:
\[ a \leq \frac{b}{R} c \leftrightarrow b \leq c *_{R} a \text{ if } c < 0 \]
for a b :: ordered-real-vector
using that pos-le-divideR-eq [of - c a - b] by simp
proof

have *: \(- b = c \cdot a \leftrightarrow b = -(c \cdot a)\)
  by (metis add.inverse_inverse)

from that \(\text{neg-le-divideR-eq \ [af \ c \ a \ b]}\)

show ?thesis by (auto simp add: le-less *)

qed

lemma \(\text{neg-minus-divideR-le-eq \ [field-simps]}\):
  \(- (b / R \ a) \leq a \leftrightarrow c \cdot a \leq - b \) if \(c < 0\)
  for \(a \ b :: \ 'a :: \text{ordered-real-vector}\)

using that \(\text{pos-minus-divideR-le-eq \ [of \ c \ b \ a]}\) by (simp add: le-less)

lemma \(\text{neg-minus-divideR-less-eq \ [field-simps]}\):
  \(- (b / R \ a) < a \leftrightarrow c \cdot a < - b \) if \(c < 0\)
  for \(a \ b :: \ 'a :: \text{ordered-real-vector}\)

using that \(\text{by (simp add: le-less iff)}\)

lemma \(\text{neg-less-minus-divideR-eq \ [field-simps]}\):
  \(- (b / R \ a) < a \leftrightarrow c \cdot a < - b \) if \(c < 0\)
  for \(a \ b :: \ 'a :: \text{ordered-real-vector}\)

proof

have *: \(- b = c \cdot a \leftrightarrow b = -(c \cdot a)\)
  by (metis add.inverse_inverse)

from that \(\text{neg-le-minus-divideR-eq \ [af \ c \ a \ b]}\)

show ?thesis by (auto simp add: le-less *)

qed
by (auto simp add: field-simps)

lemma [field-split-simps]:
  \[ 0 < c \implies a \leq b \land c \iff (if \ c > 0 \then c \ast_R a \leq b \else if \ c < 0 \then b \leq c \ast_R a \else a \leq 0) \]
  \[ 0 < c \implies a < b \land c \iff (if \ c > 0 \then c \ast_R a < b \else if \ c < 0 \then b < c \ast_R a \else a < 0) \]
  \[ 0 < c \implies b \land c \leq a \iff (if \ c > 0 \then b \leq c \ast_R a \else if \ c < 0 \then c \ast_R a \leq b \else a \geq 0) \]
  \[ 0 < c \implies b \land c < a \iff (if \ c > 0 \then b < c \ast_R a \else if \ c < 0 \then c \ast_R a < b \else a > 0) \]

lemma scaleR-nonneg-nonneg: \( 0 \leq a \implies 0 \leq x \implies 0 \leq a \ast_R x \)
  for \( x :: 'a::ordered-real-vector \)
  using scaleR-left-mono [of \( 0 \ x \ a \)] by simp

lemma scaleR-nonneg-nonpos: \( 0 \leq a \implies x \leq 0 \implies a \ast_R x \leq 0 \)
  for \( x :: 'a::ordered-real-vector \)
  using scaleR-left-mono [of \( x \ 0 \ a \)] by simp

lemma scaleR-nonpos-nonneg: \( a \leq 0 \implies 0 \leq x \implies a \ast_R x \leq 0 \)
  for \( x :: 'a::ordered-real-vector \)
  using scaleR-right-mono [of \( a \ 0 \ x \)] by simp

lemma split-scaleR-neg-le: \((0 \leq a \land x \leq 0) \lor (a \leq 0 \land 0 \leq x) \implies a \ast_R x \leq 0 \)
  for \( x :: 'a::ordered-real-vector \)
  by (auto simp: scaleR-nonneg-nonpos scaleR-nonpos-nonneg)

lemma le-add-iff1: \( a \ast_R e + c \leq b \ast_R e + d \iff (a - b) \ast_R e + c \leq d \)
  for \( c d e :: 'a::ordered-real-vector \)
  by (simp add: algebra-simps)

lemma le-add-iff2: \( a \ast_R e + c \leq b \ast_R e + d \iff c \leq (b - a) \ast_R e + d \)
  for \( c d e :: 'a::ordered-real-vector \)
  by (simp add: algebra-simps)

lemma scaleR-left-mono-neg: \( b \leq a \implies c \leq 0 \implies c \ast_R a \leq c \ast_R b \)
  for \( a b :: 'a::ordered-real-vector \)
  by (drule scaleR-left-mono [of \( - \ - \ c \)], simp-all)
lemma scaleR-right-mono-neg: \( b \leq a \implies c \leq 0 \implies a *_R c \leq b *_R c \)
  for \( c :: 'a::ordered-real-vector \)
  by (drule scaleR-right-mono [of - c], simp-all)

lemma scaleR-nonpos-nonpos: \( a \leq 0 \implies b \leq 0 \implies 0 \leq a *_R b \)
  for \( b :: 'a::ordered-real-vector \)
  using scaleR-right-mono-neg [of a 0 b] by simp

lemma split-scaleR-pos-le: \((0 \leq a \land 0 \leq b) \lor (a \leq 0 \land b \leq 0) \implies 0 \leq a *_R b \)
  for \( b :: 'a::ordered-real-vector \)
  by (auto simp: scaleR-nonneg-nonneg scaleR-nonpos-nonpos)

lemma zero-le-scaleR-iff:
  fixes \( b :: 'a::ordered-real-vector \)
  shows \( 0 \leq a *_R b \iff 0 < a \land 0 \leq b \lor a < 0 \land b \leq 0 \lor a = 0 \)
  (is \( \?lhs = \?rhs \))
  proof (cases \( a = 0 \))
    case True
    then show \( \?thesis \) by simp
  next
    case False
    show \( \?thesis \)
    proof
      assume \( \?lhs \)
      from \( a \neq 0 \) consider \( a > 0 \lor a < 0 \) by arith
      then show \( \?rhs \)
        proof cases
          case 1
          with \( \?lhs \) have \( \text{inverse } a *_R 0 \leq \text{inverse } a *_R (a *_R b) \)
            by (intro scaleR-mono) auto
          with 1 show \( \?thesis \)
            by simp
          next
          case 2
          with \( \?lhs \) have \( \text{inverse } a *_R 0 \leq \text{inverse } a *_R (a *_R b) \)
            by (intro scaleR-mono) auto
          with 2 show \( \?thesis \)
            by simp
        qed
    qed
    next
    assume \( \?rhs \)
    then show \( \?lhs \)
      by (auto simp: not-le [of \( a \neq 0 \)] intro!: split-scaleR-pos-le)
  qed
  qed

lemma scaleR-le-0-iff: \( a *_R b \leq 0 \iff 0 < a \land b \leq 0 \lor a < 0 \land b \leq 0 \lor b \land a = 0 \)
for \( b \cdot 'a \cdot \text{ordered-real-vector} \)
by (insert zero-\text{le-scaleR-iff} \([a - b]\)) force

\textbf{lemma} \texttt{scaleR-le-cancel-left}: \( c \cdot\text{R} \cdot a \leq c \cdot\text{R} \cdot b \iff (0 < c \rightarrow a \leq b) \land (c < 0 \rightarrow b \leq a) \)
for \( b :: 'a :: \text{ordered-real-vector} \)
by (auto simp: neq-iff scaleR-left-mono scaleR-left-mono-neg
dest: scaleR-left-mono[where \( a = \text{inverse} \ c \)] scaleR-left-mono-neg[where \( c = \text{inverse} \ c \)])

\textbf{lemma} \texttt{scaleR-le-cancel-left-pos}: \( 0 < c = \Rightarrow c \cdot\text{R} \cdot a \leq c \cdot\text{R} \cdot b \iff a \leq b \)
for \( b :: 'a :: \text{ordered-real-vector} \)
by (auto simp: scaleR-le-cancel-left)

\textbf{lemma} \texttt{scaleR-le-cancel-left-neg}: \( c < 0 = \Rightarrow c \cdot\text{R} \cdot a \leq c \cdot\text{R} \cdot b \iff b \leq a \)
for \( b :: 'a :: \text{ordered-real-vector} \)
by (auto simp: scaleR-le-cancel-left)

\textbf{lemma} \texttt{scaleR-left-le-one-le}: \( 0 \leq x = \Rightarrow a \leq 1 = \Rightarrow a \cdot\text{R} \cdot x \leq x \)
for \( x :: 'a :: \text{ordered-real-vector} \) and \( a :: \text{real} \)
using scaleR-right-mono[of \( a \cdot 1 \cdot x \)] by simp

\textbf{105.5} Real normed vector spaces

\textbf{class} \texttt{dist} =
\textbf{  fixes} \texttt{dist} :: 'a \rightarrow 'a \rightarrow \text{real}

\textbf{class} \texttt{norm} =
\textbf{  fixes} \texttt{norm} :: 'a \rightarrow \text{real}

\textbf{class} \texttt{sgn-div-norm} = \texttt{scaleR} + \texttt{norm} + \texttt{sgn} +
\textbf{  assumes} \texttt{sgn-div-norm}: \texttt{sgn} \texttt{x} = \texttt{x} / \text{R} \texttt{norm} \texttt{x}

\textbf{class} \texttt{dist-norm} = \texttt{dist} + \texttt{norm} + \texttt{minus} +
\textbf{  assumes} \texttt{dist-norm}: \texttt{dist} \texttt{x} \texttt{y} = \texttt{norm} (\texttt{x} - \texttt{y})

\textbf{class} \texttt{uniformity-dist} = \texttt{dist} + \texttt{uniformity} +
\textbf{  assumes} \texttt{uniformity-dist}: \texttt{uniformity} = (\texttt{INF} \texttt{e} \in \{0 < \text{.}\}. \texttt{principal} \{(\texttt{x}, \texttt{y}). \texttt{dist} \texttt{x} \texttt{y} < \texttt{e} \})
begin

\textbf{lemma} \texttt{eventually-uniformity-metric}:
\texttt{eventually} \texttt{P} \texttt{uniformity} \iff (\exists e > 0. \forall \texttt{x} \texttt{y}. \texttt{dist} \texttt{x} \texttt{y} < e \rightarrow \texttt{P} (\texttt{x}, \texttt{y}))

\textbf{unfolding} \texttt{uniformity-dist} by (subst eventually-\texttt{INF-base})
  (auto simp: eventually-\texttt{principal} \texttt{subset-eq} intro: \texttt{bexI[of min -.]} )

end
class real-normed-vector = real-vector + sgn-div-norm + dist-norm + uniformity-dist
+ open-uniformity +
  assumes norm-eq-zero [simp]: norm x = 0 ↔ x = 0
  and norm-triangle-ineq: norm (x + y) ≤ norm x + norm y
  and norm-scaleR [simp]: norm (scaleR a x) = |a| * norm x
begin

lemma norm-ge-zero [simp]: 0 ≤ norm x
proof -
  have 0 = norm (x + -1 *R x)
    using scaleR-add-left[of 1 -1 x] norm-scaleR[of 0 x] by (simp add: scaleR-one)
  also have ... ≤ norm x + norm (-1 *R x) by (rule norm-triangle-ineq)
  finally show ?thesis by simp
qed

end

class real-normed-algebra = real-algebra + real-normed-vector +
  assumes norm-mult-ineq: norm (x * y) ≤ norm x * norm y

class real-normed-algebra-1 = real-algebra-1 + real-normed-algebra +
  assumes norm-one [simp]: norm 1 = 1

lemma (in real-normed-algebra-1) scaleR-power [simp]: (scaleR x y) ^ n = scaleR (x ^ n) (y ^ n)
  by (induct n) (simp-all add: scaleR-one scaleR-scaleR mult-ac)

class real-normed-div-algebra = real-div-algebra + real-normed-vector +
  assumes norm-mult: norm (x * y) = norm x * norm y

class real-normed-field = real-field + real-normed-div-algebra

instance real-normed-div-algebra < real-normed-algebra-1
proof
  show norm (x * y) ≤ norm x * norm y for x y :: 'a
    by (simp add: norm-mult)
next
  have norm (1 * 1::'a) = norm (1::'a) * norm (1::'a)
    by (rule norm-mult)
  then show norm (1::'a) = 1 by simp
qed

case real-normed-vector begin

lemma norm-zero [simp]: norm (0::'a) = 0
  by simp

lemma zero-less-norm-iff [simp]: norm x > 0 ⟷ x ≠ 0
  by (simp add: order-less-le)
lemma norm-not-less-zero [simp]: \( \neg \text{norm } x < 0 \)
by (simp add: linorder-not-less)

lemma norm-le-zero-iff [simp]: \( \text{norm } x \leq 0 \leftrightarrow x = 0 \)
by (simp add: order-le-less)

lemma norm-minus-cancel [simp]: \( \text{norm } (-x) = \text{norm } x \)
proof
  have \(-1 *_R x = - (1 *_R x)\)
    unfolding add-eq-0-iff2[symmetric] scaleR-add-left[symmetric]
    using norm-eq-zero
    by fastforce
  then have \(\text{norm } (-x) = \text{norm } (\text{scaleR } (-1) x)\)
    by (simp only: scaleR-one)
  also have \(\ldots = |-1| * \text{norm } x\)
    by (rule norm-scaleR)
  finally show ?thesis by simp
qed

lemma norm-minus-commute: \( \text{norm } (a - b) = \text{norm } (b - a) \)
proof
  have \(\text{norm } (- (b - a)) = \text{norm } (b - a)\)
    by (rule norm-minus-cancel)
  then show ?thesis by simp
qed

lemma dist-add-cancel [simp]: \( \text{dist } (a + b) (a + c) = \text{dist } b c \)
by (simp add: dist-norm)

lemma dist-add-cancel2 [simp]: \( \text{dist } (b + a) (c + a) = \text{dist } b c \)
by (simp add: dist-norm)

lemma norm-uminus-minus: \( \text{norm } (-x - y) = \text{norm } (x + y) \)
by (subst (2) norm-minus-cancel[symmetric], subst minus-add-distrib) simp

lemma norm-triangle-ineq2: \( \text{norm } a - \text{norm } b \leq \text{norm } (a - b) \)
proof
  have \(\text{norm } (a - b + b) \leq \text{norm } (a - b) + \text{norm } b\)
    by (rule norm-triangle-ineq)
  then show ?thesis by simp
qed

lemma norm-triangle-ineq3: \( |\text{norm } a - \text{norm } b| \leq \text{norm } (a - b) \)
proof
  have \(\text{norm } a - \text{norm } b \leq \text{norm } (a - b)\)
    by (simp add: norm-triangle-ineq2)
  moreover have \(\text{norm } b - \text{norm } a \leq \text{norm } (a - b)\)
    by (metis norm-minus-commute norm-triangle-ineq2)
ultimately show \( ?\text{thesis} \)
  by (simp add: abs-le-iff)
qed

lemma norm-triangle-ineq4: \( \|a - b\| \leq \|a\| + \|b\| \)
proof
  have \( \|a + - b\| \leq \|a\| + \|- b\| \)
    by (rule norm-triangle-ineq)
  then show \( ?\text{thesis} \) by simp
qed

lemma norm-triangle-le-diff: \( \|x\| + \|y\| \leq e \implies \|x - y\| \leq e \)
by (meson norm-triangle-ineq4 order-trans)

lemma norm-diff-ineq: \( \|a\| - \|b\| \leq \|a + b\| \)
proof
  have \( \|a\| - \|- b\| \leq \|a + - b\| \)
    by (rule norm-triangle-ineq2)
  then show \( ?\text{thesis} \) by simp
qed

lemma norm-triangle-sub: \( \|x\| \leq \|y\| + \|x - y\| \)
using norm-triangle-ineq[of y x - y] by (simp add: field-simps)

lemma norm-triangle-le: \( \|x\| + \|y\| \leq e \implies \|x + y\| \leq e \)
by (rule norm-triangle-ineq [THEN order-trans])

lemma norm-triangle-lt: \( \|x\| + \|y\| < e \implies \|x + y\| < e \)
by (rule norm-triangle-ineq [THEN le-less-trans])

lemma norm-add-leD: \( \|a + b\| \leq c \implies \|b\| \leq \|a + c\| \)
by (metis ab-semigroup-add-class.add.commute add-commute diff-le-eq norm-diff-ineq order-trans)

lemma norm-diff-triangle-ineq: \( \| (a + b) - (c + d) \| \leq \| (a - c) + \|b - d\| \)
proof
  have \( \| (a + b) - (c + d) \| = \| (a - c) + (b - d) \| \)
    by (simp add: algebra-simps)
  also have \( \ldots \leq \| (a - c) + \|b - d\| \)
    by (rule norm-triangle-ineq)
  finally show \( ?\text{thesis} \) .
qed

lemma norm-diff-triangle-le: \( \|x - z\| \leq e1 + e2 \)
if \( \|x - y\| \leq e1 \) \( \|y - z\| \leq e2 \)
proof
  have \( \|x - (y + z - y)\| \leq \|x - y\| + \|y - z\| \)
    using norm-diff-triangle-ineq that diff-diff-eq2 by presburger
with that show \( \text{thesis by simp} \)
qed

lemma \( \text{norm-diff-triangle-less} \): \( \text{norm} (x - z) < e_1 + e_2 \)
if \( \text{norm} (x - y) < e_1 \) \( \text{norm} (y - z) < e_2 \)

proof –
  have \( \text{norm} (x - z) \leq \text{norm} (x - y) + \text{norm} (y - z) \)
  by (metis \text{norm-diff-triangle-ineq} \text{add-cancel-cancel-left} \text{diff-cancel-left} \text{diff-diff-eq})
  with that show \( \text{thesis} \) by auto
qed

lemma \( \text{norm-triangle-mono} \):
\( \text{norm} a \leq r \implies \text{norm} b \leq s \implies \text{norm} (a + b) \leq r + s \)

by (metis \text{mono-tags} \text{add-mono-thms-linordered-semiring} \text{1} \text{norm-triangle-ineq} \text{order-trans})

lemma \( \text{norm-sum} \):
\( \text{norm} \left( \sum_{i \in A} f_i \right) \leq \left( \sum_{i \in A} \text{norm} (f_i) \right) \)
for \( f :: 'a \Rightarrow 'b \)
by (induct A rule: \text{infinite-finite-induct}) (auto intro: \text{norm-triangle-mono})

lemma \( \text{sum-norm-le} \):
\( \text{norm} \left( \sum f S \right) \leq \sum g S \)
if \( \forall x. x \in S \implies \text{norm} (f x) \leq g x \)
for \( f :: 'a \Rightarrow 'b \)
by (rule \text{order-trans} [OF \text{norm-sum sum-mono}]) (simp add: that)

lemma \( \text{abs-norm-cancel} \ [\text{simp}]: |\text{norm} a| = \text{norm} a \)
by (rule \text{abs-of-nonneg} [OF \text{norm-ge-zero}])

lemma \( \text{sum-norm-bound} \):
\( \text{norm} \left( \sum f S \right) \leq \text{of-nat} \left( \text{card} S \right) \cdot K \)
if \( \forall x. x \in S \implies \text{norm} (f x) \leq K \)
for \( f :: 'a \Rightarrow 'b \)
using \text{sum-norm-le} [OF that] \text{sum-constant} [\text{symmetric}]
by simp

lemma \( \text{norm-add-less} \): \( \text{norm} x < r \implies \text{norm} y < s \implies \text{norm} (x + y) < r + s \)
by (rule \text{order-le-less-trans} [OF \text{norm-triangle-ineq add-strict-mono}])

end

lemma \( \text{dist-scaleR} \ [\text{simp}]: \text{dist} (x *_R a) (y *_R a) = |x - y| * \text{norm} a \)
for \( a :: 'a :: \text{real-normed-vector} \)
by (metis \text{dist-norm} \text{norm-scaleR} \text{scaleR-left-diff})

lemma \( \text{norm-mult-less} \): \( \text{norm} x < r \implies \text{norm} y < s \implies \text{norm} (x * y) < r * s \)
for \( x y :: 'a :: \text{real-normed-algebra} \)
by (rule \text{order-le-less-trans} [OF \text{norm-mult-ineq}]) (simp add: \text{mult-strict-mono})

lemma \( \text{norm-of-real} \ [\text{simp}]: \text{norm} (\text{of-real} r :: 'a :: \text{real-normed-algebra-1}) = |r| \)
by (simp add: of-real-def)

lemma norm-numeral [simp]: norm (numeral w::'a::real-normed-algebra-1) = numeral w
  by (subst of-real-numeral [symmetric], subst norm-of-real, simp)

lemma norm-neg-numeral [simp]: norm (- numeral w::'a::real-normed-algebra-1) = numeral w
  by (subst of-real-neg-numeral [symmetric], subst norm-of-real, simp)

lemma norm-of-real-add1 [simp]: norm (of-real x+1 ::'a::real-normed-div-algebra) = |x + 1|
  by (metis norm-of-real of-real-1 of-real-add)

lemma norm-of-real-addn [simp]: norm (of-real x+numeral b ::'a::real-normed-div-algebra) = |x + numeral b|
  by (metis norm-of-real of-real-add of-real-numeral)

lemma norm-of-int [simp]: norm (of-int z::'a::real-normed-algebra-1) = of-int z
  by (metis abs-of-nat norm-of-real of-real-of-int-eq)

lemma norm-of-nat [simp]: norm (of-nat n::'a::real-normed-algebra-1) = of-nat n
  by (simp add: of-real-of-nat-eq)

lemma nonzero-norm-inverse: a ≠ 0 ⇒ norm (inverse a) = inverse (norm a)
  for a :: 'a::real-normed-div-algebra
  by (metis inverse-unique norm-mult nonzero-norm-inverse)

lemma norm-inverse: norm (inverse a) = inverse (norm a)
  for a :: 'a::{real-normed-div-algebra,division-ring}
  by (metis inverse-zero nonzero-norm-inverse)

lemma nonzero-norm-divide: b ≠ 0 ⇒ norm (a / b) = norm a / norm b
  for a b :: 'a::real-normed-field
  by (simp add: divide-inverse norm-mult nonzero-norm-inverse)

lemma norm-divide: norm (a / b) = norm a / norm b
  for a b :: 'a::{real-normed-field,field}
  by (simp add: divide-inverse norm-mult norm-inverse)

lemma norm-inverse-le-norm:
  fixes x :: 'a::real-normed-div-algebra
  shows r ≤ norm x ⇒ 0 < r ⇒ norm (inverse x) ≤ inverse r
  by (simp add: le-imp-inverse-le norm-inverse)

lemma norm-power-ineq: norm (x ^ n) ≤ norm x ^ n
  for x :: 'a::real-normed-algebra-1
proof (induct n)
  case 0
show \( \text{norm} (x \cdot 0) \leq \text{norm} x \cdot 0 \) by simp

next

\textbf{case} (Suc \( n \))

\textbf{have} \( \text{norm} (x \cdot x \cdot n) \leq \text{norm} x \cdot \text{norm} (x \cdot n) \)

by (rule norm-mult-ineq)

\textbf{also from} Suc \textbf{have} \ldots \leq \text{norm} x \cdot \text{norm} x \cdot n

using norm-ge-zero by (rule mult-left-mono)

\textbf{finally show} \( \text{norm} (x \cdot \text{Suc} n) \leq \text{norm} x \cdot \text{Suc} n \)

by simp

qed

\textbf{lemma} \textbf{norm-power}: \( \text{norm} (x \cdot n) = \text{norm} x \cdot n \)

\textbf{for} \( x :: 'a::real-normed-div-algebra \)

\textbf{by} (induct \( n \)) (simp-all add: \textbf{norm-mult})

\textbf{lemma} \textbf{power-eq-imp-eq-norm}:

\textbf{fixes} \( w :: 'a::real-normed-div-algebra \)

\textbf{assumes} eq: \( w \cdot n = z \cdot n \) \textbf{and} \( n > 0 \)

\textbf{shows} \( \text{norm} w = \text{norm} z \)

\textbf{proof} –

\textbf{have} \( \text{norm} w \cdot n = \text{norm} z \cdot n \)

by (metis (no-types) eq \textbf{norm-power})

\textbf{then show} \( \text{thesis} \)

using \textbf{assms} by (force intro: \textbf{power-eq-imp-eq-base})

qed

\textbf{lemma} \textbf{power-eq-1-iff}:

\textbf{fixes} \( w :: 'a::real-normed-div-algebra \)

\textbf{shows} \( w \cdot n = 1 \implies \text{norm} w = 1 \lor n = 0 \)

by (metis \textbf{norm-one} \textbf{power-0-left} \textbf{power-eq-0-iff} \textbf{power-eq-imp-eq-norm} \textbf{power-one})

\textbf{lemma} \textbf{norm-mult-numeral1} [simp]: \( \text{norm} (\text{numeral} w \cdot a) = \text{numeral} w \cdot \text{norm} a \)

\textbf{for} \( a b :: 'a::\{real-normed-field,\text{field}\} \)

\textbf{by} (simp add: \textbf{norm-mult})

\textbf{lemma} \textbf{norm-mult-numeral2} [simp]: \( \text{norm} (a \cdot \text{numeral} w) = \text{norm} a \cdot \text{numeral} w \)

\textbf{for} \( a b :: 'a::\{real-normed-field,\text{field}\} \)

\textbf{by} (simp add: \textbf{norm-mult})

\textbf{lemma} \textbf{norm-divide-numeral} [simp]: \( \text{norm} (a / \text{numeral} w) = \text{norm} a / \text{numeral} w \)

\textbf{for} \( a b :: 'a::\{real-normed-field,\text{field}\} \)

\textbf{by} (simp add: \textbf{norm-divide})

\textbf{lemma} \textbf{norm-of-real-diff} [simp]?

\( \text{norm} (\text{of-real} b - \text{of-real} a :: 'a::real-normed-algebra-1) \leq |b - a| \)

\textbf{by} (metis \textbf{norm-of-real} \textbf{of-real-diff} \textbf{order-refl})
Despite a superficial resemblance, \textit{norm-eq-1} is not relevant.

\textbf{lemma} \texttt{square-norm-one}:
\begin{itemize}
  \item fixes $x :: 'a::real-normed-div-algebra$
  \item assumes $x^2 = 1$
  \item shows $\text{norm } x = 1$
\end{itemize}
\textit{by (metis assms \texttt{norm-minus-cancel norm-one power2-eq-1-iff})}

\textbf{lemma} \texttt{norm-less-p1}: $\text{norm } x < \text{norm } (of-real (\text{norm } x + 1) :: 'a)$
\textit{for } $x :: 'a::real-normed-algebra-1$
\textit{proof} –
\begin{itemize}
  \item have $\text{norm } x < \text{norm } (of-real (\text{norm } x + 1) :: 'a)$
    \textit{by (simp add: of-real-def)}
  \item then show ?thesis
    \textit{by simp}
\end{itemize}
\texttt{qed}

\textbf{lemma} \texttt{prod-norm}:
\begin{itemize}
  \item \texttt{prod (λx. norm (f x)) A = norm (prod f A)}
  \item \textit{for } $f :: 'a ⇒ 'b::\{\text{comm-semiring-1, real-normed-div-algebra}\}$
  \item \textit{by (induct A rule: infinite-finite-induct) (auto simp: \texttt{norm-mult})}
\end{itemize}

\textbf{lemma} \texttt{norm-prod-le}:
\begin{itemize}
  \item \texttt{norm (prod f A) ≤ (Π a∈A. norm (f a :: 'a::real-normed-algebra-1, \texttt{comm-monoid-mult}))}
  \item \textit{proof (induct A rule: infinite-finite-induct)}
    \item \texttt{case empty}
    \item then show ?case \texttt{by simp}
  \item \texttt{next}
    \item \texttt{case (insert a A)}
    \item then \texttt{have norm (prod f (insert a A)) ≤ norm (f a) * norm (prod f A)}
      \textit{by (simp add: norm-mult-ineq)}
    \item also \texttt{have norm (prod f A) ≤ (Π a∈A. norm (f a))}
      \textit{by (rule insert)}
    \item finally show ?case
      \textit{by (simp add: insert mult-left-mono)}
  \item \texttt{next}
    \item \texttt{case infinite}
    \item then show ?case \texttt{by simp}
\end{itemize}
\texttt{qed}

\textbf{lemma} \texttt{norm-prod-diff}:
\begin{itemize}
  \item fixes $z w :: 'i ⇒ 'a::\{\text{real-normed-algebra-1, \texttt{comm-monoid-mult}}\}$
  \item \texttt{shows (Π i. i ∈ I ⇒ norm (z i) ≤ 1) ⇒ (Π i. i ∈ I ⇒ norm (w i) ≤ 1) ⇒}
    \texttt{norm ((Π i∈I. z i) - (Π i∈I. w i)) ≤ (Σ i∈I. norm (z i - w i))}
  \item \textit{proof (induction I rule: infinite-finite-induct)}
    \item \texttt{case empty}
    \item then show ?case \texttt{by simp}
  \item \texttt{next}
    \item \texttt{case (insert i 1)}
    \item \texttt{note insert.hyps[simp]}
\end{itemize}
have norm ((∏ i∈insert i I. z i) − (∏ i∈insert i I. w i)) =
  norm ((∏ i∈I. z i) * (z i - w i) + (∏ i∈I. w i)) * w i
(is = norm (?t1 + ?t2))
  by (auto simp: field-simps)
also have ... ≤ norm ?t1 + norm ?t2
  by (rule norm-triangle-inq)
also have norm ?t1 ≤ norm (∏ i∈I. z i) * norm (z i - w i)
  by (rule norm-mult-ineq)
also have norm ?t2 ≤ norm (∏ i∈I. z i) * norm (z i - w i)
  by (rule norm-mult-ineq)
also have (∏ i∈I. norm (z i)) ≤ (∑ i∈I. norm (z i - w i))
  using insert by auto
finally show ?t2
  by (auto simp: ac-simps mult-right-mono mult-left-mono)

next
  case infinite
  then show ?case.
qed

lemma norm-power-diff:
  fixes z w :: 'a::{real-normed-algebra-1, comm-monoid-mult}
  assumes norm z ≤ 1 norm w ≤ 1
  shows norm (z^m - w^m) ≤ m * norm (z - w)
proof
  have norm (z^m - w^m) = norm ((∏ i < m. z) - (∏ i < m. w))
    by simp
  also have ... ≤ (∑ i < m. norm (z - w))
    by (intro norm-prod-diff) (auto simp: assms)
  also have ... = m * norm (z - w)
    by simp
  finally show ?thesis.
qed

105.6 Metric spaces

class metric-space = uniformity-dist + open-uniformity +
  assumes dist-eq-0-iff [simp]: dist x y = 0 <-> x = y
  and dist-triangle2: dist x y ≤ dist x z + dist y z
begin

lemma dist-self [simp]: dist x x = 0
  by simp
lemma zero-le-dist [simp]: \(0 \leq \text{dist } x \ y\)
using dist-triangle2 [of \(x\ y\)] by simp

lemma zero-less-dist-iff: \(0 \prec \text{dist } x \ y \iff x \neq y\)
by (simp add: less-le)

lemma dist-not-less-zero [simp]: \(\neg \text{dist } x \ y \prec 0\)
by (simp add: not-less)

lemma dist-le-zero-iff [simp]: \(\text{dist } x \ y \leq 0 \iff x = y\)
by (simp add: le-less)

lemma dist-commute: \(\text{dist } x \ y = \text{dist } y \ x\)
proof (rule order-antisym)
show \(\text{dist } x \ y \leq \text{dist } y \ x\)
using dist-triangle2 [of \(x\ y\ x\)] by simp
show \(\text{dist } y \ x \leq \text{dist } x \ y\)
using dist-triangle2 [of \(y\ x\ y\)] by simp
qed

lemma dist-commute-lessI: \(\text{dist } y \ x \prec e \implies \text{dist } x \ y \prec e\)
by (simp add: dist-commute)

lemma dist-triangle: \(\text{dist } x \ z \leq \text{dist } x \ y + \text{dist } y \ z\)
using dist-triangle2 [of \(x\ y\ z\)] by (simp add: dist-commute)

lemma dist-triangle3: \(\text{dist } x \ y \leq \text{dist } a \ x + \text{dist } a \ y\)
using dist-triangle2 [of \(x\ y\ a\)] by (simp add: dist-commute)

lemma abs-dist-diff-le: \(|\text{dist } a \ b - \text{dist } b \ c| \leq \text{dist } a \ c\)
using dist-triangle2 [of \(b\ c\ a\)] by simp

lemma dist-pos-lt: \(x \neq y \implies 0 \prec \text{dist } x \ y\)
by (simp add: zero-less-dist-iff)

lemma dist-nz: \(x \neq y \iff 0 \prec \text{dist } x \ y\)
by (simp add: zero-less-dist-iff)

declare dist-nz [symmetric, simp]

lemma dist-triangle-le: \(\text{dist } x \ z + \text{dist } y \ z \leq e \implies \text{dist } x \ y \leq e\)
by (rule order-trans [OF dist-triangle2])

lemma dist-triangle-lt: \(\text{dist } x \ z + \text{dist } y \ z \prec e \implies \text{dist } x \ y \prec e\)
by (rule le-less-trans [OF dist-triangle2])

lemma dist-triangle-less-add: \(\text{dist } x1 \ y \prec e1 \implies \text{dist } x2 \ y \prec e2 \implies \text{dist } x1 \ x2 \prec e1 + e2\)
by (rule dist-triangle-lt [where \(z=y\)]) simp
lemma dist-triangle-half-l: dist x1 y < e / 2 ⇒ dist x2 y < e / 2 ⇒ dist x1 x2 < e
by (rule dist-triangle-lt [where z = y]) simp

lemma dist-triangle-half-r: dist y x1 < e / 2 ⇒ dist y x2 < e / 2 ⇒ dist x1 x2 < e
by (rule dist-triangle-half-l) simp

lemma dist-triangle-third:
assumes dist x1 x2 < e / 3 dist x2 x3 < e / 3 dist x3 x4 < e / 3
shows dist x1 x4 < e
proof
  have dist x1 x3 < e / 3 + e / 3
    by (metis assms (1) assms (2) dist-commute dist-triangle-less-add)
  then have dist x1 x4 < (e / 3 + e / 3) + e / 3
    by (metis assms (3) dist-commute dist-triangle-less-add)
  then show ?thesis
    by simp
qed

subclass uniform-space
proof
  fix E x
  assume eventually E uniformity
  then obtain e where E: 0 < e / \left(\forall x y.\ dist\ x\ y < e \implies E\ (x, y)\right)
    by (auto simp: eventually-uniformity-metric)
  then show E (x, x) \forall f (x, y) in uniformity. E (y, x)
    by (auto simp: eventually-uniformity-metric dist-commute)
  show \exists D. eventually D uniformity \land (\forall x y z. D (x, y) \implies D (y, z) \implies E (x, z))
    using E dist-triangle-half-l\[\text{where } e = e\]
    unfolding eventually-uniformity-metric
    by (intro exI[of - \lambda(x, y).\ dist\ x\ y < e / 2] exI[of - e / 2] conjI)
      (auto simp: dist-commute)
qed

lemma open-dist: open S \iff (\forall x \in S. \exists e > 0, \forall y.\ dist\ x\ y < e \implies y \in S)
by (simp add: dist-commute open-uniformity eventually-uniformity-metric)

lemma open-ball: open \{y.\ dist\ x\ y < d\}
unfolding open-dist
proof (intro ballI)
  fix y
  assume *: y \in \{y.\ dist\ x\ y < d\}
  then show \exists e > 0, \forall z.\ dist\ z\ y < e \implies z \in \{y.\ dist\ x\ y < d\}
    by (auto intro!: exI[of - d - dist x y] simp: field-simps dist-triangle-lt)
qed
subclass first-countable-topology
proof
fix x
show \( \exists A : \text{nat} \Rightarrow 'a \setminus \text{set} \).
(\( \forall i. x \in A i \land \text{open} (A i) \) \( \land \forall S. \text{open} S \land x \in S \rightarrow \)
(\( \exists i. A i \subseteq S \))
proof (safe intro!: exI[of - \( y. \text{dist} x y < \text{inverse} (\text{Suc} n) \)])
fix S
assume \( \text{open} S \land x \in S \)
then obtain e where e: 0 < e and \( \{ y. \text{dist} x y < e \} \subseteq S \)
by (auto simp: open-dist subset-eq dist-commute)
moreover from e obtain i where inverse (Suc i) < e
by (auto dest!: reals-Archimedean)
ultimately show \( \exists i. \{ y. \text{dist} x y < \text{inverse} (\text{Suc} i) \} \subseteq S \)
by blast
qed (auto intro: open-ball)
qed

end

instance metric-space \( \subseteq \) t2-space
proof
fix x y z :: 'a::metric-space
assume xy: x \( \neq \) y
let ?U = \{ y'. dist x y' < dist x y / 2 \}
let ?V = \{ x', dist y x' < dist x y / 2 \}
have \( \ast : \text{d} x z \leq \text{d} x y + \text{d} y z \rightarrow \text{d} y z = \text{d} z y \rightarrow \neg (\text{d} x y * 2 < \text{d} x z \land \text{d} z y * 2 < \text{d} x z) \)
for d :: 'a \( \Rightarrow \) 'a => real and x y z :: 'a
by arith
have open ?U \( \land \) open ?V \( \land \) x \( \in \) ?U \( \land \) y \( \in \) ?V \( \land \) ?U \( \cap \) ?V = {}
using dist-pos-lt[OF xy] *[of dist, OF dist-triangle dist-commute]
using open-ball[of - dist x y / 2] by auto
then show \( \exists \text{U V}. \text{open} \text{U} \land \text{open} \text{V} \land x \in \text{U} \land y \in \text{V} \land \text{U} \cap \text{V} = {} \)
by blast
qed

Every normed vector space is a metric space.
instance real-normed-vector < metric-space
proof
fix x y z :: 'a
show \( \text{dist} x y = 0 \leftrightarrow x = y \)
by (simp add: dist-norm)
show \( \text{dist} x y \leq \text{dist} x z + \text{dist} y z \)
using norm-triangle-ineq4 [of x - z y - z] by (simp add: dist-norm)
qed
105.7 Class instances for real numbers

instantiation real :: real-normed-field
begin

definition dist-real-def: dist x y = |x - y|

definition uniformity-real-def [code def]:
  (uniformity :: (real × real) filter) = (INF e∈{0 <..}, principal {(x, y). dist x y < e})

definition open-real-def [code def]:
  open (U :: real set) \iff (\forall x∈U. eventually (\lambda(x', y). x' = x \rightarrow y ∈ U) uniformity)

definition real-norm-def [simp]: norm r = |r|

instance
  by intro-classes (auto simp: abs-mult open-real-def dist-real-def sgn-real-def uniformity-real-def)

end

declare uniformity-Abort[where 'a=real, code]

lemma dist-of-real [simp]: dist (of-real x :: 'a) (of-real y) = dist x y
  for a :: 'a::real-normed-div-algebra
  by (metis dist-norm norm-of-real of-real-diff real-norm-def)

declare [[code abort: open :: real set ⇒ bool]]

instance real :: linorder-topology
proof
  show (open :: real set ⇒ bool) = generate-topology (range lessThan ∪ range greaterThan)
  proof (rule ext, safe)
    fix S :: real set
    assume open S
    then obtain f where \forall x∈S. 0 < f x \land (\forall y. dist y x < f x \rightarrow y ∈ S)
      unfolding open-dist bchoice-iff ..
    then have *: S = (∪ x∈S. {x - f x <..} ∩ {..< x + f x})
      by (fastforce simp: dist-real-def)
    show generate-topology (range lessThan ∪ range greaterThan) S
      apply (subst *)
      apply (intro generate-topology-Union generate-topology.Int)
      apply (auto intro: generate-topology.Basis)
      done
  next
    fix S :: real set
    assume generate-topology (range lessThan ∪ range greaterThan) S
    moreover have \A a::real. open {..<a}
unfolding open-dist dist-real-def
proof clarify
fix x a :: real
assume x < a
then have \(0 < a - x \land (\forall y. |y - x| < a - x \rightarrow y \in \{..<a\})\) by auto
then show \(\exists e>0. \forall y. |y - x| < e \rightarrow y \in \{..<a\}\) ..
qed
moreover have \(\land a::real. open \{a <..\}\)
unfolding open-dist dist-real-def
proof clarify
fix x a :: real
assume a < x
then have \(0 < x - a \land (\forall y. |y - x| < x - a \rightarrow y \in \{a<..\})\) by auto
then show \(\exists e>0. \forall y. |y - x| < e \rightarrow y \in \{a<..\}\) ..
qed
ultimately show open S
by induct auto
qed

instance real :: linear-continuum-topology ..

lemmas open-real-greaterThan = open-greaterThan[where 'a=real]
lemmas open-real-lessThan = open-lessThan[where 'a=real]
lemmas open-real-greaterThanLessThan = open-greaterThanLessThan[where 'a=real]
lemmas closed-real-atMost = closed-atMost[where 'a=real]
lemmas closed-real-atLeast = closed-atLeast[where 'a=real]
lemmas closed-real-atLeastAtMost = closed-atLeastAtMost[where 'a=real]

instance real :: ordered-real-vector
by standard (auto intro: mult-left-mono mult-right-mono)

105.8 Extra type constraints

Only allow open in class topological-space.

setup (Sign.add-const-constraint
 (const-name open, SOME typ ('a::topological-space set => bool))

Only allow uniformity in class uniform-space.

setup (Sign.add-const-constraint
 (const-name uniformity, SOME typ ('a::uniformity x 'a filter))

Only allow dist in class metric-space.

setup (Sign.add-const-constraint
 (const-name dist, SOME typ ('a::metric-space => 'a => real))

Only allow norm in class real-normed-vector.

setup (Sign.add-const-constraint
 (const-name norm, SOME typ ('a::real-normed-vector => real))
105.9 Sign function

lemma norm-sgn: \[ \text{norm} \ (\text{sgn} \ x) = (\text{if} \ x = 0 \ \text{then} \ 0 \ \text{else} \ 1) \]
for x :: 'a::real-normed-vector
by (simp add: sgn-div-norm)

lemma sgn-zero [simp]: \[ \text{sgn} \ (0 ::'a::real-normed-vector) = 0 \]
by (simp add: sgn-div-norm)

lemma sgn-zero-iff: \[ \text{sgn} \ x = 0 \longleftrightarrow x = 0 \]
for x :: 'a::real-normed-vector
by (simp add: sgn-div-norm)

lemma sgn-minus: \[ \text{sgn} \ (-x) = - \text{sgn} \ x \]
for x :: 'a::real-normed-vector
by (simp add: sgn-div-norm)

lemma sgn-scaleR: \[ \text{sgn} \ (\text{scaleR} \ r \ x) = \text{scaleR} \ (\text{sgn} \ r) \ (\text{sgn} \ x) \]
for x :: 'a::real-normed-vector
by (simp add: sgn-div-norm ac-simps)

lemma sgn-one [simp]: \[ \text{sgn} \ (1 ::'a::real-normed-algebra-1) = 1 \]
by (simp add: sgn-div-norm)

lemma sgn-of-real: \[ \text{sgn} \ (\text{of-real} \ r ::'a::real-normed-algebra-1) = \text{of-real} \ (\text{sgn} \ r) \]
unfolding of-real-def by (simp only: sgn-scaleR sgn-one)

lemma sgn-mult: \[ \text{sgn} \ (x \ast y) = \text{sgn} \ x \ast \text{sgn} \ y \]
for x y :: 'a::real-normed-div-algebra
by (simp add: sgn-div-norm norm-mult)

hide-fact (open) sgn-mult

lemma real-sgn-eq: \[ \text{sgn} \ x = x / |x| \]
for x :: real
by (simp add: sgn-div-norm divide-inverse)

lemma zero-le-sgn-iff [simp]: \[ 0 \leq \text{sgn} \ x \longleftrightarrow 0 \leq x \]
for x :: real
by (cases 0::real x rule: linorder-cases) simp-all

lemma sgn-le-0-iff [simp]: \[ \text{sgn} \ x \leq 0 \longleftrightarrow x \leq 0 \]
for x :: real
by (cases 0::real x rule: linorder-cases) simp-all

lemma norm-conv-dist: \[ \text{norm} \ x = \text{dist} \ x \ 0 \]
unfolding dist-norm by simp

declare norm-conv-dist [symmetric, simp]
lemma dist-0-norm [simp]: dist 0 x = norm x
  for x :: 'a::real-normed-vector
by (simp add: dist-norm)

lemma dist-diff [simp]: dist a (a − b) = norm b dist (a − b)
  a = norm b
by (simp-all add: dist-norm)

lemma dist-of-int: dist (of-int m) (of-int n :: 'a :: real-normed-algebra-1) = of-int |m − n|
proof −
  have dist (of-int m) (of-int n :: 'a) = dist (of-int m :: 'a) (of-int m − (of-int (m − n)))
    by simp
  also have . . . = of-int |m − n| by (subst dist-diff, subst norm-of-int) simp
  finally show thesis .
qed

lemma dist-of-nat:
  dist (of-nat m) (of-nat n :: 'a :: real-normed-algebra-1) = of-int |int m − int n|
by (subst (1 2) of-int-of-nat-eq [symmetric]) (rule dist-of-int)

105.10 Bounded Linear and Bilinear Operators

lemma linearI: linear f
  if ⋀ b1 b2. f (b1 + b2) = f b1 + f b2
  ⋀ r b. f (r *R b) = r *R f b
  using that
by unfold-locales (auto simp: algebra-simps)

lemma linear-iff:
  linear f ⟷ (∀ x y. f (x + y) = f x + f y) ∧ (∀ c x. f (c *R x) = c *R f x)
(is linear f ⟷ ?rhs)
proof
  assume linear f
  then interpret f: linear f .
  show ?rhs by (simp add: f.add f.scale)
next
  assume ?rhs
  then show linear f by (intro linearI) auto
qed

lemmas linear-scaleR-left = linear-scale-left
lemmas linear-imp-scaleR = linear-imp-scale

corollary real-linearD:
  fixes f :: real ⇒ real
  assumes linear f obtains c where f = (∗) c
  by (rule linear-imp-scaleR [OF assms]) (force simp: scaleR-conv-of-real)
THEORY “Real-Vector-Spaces”

lemma linear-times-of-real: linear (λx. a * of-real x)
  by (auto intro!: linearI simp: distrib-left)
    (metis mult-scaleR-right scaleR-conv-of-real)

locale bounded-linear =
  fixes f :: 'a::real-normed-vector ⇒ 'b::real-normed-vector
  assumes bounded: ∃K. ∀x. norm (f x) ≤ norm x * K
begin

lemma pos-bounded: ∃K>0. ∀x. norm (f x) ≤ norm x * K
proof −
  obtain K where K: ∀x. norm (f x) ≤ norm x * K
    using bounded by blast
  show ?thesis
  proof (intro exI impI conjI allI)
    show 0 < max 1 K
      by (rule order-less-le-trans [OF zero-less-one max.cobounded1])
  next
    fix x
    have norm (f x) ≤ norm x * K using K .
    also have .. ≤ norm x * max 1 K
      by (rule mult-left-mono [OF max.cobounded2 norm-ge-zero])
    finally show norm (f x) ≤ norm x * max 1 K .
  qed
qed

lemma nonneg-bounded: ∃K≥0. ∀x. norm (f x) ≤ norm x * K
  using pos-bounded by (auto intro: order-less-imp-le)

lemma linear: linear f
  by (fact local.linear-axioms)
end

lemma bounded-linear-intro:
  assumes \( \forall x y. f(x + y) = f x + f y \)
  and \( \forall r x. f(scaleR r x) = scaleR r (f x) \)
  and \( \forall x. norm (f x) \leq norm x * K \)
  shows bounded-linear f
  by standard (blast intro: assms)+

locale bounded-bilinear =
  fixes prod :: 'a::real-normed-vector ⇒ 'b::real-normed-vector ⇒ 'c::real-normed-vector
    (infixl ** 70)
  assumes add-left: prod (a + a') b = prod a b + prod a' b
    and add-right: prod a (b + b') = prod a b + prod a b'
  and scaleR-left: prod (scaleR r a) b = scaleR r (prod a b)
  and scaleR-right: prod a (scaleR r b) = scaleR r (prod a b)
  and bounded: ∃K. ∀a b. norm (prod a b) ≤ norm a * norm b * K
THEORY “Real-Vector-Spaces”

begin

lemma pos-bounded: \( \exists K > 0. \forall a b. \text{norm} (a ** b) \leq \text{norm} a * \text{norm} b * K \)
proof –
  obtain \( K \) where \( \forall a b. \text{norm} (a ** b) \leq \text{norm} a * \text{norm} b * K \)
  using bounded by blast
  then have \( \text{norm} (a ** b) \leq \text{norm} a * \text{norm} b * (\max 1 K) \) for \( a b \)
  by (rule order.trans) (simp add: mult-left-mono)
  then show \( \exists \)thesis
  by force
qed

lemma nonneg-bounded: \( \exists K \geq 0. \forall a b. \text{norm} (a ** b) \leq \text{norm} a * \text{norm} b * K \)
using pos-bounded by (auto intro: order-less-imp-le)

lemma additive-right: additive (\( \lambda b. \text{prod} a b \))
by (rule additive.intros, rule add-right)

lemma additive-left: additive (\( \lambda a. \text{prod} a b \))
by (rule additive.intros, rule add-left)

lemma zero-left: \( \text{prod} 0 b = 0 \)
by (rule additive.zero [OF additive-left])

lemma zero-right: \( \text{prod} a 0 = 0 \)
by (rule additive.zero [OF additive-right])

lemma minus-left: \( \text{prod} (-a) b = -\text{prod} a b \)
by (rule additive.minus [OF additive-left])

lemma minus-right: \( \text{prod} a (-b) = -\text{prod} a b \)
by (rule additive.minus [OF additive-right])

lemma diff-left: \( \text{prod} (a - a') b = \text{prod} a b - \text{prod} a' b \)
by (rule additive.diff [OF additive-left])

lemma diff-right: \( \text{prod} a (b - b') = \text{prod} a b - \text{prod} a b' \)
by (rule additive.diff [OF additive-right])

lemma sum-left: \( \text{prod} (\sum g S) x = \sum ((\lambda i. \text{prod} (g i) x)) S \)
by (rule additive.sum [OF additive-left])

lemma sum-right: \( \text{prod} x (\sum g S) = \sum ((\lambda i. (\text{prod} x (g i)))) S \)
by (rule additive.sum [OF additive-right])

lemma bounded-linear-left: bounded-linear (\( \lambda a. a ** b \))
proof –
  obtain \( K \) where \( \forall a b. \text{norm} (a ** b) \leq \text{norm} a * \text{norm} b * K \)
using pos-bounded by blast

then show ?thesis

by (rule-tac K=norm b * K in bounded-linear-intro) (auto simp: algebra-simps scaleR-left add-left)

qed

lemma bounded-linear-right: bounded-linear (λb. a ** b)

proof –

obtain K where \( ∃ a b. \text{norm}(a ** b) ≤ \text{norm} a * \text{norm} b * K \)

using pos-bounded by blast

then show ?thesis

by (rule-tac K=norm a * K in bounded-linear-intro) (auto simp: algebra-simps scaleR-right add-right)

qed

lemma prod-diff-prod: \((x ** y - a ** b) = (x - a) ** (y - b) + (x - a) ** b + a ** (y - b)\)

by (simp add: diff-left diff-right)

lemma flip: bounded-bilinear (λx y. y ** x)

apply standard

apply (simp-all add: add-right add-left scaleR-right scaleR-left)

by (metis bounded.mult.commute)

lemma comp1:

assumes bounded-linear g

shows bounded-bilinear (λx. (**)(g x))

proof unfold-locales

interpret g: bounded-linear g by fact

show \( ∃ a a' b. g(a + a') ** b = g a ** b + g a' ** b \)

\( ∃ a b b'. g a ** (b + b') = g a ** b + g a ** b' \)

\( ∃ r a b. g(r * R a) ** b = r * R(g a ** b) \)

\( ∃ r b. g a ** (r * R b) = r * R(g a ** b) \)

by (auto simp: g.add add-left add-right g.scaleR scaleR-right scaleR-left)

from g.nonneg-bounded nonneg-bounded obtain K L

where nn: \( 0 ≤ K 0 ≤ L \)

and K: \( ∃ x. \text{norm}(g x) ≤ \text{norm} x * K \)

and L: \( ∃ a b. \text{norm}(a ** b) ≤ \text{norm} a * \text{norm} b * L \)

by auto

have \( \text{norm}(g a ** b) ≤ \text{norm} a * K * \text{norm} b * L \) for a b


then show \( ∃ K. ∃ x a b. \text{norm}(g a ** b) ≤ \text{norm} a * \text{norm} b * K \)

by (auto intro!: exI[where x=K * L] simp: ac-simps)

qed

lemma comp: bounded-linear f \implies bounded-linear g \implies bounded-bilinear (λx y. f x ** g y)

by (rule bounded-bilinear.flip[OF bounded-bilinear.comp1[OF bounded-bilinear.flip[OF comp1]]])
lemma bounded-linear-ident[simp]: bounded-linear (λx. x)
  by standard (auto intro!: exI[of - 1])

lemma bounded-linear-zero[simp]: bounded-linear (λx. 0)
  by standard (auto intro!: exI[of - 1])

lemma bounded-linear-add:
  assumes bounded-linear f
  and bounded-linear g
  shows bounded-linear (λx. f x + g x)
proof –
  interpret f: bounded-linear f by fact
  interpret g: bounded-linear g by fact
  show ?thesis
proof
  from f.bounded obtain Kf where Kf: norm (f x) ≤ norm x * Kf for x
    by blast
  from g.bounded obtain Kg where Kg: norm (g x) ≤ norm x * Kg for x
    by blast
  show ∃K. ∀x. norm (f x + g x) ≤ norm x * K
    using add-mono[OF Kf Kg] (auto simp: field-simps intro: norm-triangle-ineq order-trans)
  qed (simp-all add: f.add g.add f.scaleR g.scaleR scaleR-right-distrib)
qed

lemma bounded-linear-minus:
  assumes bounded-linear f
  shows bounded-linear (λx. − f x)
proof –
  interpret f: bounded-linear f by fact
  show ?thesis
  by unfold-locales (simp-all add: f.add f.scaleR f.bounded)
qed

lemma bounded-linear-sub: bounded-linear f → bounded-linear g → bounded-linear (λx. f x − g x)
  using bounded-linear-add[of f λx. − g x] bounded-linear-minus[of g]
  by (auto simp: algebra-simps)

lemma bounded-linear-sum:
  fixes f :: 'i ⇒ 'a::real-normed-vector ⇒ 'b::real-normed-vector
  shows (∀i. i ∈ I → bounded-linear (f i)) → bounded-linear (λx. ∑i∈I. f i x)
  by (induct I rule: infinite-finite-induct) (auto intro!: bounded-linear-add)
lemma bounded-linear-compose:
assumes bounded-linear f
and bounded-linear g
shows bounded-linear (λx. f (g x))
proof –
interpret f: bounded-linear f by fact
interpret g: bounded-linear g by fact
show ?thesis
proof unfold-locales
  show f (g (x + y)) = f (g x) + f (g y) for x y
    by (simp only: f.add g.add)
  show f (g (scaleR r x)) = scaleR r (f (g x)) for r x
    by (simp only: f.scaleR g.scaleR)
from f.pos-bounded obtain Kf where f: ⋀x. norm (f x) ≤ norm x * Kf
and Kf: 0 < Kf
  by blast
from g.pos-bounded obtain Kg where g: ⋀x. norm (g x) ≤ norm x * Kg
  by blast
  show ∃K. ⋀x. norm (f (g x)) ≤ norm x * K
    proof (intro exI allI)
      fix x
      have norm (f (g x)) ≤ norm (g x) * Kf
        using f .
      also have ... ≤ (norm x * Kg) * Kf
        using g Kf [THEN order-less-imp-le] by (rule mult-right-mono)
      also have (norm x * Kg) * Kf = norm x * (Kg * Kf)
        by (rule mult.assoc)
      finally show norm (f (g x)) ≤ norm x * (Kg * Kf) .
    qed
  qed
qed

lemma bounded-bilinear-mult: bounded-bilinear ((*) :: 'a::real-normed-algebra)
apply (rule bounded-bilinear.intro)
apply (auto simp: algebra-simps)
apply (rule-tac x=1 in exI)
apply (simp add: norm-mult-ineq)
done

lemma bounded-linear-mult-left: bounded-linear (λx::'a::real-normed-algebra. x * y)
  using bounded-bilinear-mult
  by (rule bounded-bilinear.bounded-linear-left)

lemma bounded-linear-mult-right: bounded-linear (λy::'a::real-normed-algebra. x * y)
  using bounded-bilinear-mult
  by (rule bounded-bilinear.bounded-linear-right)
lemmas bounded-linear-mult-const =
bounded-linear-mult-left [THEN bounded-linear-compose]

lemmas bounded-linear-const-mult =
bounded-linear-mult-right [THEN bounded-linear-compose]

lemma bounded-linear-divide:
bounded-linear (λx. x / y)
for y :: 'a::real-normed-field
unfolding divide-inverse by (rule bounded-linear-mult-left)

lemma bounded-bilinear-scaleR: bounded-bilinear scaleR
apply (rule bounded-bilinear.intros)
apply (auto simp: algebra-simps)
apply (rule_tac x=1 in exI, simp)
done

lemma bounded-linear-scaleR-left: bounded-linear (λr. scaleR r x)
using bounded-bilinear-scaleR
by (rule bounded-bilinear.bounded-linear-left)

lemma bounded-linear-scaleR-right: bounded-linear (λx. scaleR r x)
using bounded-bilinear-scaleR
by (rule bounded-bilinear.bounded-linear-right)

lemmas bounded-linear-scaleR-const =
bounded-linear-scaleR-left[THEN bounded-linear-compose]

lemmas bounded-linear-const-scaleR =
bounded-linear-scaleR-right[THEN bounded-linear-compose]

lemma bounded-linear-of-real:
bounded-linear (λr. of_real r)
unfolding of_real_def by (rule bounded-linear-scaleR-left)

lemma real-bounded-linear: bounded-linear f "⇒ (∃ c::real. f = (λx. x * c))
for f :: real ⇒ real
proof
{  fix x
assume bounded-linear f
then interpret bounded-linear f .
from scaleR[of x 1] have f x = x * f 1 
  by simp
}
then show ?thesis
  by (auto intro: exI[of - f 1] bounded-linear-mult-left)
qed

instance real-normed-algebra-1 ⊆ perfect-space
proof
show ¬ open {x} for x :: 'a
apply (clarsimp simp: open-dist dist-norm)
apply (rule_tac x = x + of-real (e/2) in exI)
apply simp
done
qed

105.11 Filters and Limits on Metric Space

lemma (in metric-space) nhds-metric: nhds x = (INF e∈{0 <..}. principal {y. dist y x < e})
unfolding nhds-def
proof (safe intro!: INF-eq)
fix S
assume open S x ∈ S
then obtain e where {y. dist y x < e} ⊆ S 0 < e
  by (auto simp: open-dist subset-eq)
then show ∃ e∈{0<..}. principal {y. dist y x < e} ≤ principal S
  by auto
qed (auto intro!: exI[of - {y. dist x y < e} for e] open-ball simp: dist-commute)

lemma (in metric-space) tendsto-iff: (f −→ l) F ←→ (∀ e>0. eventually (λx. dist (f x) l < e)) F
unfolding nhds-metric filterlim-INF filterlim-principal by auto

lemma tendstoD: (f −→ l) F =⇒ 0 < e =⇒ eventually (λx. dist (f x) l < e) F
by (auto simp: tendsto-iff)

lemma eventually-nhds-metric: eventually P (nhds a) ←→ (∃ d>0. ∀ x. dist x a < d =⇒ P x)
unfolding nhds-metric
by (subst eventually-INF-base)
  (auto simp: eventually-INF-base)

lemma eventually-at: eventually P (at a within S) ←→ (∃ d>0. ∀ x∈S. x ≠ a ∧ dist x a < d =⇒ P x)
for a :: 'a :: metric-space
by (auto simp: eventually-at-filter eventually-nhds-metric)
lemma frequently-at: frequently P (at a within S) \iff (\forall d>0. \exists x \in S. x \neq a \land dist x a < d \land P x)
for a :: 'a :: metric-space
unfolding frequently-def eventually-at by auto

lemma eventually-at-le: eventually P (at a within S) \iff (\exists d>0. \forall x \in S. x \neq a \land dist x a \leq d \rightarrow P x)
for a :: 'a :: metric-space
unfolding eventually-at-filter eventually-nhds-metric
apply safe
apply (rule-tac x=d/2 in exI, auto)
done

lemma eventually-at-left-real: a > (b :: real) \implies eventually (\lambda x. x \in {b<..<a})
(at-left a)
by (subst eventually-at, rule exI[of - a - b]) (force simp: dist-real-def)

lemma eventually-at-right-real: a < (b :: real) \implies eventually (\lambda x. x \in {a<..<b})
(at-right a)
by (subst eventually-at, rule exI[of - b - a]) (force simp: dist-real-def)

lemma metric-tendsto-imp-tendsto:
fixes a :: 'a :: metric-space
and b :: 'b :: metric-space
assumes f : (f \longrightarrow a) F
and le: eventually (\lambda x. dist (g x) b \leq dist (f x) a) F
shows (g \longrightarrow b) F
proof (rule tendstoD)
fix e :: real
assume 0 < e
with f have eventually (\lambda x. dist (f x) a < e) F by (rule tendstoD)
with le show eventually (\lambda x. dist (g x) b < e) F
using le-less-trans by (rule eventually-elim2)
qed

lemma filterlim-real-sequentially: LIM x sequentially. real x > at-top
apply (clarsimp simp: filterlim-at-top)
apply (rule-tac c=nat [Z + 1] in eventually-sequentiallyI, linarith)
done

lemma filterlim-nat-sequentially: filterlim nat sequentially at-top
proof
have \forall x \in at-top. Z \leq nat x for Z
by (auto intro!: eventually-at-top-linorderI[where c=int Z])
then show \?
using unfolding filterlim-at-top ..
qed

lemma filterlim-floor-sequentially: filterlim floor at-top at-top

THEORY "Real-Vector-Spaces"
proof
  have ∀x in at-top. Z ≤ ⌊x⌋ for Z
    by (auto simp: le-floor-iff intro: eventually-at-toplinorderI[where c=of_int Z])
  then show ?thesis unfolding filterlim-at-top ..
qed

lemma filterlim-sequentially-iff-filterlim-real:
  filterlim f sequentially F ←→ filterlim (λx. real (f x)) at-top F
apply (rule iffI)
subgoal using filterlim-compose filterlim-real-sequentially by blast
subgoal premises prems
proof
  have filterlim (λx. nat (floor (real (f x)))) sequentially F
    by (intro filterlim-compose[OF filterlim-nat-sequentially]
        filterlim-compose[OF filterlim-floor-sequentially] prems)
  then show ?thesis by simp
qed
done

105.11.1 Limits of Sequences

lemma lim-sequentially: X ←→ L ←→ (∀r>0. ∃no. ∀n≥no. dist (X n) L < r)
  for L :: 'a::metric-space
unfolding tendsto-iff eventually-sequentially ..

lemmas LIMSEQ-def = lim-sequentially

lemma LIMSEQ-iff-nz: X ←→ L ←→ (∀r>0. ∃no>0. ∀n≥no. dist (X n) L < r)
  for L :: 'a::metric-space
unfolding lim-sequentially by (metis Suc-leD zero-less-Suc)

lemma metric-LIMSEQ-I: (∀r. 0 < r → ∃no. ∀n≥no. dist (X n) L < r) → X ←→ L
  for L :: 'a::metric-space
by (simp add: lim-sequentially)

lemma metric-LIMSEQ-D: X ←→ L ⇒ 0 < r ⇒ ∃no. ∀n≥no. dist (X n) L < r
  for L :: 'a::metric-space
by (simp add: lim-sequentially)

lemma LIMSEQ-norm-0:
  assumes ∃n. nat. norm (f n) < 1 / real n
  shows f → 0
proof (rule metric-LIMSEQ-I)
THEORY "Real-Vector-Spaces"

fix ε :: real
assume ε > 0
then obtain N :: nat where ε > inverse N N > 0
  by (metis neg0-conv real-arch-inverse)
then have \( \text{norm} (f\ n) < \varepsilon \) if \( n \geq N \) for \( n \)
proof
  have \( \frac{1}{(\text{Suc} n)} \leq \frac{1}{N} \)
  using (\( 0 < N \) inverse-of-nat-le le-SucI)
  also have \( \ldots < \varepsilon \)
  by (metis (no-types) inverse (real N) < \varepsilon inverse-eq-divide)
finally show \( ?\text{thesis} \)
by (meson assms less-eq-real-def not-le order-trans)
qed
then show \( \exists \text{no}. \forall n \geq \text{no}. \text{dist} (f\ n) \ 0 < \varepsilon \)
by auto
qed

105.11.2 Limits of Functions

lemma \text{LIM-def}: \( f \ -a \rightarrow L \iff (\forall r > 0. \exists s > 0. \forall x. x \neq a \land \text{dist} x a < s \rightarrow \text{dist} (f\ x) L < r) \)
for \( a :: 'a::\text{metric-space} \) and \( L :: 'b::\text{metric-space} \)
unfolding tendsto_iff eventually-at by simp

lemma \text{metric-LIM-I}:
\( (\forall r. 0 < r \implies \exists s > 0. \forall x. x \neq a \land \text{dist} x a < s \rightarrow \text{dist} (f\ x) L < r) \implies f \ -a \rightarrow L \)
for \( a :: 'a::\text{metric-space} \) and \( L :: 'b::\text{metric-space} \)
by (simp add: \text{LIM-def})

lemma \text{metric-LIM-D}:
\( f \ -a \rightarrow L \implies 0 < r \implies \exists s > 0. \forall x. x \neq a \land \text{dist} x a < s \rightarrow \text{dist} (f\ x) L < r \)
for \( a :: 'a::\text{metric-space} \) and \( L :: 'b::\text{metric-space} \)
by (simp add: \text{LIM-def})

lemma \text{metric-LIM-imp-LIM}:
fixes l :: 'a::\text{metric-space}
  and m :: 'b::\text{metric-space}
assumes f: \( f \ -a \rightarrow l \)
  and le: \( \forall x. x \neq a \implies \text{dist} (g\ x) m \leq \text{dist} (f\ x) l \)
shows g \(-a\rightarrow m\)
by (rule metric-tendsto-imp-tendsto \{OF f\}) (auto simp: eventually-at-topological le)

lemma \text{metric-LIM-equal2}:
fixes a :: 'a::\text{metric-space}
assumes g \(-a\rightarrow l\ \theta < R\)
  and \( \forall x. x \neq a \implies \text{dist} x a < R \implies f\ x = g\ x \)
shows f \(-a\rightarrow l\)

proof
have \( \bigwedge S. [\text{open } S; l \in S; \forall F \ f \ x \ in \ at \ a. \ g \ x \in S] \implies \forall F \ x \ in \ at \ a. \ f \ x \in S \)
apply (clarsimp simp add: eventually-at)
apply (rule_tac x = \( \min d \ R \) in exI)
apply (auto simp: assms)
done
then show ?thesis
using assms by (simp add: tendsto-def)
qed

lemma metric-LIM-compose2:
fixes \( a :: \cdot \cdot \cdot \text{metric-space} \cdot \cdot \cdot \)
assumes \( f : f \rightarrow a \rightarrow b \)
and \( g : g \rightarrow c \)
and \( \text{inj} : \exists \ d > 0. \forall \ x. \ x \neq a \land \text{dist} \ x \ a < d \implies f \ x \neq b \)
shows \( (\lambda x. \ g (f \ x)) \rightarrow a \rightarrow c \)
using \( \text{inj} \) by (intro tendsto-compose-eventually[OF \( g \ f \ inj \)])

lemma metric-isCont-LIM-compose2:
fixes \( f :: \cdot \cdot \cdot \text{metric-space} \cdot \cdot \cdot \rightarrow l \cdot \cdot \cdot \)
assumes \( f \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \text{unfolded isCont-def}: \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \·
lemma (in metric-space) Cauchy-altdef: Cauchy \( f \) \( \iff \) (\( \forall e > 0. \ \exists M. \ \forall m \geq M. \ \forall n > m. \ \dist (f m) (f n) < e \))
(is \( \text{?lhs} \iff \text{?rhs} \))

proof
  assume \( \text{?rhs} \)
  show \( \text{?lhs} \)
  unfolding Cauchy-def
  proof (intro allI impI)
  fix \( e :: \text{real} \)
  assume \( e : e > 0 \)
  with \( \langle \text{?rhs} \rangle \) obtain \( M \) where \( M : m \geq M \implies n > m \implies \dist (f m) (f n) < e \) for \( m n \)
  by blast
  have \( \dist (f m) (f n) < e \) if \( m \geq M \) \( n \geq M \) for \( m n \)
  using \( M \) [of \( m n \)] \( M \) [of \( n m \)] \( e \) that by (cases \( m n \) rule: linorder-cases) (auto simp: dist-commute)
  then show \( \exists M. \ \forall m \geq M. \ \forall n \geq M. \ \dist (f m) (f n) < e \)
  by blast
  qed

next
  assume \( \text{?lhs} \)
  show \( \text{?rhs} \)
  proof (intro allI impI)
  fix \( e :: \text{real} \)
  assume \( e : e > 0 \)
  with \( \langle \text{Cauchy f} \rangle \) obtain \( M \) where \( \forall m n. \ m \geq M \implies n \geq M \implies \dist (f m) (f n) < e \)
  unfolding Cauchy-def by blast
  then show \( \exists M. \ \forall m \geq M. \ \forall n \geq M. \ \dist (f m) (f n) < e \)
  by (intro exI [of \(- M\)]) force
  qed

qed

lemma (in metric-space) Cauchy-altdef2: Cauchy \( s \) \( \iff \) (\( \forall e > 0. \ \exists N :: \text{nat}. \ \forall n \geq N. \ \dist (s n) (s N) < e \)) (is \( \text{?lhs} = \text{?rhs} \))

proof
  assume Cauchy \( s \)
  then show \( \text{?rhs} \) by (force simp: Cauchy-def)

next
  assume \( \text{?rhs} \)
  { fix \( e :: \text{real} \)
    assume \( e > 0 \)
    with \( \langle \text{?rhs} \rangle \) obtain \( N \) where \( N : \forall n \geq N. \ \dist (s n) (s N) < e/2 \)
    by (erule-tac \( x = e/2 \) in allE) auto
    { fix \( n m \)
      assume \( nm : N \leq m \land N \leq n \)
      then have \( \dist (s m) (s n) < e \) using \( N \)
      using dist-triangle-half-l [of \( s m s N e s n \)]
    }
  }
by blast

then have \( \exists N. \forall m \ n. N \leq m \land N \leq n \rightarrow \text{dist } (s m) (s n) < e \)
by blast

then have \( \text{lhs} \)
unfolding \text{Cauchy-def} by blast
then show \( \text{lhs} \)
by blast

qed

lemma (in metric-space) \text{metric-CauchyI}:
\[
(\forall e. 0 < e \implies \exists M. \forall m \geq M. \forall n \geq M. \text{dist } (X m) (X n) < e) \implies \text{Cauchy X}
\]
by (simp add: \text{Cauchy-def})

lemma (in metric-space) \text{CauchyI'}:
\[
(\forall e. 0 < e \implies \exists M. \forall m \geq M. \forall n > m. \text{dist } (X m) (X n) < e) \implies \text{Cauchy X}
\]
unfolding \text{Cauchy-altdef} by blast

lemma (in metric-space) \text{metric-CauchyD}:
\[
\text{Cauchy X} \implies 0 < e \implies \exists M. \forall m \geq M. \forall n \geq M. \text{dist } (X m) (X n) < e
\]
by (simp add: \text{Cauchy-def})

lemma (in metric-space) \text{metric-Cauchy-iff2}:
\[
\text{Cauchy X} \iff 0 < e \iff \exists M. \forall m \geq M. \forall n \geq M. \text{dist } (X m) (X n) < e
\]
by (simp only: \text{metric-Cauchy-iff2} \text{dist-real-def})

lemma \text{lim-1-over-n [tendsto-intros]}: ((\lambda n. 1 / of-nat n) \longrightarrow (0::'a::real-normed-field))
sequentially

proof (subst \text{lim-sequentially}, intro allI impI exI)
fix e::real and n
assume e::real and e > 0
have inverse e < of-nat (nat [inverse e + 1]) by linarith
also assume n ≥ nat [inverse e + 1]
finally show dist (1 / of-nat n :: 'a) 0 < e
  using e by (simp add: field-split-simps norm-divide)
qed

lemma (in metric-space) \text{complete-def}:
shows \( \text{complete } S = (\forall f. (\forall n. f n \in S) \land \text{Cauchy } f \longrightarrow (\exists l \in S. f \longrightarrow l)) \)
unfolding \text{complete-uniform}
proof safe
fix f :: nat ⇒ 'a
assume \( f : \forall n. \ f \ n \in S \text{ Cauchy } f \)
and \( \ast : \forall F \leq \text{principal } S. \ F \neq \text{bot} \to \text{cauchy-filter } F \to (\exists x \in S. \ F \leq \text{nhds } x) \)
then show \( \exists l \in S. \ f \to l \)
unfolding filterlim-def using \( f \)
by \{ intro \ast \[rule-format]\]
(auto simp: filtermap-sequentially-ne-bot le-principal eventually-filtermap Cauchy-uniform)

next
fix \( F :: \text{'a filter} \)
assume \( F \leq \text{principal } S \ F \neq \text{bot} \text{ cauchy-filter } F \)
assume \( \text{seq} : \forall f. (\forall n. \ f \ n \in S) \land \text{Cauchy } f \to (\exists x \in S. \ f \to x) \)
from \( \langle F \leq \text{principal } S \rangle \langle \text{cauchy-filter } F \rangle \)
have \( FF\text{-le} : F \times F \leq \text{uniformity-on } S \)
by \{ simp add: cauchy-filter-def principal-prod-principal \[symmetric\] prod-filter-mono \}

let \( ?P = \lambda P \ e. \ \text{eventually } P \ F \land (\forall x. \ P x \to x \in S) \land (\forall x y. \ P x \to P y \to \text{dist } x y < e) \)
have \( P : \exists P. \ ?P P \ \text{if } 0 < \epsilon \text{ for } \epsilon :: \text{real} \)
proof –
from that have eventually \( (\lambda (x, y). \ x \in S \land y \in S \land \text{dist } x y < \epsilon) \) \( \text{(uniformity-on } S) \)
by \{ auto simp: eventually-inf-principal eventually-uniformity-metric \}
from filter-leD[of \( FF\text{-le} \) this]
show \( ?\text{thesis} \)
by \{ auto simp: eventually-prod-same \}

qed

have \( \exists P. \ \forall n. \ ?P (P n) (1 / \text{Suc } n) \land P (\text{Suc } n) \leq P n \)
proof \{ rule dependent-nat-choice \}
show \( \exists P. \ ?P P (1 / \text{Suc } 0) \)
using \( P[\text{of } 1] \) by auto

next
fix \( P n \) assume \( ?P P (1/\text{Suc } n) \)
moreover obtain \( Q \text{ where } ?P Q (1 / \text{Suc } (\text{Suc } n)) \)
using \( P[\text{of } 1/\text{Suc } (\text{Suc } n)] \) by auto
ultimately show \( \exists Q. \ ?P Q (1 / \text{Suc } (\text{Suc } n)) \land Q \leq P \)
by \{ intro \ex[f[of - \lambda x. \ P x \land Q x]] \} \( \text{auto simp: eventually-conj-iff} \)

qed

then obtain \( P \) where \( P : \text{eventually } (P n) F \ P n x \to x \in S \)
\( P n x \to P n y \to \text{dist } x y < 1 / \text{Suc } n \ P (\text{Suc } n) \leq P n \)
for \( n x y \)
by metis
have antimonono \( P \)
using \( P(4) \) unfolding decseq-Suc-iff le-fun-def by blast

obtain \( X \) where \( X : P n (X n) \) for \( n \)
using \( P(1)[\text{THEN eventually-happens}[OF \ ?P (\text{bot})]] \) by metis
have \( \text{Cauchy } X \)
unfolding metric-Cauchy-iff2 inverse-eq-divide
proof (intro exI allI impI)
fix \(j \ m \ n::\mathbb{nat}\)
assume \(j \leq m\ \ j \leq n\)
with \(\langle\text{antimono } P\rangle\) \(X\) have \(P\ j\ (X\ m)\ P\ j\ (X\ n)\)
by (auto simp: antimono-def)
then show \(\text{dist} (X\ m)\ (X\ n) < 1 / \text{Suc}\ j\)
by (rule \(P\))
qed
moreover have \(\forall n.\ X\ n \in S\)
using \(P(2)\ X\) by auto
ultimately obtain \(x\) where \(X \longrightarrow x\ \ x \in S\)
using seq by blast
show \(\exists x \in S.\ F \leq \text{nhds}\ x\)
proof (rule bexI)
have \(\text{eventually} (\lambda y.\ \text{dist} y\ x < e)\ F\ \text{if } 0 < e\ \text{for } e::\mathbb{real}\)
proof –
from that have \(\langle\lambda n.\ 1 / \text{Suc}\ n::\mathbb{real}\rangle\longrightarrow 0 \land 0 < e / 2\)
by (subst LIMSEQ-Suc-iff) (auto intro: lim-1-over-n)
then have \(\forall F\ n\ \text{in} \text{sequentially}.\ \text{dist} (X\ n)\ x < e / 2 \land 1 / \text{Suc}\ n < e / 2\)
using \(\langle X \longrightarrow x\rangle\)
unfolding tendsto-iff order-tendsto-iff [where 'a=real] eventually-conj-iff
by blast
then obtain \(n\) where \(\text{dist} x\ (X\ n) < e / 2 \land 1 / \text{Suc}\ n < e / 2\)
by (auto simp: eventually-sequentially dist-commute)
show \?thesis
proof eventually-elim
case (elim \(y\))
then have \(\text{dist} y\ (X\ n) < 1 / \text{Suc}\ n\)
by (intro \(X\ P\))
also have \(\ldots < e / 2\) by fact
finally show \(\text{dist} y\ x < e\)
by (rule dist-triangle-half-l) fact
qed
qed
then show \(F \leq \text{nhds}\ x\)
unfolding nhds-metric le-INF-iff le-principal by auto
qed
fact
qed

apparently unused

lemma (in metric-space) totally-bounded-metric:
totally-bounded \(S \longleftrightarrow (\forall e > 0.\ \exists k.\ \text{finite} \ k \land S \subseteq (\bigcup x \in k.\ \{y.\ \text{dist} x\ y < e\}))\)
unfolding totally-bounded-def eventually-uniformity-metric imp-ex
apply (subst all-comm)
apply (intro arg-cong[where f=All] ext, safe)
subgoal for \(e\)
apply (erule allE[of - \(\lambda (x,\ y).\ \text{dist} x\ y < e\)])
apply auto
done

subgoal for e P k
  apply (intro exI[of - k])
  apply (force simp: subset-eq)
done
done

105.13.1 Cauchy Sequences are Convergent

class complete-space = metric-space +
  assumes Cauchy-convergent: Cauchy X \implies convergent X

lemma Cauchy-convergent-iff: Cauchy X \iff convergent X
  for X :: nat \Rightarrow 'a::complete-space
  by (blast intro: Cauchy-convergent convergent-Cauchy)

To prove that a Cauchy sequence converges, it suffices to show that a subsequence converges.

lemma Cauchy-converges-subseq:
  fixes u :: nat \Rightarrow 'a::metric-space
  assumes Cauchy u strict-mono r (u ◦ r) −→ l
  shows u −→ l
  proof
    have "\*: eventually (λn. dist (u n) l < e) sequentially if e > 0 for e"
    proof
      have "e/2 > 0 using that by auto"
      then obtain N1 where N1: \ A m n. m ≥ N1 \implies n ≥ N1 \implies dist (u m) (u n) < e/2
        using (Cauchy u unfolding Cauchy-def by blast)
      obtain N2 where N2: \ A n. n ≥ N2 \implies dist ((u ◦ r) n) l < e / 2
        using order-tendstoD(2)[OF iffD1[OF tendsto-dist-iff (u ◦ r) −→ l] e/2 > 0]
        unfolding eventually-sequentially by auto
      have "dist (u n) l < e if n ≥ max N1 N2 for n"
      proof
        have "dist (u n) l ≤ dist (u n) ((u ◦ r) n) + dist ((u ◦ r) n) l"
          by (rule dist-triangle)
        also have "... < e/2 + e/2"
          apply (intro add-strict-mono)
          using N1[of n r n] N2[of n] that unfolding comp-def
          by (auto simp: less-imp-le (meson assms(2) less-imp-le order.trans seq-suble)
          finally show ?thesis by simp
        qed
      then show ?thesis unfolding eventually-sequentially by blast
      qed
    have "(λn. dist (u n) l) −→ 0"
apply (rule order-tendstoI)
using * by auto (meson eventually-sequentiallyI less-le-trans zero-le-dist)
then show thesis using tendsto-dist-iff by auto
qed

105.14 The set of real numbers is a complete metric space

Proof that Cauchy sequences converge based on the one from http://pirate.shu.edu/~wachsmut/ira/numseq/proofs/cauconv.html

If sequence $X$ is Cauchy, then its limit is the lub of \{r. $\exists N. \forall n \geq N. r < X_n$\}

lemma increasing-LIMSEQ:
fixes $f$ :: nat $\Rightarrow$ real
assumes inc: $\forall n. f\ n \leq f\ (Suc\ n)$
and bdd: $\forall n. f\ n \leq l$
and en: $\forall e. 0 < e \Longrightarrow \exists n. l \leq f\ n + e$
shows $f \longrightarrow l$
proof (rule increasing-tendsto)
fix $x$
assume $x < l$
with dense[of $0\ l - x$] obtain $e$ where $0 < e < l - x$
by auto
from en[OF $\langle 0 < e \rangle$] obtain $n$ where $l - e \leq f\ n$
by (auto simp: field-simps)
with $(e < l - x) \langle 0 < e \rangle$ have $x < f\ n$
by simp
with incseq-SucI[of $f$, OF inc] show eventually $(\lambda n. x < f\ n)$ sequentially
by (auto simp: eventually-sequentially incseq_def intro: less-le-trans)
qed (use bdd in auto)

lemma real-Cauchy-convergent:
fixes $X$ :: nat $\Rightarrow$ real
assumes $X$: Cauchy $X$
shows convergent $X$
proof

define $S$ :: real set where $S = \{x. \exists M. \forall n \geq M. x < X_n\}$
then have mem-$S$: $\forall x . \forall n \geq M . x < X_n \Longrightarrow x \in S$
by auto

have bound-isUb: $y \leq x$ if $N: \forall n \geq N . X_n < x$ and $y \in S$ for $N$ and $x y ::$ real
proof

from that have $\exists M . \forall n \geq M . y < X_n$
by (simp add: $S$-def)
then obtain $M$ where $\forall n \geq M . y < X_n$ ..
then have $y < X$ (max $M\ N$) by simp
also have $\ldots < x$ using $N$ by simp
finally show thesis by (rule order-less-imp-le)
qed
obtain \( N \) where \( \forall m \geq N. \forall n \geq N. \text{dist} (X m) (X n) < 1 \)
using \( X \langle \text{THEN} \text{metric-CauchyD, OF zero-less-one} \rangle \) by auto
then have \( N: \forall n \geq N. \text{dist} (X n) (X N) < 1 \) by simp
have \([\text{simp}]: S \neq \{\}\)
proof (intro exI ex-in-conv[THEN iffD1])
  from \( N \) have \( \forall n \geq N. X N - 1 < X n \)
  by (simp add: abs-diff-less-iff dist-real-def)
  then show \( \forall s. s \in S \implies s \leq X N + 1 \)
  by (rule bound-isUb)
qed
have \([\text{simp}]: \text{bdd-above} S\)
proof
  from \( N \) have \( \forall n \geq N. X n < X N + 1 \)
  by (simp add: abs-diff-less-iff dist-real-def)
  then show \( \forall s. s \in S \implies s \leq X N + 1 \)
  by (rule bound-isUb)
qed
have \( X \longrightarrow \text{Sup} S \)
proof (rule metric-LIMSEQ-I)
fix \( r :: \text{real} \)
assume \( 0 < r \)
then have \( r: 0 < r/2 \) by simp
obtain \( N \) where \( \forall n \geq N. \forall m \geq N. \text{dist} (X n) (X m) < r/2 \)
using metric-CauchyD \([\text{OF} X \langle \rangle]\) by auto
then have \( \forall n \geq N. \text{dist} (X n) (X N) < r/2 \) by simp
then have \( N: \forall n \geq N. X N - r/2 < X n \wedge X n < X N + r/2 \)
  by (simp only: dist-real-def abs-diff-less-iff)
from \( N \) have \( \forall n \geq N. X N - r/2 < X n \) by blast
then have \( X N - r/2 \in S \) by (rule mem-S)
then have \( 1: X N - r/2 \leq \text{Sup} S \) by (simp add: cSup-upper)

from \( N \) have \( \forall n \geq N. X n < X N + r/2 \) by blast
from bound-isUb[\{OF this\}]
have \( 2: \text{Sup} S \leq X N + r/2 \)
  by (intro cSup-least) simp-all

show \( \exists N. \forall n \geq N. \text{dist} (X n) (\text{Sup} S) < r \)
proof (intro exI allI impI)
fix \( n \)
assume \( n: N \leq n \)
from \( N \) have \( X n < X N + r/2 \) and \( X N - r/2 < X n \)
  by simp-all
then show \( \text{dist} (X n) (\text{Sup} S) < r \) using \( 1 \) \( 2 \)
  by (simp add: abs-diff-less-iff dist-real-def)
qed
qed
then show \( ?\text{thesis} \) by (auto simp: convergent-def)
qed
instance real :: complete-space  
  by intro-classes (rule real-Cauchy-convergent)

class banach = real-normed-vector + complete-space

instance real :: banach ..

lemma tendsto-at-topI-sequentially:
  fixes f :: real  
  assumes mono: \( \forall X. \text{filterlim } X \text{ at-top sequentially} \implies (\lambda n. f (X n)) \longrightarrow y \)
  shows (f \longrightarrow y) at-top
proof –
  obtain A where A: decseq A open \( A n \) nhds y = (INF n. principal (A n)) for n
    by (rule nhds-countable[of y]) (rule that)
  have \( \forall m. \exists k. \forall x \geq k. f x \in A m \)
    proof (rule ccontr)
      assume \( \neg (\forall m. \exists k. \forall x \geq k. f x \in A m) \)
      then obtain m where \( \forall k. \exists x \geq k. f x \notin A m \)
        by auto
      then have \( \exists X. \forall n. (f (X n) \notin A m) \wedge \max n (X n) + 1 \leq X (Suc n) \)
        by (intro dependent-nat-choice) (auto simp del: max.bounded-iff)
      then obtain X where X: \( \forall n. f (X n) \notin A m \wedge \max n (X n) + 1 \leq X (Suc n) \)
        by auto
      have \( 1 \leq n \implies \text{real } n \leq X n \) for n
        using X[of n - 1] by auto
      then have filterlim X at-top sequentially
        by (force intro: filterlim-at-top-mono[OF filterlim-real-sequentially]
            simp: eventually-sequentially)
      from topological-tendstoD[OF *[OF this] A(2, 3), of m] X(1) show False
        by auto
    qed
  then obtain k where k m \leq x \implies f x \in A m for m x
    by metis
  then show ?thesis
    unfolding at-top-def A by (intro filterlim-base[where i=k]) auto
  qed

lemma tendsto-at-topI-sequentially-real:
  fixes f :: real  
  assumes mono: mono f
    and limseq: \( \lambda n. f (\text{real } n) \longrightarrow y \)
  shows (f \longrightarrow y) at-top
proof (rule tendsto)
  fix e :: real
  assume \( 0 < e \)
with \texttt{limseq} obtain \(N::\text{nat}\) where \(N \leq n \implies |f \ (\text{real} \ n) - y| < e\) for \(n\)
by (auto simp: lim-sequentially dist-real-def)

have le: \(fx \leq y\) for \(x::\text{real}\)
proof
  obtain \(n\) where \(x \leq \text{real-of-nat} \ n\)
  using real-arch-simple[of \(x\)] ..
  note monoD[OF mono this]
  also have \(f \ (\text{real-of-nat} \ n) \leq y\)
  by (rule LIMSEQ-le-const[OF limseq]) (auto intro: \(\text{exI}\) [of - \(n\)] monoD[OF mono])
finally show \(?\text{thesis}\).
qed

have \(\text{eventually} \ (\lambda x. \text{real} \ N \leq x) \ \text{at-top}\)
by (rule eventually-ge-at-top)
then show \(\text{eventually} \ (\lambda x. \text{dist} \ (f \ x) \ y < e) \ \text{at-top}\)
proof \text{eventually-elimi}
  case (elim \(x\))
  with \(N[\text{of} \ N]\) le have \(y - f \ (\text{real} \ N) < e\) by auto
  moreover note monoD[OF mono elim]
  ultimately show \(\text{dist} \ (f \ x) \ y < e\)
  using le[of \(x\)] by (auto simp: dist-real-def field-simps)
qed

qed
end

\section{106 Limits on Real Vector Spaces}

theory \textit{Limits}
  imports Real-Vector-Spaces
begin

\subsection{106.1 Filter going to infinity norm}

definition \textit{at-infinity} :: 'a::\text{real-normed-vector filter}
  where \textit{at-infinity} = (\text{INF} \ r. \ \text{principal} \ \{x. \ r \leq \text{norm} \ x\})

lemma eventually-at-infinity: \(\text{eventually} \ P \ \text{at-infinity} \longleftrightarrow (\exists b. \ \forall x. \ b \leq \text{norm} \ x \longrightarrow P \ x)\)
unfolding at-infinity-def
by (subst eventually-INF-base)
  (auto simp: subset-eq eventually-principal intro!: \(\text{exI}\) [of - \(\text{max} \ a \ b\) for \(a \ b\)])

corollary eventually-at-infinity-pos:
  \(\text{eventually} \ P \ \text{at-infinity} \longleftrightarrow (\exists b. \ 0 < b \land (\forall x. \ \text{norm} \ x \geq b \longrightarrow P \ x)\)
unfolding eventually-at-infinity
by (meson le-less-trans norm-ge-zero not-le zero-less-one)

lemma at-infinity-eq-at-top-bot: (at-infinity :: real filter) = sup at-top at-bot
proof –
have 1: \[ \forall n \geq u \cdot A n; \forall n \leq v . A n \] 
\[ \Rightarrow \exists b . \forall x . b \leq |x| \rightarrow A x \] for \( A \) and \( u \) \& \( v \) :: real
by (rule-tac \( x = \max ( - v ) \) \( u \) \in \( \text{exI} \) (auto simp: abs-real-def)

have 2: \( \forall x . u \leq |x| \rightarrow A x \Rightarrow \exists N . \forall n \geq N . A n \) for \( A \) and \( u \) :: real
by (meson abs-less-iff le-cases less-le-not-le)

have 3: \( \forall x . u \leq |x| \rightarrow A x \Rightarrow \exists N . \forall n \leq N . A n \) for \( A \) and \( u \) :: real
by (metis (full-types) abs-ge-self abs-minus-cancel le-minus-iff order-trans)

show ?thesis
by (auto simp: filter-eq-iff eventually-sup eventually-at-infinity intro: 1 2 3)

qed

lemma at-top-le-at-infinity: at-top \( \leq \) (at-infinity :: real filter)
unfolding at-infinity-eq-at-top-bot by simp

lemma at-bot-le-at-infinity: at-bot \( \leq \) (at-infinity :: real filter)
unfolding at-infinity-eq-at-top-bot by simp

lemma filterlim-at-top-imp-at-infinity: filterlim \( f \) at-top \( F \) \( \Rightarrow \) filterlim \( f \) at-infinity \( F \)
for \( f \) :: - \( \Rightarrow \) real
by (rule filterlim-mono[OF - at-top-le-at-infinity order-refl])

lemma filterlim-real-at-infinity-sequentially: filterlim real at-infinity sequentially
by (simp add: filterlim-at-top-imp-at-infinity filterlim-real-sequentially)

106.1.1 Boundedness

definition Bfun :: ('a :: metric-space) \( \Rightarrow \) 'a filter \( \Rightarrow \) bool
where Bfun-metric-def: Bfun \( f \) \( F \) = \( \exists y . \exists K > 0 . \) eventually \( (\lambda x . \operatorname{dist} (f x) y \leq K) \) \( F \)

abbreviation Bseq :: (nat \( \Rightarrow \) 'a :: metric-space) \( \Rightarrow \) bool
where Bseq X \( \equiv \) Bfun X sequentially

lemma Bseq-conv-Bfun: Bseq X \( \leftrightarrow \) Bfun X sequentially ..

lemma Bseq-ignore-initial-segment: Bseq X \( \Rightarrow \) Bseq \( (\lambda n . (n + k)) \)
unfolding Bfun-metric-def by (subst eventually-sequentially-seg)

lemma Bseq-offset: Bseq \( (\lambda n . (n + k)) \) \( \Rightarrow \) Bseq X
unfolding Bfun-metric-def by (subst (asm) eventually-sequentially-seg)

lemma Bfun-def: Bfun \( f \) \( F \) \( \leftrightarrow \) \( (\exists K > 0 . \) eventually \( (\lambda x . \operatorname{norm} (f x) \leq K) \) \( F \)\)
unfolding Bfun-metric-def norm-conv-dist

proof safe
  fix y K
  assume K: 0 < K and *: eventually (λx. dist (f x) y ≤ K) F
  moreover have eventually (λx. dist (f x) 0 ≤ dist (f x) y + dist 0 y) F
    by (intro always-eventually) (metis dist-commute dist-triangle)
  with * have eventually (λx. dist (f x) 0 ≤ K + dist 0 y) F
    by eventually-elim auto
  with ⟨0 < K⟩ have eventually (λx. dist (f x) 0 ≤ K + dist 0 y) F
    by (intro exI[of - K + dist 0 y]) add-pos-nonneg conjI zero-le-dist auto
qeda
assume \( 0 < K \forall n \geq N \cdot \|X_n\| \leq K \)
then show \( \exists K > 0, \forall n \cdot \|X_n\| \leq K \)
  by (intro \textit{exI \{of - max (Max (norm \cdot X \cdot \{..N\}) K)\} max.strict-coboundedI2})
  (auto intro! \{imageI not-less \{where \textit{a=nat}, THEN iffD1 \} Max-ge simp: le-max-iff-disj)
qed auto

lemma \textit{BseqE}: \textbf{Bseq} \( X \implies (\forall K \cdot 0 < K \implies \forall n \cdot \|X_n\| \leq K \implies Q) \implies Q \)
  unfolding \textbf{Bseq-def} by auto

lemma \textit{BseqD}: \textbf{Bseq} \( X \implies \exists K \cdot 0 < K \land (\forall n \cdot \|X_n\| \leq K) \)
  by (simp add: \textbf{Bseq-def})

lemma \textit{BseqI}: \( 0 < K \implies \forall n \cdot \|X_n\| \leq K \implies \textbf{Bseq} X \)
  by (auto simp: \textbf{Bseq-def})

lemma \textit{Bseq-bdd-above}: \textbf{Bseq} \( X \implies \text{bdd-above} (\text{range} X) \)
  for \( X :: \text{nat} \Rightarrow \text{real} \)
proof (elim \textit{BseqE}, intro \textit{bdd-aboveI2})
  fix \( K n \)
  assume \( 0 < K \forall n \cdot \|X_n\| \leq K \)
  then show \( X n \leq K \)
    by (auto elim!: allE[of - n])
qed

lemma \textit{Bseq-bdd-above}': \textbf{Bseq} \( X \implies \text{bdd-above} (\text{range} (\lambda n. \|X_n\|)) \)
  for \( X :: \text{nat} \Rightarrow 'a :: \text{real-normed-vector} \)
proof (elim \textit{BseqE}, intro \textit{bdd-aboveI2})
  fix \( K n \)
  assume \( 0 < K \forall n \cdot \|X_n\| \leq K \)
  then show \( \|X_n\| \leq K \)
    by (auto elim!: allE[of - n])
qed

lemma \textit{Bseq-bdd-below}: \textbf{Bseq} \( X \implies \text{bdd-below} (\text{range} X) \)
  for \( X :: \text{nat} \Rightarrow \text{real} \)
proof (elim \textit{BseqE}, intro \textit{bdd-belowI2})
  fix \( K n \)
  assume \( 0 < K \forall n \cdot \|X_n\| \leq K \)
  then show \( -K \leq X n \)
    by (auto elim!: allE[of - n])
qed

lemma \textit{Bseq-eventually-mono}:
  assumes \( \text{eventually} (\lambda n. \|X_n\| \leq \|Y_n\|) \text{ sequentially} \textbf{Bseq} \ Y \)
  shows \( \textbf{Bseq} \ X \)
proof --
  from assms(2) obtain \( K \) where \( 0 < K \) and \( \text{eventually} (\lambda n. \|X_n\| \leq K) \)
sequentially
  unfolding Bfun-def by fast
with assms(1) have eventually \( (\lambda n. \text{norm } (f n) \leq K) \) sequentially
  by (fast elim: eventually-elim2 order-trans)
with \( \emptyset < K \) show \( \text{Bseq } f \)
  unfolding Bfun-def by fast
qed

lemma lemma-NBseq-def: \( (\exists K > 0. \forall n. \text{norm } (X n) \leq K) \iff (\exists N. \forall n. \text{norm } (X n) \leq \text{real}(Suc N)) \)
proof safe
  fix \( K :: \text{real} \)
  from reals-Archimedean2 obtain \( n :: \text{nat} \) where \( K < \text{real } n \)
  then have \( K \leq \text{real } (Suc n) \) by auto
  moreover assume \( \forall m. \text{norm } (X m) \leq K \)
  ultimately have \( \forall m. \text{norm } (X m) \leq \text{real } (Suc n) \)
  by (blast intro: order-trans)
  then show \( \exists N. \forall n. \text{norm } (X n) \leq \text{real } (Suc N) \)
next
  show \( \forall N. \forall n. \text{norm } (X n) \leq \text{real } (Suc N) \implies \exists K > 0. \forall n. \text{norm } (X n) \leq K \)
    using of-nat-0-less-iff by blast
qed

Alternative definition for \( \text{Bseq} \).

lemma Bseq-iff: \( \text{Bseq } X \iff (\exists N. \forall n. \text{norm } (X n) \leq \text{real}(Suc N)) \)
by (simp add: Bseq-def lemma-NBseq-def)

lemma lemma-NBseq-def2: \( (\exists K > 0. \forall n. \text{norm } (X n) \leq K) = (\exists N. \forall n. \text{norm } (X n) < \text{real}(Suc N)) \)
proof –
  have \(*\): \( \forall N. \forall n. \text{norm } (X n) \leq 1 + \text{real } N \implies \exists N. \forall n. \text{norm } (X n) < 1 + \text{real } N \)
    by (metis add.commute le-less-trans less-add-one of-nat-Suc)
  then show ?thesis
  unfolding lemma-NBseq-def
  by (metis less-less-not-le not-less-iff-gr-or-eq of-nat-Suc)
qed

Yet another definition for \( \text{Bseq} \).

lemma Bseq-iff1a: \( \text{Bseq } X \iff (\exists N. \forall n. \text{norm } (X n) < \text{real } (Suc N)) \)
by (simp add: Bseq-def lemma-NBseq-def2)

106.1.3 A Few More Equivalence Theorems for Boundedness

Alternative formulation for boundedness.

lemma Bseq-iff2: \( \text{Bseq } X \iff (\exists k > 0. \exists x. \forall n. \text{norm } (X n + - x) \leq k) \)
by (metis BseqE BseqI′ add.commute add-cancel-right-left add-uminus-conv-diff norm-add-leD)
Alternative formulation for boundedness.

**Lemma Bseq-iff3**: \( \text{Bseq } X \Longleftrightarrow (\exists k > 0. \exists N. \forall n. \text{norm} (X n + X N) \leq k) \)

(is \( ?P \Longleftrightarrow ?Q \))

**Proof**

Assume \( ?P \)

Then obtain \( K \) where:* 0 < \( K \) and **: \( \forall n. \text{norm} (X n) \leq K \)

by (auto simp: Bseq-def)

From * have 0 < \( K + \text{norm} (X 0) \) by (rule order-less-le-trans) simp

From ** have \( \forall n. \text{norm} (X n - X 0) \leq K + \text{norm} (X 0) \)

by (auto intro: order-trans norm-triangle-ineq4)

Then have \( \forall n. \text{norm} (X n + X 0) \leq K + \text{norm} (X 0) \)

by simp

With (0 < \( K + \text{norm} (X 0) \)): show \( ?Q \) by blast

Next

Assume \( ?Q \)

Then show \( ?P \) by (auto simp: Bseq-iff2)

qed

### 106.1.4 Upper Bounds and Lubs of Bounded Sequences

**Lemma Bseq-minus-iff**: \( \text{Bseq } (\lambda n. -(X n)) \text{::} 'a::real-normed-vector) \iff \text{Bseq } X \)

by (simp add: Bseq-def)

**Lemma Bseq-add**:

fixes \( f \) :: nat \Rightarrow 'a::real-normed-vector

assumes \( \text{Bseq } f \)

shows \( \text{Bseq } (\lambda x. f x + c) \)

**Proof**

From assms obtain \( K \) where \( K: \forall x. \text{norm} (f x) \leq K \)

Unfolding Bseq-def by blast

\[
\begin{align*}
\text{fix } x :: \\
\text{have } \text{norm} (f x + c) \leq \text{norm} (f x) + \text{norm} c \text{ by (rule norm-triangle-ineq)} \\
\text{also have } \text{norm} (f x) \leq K \text{ by (rule } K) \\
\text{finally have } \text{norm} (f x + c) \leq K + \text{norm } c \text{ by simp} \\
\end{align*}
\]

Then show \( ?\text{thesis} \) by (rule BseqI)

qed

**Lemma Bseq-add-iff**: \( \text{Bseq } (\lambda x. f x + c) \iff \text{Bseq } f \)

for \( f :: \text{nat } \Rightarrow 'a::real-normed-vector \)

using Bseq-add[of c] Bseq-add[of \( \lambda x. f x + c - c \)] by auto

**Lemma Bseq-mult**:

fixes \( f g :: \text{nat } \Rightarrow 'a::real-normed-field \)

assumes \( \text{Bseq } f \) and \( \text{Bseq } g \)

shows \( \text{Bseq } (\lambda x. f x * g x) \)
proof  
  from assms obtain K1 K2 where K: norm (f x) ≤ K1 K1 > 0 norm (g x) ≤ K2 K2 > 0  
  for x  
  unfolding Bseq-def by blast  
then have norm (f x * g x) ≤ K1 * K2 for x  
  by (auto simp: norm-mult intro!: mult_mono)  
then show ?thesis by (rule BseqI')  
qed

lemma Bfun-const [simp]: Bfun (λ-::real-normed-field. c) F  
unfolding Bfun-metric-def by (auto intro!: exI[of - c] exI[of - 1::real])

lemma Bseq-cmult-iff:  
  fixes c :: 'a::real-normed-field  
  assumes c ≠ 0  
  shows Bseq (λx::'a. c * f x) ←→ Bseq f  
proof  
  assume Bseq (λx::'a. c * f x)  
  with Bfun-const have Bseq (λx. inverse c * (c * f x))  
    by (rule Bseq-mult)  
  with (c ≠ 0) show Bseq f  
  by (simp add: field-split-simps)  
qed (intro Bseq-mult Bfun-const)

lemma Bseq-subseq: Bseq f ⩾ Bseq (λx. f (g x))  
for f :: nat ⇒ 'a::real-normed-vector  
unfolding Bseq-def by auto

lemma Bseq-Suc-iff: Bseq (λn::nat. f (Suc n)) ←→ Bseq f  
for f :: nat ⇒ 'a::real-normed-vector  
using Bseq-offset[of f 1] by (auto intro: Bseq-subseq)

lemma increasing-Bseq-subseq-iff:  
  assumes ⋀x y. x ≤ y ⇒ norm (f x :: 'a::real-normed-vector) ≤ norm (f y)  
  strict-mono g  
  shows Bseq (λx::'a. f (g x)) ←→ Bseq f  
proof  
  assume Bseq (λx::'a. f (g x))  
  then obtain K where K: ⋀x. norm (f (g x)) ≤ K  
    unfolding Bseq-def by auto  
  {  
    fix x :: nat  
    from filterlim-subseq[OF assms(2)] obtain y where g y ≥ x  
      by (auto simp: filterlim-at-top eventually-at-top-linorder)  
    then have norm (f x) ≤ norm (f (g y))  
      using assms(1) by blast  
    also have norm (f (g y)) ≤ K by (rule K)  
    finally have norm (f x) ≤ K .  
  }
THEORY “Limits”

{ }
then show Bseq f by (rule BseqI’)
qed (use Bseq-subseq[of f g] in simp-all)

lemma nonneg-incseq-Bseq-subseq-iff:
fixes f :: nat ⇒ real
and g :: nat ⇒ nat
assumes ∀ x. f x ≥ 0 incseq f strict-mono g
shows Bseq (λx. f (g x)) ←→ Bseq f
using assms by (intro increasing-Bseq-subseq-iff) (auto simp: incseq-def)

lemma Bseq-eq-bounded:
range f ⊆ {a..b} =⇒ Bseq f
for a b :: real
proof (rule BseqI’ [where K=max (norm a) (norm b)])
fix n assume range f ⊆ {a..b}
then have f n ∈ {a..b}
  by blast
then show norm (f n) ≤ max (norm a) (norm b)
  by auto
qed

lemma incseq-bounded: incseq X =⇒ ∀ i. X i ≤ B =⇒ Bseq X
for B :: real
by (intro Bseq-eq-bounded[of X X 0 B]) (auto simp: incseq-def)

lemma decseq-bounded: decseq X =⇒ ∀ i. B ≤ X i =⇒ Bseq X
for B :: real
by (intro Bseq-eq-bounded[of X B X 0]) (auto simp: decseq-def)

106.1.5 Polynomial function extremal theorem, from HOL Light

lemma polyfun-extremal-lemma:
  fixes c :: nat ⇒ ‘a::real-normed-div-algebra
  assumes 0 < e
  shows ∃ M. ∀ z. M ≤ norm(z) =⇒ norm (∑ i≤n. c(i) * z^i) ≤ e * norm(z) ≤ Suc n
proof (induct n)
case 0 with assms
  show ?case
  apply (rule_tac x=norm (c 0) / e in exI)
  apply (auto simp: field-simps)
  done
next
case (Suc n)
obtain M where M: ∀ z. M ≤ norm z =⇒ norm (∑ i≤n. c i * z^i) ≤ e * norm z ≤ Suc n
  using Suc assms by blast
show ?case
proof (rule exI [where x= max M (1 + norm(c(Suc n)) / e)], clarsimp simp

THEORY "Limits"

1956

del: power-Suc

fix :: 'a

assume z1: M ≤ norm z and f + norm (c (Suc n)) / e ≤ norm z
then have z2: e + norm (c (Suc n)) ≤ e * norm z
  using assms by simp add: field-simps
have norm (∑ i≤n. c i * z^i) ≤ e * norm z ^ Suc n
  using M [OF z1] by simp
then have norm (∑ i≤n. c i * z^i) + norm (c (Suc n) * z ^ Suc n) ≤ e * norm z ^ Suc n + norm (c (Suc n) * z ^ Suc n)
byp simp
then have norm ((∑ i≤n. c i * z^i) + c (Suc n) * z ^ Suc n) ≤ e * norm z ^ Suc n
  by (blast intro: norm-triangle-le elim: )
also have ... ≤ (e + norm (c (Suc n))) * norm z ^ Suc n
  by (simp add: norm-power norm-mult algebra-simps)
also have ... ≤ (e * norm z) * norm z ^ Suc n
  by (metis z2 mult.commute mult-left-mono norm-ge-zero norm-power)
finally show norm ((∑ i≤n. c i * z^i) + c (Suc n) * z ^ Suc n) ≤ e * norm z ^ Suc (Suc n)
  by simp
qed

lemma polyfun-extremal: 
  fixes c :: nat ⇒ 'a::real-normed-div-algebra
assumes k: c k ≠ 0 1 ≤ k and kn: k≤n
  shows eventually (λz. norm (∑ i≤n. c(i) * z^i) ≥ B) at-infinity
using kn
proof (induction n)
case 0
  then show ?case
    using k by simp
next
case (Suc m)
  let ?even = ?case
  using k by simp
next
  show ?even
  proof (cases c (Suc m) = 0)
    case True
    then show ?even using Suc k
      by auto (metis antisym-conv less-eq-Suc-le not-le)
next
  case False
  then obtain M where M:
    (∀z. M ≤ norm z → norm (∑ i≤m. c i * z^i) ≤ norm (c (Suc m)) / 2
    * norm z ^ Suc m
  using polyfun-extremal-lemma [of norm(c (Suc m)) / 2 c m] Suc
  by auto
have ∃b. ∀z. b ≤ norm z → B ≤ norm (∑ i≤Suc m. c i * z^i)
proof (rule exI [where x=max M (max 1 (|B| / (norm(c (Suc m)) / 2))])

clarisimp simp del: power-Suc

fix z : 'a
assume z1 : M ≤ norm z 1 ≤ norm z
and |B| * 2 / norm (c (Suc m)) ≤ norm z
then have z2 : |B| ≤ norm (c (Suc m)) * norm z / 2
  using False by (simp add: field-simps)
have nz : norm z ≤ norm z * Suc m
by (metis (1 ≤ norm z) One-nat-def less-eq-Suc-le power-increasing power-one-right zero-less-Suc)

have * : (∃ y x. norm (c (Suc m)) * norm z / 2 ≤ norm y − norm x) ⇒ B
≤ norm (x + y)
  by (metis abs-le-iff add.commute norm-diff-ineq order-trans z2)

have norm z * norm (c (Suc m)) + 2 * norm (∑ i≤m. c i * z^i)
   ≤ norm (c (Suc m)) * norm z + norm (c (Suc m)) * norm z ^ Suc m
  using M [of z] Suc z1 by auto
also have ... ≤ 2 * (norm (c (Suc m)) * norm z ^ Suc m)
  using nz by (simp add: mult-mono del: power-Suc)
finally show B ≤ norm (∑ i≤m. c i * z^i) + c (Suc m) * z ^ Suc m
  using Suc.IH
apply (auto simp: eventually-at-infinity)
apply (rule *)
apply (simp add: field-simps norm-mult norm-power)
done
qed

then show ?even
  by (simp add: eventually-at-infinity)
qed

106.2 Convergence to Zero

definition Zfun :: ('a ⇒ 'b::real-normed-vector) ⇒ 'a filter ⇒ bool
  where Zfun f F = (∀ r>0. eventually (∀ x. norm (f x) < r) F)

lemma ZfunI: (∀ r. 0 < r ⇒ eventually (∀ x. norm (f x) < r) F) ⇒ Zfun f F
  by (simp add: Zfun-def)

lemma ZfunD: Zfun f F ⇒ 0 < r ⇒ eventually (∀ x. norm (f x) < r) F
  by (simp add: Zfun-def)

lemma Zfun-ssubst: eventually (∀ x. f x = g x) F ⇒ Zfun g F ⇒ Zfun f F
unfolding Zfun-def by (auto elim!: eventually-rev-mp)

lemma Zfun-zero: Zfun (λx. 0) F
unfolding Zfun-def by simp

lemma Zfun-norm-iff: Zfun (λx. norm (f x)) F = Zfun (λx. f x) F
unfolding Zfun-def by simp
lemma Zfun-imp-Zfun:
assumes f: Zfun f F
and g: eventually (λx. norm (g x) ≤ norm (f x) * K) F
shows Zfun (λx. g x) F
proof (cases 0 < K)
case K: True
show ?thesis
proof (rule ZfunI)
fix r :: real
assume 0 < r
then have 0 < r / K using K by simp
then have eventually (λx. norm (f x) < r / K) F
using ZfunD [OF f] by blast
with g show eventually (λx. norm (g x) < r) F
proof eventually-elim
case (elim x)
then have norm (f x) * K < r
by (simp add: pos-less-divide-eq K)
then show ?case
by (simp add: order-le-less-trans [OF elim(1)])
qed
qed
next
case False
then have K: K ≤ 0 by (simp only: not-less)
show ?thesis
proof (rule ZfunI)
fix r :: real
assume 0 < r
from g show eventually (λx. norm (g x) < r) F
proof eventually-elim
case (elim x)
also have norm (f x) * K ≤ norm (f x) * 0
using K norm-ge-zero by (rule mult-left-mono)
finally show ?case
using ⟨0 < r⟩ by simp
qed
qed
qed

lemma Zfun-le: Zfun g F ⇒ ∀x. norm (f x) ≤ norm (g x) ⇒ Zfun f F
by (erule Zfun-imp-Zfun [where K = 1]) simp

lemma Zfun-add:
assumes f: Zfun f F
and g: Zfun g F
shows Zfun (λx. f x + g x) F
proof (rule ZfunI)
fix r :: real
assume $0 < r$
then have $r: 0 < r / 2$ by simp
have eventually $(\lambda x. \text{norm } (f x) < r/2) \Rightarrow \text{F}$
using $f : r$ by (rule $\text{ZfunD}$)
moreover
have eventually $(\lambda x. \text{norm } (g x) < r/2) \Rightarrow \text{F}$
using $g : r$ by (rule $\text{ZfunD}$)
ultimately
show eventually $(\lambda x. \text{norm } (f x + g x) < r) \Rightarrow \text{F}$
proof eventually-elim
  case (elim $x$
    have $\text{norm } (f x + g x) \leq \text{norm } (f x) + \text{norm } (g x)$
    by (rule norm-triangle-ineq)
    also have $\ldots < r/2 + r/2$
      using elim by (rule add-strict-mono)
    finally show ?thesis
      by simp
  qed
qed

lemma $\text{Zfun-minus}$: $\text{Zfun } f \Rightarrow \text{Zfun } (\lambda x. - f x) \Rightarrow \text{F}$
unfolding $\text{Zfun-def}$ by simp

lemma $\text{Zfun-diff}$: $\text{Zfun } f \Rightarrow \text{Zfun } g \Rightarrow \text{Zfun } (\lambda x. f x - g x) \Rightarrow \text{F}$
using $\text{Zfun-add } [\text{of } f \Rightarrow \text{Zfun } (\lambda x. - g x)]$ by (simp add: $\text{Zfun-minus}$)

lemma (in bounded-linear) $\text{Zfun}$:
  assumes $g: \text{Zfun } g \Rightarrow \text{F}$
  shows $\text{Zfun } (\lambda x. f (g x)) \Rightarrow \text{F}$
proof
  obtain $K$ where $\text{norm } (f x) \leq \text{norm } x * K$ for $x$
  using bounded by blast
  then have eventually $(\lambda x. \text{norm } (f (g x)) \leq \text{norm } (g x) * K) \Rightarrow \text{F}$
  by simp
  with $g$ show ?thesis
  by (rule $\text{Zfun-imp-Zfun}$)
qed

lemma (in bounded-bilinear) $\text{Zfun}$:
  assumes $f: \text{Zfun } f \Rightarrow \text{F}$
  and $g: \text{Zfun } g \Rightarrow \text{F}$
  shows $\text{Zfun } (\lambda x. f x ** g x) \Rightarrow \text{F}$
proof (rule $\text{ZfunI}$)
  fix $r :: \text{real}$
  assume $r: 0 < r$
  obtain $K$ where $K: 0 < K$
    and $\text{norm-le: norm } (x ** y) \leq \text{norm } x * \text{norm } y * K$ for $x y$
    using pos-bounded by blast
  from $K$ have $K': 0 < \text{inverse } K$
by \((\text{rule positive-imp-inverse-positive})\)
have eventually \((\lambda x. \text{norm} (f x) < r)\) \(F\)
using \(f r\) by \((\text{rule ZfunD})\)
moreover
have eventually \((\lambda x. \text{norm} (g x) < \text{inverse} K)\) \(F\)
using \(g K'\) by \((\text{rule ZfunD})\)
ultimately
show eventually \((\lambda x. \text{norm} (f x ** g x) < r)\) \(F\)
proof eventually-elim
\[
\text{case (elim } x) \\
\text{have } \text{norm} (f x ** g x) \leq \text{norm} (f x) * \text{norm} (g x) * K \\
\text{by } (\text{rule norm-le}) \\
\text{also have } \text{norm} (f x) * \text{norm} (g x) * K < r * \text{inverse} K * K \\
\text{by } (\text{intro mult-strict-right-mono mult-strict-mono'} \text{norm-ge-zero elim } K) \\
\text{also from } K \text{ have } r * \text{inverse} K * K = r \\
\text{by simp} \\
\text{finally show } ?\text{case .} \\
qedsimp
\]

lemma \((\text{in bounded-bilinear})\) \(\text{Zfun-left}: \text{Zfun } f F \Longrightarrow \text{Zfun } (\lambda x. f x ** a) F\)
by \((\text{rule bounded-linear-left [THEN bounded-linear.Zfun]})\)

lemma \((\text{in bounded-bilinear})\) \(\text{Zfun-right}: \text{Zfun } f F \Longrightarrow \text{Zfun } (\lambda x. a ** f x) F\)
by \((\text{rule bounded-linear-right [THEN bounded-linear.Zfun]})\)

lemmas \(\text{Zfun-mult} = \text{bounded-bilinear.Zfun [OF bounded-bilinear-mult]}\)
lemmas \(\text{Zfun-mult-right} = \text{bounded-bilinear.Zfun-right [OF bounded-bilinear-mult]}\)
lemmas \(\text{Zfun-mult-left} = \text{bounded-bilinear.Zfun-left [OF bounded-bilinear-mult]}\)

lemma \(\text{tendsto-Zfun-iff}: (f \longrightarrow a) F = \text{Zfun } (\lambda x. f x - a) F\)
by \((\text{simp only: tendsto-iff Zfun-def dist-norm})\)

lemma \(\text{tendsto-0-le}:\)
\[
(f \longrightarrow 0) F \Longrightarrow \text{eventually } (\lambda x. \text{norm} (g x) \leq \text{norm} (f x) * K) F \Longrightarrow (g \longrightarrow 0) F \\
\text{by } (\text{simp add: Zfun-imp-Zfun tendsto-Zfun-iff})
\]

106.2.1 Distance and norms

lemma \(\text{tendsto-dist}\) \([\text{tendsto-intros}]:\)
fixes \(l m :: \cdot a::\text{metric-space}\)
assumes \(f: (f \longrightarrow l) F\)
and \(g: (g \longrightarrow m) F\)
shows \(((\lambda x. \text{dist} (f x) (g x)) \longrightarrow \text{dist } l m) F\)
proof \((\text{rule tendsto})\)
fix \(e :: \text{real}\)
assume \(0 < e\)
then have \(e^2: 0 < e/2\) by simp
from \textit{tendstoD} \[OF \, f \, e2\] \textit{tendstoD} \[OF \, g \, e2\]

\textit{show} eventually \((\lambda x. \text{dist} (\text{dist} (f \, x) \, (g \, x)) \, (\text{dist} \, l \, m) < e)\) \(F\)

\textit{proof} (eventually-elim)

\textbf{case} \((\text{elim} \, x)\)

\textit{then show} \text{dist} (\text{dist} (f \, x) \, (g \, x)) \, (\text{dist} \, l \, m) < e

\textit{unfolding} \text{dist-real-def}

\textit{using} \text{dist-triangle2} \[\text{of} \, f \, x \, g \, x \, l\]

\text{and} \text{dist-triangle2} \[\text{of} \, g \, x \, l \, m\]

\text{and} \text{dist-triangle3} \[\text{of} \, l \, m \, f \, x\]

\text{by} \text{arith}

\textit{qed}

\textit{qed}

\textbf{lemma} \textit{continuous-dist}[\textbf{continuous-intros}]:

\textit{fixes} \(f \, g : \Rightarrow 'a \Rightarrow \text{metric-space}\)

\textit{shows} \text{continuous} \(F \, f \Rightarrow \text{continuous} \, F \, g \Rightarrow \text{continuous} \, F \, (\lambda x. \text{dist} (f \, x) \, (g \, x))\)

\textit{unfolding} \text{continuous-def by} (\text{rule} \text{tendsto-dist})

\textbf{lemma} \textit{continuous-on-dist}[\textbf{continuous-intros}]:

\textit{fixes} \(f \, g : \Rightarrow 'a \Rightarrow \text{metric-space}\)

\textit{shows} \text{continuous-on} \(s \, f \Rightarrow \text{continuous-on} \, s \, g \Rightarrow \text{continuous-on} \, s \, (\lambda x. \text{dist} (f \, x) \, (g \, x))\)

\textit{unfolding} \text{continuous-on-def by} (\text{auto intro:} \text{tendsto-dist})

\textbf{lemma} \textit{continuous-at-dist}: \text{isCont} \,(\text{dist} \, a \, b)

\textit{using} \text{continuous-on-dist} \[\text{OF} \, \text{continuous-on-const} \, \text{continuous-on-id], continuous-on-eq-continuous-within}\)

\textit{by} \text{blast}

\textbf{lemma} \textit{tendsto-norm} \[\textit{tendsto-intros}: (f \rightarrow a) \, F \Rightarrow ((\lambda x. \text{norm} \, (f \, x)) \rightarrow \text{norm} \, a) \, F\)

\textit{unfolding} \text{norm-conv-dist by} (\text{intro} \text{tendsto-intros})

\textbf{lemma} \textit{continuous-norm} \[\textit{continuous-intros}]: \text{continuous} \, F \, f \Rightarrow \text{continuous} \, F \, (\lambda x. \text{norm} \, (f \, x))

\textit{unfolding} \text{continuous-def by} (\text{rule} \text{tendsto-norm})

\textbf{lemma} \textit{continuous-on-norm} \[\textit{continuous-intros}]: \text{continuous-on} \, s \, f \Rightarrow \text{continuous-on} \, s \, (\lambda x. \text{norm} \, (f \, x))

\textit{unfolding} \text{continuous-on-def by} (\text{auto intro:} \text{tendsto-norm})

\textbf{lemma} \textit{continuous-on-norm-id} \[\textit{continuous-intros}]: \text{continuous-on} \, s \, \text{norm}

\textit{by} (\text{intro} \text{continuous-on-id} \text{continuous-on-norm})

\textbf{lemma} \textit{tendsto-norm-zero}: \((f \rightarrow 0) \, F \Rightarrow ((\lambda x. \text{norm} \, (f \, x)) \rightarrow 0) \, F\)

\textit{by} (\text{drule} \text{tendsto-norm}) \text{simp}

\textbf{lemma} \textit{tendsto-norm-zero-cancel}: \((\lambda x. \text{norm} \, (f \, x)) \rightarrow 0) \, F \Rightarrow (f \rightarrow 0)
unfolding tendsto-iff dist-norm by simp

lemma tendsto-norm-zero-iff: ((λx. norm (f x)) --> 0) F <-> (f --> 0) F
  unfolding tendsto-iff dist-norm by simp

lemma tendsto-rabs [tendsto-intros]: (f --> l) F --> ((λx. |f x|) --> |l|) F
  for l :: real
  by (fold real-norm-def) (rule tendsto-norm)

lemma continuous-rabs [continuous-intros]:
  continuous F f --> continuous F (λx. |f x :: real|)
  unfolding real-norm-def [symmetric] by (rule continuous-norm)

lemma continuous-on-rabs [continuous-intros]:
  continuous-on s f --> continuous-on s (λx. |f x :: real|)
  unfolding real-norm-def [symmetric] by (rule continuous-on-norm)

lemma tendsto-rabs-zero: (f --> (0 :: real)) F --> ((λx. |f x|) --> 0) F
  by (fold real-norm-def) (rule tendsto-norm-zero)

lemma tendsto-rabs-zero-cancel: ((λx. |f x|) --> (0 :: real)) F --> (f --> 0) F
  by (fold real-norm-def) (rule tendsto-norm-zero-cancel)

lemma tendsto-rabs-zero-iff: ((λx. |f x|) --> (0 :: real)) F <-> (f --> 0) F
  by (fold real-norm-def) (rule tendsto-norm-zero-iff)

106.3 Topological Monoid

class topological-monoid-add = topological-space + monoid-add +
  assumes tendsto-add-Pair: LIM x (nhds a ×F nhds b). fst x + snd x :> nhds (a + b)

class topological-comm-monoid-add = topological-monoid-add + comm-monoid-add

lemma tendsto-add [tendsto-intros]:
  fixes a b :: 'a::topological-monoid-add
  shows (f --> a) F --> (g --> b) F --> ((λx. f x + g x) --> a + b) F
  using filterlim-compose[OF tendsto-add-Pair, of λx. (f x, g x) a b F]
  by (simp add: nhds-prod[ symmetric] tendsto-Pair)

lemma continuous-add [continuous-intros]:
  fixes f g :: - ⇒ 'b::topological-monoid-add
  shows continuous F f --> continuous F g --> continuous F (λx. f x + g x)
  unfolding continuous-def by (rule tendsto-add)

lemma continuous-on-add [continuous-intros]:
  fixes f g :: - ⇒ 'b::topological-monoid-add
  shows continuous-on s f --> continuous-on s g --> continuous-on s (λx. f x +

g x

unfolding continuous-on-def by (auto intro: tendsto-add)

lemma tendsto-add-zero:
  fixes f g :: 'a::topological-monoid-add
  shows (f → 0) F = (g → 0) F → ((λx. f x + g x) → 0) F
  by (drule (1) tendsto-add) simp

lemma tendsto-sum [tendsto-intros]:
  fixes f :: 'a ⇒ 'b::topological-comm-monoid-add
  shows (λi. i ∈ I ⇒ (f i → a i)) F = ((λx. ∑ i∈I. f i x) → (∑ i∈I. a i)) F
  by (induct I rule: infinite-finite-induct) (simp-all add: tendsto-add)

lemma tendsto-null-sum:
  fixes f :: 'a ⇒ 'b::topological-comm-monoid-add
  assumes (λi. i ∈ I ⇒ (λx. f x i) → 0) F
  shows (λi. sum (f i) I → 0) F
  using tendsto-sum [of I λy g. f y x λx. 0] assms by simp

lemma continuous-sum [continuous-intros]:
  fixes f :: 'a ⇒ 'b::t2-space ⇒ 'c::topological-comm-monoid-add
  shows (λi. i ∈ I ⇒ continuous F (f i)) ⇒ continuous F (λx. ∑ i∈I. f i x)
  unfolding continuous-def by (rule tendsto-sum)

lemma continuous-on-sum [continuous-intros]:
  fixes f :: 'a ⇒ 'b::topological-space ⇒ 'c::topological-comm-monoid-add
  shows (λi. i ∈ I ⇒ continuous-on S (f i)) ⇒ continuous-on S (λx. ∑ i∈I. f i x)
  unfolding continuous-on-def by (auto intro: tendsto-sum)

instance nat :: topological-comm-monoid-add
  by standard
  (simp add: nhds-discrete principal-prod-principal filterlim-principal eventually-principal)

instance int :: topological-comm-monoid-add
  by standard
  (simp add: nhds-discrete principal-prod-principal filterlim-principal eventually-principal)

106.3.1 Topological group

class topological-group-add = topological-monoid-add + group-add +
  assumes tendsto-uminus-nhds: (uminus → - a) (nhds a)
begin

lemma tendsto-minus [tendsto-intros]: (f → a) F = ((λx. - f x) → - a) F
  by (rule filterlim-compose[OF tendsto-uminus-nhds])
class topological-ab-group-add = topological-group-add + ab-group-add

lemma continuous-minus [continuous-intros]: continuous F f \implies \text{continuous F (} \lambda x. - f x \text{)}
for f :: 'a::t2-space \Rightarrow 'b::topological-group-add
unfolding continuous-def by (rule tendsto-minus)

lemma continuous-on-minus [continuous-intros]: continuous-on s f \implies \text{continuous-on s (} \lambda x. - f x \text{)}
for f :: - \Rightarrow 'b::topological-group-add
unfolding continuous-on-def by (auto intro: tendsto-minus)

lemma tendsto-minus-cancel: ((\lambda x. - f x) \xrightarrow{} - a) F \Rightarrow (f \xrightarrow{} a) F
for a :: 'a::topological-group-add
by (drule tendsto-minus) simp

lemma tendsto-minus-cancel-left: (f \xrightarrow{} - (y :::::topological-group-add)) F \leftrightarrow ((\lambda x. - f x) \xrightarrow{} y) F
using tendsto-minus-cancel[of f - y F] tendsto-minus[of f - y F]
by auto

lemma tendsto-diff [tendsto-intros]:
fixes a b :: 'a::topological-group-add
shows (f \xrightarrow{} a) F \Rightarrow (g \xrightarrow{} b) F \Rightarrow ((\lambda x. f x - g x) \xrightarrow{} a - b) F
using tendsto-add[of f a F \lambda x. - g x - b] by (simp add: tendsto-minus)

lemma continuous-diff [continuous-intros]:
fixes f g :: 'a::t2-space \Rightarrow 'b::topological-group-add
shows continuous F f \Rightarrow continuous F g \Rightarrow continuous F (\lambda x. f x - g x)
unfolding continuous-def by (rule tendsto-diff)

lemma continuous-on-diff [continuous-intros]:
fixes f g :: - \Rightarrow 'b::topological-group-add
shows continuous-on s f \Rightarrow continuous-on s g \Rightarrow continuous-on s (\lambda x. f x - g x)
unfolding continuous-on-def by (auto intro: tendsto-diff)

lemma continuous-on-op-minus: continuous-on (s ::'a::topological-group-add set)
(((-) x)) (rule continuous-intros | simp)+

instance real-normed-vector < topological-ab-group-add
proof
fix a b :: 'a
show ((\lambda x. fst x + snd x) \xrightarrow{} a + b) (nhds a \times_F nhds b)
unfolding tendsto-Zfun-iff add-diff-add
using tendsto-fst[OF filterlim-ident, of \((a,b)\)] tendsto-snd[OF filterlim-ident, of \((a,b)\)]
by (intro Zfun-add)
show (uminus \longrightarrow - a) (nhds a)
unfolding tendsto-Zfun-iff minus-diff-minus
using filterlim-ident[of nhds a]
by (intro Zfun-minus) (simp add: tendsto-Zfun-iff)
qed

lemmas real-tendsto-sandwich = tendsto-sandwich[where \(a=\text{real}\)]

106.3.2 Linear operators and multiplication

lemma linear-times [simp]: linear \((\lambda x. c * x)\)
for \(c::\text{''a::real-algebra}\)
by (auto simp: linearI distrib-left)

lemma (in bounded-linear) tendsto: \((g \longrightarrow a) F \Longrightarrow ((\lambda x. f (g x)) \longrightarrow f a) F\)
by (simp only: tendsto-Zfun-iff diff [symmetric] Zfun)

lemma (in bounded-linear) continuous: continuous \(F \Longrightarrow \text{continuous } F (\lambda x. f (g x))\)
using tendsto[of g - F] by (auto simp: continuous-def)

lemma (in bounded-linear) continuous-on: continuous-on \(s \Longrightarrow \text{continuous-on } s (\lambda x. f (g x))\)
using tendsto[of g] by (auto simp: continuous-on-def)

lemma (in bounded-linear) tendsto-zero: \((g \longrightarrow 0) F \Longrightarrow ((\lambda x. f (g x)) \longrightarrow 0) F\)
by (drule tendsto) (simp only: zero)

lemma (in bounded-bilinear) tendsto:
\((f \longrightarrow a) F \Longrightarrow (g \longrightarrow b) F \Longrightarrow ((\lambda x. f x ** g x) \longrightarrow a ** b) F\)
by (simp only: tendsto-Zfun-iff prod-diff-prod Zfun-add Zfun Zfun-left Zfun-right)

lemma (in bounded-bilinear) continuous:
continuous \(F \Longrightarrow \text{continuous } F (\lambda x. f x ** g x)\)
using tendsto[of f - F g] by (auto simp: continuous-def)

lemma (in bounded-bilinear) continuous-on:
continuous-on \(s \Longrightarrow \text{continuous-on } s (\lambda x. f x ** g x)\)
using tendsto[of f - g] by (auto simp: continuous-on-def)

lemma (in bounded-bilinear) tendsto-zero:
assumes \(f: (f \longrightarrow 0) F\)
and \( g \): \( g \to 0 \)

shows \( ((\lambda x. \ f \ x \ ** \ g \ x) \to 0) \)

using \( \text{tendsto} \ [\text{OF} \ f \ g] \) by (simp add: zero-left)

lemma (in bounded-bilinear) \( \text{tendsto-left-zero} \):
\( (f \to 0) \Rightarrow ((\lambda x. \ f \ x \ ** \ c) \to 0) \)

by (rule bounded-linear.tendsto-zero [OF bounded-linear-left])

lemma (in bounded-bilinear) \( \text{tendsto-right-zero} \):
\( (f \to 0) \Rightarrow ((\lambda x. \ c \ ** \ f \ x) \to 0) \)

by (rule bounded-linear.tendsto-zero [OF bounded-linear-right])

lemmas \( \text{tendsto-of-real} \) =
bounded-linear.tendsto [OF bounded-linear-of-real]

lemmas \( \text{tendsto-scaleR} \) =
bounded-bilinear.tendsto [OF bounded-bilinear-scaleR]

Analogous type class for multiplication

class \( \text{topological-semigroup-mult} = \text{topological-space} + \text{semigroup-mult} + \)

assumes \( \text{tendsto-mult-Pair} \): \( \text{LIM} x \to \text{nhds} a \times \text{F} \text{nhds} b \)

\( ((\lambda x. \ \text{fst} \ x \ * \ \text{snd} \ x) \to a \times b) \)

instance \( \text{real-normed-algebra} < \text{topological-semigroup-mult} \)

proof
fix \( a \; \text{and} \; b \)

show \( ((\lambda x. \ \text{fst} \ x \ * \ \text{snd} \ x) \to a \times b) \)

unfolding \( \text{nhds-prod[symmetric]} \)

using \( \text{tendsto-fst[OF filterlim-ident, of} \ (a,b)] \) \( \text{tendsto-snd[OF filterlim-ident, of} \ (a,b)] \)

by (simp add: bounded-bilinear.tendsto [OF bounded-bilinear-mult])

qed

lemma \( \text{tendsto-mult} \) [tendsto-intros]:
fixes \( a \; \text{and} \; b \)

shows \( (f \to a) \Rightarrow (g \to b) \Rightarrow ((\lambda x. \ f \ x \ * \ g \ x) \to a \times b) \)

using \( \text{filterlim-compose[OF tendsto-mult-Pair, of} \ \lambda x. \ (f \ x \ * \ g \ x) \text{F}] \)

by (simp add: nhds-prod[symmetric] tendsto-Pair)

lemma \( \text{tendsto-mult-left} \) \( (f \to l) \Rightarrow ((\lambda x. \ c \ * (f \ x)) \to c \times l) \)

for \( c \)  

by (rule tendsto-mult [OF tendsto-const])

lemma \( \text{tendsto-mult-right} \) \( (f \to l) \Rightarrow ((\lambda x. \ (f \ x) \ * \ c) \to l \times c) \)

for \( c \)  

by (rule tendsto-mult [OF - tendsto-const])

lemma \( \text{tendsto-mult-left-iff} \) [simp]:
\( c \neq 0 \Rightarrow \text{tendsto}(\lambda x. \ c \ * f \ x) \Rightarrow \text{tendsto} f \ l \text{F} \)

for \( c \)  

by (rule tendsto-mult [OF - tendsto-const])
by (auto simp: tendsto-mult-left dest: tendsto-mult-left [where c = 1/c])

lemma tendsto-mult-right iff [simp]:
c ≠ 0 ⇒ tendsto(λx. f x * c) (l * c) F ⇔ tendsto f l F for c :: 'a::{topological-semigroup-mult,field}
by (auto simp: tendsto-mult-right dest: tendsto-mult-left [where c = 1/c])

lemma tendsto-zero-mult-left iff [simp]:
fixes c :: 'a::{topological-semigroup-mult,field}
assumes c ≠ 0
shows (λn. c * a n) −→ 0 ⇔ a −→ 0
using assms tendsto-mult-left tendsto-mult-left iff by fastforce

lemma tendsto-zero-mult-right iff [simp]:
fixes c :: 'a::{topological-semigroup-mult,field}
assumes c ≠ 0
shows (λn. a n * c) −→ 0 ⇔ a −→ 0
using assms tendsto-mult-right tendsto-mult-right iff by fastforce

lemma tendsto-zero-divide iff [simp]:
fixes c :: 'a::{topological-semigroup-mult,field}
assumes c ≠ 0
shows (λn. a n / c) −→ 0 ⇔ a −→ 0
using tendsto-zero-mult-right iff [of 1/c a] assms by (simp add: field-simps)

lemma lim-const-over-n [tendsto-intros]:
fixes a :: 'a::real-normed-field
shows (λn. a / of-nat n) −→ 0
using tendsto-mult [OF tendsto-const [of a] lim-1-over-n] by simp

lemmas continuous-of-real [continuous-intros] =
bounded-linear.continuous [OF bounded-linear-of-real]

lemmas continuous-scaleR [continuous-intros] =
bounded-bilinear.continuous [OF bounded-bilinear-scaleR]

lemmas continuous-mult [continuous-intros] =
bounded-bilinear.continuous [OF bounded-bilinear-mult]

lemmas continuous-on-of-real [continuous-intros] =
bounded-linear.continuous-on [OF bounded-linear-of-real]

lemmas continuous-on-scaleR [continuous-intros] =
bounded-bilinear.continuous-on [OF bounded-bilinear-scaleR]

lemmas continuous-on-mult [continuous-intros] =
bounded-bilinear.continuous-on [OF bounded-bilinear-mult]

lemmas tendsto-mult-zero =
bounded-bilinear.tendsto-zero [OF bounded-bilinear-mult]

lemmas tendsto-mult-left-zero =
bounded-bilinear.tendsto-left-zero [OF bounded-bilinear-mult]
lemmas tendsto-mult-right-zero =
bounded-bilinear.tendsto-right-zero [OF bounded-bilinear-mult]

lemma continuous-mult-left:
  fixes c::'a::real-normed-algebra
  shows continuous F f \implies continuous F (λx. c * f x)
  by (rule continuous-mult [OF continuous-const])

lemma continuous-mult-right:
  fixes c::'a::real-normed-algebra
  shows continuous F f \implies continuous F (λx. f x * c)
  by (rule continuous-mult [OF - continuous-const])

lemma continuous-on-mult-left:
  fixes c::'a::real-normed-algebra
  shows continuous-on s f \implies continuous-on s (λx. c * f x)
  by (rule continuous-on-mult [OF continuous-on-const])

lemma continuous-on-mult-right:
  fixes c::'a::real-normed-algebra
  shows continuous-on s f \implies continuous-on s (λx. f x * c)
  by (rule continuous-on-mult [OF - continuous-on-const])

lemma continuous-on-mult-const [simp]:
  fixes c::'a::real-normed-algebra
  shows continuous-on s (λx. c * f x)
  by (intro continuous-on-mult-left continuous-on-id)

lemma tendsto-divide-zero:
  fixes c :: 'a::real-normed-field
  shows (f \longrightarrow 0) F \implies ((λx. f x / c) \longrightarrow 0) F
  by (cases c=0) (simp-all add: divide-inverse tendsto-mult-left-zero)

lemma tendsto-power [tendsto-intros]: (f \longrightarrow a) F \implies ((λx. f x ^ n) \longrightarrow a ^ n) F
  for f :: 'a \Rightarrow 'b::{power,real-normed-algebra}
  by (induct n) (simp-all add: tendsto-mult)

lemma tendsto-null-power: [(f \longrightarrow 0) F; 0 < n] \implies ((λx. f x ^ n) \longrightarrow 0) F
  for f :: 'a \Rightarrow 'b::{power,real-normed-algebra-1}
  using tendsto-power [of f 0 F n] by (simp add: power-0-left)

lemma continuous-power [continuous-intros]: continuous F f \implies continuous F
  (λx. (f x) ^ n)
  for f :: 'a::t2-space \Rightarrow 'b::{power,real-normed-algebra}
  unfolding continuous-def by (rule tendsto-power)
proof
lemma tendsto-power [continuous-intros]:
fixes f :: 'a::{power,real-normed-algebra}
shows continuous-on $s$ f $\Rightarrow$ continuous-on $s$ $(\lambda x. (f x) ^ n)$
unfolding continuous-on-def by (auto intro: tendsto-power)

lemma tendsto-prod [tendsto-intros]:
fixes f :: 'a => 'b::{real-normed-algebra,comm-ring-1}
s (\lambda x. \prod i\in S. f i x)$ $\Rightarrow$ $(\prod i\in S. f i x)$
unfolding continuous-on-def by (rule tendsto-prod)

lemma continuous-on-prod [continuous-intros]:
fixes f :: 'a => 'b::{real-normed-algebra,comm-ring-1}
s (\lambda x. \prod i\in S. f i x)$ $\Rightarrow$ continuous-on $s$ $(\lambda x. \prod i\in S. f i x)$

lemma tendsto-of-real-iff:
((\lambda x. of-real (f x)) :: 'a::real-normed-div-algebra) $\Rightarrow$ of-real c)
F $\Leftarrow$ (f $\Rightarrow$ c) F
unfolding tendsto-iff by simp

lemma tendsto-add-const-iff:
((\lambda x. c + f x :: 'a::real-normed-vector) $\Rightarrow$ c + d) F $\Leftarrow$ (f $\Rightarrow$ d) F
using tendsto-add[OF tendsto-const[of c], of f d]
and tendsto-add[OF tendsto-const[of -c], of \lambda x. c + f x c + d] by auto

class topological-monoid-mult = topological-semigroup-mult + monoid-mult
class topological-comm-monoid-mult = topological-monoid-mult + comm-monoid-mult

lemma tendsto-power-strong [tendsto-intros]:
fixes f :: 'a => 'b::topological-monoid-mult
assumes (f $\Rightarrow$ a) F (g $\Rightarrow$ b) F
shows (f $\Rightarrow$ g x $\Rightarrow$ a $\cdot$ b) F
proof
have (f $\Rightarrow$ a $\cdot$ b) F
  by (induction b) (auto intro: tendsto-intros assms)
also from assms(2) have eventually (\lambda x. g x = b) F
  by (simp add: nhd-discrete filterlim-principal)
  hence eventually (\lambda x. f x $\Rightarrow$ g x $\Rightarrow$ a $\cdot$ b) F
  by eventually-elim simp
  hence (f $\Rightarrow$ g x $\Rightarrow$ a $\cdot$ b) F $\Leftarrow$ ((\lambda x. f x $\Rightarrow$ g x) $\Rightarrow$ a $\cdot$ b) F
  by (intro filterlim-cong refl)
finally show thesis.

qed

lemma continuous-mult [continuous-intros]:
  fixes f g :: - ⇒ 'b::topological-semigroup-mult
  shows continuous F f ⇒ continuous F g ⇒ continuous F (λx. f x * g x)
  unfolding continuous-def by (rule tendsto-mult)

lemma continuous-power [continuous-intros]:
  fixes f :: - ⇒ 'b::topological-monoid-mult
  shows continuous F f ⇒ continuous F g ⇒ continuous F (λx. f x ^ g x)
  unfolding continuous-def by (rule tendsto-power-strong) auto

lemma continuous-on-mult [continuous-intros]:
  fixes f g :: - ⇒ 'b::topological-semigroup-mult
  shows continuous-on A f ⇒ continuous-on A g ⇒ continuous-on A (λx. f x * g x)
  unfolding continuous-on-def by (auto intro: tendsto-mult)

lemma continuous-on-power [continuous-intros]:
  fixes f :: - ⇒ 'b::topological-monoid-mult
  shows continuous-on A f ⇒ continuous-on A g ⇒ continuous-on A (λx. f x ^ g x)
  unfolding continuous-on-def by (auto intro: tendsto-power-strong)

lemma tendsto-mult-one:
  fixes f g :: - ⇒ 'b::topological-monoid-mult
  shows (f ---\rightarrow 1) F ⇒ (g ---\rightarrow 1) F ⇒ ((λx. f x * g x) ---\rightarrow 1) F
  by (drule (1) tendsto-mult) simp

lemma tendsto-prod [tendsto-intros]:
  fixes f :: 'a ⇒ 'b ⇒ 'c::topological-comm-monoid-mult
  shows (\prod i. i ∈ I ⇒ f i ---\rightarrow a i) F ⇒ ((λx. \prod i∈I. f i x) ---\rightarrow (\prod i∈I. a i)) F
  by (induct I rule: infinite-finite-induct) (simp-all add: tendsto-mult)

lemma tendsto-one-prod':
  fixes f :: 'a ⇒ 'b ⇒ 'c::topological-comm-monoid-mult
  assumes \prod i. i ∈ I ⇒ ((λx. f x i) ---\rightarrow 1) F
  shows ((\lambda i. prod (f i) I) ---\rightarrow 1) F
  using tendsto-prod' [of I λx y. f y x λx. 1] assms by simp

lemma continuous-prod [continuous-intros]:
  fixes f :: 'a ⇒ 'b::t2-space ⇒ 'c::topological-comm-monoid-mult
  shows (\prod i. i ∈ I ⇒ continuous F (f i)) ⇒ continuous F (λx. \prod i∈I. f i x)
  unfolding continuous-def by (rule tendsto-prod')

lemma continuous-on-prod' [continuous-intros]:
  fixes f :: 'a ⇒ 'b::topological-space ⇒ 'c::topological-comm-monoid-mult
shows $(\forall i. i \in I \Rightarrow \text{continuous-on } S (f i)) \Rightarrow \text{continuous-on } S (\lambda x. \prod_{i \in I} f_i)$

unfolding continuous-on-def by (auto intro: tendsto-prod')

instance nat :: topological-comm-monoid-mult
by standard
(simp add: nhds-discrete principal-prod-principal filterlim-principal eventually-principal)

instance int :: topological-comm-monoid-mult
by standard
(simp add: nhds-discrete principal-prod-principal filterlim-principal eventually-principal)

class comm-real-normed-algebra-1 = real-normed-algebra-1 + comm-monoid-mult

context real-normed-field
begin
subclass comm-real-normed-algebra-1
proof
from norm-mult[of 1 :: 'a 1] show norm 1 = 1 by simp
qed (simp-all add: norm-mult)
end

106.3.3 Inverse and division

lemma (in bounded-bilinear) Zfun-prod-Bfun:
assumes f: Zfun f F
and g: Bfun g F
shows Zfun (\lambda x. f x ** g x) F
proof –
obtain K where K: 0 ≤ K
and norm-le: \forall x y. \text{norm} (x ** y) ≤ \text{norm} x * \text{norm} y * K
using nonneg-bounded by blast
obtain B where B: 0 < B
and norm-g: eventually (\lambda x. \text{norm} (g x) ≤ B) F
using g by (rule BfunE)
have eventually (\lambda x. \text{norm} (f x ** g x) ≤ \text{norm} (f x) * (B * K)) F
using norm-g proof eventually-elim
case (elim x)
have norm (f x ** g x) ≤ norm (f x) * norm (g x) * K
by (rule norm-le)
also have ... ≤ norm (f x) * B * K
by (intro mult-mono' order-refl norm-g norm-ge-zero mult-nonneg-nonneg K elim)
also have ... = norm (f x) * (B * K)
by (rule mult.assoc)
finally show norm (f x ** g x) ≤ norm (f x) * (B * K).
qed
with f show \( \text{thesis} \)
  by (rule Zfun-imp-Zfun)
qed

lemma (in bounded-bilinear) Bfun-prod-Zfun:
assumes f: Bfun f F
  and g: Zfun g F
shows Zfun (\( \lambda x. f x \ast g x \)) F
using flip g f by (rule bounded-bilinear.Zfun-prod-Bfun)

lemma Bfun-inverse:
fixes a :: 'a::real-normed-div-algebra
assumes f: (f \( \longrightarrow \) a) F
assumes a: a \( \neq \) 0
shows Bfun (\( \lambda x. \) inverse (f x)) F
proof
  from a have 0 < norm a by simp
  then have \( \exists r > 0. r < \text{norm} \ a \) by (rule dense)
  then obtain r where r1: 0 < r and r2: r < norm a
    by blast
  have eventually (\( \lambda x. \) dist (f x) a < r) F
    using tendstoD [OF f r1] by blast
  then have eventually (\( \lambda x. \) norm (inverse (f x)) \leq \text{inverse} \ (norm \ a - r)) F
proof eventually-elim
    case (elim x)
      then have 1: norm (f x - a) < r
        by (simp add: dist-norm)
      then have 2: f x \( \neq \) 0 using r2 by auto
      then have norm (inverse (f x)) = inverse (norm (f x))
        by (rule nonzero-norm-inverse)
      also have \ldots \leq \text{inverse} \ (norm \ a - r)
    proof (rule le-imp-inverse-le)
      show 0 < norm a - r
        using r2 by simp
        have norm a - norm (f x) \leq \text{norm} \ (a - f x)
          by (rule norm-triangle-ineq2)
        also have \ldots = \text{norm} \ (f x - a)
          by (rule norm-minus-commute)
        also have \ldots < r using 1
      finally show norm a - r \leq \text{norm} \ (f x)
        by simp
    qed
  finally show \text{thesis} by (rule BfunI)
qed

lemma tendsto-inverse [tendsto-intros]:
fixes a :: 'a::real-normed-div-algebra
assumes \( f : (f \longrightarrow a) F \)
and \( a : a \neq 0 \)
shows \( ((\lambda x. \text{inverse } (f \, x)) \longrightarrow \text{inverse } a) F \)
proof –
from \( a \) have \( 0 < \text{norm } a \) by simp
with \( f \) have eventually \((\lambda x. \text{dist } (f \, x) a < \text{norm } a) F \)
by (rule tendstoD)
then have eventually \((\lambda x. \, f \, x \neq 0) F \)
unfolding dist-norm by (auto elim!: eventually_mono)
with \( a \) have eventually \((\lambda x. \text{inverse } (f \, x) - \text{inverse } a =
- (\text{inverse } (f \, x) * (f \, x - a) * \text{inverse } a)) F \)
by (auto elim!: eventually_mono simp: inverse_diff_inverse)
moreover have \( \text{Zfun } (\lambda x. - (\text{inverse } (f \, x) * (f \, x - a) * \text{inverse } a)) F \)
by (intro Zfun_minus Zfun_mult_left
Bfun_inverse [OF f a] f [unfolded tendsto-Zfun_iff])
ultimately show \?thesis
unfolding tendsto-Zfun_iff by (rule Zfun_subst)
qed

lemma continuous-inverse:
fixes \( f :: 'a::t2-space \Rightarrow 'b::real-normed-div-algebra \)
assumes continuous \( F \, f \)
and \( f \, (\text{Lim } F \, (\lambda x. \, x)) \neq 0 \)
shows continuous \( F \, (\lambda x. \text{inverse } (f \, x)) \)
using assms unfolding continuous_def by (rule tendsto_inverse)

lemma continuous-at-within-inverse[continuous-intros]:
fixes \( f :: 'a::t2-space \Rightarrow 'b::real-normed-div-algebra \)
assumes continuous \( (\text{at } a \, \text{within } s) \, f \)
and \( f \, a \neq 0 \)
shows continuous \( (\text{at } a \, \text{within } s) \, (\lambda x. \text{inverse } (f \, x)) \)
using assms unfolding continuous_within by (rule tendsto_inverse)

lemma continuous-on-inverse[continuous-intros]:
fixes \( f :: 'a::topological-space \Rightarrow 'b::real-normed-div-algebra \)
assumes continuous-on \( s \, f \)
and \( \forall x \in s. f \, x \neq 0 \)
shows continuous-on \( s \, (\lambda x. \text{inverse } (f \, x)) \)
using assms unfolding continuous_on_def by (blast intro: tendsto_inverse)

lemma tendsto-divide [tendsto-intros]:
fixes \( a \, b :: 'a::real-normed-field \)
shows \( (f \longrightarrow a) \, F \implies (g \longrightarrow b) \, F \implies b \neq 0 \implies ((\lambda x. \, f \, x \, / \, g \, x) \longrightarrow a \, / \, b) \, F \)
by (simp add: tendsto_mult tendsto_inverse divide_inverse)

lemma continuous-divide:
fixes \( f \, g :: 'a::t2-space \Rightarrow 'b::real-normed-field \)
assumes continuous \( F f \)
  and continuous \( F g \)
  and \( g \ (\text{Lim} \ F \ (\lambda x. \ x)) \neq 0 \)
shows continuous \( F \ (\lambda x. \ (f x) / (g x)) \)
using assms unfolding continuous-def by (rule tendsto-divide)

lemma continuous-at-within-divide[continuous-intros]:
fixes \( f g :: 'a::t2-space \Rightarrow 'b::real-normed-field \)
assumes continuous (at a within s) \( f \) continuous (at a within s) \( g \)
  and \( g a \neq 0 \)
shows continuous (at a within s) \( (\lambda x. \ (f x) / (g x)) \)
using assms unfolding continuous-within by (rule tendsto-divide)

lemma isCont-divide[continuous-intros, simp]:
fixes \( f g :: 'a::t2-space \Rightarrow 'b::real-normed-field \)
assumes isCont \( f a \) isCont \( g a \) \( g a \neq 0 \)
shows isCont \( (\lambda x. \ (f x) / (g x)) a \)
using assms unfolding continuous-at by (rule tendsto-divide)

lemma continuous-on-divide[continuous-intros]:
fixes \( f :: 'a::topological-space \Rightarrow 'b::real-normed-field \)
assumes continuous-on s \( f \) continuous-on s \( g \)
  and \( \forall x \in s. \ g x \neq 0 \)
shows continuous-on s \( (\lambda x. \ (f x) / (g x)) \)
using assms unfolding continuous-on-def by (blast intro: tendsto-divide)

lemma tendsto-sgn [tendsto-intros]: \( (f \longrightarrow l) \ F \Longrightarrow l \neq 0 \Longrightarrow ((\lambda x. \ sgn \ (f x)) \longrightarrow sgn \ l) \ F \)
for \( l :: 'a::real-normed-vector \)
unfolding sgn-div-norm by (simp add: tendsto-intros)

lemma continuous-sgn:
fixes \( f :: 'a::t2-space \Rightarrow 'b::real-normed-vector \)
assumes continuous \( F f \)
  and \( f \ (\text{Lim} \ F \ (\lambda x. \ x)) \neq 0 \)
shows continuous \( F \ (\lambda x. \ sgn \ (f x)) \)
using assms unfolding continuous-def by (rule tendsto-sgn)

lemma continuous-at-within-sgn[continuous-intros]:
fixes \( f :: 'a::t2-space \Rightarrow 'b::real-normed-vector \)
assumes continuous (at a within s) \( f \)
  and \( f a \neq 0 \)
shows continuous (at a within s) \( (\lambda x. \ sgn \ (f x)) \)
using assms unfolding continuous-within by (rule tendsto-sgn)

lemma isCont-sgn[continuous-intros]:
fixes \( f :: 'a::t2-space \Rightarrow 'b::real-normed-vector \)
assumes isCont \( f a \)
  and \( f a \neq 0 \)
THEORY “Limits”

shows isCont (λx. sgn (f x)) a
using assms unfolding continuous-at by (rule tendsto-sgn)

lemma continuous-on-sgn[continuous-intros]:
fixes f :: 'a::topological-space ⇒ 'b::real-normed-vector
assumes continuous-on s f
and ∀ x∈s. f x ≠ 0
shows continuous-on s (λx. sgn (f x))
using assms unfolding continuous-on-def by (blast intro: tendsto-sgn)

lemma filterlim-at-infinity:
fixes f :: 'a::real-normed-vector
assumes 0 ≤ c
shows (LIM x F. f x :> at-infinity) ⟷ (∀ r>c. eventually (λx. r ≤ norm (f x)) F)
unfolding filterlim-iff eventually-at-infinity
proof safe
fix P :: 'a ⇒ bool
fix b
assume *: ∀ r>c. eventually (λx. r ≤ norm (f x)) F
assume P: ∀ x. b ≤ norm x ⟹ P x
have max b (c + 1) > c by auto
with * have eventually (λx. max b (c + 1) ≤ norm (f x)) F
  by auto
then show eventually (λx. P (f x)) F
proof eventually-elim
  case (elim x)
    with P show P (f x) by auto
qed
qed force

lemma filterlim-at-infinity-imp-norm-at-top:
fixes F
assumes filterlim f at-infinity F
shows filterlim (λx. norm (f x)) at-top F
proof –
  { fix r :: real
    have ∀ x in F. r ≤ norm (f x) using filterlim-at-infinity[of 0 F] assms
      by (cases r > 0)
        (auto simp: not-less intro: always-eventually order.trans[OF - norm-ge-zero])
  }
thus ?thesis by (auto simp: filterlim-at-top)
qed

lemma filterlim-norm-at-top-imp-at-infinity:
fixes F
assumes filterlim (λx. norm (f x)) at-top F
shows filterlim f at-infinity F
using `filterlim-at-infinity[of 0 f F]` assms by (auto simp: `filterlim-at-top`)

lemma `filterlim-norm-at-top`: `filterlim norm at-top at-infinity`
  by (rule `filterlim-at-infinity-imp-norm-at-top`) (rule `filterlim-ident`)

lemma `filterlim-at-infinity-conv-norm-at-top`:
  `filterlim f at-infinity G` `←→` `filterlim (λx. norm (f x)) at-top G`
  by (auto simp: `filterlim-at-infinity` [OF `order.refl`] `filterlim-at-top-ge` [of `-0`])

lemma `eventually-not-equal-at-infinity`:
  `eventually (λx. x ≠ (a :: 'a :: {real-normed-vector}')) at-infinity`
  proof –
    from `filterlim-norm-at-top[where 'a = 'a]`
    have `∀ F x in at-infinity. norm a < norm (x::'a)` by (auto simp: `filterlim-at-top-dense`)
    thus `?thesis` by eventually-elim auto
  qed

lemma `filterlim-int-of-nat-at-topD`:
  fixes `F`
  assumes `filterlim (λx. f (int x)) F at-top`
  shows `filterlim f F at-top`
  proof –
    have `filterlim (λx. f (int (nat x))) F at-top`
      by (rule `filterlim-compose[of assms filterlim-nat-sequentially]`)
    also have `?this `←→` `filterlim f F at-top`
      by (intro `filterlim-cong refl eventually-mono` [OF `eventually-ge-at-top[of 0::int]`]) auto
    finally show `?thesis` .
  qed

lemma `filterlim-int-sequentially` [tendsto-intros]:
  `filterlim int at-top sequentially`
  unfolding `filterlim-at-top`
  proof
    fix `C :: int`
    show `eventually (λn. int n ≥ C) at-top`
      using `eventually-ge-at-top[of nat [C]]` by eventually-elim linarith
  qed

lemma `filterlim-real-of-int-at-top` [tendsto-intros]:
  `filterlim real-of-int at-top at-top`
  unfolding `filterlim-at-top`
  proof
    fix `C :: real`
    show `eventually (λn. real-of-int n ≥ C) at-top`
      using `eventually-ge-at-top[of [C]]` by eventually-elim linarith
  qed

lemma `filterlim-abs-real`: `filterlim (abs::real ⇒ real) at-top at-top`
proof (subst filterlim-cong[OF refl refl])
  from eventually-ge-at-top[of 0::real] show eventually (λx::real. |x| = x) at-top
    by eventually-elim simp
qed (simp-all add: filterlim-ident)

lemma filterlim-of-real-at-infinity [tendsto-intros]:
  filterlim (of-real :: real ⇒ 'a :: real-normed-algebra-1) at-infinity at-top
  by (intro filterlim-norm-at-top-imp-at-infinity) (auto simp: filterlim-abs-real)

lemma not-tendsto-and-filterlim-at-infinity:
  fixes c :: 'a::real-normed-vector
  assumes F ≠ bot
  and (f ----> c) F
  and filterlim f at-infinity F
  shows False
proof −
  from tendstoD[OF assms(2), of 1/2]
  have eventually (λx. dist (f x) c < 1/2) F
    by simp
  moreover
  from filterlim-at-infinity[of norm c f F] assms(3)
  have eventually (λx. norm (f x) ≥ norm c + 1) F by simp
ultimately have eventually (λx. False) F
proof eventually-elim
  fix x
  assume A: dist (f x) c < 1/2
  assume norm (f x) ≥ norm c + 1
  also have norm (f x) = dist (f x) 0 by simp
  also have ... ≤ dist (f x) c + dist c 0 by (rule dist-triangle)
  finally show False using A by simp
qed
with assms show False by simp
qed

lemma filterlim-at-infinity-imp-not-convergent:
  assumes filterlim f at-infinity sequentially
  shows ¬ convergent f
  by (rule notI, rule not-tendsto-and-filterlim-at-infinity[OF - - assms])
    (simp-all add: convergent-LIMSEQ-iff)

lemma filterlim-at-infinity-imp-eventually-ne:
  assumes filterlim f at-infinity F
  shows eventually (λz. f z ≠ c) F
proof −
  have norm c + 1 > 0
    by (intro add-nonneg-pos) simp-all
  with filterlim-at-infinity[of norm refl, of f F] assms
  have eventually (λz. norm (f z) ≥ norm c + 1) F
    by blast
then show thesis
  by eventually-elim auto
qed

lemma tendsto-of-nat [tendsto-intros]:
  filterlim (of-nat :: nat ⇒ 'a::real-normed-algebra-1) at-infinity sequentially
proof (subst filterlim-at-infinity[OF order.refl], intro allI impI)
  fix r :: real
  assume r: r > 0
  define n where n = nat ⌈r⌉
  from r have n: ∀ m≥n. of-nat m ≥ r
    unfolding n-def by linarith
  from eventually-ge-at-top[of n] show eventually (λm. norm (of-nat m :: 'a) ≥ r) sequentially
    by eventually-elim (use n in simp-all)
qed

106.4 Relate at, at-left and at-right

This lemmas are useful for conversion between at x to at-left x and at-right x and also at-right (0::'a).

lemmas filterlim-split-at-real = filterlim-split-at[where 'a=real]

lemma filtermap-nhds-shift: filtermap (λx. x - d) (nhds a) = nhds (a - d)
  for a d :: 'a::real-normed-vector
  by (rule filtermap-fun-inverse[where g=λx. x + d])
    (auto intro!: tendsto-eq-intros filterlim-ident)

lemma filtermap-nhds-minus: filtermap (λx. - x) (nhds a) = nhds (- a)
  for a :: 'a::real-normed-vector
  by (rule filtermap-fun-inverse[where g=uminus])
    (auto intro!: tendsto-eq-intros filterlim-ident)

lemma filtermap-at-shift: filtermap (λx. x - d) (at a) = at (a - d)
  for a d :: 'a::real-normed-vector
  by (simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-shift[symmetric])

lemma filtermap-at-right-shift: filtermap (λx. x - d) (at-right a) = at-right (a - d)
  for a d :: real
  by (simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-shift[symmetric])

lemma at-right-to-0: at-right a = filtermap (λx. x + a) (at-right 0)
  for a :: real
  using filtermap-at-right-shift[of -a 0] by simp

lemma filterlim-at-right-to-0:
  filterlim f F (at-right a) ⟷ filterlim (λx. f (x + a)) F (at-right 0)
  for a :: real
THEORY “Limits”

unfolding filterlim-def filtermap-filtermap at-right-to-0[of a] ..

lemma eventually-at-right-to-0:
  eventually P (at-right a) ↔ eventually (λx. P (x + a)) (at-right 0)
  for a :: real
unfolding at-right-to-0[of a] by (simp add: eventually-filtermap)

lemma at-to-0: at a = filtermap (λx. x + a) (at 0)
  for a :: 'a::real-normed-vector
  using filtermap-at-shift[of -a 0] by simp

lemma filterlim-at-to-0:
  filterlim f F (at a) ↔ filterlim (λx. f (x + a)) F (at 0)
  for a :: 'a::real-normed-vector
unfolding filterlim-def filtermap-filtermap at-to-0[of a] ..

lemma eventually-at-to-0:
  eventually P (at a) ↔ eventually (λx. P (x + a)) (at 0)
  for a :: 'a::real-normed-vector
unfolding at-to-0[of a] by (simp add: eventually-filtermap)

lemma filtermap-at-minus: filtermap (λx. - x) (at a) = at (- a)
  for a :: 'a::real-normed-vector
  by (simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus[symmetric])

lemma at-left-minus: at-left a = filtermap (λx. - x) (at-right (- a))
  for a :: real
  by (simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus[symmetric])

lemma at-right-minus: at-right a = filtermap (λx. - x) (at-left (- a))
  for a :: real
  by (simp add: filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus[symmetric])

lemma filterlim-at-left-to-right:
  filterlim f F (at-left a) ↔ filterlim (λx. f (- x)) F (at-right (-a))
  for a :: real
unfolding filterlim-def filtermap-filtermap at-left-minus[of a] ..

lemma eventually-at-left-to-right:
  eventually P (at-left a) ↔ eventually (λx. P (- x)) (at-right (-a))
  for a :: real
unfolding at-left-minus[of a] by (simp add: eventually-filtermap)

lemma filterlim-uminus-at-top-at-bot: LIM x at-bot. - x :: real ⇒ at-top
  unfolding filterlim-at-top eventually-at-bot-dense
  by (metis leI minus-less-iff order-less-asym)

lemma filterlim-uminus-at-bot-at-top: LIM x at-top. - x :: real ⇒ at-bot
unfolding filterlim-at-bot eventually-at-top-dense
by (metis leI less-minus-iff order-less-asym)

lemma at-bot-mirror:
supports \( (\text{at-bot} :: (\text{orderd-ab-group-add,linorder}) \text{ filter}) = \text{filtermap} \text{ uminus} \text{ at-top} \)
apply (\text{rule filtermap-fun-inverse[of uminus, symmetric]})
subgoal unfolding filterlim-at-top filterlim-at-bot eventually-at-bot-linorder using le-minus-iff by auto
subgoal unfolding filterlim-at-bot eventually-at-top-linorder using minus-le-iff by auto
by auto

lemma at-top-mirror:
supports \( (\text{at-top} :: (\text{orderd-ab-group-add,linorder}) \text{ filter}) = \text{filtermap} \text{ uminus} \text{ at-bot} \)
apply (\text{subst at-bot-mirror})
by (\text{auto simp: filtermap-filtermap})

lemma filterlim-at-top-mirror: \( \lim x \text{ at-top}. f x > F \iff \lim x \text{ at-bot}. f (-x :: real) > F \)
unfolding filterlim-def at-top-mirror filtermap-filtermap ..

lemma filterlim-at-bot-mirror: \( \lim x \text{ at-bot}. f x > F \iff \lim x \text{ at-top}. f (-x :: real) > F \)
unfolding filterlim-def at-bot-mirror filtermap-filtermap ..

lemma filterlim-uminus-at-top: \( \lim X \text{ at-top}. f x > at-top \iff \lim X \text{ at-bot}. -f x > at-bot \)
using filterlim-compose[\text{OF filterlim-uminus-at-bot-at-top, of f F}]
and filterlim-compose[\text{OF filterlim-uminus-at-top-at-bot, of } \lambda x. -f x F]
by auto

lemma tendsto-at-botI-sequentially:
fixes f :: real \(\Rightarrow \) 'b::first-countable-topology
assumes *: \( \forall X. \text{filterlim} X \text{ at-bot sequentially } \Rightarrow (\lambda n f (X n)) \rightarrow y \)
shows \( f \rightarrow y \) at-bot
unfolding filterlim-at-bot-mirror
proof (\text{rule tendsto-at-topI-sequentially})
fix X :: nat \(\Rightarrow\) \(real\ \text{assume} \text{filterlim} X \text{ at-top sequentially}
thus \( (\lambda n f (-X n)) \rightarrow y \) \(\text{by (intro *) (auto simp: filterlim-uminus-at-top)}\)
qed

lemma filterlim-at-infinity-imp-filterlim-at-top:
assumes \( \text{filterlim} (f :: 'a \Rightarrow real) \text{ at-infinity} F \)
assumes eventually \( (\lambda x. f x > 0) \) \( F \)
shows \( \text{filterlim} f \text{ at-top} F \)
proof –
from assms(2) have *: eventually \( (\lambda x. \text{norm} f x = f x) \) \( F \) by eventually-elim
simp
  from assms(1) show ?thesis unfolding filterlim-at-infinity-conv-norm-at-top
  by (subst (asm) filterlim-cong[OF refl refl])
qed

lemma filterlim-at-infinity-imp-filterlim-at-bot:
  assumes filterlim (f :: 'a ⇒ real) at-infinity F
  assumes eventually (λx. f x < 0) F
  shows filterlim f at-bot F
proof –
  from assms(2) have *: eventually (λx. norm (f x) = −f x) F
  by eventually-elim
simp
  from assms(1) have filterlim (λx. −f x) at-top F
  unfolding filterlim-at-infinity-conv-norm-at-top
  by (subst (asm) filterlim-cong[OF refl refl])
  thus ?thesis by (simp add: filterlim-uminus-at-top)
qed

lemma filterlim-uminus-at-bot: (LIM x F. f x :> at-bot) ←→ (LIM x F. −(f x) :> at-top)
  unfolding filterlim-uminus-at-top
  by simp

lemma filterlim-inverse-at-top-right: LIM x at-right (0::real). inverse x :> at-top
  unfolding filterlim-at-top-gt[where c=0] eventually-at-filter
proof safe
  fix Z :: real
  assume [arith]: 0 < Z
  then have eventually (λx. x < inverse Z) (nhds 0)
  by (auto simp: eventually-nhds-metric dist-real-def intro: cxI[of - |inverse Z|])
  then show eventually (λx. x ≠ 0 → x ∈ {0<..} → Z ≤ inverse x) (nhds 0)
  by (auto elim!: eventually-mono simp: inverse-eq-divide field-simps)
qed

lemma tendsto-inverse-0:
  fixes x :: - ⇒ 'a::real-normed-div-algebra
  shows (inverse −→ (0::'a)) at-infinity
  unfolding tendsto-Zfun-iff diff-0-right Zfun-def eventually-at-infinity
proof safe
  fix r :: real
  assume 0 < r
  show ∃b. ∀x. b ≤ norm x → norm (inverse x :: 'a) < r
  proof (intro exI[of - inverse (r / 2)] allI implI)
    fix x :: 'a
    from .0 < r have 0 < inverse (r / 2) by simp
    also assume *: inverse (r / 2) ≤ norm x
    finally show norm (inverse x) < r
    using * (0 < r)
    by (subst nonzero-norm-inverse) (simp-all add: inverse-eq-divide field-simps)
  qed
lemma tendsto-add-filterlim-at-infinity:
  fixes c :: 'b::real-normed-vector
  and F :: 'a filter
  assumes (f ----> c) F
  and filterlim g at-infinity F
  shows filterlim (λx. f x + g x) at-infinity F
proof (subst filterlim-at-infinity[OF order-refl, safe])
  fix r :: real
  assume r: r > 0
  from assms(1) have ((λx. norm (f x)) ----> norm c) F
    by (rule tendsto-norm)
  then have eventually (λx. norm (f x) < norm c + 1) F
    by (rule order-tendstoD) simp-all
  moreover from r have r + norm c + 1 > 0
    by (intro add-pos-nonneg) simp-all
  with assms(2) have eventually (λx. norm (g x) ≥ r + norm c + 1) F
    unfolding filterlim-at-infinity[OF order-refl]
    by (elim allE[of - r + norm c + 1]) simp-all
  ultimately show eventually (λx. norm (f x + g x) ≥ r) F
proof eventually-elim
  fix x :: 'a
  assume A: norm (f x) < norm c + 1 and B: r + norm c + 1 ≤ norm (g x)
  from A B have r ≤ norm (g x) − norm (f x)
    by simp
  also have norm (g x) − norm (f x) ≤ norm (g x + f x)
    by (rule norm-diff-ineq)
  finally show r ≤ norm (f x + g x)
    by (simp add: add-ac)
qed

lemma tendsto-add-filterlim-at-infinity':
  fixes c :: 'b::real-normed-vector
  and F :: 'a filter
  assumes filterlim f at-infinity F
  and (g ----> c) F
  shows filterlim (λx. f x + g x) at-infinity F
  by (subst add.commute) (rule tendsto-add-filterlim-at-infinity assms)+

lemma filterlim-inverse-at-right-top: LIM x at-top. inverse x := at-right (0::real)
unfolding filterlim-at
by (auto simp: eventually-at-top-dense)
  (metis tendsto-inverse-0 filterlim-mono at-top-le-at-infinity order-refl)

lemma filterlim-inverse-at-top:
  (f ----> (0 :: real)) F ⟹ eventually (λx. 0 < f x) F ⟹ LIM x F. inverse (f x) := at-top
by (intro filterlim-compose[of filterlim-inverse-at-top-right])
   (simp add: filterlim-def eventually-filtermap eventually-mono at-within-def le-principal)

lemma filterlim-inverse-at-bot-neg:
  \[ \lim x \text{ (at-left (0::real)). inverse x :> at-bot} \]
by (simp add: filterlim-inverse-at-top-right filterlim-uminus-at-bot filterlim-at-left-to-right)

lemma filterlim-inverse-at-bot:
  \[ (f \longrightarrow (0 :: real)) F = \lim x F. inverse (f x) :> at-bot \]
unfolding filterlim-uminus-at-bot inverse-minus-eq[symmetric]
by (rule filterlim-inverse-at-top) (simp-all add: tendsto-minus-cancel-left[symmetric])

lemma at-right-to-top: \[ \text{(at-right (0::real)) = filtermap inverse at-top} \]
by (intro filtermap-fun-inverse[symmetric], where \( g = \text{inverse} \))
   (auto intro: filterlim-inverse-at-top-right filterlim-inverse-at-right-top)

lemma eventually-at-right-to-top:
  \[ \text{eventually P (at-right (0::real))} \iff \text{eventually (\( \lambda x. P (inverse x) \)) at-top} \]
unfolding at-right-to-top eventually-filtermap ..

lemma filterlim-at-right-to-top:
  \[ \text{filterlim f F (at-right (0::real))} \iff \text{at-right-top} \text{ f (inverse x) :> F} \]
unfolding filterlim-def at-right-to-top filtermap-filtermap ..

lemma at-top-to-right: \[ \text{at-top = filtermap inverse (at-right (0::real))} \]
unfolding at-right-to-top filtermap-filtermap inverse-inverse-eq filtermap-ident ..

lemma eventually-at-top-to-right:
  \[ \text{eventually P at-top} \iff \text{eventually (\( \lambda x. P (inverse x) \)) at-right (0::real)} \]
unfolding at-top-to-right eventually-filtermap ..

lemma filterlim-at-top-to-right:
  \[ \text{filterlim f F at-top} \iff \text{at-right-top} \text{ f (inverse x) :> F} \]
unfolding filterlim-def at-top-to-right filtermap-filtermap ..

lemma filterlim-inverse-at-infinity:
  fixes \( x :: - \Rightarrow 'a::{real-normed-div-algebra, division-ring} \)
  shows \[ \text{filterlim inverse at-infinity (at (0::'a))} \]
unfolding filterlim-at-infinity[of order-refl]
proof safe
  fix \( r :: \text{real} \)
  assume \( 0 < r \)
  then show \[ \text{eventually (\( \lambda x::'a. r \leq \text{norm (inverse x)} \)) at 0} \]
    unfolding eventually-at norm-inverse
    by (intro exI[of - inverse r])
      (auto simp: norm-conv-dist[symmetric] field-simps inverse-eq-divide)
qed
lemma filterlim-inverse-at-iff:
fixes g :: 'a ⇒ 'b::{real-normed-div-algebra, division-ring}
shows (LIM x F. inverse (g x) :> at 0) ⇔ (LIM x F. g x :> at-infinity)
unfolding filterlim-def filtermap-filtermap[symmetric]
proof
assume filtermap g F ≤ at-infinity
then have filtermap inverse (filtermap g F) ≤ filtermap inverse at-infinity
  by (rule filtermap-mono)
also have ... ≤ at 0
  using tendsto-inverse-0[where 'a='b]
  by (auto intro: exI[of - 1]
  simp: le-principal eventually-filtermap filterlim-def filtermap-filtermap
finally show filtermap inverse (filtermap g F) ≤ at 0.
next
assume filtermap inverse (filtermap g F) ≤ at 0
then have filtermap inverse (filtermap inverse (filtermap g F)) ≤ filtermap inverse (at 0)
  by (rule filtermap-mono)
with filterlim-inverse-at-infinity show filtermap g F ≤ at-infinity
  by (auto intro: order-trans simp: filterlim-def filtermap-filtermap)
qed

lemma tendsto-mult-filterlim-at-infinity:
fixes c :: 'a::real-normed-field
assumes (f −→ c) F c ≠ 0
assumes filterlim g at-infinity F
shows filterlim (λx. f x * g x) at-infinity F
proof
  have ((λx. inverse (f x) * inverse (g x)) −→ inverse c * 0) F
    by (intro tendsto-mult tendsto-inverse assms filterlim-compose[OF tendsto-inverse-0])
  then have filterlim (λx. inverse (f x) * inverse (g x)) (at (inverse c * 0)) F
    unfolding filterlim-at
    using assms
    by (auto intro: filterlim-at-infinitely-ne tendsto-infinitely-ne eventually-conj)
  then show ?thesis
    by (subst filterlim-inverse-at-iff[symmetric]) simp-all
qed

lemma tendsto-inverse-0-at-top: LIM x F. f x :> at-top ⇒ ((λx. inverse (f x) :: real) −→ 0) F
by (metis filterlim-at filterlim-mono[OF at-top-le-at-infinity order-refl] filterlim-inverse-at-iff)

lemma real-tendsto-divide-at-top:
fixes c::real
assumes (f −→ c) F
assumes filterlim g at-top F
shows ((λx. f x / g x) −→ 0) F
theory "Limits"

by (auto simp: divide_inverse_commute
  intro!: tendsto_mult THEN tendsto_eq_rhs tendsto_inverse_0_at_top assms)

lemma mult_nat_left_at_top: \( c > 0 \Rightarrow \) filterlim \((\lambda x. c * x)\) at_top sequentially
  for \( c :: \text{nat} \)
by (rule filterlim_subseq) (auto simp: strict_mono_def)

lemma mult_nat_right_at_top: \( c > 0 \Rightarrow \) filterlim \((\lambda x. x * c)\) at_top sequentially
  for \( c :: \text{nat} \)
by (rule filterlim_subseq) (auto simp: strict_mono_def)

lemma filterlim_times_pos: \( LIM x F1. c * f x : \) at_right \( l \)
if filterlim \( f \) (at_right \( p \)) \( F1 0 < c \ l = c * p \)
for \( c :: 'a::\{\text{linordered-field}, \text{linorder-topology}\} \)
unfolding filterlim_iff
proof safe
fix \( P \)
assume \( \forall F x \in \text{at-right} \ l. P x \)
then obtain \( d \) where \( c * p < d \land y > c * p \Rightarrow y < d \Rightarrow P y \)
unfolding (l = -)
eventually_at_right_field
by auto
then have \( \forall F a \in \text{at-right} \ p. P (c * a) \)
  by (auto simp: eventually_at_right_field \( \theta < c \) \ field-simps intro!: exI[where \( x=d/c] \))
from that(1)[unfolded filterlim_iff, rule-format, OF this]
show \( \forall F x \in F1. P (c * f x) \).
qed

lemma filtermap_nhds_times: \( c \neq 0 \Rightarrow \) filtermap \((\times c)\) \((\text{nhds} \ a)\) = \(\text{nhds} (c * a)\)
  for \( a :: 'a::\text{real-normed-field} \)
by (rule filtermap_fun_inverse[where \( g=\lambda x. \text{inverse} c * x \)])
(auto intro!: tendsto_eqintros filterlim_ident)

lemma filtermap_times_pos_at_right:
  fixes \( c :: 'a::\{\text{linordered-field}, \text{linorder-topology}\} \)
  assumes \( c > 0 \)
  shows filtermap \((\times c)\) \((\text{at-right} \ p)\) = \(\text{at-right} (c * p)\)
  using assms
  by (intro filtermap_fun_inverse[where \( g=\lambda x. \text{inverse} c * x \)])
(auto intro!: filterlim_ident filterlim_times_pos)

lemma at_to_infinity: \((\text{at} (\theta::'a::\{\text{real-normed-field}, \text{field}\}))\) = \(\text{filtermap inverse \ at-infinity}\)
proof (rule antisym)
  have \(\text{inverse} \to (\theta::'a)\) \(\text{at-infinity}\)
    by (fact tendsto_inverse_0)
  then show \(\text{filtermap inverse \ at-infinity} \leq \text{at} (\theta::'a)\)
    using filterlim_def filterlim_ident filterlim_inverse_at_iff
    by (fastforce)
THEORY "Limits"
next
have filtermap inverse (filtermap inverse (at (0::'a))) ≤ filtermap inverse at-infinity
  using filterlim-inverse-at-infinity unfolding filterlim-def
  by (rule filtermap-mono)
then show at (0::'a) ≤ filtermap inverse at-infinity
  by (simp add: filtermap-ident filtermap-filtermap)
qed

lemma lim-at-infinity-0:
  fixes l :: 'a::{real-normed_field, field}
  shows (f ----> l) at-infinity <-> ((f o inverse) ----> l) (at (0::'a))
  by (simp add: tendsto-compose-filtermap at-to-infinity filtermap-filtermap)

lemma lim-zero-infinity:
  fixes l :: 'a::{real-normed_field, field}
  shows (λx. f (1 / x)) ----> l (at (0::'a)) = (f ----> l) at-infinity
  by (simp add: inverse-eq-divide lim-at-infinity-0 comp-def)

We only show rules for multiplication and addition when the functions are
either against a real value or against infinity. Further rules are easy to derive

lemma filterlim-tendsto-pos-mult-at-top:
  assumes f: (f ----> c) F
  and c: 0 < c
  and g: LIM x F. g x :> at-top
  shows LIM x F. (f x * g x :: real) :> at-top
  unfolding filterlim-at-top-mult-at-top[where c=0]
proof safe
  fix Z :: real
  assume 0 < Z
  from f ⟨0 < c⟩ have eventually (λx. c / 2 < f x) F
    by (auto dest!: tendstoD[where e=c / 2] elim!: eventually-mono
        simp: dist-real-def abs-real-def split: if-split_asm)
  moreover from g have eventually (λx. (Z / c * 2) ≤ g x) F
    unfolding filterlim-at-top by auto
  ultimately show eventually (λx. Z ≤ f x * g x) F
proof eventually-elim
  case (elim x)
  with ⟨0 < Z⟩ ⟨0 < c⟩ have c / 2 * (Z / c * 2) ≤ f x * g x
    by (intro mult-mono) (auto simp: zero-le-divide-iff)
  with ⟨0 < c⟩ show Z ≤ f x * g x
    by simp
qed

lemma filterlim-at-top-mult-at-top:
  assumes f: LIM x F. f x :> at-top
  and g: LIM x F. g x :> at-top
  shows LIM x F. (f x * g x :: real) :> at-top
unfolding filterlim-at-top-\text{gt}[\text{where } c=0]

proof {safe
  \fix \hfill Z :: real
  \assume 0 < Z
  from f have eventually (\lambda x. 1 \leq f x) F
  unfolding filterlim-at-top by auto
  moreover from g have eventually (\lambda x. Z \leq g x) F
  unfolding filterlim-at-top by auto
  ultimately show eventually (\lambda x. Z \leq f \ast g x) F
proof {eventually-elim
  case \elim x
  with \langle 0 < Z \rangle have 1 \ast Z \leq f \ast g x
  by (intro mult-mono) (auto simp: zero-le-divide-iff)
  then show Z \leq f \ast g x
  by simp
qed

lemma filterlim-at-top-mult-tendsto-pos:
  \assumes f: \( f \longrightarrow c \) F
  and c: 0 < c
  and g: LIM x F. g x :> at-top
  shows LIM x F. (g x \ast f x:: real) :> at-top
by (auto simp: mult.commute intro!: filterlim-tendsto-pos-mult-at-top f c g)

lemma filterlim-tendsto-pos-mult-at-bot:
  \fixes c :: real
  \assumes \( f \longrightarrow c \) F \( 0 < c \) filterlim g at-bot F
  shows LIM x F. f x \ast g x :> at-bot
using filterlim-tendsto-pos-mult-at-top[OF assms(1,2), of \( \lambda x. - g x \)] assms(3)
unfolding filterlim-uminus-at-bot by simp

lemma filterlim-tendsto-neg-mult-at-bot:
  \fixes c :: real
  \assumes c: \( f \longrightarrow c \) F \( c < 0 \) and g: filterlim g at-top F
  shows LIM x F. f x \ast g x :> at-bot
using c filterlim-tendsto-pos-mult-at-top[of \( \lambda x. - f x - c F \), OF - - g]
unfolding filterlim-uminus-at-bot tendsto-minus-cancel-left by simp

lemma filterlim-pow-at-top:
  \fixes f :: 'a \Rightarrow real
  \assumes \( 0 < n \)
  and f; LIM x F. f x :> at-top
  shows LIM x F. (f x)\^n :: real :> at-top
using \( 0 < n \)
proof (induct n)
  case \( \text{0} \)
  then show \?case by simp
next
case (Suc n) with f show ?case
  by (cases n = 0) (auto intro!: filterlim-at-top-mult-at-top)
qed

lemma filterlim-pow-at-bot-even:
  fixes f :: real
  shows 0 < n =⇒ LIM x F. f x :> at-bot =⇒ even n =⇒ LIM x F. (f x) ^ n :> at-top
  using filterlim-pow-at-top[of n λx. −f x F] by (simp add: filterlim-uminus-at-top)

lemma filterlim-pow-at-bot-odd:
  fixes f :: real
  shows 0 < n =⇒ LIM x F. f x :> at-bot =⇒ odd n =⇒ LIM x F. (f x) ^ n :> at-bot
  using filterlim-pow-at-top[of n λx. −f x F] by (simp add: filterlim-uminus-at-bot)

lemma filterlim-power-at-infinity [tendsto-intros]:
  fixes F and f :: 'a ⇒ 'b :: real-normed-div-algebra
  assumes filterlim f at-infinity F n > 0
  shows filterlim (λx. f x ^ n) at-infinity F
  by (rule filterlim-norm-at-top-imp-at-infinity)
    (auto simp: norm-power intro: filterlim-pow-at-top assms intro: filterlim-at-infinity-imp-norm-at-top)

lemma filterlim-tendsto-add-at-top:
  assumes f: (f −−−→ c) F and g: LIM x F. g x :> at-top
  shows LIM x F. (f x + g x :: real) :> at-top
  unfolding filterlim-at-top-gt[where c=0]
proof safe
  fix Z :: real
  assume 0 < Z
  from f have eventually (λx. c − 1 < f x) F
    by (auto dest!: tendstoD[where e=1] elim!: eventually-mono simp: dist-real-def)
  moreover from g have eventually (λx. Z − (c − 1) ≤ g x) F
    unfolding filterlim-at-top by auto
  ultimately show eventually (λx. Z ≤ f x + g x) F
    by eventually-elim simp
qed

lemma LIM-at-top-divide:
  fixes f g :: 'a ⇒ real
  assumes f: (f −−−→ a) F 0 < a
    and g: (g −−−→ 0) F eventually (λx. 0 < g x) F
  shows LIM x F. f x / g x :> at-top
  unfolding divide-inverse
by (rule filterlim-tendsto-pos-mult-at-top[OF f]) (rule filterlim-inverse-at-top[OF g])
lemma filterlim-at-top-add-at-top:
assumes f: LIM x F. f x : at-top
and g: LIM x F. g x : at-top
shows LIM x F. (f x + g x : real) : at-top
unfolding filterlim-at-top-gt[where c=0]
proof safe
fix Z :: real
assume 0 < Z
from f have eventually (λx. 0 ≤ f x) F
  unfolding filterlim-at-top by auto
moreover from g have eventually (λx. Z ≤ g x) F
  unfolding filterlim-at-top by auto
ultimately show eventually (λx. Z ≤ f x + g x) F
  by eventually-elim simp
qed

lemma tendsto-divide-0:
fixes f :: real ⇒ 'a::{real-normed-div-algebra, division-ring}
assumes f: (f −−−→ c) F
and g: LIM x F. g x : at-infinity
shows ((λx. f x / g x) −−−→ 0) F
using tendsto-mult[OF f filterlim-compose[OF tendsto-inverse-0 g]]
by (simp add: divide-inverse)

lemma linear-plus-1-le-power:
fixes x :: real
assumes x: 0 ≤ x
shows real n * x + 1 ≤ (x + 1) ^ n
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from x have real (Suc n) * x + 1 ≤ (x + 1) * (real n * x + 1)
    by (simp add: field-simps)
  also have ... ≤ (x + 1) ^ Suc n
    using Suc x by (simp add: mult-left-mono)
  finally show ?case .
qed

lemma filterlim-realpow-sequentially-gt1:
fixes x :: 'a :: real-normed-div-algebra
assumes x[arith]: 1 < norm x
shows LIM n sequentially. x ^ n : at-infinity
proof (intro filterlim-at-infinity[THEN iffD2] allI impI)
  fix y :: real
  assume 0 < y
  have 0 < norm x - 1 by simp
  then obtain N :: nat where y < real N * (norm x - 1)
also have \( \ldots \leq \text{real } N \cdot (\text{norm } x - 1) + 1 \)
by \text{simp}
also have \( \ldots \leq (\text{norm } x - 1 + 1) \cdot N \)
by (rule linear-plus-1-le-power) \text{simp}
also have \( \ldots = \text{norm } x \cdot N \)
by \text{simp}
finally have \( \forall n \geq N. \ y \leq \text{norm } x \cdot n \)
by (metis order-less-le-trans power-increasing order-less-imp-le x)
then show eventually \((\lambda n. \ y \leq \text{norm } (x ^ n))\) sequentially
unfolding eventually-sequentially
by (auto simp: norm-power)
qed \text{simp}

lemma filterlim-divide-at-infinity:
fixes \( f \) :: 'a :: real-normed-field
assumes filterlim f (nhds c) F filterlim g (at 0) F c \neq 0
shows filterlim \((\lambda x. f x / g x)\) at-infinity F
proof –
  have filterlim \((\lambda x. f x \cdot \text{inverse } (g x))\) at-infinity F
    by (intro tendsto-mult-filterlim-at-infinity[of assms(1,3)])
  thus \(?thesis\) by (simp add: field-simps)
qed

106.5 Floor and Ceiling

lemma eventually-floor-less:
fixes \( f \) :: 'a \Rightarrow 'b::{order-topology,floor-ceiling}
assumes \( f: (f \longrightarrow l) F \)
and \( l: l \notin \mathbb{Z} \)
shows \( \forall F \ x \in F. \ \text{of-int } (\text{floor } l) < f x \)
by (intro order-tendstoD[OF f]) (metis Ints-of-int antisym-conv2 floor-correct l)

lemma eventually-less-ceiling:
fixes \( f \) :: 'a \Rightarrow 'b::{order-topology,floor-ceiling}
assumes \( f: (f \longrightarrow l) F \)
and \( l: l \notin \mathbb{Z} \)
shows \( \forall F \ x \in F. \ f x < \text{of-int } (\text{ceiling } l) \)
by (intro order-tendstoD[OF f]) (metis Ints-of-int l le-of-int-ceiling less-le)

lemma eventually-floor-eq:
fixes \( f \) :: 'a \Rightarrow 'b::{order-topology,floor-ceiling}
assumes \( f: (f \longrightarrow l) F \)
and \( l: l \notin \mathbb{Z} \)
shows \( \forall F \ x \in F. \ \text{floor } (f x) = \text{floor } l \)
using eventually-floor-less[OF assms] eventually-less-ceiling[OF assms]
by eventually-elim (meson floor-less-iff less-ceiling-iff not-less-iff-gr-or-eq)
lemma eventually-ceiling-eq:
  fixes f::'a ⇒ 'b::{order-topology,floor-ceiling}
  assumes f: (f −−−→ l) F
  and l: l /∈ ℤ
  shows ∀F x in F. ceiling (f x) = ceiling l
  using eventually-floor-less[OF assms] eventually-less-ceiling[OF assms]
  by eventually-elim (meson floor-less-iff less-ceiling-iff not-less-iff-gr-or-eq)

lemma tendsto-of-int-floor:
  fixes f::'a ⇒ 'b::{order-topology,floor-ceiling}
  assumes (f −−−→ l) F
  and l /∈ ℤ
  shows ((λx. of-int (floor (f x)) :: 'c::{ring-1,topological-space}) −−−→ of-int (floor l)) F
  using eventually-floor-eq[OF assms]
  by (simp add: eventually-mono topological-tendstoI)

lemma continuous-on-of-int-floor:
  continuous-on (UNIV − ℤ) (λx. of-int (floor x) :: 'c::{ring-1,topological-space})
  unfolding continuous-on-def
  by (auto intro!: tendsto-of-int-floor)

lemma continuous-on-of-int-ceiling:
  continuous-on (UNIV − ℤ) (λx. of-int (ceiling x) :: 'c::{ring-1,topological-space})
  unfolding continuous-on-def
  by (auto intro!: tendsto-of-int-ceiling)

106.6 Limits of Sequences

lemma [trans]: X = Y ⇒ Y −−−→ z ⇒ X −−−→ z
  by simp

lemma LIMSEQ-iff:
  fixes L :: 'a::real-normed-vector
  shows (X −−−→ L) = (∀r>0. ∃no. ∀n ≥ no. norm (X n − L) < r)
  unfolding lim-sequentially dist-norm ..
lemma LIMSEQ-I: \((\forall r. 0 < r \Longrightarrow \exists n \geq \text{no. norm} (X n - L) < r) \Longrightarrow X \longrightarrow L\)
for \(L :: 'a::\text{real-normed-vector}\)
by (simp add: LIMSEQ-iff)

lemma LIMSEQ-D: \(X \longrightarrow L \Longrightarrow 0 < r \Longrightarrow \exists n \geq \text{no. norm} (X n - L) < r\)
for \(L :: 'a::\text{real-normed-vector}\)
by (simp add: LIMSEQ-iff)

lemma LIMSEQ-linear: \(X \longrightarrow x = \Longrightarrow l > 0 = \Longrightarrow (\lambda n. X(n \cdot l)) \longrightarrow x\)
unfolding tendsto-def eventually-sequentially
by (metis div-le-dividend div-mult-self1-is-m le_trans mult.commute)

Transformation of limit.
lemma Lim-transform: \((g \longrightarrow a) \ F = \Longrightarrow ((\lambda x. f x - g x) \longrightarrow 0) \ F = \Longrightarrow (f \longrightarrow a) \ F\)
for \(a b :: 'a::\text{real-normed-vector}\)
using tendsto-add [of g a F \(\lambda x. f x - g x\ 0\)] by simp

lemma Lim-transform2: \((f \longrightarrow a) \ F = \Longrightarrow ((\lambda x. f x - g x) \longrightarrow 0) \ F = \Longrightarrow (g \longrightarrow a) \ F\)
for \(a b :: 'a::\text{real-normed-vector}\)
by (erule Lim-transform) (simp add: tendsto-minus-cancel)

proposition Lim-transform-eq: \(((\lambda x. f x - g x) \longrightarrow 0) \ F = \Longrightarrow (f \longrightarrow a) \ F\)
for \(a :: 'a::\text{real-normed-vector}\)
using Lim-transform Lim-transform2 by blast

lemma Lim-transform-eventually:
\[ [ (f \longrightarrow l) \ F; \text{eventually} (\lambda x. f x = g x) \ F ] = \Longrightarrow (g \longrightarrow l) \ F \]
using eventually-elim2 by (fastforce simp add: tendsto-def)

lemma Lim-transform-within:
assumes \((f \longrightarrow l) (\text{at x within } S)\)
and \(\emptyset < d\)
and \(\forall x'. x' \in S \Longrightarrow 0 < \text{dist } x' x \Longrightarrow \text{dist } x' x < d \Longrightarrow f x' = g x'\)
shows \((g \longrightarrow l) (\text{at x within } S)\)
proof (rule Lim-transform-eventually)
show eventually (\(\lambda x. f x = g x\)) (at x within S)
using assms by (auto simp: eventually-at)
show \((f \longrightarrow l) (\text{at x within } S)\)
by fact
qed

lemma filterlim-transform-within:
assumes filterlim g G (at x within S)
assumes \(G \leq F \emptyset < d \ (\forall x'. x' \in S \Longrightarrow 0 < \text{dist } x' x \Longrightarrow \text{dist } x' x < d \Longrightarrow f\)
THEORY "Limits"  

\[ x' = g \, x' \]

shows \( \lim f \, F \) (at \( x \) within \( S \))
using \( \text{assms} \)
apply (elim filterlim-mono-eventually)
unfolding eventually-at by auto

Common case assuming being away from some crucial point like 0.

**lemma Lim-transform-away-within:**

fixes \( a \, b :: 'a::t1-space \)
assumes \( a \neq b \)
and \( \forall x \in S. \, x \neq a \land x \neq b \longrightarrow f \, x = g \, x \)
and \( (f \longrightarrow l) \, (\text{at } a \, \text{within } S) \)
shows \( (g \longrightarrow l) \, (\text{at } a \, \text{within } S) \)
proof (rule Lim-transform-eventually)
show \( (f \longrightarrow l) \, (\text{at } a \, \text{within } S) \)
by fact
show eventually \( (\lambda x. \, f \, x = g \, x) \) (at \( a \) within \( S \))
unfolding eventually-at-topological
by (rule extI [where \( x =- \{b\} \]) (simp add: open-Compl assms)
qed

**lemma Lim-transform-away-at:**

fixes \( a \, b :: 'a::t1-space \)
assumes \( ab: \, a \neq b \)
and \( \forall x. \, x \neq a \land x \neq b \longrightarrow f \, x = g \, x \)
and \( fl: \, (f \longrightarrow l) \, (\text{at } a) \)
shows \( (g \longrightarrow l) \, (\text{at } a) \)
using Lim-transform-away-within[OF \( ab \), of UNIV \( f \, g \, l \)] \( fg \, fl \) by simp

Alternatively, within an open set.

**lemma Lim-transform-within-open:**

assumes \( (f \longrightarrow l) \, (\text{at } a \, \text{within } T) \)
and open \( S \) and \( a \in S \)
and \( \forall x. \, x \in S \longrightarrow x \neq a \longrightarrow f \, x = g \, x \)
shows \( (g \longrightarrow l) \, (\text{at } a \, \text{within } T) \)
proof (rule Lim-transform-eventually)
show eventually \( (\lambda x. \, f \, x = g \, x) \) (at \( a \) within \( T \))
unfolding eventually-at-topological
using assms by auto
show \( (f \longrightarrow l) \) (at \( a \) within \( T \)) by fact
qed

A congruence rule allowing us to transform limits assuming not at point.

**lemma Lim-cong-within:**

assumes \( a = b \)
and \( x = y \)
and \( S = T \)
and \( \forall x. \, x \neq b \longrightarrow x \in T \longrightarrow f \, x = g \, x \)
shows \( (f \longrightarrow x) \) (at \( a \, \text{within } S) \leftrightarrow (g \longrightarrow y) \) (at \( b \, \text{within } T) \)
unfolding tendsto-def eventually-at-topological
using assms by simp

lemma Lim-cong-at:
  assumes a = b x = y
  and \( \forall x. x \neq a \implies f x = g x \)
  shows ((\( \lambda x. f x \) ----> x) (at a) \( \iff \) \((g ----> y) (at a) \))
unfolding tendsto-def eventually-at-topological
using assms by simp

An unbounded sequence’s inverse tends to 0.

lemma LIMSEQ-inverse-zero:
  assumes \( \forall r :: real. \exists N. \forall n \geq N. r < X n \)
  shows ((\( \lambda n. inverse(X n) \) ----> 0)
apply (rule filterlim-compose[OF tendsto-inverse-0])
by (metis assms eventually-at-top-linorderI filterlim-at-top-dense filterlim-at-top-imp-at-infinity)

The sequence \((1 ::'a) / n\) tends to 0 as \( n \) tends to infinity.

lemma LIMSEQ-inverse-real-of-nat:
  shows ((\( \lambda n. inverse(real(Suc n)) \) ----> 0)
by (metis filterlim-compose tendsto-inverse-0 filterlim-mono order-refl filterlim-Suc filterlim-compose[OF filterlim-real-sequentially] at-top-le-at-infinity)

The sequence \( r + (1 ::'a) / n\) tends to \( r\) as \( n \) tends to infinity is now easily proved.

lemma LIMSEQ-inverse-real-of-nat-add:
  shows ((\( \lambda n. r + inverse(real(Suc n)) \) ----> r)
using tendsto-add [OF tendsto-const LIMSEQ-inverse-real-of-nat] by auto

lemma LIMSEQ-inverse-real-of-nat-add-minus:
  shows ((\( \lambda n. r + inverse(real(Suc n)) \) ----> r)
using tendsto-add [OF tendsto-const tendsto-minus [OF LIMSEQ-inverse-real-of-nat]]
by auto

lemma LIMSEQ-inverse-real-of-nat-add-minus-mult:
  shows ((\( \lambda n. r * (1 + - inverse(real(Suc n))) \) ----> r)
using tendsto-mult [OF tendsto-const LIMSEQ-inverse-real-of-nat-add-minus [of 1]]
by auto

lemma lim-inverse-n:
  shows ((\( \lambda n. inverse(of-nat n) \) ----> (0 ::'a::real-normed-field))
sequentially
using lim-1-over-n by (simp add: inverse-eq-divide)

lemma lim-inverse-n':
  shows ((\( \lambda n. 1 / n \) ----> 0) sequentially
using lim-inverse-n
by (simp add: inverse-eq-divide)

lemma LIMSEQ-Suc-n-over-n:
  shows ((\( \lambda n. of-nat(Suc n) / of-nat n ::'a::real-normed-field \) ----> 1)
proof (rule Lim-transform-eventually)
  show eventually (λn. 1 + inverse (of-nat n :: 'a) = of-nat (Suc n) / of-nat n)
  sequentially
    using eventually-gt-at-top[of 0::nat]
    by eventually-elim (simp add: field-simps)
  have (λn. 1 + inverse (of-nat n :: 'a) → 1 + 0
    by (intro tendsto-add tendsto-const lim-inverse-n)
  then show (λn. 1 + inverse (of-nat n :: 'a) → 1
    by simp
  qed

lemma LIMSEQ-n-over-Suc-n: (λn. of-nat n / of-nat (Suc n) :: 'a :: real-normed-field)
  → 1
proof (rule Lim-transform-eventually)
  show eventually (λn. inverse (of-nat (Suc n) / of-nat n :: 'a) =
  of-nat n / of-nat (Suc n)) sequentially
    using eventually-gt-at-top[of 0::nat]
    by eventually-elim (simp add: field-simps del: of-nat-Suc)
  have (λn. inverse (of-nat (Suc n) / of-nat n :: 'a) → inverse 1
    by (intro tendsto-inverse LIMSEQ-Suc-n-over-n) simp-all
  then show (λn. inverse (of-nat (Suc n) / of-nat n :: 'a) → 1
    by simp
  qed

106.7 Convergence on sequences

lemma convergent-cong:
  assumes eventually (λx. f x = g x) sequentially
  shows convergent f ↔ convergent g
  unfolding convergent-def
  by (subst filterlim-cong[OF refl refl assms]) (rule refl)

lemma convergent-Suc-iff: convergent (λn. f (Suc n)) ↔ convergent f
  by (auto simp: convergent-def LIMSEQ-Suc-iff)

lemma convergent-ignore-initial-segment: convergent (λn. f (n + m)) = convergent f
proof (induct m arbitrary: f)
  case 0
  then show ?case by simp
next
  case (Suc m)
  have convergent (λn. f (n + Suc m)) ↔ convergent (λn. f (Suc n + m))
    by simp
  also have ... ↔ convergent (λn. f (n + m))
    by (rule convergent-Suc-iff)
  also have ... ↔ convergent f
    by (rule Suc)
  finally show ?case.
theory "Limits"

lemma convergent-add:
  fixes X Y :: 'a::{topological-monoid-add}
  assumes convergent (λn. X n)
  and convergent (λn. Y n)
  shows convergent (λn. X n + Y n)
  using assms unfolding convergent-def by (blast intro: tendsto-add)

lemma convergent-sum:
  fixes X :: 'b::{topological-comm-monoid-add}
  shows (∀i. i ∈ A → convergent (λn. X i n)) → convergent (λn. ∑ i∈A. X i n)
  by (induct A rule: infinite-finite-induct) (simp-all add: convergent-const convergent-add)

lemma (in bounded-linear) convergent:
  assumes convergent (λn. X n)
  shows convergent (λn. f (X n))
  using assms unfolding convergent-def by (blast intro: tendsto)

lemma (in bounded-bilinear) convergent:
  assumes convergent (λn. X n)
  and convergent (λn. Y n)
  shows convergent (λn. X n ** Y n)
  using assms unfolding convergent-def by (blast intro: tendsto)

lemma convergent-minus_iff:
  fixes X :: 'a::{topological-group-add}
  shows convergent X ←→ convergent (λn. − X n)
  unfolding convergent-def by (force dest: tendsto-minus)

lemma convergent-diff:
  fixes X Y :: 'a::{topological-group-add}
  assumes convergent (λn. X n)
  assumes convergent (λn. Y n)
  shows convergent (λn. X n − Y n)
  using assms unfolding convergent-def by (blast intro: tendsto-diff)

lemma convergent-norm:
  assumes convergent f
  shows convergent (λn. norm (f n))
proof –
  from assms have f --> lim f
      by (simp add: convergent-LIMSEQ_iff)
  then have (λn. norm (f n)) --> norm (lim f)
      by (rule tendsto-norm)
  then show ?thesis
      by (auto simp: convergent-def)
qed
lemma convergent-of-real:
  convergent f \implies \text{convergent} (\lambda n. \text{of-real} (f \ n) :: 'a::real-normed-algebra-1)
unfolding convergent-def by (blast intro: tendsto-of-real)

lemma convergent-add-const-iff:
  convergent (\lambda n. c + f \ n :: 'a::topological-ab-group-add) \iff \text{convergent} f
proof
  assume convergent (\lambda n. c + f \ n)
  from convergent-diff[OF this convergent-const[of c]] show convergent f
  by simp
next
  assume convergent f
  from convergent-add[OF convergent-const[of c] this] show convergent (\lambda n. c + f \ n)
  by simp
qed

lemma convergent-add-const-right-iff:
  convergent (\lambda n. f \ n + c :: 'a::topological-ab-group-add) \iff \text{convergent} f
using convergent-add-const-iff[of c f] by (simp add: add-ac)

lemma convergent-diff-const-right-iff:
  convergent (\lambda n. f \ n - c :: 'a::topological-ab-group-add) \iff \text{convergent} f
using convergent-add-const-right-iff[of f - c] by (simp add: add-ac)

lemma convergent-mult:
  fixes X Y :: nat \Rightarrow 'a::topological-semigroup-mult
  assumes convergent (\lambda n. X \ n)
  and convergent (\lambda n. Y \ n)
  shows convergent (\lambda n. X \ n * Y \ n)
  unfolding convergent-def by (blast intro: tendsto-mult)

lemma convergent-mult-const-iff:
  assumes c \neq 0
  shows convergent (\lambda n. c * f \ n :: 'a::{field,topological-semigroup-mult}) \iff \text{convergent} f
proof
  assume convergent (\lambda n. c * f \ n)
  from assms convergent-mult[OF this convergent-const[of inverse c]]
  show convergent f by (simp add: field-simps)
next
  assume convergent f
  from convergent-mult[OF convergent-const[of c] this] show convergent (\lambda n. c * f \ n)
  by simp
qed

lemma convergent-mult-const-right-iff:
fixes \( c \) :: 'a::{field, topological-semigroup-mult}
assumes \( c \neq 0 \)
shows convergent \((\lambda n. f \circ n \cdot c) \iff \) convergent \( f \)
using convergent-mult-const-iff[OF assms, of \( f \)] by (simp add: mult-ac)

lemma convergent-imp-Bseq: convergent \( f \) \implies Bseq \( f \)
by (simp add: Cauchy-Bseq convergent-Cauchy)

A monotone sequence converges to its least upper bound.

lemma LIMSEQ-incseq-SUP:
fixes \( X \) :: nat \Rightarrow 'a::{conditionally-complete-linorder, linorder-topology}
assumes \( u \): bdd-above (range \( X \))
and \( X \): incseq \( X \)
shows \( X \longrightarrow (\sup i. X \circ i) \)
by (rule order-tendstoI)
(auto simp: eventually-sequentially u less-cSUP-iff
intro: \( X[\THEN incseqD \ THEN \ le-less-trans \ less-cSUP-D \ OF \ u] \))

lemma LIMSEQ-decseq-INF:
fixes \( X \) :: nat \Rightarrow 'a::{conditionally-complete-linorder, linorder-topology}
assumes \( u \): bdd-below (range \( X \))
and \( X \): decseq \( X \)
shows \( X \longrightarrow (\inf i. X \circ i) \)
by (rule order-tendstoI)
(auto simp: eventually-sequentially u cINF-less-iff
intro: \( X[\THEN decseqD \ THEN \ le-less-trans \ less-cINF-D \ OF \ u] \))

Main monotonicity theorem.

lemma Bseq-monoseq-convergent: Bseq \( X \) \implies monoseq \( X \) \iff convergent \( X \)
for \( X \) :: nat \Rightarrow real
by (auto simp: monoseq-iff convergent-def intro: LIMSEQ-decseq-INF LIMSEQ-incseq-SUP
dest: Bseq-bdd-above Bseq-bdd-below)

lemma Bseq-mono-convergent: Bseq \( X \) \implies (\forall m. m \leq n \rightarrow X \circ m \leq X \circ n) \implies
convergent \( X \)
for \( X \) :: nat \Rightarrow real
by (auto intro!: Bseq-monoseq-convergent incseq-imp-monoseq simp: incseq-def)

lemma monoseq-imp-convergent-iff-Bseq: monoseq \( f \) \implies convergent \( f \) \iff Bseq \( f \)
for \( f \) :: nat \Rightarrow real
using Bseq-monoseq-convergent[of \( f \)] convergent-imp-Bseq[of \( f \)] by blast

lemma Bseq-monoseq-convergent'-inc:
fixes \( f \) :: nat \Rightarrow real
shows Bseq \((\lambda n. f \circ (n + M)) \implies (\forall m. m \leq n \implies m \leq n \implies f \circ m \leq f \circ n) \implies
convergent \( f \)
by (subst convergent-ignore-initial-segment [symmetric, of - \( M \)])
(auto intro!: Bseq-monoseq-convergent simp: monoseq-def)
THEORY "Limits"

lemma Bseq-monoseq-convergent':dec:
  fixes f :: nat ⇒ real
  shows Bseq (λn. f (n + M)) → (∀m n. M ≤ m → m ≤ n → f m ≥ f n)
  → convergent f
  by (subst convergent-ignore-initial-segment [symmetric, of - M])
      (auto intro!: Bseq-monoseq-convergent simp: monoseq-def)

lemma Cauchy-iff: Cauchy X ⇐⇒ (∀e>0. ∃M. ∀m≥M. ∀n≥M. norm (X m - X n) < e)
  for X :: nat ⇒ 'a::real-normed-vector
  unfolding Cauchy-def dist-norm ..

lemma CauchyI: (∀e. 0 < e → ∃M. ∀m≥M. ∀n≥M. norm (X m - X n) < e) → Cauchy X
  for X :: nat ⇒ 'a::real-normed-vector
  by (simp add: Cauchy-iff)

lemma CauchyD: Cauchy X → 0 < e → ∃M. ∀m≥M. ∀n≥M. norm (X m - X n) < e
  for X :: nat ⇒ 'a::real-normed-vector
  by (simp add: Cauchy-iff)

lemma incseq-convergent:
  fixes X :: nat ⇒ real
  assumes incseq X and ∀i. X i ≤ B
  obtains L where X −−−−→ L ∀i. X i ≤ L
  proof atomize-elim
    from incseq-bounded[OF assms] ⟨incseq X⟩ Bseq-monoseq-convergent[of X]
    obtain L where X −−−−→ L
      by (auto simp: convergent-def monoseq-def incseq-def)
    with ⟨incseq X⟩ show ∃L. X −−−−→ L ∧ (∀i. X i ≤ L)
      by (auto intro!: exI [of - L] incseq-le)
  qed

lemma decseq-convergent:
  fixes X :: nat ⇒ real
  assumes decseq X and ∀i. B ≤ X i
  obtains L where X −−−−→ L ∀i. L ≤ X i
  proof atomize-elim
    from decseq-bounded[OF assms] ⟨decseq X⟩ Bseq-monoseq-convergent[of X]
    obtain L where X −−−−→ L
      by (auto simp: convergent-def monoseq-def decseq-def)
    with ⟨decseq X⟩ show ∃L. X −−−−→ L ∧ (∀i. L ≤ X i)
      by (auto intro!: exI [of - L] decseq-ge)
  qed

lemma monoseq-convergent:
fixes $X :: \text{nat} \Rightarrow \text{real}$
assumes $X$: monoseq $X$ and $B$: $\forall i. |X i| \leq B$
obtains $L$ where $X \longrightarrow L$
using $X$ unfolding monoseq-iff
proof
assume incseq $X$
show thesis
  using abs-le-D1 [OF $B$] incseq-convergent [OF $\langle \text{incseq } X \rangle$] that by meson
next
assume decseq $X$
show thesis
  using decseq-convergent [OF $\langle \text{decseq } X \rangle$] that
  by (metis $B$ abs-le-iff add.inverse-inverse neg-le-iff-le)
qed

106.8 Power Sequences

lemma Bseq-realpow: $0 \leq x 
\Rightarrow x \leq 1 \Rightarrow Bseq (\lambda n. x ^ n)$
for $x :: \text{real}$
by (metis decseq-bounded decseq-def power-decreasing zero-le-power)

lemma monoseq-realpow: $0 \leq x 
\Rightarrow x \leq 1 \Rightarrow monoseq (\lambda n. x ^ n)$
for $x :: \text{real}$
using monoseq-def power-decreasing by blast

lemma convergent-realpow: $0 \leq x 
\Rightarrow x \leq 1 \Rightarrow convergent (\lambda n. x ^ n)$
for $x :: \text{real}$
by (blast intro!: Bseq-monoseq-convergent Bseq-realpow monoseq-realpow)

lemma LIMSEQ-inverse-realpow-zero: $1 < x 
\Rightarrow (\lambda n. \text{inverse } (x ^ n)) \longrightarrow 0$
for $x :: \text{real}$
by (rule filterlim-compose[OF tendsto-inverse-0 filterlim-realpow-sequentially-gt1]) simp

lemma LIMSEQ-realpow-zero:
  fixes $x :: \text{real}$
  assumes $0 \leq x < 1$
  shows $(\lambda n. x ^ n) \longrightarrow 0$
proof (cases $x = 0$)
  case False
  with $(0 \leq x)$ have $x0: 0 < x$ by simp
  then have $1 < \text{inverse } x$
    using $(x < 1)$ by (rule one-less-inverse)
  then have $(\lambda n. \text{inverse } (\text{inverse } x ^ n)) \longrightarrow 0$
    by (rule LIMSEQ-inverse-realpow-zero)
  then show ?thesis by (simp add: power-inverse)
next
  case True
  show ?thesis
by (rule LIMSEQ-imp-Suc) (simp add: True)

qed

lemma LIMSEQ-power-zero [tendsto-intros]: \( \text{norm } x < 1 \implies (\lambda n. x \cdot n) \longrightarrow 0 \)
for \( x :: 'a::real-normed-algebra-1 \)
apply (drule LIMSEQ-realpow-zero [OF norm-ge-zero])
by (simp add: Zfun-le norm-power-ineq tendsto-Zfun-iff)

lemma LIMSEQ-divide-realpow-zero: \( 1 < x \implies (\lambda n. a / (x \cdot n) :: real) \longrightarrow 0 \)
by (rule tendsto-divide-0 [OF tendsto-const filterlim-realpow-sequentially-gt1])

simp

lemma tendsto-power-zero:
fixes \( x :: 'a::real-normed-algebra-1 \)
assumes \( \text{filterlim } f \text{ at-top } F \)
assumes \( \text{norm } x < 1 \)
shows \( (\lambda y. x ^ (f y)) \longrightarrow 0 ) F \)
proof (rule tendstoI)
fix \( e :: real \)
assume \( 0 < e \)
from tendstoD [OF LIMSEQ-power-zero [OF \langle \text{norm } x < 1 \rangle \langle 0 < e \rangle ]]
have \( \forall F \cdot \text{ eventually-sequentially } (\text{norm } (x \cdot xa) < e) \)
by (auto simp: eventually-sequentially)
then obtain \( N \) where \( \text{norm } (x \cdot n) < e \) if \( n \geq N \) for \( n \)
by (auto simp: N)
then show \( \forall F \cdot \text{ eventually-elim } (\text{auto simp: N}) \)
qed

Limit of \( c^n \) for \( |c| < (1 :: 'a) \).

lemma LIMSEQ-abs-realpow-zero: \( |c| < 1 \implies (\lambda n. |c| \cdot n :: real) \longrightarrow 0 \)
by (rule LIMSEQ-realpow-zero [OF abs-ge-zero])

lemma LIMSEQ-abs-realpow-zero2: \( |c| < 1 \implies (\lambda n. c \cdot n :: real) \longrightarrow 0 \)
by (rule LIMSEQ-power-zero) simp

106.9 Limits of Functions

lemma LIM-eq: \( f \cdot -a \rightarrow L = (\forall r>0. \exists s>0. \forall x. x \neq a \wedge \text{norm } (x - a) < s \implies \text{norm } (f \cdot (x - L)) < r) \)
for \( a :: 'a::real-normed-vector \) and \( L :: 'b::real-normed-vector \)
by (simp add: LIM-def dist-norm)
\( \forall r. \ 0 < r \implies \exists s>0. \ \forall x. \ x \neq a \land \text{norm } (x - a) < s \implies \text{norm } (f x - L) < r \) \implies \( f \to a \to L \)

for \( a \) :: \'a::real-normed-vector and \( L \) :: \'b::real-normed-vector
by (simp add: LIM-eq)

**Lemma LIM-D:** \( f \to a \to L \implies 0 < r \implies \exists s>0. \ \forall x. \ x \neq a \land \text{norm } (x - a) < s \implies \text{norm } (f x - L) < r \)

for \( a \) :: \'a::real-normed-vector and \( L \) :: \'b::real-normed-vector
by (simp add: LIM-eq)

**Lemma LIM-offset:** \( f \to a \to L \implies (\lambda x. \ f (x + k)) - (a - k) \to L \)

for \( a \) :: \'a::real-normed-vector
by (simp add: filtermap-at-shift[symmetric, of a k] filterlim_def filtermap_filtermap)

**Lemma LIM-offset-zero:** \( f \to a \to L \implies (\lambda h. \ f (a + h)) - 0 \to L \)

for \( a \) :: \'a::real-normed-vector
by (drule LIM-offset [where \( k = a \)]) (simp add: add.commute)

**Lemma LIM-offset-zero-cancel:** \( (\lambda h. \ f (a + h)) - 0 \to L \implies f \to a \to L \)

for \( a \) :: \'a::real-normed-vector
by (drule LIM-offset [where \( k = - a \)]) simp

**Lemma tendsto-offset-zero-iff:**
fixes \( f \) :: \'a::real-normed-vector \Rightarrow -
assumes \( a \in S \) open \( S \)
shows \( (f \to L) \ (a \text{ within } S) \iff ((\lambda h. \ f (a + h)) - 0 \to L) \ (a \text{ at } 0) \)
by (metis LIM-offset-zero-iff assms tendsto-within-open)

**Lemma LIM-zero:** \( (f \to l) \ F \implies ((\lambda x. \ f x - l) \to 0) \ F \)

for \( f \) :: \'a \Rightarrow \'b::real-normed-vector
 unfolding tendsto-iff dist-norm by simp

**Lemma LIM-zero-cancel:**
fixes \( f \) :: \'a::topological-space \Rightarrow \'b::real-normed-vector
shows \( ((\lambda x. \ f x - l) \to 0) \ F \implies (f \to l) \ F \)
 unfolding tendsto-iff dist-norm by simp

**Lemma LIM-zero-iff:** \( ((\lambda x. \ f x - l) \to 0) \ F = (f \to l) \ F \)

for \( f \) :: \'a \Rightarrow \'b::real-normed-vector
 unfolding tendsto-iff dist-norm by simp

**Lemma LIM-imp-LIM:**
fixes \( f \) :: \'a::topological-space \Rightarrow \'b::real-normed-vector
fixes \( g \) :: \'a::topological-space \Rightarrow \'c::real-normed-vector
assumes \( f : f \to a \to l \)
and le: $\forall x. x \neq a \Rightarrow \text{norm} (g x - m) \leq \text{norm} (f x - l)$
shows $g \to_a m$
by (rule metric-LIM-imp-LIM [OF f]) (simp add: dist-norm le)

**lemma LIM-equal2:**
fixes $f g :: 'a::real-normed-vector \Rightarrow 'b::topological-space$
assumes $0 < R$
and $\forall x. x \neq a \Rightarrow \text{norm} (x - a) < R \Rightarrow f x = g x$
shows $g \to_a l \Rightarrow f \to_a l$
by (rule metric-LIM-equal2 [OF - assms]) (simp-all add: dist-norm)

**lemma LIM-compose2:**
fixes $a :: 'a::real-normed-vector$
assumes $f :: f \to_a b$
and $g :: g \to_b c$
and inj: $\exists d > 0. \forall x. x \neq a \wedge \text{norm} (x - a) < d \longrightarrow f x \neq b$
shows $(\lambda x. g (f x)) \to c$
by (rule metric-LIM-compose2 [OF f g inj [folded dist-norm]])

**lemma real-LIM-sandwich-zero:**
fixes $f g :: 'a::topological-space \Rightarrow \text{real}$
assumes $f :: f \to_a 0$
and $1 :: \forall x. x \neq a \Rightarrow 0 \leq g x$
and $2 :: \forall x. x \neq a \Rightarrow g x \leq f x$
shows $g \to_a 0$
proof (rule LIM-imp-LIM [OF f])
fix $x$
assume $x :: x \neq a$
with $1$ have $\text{norm} (g x - 0) = g x \text{ by simp}$
also have $g x \leq f x \text{ by (rule 2 [OF x])}$
also have $f x \leq |f x| \text{ by (rule abs-ge-self)}$
also have $|f x| = \text{norm} (f x - 0) \text{ by simp}$
finally show $\text{norm} (g x - 0) \leq \text{norm} (f x - 0)$ .
qed

### 106.10 Continuity

**lemma LIM-isCont-iff:** $(f \to_a f a) = ((\lambda h. f (a + h)) -0 \to f a)$
for $f :: 'a::real-normed-vector \Rightarrow 'b::topological-space$
by (rule iffI [OF LIM-offset-zero LIM-offset-zero-cancel])

**lemma isCont-iff:** $\text{isCont} f x = (\lambda h. f (x + h)) -0 \to f x$
for $f :: 'a::real-normed-vector \Rightarrow 'b::topological-space$
by (simp add: isCont-def LIM-isCont-iff)

**lemma isCont-LIM-compose2:**
fixes $a :: 'a::real-normed-vector$
assumes $f :: [unfolded isCont-def] :: isCont f a$
and $g :: g \to_f a$

THEORY "Limits"

and inj: ∃d>0. ∀x. x ≠ a ∧ norm (x - a) < d → f x ≠ f a

shows (λx. g (f x)) a → l

by (rule LIM-compose2 [OF f g inj])

lemma isCont-norm [simp]: isCont f a → isCont (λx. norm (f x)) a

for f :: 'a::t2-space ⇒ 'b::real-normed-vector

by (fact continuous-norm)

lemma isCont-rabs [simp]: isCont f a → isCont (λx. |f x|) a

for f :: 'a::t2-space ⇒ real

by (fact continuous-rabs)

lemma isCont-add [simp]: isCont f a → isCont g a → isCont (λx. f x + g x) a

for f :: 'a::t2-space ⇒ 'b::topological-monoid-add

by (fact continuous-add)

lemma isCont-minus [simp]: isCont f a → isCont (λx. - f x) a

for f :: 'a::t2-space ⇒ 'b::real-normed-vector

by (fact continuous-minus)

lemma isCont-diff [simp]: isCont f a → isCont g a → isCont (λx. f x - g x) a

for f :: 'a::t2-space ⇒ 'b::real-normed-vector

by (fact continuous-diff)

lemma isCont-mult [simp]: isCont f a → isCont g a → isCont (λx. f x * g x) a

for f g :: 'a::t2-space ⇒ 'b::real-normed-algebra

by (fact continuous-mult)

lemma (in bounded-linear) isCont: isCont g a → isCont (λx. f (g x)) a

by (fact continuous)

lemma (in bounded-bilinear) isCont: isCont f a → isCont g a → isCont (λx. f x ** g x) a

by (fact continuous)

lemmas isCont-scaleR [simp] = bounded-bilinear.isCont [OF bounded-bilinear-scaleR]

lemmas isCont-of-real [simp] = bounded-linear.isCont [OF bounded-linear-of-real]

lemma isCont-power [simp]: isCont f a → isCont (λx. f x ^ n) a

for f :: 'a::t2-space ⇒ 'b::{power,real-normed-algebra}

by (fact continuous-power)

lemma isCont-sum [simp]: ∀i∈A. isCont (f i) a → isCont (λx. ∑i∈A. f i x) a
lemma uniformly-continuous-on-def:
  fixes f :: 'a::metric-space ⇒ 'b::metric-space
  shows uniformly-continuous-on UNIV f

abbreviation isUCont :: ['a::metric-space ⇒ 'b::metric-space] ⇒ bool
  where isUCont f ≡ uniformly-continuous-on UNIV f

lemma isUCont-def: isUCont f ≡ (∀ r > 0. ∃ s > 0. ∀ x y. dist x y < s → dist (f x) (f y) < r)
  by (auto simp: uniformly-continuous-on-def dist-commute)

lemma isUCont-isCont: isUCont f =⇒ isCont f x
  by (drule uniformly-continuous-imp-continuous) (simp add: continuous-on-eq-continuous-at)

lemma uniformly-continuous-on-Cauchy:
  fixes f :: 'a::metric-space ⇒ 'b::metric-space
  assumes uniformly-continuous-on S f Cauchy X \∀ n. X n ∈ S
  shows Cauchy (λn. f (X n))
  using assms
  unfolding uniformly-continuous-on-def
  by (meson Cauchy-def)

lemma isUCont-Cauchy: isUCont f =⇒ Cauchy X =⇒ Cauchy (λn. f (X n))
  by (rule uniformly-continuous-on-Cauchy[where S=UNIV and f=f]) simp-all

lemma uniformly-continuous-imp-Cauchy-continuous:
  fixes f :: 'a::metric-space ⇒ 'b::metric-space
  shows [uniformly-continuous-on S f; Cauchy σ; \∀ n. (σ n) ∈ S] =⇒ Cauchy(f o σ)
  by (simp add: uniformly-continuous-on-Cauchy-def) meson

lemma (in bounded-linear) isUCont: isUCont f
  unfolding isUCont-def
  proof (intro allI impI)
    fix r :: real
    assume r < r
    obtain K where K: 0 < K and norm-le: norm (f x) ≤ norm x * K for x
      using pos-bounded by blast
    show ∃ s > 0. ∀ x y. norm (x - y) < s → norm (f x - f y) < r
      proof (rule exI, safe)
        from r K show 0 < r / K by simp
next
  fix \( x \, y :: 'a \)
  assume \( xy: \text{norm} (x - y) < r / K \)
  have \( \text{norm} (f x - f y) = \text{norm} (f (x - y)) \) by (simp only: diff)
  also have \( \ldots \leq \text{norm} (x - y) * K \) by (rule norm-le)
  also from \( K \, xy \) have \( \ldots < r \) by (simp only: pos-less-divide-eq)
  finally show \( \text{norm} (f x - f y) < r \).
qed

lemma \( \text{(in bounded-linear) Cauchy}: \text{Cauchy} X \implies \text{Cauchy} (\lambda n. f (X \, n)) \)
by (rule isUCont [THEN isUCont-Cauchy])

lemma \( \text{LIM-less-bound}: \)
  fixes \( f :: \text{real} \Rightarrow \text{real} \)
  assumes \( \text{ev}: b < x \forall x' \in \{b <..< x\}, 0 \leq f \, x' \) and \( \text{isCont} \, f \, x \)
  shows \( 0 \leq f \, x \)
proof (rule tendsto-lowerbound)
  show \( (f \longrightarrow f \, x) \) (at-left \( x \))
    using \( \text{isCont} \, f \, x \)
      by (simp add: filterlim-at-split isCont-def)
  show eventually \( (x, 0 \leq f \, x) \) (at-left \( x \))
    using \( \text{ev} \)
      by (auto simp: eventually-at dist-real-def intro: exI [of - x - b])
qed simp

106.12 Nested Intervals and Bisection – Needed forCompactness

lemma \( \text{nested-sequence-unique}: \)
  assumes \( \forall n. f \, n \leq f \, (\text{Suc} \, n) \forall n. g \, (\text{Suc} \, n) \leq g \, n \forall n. f \, n \leq g \, n \lambda n. f \, n - g \, n \) \longrightarrow \( 0 \)
  shows \( \exists l::\text{real}. ((\forall n. f \, n \leq l) \land f \longrightarrow l) \land ((\forall n. l \leq g \, n) \land g \longrightarrow l) \)
proof –
  have \( \text{incseq} \, f \) unfolding \( \text{incseq-Suc-iff} \)
    by fact
  have \( \text{decseq} \, g \) unfolding \( \text{decseq-Suc-iff} \)
    by fact
  have \( f \, n \leq g \, 0 \) for \( n \)
proof –
  from \( \text{decseq} \, g \)
    have \( g \, n \leq g \, 0 \)
      by (rule decseqD) simp
  with \( \forall n. f \, n \leq g \, n[:\text{THEN spec, of} \, n] \) show \( \text{?thesis} \)
    by auto
qed
then obtain \( u \) where \( f \longrightarrow u \forall i. f \, i \leq u \)
  using \( \text{incseq-convergent}[:\text{OF} \, (\text{incseq} \, f)] \)
by auto
moreover have \( f \, 0 \leq g \, n \) for \( n \)
proof –
  from \( \text{incseq} \, f \)
    have \( f \, 0 \leq f \, n \) by (rule incseqD) simp
  with \( \forall n. f \, n \leq g \, n[:\text{THEN spec, of} \, n] \) show \( \text{?thesis} \)
    by simp
qed
then obtain \( l \) where \( g \longrightarrow l \forall i. l \leq g i \)

using \( \text{decseq-convergent}(\text{OF } \text{decseq } g) \) by auto

moreover note \( \text{LIMSEQ-unique}(\text{OF } \text{assms}(4)) \) tendsto-diff[\( \text{OF } f \longrightarrow w : g \longrightarrow l \)]

ultimately show ?thesis by auto

dqed

lemma Bolzano[consumes 1, case-names trans local]:

fixes \( P :: \text{real} \Rightarrow \text{real} \Rightarrow \text{bool} \)

assumes [arith]: \( a \leq b \)

and trans: \( \forall a b c. P a b \Longrightarrow P b c \Longrightarrow a \leq b \leq c \Longrightarrow P a c \)

and local: \( \forall x. a \leq x \Longrightarrow x \leq b \Longrightarrow \exists d>0. \forall a b. a \leq x \wedge x \leq b \wedge b-a < d \longrightarrow P a b \)

shows \( P a b \)

proof –

define \( \text{bisect} \) where \( \text{bisect} = \)

\( \text{rec-nat } (a, b) \text{ (} \lambda n. (x, y) \text{ if } P x ((x+y) / 2) \text{ then } ((x+y)/2, y) \text{ else } (x, (x+y)/2)) \)

define \( l u \) where \( l n = \text{fst } (\text{bisect } n) \) and \( u n = \text{snd } (\text{bisect } n) \) for \( n \)

have \( \text{lsimp; } l 0 = a \wedge l (\text{Suc } n) = (\text{if } P (l n) (((l n + u n) / 2) \text{ then } (l n + u n) / 2 \text{ else } l n)) \)

and \( \text{usimp; } u 0 = b \wedge u (\text{Suc } n) = (\text{if } P (l n) (((l n + u n) / 2) \text{ then } u n \text{ else } (l n + u n) / 2)) \)

by (simp-all add: l-def u-def bisect-def split: prod.split)

have \( \text{lsimp; } l n \leq u n \) for \( n \) by (induct \( n \)) auto

have \( \exists x. ((\forall n. l n \leq x) \wedge l \longrightarrow x) \wedge ((\forall n. x \leq u n) \wedge u \longrightarrow x) \)

proof (safe intro!; nested-sequence-unique)

show \( l n \leq l (\text{Suc } n) u (\text{Suc } n) \leq u n \) for \( n \)

by (induct \( n \)) auto

next

have \( l n - u n = (a - b) / 2^n \) for \( n \)

by (induct \( n \)) (auto simp: field-simps)

then show \( \lambda n. l n - u n \longrightarrow 0 \)

by (simp add: LIMSEQ-divide-realpow-zero)

dfact

then obtain \( x \) where \( x: \forall n. l n \leq x \wedge n. x \leq u n \) and \( l \longrightarrow x u \longrightarrow x \)

by auto

obtain \( d \) where \( 0 < d \) and \( d: a \leq x \Longrightarrow x \leq b \Longrightarrow b-a < d \Longrightarrow P a b \) for \( a b \)

using \( (d 0 \leq x) \) \( (x \leq u 0) \) local[of \( x \)] by auto

show \( P a b \)

proof (rule ccontr)

assume \( \neg P a b \)

have \( \neg P (l n) (u n) \) for \( n \)

proof (induct \( n \))

case \( 0 \)

show \( P a b \)

proof (rule ccontr)

assume \( \neg P a b \)

have \( \neg P (l n) (u n) \) for \( n \)

proof (induct \( n \))

case \( 0 \)
then show \(\neg P\ a\ b\)
  by (simp add: \(\neg P\ a\ b\))
next
  case (Suc \(n\))
  with \(\text{trans[of}\ l\ n\ (l\ n + u\ n)/2\ u\ n]\) show \(\neg P\ a\ b\)
    by auto
qed
moreover

\begin{itemize}
  \item have eventually \((\lambda n. x - d / 2 < l\ n)\) sequentially
    using \((\theta < d; \theta \longrightarrow x)\) by (intro order-tendstoD[of - x]) auto
  \item moreover have eventually \((\lambda n. u\ n < x + d / 2)\) sequentially
    using \((\theta < d; \theta \longrightarrow x)\) by (intro order-tendstoD[of - x]) auto
  \item ultimately have eventually \((\lambda n. P (l\ n) (u\ n))\) sequentially
\end{itemize}

proof eventually-elim
  case (elim \(n\))
  from \(\text{add-strict-mono[of this]}\) have \(u\ n - l\ n < d\) by simp
  with \(x\) show \(P (l\ n) (u\ n)\) by (rule \(d\))
qed

ultimately show False by simp
qed

lemma \(\text{compact-Icc(simp, intro): compact \{a .. b::real\}}\)
proof (cases \(a \leq b\), rule \(\text{compactI}\))
  fix \(C\)
  assume \(C\): \(a \leq b\) \(\forall t \in C.\ \text{open }\{a .. b\} \subseteq \bigcup C\)
  define \(T\) where \(T = \{a .. b\}\)
  from \(\text{C(1,3) show }\exists C' \subseteq C.\ \text{finite }C' \land \{a .. b\} \subseteq \bigcup C'\)
  proof (induct rule: Bolzano)
    case (trans \(a\ b\ c\))
    then have \(*: \{a .. c\} = \{a .. b\} \cup \{b .. c\}\)
      by auto
    with \(\text{trans obtain}\ C1\ C2\)
      where \(C1 \subseteq C\ \text{finite }C1\ \{a .. b\} \subseteq \bigcup C1\ C2 \subseteq C\ \text{finite }C2\ \{b .. c\} \subseteq \bigcup C2\)
      by auto
    with \(\text{trans show }\neg P\ a\ b\ c\)
      unfolding \(*\) by (intro exI[of - \(C1 \cup C2\)]) auto
  next
    case (local \(x\))
    with \(C\) have \(x \in \bigcup C\) by auto
    with \(\text{C(2) obtain } e\ where \ x \in e \ \text{open } e \ c \in C\)
      by auto
    then obtain \(e\) where \(\theta < e\ \{x - e <..< x + e\} \subseteq e\)
      by (auto simp: open-dist dist-real-def subset-eq Ball-def abs-less-iff)
    with \(\text{c \in C}\) show \(\neg P\)
      by (safe intro!: exI[of - \(e/2\)] exI[of - \{c\}]) auto
  qed
**THEORY “Limits”**

**lemma** continuous-image-closed-interval:

fixes \( a, b \) and \( f : \text{real} \Rightarrow \text{real} \)
defines \( S \equiv \{a..b\} \)
assumes \( a \leq b \) and \( f: \text{continuous-on } S f \)
shows \( \exists c \ d. f'S = \{c..d\} \land c \leq d \)

**proof**

have \( S: \text{compact } S \neq \{\} \)
  using \((a \leq b)\) by \((\text{auto simp: S-def})\)
obtain \( c \) where \( c \in S \land \forall d \in S. f d \leq f c \)
  using continuous-attains-sup \([OF S f]\) by \(\text{auto}\)
moreover obtain \( d \) where \( d \in S \land \forall c \in S. f d \leq f c \)
  using continuous-attains-inf \([OF S f]\) by \(\text{auto}\)
moreover have \( \text{connected } (f'S) \)
  using connected-continuous-image \([OF f]\) connected-Icc by \(\text{(auto simp: S-def)}\)
ultimately have \( f' S = \{f d .. f c\} \land f d \leq f c \)
  by \(\text{(auto simp: connected-iff-interval)}\)
then show \( \text{thesis} \)
  by \(\text{auto}\)

qed

**lemma** open-Collect-positive:

fixes \( f : \text{topological-space} \Rightarrow \text{real} \)
assumes \( f: \text{continuous-on } s f \)
shows \( \exists A. \text{open } A \land A \cap s = \{x \in s. 0 < f x\} \)
  using continuous-on-open-invariant \([\text{THEN iffD1, OF f, rule-format, of } \{0 <\}]\)
  by \(\text{(auto simp: Int-def field-simps)}\)

**lemma** open-Collect-less-Int:

fixes \( f, g : \text{topological-space} \Rightarrow \text{real} \)
and \( g: \text{continuous-on } s g \)
shows \( \exists A. \text{open } A \land A \cap s = \{x \in s. f x < g x\} \)
  using open-Collect-positive \([\text{OF continuous-on-diff}[OF g f]]\) by \(\text{(simp add: field-simps)}\)

106.13 Boundedness of continuous functions

By bisection, function continuous on closed interval is bounded above

**lemma** isCont-eq-Ub:

fixes \( f : \text{real} \Rightarrow \text{linorder-topology} \)
shows \( a \leq b \implies \forall x: \text{real}. a \leq x \land x \leq b \implies \text{isCont } f x \implies \exists M. (\forall x. a \leq x \land x \leq b \implies f x \leq M) \land (\exists x. a \leq x \land x \leq b \land f x = M) \)
  using continuous-attains-sup \([\text{of } \{a..b\} f]\)
  by \(\text{(auto simp: continuous-at-imp-continuous-on Ball-def Bex-def)}\)

**lemma** isCont-eq-Lb:

fixes \( f : \text{real} \Rightarrow \text{linorder-topology} \)
shows $a \leq b \implies \forall x. a \leq x \land x \leq b \implies \text{isCont } f x$  

$\exists M. (\forall x. a \leq x \land x \leq b \implies M \leq f x) \land (\exists x. a \leq x \land x \leq b \land f x = M)$

using continuous-attains-inf[of \{a..b\} f]  
by (auto simp: continuous-at-imp-continuous-on Ball-def Bex-def)

lemma isCont-bounded:  
fixes $f :: \text{real} \Rightarrow 'a::linorder-topology$  
shows $a \leq b \implies \forall x. a \leq x \land x \leq b \implies \text{isCont } f x$  
$\exists M. (\forall x. a \leq x \land x \leq b \implies f x \leq M) \land (\forall N. N < M \implies (\exists x. a \leq x \land x \leq b \land f x = M))$  
using isCont-eq-Ub[of a b f] by auto

lemma IVT-objl:  
(f $a \leq y \land y \leq f b \land a \leq b \land (\forall x. a \leq x \land x \leq b \implies \text{isCont } f x))$  
$(\exists x. a \leq x \land x \leq b \land f x = y)$  
for $a$ $y :: \text{real}$  
by (blast intro: IVT)

lemma IVT2-objl:  
(f $b \leq y \land y \leq f a \land a \leq b \land (\forall x. a \leq x \land x \leq b \implies \text{isCont } f x))$  
$(\exists x. a \leq x \land x \leq b \land f x = y)$  
for $b$ $y :: \text{real}$  
by (blast intro: IVT2)

lemma isCont-Lb-Ub:  
fixes $f :: \text{real} \Rightarrow \text{real}$  
assumes $a \leq b \forall x. a \leq x \land x \leq b \implies \text{isCont } f x$  
shows $\exists L M. (\forall x. a \leq x \land x \leq b \implies L \leq f x \land f x \leq M) \land$  
$(\forall y. L \leq y \land y \leq M \implies (\exists x. a \leq x \land x \leq b \land (f x = y)))$

proof –  
obtain $M$ where $M :: a \leq M \land M \leq b$  
by (auto simp: isCont-eq-Ub[of assms])  

obtain $L$ where $L :: L \leq a$  
by (auto simp: isCont-eq-Lb[of assms])  

have $(\forall x. a \leq x \land x \leq b \implies f L \leq f x \land f x \leq f M)$  
by (auto simp: isCont-eq-Lb[of assms])  

moreover  
have $(\forall y. f L \leq y \land y \leq f M \implies (\exists x \geq a. x \leq b \land f x = y))$  
by (auto simp: isCont-eq-Ub[of assms])  

proof (cases $L \leq M$)  

  case True  
  then show ?thesis  
  using IVT[of f L - M]$\cdot$  
  M $\cdot$  
  N  
  assms  
  by (metis order.trans)  
next
case False then show ?thesis
  using IVT2[of f L - M]
  by (metis L(2) M(1) assms(2) le_cases order.trans)
qed
ultimately show ?thesis
  by blast
qed

Continuity of inverse function.

lemma isCont-inverse-function:
  fixes f g :: real ⇒ real
  assumes d: 0 < d
  and inj: ∀z. |z - x| ≤ d ⟹ g (f z) = z
  and cont: ∀z. |z - x| ≤ d ⟹ isCont f z
  shows isCont g (f x)
proof -
  let ?A = f (x - d)
  let ?B = f (x + d)
  let ?D = {x - d..x + d}

  have f: continuous-on ?D f
    using cont by (intro continuous-at-imp-continuous-on ballI) auto
  then have g: continuous-on (f'?D) g
    using inj by (intro continuous-on-inv) auto

  from d f have {min ?A ?B <..< max ?A ?B} ⊆ f : ?D
    by (intro connected-contains-Ioo connected-continuous-image) (auto split: split-min split-max)
  with g have continuous-on {min ?A ?B <..< max ?A ?B} g
    by (rule continuous-on-subset)
  moreover
  have (?A < f x ∧ f x < ?B) ∨ (?B < f x ∧ f x < ?A)
    using d inj by (intro continuous-inj-imp-mono[OF - - f inj-on-imageI2[of g, OF inj-onI]]) auto
  then have f x ∈ {min ?A ?B <..< max ?A ?B}
    by auto
  ultimately show ?thesis
    by (simp add: continuous-on-eq-continuous-at)
qed

lemma isCont-inverse-function2:
  fixes f g :: real ⇒ real
  shows "[a < x; x < b; \forall z. [a ≤ z; z ≤ b] ⟹ g (f z) = z; \forall z. [a ≤ z; z ≤ b] ⟹ isCont f z] ⟹ isCont g (f x)"
apply (rule isCont-inverse-function [where f=f and d=min (x - a) (b - x)])
apply (simp-all add: abs-le-iff)
done
lemma LIM-fun-gt-zero: f −→ l \implies 0 < l \implies \exists r. 0 < r \land (\forall x. x \neq c \land |c - x| < r \implies 0 < f x)
  for f :: real \Rightarrow real
  by (force simp: dest: LIM-D)

lemma LIM-fun-less-zero: f −→ l \implies l < 0 \implies \exists r. 0 < r \land (\forall x. x \neq c \land |c - x| < r \implies f x < 0)
  for f :: real \Rightarrow real
  by (drule LIM-D [where r=-l]) force

lemma LIM-fun-not-zero: f −→ l \implies l \neq 0 \implies \exists r. 0 < r \land (\forall x. x \neq c \land |c - x| < r \implies f x \neq 0)
  for f :: real \Rightarrow real
  using LIM-fun-gt-zero[of \ l \ c] LIM-fun-less-zero[of \ l \ c] by (auto simp: neq_iff)

end

theory Inequalities
  imports Real-Vector-Spaces
begin

lemma Chebyshev-sum-upper:
  fixes a b :: nat \Rightarrow 'a::{linordered_idom}
  assumes \(\forall i j. i \leq j \implies j < n \implies a i \leq a j\)
  assumes \(\forall i j. i \leq j \implies j < n \implies b i \geq b j\)
  shows \(\forall n \cdot (\sum k=0..<n. a k * b k) \leq (\sum j=0..<n. a k) * (\sum k=0..<n. b k)\)

proof -
  let \(?S = (\sum j=0..<n. (\sum k=0..<n. (a j - a k) * (b j - b k)))\)
  have \(2 * (\sum j=0..<n. (a j * b j)) - (\sum j=0..<n. b j) * (\sum k=0..<n. a k) = \(?S\)
    by (simp only: one_add_one[symmetric] algebra_simps)
    (simp add: algebra_simps sum_subtractf sum_distrib sum_swap[of \ l i j. a i * b j] sum_distrib_left)
  also
  \{ fix i j :: nat assume i<n j<n 
    hence a i - a j \leq 0 \land b i - b j \geq 0 \lor a i - a j \geq 0 \land b i - b j \leq 0 
    using assms by (cases i \leq j) (auto simp: algebra_simps) \}
  then have \(?S \leq 0\)
    by (auto intro!: sum_nonpos simp: mult_le_0_iff)
  finally show \(?thesis\) by (simp add: algebra_simps)
qed

lemma Chebyshev-sum-upper-nat:
  fixes a b :: nat \Rightarrow nat
  shows \(\forall i j. [i \leq j; j \leq n] \implies a i \leq a j\) \(\Rightarrow\)
  \(\forall i j. [i \leq j; j \leq n] \implies b i \geq b j\) \(\Rightarrow\)
107 Infinite Series

theory Series
imports Limits Inequalities
begin

107.1 Definition of infinite summability

definition sums :: (nat ⇒ 'a::{topological-space, comm-monoid-add}) ⇒ 'a ⇒ bool
  (infixr sums 80)
where f sums s ⇐→ (λn. ∑i< n. f i) −→ s

definition summable :: (nat ⇒ 'a::{topological-space, comm-monoid-add}) ⇒ bool
where summable f ⇐→ (∃s. f sums s)

definition suminf :: (nat ⇒ 'a::{topological-space, comm-monoid-add}) ⇒ 'a
  (binder ∑)
where suminf f = (THE s. f sums s)

Variants of the definition

lemma sums-def': f sums s ⇐→ (λn. ∑i=0..< n. f i) −→ s
  unfolding sums-def
  apply (subst LIMSEQ-Suc-iff [symmetric])
  apply (simp only: lessThan-Suc-atMost atLeast0AtMost)
  done

lemma sums-def-le: f sums s ⇐→ (λn. ∑i=0..< n. f i) −→ s
  by (simp add: sums-def' atMost-atLeast0)

lemma bounded-imp-summable:
  fixes a :: nat ⇒ 'a::{conditionally-complete-linorder,linorder-topology,linordered-comm-semiring-strict}
  assumes 0: ∃n. a n ≥ 0 and bounded: ∃n. (∑k≤n. a k) ≤ B
  shows summable a
proof –
  have bdd-above (range(λn. ∑k≤n. a k))
    by (meson bdd-above12 bounded)
  moreover have incseq (λn. ∑k≤n. a k)
    by (simp add: mono-def 0 sum-mono2)
  ultimately obtain s where (λn. ∑k≤n. a k) −→ s
    using LIMSEQ-incseq-SUP by blast
  then show ?thesis
    by (auto simp: sums-def-le summable-def)
107.2 Infinite summability on topological monoids

**lemma** sums-subst [trans]: \( f = g \implies g \text{ sums } z \implies f \text{ sums } z \)
by simp

**lemma** sums-cong: \((\forall n. f n = g n) \implies f \text{ sums } c \iff g \text{ sums } c\)
by (drule ext) simp

**lemma** sums-summable: \( f \text{ sums } l \implies \text{summable } f \)
by (simp add: sums-def summable-def, blast)

**lemma** summable-iff-convergent: \( \text{summable } f \iff \text{convergent } (\lambda n. \sum_{i < n} f i) \)
by (simp add: summable-def sums-def convergent-def)

**lemma** summable-iff-convergent': \( \text{summable } f \iff \text{convergent } (\lambda n. \text{sum } f \{..n\}) \)
by (simp-all only: summable-iff-convergent convergent-def lessThan-Suc-atMost [symmetric] LIMSEQ-Suc-iff[of \( \lambda n. \text{sum } f \{..<n\} \)])

**lemma** suminf-eq-lim: \( \text{suminf } f = \lim (\lambda n. \sum_{i < n} f i) \)
by (simp add: suminf-def sums-def lim-def)

**lemma** sums-zero [simp, intro]: \( (\lambda n. 0) \text{ sums } 0 \)
unfolding sums-def by simp

**lemma** summable-zero [simp, intro]: \( \text{summable } (\lambda n. 0) \)
by (rule sums-zero [THEN sums-summable])

**lemma** sums-group: \( f \text{ sums } s \implies 0 < k \implies (\lambda n. \text{sum } f \{n * k ..< n * k + k\}) \text{ sums } s \)
apply (simp only: sums-def sum.nat-group tendsto-def sequentially)
apply (erule all-forward imp-forward exE | assumption)+
apply (rule-tac x=N in exI)
by (metis le-square mult.commute mult.left-neutral mult.le-cancel2 mult.le-mono)

**lemma** suminf-cong: \( (\forall n. f n = g n) \implies \text{suminf } f = \text{suminf } g \)
by (rule arg-cong[of f g, rule ext]) simp

**lemma** summable-cong:
fixes \( f, g :: \text{nat} \Rightarrow 'a::real-normed-vector \)
assumes eventually \((\lambda x. f x = g x) \text{ sequentially} \)
shows \( \text{summable } f = \text{summable } g \)
proof -
from assms obtain \( N \) where \( \forall n \geq N. f n = g n \)
  by (auto simp: eventually-at-top-inorder)
define \( C \) where \( C = (\sum_{k < N} f k - g k) \)
from eventually-ge-at-top[of \( N \)]
have eventually \((\lambda n. \text{sum } f \{..<n\} = C + \text{sum } g \{..<n\}) \text{ sequentially} \)
proof eventually-elim
  case (elim n)
  then have \{..n\} = \{..<N\} \cup \{N..<n\}
  by auto
  also have \(\sum f\{..<n\} = \sum f\{..<N\} + \sum f\{N..<n\}\)
  by (intro sum.union-disjoint) auto
  also from \(N\) have \(\sum f\{N..<n\} = \sum g\{N..<n\}\)
  by (intro sum.cong) simp-all
  also have \(\sum f\{..<N\} + \sum g\{N..<n\} = C + (\sum g\{..<N\} + \sum g\{N..<n\}\)
  unfolding C-def by (simp add: algebra-simps sum-subtractf)
  also have \(\sum g\{..<N\} + \sum g\{N..<n\} = \sum g(\{..<N\} \cup \{N..<n\}\)
  by (rule sum.union-disjoint [symmetric]) auto
  finally show \(\sum f\{..<n\} = C + \sum g\{..<n\}\).
qed
from convergent-cong[OF this] show ?thesis
  by (simp add: summable_iff_convergent convergent_add_const_iff)
qed

lemma sums-finite:
  assumes [simp]: finite \(N\)
  and \(f\): \(\forall n. n \notin \(N\) \Rightarrow f n = 0\)
  shows \(f\) sums \((\sum n \in N. f n)\)
proof
  have eq: \(\sum f\{..<n + Suc\ (Max \(N\))\} = \sum f\{..<N\}\) for \(n\)
  by (rule sum.mono_neutral_right) (auto simp: add_increasing less_Suc_le f)
  show ?thesis
  unfolding sums_def
  by (rule LIMSEQ_offset[of - Suc (Max \(N\))])
    (simp add: eq atLeast0LessThan del: add_Suc_right)
qed

corollary sums-0: \((\forall n. f n = 0) \Rightarrow (f\) sums 0\)
  by (metis (no_types) finite.emptyI sum.empty sums-finite)

lemma summable-finite: finite \(N\) \(\Rightarrow (\forall n. n \notin \(N\) \Rightarrow f n = 0) \Rightarrow \) summable \(f\)
  by (rule sums-summable) (rule sums-finite)

lemma sums-If-finite-set: finite \(A\) \(\Rightarrow (\lambda r. \text{if } r \in A \text{ then } f r \text{ else } 0)\) \(\) sums \((\sum r \in A. f r)\)
  using sums-finite[of \(A\) (\(\lambda r. \text{if } r \in A \text{ then } f r \text{ else } 0\))] by simp

lemma summable-If-finite-set[simp, intro]: finite \(A\) \(\Rightarrow\) summable \((\lambda r. \text{if } r \in A \text{ then } f r \text{ else } 0)\)
  by (rule sums-summable) (rule sums-If-finite-set)

lemma sums-If-finite: finite \(\{r. P r\}\) \(\Rightarrow (\lambda r. \text{if } P r \text{ then } f r \text{ else } 0)\) \(\) sums \((\sum r |
using sums-If-finite-set[of \{r. P r\}] by simp

lemma summable-If-finite[simp, intro]: finite \{r. P r\} \implies summable (\lambda r. if P r then f r else 0)
by (rule sums-summable) (rule sums-If-finite)

lemma sums-single: (\lambda r. if r = i then f r else 0) sums f i
using sums-If-finite[of \lambda r. r = i] by simp

lemma summable-single[simp, intro]: summable (\lambda r. if r = i then f r else 0)
by (rule sums-summable) (rule sums-single)

context
fixes f :: nat ⇒ 'a::{t2-space,comm-monoid-add}
begin

lemma summable-sums[intro]: summable f \implies f sums (suminf f)
by (simp add: summable-def sums-def suminf-def)
  (metis convergent-LIMSEQ-iff convergent-def lim-def)

lemma summable-LIMSEQ: summable f \implies (\lambda n. \sum i<n. f i) −−−−→ suminf f
by (rule summable-sums [unfolded sums-def])

lemma summable-LIMSEQ’: summable f \implies (\lambda n. \sum i\le n. f i) −−−−→ suminf f
using sums-def-le by blast

lemma sums-unique: f sums s \implies s = suminf f
by (metis limI suminf-eq-lim sums-def)

lemma sums-iff: f sums x \iff summable f ∧ suminf f = x
by (metis summable-sums sums-summable sums-unique)

lemma summable-sums-iff: summable f \iff f sums suminf f
by (auto simp: sums-iff summable-sums)

lemma sums-unique2: f sums a \implies f sums b \implies a = b
for a b :: 'a
by (simp add: sums-iff)

lemma suminf-finite:
assumes N: finite N
and f: \\\(\forall n. n \notin N \implies f n = 0\)
shows suminf f = (\sum n\in N. f n)
using sums-finite[OF assms, THEN sums-unique] by simp
end

lemma suminf-zero[simp]: suminf (\lambda n. 0::'a::{t2-space, comm-monoid-add}) = 0
THEORY "Series" 2017

by (rule sums-zero [THEN sums-unique, symmetric])

107.3 Infinite summability on ordered, topological monoids

lemma sums-le: \( \forall n \cdot f n \leq g n \Rightarrow \text{sums } s \Rightarrow g \text{ sums } t \Rightarrow s \leq t \)
for \( f, g : \text{nat} \Rightarrow 'a :: \{\text{ordered-comm-monoid-add, linorder-topology}\} \)
by (rule LIMSEQ-le) (auto intro: sum-mono simp: sums-def)

context
fixes \( f : \text{nat} \Rightarrow 'a :: \{\text{ordered-comm-monoid-add, linorder-topology}\} \)
begin

lemma suminf-le: \( \forall n \cdot f n \leq g n \Rightarrow \text{summable } f \Rightarrow \text{summable } g \Rightarrow \text{suminf } f \leq \text{suminf } g \)
by (auto dest: sums-summable intro: sums-le)

lemma sum-le-suminf: shows \( \text{summable } f \Rightarrow \text{finite } I \Rightarrow \forall m \in - I \cdot 0 \leq f m \Rightarrow \text{sum } f I \leq \text{suminf } f \)
using sum-le-suminf by force

lemma suminf-nonneg: \( \text{summable } f \Rightarrow \forall n \cdot 0 \leq f n \Rightarrow 0 \leq \text{suminf } f \)
using sum-le-suminf by force

lemma suminf-const: \( \text{summable } f \Rightarrow (\forall n. \text{sum } f \{..<n\} \leq x) \Rightarrow \text{suminf } f \leq x \)
by (metis LIMSEQ-const2 summable-LIMSEQ)

lemma suminf-eq-zero-iff: \( \forall n \cdot 0 \leq f n \Rightarrow \text{suminf } f = 0 \leftrightarrow (\forall n. f n = 0) \)
proof
assume \( \text{suminf } f = 0 \) and \( \pos: \forall n \cdot 0 \leq f n \)
then have \( f : (\lambda n. \sum i<n. f i) \rightarrow 0 \)
using \( \text{summable-LIMSEQ}[\text{of } f] \) by simp
then have \( \forall i. (\sum n \in \{i\}. f n) \leq 0 \)
proof (rule \text{LIMSEQ-le-const})
show \( \exists N. \forall n \geq N. (\sum n \in \{i\}. f n) \leq \text{sum } f \{..<n\} \) for \( i \)
using \( \pos \) by (intro \text{exI}[\text{of } - \text{Suc } i] \text{ allI impl sum-mono2} \) auto
qed
with \( \pos \) show \( \forall n. f n = 0 \)
by (auto intro!: antisym)
qed (metis \text{suminf-zero fun-eq-iff})

lemma suminf-pos-iff: \( \text{summable } f \Rightarrow \forall n. 0 \leq f n \Rightarrow 0 < \text{suminf } f \leftrightarrow (\exists i. 0 < f i) \)
using \( \text{sum-le-suminf}[\text{of } \{\}] \) \text{suminf-eq-zero-iff} by (simp add: less-le)

lemma suminf-pos2: assumes \( \text{summable } f \forall n. 0 \leq f n 0 < f i \)
shows \( 0 < \text{suminf } f \)
proof –

have 0 < (∑ n< Suc i. f n)
  using assms by (intro sum-pos2[where \( i=0 \)]) auto
also have \( \ldots \leq \operatorname{suminf} f \)
  using assms by (intro sum-le-suminf) auto
finally show \( \text{thesis} \).
qed

lemma suminf-pos: summable \( f \) \( \Longrightarrow \) \( \forall \, n. \, 0 < f \, n \Longrightarrow 0 < \operatorname{suminf} f \)
by (intro suminf-pos2[where \( i=0 \)]) (auto intro: less-imp-le)
end

context

fixes \( f :: \text{nat} \)
\( \text{a} :: \{\text{ordered-cancel-comm-monoid-add,linorder-topology}\} \)
begin

lemma sum-less-suminf2:
  summable \( f \) \( \Longrightarrow \) \( \forall \, m \geq n. \, 0 \leq f \, m \Longrightarrow n \leq i \Longrightarrow 0 < f \, i \Longrightarrow \operatorname{sum} \{..<n\} < \operatorname{sum} \{..<i\} \operatorname{suminf} \, f \)
  using \( \operatorname{sum-le-suminf}[\text{of} \, f \{..< \text{Suc} \, i\}] \)
  and \( \operatorname{add-strict-increasing}[\text{of} \, f \, i \, \operatorname{sum} \{..<n\} \, \operatorname{sum} \{..<i\}] \)
  and \( \operatorname{sum-mono2}[\text{of} \{..<i\} \{..<n\}] \)
  by (auto simp: less-imp-le ac-simps)

lemma sum-less-suminf:
  summable \( f \) \( \Longrightarrow \) \( \forall \, m \geq n. \, 0 < f \, m \Longrightarrow \operatorname{sum} \{..<n\} < \operatorname{suminf} \, f \)
  using sum-less-suminf2[of \( n \) \( n \)] by (simp add: less-imp-le)
end

lemma summableI-nonneg-bounded:
fixes \( f :: \text{nat} \) \( \Rightarrow \text{a} :: \{\text{ordered-comm-monoid-add,linorder-topology,conditionally-complete-linorder}\} \)
assumes \( \text{pos}[\text{simp}]: \forall \, n. \, 0 \leq f \, n \)
  and \( \text{le} : \forall n. \, (\sum i<n. f \, i) \leq x \)
shows summable \( f \)
unfolding summable-def sums-def [abs-def]
proof (rule \text{exI} \text{LIMSEQ-incseq-SUP}+)
  show \( \operatorname{bdd-above} (\range (\lambda n. \, \operatorname{sum} \{..<n\})) \)
    using \( \text{le} \) by (auto simp: \( \operatorname{bdd-above-def} \))
  show \( \operatorname{incseq} (\lambda n. \, \operatorname{sum} \{..<n\}) \)
    by (auto simp: \( \operatorname{mono-def introl} : \operatorname{sum-mono2} \))
qed

lemma summableI[intro, simp]: summable \( f \)
for \( f :: \text{nat} \Rightarrow \text{a} :: \{\text{canonically-ordered-monoid-add,linorder-topology,complete-linorder}\} \)
by (intro summableI-nonneg-bounded[where \( x=\text{top} \)] \( \text{zero-le \, top-greatest} \))

lemma suminf-eq-SUP-real:
assumes $X$: summable $X \land i. \, 0 \leq X \ i$ shows $\text{suminf} \ X = (\sup i. \, \sum n < i. \, X \ n)$.

by (intro LIMSEQ-unique[OF summable-LIMSEQ] X LIMSEQ-incseq-SUP)
(auto intro!: bdd-aboveI2[where $M = \sum i. \, X \ i$] sum-le-suminf X monoI sum-mono2)

107.4 Infinite summability on topological monoids

context
fixes $f \ g :: \text{nat} \Rightarrow 'a :: \{ \text{t2-space, topological-comm-monoid-add} \}$

begin

lemma sums-Suc:
  assumes $(\lambda n. \, f (\text{Suc} \ n))$ sums $l$
  shows $f$ sums $(l + f \ 0)$

proof
  have $(\lambda n. \, (\sum i < n. \, f (\text{Suc} \ i)) + f \ 0)$ $\longrightarrow l + f \ 0$
    using assms by (auto intro!: tendsto-add simp: sums-def)
  moreover have $(\sum i < n. \, f (\text{Suc} \ i)) + f \ 0 = (\sum i < \text{Suc} \ n. \, f \ i)$
    for $n$
    unfolding lessThan-Suc-eq-insert-0 by (simp add: ac-simps sum.atLeast1-atMost-eq image-Suc-lessThan)
  ultimately show $?thesis$
    by (auto simp: sums-def simp del: sum.lessThan-Suc intro: LIMSEQ-Suc-iff[THEN iffD1])

qed

lemma sums-add: $f$ sums $a$ $\Longrightarrow$ $g$ sums $b$ $\Longrightarrow$ $(\lambda n. \, f \ n + g \ n)$ sums $(a + b)$
  unfolding sums-def by (simp add: sum.distrib tendsto-add)

lemma summable-add: summable $f$ $\Longrightarrow$ summable $g$ $\Longrightarrow$ summable $(\lambda n. \, f \ n + g \ n)$
  unfolding summable-def by (auto intro: sums-add)

lemma suminf-add: summable $f$ $\Longrightarrow$ summable $g$ $\Longrightarrow$ $\text{suminf} \ f + \text{suminf} \ g$
  $(\sum n. \, f \ n + g \ n)$
  by (intro sums-unique sums-add summable-sums)

end

context
fixes $f :: 'i \Rightarrow \text{nat} \Rightarrow 'a :: \{ \text{t2-space, topological-comm-monoid-add} \}$
  and $I :: 'i \ set$

begin

lemma sums-sum: $(\forall i. \, i \in I \Longrightarrow (f \ i) \ \text{sums} \ (x \ i))$ $\Longrightarrow$ $(\lambda n. \, \sum i \in I. \, f \ i \ n)$ sums $(\sum i \in I. \, x \ i)$
  by (induct $I$ rule: infinite-finite-induct) (auto intro!: sums-add)

lemma suminf-sum: $(\forall i. \, i \in I \Longrightarrow \text{summable} \ (f \ i))$ $\Longrightarrow$ $(\sum n. \, \sum i \in I. \, f \ i \ n)$
  $(\sum i \in I. \, \sum n. \, f \ i \ n)$
  by (induct $I$ rule: infinite-finite-induct) (auto intro!: sums-add)
using sums-unique[OF sums-sum, OF summable-sums] by simp

lemma summable-sum: \( \forall i. \ i \in I \implies \text{summable } (f_i) \implies \text{summable } (\lambda n. \sum_{i \in I} f_i n) \)

end

lemma sums-If-finite-set' :
  fixes \( f, g : \mathbb{nat} \)
  assumes \( g \text{ sums } S \) and \( \text{finite } A \) and \( S' = S + (\sum_{n \in A} f_n - g_n) \)
  shows \( (\lambda n. \text{if } n \in A \text{ then } f_n \text{ else } g_n) \text{ sums } S' \)
proof
  have \( (\lambda n. g_n + (\text{if } n \in A \text{ then } f_n - g_n \text{ else } 0)) \text{ sums } (S + (\sum_{n \in A} f_n - g_n)) \)
  by \( \text{intro sums-add assms sums-If-finite-set} \)
  also have \( \ldots \text{←→ } (\lambda n. f_n) \text{ sums } S \)
  by \( \text{simp add: fun-eq-iff} \)
  finally show \( \text{thesis using assms by simp} \)
qed

107.5 Infinite summability on real normed vector spaces

context
  fixes \( f : \mathbb{nat} \Rightarrow 'a::real-normed-vector \)
begin

lemma sums-Suc-iff: \( (\lambda n. f \ (SUC n)) \text{ sums } s \overset{\sim}{\longleftrightarrow} f \text{ sums } (s + f 0) \)
proof
  have \( f \text{ sums } (s + f 0) \overset{\sim}{\longleftrightarrow} (\lambda i. \sum_{j < Suc i} f_j) \overset{\sim}{\longrightarrow} s + f 0 \)
  by \( \text{subst LimSEQ-Suc-iff} \) \( \text{simp add: sums-def} \)
  also have \( \ldots \overset{\sim}{\longleftrightarrow} (\lambda i. (\sum_{j < Suc i} f_j) + f 0) \overset{\sim}{\longrightarrow} s + f 0 \)
  by \( \text{simp add: ac-simps lessThan-Suc-insert-0 image-Suc-lessThan sum.atLeast1-atMost-eq} \)
  also have \( \ldots \overset{\sim}{\longleftrightarrow} (\lambda n. f \ (SUC n)) \text{ sums } s \)
  proof
    assume \( (\lambda i. (\sum_{j < Suc i} f_j) + f 0) \overset{\sim}{\longrightarrow} s + f 0 \)
    with tendsto-add[OF this tendsto-const, of \(- f 0\)] show \( (\lambda i. f \ (SUC i)) \text{ sums } s \)
    by \( \text{simp add: sums-def} \)
  qed \( \text{(auto intro: tendsto-add simp: sums-def)} \)
  finally show \( \text{thesis} .. \)
qed

lemma summable-Suc-iff: summable \( (\lambda n. f \ (SUC n)) = \text{summable } f \)
proof
  assume summable \( f \)
  then have \( f \text{ sums } \text{suminf } f \)
  by \( \text{rule summable-sums} \)
  then have \( (\lambda n. f \ (SUC n)) \text{ sums } (\text{suminf } f - f 0) \)
by (simp add: sums-Suc-iff)
then show summable (λn. f (Suc n))
  unfolding summable-def by blast
qed (auto simp: sums-Suc-iff summable-def)

lemma sums-Suc-imp: f 0 = 0 =⇒ (λn. f (Suc n)) sums s =⇒ (λn. f n) sums s
  using sums-Suc-iff by simp

end

context fixes f :: nat ⇒ 'a::real-normed-vector
begin

lemma sums-diff: f sums a =⇒ g sums b =⇒ (λn. f n - g n) sums (a - b)
  unfolding sums-def by (simp add: sum-subtractf tendsto-diff)

lemma summable-diff: summable f =⇒ summable g =⇒ summable (λn. f n - g n)
  unfolding summable-def by (auto intro: sums-diff)

lemma suminf-diff: summable f =⇒ summable g =⇒ suminf f - suminf g =
  (∑n. f n - g n)
  by (intro sums-unique sums-diff summable-sums)

lemma sums-minus: f sums a =⇒ (λn. - f n) sums (- a)
  unfolding sums-def by (simp add: sum-negf tendsto-minus)

lemma summable-minus: summable f =⇒ summable (λn. - f n)
  unfolding summable-def by (auto intro: sums-minus)

lemma suminf-minus: summable f =⇒ (∑n. - f n) = - (∑n. f n)
  by (intro sums-unique [symmetric] sums-minus summable-sums)

lemma sums-iff-shift: (λi. f (i + n)) sums s =⇒ f sums (s + (∑i<n. f i))
  proof (induct n arbitrary: s)
    case 0
    then show ?case by simp
  next
    case (Suc n)
    then have (λi. f (Suc i + n)) sums s =⇒ (λi. f (i + n)) sums (s + f n)
      by (subst sums-Suc-iff) simp
    with Suc show ?case
      by (simp add: ac-simps)
  qed

corollary sums-iff-shift': (λi. f (i + n)) sums (s - (∑i<n. f i)) =⇒ f sums s
  by (simp add: sums-iff-shift)
lemma sums-zero-iff-shift:
  assumes \( \forall i. \ i < n \implies f_i = 0 \)
  shows \( (\lambda i. \ f_{i+n}) \text{ sums } s \iff (\lambda i. \ f_i) \text{ sums } s \)
  by (simp add: assms sums-iff-shift)

lemma summable-iff-shift: summable \( (\lambda n. \ f_{n+k}) \iff \text{summable } f \)
  by (metis diff-add-cancel summable-def sums-iff-shift [abs-def])

lemma sums-split-initial-segment: \( f \text{ sums } s \iff (\lambda i. \ f_{i+n}) \text{ sums } (s - \sum_{i<n} f_i) \)
  by (simp add: sums-iff-shift)

lemma summable-ignore-initial-segment: summable \( f \iff \text{summable } (\lambda n. \ f_{n+k}) \)
  by (simp add: summable-iff-shift)

lemma suminf-minus-initial-segment: \( \text{summable } f \iff \sum_n f_{n+k} = \sum_n f_n - \sum_{i<k} f_i \)
  by (rule sums-unique[symmetric]) (auto simp: sums-iff-shift)

lemma suminf-split-head: \( \text{summable } f \iff (\sum_n f_{\text{Suc } n}) = \text{suminf } f - f_0 \)
  using suminf-split-initial-segment[of 1] by simp

lemma suminf-exist-split:
  fixes \( r :: \text{real} \)
  assumes \( 0 < r \text{ and } \text{summable } f \)
  shows \( \exists N. \ \forall n \geq N. \ \text{norm } (\sum i. \ f_{i+n}) < r \)
  proof
    from LIMSEQ-D[OF summable-LIMSEQ[OF ⟨summable f⟩] ⟨0 < r⟩]
    obtain \( N :: \text{nat} \) where \( \forall n \geq N. \ \text{norm } \sum f \{..<n\} - \text{suminf } f < r \)
    by auto
    then show ?thesis
      by (auto simp: norm-minus-commute suminf-minus-initial-segment[OF ⟨summable f⟩])
  qed

lemma summable-LIMSEQ-zero: \( \text{summable } f \iff f \longrightarrow 0 \)
  apply (drule summable-iff-convergent [THEN iffD1])
  apply (drule convergent-Cauchy)
  apply (simp only: Cauchy-iff LIMSEQ-iff)
  by (metis add.commute add-diff-cancel_right' diff-zero le-SucI sum_lessThan_Suc)

lemma summable-imp-convergent: \( \text{summable } f \implies \text{convergent } f \)
  by (force dest!: summable-LIMSEQ-zero simp: convergent-def)
lemma summable-imp-Bseq: summable f \implies Bseq f
by (simp add: convergent-imp-Bseq summable-imp-convergent)

end

lemma summable-minus-iff: summable (\lambda n. - f n) \iff summable f
for f :: nat \Rightarrow 'a::real-normed-vector
by (auto dest: summable-minus)

lemma (in bounded-linear) sums: (\lambda n. X n) sums a \implies (\lambda n. f (X n)) sums (f a)
unfolding sums-def by (drule tendsto) (simp only: sum)

lemma (in bounded-linear) summable: summable (\lambda n. X n) \implies summable (\lambda n. f (X n))
unfolding summable-def by (auto intro: sums)

lemma (in bounded-linear) suminf: summable (\lambda n. X n) \implies f (\sum n. X n) = (\sum n. f (X n))
by (intro sums-unique sums summable-sums)

lemmas sums-of-real = bounded-linear.sums [OF bounded-linear-of-real]
lemmas summable-of-real = bounded-linearsummable [OF bounded-linear-of-real]
lemmas suminf-of-real = bounded-linear.suminf [OF bounded-linear-of-real]

lemmas sums-scaleR-left = bounded-linear.sums[OF bounded-linear-scaleR-left]
lemmas summable-scaleR-left = bounded-linearsummable[OF bounded-linear-scaleR-left]
lemmas suminf-scaleR-left = bounded-linearsuminf[OF bounded-linear-scaleR-left]

lemmas sums-scaleR-right = bounded-linear.sums[OF bounded-linear-scaleR-right]
lemmas summable-scaleR-right = bounded-linearsummable[OF bounded-linear-scaleR-right]
lemmas suminf-scaleR-right = bounded-linearsuminf[OF bounded-linear-scaleR-right]

lemma summable-const-iff: summable (\lambda n. c) \iff c = 0
for c :: 'a::real-normed-vector
proof
have \neg summable (\lambda n. c) if c \neq 0
proof
from that have filterlim (\lambda n. of-nat n * norm c) at-top sequentially
by (subst mult.commute)
(auto intro: filterlim-tendsto-pos-mult-at-top filterlim-real-sequentially)
then have \neg convergent (\lambda n. norm (\sum k<n. c))
by (intro filterlim-at-infinity-imp-not-convergent filterlim-at-top-imp-at-infinity)
(simp-all add: sum-constant-scaleR)
then show ?thesis
unfolding summable-iff-convergent using convergent-norm by blast
qed
then show ?thesis by auto
qed
107.6 Infinite summability on real normed algebras

context
  fixes f :: nat ⇒ ′a::real-normed-algebra

begin

lemma sums-mult: f sums a ⇒ (∑n. c * f n) sums (c * a)
  by (rule bounded-linear.sums [OF bounded-linear-mult-right])

lemma summable-mult: summable f ⇒ summable (∑n. c * f n)
  by (rule bounded-linearsummable [OF bounded-linear-mult-right])

lemma suminf-mult: summable f ⇒ suminf (∑n. c * f n) = c * suminf f
  by (rule bounded-linear.suminf [OF bounded-linear-mult-right, symmetric])

lemma sums-mult2: f sums a ⇒ (∑n. f n * c) sums (a * c)
  by (rule bounded-linear.sums [OF bounded-linear-mult-left])

lemma summable-mult2: summable f ⇒ summable (∑n. f n * c)
  by (rule bounded-linear.summable [OF bounded-linear-mult-left])

lemma suminf-mult2: summable f ⇒ suminf f * c = (∑n. f n * c)
  by (rule bounded-linear.suminf [OF bounded-linear-mult-left])

end

lemma sums-mult-iff:
  fixes f :: nat ⇒ ′a::{real-normed-algebra,field}
  assumes c ≠ 0
  shows (∑n. c * f n) sums (c * d) ←→ f sums d
  using sums-mult[of f d c] sums-mult[of λn. c * f n c * d inverse c]
  by (force simp: field-simps assms)

lemma sums-mult2-iff:
  fixes f :: nat ⇒ ′a::{real-normed-algebra,field}
  assumes c ≠ 0
  shows (∑n. f n * c) sums (d * c) ←→ f sums d
  using sums-mult2-iff[of f d c] sums-mult2-iff[of λn. c * f n c * d inverse c]
  by (simp add: mult.commute)

lemma sums-of-real-iff:
  (λn. of-real (f n) :: ′a::real-normed-div-algebra) sums-of-real c ←→ f sums c

107.7 Infinite summability on real normed fields

context
  fixes c :: ′a::real-normed-field

begin

lemma sums-divide: f sums a ⇒ (∑n. f n / c) sums (a / c)
by (rule bounded-linear.sums [OF bounded-linear-divide])

lemma summable-divide: summable f \implies summable (\lambda n. f n / c)
  by (rule bounded-linearsummable [OF bounded-linear-divide])

lemma suminf-divide: summable f \implies suminf (\lambda n. f n / c) = suminf f / c
  by (rule bounded-linear.suminf [OF bounded-linear-divide, symmetric])

lemma summable-inverse-divide: summable (inverse \circ f) \implies summable (\lambda n. c / f n)
  by (auto dest: summable-mult [of - c] simp: field-simps)

lemma sums-mult-D: (\lambda n. c \cdot f n) sums a \implies c \neq 0 \implies f sums (a / c)
  using sums-mult-iff by fastforce

lemma summable-mult-D: summable (\lambda n. c \cdot f n) \implies c \neq 0 \implies summable f
  by (auto dest: summable-divide)

Sum of a geometric progression.

lemma geometric-sums:
  assumes norm c < 1
  shows (\lambda n. c ^ n) sums (1 / (1 - c))
proof
  have neq-0: c - 1 \neq 0
    using assms by auto
  then have (\lambda n. c ^ n / (c - 1) - 1 / (c - 1)) \longrightarrow 0 / (c - 1) - 1 / (c - 1)
    by (intro tendsto-intros assms)
  then have (\lambda n. (c ^ n - 1) / (c - 1)) \longrightarrow 1 / (1 - c)
    by (simp add: nonzero-minus-divide-right [OF neq-0] diff-divide-distrib)
  with neq-0 show (\lambda n. c ^ n) sums (1 / (1 - c))
    by (simp add: sums-def geometric-sum)
qed

lemma summable-geometric-iff: summable (\lambda n. c ^ n) \iff norm c < 1
proof
  assume summable (\lambda n. c ^ n :: 'a :: real-normed-field)
  then have (\lambda n. norm c ^ n) \longrightarrow 0
    by (simp add: norm-power [symmetric] tendsto-norm-zero-iff summable-LIMSEQ-zero)
  from order-tendstoD (2) [OF this zero-less-one] obtain n where norm c ^ n < 1
    by (auto simp: eventually-at-top-linorder)
  then show norm c < 1 using one-le-power [of norm c n]
by (cases norm $c \geq 1$) (linarith, simp)
qed (rule summable-geometric)

end

lemma power-half-series: $(\lambda n. (1/2::real) ^{\text{Suc } n})$ sums 1
proof
  have 2: $(\lambda n. (1/2::real) ^n)$ sums 2
    using geometric-sums [of $1/2::real$] by auto
  have $(\lambda n. (1/2::real) ^{\text{Suc } n}) = (\lambda n. (1 / 2) ^ n / 2)$
    by (simp add: mult.commute)
  then show ?thesis
    using sums-divide [OF 2, of 2] by simp
qed

107.8 Telescoping

lemma telescope-sums:
  fixes $c :: a::real-normed-vector$
  assumes $f \longrightarrow c$
  shows $(\lambda n. f (\text{Suc } n) - f n)$ sums $(c - f 0)$
unfolding sums-def
proof (subst LIMSEQ-Suc-iff [symmetric])
  have $(\lambda n. \sum k<Suc n. f (\text{Suc } k) - f k) = (\lambda n. f (\text{Suc } n) - f 0)$
    by (simp add: lessThan-Suc-atMost atLeast0AtMost [symmetric] sum-Suc-diff)
  also have \ldots $\longrightarrow c - f 0$
    by (intro tendsto-diff LIMSEQ-Suc[OF assms] tendsto-const)
  finally show $(\lambda n. \sum n<Suc n. f (\text{Suc } n) - f n) \longrightarrow c - f 0$.
qed

lemma telescope-sums':
  fixes $c :: a::real-normed-vector$
  assumes $f \longrightarrow c$
  shows $(\lambda n. f n - f (\text{Suc } n))$ sums $(f 0 - c)$
using sums-minus[OF telescope-sums[OF assms]] by (simp add: algebra-simps)

lemma telescope-summable:
  fixes $c :: a::real-normed-vector$
  assumes $f \longrightarrow c$
  shows summable $(\lambda n. f (\text{Suc } n) - f n)$
using telescope-sums[OF assms] by (simp add: sums-iff)

lemma telescope-summable':
  fixes $c :: a::real-normed-vector$
  assumes $f \longrightarrow c$
  shows summable $(\lambda n. f n - f (\text{Suc } n))$
using summable-minus[OF telescope-summable[OF assms]] by (simp add: algebra-simps)
107.9 Infinite summability on Banach spaces

Cauchy-type criterion for convergence of series (c.f. Harrison).

**Lemma** summable-Cauchy: summable \( f \) \( \iff \) (\( \forall e > 0. \exists N. \forall m \geq N. \forall n. \) \( \| \sum_{m.. < n} \| < e \))

**Proof**

1. Assume \( f \) is summable.
2. Show \( \forall e > 0. \exists N. \forall m \geq N. \forall n. \) \( \| \sum_{m.. < n} \| < e \)
   - Fix \( e \) and assume \( 0 < e \).
   - Obtain \( M \) where \( M \) is the maximum such that \( \| \sum_{m.. < n} \| < e \) for \( m \geq M \).
   - Using \( f \) by (force simp add: summable-iff-convergent Cauchy-iff)
     - Show \( \| \sum_{m.. < n} \| < e \) if \( m \geq N \) and \( n \geq N \)
       - Cases \( m \leq n \) and \( n \leq m \)
         - Assume \( m \leq n \) and show \( \| \sum_{m.. < n} \| < e \)
           - By (metis mono-tags, hide-lams) \( \| \sum_{m.. < n} \| < e \)
         - Assume \( n \leq m \) and show \( \| \sum_{m.. < n} \| < e \)
           - By blast
   - Then show \( \forall N. \forall m \geq N. \forall n. \) \( \| \sum_{m.. < n} \| < e \)
     - By blast
3. Assume \( r \) is the right-hand side.
4. Show \( f \) is summable.
5. Using \( f \) by (force simp add: summable-iff-convergent Cauchy-iff)
6. Show \( \forall e > 0. \exists N. \forall m \geq N. \forall n. \) \( \| \sum_{m.. < n} \| < e \)
   - Assume \( 0 < e \) and show \( \| \sum_{m.. < n} \| < e \) if \( m \geq N \) and \( n \geq N \)
     - Cases \( m \leq n \) and \( n \leq m \)
       - Assume \( m \leq n \) and show \( \| \sum_{m.. < n} \| < e \)
         - By (metis Groups-Big, sum-diff N finite-lessThan lessThan-minus-lessThan lessThan-subset-iff norm-minus-commute \( m \geq N \))
       - Assume \( n \leq m \) and show \( \| \sum_{m.. < n} \| < e \)
         - By (metis Groups-Big, sum-diff N finite-lessThan lessThan-minus-lessThan lessThan-subset-iff norm-minus-commute \( m \geq N \))
lessThan-subset-iff \langle n \geq N \rangle

qed

then show \exists M. \forall m \geq M. \forall n \geq M. \text{norm} (\text{sum} f \{..<m\} - \text{sum} f \{..<n\}) < e

by blast

qed

qed

lemma summable-Cauchy':

fixes f :: nat \Rightarrow 'a :: banach

assumes eventually (\lambda m. \forall n \geq m. \text{norm} (\text{sum} f \{m..<n\}) \leq g m) sequentially

assumes filterlim g (nhds 0) sequentially

shows summable f

proof (subst summable-Cauchy, intro allI impI, goal-cases)

case (1 e)

from order-tendstoD \[2\] (OF assms \[2\] this) and assms \[1\]

have eventually (\lambda m. \forall n. \text{norm} (\text{sum} f \{m..<n\}) < e) at-top

proof eventually-elim

case (elim m)

show \(?case

proof

fix n

from elim show \text{norm} (\text{sum} f \{m..<n\}) < e

by (cases n \geq m) auto

qed

qed

thus \(?case by (auto simp: eventually-at-top-linorder)

qed

context

fixes f :: nat \Rightarrow 'a::banach

begin

Absolute convergence imples normal convergence.

lemma summable-norm-cancel: summable (\lambda n. \text{norm} (f n)) \implies summable f

unfolding summable-Cauchy

apply (erule all-forward imp-forward ex-forward | assumption)+

apply (fastforce simp add: order-le-less-trans \[OF norm-sum\] order-le-less-trans \[OF abs-ge-self\])

done

lemma summable-norm: summable (\lambda n. \text{norm} (f n)) \implies \text{norm} (\text{suminf} f) \leq (\sum n. \text{norm} (f n))

by (auto intro: LIMSEQ-le tendsto-norm summable-norm-cancel summable-LIMSEQ norm-sum)

Comparison tests.

lemma summable-comparison-test:

assumes fg: \exists N. \forall n \geq N. \text{norm} (f n) \leq g n and g: summable g

shows summable f
proof –

obtain $N$ where $N: \forall n. \ n \geq N \implies \text{norm}(f n) \leq g n$

using assms by blast

show ?thesis

proof (clarsimp simp add: summable-Cauchy)

fix $e :: \text{real}$

assume $0 < e$

then obtain $Ng$ where $Ng: \forall m. \ m \geq Ng \implies \text{norm}(\sum g \{m \ldots < n\}) < e$

using $g$ by (fastforce simp: summable-Cauchy)

with $N$ have $\text{norm}(\sum f \{m \ldots < n\}) < e$ if $m \geq \max N \ Ng$ for $m n$

proof –

have $\text{norm}(\sum f \{m \ldots < n\}) \leq \sum g \{m \ldots < n\}$

using $N$ that by (force intro: sum-norm-le)

also have $\ldots \leq \text{norm}(\sum g \{m \ldots < n\})$

by simp

also have $\ldots < e$

using $Ng$ that by auto

finally show ?thesis .

qed

then show $\exists N. \ \forall m \geq N. \ \forall n. \ \text{norm}(\sum f \{m \ldots < n\}) < e$

by blast

qed

qed

lemma summable-comparison-test-ev:

\[ \text{eventually}\ \bigl(\lambda n. \ \text{norm}(f n) \leq g n \bigr) \quad \text{sequentially} \implies \text{summable} \ g \quad \implies \text{summable} \ f \]

by (rule summable-comparison-test) (auto simp: eventually-at-top-linorder)

A better argument order.

lemma summable-comparison-test': summable $g \implies (\forall n. \ n \geq N \implies \text{norm}(f n) \leq g n) \implies \text{summable} \ f$

by (rule summable-comparison-test) auto

107.10 The Ratio Test

lemma summable-ratio-test:

assumes $c < 1 \ \forall n. \ n \geq N \implies \text{norm}(f (\text{Suc} n)) \leq c \ast \text{norm}(f n)$

shows summable $f$

proof (cases $0 < c$)

case True

show summable $f$

proof (rule summable-comparison-test)

show $\exists N'. \ \forall n \geq N'. \ \text{norm}(f n) \leq (\text{norm}(f N) / (c ^ \text{N})) \ast c ^ \text{n}$

proof (intro exI allI impI)

fix $n$

assume $N \leq n$

then show $\text{norm}(f n) \leq (\text{norm}(f N) / (c ^ \text{N})) \ast c ^ \text{n}$

proof (induct rule: inc-induct)

case base
with True show ?case by simp
next
case (step m)
  have norm (f (Suc m)) / c ^ Suc m * c ^ n ≤ norm (f m) / c ^ m * c ^ n
    using ⟨0 < c | c < 1⟩ assms(2)[OF N ≤ m] by (simp add: field-simps)
  with step show ?case by simp
qed
show summable (λn. norm (f N) / c ^ N * c ^ n)
  using ⟨0 < c | c < 1⟩ by (intro summable-mult summable-geometric)
show summable f
  by (intro sums-summable[OF sums-finite, of {.. Suc N}]) (auto simp add: not-le Suc-less-eq2)
qed

Relations among convergence and absolute convergence for power series.

lemma Abel-lemma:
  fixes a :: 'a::real-normed-vector
  assumes r: 0 ≤ r
          and r0: r < r0
          and M: ⋀n. norm (a n) * r0 ^ n ≤ M
  shows summable (λn. norm (a n) * r ^ n)
proof (rule summable-comparison-test)
  show summable (λn. M * (r / r0) ^ n)
    using assms by (auto simp add: summable-mul summable-geometric)
  show norm (norm (a n) * r ^ n) ≤ M * (r / r0) ^ n for n
    using r r0 M [of n] dual-order.order-iff-strict
    by (fastforce simp add: abs-mul field-simps)
qed

Summability of geometric series for real algebras.

lemma complete-algebra-summable-geometric:
  fixes x :: 'a::real-normed-algebra-1,banach
  assumes norm x < 1
  shows summable (λn. x ^ n)
proof (rule summable-comparison-test)
  show \( \exists N. \forall n \geq N. \, \text{norm}(x^n) \leq \text{norm}(x^n) \)
  by (simp add: norm-power-ineq)
from `assms` show summable \((\lambda n. \text{norm}(x^n))\)
  by (simp add: summable-geometric)
qed

107.11 Cauchy Product Formula

Proof based on Analysis WebNotes: Chapter 07, Class 41 http://www.math.unl.edu/~webnotes/classes/class41/prp77.htm

lemma Cauchy-product-sums:
  fixes a b :: nat ⇒ 'a::{real_normed_algebra,banach}
  assumes a: summable \((\lambda k. \text{norm}(a(k)))\)
  and b: summable \((\lambda k. \text{norm}(b(k)))\)
  shows \((\lambda k. \sum_{i=k}^{n} a(i) \cdot b(k-i)) \text{ sums } ((\sum k. \, a(k)) \cdot (\sum k. \, b(k)))\)
proof –
  let \( ?S1 = \lambda n::nat. \{..<n\} \times \{..<n\} \)
  let \( ?S2 = \lambda n::nat. \{(i,j). \, i + j < n\} \)
  have S1-mono: \( \forall m. \, m \leq n \Longrightarrow ?S1 \, m \subseteq \, ?S1 \, n \text{ by auto} \)
  have S2-le-S1: \( \forall n. \, ?S2 \, n \subseteq \, ?S1 \, n \text{ by auto} \)
  have S1-le-S2: \( \forall n. \, ?S1 \, (n + 2) \subseteq \, ?S2 \, n \text{ by auto} \)
  have finite-S1: \( \forall n. \, \text{finite} \, (?S1 \, n) \text{ by simp} \)
  with S2-le-S1 have finite-S2: \( \forall n. \, \text{finite} \, (?S2 \, n) \text{ by (rule finite-subset)} \)
      let \( ?g = \lambda(i,j). \, a(i) \cdot b(j) \)
      let \( ?f = \lambda(i,j). \, \text{norm}(a(i)) \cdot \text{norm}(b(j)) \)
      have f-nonneg: \( \forall x. \, 0 \leq ?f \, x \text{ by auto} \)
      then have norm-sum-f: \( \forall A. \, \text{norm} \, (?f \, A) = \sum \, ?f \, A \)
          unfolding real-norm_def
          by (simp only: abs-of-nonneg_sum-nonneg [rule-format])
      have \((\lambda n. \, \sum_{k<n} \, a(k)) \cdot (\sum_{k<n} \, b(k))\) \(\text{ à }\) \((\sum k. \, a(k)) \cdot (\sum k. \, b(k))\)
          by (intro tendsto-mult summable-LIMSEQ summable-norm-cancel [OF `a] summable-norm-cancel [OF `b])
      then have 1: \((\lambda n. \, \text{sum} \, ?g \, (?S1 \, n))\) \(\text{ à }\) \((\sum k. \, a(k)) \cdot (\sum k. \, b(k))\)
          by (simp only: sum-product_sum.Sigma [rule-format] finite-lessThan)
      have \((\lambda n. \, \text{norm}(a(k))) \cdot (\sum_{k<n} \, \text{norm}(b(k)))\) \(\text{ à }\) \((\sum k. \, \text{norm}(a(k))) \cdot (\sum k. \, \text{norm}(b(k)))\)
          using a b by (intro tendsto-mult summable-LIMSEQ)
      then have \((\lambda n. \, \text{sum} \, ?f \, (?S1 \, n))\) \(\text{ à }\) \((\sum k. \, \text{norm}(a(k))) \cdot (\sum k. \, \text{norm}(b(k)))\)
          by (simp only: sum-product_sum.Sigma [rule-format] finite-lessThan)
      then have convergent \((\lambda n. \, \text{sum} \, ?f \, (?S1 \, n))\)
          by (rule convergentI)
      then have Cauchy: \( \text{Cauchy} \, (\lambda n. \, \text{sum} \, ?f \, (?S1 \, n)) \)
          by (rule convergent-Cauchy)
      have Zfun \((\lambda n. \, \text{sum} \, ?f \, (?S1 \, n - ?S2 \, n))\) sequentially
proof (rule ZfunI, simp only: eventually-sequentially norm-sum-f)
fix \( r :: \text{real} \)
assume \( r : 0 < r \)

from \( \text{CauchyD} [\text{OF Cauchy } r] \) obtain \( N \)
where \( \forall m \geq N, \forall n \geq N. \; \text{norm} (\sum \ ?f (\ ?S1 m) - \sum \ ?f (\ ?S1 n)) < r \) ..
then have \( \forall m n. \; N \leq n \implies n \leq m \implies \text{norm} (\sum \ ?f (\ ?S1 m - \ ?S1 n)) < r \)
by (simp only: \text{sum-diff finite-S1 S1-mono})

then have \( \exists N. \forall n \geq N. \sum \ ?f (\ ?S1 n - \ ?S2 n) < r \)
proof (intro \text{exI allI impI})
fix \( n \)
assume \( 2 * N \leq n \)
then have \( n : N \leq n \div 2 \) by simp
have \( \sum \ ?f (\ ?S1 n - \ ?S2 n) \leq \sum \ ?f (\ ?S1 n - \ ?S1 (n \div 2)) \)
by (intro \text{sum-mono2 finite-Diff finite-S1 f-nonneg Diff-mono subset-refl S1-le-S2})
also have \( \ldots < r \)
using \( n \div-le-dividend \) by (rule \( N \))
finally show \( \sum \ ?f (\ ?S1 n - \ ?S2 n) < r \).
qed

then have \( \text{Zfun} (\lambda n. \sum \ ?g (\ ?S1 n - \ ?S2 n)) \) sequentially
apply (rule \text{Zfun-le} [\text{rule-format}])
apply (simp only: \text{norm-sum-f})
apply (rule \text{order-trans} [\text{OF norm-sum sum-mono}])
apply (auto simp add: \text{norm-mult-ineq})
done

then have \( 2 : (\lambda n. \sum \ ?g (\ ?S1 n) - \sum \ ?g (\ ?S2 n)) \longrightarrow 0 \)
unfolding \text{tendsto-Zfun-iff diff-0-right}
by (simp only: \text{sum-diff finite-S1 S2-le-S1})

with \( 1 \) have \( (\lambda n. \sum \ ?g (\ ?S2 n)) \longrightarrow (\sum k. \ a k) * (\sum k. \ b k) \)
by (rule \text{Lim-transform2})
then show \( ?\text{thesis} \)
by (simp only: \text{sums-def sum.triangle-reindex})

qed

lemma \text{Cauchy-product}:
fixes \( a b :: \text{nat} \Rightarrow 'a::{\text{real-normed-algebra,banach}} \)
assumes \( \text{summable} (\lambda k. \text{norm} (\ a k) ) \)
and \( \text{summable} (\lambda k. \text{norm} (\ b k)) \)
shows \( (\sum k. \ a k) * (\sum k. \ b k) = (\sum k. \sum i \leq k. \ a i * b (k - i)) \)
using \( \text{assms} \) by (rule \text{Cauchy-product-sums} \text{[THEN sums-unique]})

lemma \text{summable-Cauchy-product}:
fixes \( a b :: \text{nat} \Rightarrow 'a::{\text{real-normed-algebra,banach}} \)
assumes \( \text{summable} (\lambda k. \text{norm} (\ a k) ) \)
and \( \text{summable} (\lambda k. \text{norm} (\ b k)) \)
shows \( \text{summable} (\lambda k. \sum i \leq k. \ a i * b (k - i)) \)

107.12 Series on reals

lemma summable-norm-comparison-test:
\exists N. \forall n\geq N. \text{norm}(f n) \leq g n \implies \text{summable } g \implies \text{summable } (\lambda n. \text{norm}(f n))
by (rule summable-comparison-test) auto

lemma summable-rabs-comparison-test: \exists N. \forall n\geq N. |f n| \leq g n \implies \text{summable } g \implies \text{summable } (\lambda n. |f n|)
for f :: nat \Rightarrow real
by (rule summable-comparison-test) auto

lemma summable-rabs-cancel: \exists N. \forall n\geq N. |f n| \leq g n \implies \text{summable } g \implies \text{summable } (\lambda n. |f n|)
for f :: nat \Rightarrow real
by (rule summable-norm-cancel) simp

lemma summable-rabs:
\exists N. \forall n\geq N. |f n| \implies |\text{suminf } f| \leq (\sum_n |f n|)
for f :: nat \Rightarrow real
by (fold real-norm-def) (rule summable-norm)

lemma summable-zero-power [simp]: \exists N. \forall n\geq N. 0^n = (\lambda n. if n = 0 then 0 ^ 0 else 0)
by (intro ext) (simp add: zero-power)

ultimately show \?thesis by simp
qed

lemma summable-zero-power' [simp]: \exists N. \forall n\geq N. f n * 0^n = (\lambda n. if n = 0 then 0 ^ 0 else 0)
by (intro ext) (simp add: zero-power)

ultimately show \?thesis by simp
qed

lemma summable-power-series:
fixes z :: real
assumes le-1: \forall i. f i \leq 1
and nonneg: \forall i. 0 \leq f i
and z: 0 \leq z \& z < 1
shows \exists N. \forall i. f n * z^n \leq z^n
proof (rule summable-comparison-test[OF - summable-geometric])
show \forall n. \exists N. \forall n\geq N. \text{norm}(f n * z^n) \leq z^n
using z by (auto simp: less_imp_le)
show \forall n. \exists N. \forall na\geq N. \text{norm}(f na * z^n) \leq z^n
using z
by (auto intro!: extI[af - 0] mult-left-le-one-le simp: abs-mult nonneg power-abs)
lemma summable-0-powser:summable (λn.f n * 0 ^ n :: 'a::real-normed-div-algebra)
proof -
  have A: (λn.f n * 0 ^ n) = (λn.if n = 0 then f n else 0)
      by (intro ext) auto
  then show ?thesis
      by (subst A) simp-all
qed

lemma summable-powser-split-head:
  summable (λn.f (Suc n) * z ^ n :: 'a::real-normed-div-algebra)
    = summable (λn.f n * z ^ n)
proof -
  have summable (λn.f (Suc n) * z ^ n)
    = summable (λn.f (Suc n) * z ^ Suc n)
    (is ?lhs <-> ?rhs)
    proof
      show ?rhs if ?lhs
        using summable-mult2[OF that, of z]
        by (simp add: power-commutes algebra-simps)
      show ?lhs if ?rhs
        using summable-mult2[OF that, of inverse z]
        by (cases z ≠ 0, subst (asm) power-Suc2) (simp-all add: algebra-simps)
    qed
  also have ... = summable (λn.f n * z ^ n)
    by (rule summable-Suc-iff)
  finally show ?thesis .
qed

lemma summable-powser-ignore-initial-segment:
  fixes f :: nat ⇒ 'a::{real-normed-div-algebra,banach}
  assumes summable (λn.f n * z ^ n)
  shows suminf (λn.f n * z ^ n) = f 0 + suminf (λn.f (Suc n) * z ^ n) * z
proof (induction m)
  case (Suc m)
  have summable (λn.f (Suc m) * z ^ n)
    = summable (λn.f (Suc m + Suc n) * z ^ n)
    (is ?lhs)
    proof
      show ?rhs if ?lhs
        by (rule Suc.IH)
    qed
    also have ... = summable (λn.f (Suc n + m) * z ^ n)
      by (rule summable-powser-split-head)
    also have ... = summable (λn.f n * z ^ n)
      by (rule Suc.IH)
    finally show ?case .
  qed simp-all
and \( \text{suminf} (\lambda n. f (\text{Suc} n) \cdot z \cdot n) \cdot z = \text{suminf} (\lambda n. f n \cdot z \cdot n) - f 0 \)

and summable \((\lambda n. f (\text{Suc} n) \cdot z \cdot n)\)

proof

from \( \text{assms show} \) summable \((\lambda n. f (\text{Suc} n) \cdot z \cdot n)\)

by \((\text{subst summable-powser-split-head})\)

from \(\text{suminf-mult2[of this, of z]}\)

have \((\sum n. f (\text{Suc} n) \cdot z \cdot n) \cdot z = (\sum n. f (\text{Suc} n) \cdot z \cdot \text{Suc} n)\)

by \((\text{simp add: power-commutes algebra-simps})\)

also from \(\text{assms have} \ldots = \text{suminf} (\lambda n. f n \cdot z \cdot n) - f 0\)

by \((\text{subst suminf-split-head simp-all})\)

finally show \(\text{suminf} (\lambda n. f n \cdot z \cdot n) = f 0 + \text{suminf} (\lambda n. f (\text{Suc} n) \cdot z \cdot n)\)

\(\ast z\)

by simp

then show \(\text{suminf} (\lambda n. f (\text{Suc} n) \cdot z \cdot n) \cdot z = \text{suminf} (\lambda n. f n \cdot z \cdot n) - f 0\)

by simp

qed

lemma summable_partial_sum_bound:

fixes \(f :: \text{nat} \Rightarrow 'a :: \text{banach}\)

and \(e :: \text{real}\)

assumes summable: \(\text{summable} f\)

and \(e > 0\)

obtains \(N\) where \(\forall m n. m \geq N \Rightarrow \text{norm} (\sum k=m..n. f k) < e\)

proof

from \(\text{summable have Cauchy (\lambda n. \sum k<n. f k)}\)

by \((\text{simp add: Cauchy-convergent-iff summable-iff-convergent})\)

from CauchyD \(\text{OF this e} \) obtain \(N\)

where \(N: \forall m n. m \geq N \Rightarrow n \geq N \Rightarrow \text{norm} ((\sum k<m. f k) = (\sum k<n. f k)) < e\)

by blast

have \(\text{norm} (\sum k=m..n. f k) < e\) if \(m: m \geq N\) for \(m n\)

proof \((\text{cases} n \geq m)\)

case \(\text{True}\)

with \(m\) have \(\text{norm} ((\sum k<Suc n. f k) = (\sum k<m. f k)) < e\)

by \((\text{intro N simp-all})\)

also from \(\text{True have} (\sum k<Suc n. f k) = (\sum k<m. f k) = (\sum k=m..n. f k)\)

by \((\text{subst sum-diff [symmetric]}\) \((\text{simp-all add: sum.last-plus})\)

finally show \(?thesis\).

next

case \(\text{False}\)

with \(e\) show \(?thesis\) by simp-all

qed

then show \(?thesis\) by \((\text{rule that})\)

qed

lemma powser_sums_if:

\((\lambda n. (\text{if } n = m \text{ then } 1 \text{ else } 0) \cdot z \cdot n)\) sums \(z'\)

proof

have \((\lambda n. (\text{if } n = m \text{ then } 1 \text{ else } 0) \cdot z \cdot n) = (\lambda n. \text{if } n = m \text{ then } z \cdot n \text{ else } 0)\)
by (intro ext) auto
then show ?thesis
  by (simp add: sums-single)
qed

lemma
  fixes f :: nat ⇒ real
  assumes summable f
  and inj g
  and pos: ∀x. 0 ≤ f x
  shows summable-reindex: summable (f ∘ g)
  and suminf-reindex-mono: suminf (f ∘ g) ≤ suminf f
  and suminf-reindex: (∀x. x /∈ range g =⇒ f x = 0) =⇒ suminf (f ∘ g) = suminf f
proof –
  from ⟨inj g⟩ have [simp]: ∀A. inj-on g A
  by (rule subset-inj-on)
  simp
  have smaller: ∀n. (∑i<n. (f ∘ g) i) ≤ suminf f
  proof
    fix n
    have ∀n' ∈ (g ' {..<n}). n' < Suc (Max (g ' {..<n}))
      by (metis Max-ge finite-imageI finite-lessThan not-le not-less-eq)
    then obtain m where n': ∀n'. n' < n =⇒ g n' < m
      by blast
    have (∑i<n. f (g i)) = sum f (g ' {..<n})
      by (simp add: sum.reindex)
    also have ... ≤ (∑i<m. f i)
      by (rule sum-mono2) (auto simp add: pos n[rule-format!])
    also have ... ≤ suminf f
      using (summable f)
    by (rule sum-le-suminf) (simp-all add: pos)
    finally show (∑i<n. (f ∘ g) i) ≤ suminf f
      by simp
  qed

have incseq (λn. ∑i<n. (f ∘ g) i)
  by (rule incseq-SucI) (auto simp add: pos)
then obtain L where L: (∀n. ∑i<n. (f ∘ g) i) → L
  using smaller by (rule incseq-convergent)
then have (f ∘ g) sums L
  by (simp add: sums-def)
then show summable (f ∘ g)
  by (auto simp add: sums-iff)

then have (∀n. ∑i<n. (f ∘ g) i) → suminf (f ∘ g)
  by (rule summable-LIMSEQ)
then show le: suminf (f ∘ g) ≤ suminf f
by(rule LIMSEQ-le-const2)(blast intro: smaller\[rule-format\])

assume f: \(\forall x. x \notin \text{range}\ g \implies f x = 0\)

from (summable f) have suminf f \leq suminf (f \circ g)
proof (rule suminf-le-const)
  fix n
  have \(\forall n' \in (g^{-1}\{..<n\}). n' < \text{Suc}(\text{Max}(g^{-1}\{..<n\}))\)
  by(auto intro: Max-ge simp add: finite-vimageI less-Suc-eq-le)
  then obtain m where n': \(\forall n'. g n' < n \implies n' < m\)
  by blast
  have \(\sum_{i<n} f i = (\sum_{i\in\{..<n\}} \cap \text{range}\ g, f i)\)
  using f by(auto intro: sum.mono-neutral-cong-right)
  also have \(\ldots = (\sum_{i\in g^{-1}\{..<n\}} (f \circ g) i)\)
  by (rule sum.reindex-cong[where l=g])(auto)
  also have \(\ldots \leq (\sum_{i<m} (f \circ g) i)\)
  by (rule sum-mono2)(auto simp add: pos n)
  also have \(\ldots \leq \text{suminf}(f \circ g)\)
  using (summable (f \circ g)) by (rule sum-le-suminf) (simp-all add: pos)
  finally show \(\sum f \{..<n\} \leq \text{suminf}(f \circ g)\).
qed
with le show \(\text{suminf}(f \circ g) = \text{suminf} f\)
by (rule antisym)
qed

lemma sums-mono-reindex:
  assumes subseq: strict-mono g
  and zero: \(\forall n. n \notin \text{range}\ g \implies f n = 0\)
  shows \((\lambda n. f\ (g\ n))\ \text{sums} \leftrightarrow f\ \text{sums} c\)
unfolding sums-def
proof
  assume lim: \((\lambda n. \sum_{k<n} f\ k) \longrightarrow c\)
  have \((\lambda n. \sum_{k<n} f\ (g\ k)) = (\lambda n. \sum_{k<g\ n} f\ k)\)
  proof
    fix n :: nat
    from subseq have \((\sum_{k<n} f\ (g\ k)) = (\sum_{k\in g^{-1}\{..<n\}} f\ k)\)
    by (subst sum.reindex) (auto intro: strict-mono-imp-inj-on)
    also from subseq have \(\ldots = (\sum_{k<g\ n} f\ k)\)
    by (intro sum.mono-neutral-left ballI zero)
    (auto simp: strict-mono-less-monotone-expr)
    finally show \((\sum_{k<n} f\ (g\ k)) = (\sum_{k<g\ n} f\ k)\).
  qed
  also from LIMSEQ-subseq-LIMSEQ[OF lim subseq] have \(\ldots \longrightarrow c\)
  by (simp only: o-def)
  finally show \((\lambda n. \sum_{k<n} f\ (g\ k)) \longrightarrow c\).
next
  assume lim: \((\lambda n. \sum_{k<n} f\ (g\ k)) \longrightarrow c\)
  define g-inv where g-inv n = (LEAST m. g m \geq n) for n
  from filterlim-subseq[OF subseq] have g-inv-ex: \(\exists m. g m \geq n\) for n
by (auto simp: filterlim-at-top eventually-at-top-linorder)
then have \( g\)-inv: \( g\ (g\text{-inv} \ n) \geq n \) for \( n \)
  unfolding \( g\)-inv-def by (rule LeastI-ex)
have \( g\)-inv-least: \( m \geq g\text{-inv} \ n \) if \( g\ m \geq n \) for \( m \ n \)
  using that unfolding \( g\)-inv-def by (rule Least-le)
have \( g\)-inv-least': \( g\ m < n \) if \( m < g\text{-inv} \ n \) for \( m \ n \)
  using that \( g\)-inv-least[of \( n \ \ m \)] by linarith
have \( (\lambda n. \sum _{k < n} f\ k) = (\lambda n. \sum _{k < g\text{-inv} \ n} f\ (g\ k)) \)
proof
  fix \( n :: \text{nat} \)
  { fix \( k \)
  assume \( k: k \in \{..<n\} \setminus g^-\{..<g\text{-inv} \ n\} \)
  have \( k \notin \text{range } g \)
    proof (rule notI, elim imageE)
      fix \( l \)
      assume \( l: k = g\ l \)
    have \( g\ l < g\ (g\text{-inv} \ n) \)
      by (rule less-le-trans[of \( - g\text{-inv} \)] \( \text{use } k\ l \) in simp-all)
    with subseq have \( l < g\text{-inv} \ n \)
      by (simp add: strict-mono-less)
    with \( k\ l\ ) show False
      by simp
    qed
  then have \( f\ k = 0 \)
    by (rule zero)
  }
  with \( g\)-inv-least' \( g\)-inv have \( (\sum _{k < n} f\ k) = (\sum _{k \in g^-\{..<g\text{-inv} \ n\}} f\ k) \)
    by (intro sum_mono-neutral-right) auto
  also from subseq have \( \ldots = (\sum _{k < g\text{-inv} \ n} f\ (g\ k)) \)
    using strict-mono-imp-inj-on by (subst sum.reindex) simp-all
  finally show \( (\sum _{k < n} f\ k) = (\sum _{k < g\text{-inv} \ n} f\ (g\ k)) \).
qed

also { fix \( K\ n :: \text{nat} \)
  assume \( g\ K \leq n \)
  also have \( n \leq g\ (g\text{-inv} \ n) \)
    by (rule g-inv)
  finally have \( K \leq g\text{-inv} \ n \)
    using subseq by (simp add: strict-mono-less-eq)
}
then have filterlim \( g\text{-inv} \) at-top sequentially
  by (auto simp: filterlim-at-top eventually-at-top-linorder)
with \( \lim \) have \( (\lambda n. \sum _{k < g\text{-inv} \ n} f\ (g\ k)) \longrightarrow c \)
  by (rule filterlim-compose)
finally show \( (\lambda n. \sum _{k < n} f\ k) \longrightarrow c \).
qed

lemma summable-mono-reindex:
THEORY "Series"

assumes subseq: strict-mono g
   and zero: \( \forall n. n \notin \text{range } g \implies f n = 0 \)
shows summable \( (\lambda n. f (g n)) \leftrightarrow \text{summable } f \)
using sums-mono-reindex[of g f, OF assms] by (simp add: summable-def)

lemma suminf-mono-reindex:
fixes f :: nat \Rightarrow 'a::(t2-space,comm-monoid-add)
assumes strict-mono g \( \forall n. n \notin \text{range } g \implies f n = 0 \)
shows \( \text{suminf } (\lambda n. f (g n)) = \text{suminf } f \)
proof (cases summable f)
case True
with sums-mono-reindex[of g f, OF assms]
and summable-mono-reindex[of g f, OF assms]
show ?thesis
  by (simp add: sums-iff)
next
case False
then have \( \neg (\exists c. f \text{ sums } c) \)
  unfolding summable-def by blast
moreover from False have \( \neg \text{summable } (\lambda n. f (g n)) \)
  using summable-mono-reindex[of g f, OF assms] by simp
then have \( \neg (\exists c. (\lambda n. f (g n)) \text{ sums } c) \)
  unfolding summable-def by blast
then have \( \text{suminf } (\lambda n. f (g n)) = \text{The } (\lambda -. \text{False}) \)
  by (simp add: suminf-def)
ultimately show ?thesis by simp
qed

lemma summable-bounded-partials:
fixes f :: nat \Rightarrow 'a::(real-normed-vector,complete-space)
assumes bound: eventually \( (\lambda x0. \forall a \geq x0. \forall b > a. \norm{\sum f \{a<..b\}} \leq g a} \)
sequentially
assumes g: g \( \xrightarrow{-} 0 \)
shows \( \text{summable } f \)
proof (intro Cauchy-convergent CauchyI', goal-cases)
case (1 \( \epsilon \))
with g have eventually \( (\lambda x. |g x| < \epsilon) \) sequentially
  by (auto simp: tendsto iff)
from eventually-conj[OF this bound] obtain x0 where x0:
\( \forall x. x \geq x0 \implies |g x| < \epsilon \)
\( \forall a b. x0 \leq a < b \implies \norm{\sum f \{a<..b\}} \leq g a \)
  unfolding eventually-at-top-linorder by auto
show ?case
proof (intro exI[of - x0] allI impI)
fix m n assume mn: x0 \( \leq m m < n \)
have dist \( (\sum f \{..m\}) (\sum f \{..n\}) = \norm{\sum f \{..n\} - \sum f \{..m\}} \)
by (simp add: dist-norm norm-minus-commute)
also have \( \sum f \{ \ldots \cdot n \} = \sum f \{ \ldots \cdot m \} \)
using \( mn \) by (intro Groups-Big сумму-diff [symmetric]) auto
also have \( \{ \ldots \cdot n \} - \{ \ldots \cdot m \} = \{ m < \ldots \cdot n \} \)
using \( mn \) by auto
also have \( \text{norm} (\sum f \{ m < \ldots \cdot n \}) \leq g \cdot m \)
using \( mn \) by (intro \( z\cdot 0 \)) auto
also have \( \ldots \leq |g \cdot m| \) by simp
also have \( \ldots < \varepsilon \)
using \( mn \) by (intro \( x\cdot 0 \)) auto
finally show \( \text{dist} (\sum f \{ \ldots \cdot m \}) (\sum f \{ \ldots \cdot n \}) < \varepsilon \).
qed
qed
end

108 Differentiation

theory Deriv
  imports Limits
begin

108.1 Frechet derivative

definition has-derivative :: \('a::real-normed-vector \Rightarrow 'b::real-normed-vector\) \Rightarrow
  \('a \Rightarrow 'b\) \Rightarrow 'a filter \Rightarrow bool (infix \( \text{has-}' \cdot \text{derivative} \)) 50
where \( f \text{ has-derivative } f' \) \( F \xleftarrow{} \)
  bounded-linear \( f' \wedge \)
  \( ((\lambda y. ( (f \cdot y - f (\text{Lim } F (\lambda x. x))) - f' \cdot (y - \text{Lim } F (\lambda x. x))) / \text{R norm} (y - \text{Lim } F (\lambda x. x))) \xrightarrow{} 0) \) \( F \)

Usually the filter \( F \) is at \( x \) within \( s \). \( (f \text{ has-derivative } D) \) (at \( x \) within \( s \)) means: \( D \) is the derivative of function \( f \) at point \( x \) within the set \( s \). Where \( s \) is used to express left or right sided derivatives. In most cases \( s \) is either a variable or \( UNIV \).

These are the only cases we’ll care about, probably.

lemma has-derivative-within: \( (f \text{ has-derivative } f') \) (at \( x \) within \( s \)) \( \xleftrightarrow{} \)
  bounded-linear \( f' \wedge ((\lambda y. (1 / \text{norm}(y - x))) \star \text{R} (f \cdot y - (f \cdot x + f' \cdot (y - x)))) \xrightarrow{} 0 \) \( \) (at \( x \) within \( s \))
unfolding has-derivative-def tendsto-iff
by (subst eventually-Lim-ident-at) (auto simp add: field-simps)

lemma has-derivative-eq-rhs: \( (f \text{ has-derivative } f') \) \( F \xrightarrow{} f' = g' \xrightarrow{} (f \text{ has-derivative } g') \) \( F \)
by simp

definition has-field-derivative :: \('a::real-normed-field \Rightarrow 'a\) \Rightarrow \('a \Rightarrow 'a\) filter \Rightarrow bool (infix \( \text{has-}' \cdot \text{field-'}\cdot \text{derivative} \)) 50
where \( f \text{ has-field-derivative } D \) \( F \xleftarrow{} (f \text{ has-derivative } (* D) \) \( F \)
lemma DERIV-cong: (f has-field-derivative X) F \Longrightarrow X = Y \Longrightarrow (f has-field-derivative Y) F
  by simp

definition has-vector-derivative :: (real ⇒ 'b::real-normed-vector) ⇒ 'b ⇒ real filter ⇒ bool
  (infix has'-'vector'-derivative 50)
where (f has-vector-derivative f') net ⇐ (f has-derivative (λx. x *_R f')) net

lemma has-vector-derivative-eq-rhs:
  (f has-vector-derivative X) F \Longrightarrow X = Y \Longrightarrow (f has-vector-derivative Y) F
  by simp

named-theorems derivative-intros structural introduction rules for derivatives
setup :
  let
  val eq-thms = @
    {thms has-derivative-eq-rhs DERIV-cong has-vector-derivative-eq-rhs}
  fun eq-rule thm = get-first (try (fn eq-thm => eq-thm OF [thm])) eq-thms
in
  Global-Theory.add-thms-dynamic
    (binding: derivative-intros),
    fn context =>
      Named-Theorems.get (Context.proof-of context) named-theorems (derivative-intros)
    |> map-filter eq-rule)
end

The following syntax is only used as a legacy syntax.

abbreviation (input)
  FDERIV :: ('a::real-normed-vector ⇒ 'b::real-normed-vector) ⇒ 'a ⇒ ('a ⇒ 'b)
  ⇒ bool
  ((FDERIV (-)/ (-)/ => (-)) [1000, 1000, 60] 60)
where FDERIV f x := f' ≡ (f has-derivative f') (at x)

lemma has-derivative-bounded-linear: (f has-derivative f') F \Longrightarrow bounded-linear f'
  by (simp add: has-derivative-def)

lemma has-derivative-linear: (f has-derivative f') F \Longrightarrow linear f'
  using bounded-linear.linear[OF has-derivative-bounded-linear'] .

lemma has-derivative-ident[derivative-intros, simp]: ((λx. x) has-derivative (λx. x)) F
  by (simp add: has-derivative-def)

lemma has-derivative-id [derivative-intros, simp]: (id has-derivative id) (at a)
  by (metis eq-id-iff has-derivative-ident)

lemma has-derivative-const[derivative-intros, simp]: ((λx. c) has-derivative (λx.
lemma (in bounded-linear) bounded-linear: bounded-linear f ..

lemma (in bounded-linear) has-derivative:
  \( (g \text{ has-derivative } g') F \implies ((\lambda x. f (g x)) \text{ has-derivative } (\lambda x. f (g' x))) F \)

unfolding has-derivative-def
by (auto simp add: bounded-linear-compose OF bounded-linear scaleR diff dest: tendsto)

lemmas has-derivative-scaleR-right [derivative-intros] = bounded-linear.has-derivative [OF bounded-linear-scaleR-right]

lemmas has-derivative-scaleR-left [derivative-intros] = bounded-linear.has-derivative [OF bounded-linear-scaleR-left]

lemmas has-derivative-mult-right [derivative-intros] = bounded-linear.has-derivative [OF bounded-linear-mult-right]

lemmas has-derivative-mult-left [derivative-intros] = bounded-linear.has-derivative [OF bounded-linear-mult-left]

lemmas has-derivative-of-real[derivative-intros, simp] = bounded-linear.has-derivative[OF bounded-linear-of-real]

lemma has-derivative-add[simp, derivative-intros]:
  assumes f: (f has-derivative f') F
      and g: (g has-derivative g') F
  shows ((\lambda x. f x + g x) has-derivative (\lambda x. f' x + g' x)) F

unfolding has-derivative-def
proof safe
let \( ?x = \text{ Lim } F (\lambda x. x) \)
let \( ?D = \lambda y. ((f y - f ?x) - f' (y - ?x)) / \text{norm } (y - ?x) \)
have ((\lambda x. ?D f f' x + ?D g g' x) \longrightarrow (0 + 0)) F
  using f g by (intro tendsto-add) (auto simp: has-derivative-def)
then show ((\lambda x. f x + g x) (\lambda x. f' x + g' x) \longrightarrow 0) F
  by (simp add: field-simps scaleR-add-right scaleR-diff-right)
qed (blast intro: bounded-linear-add f g has-derivative-bounded-linear)

lemma has-derivative-sum[simp, derivative-intros]:
  (\lambda i. i \in I \implies (f i has-derivative f' i) F) \implies
  ((\lambda x. \sum i\in I. f i x) has-derivative (\lambda x. \sum i\in I. f' i x)) F
by (induct I rule: infinite-finite-induct) simp-all

lemma has-derivative-minus[simp, derivative-intros]:
  (f has-derivative f') F \implies ((\lambda x. - f x) has-derivative (\lambda x. - f' x)) F
using has-derivative-scaleR-right[of f f' F -1] by simp
THEORY "Deriv" 2043

lemma has-derivative-diff [simp, derivative-intros]:
(f has-derivative f') F =\>\> (g has-derivative g') F =\>\> 
((\lambda x. f x - g x) has-derivative (\lambda x. f' x - g' x)) F
by (simp only: diff-conv-add-uminus has-derivative-add has-derivative-minus)

lemma has-derivative-at-within:
(f has-derivative f') (at x within s) =\>\> 
(bounded-linear f' \land ((\lambda y. (f y - f x) - f' (y - x)) / R norm (y - x)) ----> 0) (at x within s)
by (cases at x within s = bot) (simp-all add: has-derivative-def Lim-ident-at)

lemma has-derivative-iff-norm:
(f has-derivative f') (at x within s) ----> 
bounded-linear f' \land ((\lambda y. (f y - f x) - f' (y - x)) / norm (y - x)) ----> 0 (at x within s)
using tendsto-norm-zero-iff[of - at x within s, where 'b='b, symmetric]
by (simp add: has-derivative-at-within-diff-conv-add-uminus)

lemma has-derivative-at:
(f has-derivative D) (at x) =\>\> 
(bounded-linear D \land ((\lambda h. (f (x + h) - f x - D * h) / norm h) --\> 0) ----> 0)
unfolding has-derivative-iff-norm LIM-offset-zero-iff[of - x] by simp

lemma field-has-derivative-at:
fixes x :: 'a::real-normed-field
shows (f has-derivative (* D)) (at x) =\>\> (\lambda h. (f (x + h) - f x) / h) --\> 0 ----> D
(is ?lhs = ?rhs)
proof -
have ?lhs = (\lambda h. norm (f (x + h) - f x - D * h) / norm h) --\> 0 ----> 0
by (simp add: bounded-linear-mult-right has-derivative-at)
also have ... = (\lambda y. norm ((f (x + y) - f x - D * y) / y)) --\> 0 ----> 0
by (simp cong: Lim-cong flip: nonzero-norm-divide)
also have ... = (\lambda y. norm ((f (x + y) - f x) / y - D / y * y)) --\> 0 ----> 0
by (simp only: diff-distribute times-distribute eq left (symmetric))
also have ... = ?rhs
by (simp add: tendsto-norm-zero-iff Lim-zero-iff cong: Lim-cong)
finally show ?thesis .
qed

lemma has-derivative-iff-Ex:
(f has-derivative f') (at x) =\>\> 
(bounded-linear f' \land (\exists e. (\forall h. f (x+h) = f x + f' h + e h) \land ((\lambda h. norm (e h) / norm h) --\> 0) (at 0)))
unfolding has-derivative-at by force

lemma has-derivative-at-within-iff-Ex:
assumes x \in S open S
shows (f has-derivative f') (at x within S) =\>\> 
(bounded-linear f' \land (\exists e. (\forall x+h \in S \rightarrow f (x+h) = f x + f' h + e h))
\( ((\lambda h. \text{norm } (e \cdot h) / \text{norm } h) \rightarrow 0) \text{ (at } 0) \) (at 0)

(is \_lhs = \_rhs)

**proof** safe

show bounded-linear \( f' \)

if \( (f \text{ has-derivative } f') \text{ (at } x \text{ within } S) \)

using has-derivative-bounded-linear that by blast

show \( \exists e. \ (\forall h. x + h \in S \rightarrow f(x + h) = f x + f' h + e h) \land (\lambda h. \text{norm } (e \cdot h) / \text{norm } h) \rightarrow 0 \) (at 0)

if \( (f \text{ has-derivative } f') \text{ (at } x \text{ within } S) \)

by (metis (full-types) assms that has-derivative-iff-Ex at-within-open)

**show** \( (f \text{ has-derivative } f') \text{ (at } x \text{ within } S) \)

if bounded-linear \( f' \)

and eq [rule-formal]: \( \forall h. x + h \in S \rightarrow f(x + h) = f x + f' h + e h \)

and \( 0: (\lambda h. \text{norm } (e \cdot (h::'a):'b) / \text{norm } h) -0 \rightarrow 0 \)

for \( e \)

**proof**

have 1: \( f y - f x = f'(y-x) + e(y-x) \) if \( y \in S \) for \( y \)

using eq [of \( y-x \)] that by simp

have 2: \( (\lambda y. \text{norm } (e(y-x)) / \text{norm } (y-x)) \rightarrow 0 \) (at 0)

by (simp add: 0 assms tendsto-offset-zero-iff)

have \( ((\lambda y. \text{norm } (f y - f x - f'(y-x)) / \text{norm } (y-x)) \rightarrow 0) \text{ (at } x \text{ within } S) \)

by (simp add: Lim-cong-within 1 2)

then show ?thesis

by (simp add: has-derivative-iff-norm (bounded-linear \( f' \)))

qed

**lemma** has-derivativeI:

bounded-linear \( f' \implies ((\lambda y. (f y - f x) - f'(y-x)) / \text{norm } (y-x)) \rightarrow 0 \) (at 0)

by (simp add: has-derivative-at-within)

**lemma** has-derivativeI-sandwich:

assumes \( e: 0 < e \)

and bounded: bounded-linear \( f' \)

and sandwich: \( (\forall y. y \in S \implies y \neq x \implies \text{dist } y x < e \implies \) \)

\( \text{norm } ((f y - f x) - f'(y-x)) / \text{norm } (y-x) \leq H y) \)

and \( (H \rightarrow 0) \) (at \( x \) within \( S \))

shows \( (f \text{ has-derivative } f') \text{ (at } x \text{ within } S) \)

**unfolding** has-derivative-iff-norm

**proof** safe

show \( ((\lambda y. \text{norm } (f y - f x - f'(y-x)) / \text{norm } (y-x)) \rightarrow 0) \text{ (at } x \text{ within } S) \)

proof (rule tendsto-sandwich[where \( f=\lambda x. 0 \)])

show \( (H \rightarrow 0) \) (at \( x \) within \( S \)) by fact

show eventually \( (\lambda n. \text{norm } (f n - f x - f'(n-x)) / \text{norm } (n-x) \leq H n) \)

(at \( x \) within \( S \))
unfolding eventually-at using e sandwich by auto
qed (auto simp: le-divide-eq)
qed fact

lemma has-derivative-subset:
(f has-derivative f') (at x within s) \implies t \subseteq s \implies (f has-derivative f') (at x within t)
by (auto simp add: has-derivative-iff-norm intro: tendsto-within-subset)

lemmas has-derivative-within-subset = has-derivative-subset

lemma has-derivative-within-singleton-iff:
(f has-derivative g) (at x within {x}) \iff bounded-linear g
by (auto intro!: has-derivativeI-sandwich[where e=1] has-derivative-bounded-linear)

108.1.1 Limit transformation for derivatives

lemma has-derivative-transform-within:
assumes (f has-derivative f') (at x within s)
and 0 < d
and x \in s
and \( \forall x', x \in s; \ dist \ x' x < d \Rightarrow f x' = g x' \)
shows (g has-derivative f') (at x within s)
using assms
unfolding has-derivative-within
by (force simp add: intro: Lim-transform-within-within)

lemma has-derivative-transform-within-open:
assumes (f has-derivative f') (at x within t)
and open s
and x \in s
and \( \forall x. x \in s \Rightarrow f x = g x \)
shows (g has-derivative f') (at x within t)
using assms unfolding has-derivative-within-within
by (force simp add: intro: Lim-transform-within-within-open)

lemma has-derivative-transform:
assumes x \in s \( \forall x. x \in s \Rightarrow g x = f x \)
assumes (f has-derivative f') (at x within s)
shows (g has-derivative f') (at x within s)
using assms
by (intro has-derivative-transform-within[OF - zero-less-one, where g=g]) auto

lemma has-derivative-transform-eventually:
assumes (f has-derivative f') (at x within s)
(\( \forall x' \ in \ at \ x \ within \ s. \ f x' = g x' \))
assumes f x = g x \in s
shows (g has-derivative f') (at x within s)
using assms
proof  
  from assms(2,3) obtain d where d > 0 \( \forall x'. x' \in s \implies \text{dist } x x' < d \implies f x' = g x' \)  
    by (force simp: eventually-at)  
  from has-derivative-transform-within[OF assms(1) this(1) assms(4) this(2)]  
  show ?thesis .  
qed

lemma has-field-derivative-transform-within:  
  assumes \( (f \text{ has-field-derivative } f') \ (at \ a \ within S) \)  
  and \( 0 < d \)  
  and \( a \in S \)  
  and \( \forall x. \ [x \in S; \ \text{dist } x a < d] \implies f x = g x \)  
  shows \( (g \text{ has-field-derivative } f') \ (at \ a \ within S) \)  
using assms unfolding has-field-derivative-def  
by (metis has-derivative-transform-within)

lemma has-field-derivative-transform-within-open:  
  assumes \( (f \text{ has-field-derivative } f') \ (at \ a) \)  
  and \( \text{open } S \ a \in S \)  
  and \( \forall x. \ x \in S \implies f x = g x \)  
  shows \( (g \text{ has-field-derivative } f') \ (at \ a) \)  
using assms unfolding has-field-derivative-def  
by (metis has-derivative-transform-within-open)

\section{108.2 Continuity}  

lemma has-derivative-continuous:  
  assumes \( f: (f \text{ has-derivative } f') \ (at \ x \ within S) \)  
  shows \( \text{continuous } (at \ x \ within S) \ f \)  
proof  
  from \( f \) interpret \( F: \text{bounded-linear } f' \)  
  by (rule has-derivative-bounded-linear)  
  note \( F.\text{tendsto[tendsto-intros]} \)  
  let \( \?L = \lambda f. (f \longrightarrow 0) \ (at \ x \ within S) \)  
  have \( \?L \ (\lambda y. \text{norm } ((f y - f x) - f' (y - x)) / \text{norm } (y - x)) \)  
    using \( f \) unfolding has-derivative-iff-norm by blast  
  then have \( \?L \ (\lambda y. \text{norm } ((f y - f x) - f' (y - x)) / \text{norm } (y - x) * \text{norm } (y - x)) \)  
    (is \( ?m \))  
    by (rule tendsto-mult-zero) (auto intro!: tendsto-eq-intros)  
  also have \( ?m \longrightarrow \?L \ (\lambda y. \text{norm } ((f y - f x) - f' (y - x))) \)  
    by (intro filterlim-cong) (simp-all add: eventually-at-filter)  
  finally have \( \?L \ (\lambda y. (f y - f x) - f' (y - x)) \)  
    by (rule tendsto-norm-zero-cancel)  
  then have \( \?L \ (\lambda y. (f y - f x) - f' (y - x) + f' (y - x)) \)  
    by (rule tendsto-eq-intros) (auto intro!: tendsto-eq-intros simp: F.zero)  
  then have \( \?L \ (\lambda y. f y - f x) \)  
    by simp  
  from tendsto-add[OF this tendsto-const, of f x] show \( ?thesis \)
by (simp add: continuous-within)

qed

108.3 Composition

lemma tendsto-at-iff-tendsto-nhds-within:
\[ f(x) = y \Rightarrow (f \rightarrow y) \text{ (at } x \text{ within } s) \leftrightarrow (f \rightarrow y) \text{ (inf } \text{nhd} x \text{ (principal } s)) \]

unfolding tendsto-def eventually-inf-principal eventually-at-filter
by (intro ext all-cong imp-cong) (auto elim!: eventually-mono)

lemma has-derivative-in-compose:
\[ \text{assumes } f: (f \text{ has-derivative } f') \text{ (at } x \text{ within } s) \]
\[ \text{and } g: (g \text{ has-derivative } g') \text{ (at } f(x) \text{ within } (f's)) \]
\[ \text{shows } (λ x. g(f x)) \text{ has-derivative } (λ x. g'(f' x)) \text{ (at } x \text{ within } s) \]

proof -
from \( f \) interpret \( F: \text{bounded-linear } f' \)
by (rule has-derivative-bounded-linear)
from \( g \) interpret \( G: \text{bounded-linear } g' \)
by (rule has-derivative-bounded-linear)
from \( F \) bounded obtain \( kF \) where \( kF: \text{\( ☐ \)(∀ x. norm } (f' x) \leq norm x * kF \) \)
by fast
from \( G \) bounded obtain \( kG \) where \( kG: \text{\( ☐ \)(∀ x. norm } (g' x) \leq norm x * kG \) \)
by fast

note \( G.\text{tendsto}[\text{tendsto-intros}] \)
let \( ?L = λ f. (f \rightarrow 0) \text{ (at } x \text{ within } s) \)
let \( ?D = λ f x y. (f y - f x) - f'(y - x) \)
let \( ?N = λ f x y. \text{norm } (?D f x y)/\text{norm } (y - x) \)
let \( ?gf = λ x. g(f x) \) and \( ?gf' = λ x. g'(f' x) \)
define \( Nf \) where \( Nf = ?N f f' x \)
define \( Ng \) where \( [\text{abs-def}]: Ng y = ?N g g'(f x) (f y) \text{ for } y \)

show \( ?\text{thesis} \)
proof (rule has-derivativeI-sandwich[of 1])
show \( \text{bounded-linear } (λ x. g'(f' x)) \)
using \( fg \) by (blast intro: bounded-linear-compose has-derivative-bounded-linear)
next
fix \( y :: 'a \)
assume \( \neg y \neq x \)
have \( ?N ?gf ?gf' x y = \text{norm } (g'(?f f' x y) + ?D g g'(f x) (f y))/\text{norm } (y - x) \)
by (simp add: G.diff G.add field-simps)
also have \( \ldots \leq \text{norm } (g'(f f' x y)))/\text{norm } (y - x) + Ng y * (\text{norm } (f y - f x))/\text{norm } (y - x) \)
by (simp add: add-divide-distrib[symmetric] divide-right-mono norm-triangle-ineq G.zero Ng-def)
also have \( \ldots \leq Nf y * kG + Ng y * (Nf y + kF) \)
proof (intro add-mono mult-left-mono)
have \( \| f - f \| = \| D f f' x y + f' (y - x) \| \) by simp
also have \( \ldots \leq \| D f f' x y \| + \| f' (y - x) \| \)
by (rule norm-triangle-ineq)
also have \( \ldots \leq \| D f f' x y \| + \| y - x \| * kF \)
using \( kF \) by (intro add-mono) simp
finally show \( \| f - f \| / \| y - x \| \leq Nf y + kF \)
by (simp add: neq Nf-def field-simps)
qed (use \( kG \) in (simp-all add: \( Ng-def Nf-def neq zero-le-divide-iff field-simps \))
finally show \( \forall N \exists g \exists f' \exists x \exists y \leq Nf y + kG + Ng y * (Nf y + kF) \).
next
have [tendsto-intros]: \( ?L Nf \)
using \( f \) unfolding has-derivative-iff-norm Nf-def ..
from \( f \) have \( f (x) \to 0 \) (at \( x \) within \( s \))
by (blast intro: has-derivative-continuous continuous-within\([THEN \ iffD1]\])
then have \( f' (\lim x at x within s. f x :> inf (nhds (f x)) (principal (f'))\)
unfolding filterlim-def
by (simp add: eventually-filtermap eventually-at-filter le-principal)
have \( ((?N g g' (f x)) \to 0) \) (at \( f \) within \( f' \))
using \( g \) unfolding has-derivative-iff-norm ..
then have \( g' : ((?N g g' (f x)) \to 0) \) (inf (nhds (f x)) (principal (f')))
by (rule tendsto-at-iff-tendsto-nhds-within\([THEN \ iffD1, rotated]\)) simp
have [tendsto-intros]: \( ?L Ng \)
unfolding Ng-def by (rule filterlim-compose\([OF \ g' f']\))
show \( ((\lambda y. Nf y * kG + Ng y * (Nf y + kF)) \to 0) \) (at \( x \) within \( s \))
by (intro tendsto-eq-intros) auto
qed simp
qed

lemma has-derivative-compose:
\( (f has-derivative f') (at x within s) \to (g has-derivative g') (at (f x)) \to \)
\( ((\lambda x. g (f x)) has-derivative (\lambda x. g' (f' x))) (at x within s) \)
by (blast intro: has-derivative-in-compose has-derivative-subset)

lemma has-derivative-in-compose2:
assumes \( f x, x t \to (g has-derivative g' x) (at x within t) \)
assumes \( f' s \subseteq t x s \)
assumes \( f has-derivative f' (at x within s) \)
shows \( ((\lambda x. g (f x)) has-derivative (\lambda y. g' (f x) (f' y))) (at x within s) \)
using assms
by (auto intro: has-derivative-within-subset intro!: has-derivative-in-compose[OF \( f f' x s g \)])

lemma (in bounded-bilinear) FDERIV:
assumes \( f : (f has-derivative f') (at x within s) \) and \( g : (g has-derivative g') (at x within s) \)
shows \( ((\lambda x. f x ** g x) has-derivative (\lambda h. f x ** g' h + f' h ** g x)) (at x \)
within s)
proof -
  from bounded-linear.bounded [OF has-derivative-bounded-linear [OF f]]
  obtain KF where norm-F: \( \forall x. \|f'(x)\| \leq \|x\| K F \) by fast

  from pos-bounded obtain K
   where K: \( 0 < K \) and norm-prod: \( \forall a b. \|a \cdot b\| \leq \|a\| \cdot \|b\| \)
   * K
     by fast
  let \( ?D = \lambda f' y. f y - f x - f'(y - x) \)
  let \( ?N = \lambda f' y. \|?D f' y\| / \|y - x\| \)
  define Ng where Ng = ?N g g'
  define Nf where Nf = ?N f f'

  let \( ?\text{fun1} = \lambda y. \|f y \cdot g - f x \cdot g - (f x \cdot g' (y - x) + f'(y - x)) \cdot g(x)\| / \|y - x\| \)
  let \( ?\text{fun2} = \lambda y. \|f x\| \cdot \|N f y + N g y \cdot K + K F \cdot \|g y - g x\| \cdot K\| \)
  let \( ?F = at x within s \)

  show ?thesis
  proof (rule has-derivativeI-sandwich[of 1])
    show bounded-linear (\( \lambda x. \|f f' h + f' h \cdot g\| \cdot g(x)\))
      by (intro bounded-linear-add-bound)
        bounded-linear-compose [OF bounded-linear-right] bounded-linear-compose
        [OF bounded-linear-left]
        has-derivative-bounded-linear [OF g] has-derivative-bounded-linear [OF f])
  next
  from g have \( (g \rightarrow g x) ?F \)
    by (intro continuous-within[THEN iffD1] has-derivative-continuous)
  moreover from f g have \( (N g \rightarrow 0) ?F (N f \rightarrow 0) ?F \)
    by (simp-all add: has-derivative-iff-norm Ng-def Nf-def)
  ultimately have \( (?\text{fun2} \rightarrow \|f x\| \cdot \|g y - g x\| \cdot K + \|f\| \cdot \|g y\| \cdot K + K F \cdot \|g y - g x\| \cdot K) ?F \)
    by (intro tendsto-intros) (simp-all add: LIM-zero-iff)
  then show \( (?\text{fun2} \rightarrow 0) ?F \)
    by simp
  next
  fix y :: 'd
  assume \( y \neq x \)
  have \( ?\text{fun1} y = \)
    \( \|f x \cdot g - f y \cdot g' + ?D f f' y \cdot g y + f'(y - x) \cdot g(y - x)\| / \|g(x)\| \)
  norm (\( y - x \))
    by (simp add: diff-left diff-right add-left add-right field-simps)
  also have \( \ldots \leq \|f x\| \cdot \|f y\| \cdot K + \|f\| \cdot \|g y\| \cdot K + K F \cdot \|g y - g x\| \cdot K \)
    by (intro divide-right-mono mult-mono')
    order-trans [OF norm-triangle-ineq add-mono]
also have ... = ?fun2 y
  by (simp add: add-divide-distrib Ng-def Nf-def)
finally show ?fun1 y ≤ ?fun2 y.
qed simp

lemma has-derivative-mult[simp, derivative-intros] = bounded-bilinear.FDERIV[OF bounded-bilinear-mult]
lemma has-derivative-scaleR[simp, derivative-intros] = bounded-bilinear.FDERIV[OF bounded-bilinear-scaleR]

lemma has-derivative-prod[simp, derivative-intros]:
  fixes f :: 'a::real-normed-vector ⇒ 'b::real-normed-field
  shows ((λx. ∏ i∈I. f i x) has-derivative (λy. ∑ i∈I. f' i y * (∏ j∈I - {i}. f j x))) (at x within S)
  proof (induct I rule: infinite-finite-induct)
    case empty
    then show ?case by simp
  next
    case (insert i I)
    let ?P = λy. f i x * (∑ i∈I. f' i y * (∏ j∈I - {i}. f j x)) + (f' i y) * (∏ i∈I. f i x)
    have ((λx. f i x * (∏ i∈I. f i x)) has-derivative ?P) (at x within S)
      using insert by (intro has-derivative-mult) auto
    also have ?P = (λy. ∑ i∈insert i I. f' i y * (∏ j∈insert i I - {i'}. f j x))
      using insert(1,2)
      by (auto simp add: sum-distrib-left insert-Diff-if intro: ext sum.cong)
    finally show ?case
      using insert by simp
  qed

lemma has-derivative-power[simp, derivative-intros]:
  fixes f :: 'a::real-normed-vector ⇒ 'b::real-normed-field
  assumes f :: (f has-derivative f') (at x within S)
  shows ((λx. f x ^ n) has-derivative (λy. of-nat n * f' y * f x ^ (n - 1))) (at x within S)
  using has-derivative-prod[OF f, of {..< n}] by (simp add: prod-constant ac-simps)

lemma has-derivative-inverse':
  fixes x :: 'a::real-normed-div-algebra
  assumes x ∉ 0
  shows (inverse has-derivative (λh. - (inverse x * h * inverse x))) (at x within
\textsc{THEORY “Deriv”}

\begin{quote}
S)
\begin{itemize}
\item \textbf{lemma} \texttt{has-derivative-inverse}: \texttt{simp, derivative-intros}:
\begin{itemize}
\item \textbf{fixes} \texttt{f : -} \Rightarrow \texttt{‘a::real-normed-div-algebra}
\item \textbf{assumes} \texttt{x: f x \neq 0}
\item \textbf{and} \texttt{f: (f has-derivative f') (at x within S)}
\item \textbf{shows} \texttt{((λx. inverse (f x)) has-derivative (λh. \texttt{−} inverse (f x) * f' h * inverse (f x)))}
\item \texttt{(at x within S)}
\item \textbf{using} \texttt{has-derivative-compose[OF f has-derivative-inverse', OF x]}.
\end{itemize}
\end{itemize}
\end{quote}

\begin{quote}
\begin{itemize}
\item \textbf{next}
\item \textbf{fix} \texttt{y :: ‘a}
\item \textbf{assume} \texttt{h: y \neq x dist y x < norm x}
\item \textbf{then have} \texttt{y \neq 0} \texttt{by auto}
\item \textbf{have} \texttt{norm (inverse y \texttt{−} inverse x \texttt{−} ?f (y \texttt{−} x)) / norm (y \texttt{−} x)}
\item \texttt{= norm (\texttt{−} (inverse y \texttt{∗} (y \texttt{−} x) \texttt{∗} inverse x \texttt{−} inverse x \texttt{∗} (y \texttt{−} x) \texttt{∗} inverse x) x) / norm (y \texttt{−} x)}
\item \textbf{by} \texttt{simp add: (y \neq 0), inverse-diff-inverse x}
\item \textbf{also have} \texttt{... = norm (inverse y \texttt{−} inverse x) \texttt{∗} (y \texttt{−} x) \texttt{∗} inverse x / norm (y \texttt{−} x)}
\item \textbf{by} \texttt{simp add: left-diff-distrib norm-minus-commute}
\item \textbf{also have} \texttt{... \leq norm (inverse y \texttt{−} inverse x) \texttt{∗} norm (y \texttt{−} x) \texttt{∗} norm (inverse x)}
\item \textbf{by} \texttt{simp}
\item \textbf{finally show} \texttt{norm (inverse y \texttt{−} inverse x \texttt{−} ?f (y \texttt{−} x)) / norm (y \texttt{−} x) \leq}
\item \texttt{norm (inverse y \texttt{−} inverse x) \texttt{∗} norm (inverse x)}.
\item \textbf{qed}
\end{itemize}
\end{quote}

\begin{quote}
\begin{itemize}
\item \textbf{lemma} \texttt{has-derivative-divide[simp, derivative-intros]}:
\item \textbf{fixes} \texttt{f :: -} \Rightarrow \texttt{‘a::real-normed-div-algebra}
\item \textbf{assumes} \texttt{x: g x \neq 0}
\item \textbf{and} \texttt{f: (f has-derivative f') (at x within S)}
\item \textbf{and} \texttt{g: (g has-derivative g') (at x within S)}
\item \textbf{assumes} \texttt{x: g x \neq 0}
\item \textbf{shows} \texttt{((λx. f x / g x) has-derivative}
\item \texttt{((λx. f x \texttt{∗} (inverse (g x) \texttt{∗} g' h * inverse (g x)) + f' h / g x)) (at x within S)}
\end{itemize}
\end{quote}
using has-derivative-mult[OF f has-derivative-inverse[OF x g]]
by (simp add: field-simps)

Conventional form requires mult-AC laws. Types real and complex only.

lemma has-derivative-divide[derivative-intros]:
  fixes f :: - a :: real-normed-field
  assumes f: (f has-derivative f') (at x within S)
          and g: (g has-derivative g') (at x within S)
          and x: g x ≠ 0
  shows ((λx. f x / g x) has-derivative (λh. (f' h * g x - f x * g' h) / (g x * g x))) (at x within S)
proof
  have f' h / g x - f x * (inverse (g x) * g' h * inverse (g x)) =
    (f' h * g x - f x * g' h) / (g x * g x) for h
    by (simp add: field-simps x)
  then show ?thesis
    using has-derivative-divide [OF f g] x
    by simp
qed

108.4 Uniqueness

This can not generally shown for (has-derivative), as we need to approach
the point from all directions. There is a proof in Analysis for euclidean-space.

lemma has-derivative-at2: (f has-derivative f') (at x) ←→
  bounded-linear f' ∧ ((λy. (1 / (norm(y - x))) *R (f y - (f x + f' (y - x))))
  ----→ 0) (at x)
  using has-derivative-within [of f f' x UNIV]
  by simp

lemma has-derivative-zero-unique:
  assumes ((λx. 0) has-derivative F) (at x)
  shows F = (λh. 0)
proof
  interpret F:: bounded-linear F
  using assms by (rule has-derivative-bounded-linear)
  let ?r = λh. norm (F h) / norm h
  have *: ?r → 0
  using assms unfolding has-derivative-at by simp
  show F = (λh. 0)
  proof
    show F h = 0 for h
    proof (rule ccontr)
      assume **: ¬ ?thesis
      then have h: h ≠ 0
        by (auto simp add: F.zero)
      with ** have 0 < ?r h
        by simp
      from LIM-D [OF * this] obtain S
where $S$: $0 < S$ and $r$: $\forall x. x \neq 0 \implies \text{norm } x < S \implies ?r x < ?r h$

by auto
from dense [OF $S$] obtain $t$ where $t$: $0 < t \land t < S$ ..
let $?x = \text{scaleR} \left( t / \text{norm } h \right) h$
have $?x \neq 0$ and $\text{norm } ?x < S$
  using $t$ $h$ by simp-all
then have $?r ?x < ?r h$
    by (rule $r$)
then show False
  using $t$ $h$ by (simp add: F.scaleR)
qed

lemma has-derivative-unique:
  assumes (f has-derivative F) (at x)
  and (f has-derivative F') (at x)
  shows $F = F'$
proof -
  have $(\lambda x. 0)$ has-derivative $(\lambda h. F h - F' h)$ (at x)
    using has-derivative-diff [OF assms] by simp
  then have $(\lambda h. F h - F' h) = (\lambda h. 0)$
    by (rule has-derivative-zero-unique)
  then show $F = F'$
    unfolding fun-eq iff right-minus-eq .
qed

108.5 Differentiability predicate

definition differentiable :: ('a::real-normed-vector => 'b::real-normed-vector) => 'a filter => bool
  (infix differentiable 50)
where f differentiable F = ($\exists D. (f has-derivative D) F$)

lemma differentiable-subset:
  f differentiable (at x within s) = t $\subseteq$ s $\implies$ f differentiable (at x within t)
unfolding differentiable-def by (blast intro: has-derivative-subset)

lemmas differentiable-within-subset = differentiable-subset

lemma differentiable-ident [simp, derivative-intros]: (\lambda x. x) differentiable F
unfolding differentiable-def by (blast intro: has-derivative-ident)

lemma differentiable-const [simp, derivative-intros]: (\lambda z. a) differentiable F
unfolding differentiable-def by (blast intro: has-derivative-const)

lemma differentiable-in-compose:
  f differentiable (at (g x) within (g's)) $\implies$ g differentiable (at x within s) $\implies$
  (\lambda x. f (g x)) differentiable (at x within s)
unfolding differentiable-def by (blast intro: has-derivative-in-compose)

lemma differentiable-compose:
f differentiable (at (g x)) => g differentiable (at x within s) =>
(\lambda x. f (g x)) differentiable (at x within s)
by (blast intro: differentiable-in-compose differentiable-subset)

lemma differentiable-add [simp, derivative-intros]:
f differentiable F => g differentiable F => (\lambda x. f x + g x) differentiable F
unfolding differentiable-def by (blast intro: has-derivative-add)

lemma differentiable-sum [simp, derivative-intros]:
assumes finite s \forall a \in s. (f a) differentiable net
shows (\lambda x. sum (\lambda a. f a x) s) differentiable net
proof -
  from bchoice[OF assms(2)[unfolded differentiable-def]]
  show ?thesis
    by (auto intro!: has-derivative-sum simp: differentiable-def)
qed

lemma differentiable-minus [simp, derivative-intros]:
f differentiable F => (\lambda x. - f x) differentiable F
unfolding differentiable-def by (blast intro: has-derivative-minus)

lemma differentiable-diff [simp, derivative-intros]:
f differentiable F => g differentiable F => (\lambda x. f x - g x) differentiable F
unfolding differentiable-def by (blast intro: has-derivative-diff)

lemma differentiable-mult [simp, derivative-intros]:
fixes f g :: 'a::real-normed-vector => 'b::real-normed-algebra
shows f differentiable (at x within s) => g differentiable (at x within s) =>
(\lambda x. f x * g x) differentiable (at x within s)
unfolding differentiable-def by (blast intro: has-derivative-mult)

lemma differentiable-inverse [simp, derivative-intros]:
fixes f :: 'a::real-normed-vector => 'b::real-normed-field
shows f differentiable (at x within s) => f x \neq 0 =>
(\lambda x. inverse (f x)) differentiable (at x within s)
unfolding differentiable-def by (blast intro: has-derivative-inverse)

lemma differentiable-divide [simp, derivative-intros]:
fixes f g :: 'a::real-normed-vector => 'b::real-normed-field
shows f differentiable (at x within s) => g differentiable (at x within s) =>
g x \neq 0 => (\lambda x. f x / g x) differentiable (at x within s)
unfolding divide-inverse by simp

lemma differentiable-power [simp, derivative-intros]:
fixes f g :: 'a::real-normed-vector => 'b::real-normed-field
shows f differentiable (at x within s) => (\lambda x. f x ^ n) differentiable (at x within
unfolding differentiable-def by (blast intro: has-derivative-power)

lemma differentiable-scaleR [simp, derivative-intros]:
\( f \) differentiable (at \( x \) within \( s \)) \( \Rightarrow \) \( g \) differentiable (at \( x \) within \( s \))

unfolding differentiable-def by (blast intro: has-derivative-scaleR)

lemma has-derivative-imp-has-field-derivative:
\( (f \text{ has-derivative } D) \) \( \Rightarrow \) \( \forall x. (x \ast R g x) \text{ differentiable (at } x \text{ within } s) \)

unfolding has-field-derivative-def by (rule has-derivative-eq-rhs[of \( f D \)]) (simp-all add: fun-eq-iff mult.commute)

lemma has-field-derivative-imp-has-derivative:
\( (f \text{ has-field-derivative } D) \) \( \Rightarrow \) \( (f \text{ has-derivative } (\ast) \) \( D) \) \( F \)

by (simp add: has-field-derivative-def)

lemma DERIV-subset:
\( (f \text{ has-field-derivative } f') \) (at \( x \) within \( s \)) \( \Rightarrow \) \( t \subseteq s \)

by (simp add: has-field-derivative-def has-derivative-within-subset)

lemma has-field-derivative-at-within:
\( (f \text{ has-field-derivative } f') \) (at \( x \)) \( \Rightarrow \) \( (f \text{ has-field-derivative } f') \) (at \( x \) within \( s \))

using DERIV-subset by blast

abbreviation (input)
\( \text{DERIV} : ('a::real-normed-field \Rightarrow 'a) \Rightarrow 'a \Rightarrow bool \)
\((\text{DERIV} (-)/ (-)/ :: (-)) [1000, 1000, 60] 60) \)
where DERIV \( f x := D \equiv (f \text{ has-field-derivative } D) \) (at \( x \))

abbreviation has-real-derivative :: (real \Rightarrow real) \Rightarrow real \Rightarrow real filter \Rightarrow bool
\( \) (infix (has-'real'-derivative) 50)
where \( (f \text{ has-real-derivative } D) \) \( F \equiv (f \text{ has-field-derivative } D) \) \( F \)

lemma real-differentiable-def:
\( f \) differentiable at \( x \) within \( s \) \( \iff \) \( \exists D. (f \text{ has-real-derivative } D) \) (at \( x \) within \( s \))

proof safe
assume \( f \) differentiable at \( x \) within \( s \)
then obtain \( f' \) where \( \ast : (f \text{ has-derivative } f') \) (at \( x \) within \( s \))

unfolding differentiable-def by auto
then obtain \( c \) where \( f' = (\ast) \) \( c \)

by (metis real-bounded-linear has-derivative-bounded-linear mult.commute fun-eq-iff)
with \( \ast \) show \( \exists D. (f \text{ has-real-derivative } D) \) (at \( x \) within \( s \))

unfolding has-field-derivative-def by auto
qed (auto simp: differentiable-def has-field-derivative-def)

lemma real-differentiableE [elim?]:
assumes \( f : f \) differentiable (at \( x \) within \( s \))
obtains $df$ where ($f$ has-real-derivative $df$) (at $x$ within $s$) using assms by (auto simp: real-differentiable-def)

lemma has-field-derivative-iff:
($f$ has-field-derivative $D$) (at $x$ within $S$) $\iff$ $((\lambda y. (f y - f x) / (y - x)) \longrightarrow D)$ (at $x$ within $S$)

proof
have $((\lambda y. \text{norm} (f y - f x - D * (y - x)) / \text{norm} (y - x)) \longrightarrow 0)$ (at $x$ within $S$)
apply (subst tendsto-norm-zero-iff [symmetric], rule filterlim-cong)
apply (simp-all add: eventually-at-filter field-simps nonzero-norm-divide)
done
then show ?thesis by (simp add: has-field-derivative-def has-derivative-iff-norm bounded-linear-mult-right LIM-zero-iff)
qed

lemma DERIV-def: $\text{DERIV } f \ x :> D \iff (\lambda h. (f (x + h) - f x) / h) \rightarrow 0 \rightarrow D$
unfolding field-has-derivative-at has-field-derivative-def has-field-derivative-iff ..

lemma mult-commute-ords: $(\lambda x. x \ast c) = (\ast) \ c$
for $c :: 'a::ab-semigroup-mult$
by (simp add: fun-eq_iff mult.commute)

lemma DERIV-compose-FDERIV:
fixes $f :: \text{real} \Rightarrow \text{real}$
assumes $\text{DERIV } f \ g \ x :> f'$
assumes ($g$ has-derivative $g'$) (at $x$ within $s$)
shows $((\lambda x. f (g x)) \text{ has-derivative } (\lambda x. g' x \ast f'))$ (at $x$ within $s$)
using assms has-derivative-compose[of $g$ $g'$ $x$ $s$ $\ast f'$]
by (auto simp: has-field-derivative-def ac-simps)

108.6 Vector derivative

lemma has-field-derivative-iff-has-vector-derivative:
($f$ has-field-derivative $y$) $F \iff$ ($f$ has-vector-derivative $y$) $F$
unfolding has-vector-derivative-def has-field-derivative-def real-scaleR_def mult-commute-ords ..

lemma has-field-derivative-subset:
($f$ has-field-derivative $y$) (at $x$ within $s$) $\Rightarrow$ $t \subseteq s \Rightarrow$
($f$ has-field-derivative $y$) (at $x$ within $t$)
unfolding has-field-derivative-def by (rule has-derivative-subset)

lemma has-vector-derivative-const[simp, derivative-intros]: $((\lambda x. c) \text{ has-vector-derivative } 0)$ net
by (auto simp: has-vector-derivative-def)
lemma has-vector-derivative-id[simp, derivative-intros]: \((\lambda x . x)\) has-vector-derivative 1
  by (auto simp: has-vector-derivative-def)

lemma has-vector-derivative-minus[derivative-intros]:
  \((f \text{ has-vector-derivative } f')\) net \(\Rightarrow\) \((\lambda x . -f x)\) has-vector-derivative \((-f')\)
  by (auto simp: has-vector-derivative-def)

lemma has-vector-derivative-add[derivative-intros]:
  \((f \text{ has-vector-derivative } f')\) net \(\Rightarrow\) \((g \text{ has-vector-derivative } g')\) net \(\Rightarrow\)
  \((\lambda x . f x + g x)\) has-vector-derivative \((f' + g')\)
  by (auto simp: has-vector-derivative-def scaleR-right-distrib)

lemma has-vector-derivative-sum[derivative-intros]:
  \((\forall i . i \in I \Rightarrow (f i \text{ has-vector-derivative } f'_i))\) net \(\Rightarrow\)
  \((\lambda x . \sum_{i \in I} f'_i)\) has-vector-derivative \((\sum_{i \in I} f'_i)\)
  by (auto simp: has-vector-derivative-def fun-eq-iff scaleR-sum-right intro: derivative-eq-intros)

lemma has-vector-derivative-diff[derivative-intros]:
  \((f \text{ has-vector-derivative } f')\) net \(\Rightarrow\) \((g \text{ has-vector-derivative } g')\) net \(\Rightarrow\)
  \((\lambda x . f x - g x)\) has-vector-derivative \((f' - g')\)
  by (auto simp: has-vector-derivative-def scaleR-diff-right)

lemma has-vector-derivative-diff-const:
  \((\lambda t . g t + z)\) has-vector-derivative \((\lambda t . f t)\)
  net
  apply (intro iffI)
  apply (force dest: has-vector-derivative-diff [where g = \lambda t . z, OF - has-vector-derivative-const])
  apply (force dest: has-vector-derivative-add [OF - has-vector-derivative-const])
  done

lemma has-vector-derivative-diff-const:
  \((\lambda t . g t - z)\) has-vector-derivative \((\lambda t . f t)\)
  net
  using has-vector-derivative-add-const [where z = -z]
  by simp

lemma (in bounded-linear) has-vector-derivative:
  assumes \((g \text{ has-vector-derivative } g')\) F
  shows \((\lambda x . f (g x))\) has-vector-derivative \((f g')\) F
  using has-derivative[OF assms[unfolded has-vector-derivative-def]]
  by (simp add: has-vector-derivative-def scaleR)

lemma (in bounded-bilinear) has-vector-derivative:
  assumes \((f \text{ has-vector-derivative } f')\) (at x within s)
  and \((g \text{ has-vector-derivative } g')\) (at x within s)
  shows \((\lambda x . f x ** g x)\) has-vector-derivative \((f x ** g' + f' ** g x)\) (at x within s)
  using FDERIV[OF assms[1-2][unfolded has-vector-derivative-def]]
by (simp add: has-vector-derivative-def scaleR-right scaleR-left scaleR-right-distrib)

lemma has-vector-derivative-scaleR[derivative-intros]:
\[(f \text{ has-vector-derivative } f') (at \ x \ within \ s) \Longrightarrow (g \text{ has-vector-derivative } g') (at \ x \ within \ s) \]
\[((\lambda x. \ f x * _R \ g x) \text{ has-vector-derivative } (f x * _R \ g' + f' * _R \ g x)) (at \ x \ within \ s)\]

unfolding has-field-derivative-iff-has-vector-derivative
by (rule bounded-bilinear.has-vector-derivative[OF bounded-bilinear-scaleR])

lemma has-vector-derivative-mult[derivative-intros]:
\[(f \text{ has-vector-derivative } f') (at \ x \ within \ s) \Longrightarrow (g \text{ has-vector-derivative } g') (at \ x \ within \ s) \]
\[((\lambda x. \ f x * g x) \text{ has-vector-derivative } (f x * g' + f' * g x)) (at \ x \ within \ s)\]
for \(f g :: \text{real-normed-algebra}\)
by (rule bounded-bilinear.has-vector-derivative[OF bounded-bilinear-mult])

lemma has-vector-derivative-of-real[derivative-intros]:
\[(f \text{ has-vector-derivative } D) \ F \Longrightarrow ((\lambda x. \ \text{of-real} (f x)) \text{ has-vector-derivative } (\text{of-real} D)) \ F\]
by (simp add: has-field-derivative-iff-has-vector-derivative)

lemma has-vector-derivative-real-field:
\[(f \text{ has-field-derivative } f') (at \ \text{of-real} a) \Longrightarrow ((\lambda x. \ f (\text{of-real} x)) \text{ has-vector-derivative } f') (at \ a \ within \ s)\]
using has-derivative-compose[of of-real of-real a - f' f']
by (simp add: scaleR-conv-of-real ac-simps has-vector-derivative-def has-field-derivative-def)

lemma has-vector-derivative-continuous:
\[(f \text{ has-vector-derivative } D) (at \ x \ within \ s) \Longrightarrow \text{continuous } (at \ x \ within \ s) \ F\]
by (auto intro: has-derivative-continuous simp: has-vector-derivative-def)

lemma continuous-on-vector-derivative:
\[(\forall x. \ x \in S \Longrightarrow (f \text{ has-vector-derivative } f' \ x) (at \ x \ within \ S)) \Longrightarrow \text{continuous-on } S \ f\]
by (auto simp: continuous-on-eq-continuous-within intro!: has-vector-derivative-continuous)

lemma has-vector-derivative-mult-right[derivative-intros]:
\text{fixes } a :: 'a::real-normed-algebra
\text{shows } (f \text{ has-vector-derivative } x) \ F \Longrightarrow ((\lambda x. \ a * f x) \text{ has-vector-derivative } (a * x)) \ F\]
by (rule bounded-linear.has-vector-derivative[OF bounded-linear-mult-right])

lemma has-vector-derivative-mult-left[derivative-intros]:
\text{fixes } a :: 'a::real-normed-algebra
\text{shows } (f \text{ has-vector-derivative } x) \ F \Longrightarrow ((\lambda x. \ f x * a) \text{ has-vector-derivative } (x * a)) \ F\]
by (rule bounded-linear.has-vector-derivative[OF bounded-linear-mult-left])
108.7 Derivatives

**Lemma DERIV-D**: \( \text{DERIV } f \ x : \Rightarrow \ (\lambda h. (f \ (x + h) - f \ x) / h) \leftarrow 0 \Rightarrow D \)

by (simp add: DERIV-def)

**Lemma has-field-derivativeD**: 
\( (f \text{ has-field-derivative } D) \ (at \ x \ within \ S) \Rightarrow \ ((\lambda x. (f \ y - f \ x) / (y - x)) \leftarrow 0 \Rightarrow D) \ (at \ x \ within \ S) \)

by (simp add: has-field-derivative-iff)

**Lemma DERIV-const**: 
\( (\lambda x. k) \text{ has-field-derivative } 0 \)

by (rule has-derivative-imp-has-field-derivative)

**Lemma DERIV-ident**: 
\( (\lambda x. x) \text{ has-field-derivative } 1 \)

by (rule has-derivative-imp-has-field-derivative)

**Lemma field-differentiable-add**: 
\( f \text{ has-field-derivative } f' \) \( \Rightarrow \) 
\( g \text{ has-field-derivative } g' \) \( \Rightarrow \) 
\( (\lambda z. f \ z + g \ z) \text{ has-field-derivative } f' + g' \)

by (auto simp: has-field-derivative-def field-simps mult-commute-abs)

**Corollary DERIV-add**: 
\( (f \text{ has-field-derivative } D) \ (at \ x \ within \ s) \Rightarrow (g \text{ has-field-derivative } E) \ (at \ x \ within \ s) \Rightarrow \ ((\lambda x. f \ x + g \ x) \text{ has-field-derivative } D + E) \ (at \ x \ within \ s) \)

by (rule field-differentiable-add)

**Lemma field-differentiable-minus**: 
\( f \text{ has-field-derivative } f' \) \( \Rightarrow \) 
\( (\lambda z. - f \ z) \text{ has-field-derivative } -f' \)

by (auto simp: has-field-derivative-def field-simps mult-commute-abs)

**Corollary DERIV-minus**: 
\( (f \text{ has-field-derivative } D) \ (at \ x \ within \ s) \Rightarrow \ ((\lambda x. - f \ x) \text{ has-field-derivative } -D) \ (at \ x \ within \ s) \)

by (rule field-differentiable-minus)

**Lemma field-differentiable-diff**: 
\( (f \text{ has-field-derivative } f') \ F \Rightarrow \) 
\( (g \text{ has-field-derivative } g') \ F \Rightarrow \) 
\( ((\lambda z. f \ z - g \ z) \text{ has-field-derivative } f' - g') \ F \)

by (simp only: diff-conv-add-uminus field-differentiable-add field-differentiable-minus)

**Corollary DERIV-diff**: 
\( (f \text{ has-field-derivative } D) \ (at \ x \ within \ s) \Rightarrow \) 
\( (g \text{ has-field-derivative } E) \ (at \ x \ within \ s) \Rightarrow \) 
\( ((\lambda x. f \ x - g \ x) \text{ has-field-derivative } D - E) \ (at \ x \ within \ s) \)

by (rule field-differentiable-diff)

**Lemma DERIV-continuous**: 
\( (f \text{ has-field-derivative } D) \ (at \ x \ within \ s) \Rightarrow \) 
continu-
THEORY "Deriv"

ous (at x within s) f
  by (drule has-derivative-continuous[OF has-field-derivative-imp-has-derivative])
simp

corollary DERIV-isCont: DERIV f x :> D ==> isCont f x
  by (rule DERIV-continuous)

lemma DERIV-atLeastAtMost-imp-continuous-on:
  assumes \( \forall x. [a \leq x; x \leq b] \implies \exists y. \text{DERIV } f x :> y \)
  shows continuous-on \( \{a..b\} \) f
  by (meson DERIV-isCont assms atLeastAtMost-iff continuous-at-imp-continuous-at-within
continuous-on-eq-continuous-within)

lemma DERIV-continuous-on:
  (\( \forall x. x \in s \implies (f \text{ has-field-derivative } (D x)) \text{ (at x within s)) } \implies \text{continuous-on } s \ f \)
  unfolding continuous-on-eq-continuous-within
  by (intro continuous-at-imp-continuous-on ballI DERIV-continuous)

lemma DERIV-mult':
  (f has-field-derivative D) (at x within s) ==> (g has-field-derivative E) (at x within s) 
  ==> 
  ((\( \lambda x. f x * g x \) has-field-derivative f x * E + D * g x) (at x within s)
  by (rule has-derivative-imp-has-field-derivative[OF has-derivative-mult])
  (auto simp: field-simps mult-commute-abs dest: has-field-derivative-imp-has-derivative)

lemma DERIV-mult[derivative-intros]:
  (f has-field-derivative Da) (at x within s) ==> (g has-field-derivative Db) (at x within s) 
  ==> 
  ((\( \lambda x. f x * g x \) has-field-derivative Da * g x + Db * f x) (at x within s)
  by (rule has-derivative-imp-has-field-derivative[OF has-derivative-mult])
  (auto simp: field-simps dest: has-field-derivative-imp-has-derivative)

Derivative of linear multiplication

lemma DERIV-cmult:
  (f has-field-derivative D) (at x within s) =>
  ((\( \lambda x. c * f x \) has-field-derivative c * D) (at x within s)
  by (drule DERIV-mult'[OF DERIV-const]) simp

lemma DERIV-cmult-right:
  (f has-field-derivative D) (at x within s) =>
  ((\( \lambda x. f x * c \) has-field-derivative D * c) (at x within s)
  using DERIV-cmult by (auto simp add: ac-simps)

lemma DERIV-cmult-Id [simp]: \( (\lambda c \text{ has-field-derivative } c) \text{ (at x within s)} \)
  using DERIV-ident [THEN DERIV-cmult, where c = c and x = x] by simp

lemma DERIV-cdivide:
  (f has-field-derivative D) (at x within s) =>
\[(\lambda x. f x / c) \text{ has-field-derivative } D / c \text{ (at } x \text{ within } s)\]

**using** `DERIV-cmult-right [of ] D x s 1 / c] by simp`

**lemma** `DERIV-unique`: `DERIV f x :> D \implies DERIV f x :> E \implies D = E`

**unfolding** `DERIV-def` by (rule LIM-unique)

**lemma** `DERIV-sum [derivative-intros]`:
\[(\forall n. n \in S \implies ((\lambda x. f x n) \text{ has-field-derivative } (f' x n)) F') \implies ((\lambda x. \text{ sum } (f x) S) \text{ has-field-derivative sum } (f' x) S) F)\]

by (rule `has-derivative-imp-has-field-derivative [OF has-derivative-sum]`)
(auto simp: `sum-distrib-left mult-commute-abs dest: has-field-derivative-imp-has-derivative`)

**lemma** `DERIV-inverse' [derivative-intros]`:
- **assumes** `(f has-field-derivative D) (at x within s)`
- and `f x \neq 0`
- **shows** `((\lambda x. inverse (f x)) \text{ has-field-derivative } (inverse (f x) * D * inverse (f x)))`
  - (at `x` within `s`)

**proof**
- have `(f has-derivative (\lambda x. x * D)) = (f has-derivative (\ast D))`
  - by (rule `arg-cong [of \lambda x. x * D]` (simp add: fun-eq_iff))

with `assms` have `(f has-derivative (\lambda x. x * D)) (at x within s)`
  - by (auto dest!: `has-derivative-imp-has-field-derivative`)

then show `?thesis using (f x \neq 0)`
  - by (auto intro: `has-derivative-imp-has-field-derivative has-derivative-inverse`)

qed

Power of \(-1\)

**lemma** `DERIV-inverse`:
\[x \neq 0 \implies ((\lambda x. inverse (x)) \text{ has-field-derivative } (inverse x \circ Suc (Suc 0))) (at x within s)\]

by (drule `DERIV-inverse' [OF `DERIV-ident`]) simp

Derivative of inverse

**lemma** `DERIV-inverse-fun`:
\[(f has-field-derivative d) (at x within s) \implies f x \neq 0 \implies ((\lambda x. inverse (f x)) \text{ has-field-derivative } (inverse (f x) \circ suc (Suc 0))))\]

(at `x` within `s`)

by (drule `(1)` `DERIV-inverse'`) (simp add: ac-simps nonzero-inverse-mult-distr)

Derivative of quotient

**lemma** `DERIV-divide [derivative-intros]`:
\[(f has-field-derivative D) (at x within s) \implies (g has-field-derivative E) (at x within s) \implies g x \neq 0 \implies ((\lambda x. f x / g x) \text{ has-field-derivative } (D * g x - f x * E) / (g x * g x)) (at x within s)\]

by (rule `has-derivative-imp-has-field-derivative [OF has-derivative-divide]`)
(auto dest: `has-field-derivative-imp-has-field-derivative simp: field-simps`
lemma DERIV-quotient:
(f has-field-derivative d) (at x within s) \implies
(g has-field-derivative e) (at x within s) \implies g x \neq 0 \implies
((\lambda y. f y / g y) has-field-derivative (d * g x - (e * f x)) / (g x \cdot Suc (Suc 0))) (at x within s)
by (drule (2) DERIV-divide) (simp add: mult.commute)

lemma DERIV-power-Suc:
(f has-field-derivative D) (at x within s) \implies
((\lambda x. f x \cdot Suc n) has-field-derivative of-nat n * (D * f x \cdot n)) (at x within s)
by (rule has-derivative-imp-has-field-derivative[OF has-derivative-power])
(auto simp: has-field-derivative-def)

lemma DERIV-power:[derivative-intros]:
(f has-field-derivative D) (at x within s) \implies
((\lambda x. f x \cdot n) has-field-derivative of-nat n * (D * f x \cdot (n - Suc 0))) (at x within s)
by (rule has-derivative-imp-has-field-derivative[OF has-derivative-power])
(auto simp: has-field-derivative-def)

lemma DERIV-pow: ((\lambda x. f x \cdot n) has-field-derivative real n * (f x \cdot (n - Suc 0))) (at x within s)
using DERIV-power[OF DERIV-ident] by simp

lemma DERIV-chain': (f has-field-derivative D) (at x within s) \implies DERIV g (f x) :> E \implies
((\lambda x. g (f x)) has-field-derivative E * D) (at x within s)
using has-derivative-compose[of f (*) D x s g (*) E]
by (simp only: has-field-derivative-def mult-commute-abs ac-simps)

corollary DERIV-chain2: DERIV f (g x) :> Da \implies (g has-field-derivative Db) (at x within s) \implies
((\lambda x. f (g x)) has-field-derivative Da * Db) (at x within s)
by (rule DERIV-chain')

Standard version

lemma DERIV-chain:
DERIV f (g x) :> Da \implies (g has-field-derivative Db) (at x within s) \implies
(f o g has-field-derivative Da * Db) (at x within s)
by (drule (1) DERIV-chain', simp add: o-def mult.commute)

lemma DERIV-image-chain:
(f has-field-derivative Da) (at (g x) within (g \cdot s)) \implies
(g has-field-derivative Db) (at x within s) \implies
(f o g has-field-derivative Da * Db) (at x within s)
using has-derivative-in-compose[of g (*) Db x s f (*) Da]
by (simp add: has-field-derivative-def o-def mult-commute-abs ac-simps)
theory "Deriv"

lemma DERIV-chain-s:
assumes \( \forall x. x \in s \Rightarrow \text{DERIV } g \ x :> g'(x) \)
and \( \text{DERIV } f \ x :> f' \)
and \( f \ x \in s \)
shows \( \text{DERIV } (\lambda x. g(f \ x)) \ x :> f' * g'(f \ x) \)
by (metis (full-types) DERIV-chain' mult.commute assms)

lemma DERIV-chain3:
assumes \( \forall x. \text{DERIV } g \ x :> g'(x) \)
and \( \text{DERIV } f \ x :> f' \)
shows \( \text{DERIV } (\lambda x. g(f \ x)) \ x :> f' * g'(f \ x) \)
by (metis UNIV-I DERIV-chain-s [of UNIV] assms)

Alternative definition for differentiability

lemma DERIV-LIM-iff:
fixes \( f :: 'a::{\text{real-normed-vector,inverse}} \Rightarrow 'a \)
shows \( (\lambda h. (f \ (a + h) - f \ a) / h) \rightarrow D) = (\lambda x. (f x - f a) / (x - a)) \rightarrow -a \rightarrow D \)
(is \( ?lhs = ?rhs \))
proof
assume \( ?lhs \)
then have \( (\lambda x. (f \ (a + (x - a)) - f \ a) / (x - a)) \rightarrow -a \rightarrow D \)
by (rule LIM-offset)
then show \( ?rhs \)
by simp
next
assume \( ?rhs \)
then have \( (\lambda x. (f \ (x + a) - f \ a) / ((x + a) - a)) \rightarrow -a \rightarrow D \)
by (rule LIM-offset)
then show \( ?lhs \)
by (simp add: add.commute)
qed

lemma has-field-derivative-cong-ev:
assumes \( x = y \)
and \( *: \text{eventually } (\lambda x. x \in S \rightarrow f \ x = g \ x) \) (nhds \( x \))
and \( u = v \ S = t \ x \in S \)
shows \( (f \text{ has-field-derivative } u) \ (at \ x \text{ within } S) = (g \text{ has-field-derivative } v) \ (at \ y \text{ within } t) \)
unfolding has-field-derivative-iff
proof (rule filterlim-cong)
from assms have \( f \ y = g \ y \)
by (auto simp: eventually-nhds)
with \( * \) show \( \forall F \ z \in at \ x \text{ within } S. (f \ z - f \ x) / (z - x) = (g \ z - g \ y) / (z - y) \)
unfolding eventually-at-filter
by eventually-elim (auto simp: assms \( f \ y = g \ y \))
qed (simp-all add: assms)
lemma has-field-derivative-cong-eventually:
assumes eventually (λx. f x = g x) (at x within S) f x = g x
shows (f has-field-derivative u) (at x within S) = (g has-field-derivative u) (at x within S)
unfolding has-field-derivative-iff
proof (rule tendsto-cong)
show ∀F y in at x within S. (f y − f x) / (y − x) = (g y − g x) / (y − x)
using assms by (auto elim: eventually-mono)
qed

lemma DERIV-cong-ev:
x = y =⇒ eventually (λx. f x = g x) (nhds x) =⇒ u = v =⇒
DERIV f x :> u ←→ DERIV g y :> v
by (rule has-field-derivative-cong-ev) simp-all

lemma DERIV-shift:
(f has-field-derivative y) (at (x + z)) = ((λx. f (x + z)) has-field-derivative y) (at x)
by (simp add: DERIV-def field-simps)

lemma DERIV-mirror: (DERIV f (− x) :> y) ←→ (DERIV (λx. f (− x)) x :> − y)
for f :: real ⇒ real and x y :: real
by (simp add: DERIV-def filterlim-at-split filterlim-at-left-to-right
tendsto-minus-cancel-left field-simps conj-commute)

lemma floor-has-real-derivative:
fixes f :: real ⇒ 'a::{floor-ceiling,order-topology}
assumes isCont f x
and f x ∈ ℤ
shows ((λx. floor (f x)) has-real-derivative 0) (at x)
proof (subst DERIV-cong-ev[OF refl - refl])
show ((λ−. floor (f x)) has-real-derivative 0) (at x)
by simp
have ∀F y in at x. ∥f y∥ = ∥f x∥
by (rule eventually-floor-eq[OF assms unfolded continuous-at])
then show ∀F y in nhds x. real-of-int ∥f y∥ = real-of-int ∥f x∥
unfolding eventually-at-filter
by eventually-elim auto
qed

lemmas has-derivative-floor[derivative-intros] =
floor-has-real-derivative[THEN DERIV-compose-FDERIV]

lemma continuous-floor:
fixes x::real
shows x ∉ ℤ =⇒ continuous (at x) (real-of-int ∘ floor)
using floor-has-real-derivative [where f=id]
by (auto simp: o-def has-field-derivative-def intro: has-derivative-continuous)
lemma continuous-frac:
  fixes x :: real
  assumes x ∈ ℤ
  shows continuous (at x) frac
proof –
  have isCont (λx. real-of-int ⌊x⌋) x
    using continuous-floor [OF assms] by (simp add: o-def)
  then have *: continuous (at x) (λx. x − real-of-int ⌊x⌋)
    by (intro continuous-intros)
  moreover have ∀F x in nhds x. frac x = x − real-of-int ⌊x⌋
    by (simp add: frac-def)
  ultimately show ?thesis
    by (simp add: LIM-imp-LIM frac-def isCont-def)
qed

Caratheodory formulation of derivative at a point

lemma CARAT-DERIV:
  (DERIV f x :> l) ←→ (∃g. (∀z. f z − f x = g z * (z − x)) ∧ isCont g x ∧ g x = l)
  (is lhs = rhs)
proof
  assume lhs
  show ∃g. (∀z. f z − f x = g z * (z − x)) ∧ isCont g x ∧ g x = l
  proof (intro exI conjI)
    let g = (λz. if z = x then l else (f z − f x) / (z − x))
    show ∀z. f z − f x = ?g z * (z − x)
      by simp
    show isCont ?g x
      using ⟨lhs⟩ by (simp add: isCont-iff DERIV-def cong: LIM-equal [rule-format])
    show ?g x = l
      by simp
  qed
next
  assume rhs
  then show rhs
    by (auto simp add: isCont-iff DERIV-def cong: LIM-cong)
qed

108.8 Local extrema

If (0::'a) < f' x then x is Locally Strictly Increasing At The Right.

lemma has-real-derivative-pos-inc-right:
  fixes f :: real ⇒ real
  assumes der: (f has-real-derivative l) (at x within S)
    and l: 0 < l
  shows ∃d > 0. ∀h > 0. x + h ∈ S → h < d → f x < f (x + h)
  using assms
proof –
from der [THEN has-field-derivativeD, THEN tendstoD, OF l, unfolded eventually-at]

obtain s where s: 0 < s
    and all: \( \forall xa. xa \in S \rightarrow xa \neq x \land \text{dist} xa x < s \rightarrow |(f xa - f x) / (xa - x)| < l \)
by (auto simp: dist-real-def)

then show \( \text{thesis} \)
proof (intro exI conjI strip)
  show \( 0 < s \) by (rule s)
next
fix h :: real
assume \( 0 < h \) \( h < s x + h \in S \)
with all \( \text{also} \) \( x + h \) \( f x < f (x + h) \)
proof (simp add: abs-if dist-real-def pos-less-divide-eq split: if-split-asm)
  assume \( - l < |(f (x + h) - f x) / h| \)
  with \( l \) have \( 0 < (f (x + h) - f x) / h \)
  by arith
  then show \( f x < f (x + h) \)
  by (simp add: pos-less-divide-eq h)
qed

lemma DERIV-pos-inc-right:
fixes f :: real \( \Rightarrow \) real
assumes der: \( \text{DERIV f x :> l} \)
    and l: \( 0 < l \)
shows \( \exists d > 0. \ \forall h > 0. \ h < d \rightarrow f x < f (x + h) \)
using has-real-derivative-pos-inc-right [OF assms]
by auto

lemma has-real-derivative-neg-dec-left:
fixes f :: real \( \Rightarrow \) real
assumes der: \( \text{(f has-real-derivative l)} \) \( \text{(at x within S)} \)
    and l: \( l < 0 \)
shows \( \exists d > 0. \ \forall h > 0. \ x - h \in S \rightarrow h < d \rightarrow f x < f (x - h) \)
proof –
  from \( l < 0 \) have l: \( - l > 0 \)
  by simp
  from der [THEN has-field-derivativeD, THEN tendstoD, OF l, unfolded eventually-at]
  obtain s where s: 0 < s
    and all: \( \forall xa. xa \in S \rightarrow xa \neq x \land \text{dist} xa x < s \rightarrow |(f xa - f x) / (xa - x)| < l \)
  by (auto simp: dist-real-def)
  then show \( \text{thesis} \)
proof (intro exI conjI strip)
  show \( 0 < s \) by (rule s)
next
fix h :: real
assume \( 0 < h \) \( h < s x - h \in S \)
with all \([of \, x - h]\) show \(f \, x < f \, (x - h)\)

proof (simp add: abs-if pos-less-divide-eq dist-real-def split: if-split-asm)

assume \((-((f \, (x-h) - f \, x) / h) < l\) and \(h: 0 < h\) with \(l\) have \(0 < (f \, (x-h) - f \, x) / h\)

by arith

then show \(f \, x < f \, (x - h)\)

by (simp add: pos-less-divide-eq h)

qed

lemma DERIV-neg-dec-left:

fixes \(f::\) real \(\Rightarrow\) real

assumes \(der::\) DERIV \(f \, x :: l\)

and \(l::l < 0\)

shows \(\exists \, d > 0. \forall \, h > 0. \, h < d \rightarrow f \, x < f \, (x - h)\)

using has-real-derivative-neg-dec-left[OF assms]

by auto

lemma has-real-derivative-pos-inc-left:

fixes \(f::\) real \(\Rightarrow\) real

shows \((f \, has-real-derivative \, l) \, (at \, x \, within \, S) \Rightarrow 0 < l \Rightarrow \exists \, d > 0. \forall \, h > 0. \, x - h \in \, S \rightarrow h < d \rightarrow f \, (x - h) < f \, x\)

by (rule has-real-derivative-neg-dec-left [of \(\lambda x. \, f \, x - l \, x \, S\)], simplified)

(auto simp add: DERIV-minus)

lemma DERIV-pos-inc-left:

fixes \(f::\) real \(\Rightarrow\) real

shows DERIV \(f \, x :: l \Rightarrow 0 < l \Rightarrow \exists \, d > 0. \forall \, h > 0. \, h < d \rightarrow f \, (x - h) < f \, x\)

using has-real-derivative-pos-inc-left

by blast

lemma has-real-derivative-neg-dec-right:

fixes \(f::\) real \(\Rightarrow\) real

shows \((f \, has-real-derivative \, l) \, (at \, x \, within \, S) \Rightarrow l < 0 \Rightarrow \exists \, d > 0. \forall \, h > 0. \, x + h \in \, S \rightarrow h < d \rightarrow f \, x > f \, (x + h)\)

by (rule has-real-derivative-pos-inc-right [of \(\lambda x. \, f \, x - l \, x \, S\)], simplified)

(auto simp add: DERIV-minus)

lemma DERIV-neg-dec-right:

fixes \(f::\) real \(\Rightarrow\) real

shows DERIV \(f \, x :: l \Rightarrow l < 0 \Rightarrow \exists \, d > 0. \forall \, h > 0. \, h < d \rightarrow f \, x > f \, (x + h)\)

using has-real-derivative-neg-dec-right by blast

lemma DERIV-local-max:

fixes \(f::\) real \(\Rightarrow\) real

assumes \(der::\) DERIV \(f \, x :: l\)
THEORY "Deriv"

and d: 0 < d
and le: ∀ y. |x - y| < d → f y ≤ f x
shows l = 0
proof (cases rule: linorder-cases [of l 0])
  case equal
  then show thesis.
next
  case less
  from DERIV-neg-dec-left [OF der less]
  obtain d' where d': 0 < d' and lt: ∀ h > 0. h < d' → f x < f (x - h)
    by blast
  obtain e where 0 < e ∧ e < d ∧ e < d'
    using field-lbound-gt-zero [OF d d']..
  with lt le [THEN spec [where x=x - e]] show thesis
    by (auto simp add: abs-if)
next
  case greater
  from DERIV-pos-inc-right [OF der greater]
  obtain d' where d': 0 < d' and lt: ∀ h > 0. h < d' → f x < f (x + h)
    by blast
  obtain e where 0 < e ∧ e < d ∧ e < d'
    using field-lbound-gt-zero [OF d d']..
  with lt le [THEN spec [where x=x + e]] show thesis
    by (auto simp add: abs-if)
qed

Similar theorem for a local minimum

lemma DERIV-local-min:
  fixes f :: real ⇒ real
  shows DERIV f x :: l ⇒ 0 < d ⇒ ∀ y. |x - y| < d → f x ≤ f y ⇒ l = 0
  by (drule DERIV-minus [THEN DERIV-local-max]) auto

In particular, if a function is locally flat

lemma DERIV-local-const:
  fixes f :: real ⇒ real
  shows DERIV f x :: l ⇒ 0 < d ⇒ ∀ y. |x - y| < d → f x = f y ⇒ l = 0
  by (auto dest!: DERIV-local-max)

108.9 Rolle's Theorem

Lemma about introducing open ball in open interval

lemma lemma-interval-lt:
  fixes a b x :: real
  assumes a < x x < b
  shows ∃ d. 0 < d ∧ (∀ y. |x - y| < d → a < y ∧ y < b)
  using linorder-linear [of x - a b - x]
proof
  assume x - a ≤ b - x
with assms show ?thesis
  by (rule-tac x = x - a in exI) auto
next
  assume b - x <= x - a
  with assms show ?thesis
  by (rule-tac x = b - x in exI) auto
qed

lemma lemma-interval: a < x ==> x < b ==> \exists d. 0 < d & (\forall y. |x - y| < d ==> a <= y & y <= b)
  for a b x :: real
  by (force dest: lemma-interval-lt)

Rolle’s Theorem. If f is defined and continuous on the closed interval [a,b] and differentiable on the open interval (a,b), and f a = f b, then there exists x0 \in (a,b) such that f'(x0) = (0::'a)

theorem Rolle-deriv:
  fixes f :: real => real
  assumes a < b and fab: f a = f b and contf: continuous-on {a..b} f
  and derf: \forall x. [a < x; x < b] ==> (f has-derivative f' x) (at x)
  shows \exists z. a < z & z < b & f' z = (\lambda v. 0)
proof
  have le: a <= b
  using (a < b) by simp
  have (a + b) / 2 \in {a..b}
  using assms(1) by auto
  then have *: {a..b} \neq {} by auto

  obtain x where x-max: \forall z. a <= z & z <= b ==> f z <= f x and a <= x x <= b
    using continuous-attains-sup[OF compact-Icc * contf]
    by (meson atLeastAtMost-iff)

  obtain x' where x'-min: \forall z. a <= z & z <= b ==> f x' <= f z and a <= x' x' <= b
    using continuous-attains-inf[OF compact-Icc * contf] by (meson atLeastAtMost-iff)
  consider a < x x < b | x = a \lor x = b
  using (a <= x) (x <= b) by arith
  then show ?thesis
proof cases
  case 1
  -- f attains its maximum within the interval
  then obtain l where der: DERIV f x :> l
    using derf differentiable-def real-differentiable-def by blast
  obtain d where d: 0 < d and bound: \forall y. |x - y| < d --> a <= y & y <= b
    using lemma-interval [OF 1] by blast
  then have bound': \forall y. |x - y| < d --> f y <= f x
    using x-max by blast
  -- the derivative at a local maximum is zero
  have l = 0
by (rule DERIV-local-max [OF der d bound'])
with 1 der derf [of x] show ?thesis
  by (metis has-derivative-unique has-field-derivative-def mult-zero-left)

next
case 2
then have fx: f b = f x by (auto simp add: fab)
consider a < x' < b | x' = a ∨ x' = b
  using (a ≤ x') (x' ≤ b) by arith
then show ?thesis
proof cases
  case 1
  — f attains its minimum within the interval
then obtain l where der: DERIV f x' := l
  using derf differentiable-def real-differentiable-def by blast
from lemma-interval [OF 1]
then have bound': ∀ y. |x' − y| < d → a ≤ y ∧ y ≤ b
  using x'-min by blast
have l = 0 by (rule DERIV-local-min [OF der d bound'])
  — the derivative at a local minimum is zero
then show ?thesis using 1 der derf [of x']
  by (metis has-derivative-unique has-field-derivative-def mult-zero-left)

next
case 2
  — f is constant throughout the interval
then have f: f b = f x' by (auto simp: fab)
from dense [OF a < b] obtain r where r: a < r r < b by blast
then have bound': ∀ y. |r − y| < d → f x' ≤ f y
  using x'-min by blast
have eq-fb: f z = f b if a ≤ z and z ≤ b for z
proof (rule order-antisym)
  show f z ≤ f b by (simp add: fx x-max that)
  show f b ≤ f z by (simp add: fx x'-min that)
qed
have bound': ∀ y. |r − y| < d → f r = f y
proof (intro strip)
  fix y :: real
  assume lt: |r − y| < d
  then have f y = f b by (simp add: eq-fb bound)
  then show f r = f y by (simp add: eq-fb r order-less-imp-le)
qed
then have bound': ∀ y. |r − y| < d → f r = f y
proof (intro strip)
  fix y :: real
  assume lt: |r − y| < d
  then have f y = f b by (simp add: eq-fb bound)
  then show f r = f y by (simp add: eq-fb r order-less-imp-le)
qed
then have bound': ∀ y. |r − y| < d → f r = f y
proof (intro strip)
  fix y :: real
  assume lt: |r − y| < d
  then have f y = f b by (simp add: eq-fb bound)
  then show f r = f y by (simp add: eq-fb r order-less-imp-le)
qed
then have l = 0 by (rule DERIV-local-const [OF der d bound'])
  — the derivative of a constant function is zero
with r der derf [of r] show ?thesis
  by (metis has-derivative-unique has-field-derivative-def mult-zero-left)
corollary Rolle:
fixes a b :: real
assumes ab: a < b f a = f b continuous-on \{a..b\} f
and dif: \(\forall x. [a < x; x < b] \implies f\) differentiable (at x)
shows \(\exists z. a < z \land z < b \land \text{DERIV} f z > 0\)
proof –
obtain f' where f': \(\forall x. [a < x; x < b] \implies (f\) has-derivative f' x\) (at x)
using dif unfolding differentiable-def by metis
then have \(\exists z. a < z \land z < b \land f' z = (\lambda x. 0)\)
by (metis Rolle-deriv [OF ab])
then show \(?thesis\)
using f' has-derivative-imp-has-field-derivative by fastforce
qed

108.10 Mean Value Theorem

theorem mvt:
fixes f :: real ⇒ real
assumes a < b
and contf: continuous-on \{a..b\} f
and dif: \(\forall x. [a < x; x < b] \implies (f\) has-derivative f' x\) (at x)
obtains \(\xi\) where \(a < \xi \land \xi < b \land (f b - f a) / (b - a) = (\lambda y. 0)\)
proof (intro Rolle-deriv[OF a < b])
fix x
assume x: a < x x < b
show \((\lambda y. f x - (f b - f a) / (b - a) \cdot y) = (\lambda y. 0)\) (at x)
by (intro derivative-intros derf[OF x])
qed (use assms in ⟨auto intro!: continuous-intros simp: field-simps⟩)
then obtain \(\xi\) where
\(a < \xi \land \xi < b \land (\lambda y. f' \xi y - (f b - f a) / (b - a) \cdot y) = (\lambda y. 0)\)
by metis
then show \(?thesis\)
by (metis (no-types, hide-lams) that add.right-neutral add-diff-cancel-left' add-diff-eq
\(\langle a < b\rangle\)
less-irrefl nonzero-eq-divide-eq)
qed

theorem MVT:
fixes a b :: real
assumes lt: a < b
and contf: continuous-on \{a..b\} f
and dif: \(\forall x. [a < x; x < b] \implies f\) differentiable (at x)
shows \( \exists l \ z. \ a < z \land z < b \land \text{DERIV } f \ z :> l \land f b - f a = (b - a) \ast l \)

proof –

obtain \( f' :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real} \)
where \( \text{derf}: \forall x. \ a < x \Rightarrow x < b \Rightarrow (f \text{-derivative } f' \ x) \ (at \ x) \)
using \( \text{dif unfolding differentiable-def by metis} \)
then obtain \( z \) where \( a < z z < b f b - f a = (f' \ z) \ (b - a) \)
using \( \text{met [OF lt contf]} \) by blast
then show \( \text{?thesis} \)
by \( (\text{simp add: ac-simps}) \)
\( (\text{metis derf dif has-derivative-unique has-field-derivative-imp-has-derivative real-differentiable-def}) \)

qed

corollary \( \text{MVT2}: \)
assumes \( a < b \) and \( \text{der}: \forall x. \ [a \leq x; x \leq b] \Rightarrow \text{DERIV } f \ x :> f' \ x \)
shows \( \exists z :: \text{real}. \ a < z \land z < b \land (f b - f a = (b - a) \ast f' \ z) \)

proof –

have \( \exists l \ z. \ a < z \land \)
\( z < b \land \)
\( (f \text{-derivative } l) \ (at \ z) \land \)
\( f b - f a = (b - a) \ast l \)

proof (rule \( \text{MVT [OF (a < b)]} \))

show \( \text{continuous-on } \{a..b\} \ f \)
by \( (\text{meson DERIV-continuous atLeastAtMost-iff continuous-at-imp-continuous-on der}) \)

show \( \forall x. \ [a < x; x < b] \Rightarrow f \text{ differentiable } (at \ x) \)
using \( \text{assms by (force dest: order-les-simp add: real-differentiable-def)} \)

qed

with \( \text{assms show } \text{?thesis} \)


by \( (\text{blast dest: DERIV-unique order-les-simp-le}) \)

qed


lemma \( \text{pos-deriv-imp-strict-mono}: \)

assumes \( \forall x. \ (f \text{-has-derivative } f' \ x) \ (at \ x) \)

assumes \( \forall x. \ f' \ x > 0 \)

shows \( \text{strict-mono } f \)
proof (rule \( \text{strict-monoI} \))

fix \( x \ y :: \text{real assume } xy: x < y \)

from \( \text{assms and } xy \) have \( \exists z :: x < z < y \land f y - f x = (y - x) \ast f' z \)
by \( (\text{intro MVT2}) \) (auto dest: connectedD-interval)

then obtain \( z \) where \( z: z > x z < y f y - f x = (y - x) \ast f' z \) by blast

note \( f y - f x = (y - x) \ast f' z \) using \( xy \text{ assms by (intro mult-pos-pos) auto} \)
also have \( (y - x) \ast f' z > 0 \) using \( xy \text{ assms by (intro mult-pos-pos) auto} \)
finally show \( f x < f y \) by simp

qed


proposition \( \text{derie-nonneg-imp-mono}: \)

assumes \( \text{deri}: \forall x. \ x \in \{a..b\} \Rightarrow (g \text{-has-derivative } g' \ x) \ (at \ x) \)

assumes \( \text{nonneg}: \forall x. \ x \in \{a..b\} \Rightarrow g' \ x \geq 0 \)
assumes \( ab: a \leq b \)
shows \( g a \leq g b \)
proof (cases \( a < b \))
assume \( a < b \)
from deriv have \( \forall x. \ [x \geq a; x \leq b] \implies (g \text{ has-real-derivative } g' x) \) (at \( x \)) by simp
with MVT2[OF \( : a < b \)] and deriv
obtain \( \xi \) where \( \xi - ab: \xi > a \xi < b \) and \( g-ab: g b - g a = (b - a) \ast g' \xi \) by blast
from \( \xi - ab \) ab nonneg have \( (b - a) \ast g' \xi \geq 0 \) by simp
qed (insert ab, simp)

108.10.1 A function is constant if its derivative is 0 over an interval.

lemma DERIV-isconst-end:
fixes \( f :: \text{real} \Rightarrow \text{real} \)
assumes \( a < b \) and \( \text{contf}: \text{continuous-on} \ \{a..b\} \ f \)
and \( 0; \forall x. \ [a < x; x < b] \implies \text{DERIV } f x > 0 \)
shows \( f b = f a \)
using MVT [OF \( : a < b \)] 0 DERIV-unique contf real-differentiable-def
by (fastforce simp: algebra-simps)

lemma DERIV-isconst2:
fixes \( f :: \text{real} \Rightarrow \text{real} \)
assumes \( a < b \) and \( \text{contf}: \text{continuous-on} \ \{a..b\} \ f \) and \( \text{derf}: \forall x. \ [a < x; x < b] \implies \text{DERIV } f x > 0 \)
and \( a \leq x x \leq b \)
shows \( f x = f a \)
proof (cases \( x = a \))
case True
have \( \ast: \text{continuous-on} \ \{a..x\} \ f \)
using \( x \leq b \) contf continuous-on-subset by fastforce
show \( \text{thesis} \)
by (rule DERIV-isconst-end [OF True \( \ast \)]) (use \( x \leq b \) derf in auto)
qed (use \( a \leq x \) in auto)

lemma DERIV-isconst3:
fixes \( a b x y :: \text{real} \)
assumes \( a < b \)
and \( x \in \{a <..< b\} \)
and \( y \in \{a <..< b\} \)
and \( \text{derivable}: \forall x. \ x \in \{a <..< b\} \implies \text{DERIV } f x > 0 \)
shows \( f x = f y \)
proof (cases \( x = y \))
case False
let \( ?a = \text{min } x y \)
let \( ?b = \text{max } x y \)
have \(
\ast\): \(\text{DERIV} \ f \ z :> 0\) if \(\ ?a \leq z \leq \ ?b\) for \(z\)

proof 
  have \(\ ?a < z \ \text{and} \ z < \ ?b\)
  using that \(\{x \in \{\ ?a <..< \ ?b\}\} \ \text{and} \ \{y \in \{\ ?a <..< \ ?b\}\}\) by \(\text{auto}\)
  then have \(z \in \{\ ?a <..< \ ?b\}\) by \(\text{auto}\)
  then show \(\text{DERIV} \ f \ z :> 0\) by \(\text{(rule derivable)}\)
qed

have \(\text{isCont}: \text{continuous-on} \ \{\ ?a..\ ?b\} \ f\)
by \(\text{(meson} \ \ast \ \text{DERIV-continuous-on atLeastAtMost-iff has-field-derivative-at-within)}\)
have \(\text{DERIV}: \ \forall \ z. \ [\ ?a < z; z < \ ?b] \ \Rightarrow \ \text{DERIV} \ f \ z :> 0\)
using \(\ast\) by \(\text{auto}\)
show \(?\text{thesis}\) by \(\text{auto}\)
qed auto

lemma \(\text{DERIV-isconst-all}:\)
fixes \(f::\text{real} \Rightarrow \text{real}\)
shows \(\forall x. \ \text{DERIV} \ f \ x :> 0 \ \Rightarrow \ f \ x = f \ y\)
apply \(\text{(rule linorder-cases [of } x \ y])}\)
apply \(\text{(metis DERIV-continuous DERIV-unique continuous-at-imp-continuous-on)+}\)
done

lemma \(\text{DERIV-const-ratio-const}:\)
fixes \(f::\text{real} \Rightarrow \text{real}\)
assumes \(a \neq b\) and \(\text{df}: \ \forall x. \ \text{DERIV} \ f \ x :> k\)
shows \((f \ b - f \ a) / (b - a) = k\)
proof \(\text{(cases } a \ b \ \text{rule: linorder-cases)}\)
  case \(\text{less}\)
  show \(?\text{thesis}\)
  using \(\text{MVT [OF less]} \ \text{df}\)
  by \(\text{(metis DERIV-continuous DERIV-unique continuous-at-imp-continuous-on}\)
  \(\text{real-differentiable-def)}\)
next
  case \(\text{greater}\)
  have \((f \ b - f \ a) = (a - b) * k\)
  using \(\text{MVT [OF greater]} \ \text{df}\)
  \(\text{(metis DERIV-continuous DERIV-unique continuous-at-imp-continuous-on}\)
  \(\text{real-differentiable-def)}\)
  then show \(?\text{thesis}\)
  by \(\text{(simp add: algebra-simps)}\)
qed auto

lemma \(\text{DERIV-const-ratio-const2}:\)
fixes \(f::\text{real} \Rightarrow \text{real}\)
assumes \(a \neq b\) and \(\text{df}: \ \forall x. \ \text{DERIV} \ f \ x :> k\)
shows \((f \ b - f \ a) / (b - a) = k\)
using \(\text{DERIV-const-ratio-const [OF } \text{assms]} \ \ast \ \text{by } \text{auto}\)
lemma real-average-minus-first [simp]: 
\[(a + b) / 2 - a = (b - a) / 2\]
for \(a \ b::\ real\)
by simp

lemma real-average-minus-second [simp]: 
\[(b + a) / 2 - a = (b - a) / 2\]
for \(a \ b::\ real\)
by simp

Gallileo’s ”trick”: average velocity = av. of end velocities.

lemma DERIV-const-average::
fixes \(v::\ real\) ⇒ real
and \(a \ b::\ real\)
assumes neq: \(a \neq b\)
and der: \(\forall x.\) DERIV \(v\ x\ > k\)
shows \(v (((a + b) / 2) - v a)/((a + b) / 2 - a) = k\)
proof (cases rule: linorder-cases [of a b])
case equal
with neq show ?thesis by simp
next
case less
have \((v b - v a)/ (b - a) = k\)
  by (rule DERIV-const-ratio-const2 [OF neq der])
then have \((b - a) * ((v b - v a)/(b - a)) = (b - a) * k\)
  by simp
moreover have \((v (((a + b) / 2) - v a)/((a + b) / 2 - a)) = k\)
  by (rule DERIV-const-ratio-const2 [OF - der]) (simp add: neq)
ultimately show ?thesis
  using neq by force
next
case greater
have \((v b - v a)/ (b - a) = k\)
  by (rule DERIV-const-ratio-const2 [OF neq der])
then have \((b - a) * ((v b - v a)/(b - a)) = (b - a) * k\)
  by simp
moreover have \((v (((b + a) / 2) - v a)/((b + a) / 2 - a)) = k\)
  by (rule DERIV-const-ratio-const2 [OF - der]) (simp add: neq)
ultimately show ?thesis
  using neq by (force simp add: add.commute)
qed

108.10.2 A function with positive derivative is increasing

A simple proof using the MVT, by Jeremy Avigad. And variants.

lemma DERIV-pos-imp-increasing-open::
fixes \(a \ b::\ real\)
and \(f::\ real\ ⇒ real\)
assumes \(a < b\)
and \(\forall x.\ a < x \Longrightarrow x < b \Longrightarrow (\exists y.\ DERIV f\ x\ > y \land y > 0)\)
and \text{con}: \text{continuous-on} \{a..b\} f

shows \( f a < f b \)

proof (rule ccontr)

assume \( f: \neg \text{thesis} \)

have \( \exists l \ z. \ a < z \land z < b \land \text{DERIV} f z :> l \land f b - f a = (b - a) \ast l \)

by (rule MVT) (use assms \text{real-differentiable-def} in \text{force+})

then obtain \( l \ z \) where \( z: a < z < b \ \text{DERIV} f z :> l \) and \( f b - f a = (b - a) \ast l \)

by auto

with assms \( f \) have \( \neg l > 0 \)

by (metis \text{linorder-not-le} \text{mult-le-0-iff} \text{diff-le-0-iff-le})

with assms \( z \) show False

by (metis \text{DERIV-unique})

qed

lemma \text{DERIV-pos-imp-increasing}:

fixes \( a \ b :: \text{real} \) and \( f :: \text{real} \Rightarrow \text{real} \)

assumes \( a < b \)

and \( \text{der}: \forall x. [a \leq x; x \leq b] \Rightarrow \exists y. \text{DERIV} f x :> y \land y > 0 \)

shows \( f a < f b \)

by (metis \text{less-le-not-le} \text{DERIV-atLeastAtMost-imp-continuous-on} \text{DERIV-pos-imp-increasing-open} \text{OF} \langle a < b \rangle \text{der})

lemma \text{DERIV-nonneg-imp-nondecreasing}:

fixes \( a \ b :: \text{real} \)

and \( f :: \text{real} \Rightarrow \text{real} \)

assumes \( a \leq b \)

and \( \forall x. [a \leq x; x \leq b] \Rightarrow \exists y. \text{DERIV} f x :> y \land y \geq 0 \)

shows \( f a \leq f b \)

proof (rule ccontr, cases \( a = b \))

assume \( \neg \text{thesis and} \ a = b \)

then show False by auto

next

assume \( *: \neg \text{thesis} \)

assume \( a \neq b \)

with \( a \leq b \) have \( a < b \)

by linarith

moreover have \text{continuous-on} \{a..b\} f

by (meson \text{DERIV-isCont} \text{assms}2 \text{atLeastAtMost-iff} \text{continuous-at-imp-continuous-on})

ultimately have \( \exists l \ z. \ a < z \land z < b \land \text{DERIV} f z :> l \land f b - f a = (b - a) \ast l \)

* \( l \)

using \text{assms MVT} \langle \text{OF} \langle a < b \rangle, \text{of} f \rangle \text{real-differentiable-def} \text{less-eq-real-def} \text{by} \text{blast}\)

then obtain \( l \ z \) where \( l z: a < z < b \ \text{DERIV} f z :> l \) and \( **: f b - f a = (b - a) \ast l \)

by auto

with \( * \) have \( a < b \ \text{must} \text{f b < f a} \text{by} \text{auto} \)

with \( ** \) have \( \neg l \geq 0 \) by (\text{auto simp add: not-le algebra-simps})

(\text{metis} \text{add-le-cancel-right} \text{assms}1 \text{less-eq-real-def} \text{mult-right-mono} \text{add-left-mono} \text{add-right-mono})
linear order-refl)
  with assms lz show False
  by (metis DERIV-unique order-less-imp-le)
qed

lemma DERIV-neg-imp-decreasing-open:
fixes a b :: real
  and f :: real ⇒ real
assumes a < b
  and \( \forall x. \ a < x \Rightarrow x < b \Rightarrow \exists y. \ DERIV f x :> y \land y < 0 \)
  and \( \text{con: continuous-on } \{ a..b \} \ f \)
shows \( f a > f b \)
proof
  have \( (\lambda x. \ -f x) \ a < (\lambda x. \ -f x) \ b \)
  proof (rule DERIV-pos-imp-increasing-open [of a b \( \lambda x. \ -f x \)]
  show \( \forall x. \ [ a < x; x < b ] \Rightarrow \exists y. \ ((\lambda x. \ -f x) \text{ has-real-derivative } y) \ (at x) \land 0 < y \)
    using assms
    by simp (metis field-differentiable-minus neg-0-less-iff-less)
  show continuous-on \( \{ a..b \} \ (\lambda x. \ -f x) \)
    using con continuous-on-minus by blast
  qed (use assms in auto)
  then show \(?thesis\)
    by simp
qed

lemma DERIV-neg-imp-decreasing:
fixes a b :: real and f :: real ⇒ real
assumes a < b
  and \( \forall x. \ [ a \leq x; x \leq b ] \Rightarrow \exists y. \ DERIV f x :> y \land y < 0 \)
shows \( f a > f b \)
by (metis less-le-not-le DERIV-atLeastAtMost-imp-continuous-on DERIV-neg-imp-decreasing-open [OF \( a < b \) \( \text{der} \)]

lemma DERIV-nonpos-imp-nonincreasing:
fixes a b :: real
  and f :: real ⇒ real
assumes a ≤ b
  and \( \forall x. \ [ a \leq x; x \leq b ] \Rightarrow \exists y. \ DERIV f x :> y \land y \leq 0 \)
shows \( f a \geq f b \)
proof
  have \( (\lambda x. \ -f x) \ a \leq (\lambda x. \ -f x) \ b \)
    using DERIV-nonneg-imp-nondecreasing [of a \( \lambda x. \ -f x \) \( \text{assms} \) DERIV-minus
    by fastforce
  then show \(?thesis\)
    by simp
qed

lemma DERIV-pos-imp-increasing-at-bot:
THEORY "Deriv"

fixes \( f : \mathbb{R} \rightarrow \mathbb{R} \)
assumes \( \forall x. \ x \leq b \implies (\exists y. \ \text{DERIV} f x :> y \land y > 0) \)
and \( \lim (f \longrightarrow \text{flim}) \text{ at-bot} \)
shows \( \text{flim} < f b \)

proof
have \( \exists N. \ \forall n \leq N. \ f n \leq f(b - 1) \)
by (rule-tac x=\( b - 2 \) in exI) (force intro: order.strict-implies-order DERIV-pos-imp-increasing assms)
then have \( \text{flim} \leq f(b - 1) \)
by (auto simp: eventually-at-bot-linorder tendsto-upperbound [OF lim])
also have \( \ldots < f b \)
by (force intro: DERIV-pos-imp-increasing [where \( f=f \)] assms)
finally show \( ?\text{thesis} \).
qed

lemma DERIV-neg-imp-decreasing-at-top:
fixes \( f : \mathbb{R} \rightarrow \mathbb{R} \)
assumes \( \text{der}: \forall x. \ x \geq b \implies \exists y. \ \text{DERIV} f x :> y \land y < 0 \)
and \( \lim (f \longrightarrow \text{flim}) \text{ at-top} \)
shows \( \text{flim} < f b \)

apply (rule DERIV-pos-imp-increasing-at-bot [where \( f=\lambda i. \ f(-i) \) and \( b = \) \( -b \), simplified])
apply (metis DERIV-mirror \( \text{der} \) le-minus-iff neg-0-less-iff-less)
apply (metis filterlim-at-top-mirror lim)
done

Derivative of inverse function

lemma DERIV-inverse-function:
fixes \( f \ g : \mathbb{R} \rightarrow \mathbb{R} \)
assumes \( \text{der}: \ \text{DERIV} f \ (g x) :> D \)
and \( \text{neg:} \ D \neq 0 \)
and \( x: \ a < x < b \)
and \( \text{inj}: \ \forall y. \ [a < y; \ y < b] \implies f \ (g y) = y \)
and \( \text{cont:} \ \text{isCont} g x \)
shows \( \text{DERIV} g x :> \text{inverse} D \)

unfolding has-field-derivative-iff

proof (rule LIM-equal2)
show \( 0 < \min (x - a) (b - x) \)
using \( x \) by arith

next
fix \( y \)
assume \( \text{norm} (y - x) < \min (x - a) (b - x) \)
then have \( a < y \) and \( y < b \)
by (simp-all add: abs-less-iff)
then show \( (g y - g x) / (y - x) = \text{inverse} ((f \ (g y) - x) / (g y - g x)) \)
by (simp add: inj)

next
have \( (\lambda z. \ (f z - f (g x)) / (z - g x)) \rightarrow D \)
by (rule der [unfolded has-field-derivative-iff])
108.11 Generalized Mean Value Theorem

\textbf{theorem GMVT:}

\begin{verbatim}
fixes a b :: real
assumes alb: a < b
and fc: \( \forall x. a \leq x \wedge x \leq b \rightarrow \text{isCont f x} \)
and fd: \( \forall x. a < x \wedge x < b \rightarrow f \text{ differentiable (at x)} \)
and gc: \( \forall x. a \leq x \wedge x \leq b \rightarrow \text{isCont g x} \)
and gd: \( \forall x. a < x \wedge x < b \rightarrow g \text{ differentiable (at x)} \)
shows \( \exists g'c f'c. \)
\( \text{DERIV g c} :: g'c \wedge \text{DERIV f c} :: f'c \wedge a < c \wedge c < b \wedge (f b - f a) \ast g'c = (g b - g a) \ast f'c \)
\end{verbatim}

\textbf{proof –}

\begin{verbatim}
let \( ?h = \lambda x. (f b - f a) \ast g x - (g b - g a) \ast f x \)
have \( \exists l z. a < z \wedge z < b \wedge \text{DERIV ?h z} :: l \wedge ?h b - ?h a = (b - a) \ast l \)
\end{verbatim}

\textbf{proof (rule MVT)}

\begin{verbatim}
from assms show \( a < b \) by simp
show \( \text{continuous-on \{a..b\} ?h} \)
  by (simp add: continuous-at-imp-continuous-on fc gc)
show \( \lambda x. [a < x; x < b] \rightarrow ?h \text{ differentiable (at x)} \)
  using fd gd by simp
qed
\end{verbatim}

\begin{verbatim}
then obtain l where \( l: \exists z. a < z \wedge z < b \wedge \text{DERIV ?h z} :: l \wedge ?h b - ?h a = (b - a) \ast l ..
\end{verbatim}

\begin{verbatim}
then obtain c where \( c: a < c \wedge c < b \wedge \text{DERIV ?h c} :: l \wedge ?h b - ?h a = (b - a) \ast l ..
\end{verbatim}
from $c$ have $cint: a < c \land c < b$ by auto
then obtain $g'c$ where $g'c: \text{DERIV } g c :> g'c$
  using $gd$ real-differentiable-def by blast
from $c$ have $a < c \land c < b$ by auto
then obtain $f'c$ where $f'c: \text{DERIV } f c :> f'c$
  using $fd$ real-differentiable-def by blast

from $c$ have $\text{DERIV } ?h c :> l$ by auto
moreover have $\text{DERIV } ?h c :> (g'c * (f b - f a) - f'c * (g b - g a))$
  using $g'c f'c$ by (auto intro: derivative-eq-intros)
ultimately have $\text{leq: } l = g'c * (f b - f a) - f'c * (g b - g a)$ by (rule $\text{DERIV-unique}$)

have $?h b - ?h a = (b - a) * (g'c * (f b - f a) - f'c * (g b - g a))$
proof
  from $c$ have $?h b - ?h a = (b - a) * l$ by auto
  also from $\text{leq have } \ldots = (b - a) * (g'c * (f b - f a) - f'c * (g b - g a))$ by simp
finally show $\text{?thesis by simp}$
qed
moreover have $?h b - ?h a = 0$
proof
  have $?h b - ?h a = \ldots$
    by (simp add: algebra-simps)
then show $\text{?thesis by auto}$
qed
ultimately have $(b - a) * (g'c * (f b - f a) - f'c * (g b - g a)) = 0$ by auto
with $ab$ have $g'c * (f b - f a) - f'c * (g b - g a) = 0$ by simp
then have $g'c * (f b - f a) = f'c * (g b - g a)$ by simp
then have $(f b - f a) * g'c = (g b - g a) * f'c$ by (simp add: ac-simps)
with $g'c f'c$ $cint$ show $\text{?thesis by auto}$
qed

lemma $\text{GMVT'}$:
fixes $f \ g :: \text{real } \Rightarrow \text{real}$
assumes $a < b$
and $\text{isCont-f: } \forall z. a \leq z \Rightarrow z \leq b \Rightarrow \text{isCont } f z$
and $\text{isCont-g: } \forall z. a \leq z \Rightarrow z \leq b \Rightarrow \text{isCont } g z$
and $\text{DERIV-g: } \forall z. a < z \Rightarrow z < b \Rightarrow \text{DERIV } g z :> (g' z)$
and $\text{DERIV-f: } \forall z. a < z \Rightarrow z < b \Rightarrow \text{DERIV } f z :> (f' z)$
shows $\exists c. a < c \land c < b \land (f b - f a) * g' c = (g b - g a) * f' c$
proof
  have $\exists g'c f'c c. \text{DERIV } g c :> g'c \land \text{DERIV } f c :> f'c \land$
    $a < c \land c < b \land (f b - f a) * g'c = (g b - g a) * f'c$
  using $\text{assms by (intro GMVT') (force simp: real-differentiable-def)}$
then obtain $c$ where $a < c c < b (f b - f a) * g' c = (g b - g a) * f' c$
THEORY "Deriv"

using DERIV-f DERIV-g by (force dest: DERIV-unique)
then show ?thesis
by auto
qed

108.12 L'Hopitals rule

lemma isCont-If-ge:
  fixes a :: 'a :: linorder-topology
  assumes continuous (at-left a) g and f: (f ----> g a) (at-right a)
  shows isCont (λx. if x ≤ a then g x else f x) a (is isCont ?gf a)
proof –
  have g: (g ----> g a) (at-left a)
    using assms continuous-within by blast
  show ?thesis
    unfolding isCont-def continuous-within
    proof (intro filterlim-split-at; simp)
      show (?gf ----> g a) (at-left a)
        by (subst filterlim-cong [OF refl refl], where g=g)
      show (?gf ----> g a) (at-right a)
        by (subst filterlim-cong [OF refl refl], where g=f)
    qed
  qed

lemma lhopital-right-0:
  fixes f0 g0 :: real ⇒ real
  assumes f-0: (f0 ----> 0) (at-right 0)
    and g-0: (g0 ----> 0) (at-right 0)
    and ev:
      eventually (λx. g0 x ≠ 0) (at-right 0)
      eventually (λx. g' x ≠ 0) (at-right 0)
      eventually (λx. DERIV f0 x :> f' x) (at-right 0)
      eventually (λx. DERIV g0 x :> g' x) (at-right 0)
      and lim: filterlim (λ x. (f' x / g' x)) F (at-right 0)
  shows filterlim (λ x. f0 x / g0 x) F (at-right 0)
proof –
  define f where [abs-def]: f x = (if x ≤ 0 then 0 else f0 x) for x
  then have f 0 = 0 by simp

  define g where [abs-def]: g x = (if x ≤ 0 then 0 else g0 x) for x
  then have g 0 = 0 by simp

  have eventually (λx. g0 x ≠ 0 ∧ g' x ≠ 0 ∧
      DERIV f0 x :> (f' x) ∧ DERIV g0 x :> (g' x)) (at-right 0)
    using ev by eventually-elim auto
  then obtain a where [arith]: 0 < a
    and g0-neq-0: ∀x. 0 < x ⇒ x < a ⇒ g0 x ≠ 0
and \( g'\text{-neq-0: } \forall x. \ 0 < x \implies x < a \implies g' x \neq 0 \)

and \( f0: \forall x. \ 0 < x \implies x < a \implies \text{DERIV } f0 \) \( \forall x \implies (f' x) \)

and \( g0: \forall x. \ 0 < x \implies x < a \implies \text{DERIV } g0 \) \( \forall x \implies (g' x) \)

**unfolding eventually-at** by (auto simp: dist-real-def)

have \( g\text{-neq-0: } \forall x. \ 0 < x \implies x < a \implies g x \neq 0 \)

using \( g0\text{-neq-0} \)

**have** \( f: \text{DERIV } f x \implies (f' x) \) if \( x: \ 0 < x x < a \) for \( x \)

using that

by (intro DERIV-cong-ev\[THEN iffD1, OF - - f0\[OF x]\])

(auto simp: f-def eventually-nhds-metric dist-real-def intro: \( \text{exI[of - } x] \))

have \( g: \text{DERIV } g x \implies (g' x) \) if \( x: \ 0 < x x < a \) for \( x \)

using that

by (intro DERIV-cong-ev\[THEN iffD1, OF - - g0\[OF x]\])

(auto simp: g-def eventually-nhds-metric dist-real-def intro: \( \text{exI[of - } x] \))

**have** \( \text{isCont } f0 \)

**unfolding f-def** by (intro isCont-If-ge f0 continuous-const)

**have** \( g0 \)

**unfolding g-def** by (intro isCont-If-ge g0 continuous-const)

**have** \( \exists \xi. \ \forall x \in \{0 <..< a\}. \ 0 < \xi x \land \xi x < x \land f x \land g x = f' (\xi x) / g' (\xi x) \)

**proof** (rule choice, rule ballI)

fix \( x \)

**assume** \( x \in \{0 <..< a\} \)

then **have** \( x[\text{arith}]: \ 0 < x x < a \) by auto

with \( g'\text{-neq-0} \) \( g\text{-neq-0 } \langle g0 = 0 \rangle \) **have** \( g': \forall x. \ 0 < x \implies x < a \implies 0 \neq g' x \)

by auto

**have** \( \forall x. \ 0 \leq x \implies x < a \implies \text{isCont } f x \)

using \( \text{isCont f0} \)

**have** \( \forall x. \ 0 \leq x \implies x < a \implies \text{isCont } g x \)

using \( \text{isCont g0} \)

**ultimately** have \( \exists c. \ 0 < c \land c < x \land (f x - f 0) * g' c = (g x - g 0) * f' c \)

using \( f g \) \( x a \) by (intro GMVT) auto

then **obtain** \( c \) **where** \( 0: \ 0 < c c < x \land (f x - f 0) * g' c = (g x - g 0) * f' c \)

by blast

**moreover**

from \( \ast \) \( g'(1)[of c] \) \( g'(2) \) **have** \( (f x - f 0) / (g x - g 0) = f' c / g' c \)

by \( \text{simp add: field-simps} \)

ultimately show \( \exists y. \ 0 < y \land y < x \land f x / g x = f' y / g' y \)

using \( f 0 = 0 \)

\( \langle g0 = 0 \rangle \) by (auto intro!: \( \text{exI[of - } c] \))

**qed**

then **obtain** \( \xi \) **where** \( \forall x \in \{0 <..< a\}. \ 0 < \xi x \land \xi x < x \land f x / g x = f' (\xi x) / g' (\xi x) \)

then **have** \( \xi: \text{eventually } (\lambda x. \ 0 < \xi x \land \xi x < x \land f x / g x = f' (\xi x) / g' (\xi x)) \)

...
from ζ have eventually (λx. norm (ζ x) ≤ x) (at-right 0)
  by eventually-elim auto

then have ((λx. norm (ζ x)) −→ 0) (at-right 0)
  by (rule-tac real-tendsto-sandwich[where f=λx. 0 and h=λx. x]) auto

then have (ζ −→ 0) (at-right 0)
  by (rule tendsto-norm-zero-cancel)

with ζ have filterlim ζ (at-right 0) (at-right 0)
  by (auto elim!: eventually-mono simp: filterlim-at)

from this lim have filterlim (λt. f ′(ζ t) / g ′(ζ t)) F (at-right 0)
  by (rule-tac filterlim-compose[of - - - ζ])

ultimately have filterlim (λt. f t / g t) F (at-right 0) (is ?P)
  by (rule-tac filterlim-cong[THEN iffD1, OF refl refl])
    (auto elim: eventually-mono)

also have ?P ⇔ ?thesis
  by (rule filterlim-cong) (auto simp: f-def g-def eventually-at-filter)

finally show ?thesis.

qed

lemma lhopital-right:
(f −→ 0) (at-right x) ⇒ (g −→ 0) (at-right x) ⇒
  eventually (λx. g x ≠ 0) (at-right x) ⇒
  eventually (λx. g ′ x ≠ 0) (at-right x) ⇒
  eventually (λx. DERIV f x :> f ′ x) (at-right x) ⇒
  eventually (λx. DERIV g x :> g ′ x) (at-right x) ⇒
  filterlim (λ x. (f ′ x / g ′ x)) F (at-right x) ⇒
  filterlim (λ x. f x / g x) F (at-right x)
for x :: real

unfolding eventually-at-to-0[of - x] filterlim-at-right-to-0[of - x] DERIV-shift
by (rule lhopital-right-0)

lemma lhopital-left:
(f −→ 0) (at-left x) ⇒ (g −→ 0) (at-left x) ⇒
  eventually (λx. g x ≠ 0) (at-left x) ⇒
  eventually (λx. g ′ x ≠ 0) (at-left x) ⇒
  eventually (λx. DERIV f x :> f ′ x) (at-left x) ⇒
  eventually (λx. DERIV g x :> g ′ x) (at-left x) ⇒
  filterlim (λ x. (f ′ x / g ′ x)) F (at-left x) ⇒
  filterlim (λ x. f x / g x) F (at-left x)
for x :: real

unfolding eventually-at-left-to-right filterlim-at-left-to-right DERIV-mirror
by (rule lhopital-right[where f=λx. − f ′(− x)]) (auto simp: DERIV-mirror)

lemma lhopital:
(f −→ 0) (at x) ⇒ (g −→ 0) (at x) ⇒
  eventually (λx. g x ≠ 0) (at x) ⇒
  eventually (λx. g ′ x ≠ 0) (at x) ⇒
THEORY "Deriv"

eventually (λx. DERIV f x :: f’ x) (at x) \implies 

eventually (λx. DERIV g x :: g’ x) (at x) \implies 

filterlim (λ x. (f’ x / g’ x)) F (at x) \implies 

filterlim (λ x. f x / g x) F (at x) 

for x :: real 

unfolding \texttt{eventually-at-split filterlim-at-split} 

by (auto intro!: lhopital-right[of f x g g’ f’] lhopital-left[of f x g g’ f’]) 

\begin{verbatim}
lemma lhopital-right-0-at-top:
  fixes f g :: real ⇒ real
  assumes g-0: LIM x at-right 0. g x :: at-top 
  and ev: 
    eventually (λx. g’ x ≠ 0) (at-right 0) 
    eventually (λx. DERIV f x :: f’ x) (at-right 0) 
    eventually (λx. DERIV g x :: g’ x) (at-right 0) 
  and lim: ((λ x. (f’ x / g’ x)) →→ x) (at-right 0) 
  shows ((λ x. f x / g x) →→ x) (at-right 0) 

unfolding tendsto-iff 
proof safe 
  fix e :: real 
  assume 0 < e 
  with \texttt{lim[unfolded tendsto-iff, rule-format, of e / 4]} 
  have eventually (λt. dist (f’ t / g’ t) x < e / 4) (at-right 0) 
    by simp 
  from eventually-conj[OF eventually-conj[OF ev(1) ev(2)] eventually-conj[OF ev(3) this]] 
  obtain a where [arith]: 0 < a 
    and g’-neq-0: ∀x. 0 < x ⇒ x < a ⇒ g’ x ≠ 0 
    and f0: ∀x. 0 < x ⇒ x ≤ a ⇒ DERIV f x :: (f’ x) 
    and g0: ∀x. 0 < x ⇒ x ≤ a ⇒ DERIV g x :: (g’ x) 
    and Df: ∀t. 0 < t ⇒ t < a ⇒ dist (f’ t / g’ t) x < e / 4 
  unfolding eventually-at-le by (auto simp: dist-real-def) 

from Df have eventually (λt. t < a) (at-right 0) eventually (λt::real. 0 < t) (at-right 0) 
  unfolding eventually-at by (auto intro!: exI[of - a] simp: dist-real-def) 

moreover 
  have eventually (λt. 0 < g t) (at-right 0) eventually (λt. g a < g t) (at-right 0) 
    using g-0 by (auto elim: eventually-mono simp: filterlim-at-top-dense) 

moreover 
  have \texttt{inv-g: ((λx. inverse (g x)) →→ 0) (at-right 0)} 
    using tendsto-inverse-0 filterlim-mono[OF g-0 at-top-le-at-infinity order-refl] 
    by (rule filterlim-compose) 
  then have ((λx. norm (1 – g a * inverse (g x))) →→ norm (1 – g a * 0)) (at-right 0) 
    by (intro tendsto-intros) 
\end{verbatim}

```
THEORY "Deriv"

lemma lhopital-right-at-top

qed

moreover

from inv-g have \((\lambda t. \text{norm} \ (f a - x * g a) * \text{inverse} \ (g t)) \rightarrow \text{norm} \ (f a - x * g a) * 0)\)

(at-right 0)

by (simp add: inverse-eg-divide)

from this[unfolded tendsto-iff, rule-format, of 1]

have eventually \((\lambda x. \text{norm} \ (1 - g a / g x) < 2)\) (at-right 0)

by (auto elim!: eventually-mono simp: dist-real-def)

ultimately show eventually \((\lambda t. \text{dist} \ (f t / g t) \ x < e)\) (at-right 0)

proof (eventually-elim)

fix t assume \(\{\text{arith}\}: 0 < t \ t < a \ g a < g t \ 0 < g t\)

assume ineq: \(\text{norm} \ (1 - g a / g t) < 2 \ \text{norm} \ (f a - x * g a) / \text{norm} \ (g t) < e / 2\)

have \(\exists y. t < y \land y < a \land (g a - g t) * f' y = (f a - f t) * g' y\)

using f0 g0 t(1,2) by (intro GMVT) (force intro!: DERIV-isCont)+

then obtain y where [arith]: \(t < y \ y < a\)

and D-eq0: \((g a - g t) * f' y = (f a - f t) * g' y\)

by blast

from D-eq0 have D-eq: \((f t - f a) / \ (g t - g a) = f' y / g' y\)

using \(g a < g t\) \(g'\)-neq-0[of y] by (auto simp add: field-simps)

have \(\ast: f t / g t - x = ((f t - f a) / (g t - g a) - x) \ast (1 - g a / g t) + (f a - x * g a) / g t\)

by (simp add: field-simps)

have \(\text{norm} \ (f t / g t - x) \leq \text{norm} \ ((f t - f a) / (g t - g a) - x) \ast (1 - g a / g t) + \text{norm} \ ((f a - x * g a) / g t)\)

unfolding * by (rule norm-triangle-ineq)

also have \(\ldots = \text{dist} \ (f' y / g' y) \ x \ast \text{norm} \ (1 - g a / g t) + \text{norm} \ (f a - x * g a) / \text{norm} \ (g t)\)

by (simp add: abs-mult D-eq dist-real-def)

also have \(\ldots < (e / 4) * 2 + e / 2\)

using ineq D[of y] :0 < e by (intro add-le-less-mono mult-mono) auto

finally show \(\text{dist} \ (f t / g t) \ x < e\)

by (simp add: dist-real-def)

qed

qed

lemma lhopital-right-at-top:
THEORY "Deriv"

LIM x at-right x. (g :: real ⇒ real) x :: at-top →
  eventually (λx. g' x ≠ 0) (at-right x) →
  eventually (λx. DERIV f x :: f' x) (at-right x) →
  eventually (λx. DERIV g x :: g' x) (at-right x) →
  (((λ x. f' x / g' x)) → y) (at-right x) →
  (((λ x. f x / g x)) → y) (at-right x)

unfolding eventually-at-right-to-0[of - x] filterlim-at-right-to-0[of - x] DERIV-shift
by (rule lhopital-right-0-at-top)

lemma lhopital-left-at-top:
LIM x at-left x. g x :: at-top →
  eventually (λx. g' x ≠ 0) (at-left x) →
  eventually (λx. DERIV f x :: f' x) (at-left x) →
  eventually (λx. DERIV g x :: g' x) (at-left x) →
  (((λ x. f' x / g' x)) → y) (at-left x) →
  (((λ x. f x / g x)) → y) (at-left x)

for x :: real
unfolding eventually-at-left-to-right filterlim-at-left-to-right DERIV-mirror
by (rule lhopital-right-at-top[where f' = λx. - f' (- x)]) (auto simp: DERIV-mirror)

lemma lhopital-at-top:
LIM x at x. (g :: real ⇒ real) x :: at-top →
  eventually (λx. g' x ≠ 0) (at x) →
  eventually (λx. DERIV f x :: f' x) (at x) →
  eventually (λx. DERIV g x :: g' x) (at x) →
  (((λ x. f' x / g' x)) → y) (at x) →
  (((λ x. f x / g x)) → y) (at x)

unfolding eventually-at-split filterlim-at-split
by (auto intro: lhopital-right-at-top[of g x g' f f' | lhopital-left-at-top[of g x g' f f'])

lemma lhopital-at-top-at-top:
fixes f g :: real ⇒ real
assumes g-0: LIM x at-top. g x :: at-top
  and g': eventually (λx. g' x ≠ 0) at-top
  and Df: eventually (λx. DERIV f x :: f' x) at-top
  and Dg: eventually (λx. DERIV g x :: g' x) at-top
  and lim: (((λ x. f' x / g' x)) → x) at-top
shows (((λ x. f x / g x)) → x) at-top
unfolding filterlim-at-top-to-right

proof (rule lhopital-right-0-at-top)
let ?F = λx. f (inverse x)
let ?G = λx. g (inverse x)
let ?R = at-right (θ :: real)
let ?D = λf'. f' (inverse x) * - (inverse x ` Suc (Suc 0))
show LIM x ?R. ?G x :: at-top
  using g-0 unfolding filterlim-at-top-to-right .
show eventually (λx. DERIV ?G x :: ?D g' x) ?R
  unfolding eventually-at-right-to-top
using $Dg$ eventually-ge-at-top[where $c=1$]
by eventually-elim (rule derivative-eq-intros DERIV-chain'[where $f=inverse$]
| simp)+
show eventually $(\lambda x. \text{DERIV } ?F x :> ?D f' x) ?R$
  unfolding eventually-at-right-to-top
using $Df$ eventually-ge-at-top[where $c=1$]
by eventually-elim (rule derivative-eq-intros DERIV-chain'[where $f=inverse$]
| simp)+
show eventually $(\lambda x. ?D g' x \neq 0) ?R$
  unfolding eventually-at-right-to-top
using $g'$ eventually-ge-at-top[where $c=1$]
by eventually-elim auto
show $((\lambda x. ?D f' x / ?D g' x) \longrightarrow x) ?R$
unfolding filterlim-at-right-to-top
apply (intro filterlim-cong[THEN iffD2, OF refl refl - lim])
using eventually-ge-at-top[where $c=1$]
by eventually-elim simp
qed

lemma lhospital-right-at-top-at-top:
fixes $f$ $g$ :: real $\Rightarrow$ real
assumes $f$-$0$: $\text{LIM } x \text{ at-right } a. f x :> \text{ at-top}$
assumes $g$-$0$: $\text{LIM } x \text{ at-right } a. g x :> \text{ at-top}$
  and ev:
    eventually $(\lambda x. \text{DERIV } f x :> f' x) (\text{at-right } a)$
    eventually $(\lambda x. \text{DERIV } g x :> g' x) (\text{at-right } a)$
  and lim: filterlim $(\lambda x. (f' x / g' x)) \text{ at-top (at-right } a)$
shows filterlim $(\lambda x. f x / g x) \text{ at-top (at-right } a)$
proof −
from lim have pos: eventually $(\lambda x. f' x / g' x > 0) (\text{at-right } a)$
  unfolding filterlim-at-top-dense by blast
have $((\lambda x. g x / f x) \longrightarrow 0) (\text{at-right } a)$
proof (rule lhospital-right-at-top)
  from pos show eventually $(\lambda x. f' x \neq 0) (\text{at-right } a)$ by eventually-elim auto
  from tendsto-inverse-0-at-top[OF lim]
  show $((\lambda x. g' x / f' x) \longrightarrow 0) (\text{at-right } a)$ by simp
qed fact+
moreover from $f$-$0$ $g$-$0$
  have eventually $(\lambda x. f x > 0) (\text{at-right } a)$ eventually $(\lambda x. g x > 0) (\text{at-right } a)$
  unfolding filterlim-at-top-dense by blast+
  hence eventually $(\lambda x. g x / f x > 0) (\text{at-right } a)$ by eventually-elim simp
ultimately have filterlim $(\lambda x. \text{inverse } (g x / f x)) \text{ at-top (at-right } a)$
  by (rule filterlim-inverse-at-top)
thus $?\text{thesis by simp}$
qed

lemma lhospital-right-at-top-at-bot:
fixes $f$ $g$ :: real $\Rightarrow$ real
THEORY "Deriv"

assumes $f\cdot0$: $\text{LIM} \ x \ \text{at-right} \ a. \ f \ x \ := \ \text{at-top}$
assumes $g\cdot0$: $\text{LIM} \ x \ \text{at-right} \ a. \ g \ x \ := \ \text{at-bot}$
and $ev$:
  eventually $(\lambda x. \ \text{DERIV} \ f \ x \ := \ f' \ x) \ \text{(at-right} \ a)$
  eventually $(\lambda x. \ \text{DERIV} \ g \ x \ := \ g' \ x) \ \text{(at-right} \ a)$
and $\lim$: $\text{filterlim} \ (\lambda x. \ (f' \ x / g' \ x)) \ \text{at-bot} \ \text{(at-right} \ a)$
shows $\text{filterlim} \ (\lambda x. \ f \ x / g \ x) \ \text{at-bot} \ \text{(at-right} \ a)$

proof
from $ev(2)$ have $ev':$ eventually $(\lambda x. \ \text{DERIV} \ (\lambda x. \ -g \ x) \ x := -g' \ x) \ \text{(at-right} \ a)$
  by eventually-elim (auto intro: derivative-intros)
have $\text{filterlim} \ (\lambda x. \ f \ x \ / \ (-g \ x)) \ \text{at-top} \ \text{(at-right} \ a)$
  by (rule lhopital-right-at-top-at-top[where $f' = f' \ \text{and} \ g' = \lambda x. \ -g' \ x$])
  (insert assms ev', auto simp: filterlim-uminus-at-bot)
  hence $\text{filterlim} \ (\lambda x. \ -(f \ x / g \ x)) \ \text{at-top} \ \text{(at-right} \ a)$ by simp
thus $?\text{thesis}$ by (simp add: filterlim-uminus-at-bot)
qed

lemma lhopital-left-at-top-at-top:
fixes $f \ g :: \ \text{real} \ \Rightarrow \ \text{real}$
assumes $f\cdot0$: $\text{LIM} \ x \ \text{at-left} \ a. \ f \ x \ := \ \text{at-top}$
assumes $g\cdot0$: $\text{LIM} \ x \ \text{at-left} \ a. \ g \ x \ := \ \text{at-top}$
and $ev$:
  eventually $(\lambda x. \ \text{DERIV} \ f \ x \ := \ f' \ x) \ \text{(at-left} \ a)$
  eventually $(\lambda x. \ \text{DERIV} \ g \ x \ := \ g' \ x) \ \text{(at-left} \ a)$
and $\lim$: $\text{filterlim} \ (\lambda x. \ (f' \ x / g' \ x)) \ \text{at-top} \ \text{(at-left} \ a)$
shows $\text{filterlim} \ (\lambda x. \ f \ x / g \ x) \ \text{at-top} \ \text{(at-left} \ a)$
by (insert assms, unfold eventually-at-left-to-right filterlim-at-left-to-right DERIV-mirror,
  rule lhopital-right-at-top-at-top[where $f' = \lambda x. \ -f' \ (-x)$])
  (insert assms, auto simp: DERIV-mirror)

lemma lhopital-left-at-top-at-bot:
fixes $f \ g :: \ \text{real} \ \Rightarrow \ \text{real}$
assumes $f\cdot0$: $\text{LIM} \ x \ \text{at-left} \ a. \ f \ x \ := \ \text{at-top}$
assumes $g\cdot0$: $\text{LIM} \ x \ \text{at-left} \ a. \ g \ x \ := \ \text{at-bot}$
and $ev$:
  eventually $(\lambda x. \ \text{DERIV} \ f \ x \ := \ f' \ x) \ \text{(at-left} \ a)$
  eventually $(\lambda x. \ \text{DERIV} \ g \ x \ := \ g' \ x) \ \text{(at-left} \ a)$
and $\lim$: $\text{filterlim} \ (\lambda x. \ (f' \ x / g' \ x)) \ \text{at-bot} \ \text{(at-left} \ a)$
shows $\text{filterlim} \ (\lambda x. \ f \ x / g \ x) \ \text{at-bot} \ \text{(at-left} \ a)$
by (insert assms, unfold eventually-at-left-to-right filterlim-at-left-to-right DERIV-mirror,
  rule lhopital-right-at-top-at-bot[where $f' = \lambda x. \ -f' \ (-x)$])
  (insert assms, auto simp: DERIV-mirror)

lemma lhopital-at-top-at-top:
fixes $f \ g :: \ \text{real} \ \Rightarrow \ \text{real}$
assumes $f\cdot0$: $\text{LIM} \ x \ \text{at} \ a. \ f \ x \ := \ \text{at-top}$
assumes $g\cdot0$: $\text{LIM} \ x \ \text{at} \ a. \ g \ x \ := \ \text{at-top}$
and $ev$:
THEORY “NthRoot”

eventually (λx. DERIV f x := f' x) (at a)
eventually (λx. DERIV g x := g' x) (at a)
and lim: filterlim (λ x. (f' x / g' x)) at-top (at a)
shows filterlim (λ x. f x / g x) at-top (at a)
using assms unfolding eventually-at-split filterlim-at-split
by (auto intro!: lhopital-right-at-top-at-top[of f a g f' g']
  lhopital-left-at-top-at-top[of f a g f' g'])

lemma lhopital-at-top-at-bot:
fixes f g :: real
assumes f-0: LIM x at a. f x := at-top
assumes g-0: LIM x at a. g x := at-bot
and ev:
  eventually (λx. DERIV f x := f' x) (at a)
eventually (λx. DERIV g x := g' x) (at a)
and lim: filterlim (λ x. (f' x / g' x)) at-bot (at a)
shows filterlim (λ x. f x / g x) at-bot (at a)
using assms unfolding eventually-at-split filterlim-at-split
by (auto intro!: lhopital-right-at-top-at-bot[of f a g f' g']
  lhopital-left-at-top-at-bot[of f a g f' g'])

end

109  Nth Roots of Real Numbers

theory NthRoot
  imports Deriv
begin

109.1  Existence of Nth Root

Existence follows from the Intermediate Value Theorem

lemma realpow-pos-nth:
fixes a :: real
assumes n: 0 < n
and a: 0 < a
shows ∃ r>0. r ^ n = a
proof -
  have ∃ r>0. r ≤ (max 1 a) ∧ r ^ n = a
  proof (rule IVT)
    show 0 ≤ n ≤ a
      using n a by (simp add: power-0-left)
    show 0 ≤ max 1 a
      by simp
    from n have n1: 1 ≤ n
      by simp
    have a ≤ max 1 a ^ 1
      by simp
  end

end
also have \( \max 1 a \cdot 1 \leq \max 1 a \cdot n \n\)
using \( n1 \) by (rule power-increasing) simp
finally show \( a \leq \max 1 a \cdot n \n\).
show \( \forall r. 0 \leq r \land r \leq \max 1 a \to isCont (\lambda x. x \cdot n) \) r
by simp
qed
then obtain \( r \) where \( r \cdot n = a \n\)
by fast
with \( n \) a have \( r \neq 0 \n\)
by (auto simp add: power-0-left)
with \( r \) have \( 0 < r \land r \cdot n = a \n\)
by simp
then show \( \text{thesis} \n\).
qed

lemma realpow-pos-nth2: \((\theta::real) < a \Longrightarrow \exists r>0. r \cdot \text{Suc} n = a \\)
by (blast intro: realpow-pos-nth)

Uniqueness of nth positive root.
lemma realpow-pos-nth-unique: \(0 < n \Longrightarrow 0 < a \Longrightarrow \exists! r. 0 < r \land r \cdot n = a \\)
for \( a :: \text{real} \n\)
by (auto intro!: realpow-pos-nth simp: power-eq-iff-eq-base)

109.2 Nth Root

We define roots of negative reals such that \( \text{root} n (-x) = - \text{root} n x \). This allows us to omit side conditions from many theorems.

lemma inj-sgn-power:
assumes \( 0 < n \n\)
shows \( \text{inj} (\lambda y. \text{sgn} y \cdot |y| \cdot n :: \text{real}) \n\)
(is inj \( f \) )
proof (rule injI)
have \( x: (0 < a \land b < 0) \lor (a < 0 \land 0 < b) \Longrightarrow a \neq b \n\)
for \( a b :: \text{real} \n\)
by auto
fix \( x y \n\)
assume \( \forall x y \n\)
with \( \text{power-eq-iff-eq-base[of n |x| |y| |0 < n| \) show} x = y \n\)
by (cases rule: linorder-cases[of 0 x, case-product linorder-cases[of 0 y]]
(simp-all add: x)
qed

lemma sgn-power-injE:
\( \text{sgn} a \cdot |a| \cdot n = x \Longrightarrow x = \text{sgn} b \cdot |b| \cdot n \Longrightarrow 0 < n \Longrightarrow a = b \n\)
for \( a b :: \text{real} \n\)
using inj-sgn-power[THEN injD, of n a b] by simp

definition root :: \( \text{nat} \Rightarrow \text{real} \Rightarrow \text{real} \n\)
where \( \text{root } n \ x = (\text{if } n = 0 \text{ then } 0 \text{ else the-inv } (\lambda y. \text{sgn } y \ast |y| \ ^n) \ x) \)

**lemma root-0** [simp]: \( \text{root } 0 \ x = 0 \)
by (simp add: root-def)

**lemma root-sgn-power**: \( 0 < n \implies \text{root } n \ (\text{sgn } y \ast |y| \ ^n) = y \)
using the-inv-f-f [OF inj-sgn-power] by (simp add: root-def)

**lemma sgn-power-root**: assumes \( 0 < n \)
shows \( \text{sgn } (\text{root } n \ x) \ast |(\text{root } n \ x)| \ ^n = x \)
(is \( \text{if } (\text{root } n \ x) = x \)
proof (cases \( x = 0 \)
  case True
  with assms root-sgn-power[of n 0]
  show \( \text{thesis} \)
by simp
next
  case False
  with realpow-pos-nth[of \( n \) \( 0 \) \( x \)]
  obtain \( r \) where \( 0 < r \ast r \ ^n = |x| \)
  by auto
  with \( x \neq 0 \) have \( S \): \( x \in \text{range } \text{if} \)
  (auto simp: sgn-zero-iff)
  (auto simp: sgn-power-root)
from \( 0 < n \ast \text{f-the-inv-into-f[of } n \ast 0 \text{]} \) this show \( \text{thesis} \)
by (simp add: root-def)

**lemma split-root**: \( P (\text{root } n \ x) \iff (n = 0 \longrightarrow P \ 0) \land (0 < n \longrightarrow (\forall y. \text{sgn } y \ast |y| \ ^n = x \longrightarrow P y)) \)
proof (cases \( n = 0 \)
  case True
  then show \( \text{thesis} \)
by simp
next
  case False
  then show \( \text{thesis} \)
by simp (metis root-sgn-power sgn-power-root)

**lemma real-root-zero** [simp]: \( \text{root } n \ 0 = 0 \)
by (simp split: split-root add: sgn-zero-iff)

**lemma real-root-minus**: \( \text{root } n \ (-x) = -\text{root } n \ x \)
by (clarsimp split: split-root elim!: sgn-power-injE simp: sgn-minus)

**lemma real-root-less-mono**: \( 0 < n \implies x < y \implies \text{root } n \ x < \text{root } n \ y \)
proof (clarsimp split: split-root)
  have \*: \( 0 < b \implies a < 0 \implies \neg a > b \)
  for \( a b :: \text{real} \)
  by auto

THEORY "NthRoot"

fix a b :: real
assume 0 < n sgn a * |a| ^ n < sgn b * |b| ^ n
then show a < b
  using power-less-imp-less-base[of a n b]
  power-less-imp-less-base[of - b n - a]
  by (simp add: sgn-real-def * [of a ^ n - ((- b) ^ n)]
  split: if-split-asm)
qed

lemma real-root-gt-zero: 0 < n ⇒ 0 < x ⇒ 0 < root n x
  using real-root-less-mono[of n 0 x]
  by simp

lemma real-root-ge-zero: 0 ≤ x ⇒ 0 ≤ root n x
  using real-root-gt-zero[of n x]
  by (cases n = 0) (auto simp add: le-less)

lemma real-root-pow-pos: 0 < n ⇒ 0 < x ⇒ root n x ^ n = x
  using sgn-power-root[of n x] real-root-gt-zero[of n x]
  by simp

lemma real-root-pow-pos2: 0 < n ⇒ 0 ≤ x ⇒ root n x ^ n = x
  by (auto simp add: order-le-less real-root-pow-pos)

lemma sgn-root: 0 < n ⇒ sgn (root n x) = sgn x
  by (auto split: split-root simp: sgn-real-def)

lemma odd-real-root-pow: odd n ⇒ root n x ^ n = x
  using sgn-power-root[of n x]
  by (simp add: odd-pos sgn-real-def split: if-split-asm)

lemma real-root-power-cancel: 0 < n ⇒ 0 < x ⇒ root n (x ^ n) = x
  using root-sgn-power[of n x]
  by (auto simp add: le-less power-0-left)

lemma odd-real-root-power-cancel: odd n ⇒ root n (x ^ n) = x
  using root-sgn-power[of n x]
  by (simp add: odd-pos sgn-real-def split: if-split-asm)

lemma real-root-pos-unique: 0 < n ⇒ 0 ≤ y ⇒ y ^ n = x ⇒ root n x = y
  using root-sgn-power[of n y]
  by (auto simp add: le-less power-0-left)

lemma odd-real-root-unique: odd n ⇒ y ^ n = x ⇒ root n x = y
  by (erule subst, rule odd-real-root-power-cancel)

lemma real-root-one [simp]: 0 < n ⇒ root n 1 = 1
  by (simp add: real-root-pos-unique)

Root function is strictly monotonic, hence injective.

lemma real-root-le-mono: 0 < n ⇒ x ≤ y ⇒ root n x ≤ root n y
  by (auto simp add: order-le-less real-root-less-mono)
lemma real-root-less-iff [simp]: $0 < n \implies \sqrt[n]{x} < \sqrt[n]{y} \iff x < y$
by (cases $x < y$) (simp-all add: real-root-less-mono linorder-not-less real-root-le-mono)

lemma real-root-le-iff [simp]: $0 < n \implies \sqrt[n]{x} \leq \sqrt[n]{y} \iff x \leq y$
by (cases $x \leq y$) (simp-all add: real-root-le-mono linorder-not-le real-root-less-mono)

lemma real-root-eq-iff [simp]: $0 < n \implies \sqrt[n]{x} = \sqrt[n]{y} \iff x = y$
by (simp add: order-eq-iff)

lemmas real-root-gt-0-iff [simp] = real-root-less-iff [where $x = 0$, simplified]
lemmas real-root-lt-0-iff [simp] = real-root-less-iff [where $y = 0$, simplified]
lemmas real-root-ge-0-iff [simp] = real-root-le-iff [where $x = 0$, simplified]
lemmas real-root-le-0-iff [simp] = real-root-le-iff [where $y = 0$, simplified]
lemmas real-root-eq-0-iff [simp] = real-root-eq-iff [where $y = 0$, simplified]

lemma real-root-gt-1-iff [simp]: $0 < n \implies 1 < \sqrt[n]{y} \iff 1 < y$
using real-root-less-iff [where $x = 1$] by simp

lemma real-root-lt-1-iff [simp]: $0 < n \implies \sqrt[n]{x} < 1 \iff x < 1$
using real-root-less-iff [where $y = 1$] by simp

lemma real-root-ge-1-iff [simp]: $0 < n \implies 1 \leq \sqrt[n]{y} \iff 1 \leq y$
using real-root-le-iff [where $x = 1$] by simp

lemma real-root-le-1-iff [simp]: $0 < n \implies \sqrt[n]{x} \leq 1 \iff x \leq 1$
using real-root-le-iff [where $y = 1$] by simp

lemma real-root-eq-1-iff [simp]: $0 < n \implies \sqrt[n]{x} = 1 \iff x = 1$
using real-root-eq-iff [where $y = 1$] by simp

Roots of multiplication and division.
lemma real-root-mult: root $n$ $(x \cdot y) = root n x \cdot root n y$
by (auto split: split-root elim!: sgn-power-injE
  simp: sgn-mult abs-mult power-mult-distrib)

lemma real-root-inverse: root $n$ $(inverse x) = inverse (root n x)$
by (auto split: split-root elim!: sgn-power-injE
  simp: power-inverse)

lemma real-root-divide: root $n$ $(x / y) = root n x / root n y$
by (simp add: divide-inverse real-root-mult real-root-inverse)

lemma real-root-abs: $0 < n \implies root n |x| = |root n x|$
by (simp add: abs-if real-root-minus)

lemma root-abs-power: $n > 0 \implies abs (root n (y ^ n)) = abs y$
using root-sgn-power [of n]
by (metis abs-ge-zero power-abs real-root-abs real-root-power-cancel)
lemma real-root-power: \(0 < n \implies \sqrt[n]{x^k} = \sqrt[n]{x}^k\)
by (induct k) (simp-all add: real-root-mul)

Roots of roots.

lemma real-root-Suc-0 [simp]: \(\sqrt{\text{Suc } 0} x = x\)
by (simp add: odd-real-root-unique)

lemma real-root-mult-exp: \(\sqrt{m \cdot n} x = \sqrt{m} (\sqrt{n} x)\)
by (auto split: split-root elim!: sgn-power-injE simp add: symmetric)

lemma real-root-commute: \(\sqrt{m} (\sqrt{n} x) = \sqrt{n} (\sqrt{m} x)\)
by (simp add: real-root-mult-exp symmetric mult.commute)

Monotonicity in first argument.

lemma real-root-strict-decreasing:
assumes \(0 < n\ \&\ n < N\ 1 < x\)
shows \(\sqrt{N} x < \sqrt{n} x\)
proof –
from assms have \(\sqrt{n} (\sqrt{N} x) ^ n < \sqrt{n} (\sqrt{n} x) ^ N\)
by (simp add: real-root-commute power-strict-increasing del: real-root-pow-pos2)
with assms show \(?thesis by simp\)
qed

lemma real-root-strict-increasing:
assumes \(0 < n\ 0 < N\ 0 < x x < 1\)
shows \(\sqrt{n} x < \sqrt{N} x\)
proof –
from assms have \(\sqrt{N} (\sqrt{n} x) ^ n < \sqrt{n} (\sqrt{n} x) ^ N\)
by (simp add: real-root-commute power-strict-decreasing del: real-root-pow-pos2)
with assms show \(?thesis by simp\)
qed

lemma real-root-decreasing: \(0 < n \implies n \leq N \implies 1 \leq x \implies \sqrt{N} x \leq \sqrt{n} x\)
by (auto simp add: order-le-less real-root-strict-decreasing)

lemma real-root-increasing: \(0 < n \implies n \leq N \implies 0 \leq x \implies x \leq 1 \implies \sqrt{n} x \leq \sqrt{N} x\)
by (auto simp add: order-le-less real-root-strict-increasing)

Continuity and derivatives.

lemma isCont-real-root: isCont (root n) x
proof (cases n > 0)
case True
let \(f = \lambda y::real. \text{sgn } y \cdot |y|^n\)
have continuous-on \(\{0..\} \cup \{.. 0\}\) \((\lambda x. \text{if } 0 < x \text{ then } x ^ n \text{ else } -((-x) ^ n)\) :: real)
using True by (intro continuous-on-If continuous-intros) auto
then have continuous-on UNIV ?f
  by (rule continuous-on-cong[THEN iffD1, rotated 2]) (auto simp: not-less le-less True)
then have [simp]: ?f x for x
  by (simp add: continuous-on-eq-continuous-at)
have isCont (root n) (?f (root n x))
  by (rule isCont-inverse-function [where f=?f and d=1]) (auto simp: root-sgn-power)
then show ?thesis
  by (simp add: sgn-power-root True)
next
case False then show ?thesis
  by (simp add: root-def [abs-def])
qed

lemma tendsto-real-root [tendsto-intros]:
(f ----> x) F ==> ((\lam x. root n (f x)) ----> root n x) F
using isCont-tendsto-compose[OF isCont-real-root, of f x F].

lemma continuous-real-root [continuous-intros]:
continuous F f ==> continuous F (\lam x. root n (f x))
unfolding continuous-def by (rule tendsto-real-root)

lemma continuous-on-real-root [continuous-intros]:
continuous-on s f ==> continuous-on s (\lam x. root n (f x))
unfolding continuous-on-def by (auto intro: tendsto-real-root)

lemma DERIV-real-root:
assumes n: 0 < n
  and x: 0 < x
shows DERIV (root n) x :> inverse (real n * root n x ^ (n - Suc 0))
proof (rule DERIV-inverse-function)
  show 0 < x
    using x .
  show x < x + 1
    by simp
  show DERIV (\lam x. x ^ n) (root n x) :> real n * root n x ^ (n - Suc 0)
    by (rule DERIV-pow)
  show real n * root n x ^ (n - Suc 0) 0
    using n x by simp
  show isCont (root n) x
    by (rule isCont-real-root)
qed (use n in auto)

lemma DERIV-odd-real-root:
assumes n: odd n
  and x: x 0

shows $\text{DERIV} (\root n \of x) x :> \text{inverse} \ (\text{real} \ n \ \ast \ \root n \of x \ ^ \ (n - \text{Suc} \ 0))$

**proof** (rule DERIV-inverse-function)

- show $x - 1 < x < x + 1$
  - by auto
- show $\text{DERIV} (\lambda x. \ x ^ n) (\root n \of x) x :> \text{real} \ n \ \ast \ \root n \of x ^ (n - \text{Suc} \ 0)$
  - by (rule DERIV-pow)
- show $\text{real} \ n \ \ast \ \root n \of x ^ (n - \text{Suc} \ 0) \neq 0$
  - using odd-pos [OF n] $x$ by simp
- show $\text{isCont} \ (\root n) x$
  - by (rule isCont-real-root)

**qed** (use $n$ odd-real-root-pow in auto)

**lemma** $\text{DERIV-even-real-root}$:

- assumes $n: 0 < n$
  - and even $n$
  - and $x: x < 0$
- shows $\text{DERIV} (\root n \of x) x :> \text{inverse} \ (- \ \text{real} \ n \ \ast \ \root n \of x \ ^ (n - \text{Suc} \ 0))$

**proof** (rule DERIV-inverse-function)

- show $x - 1 < x$
  - by simp
- show $x < 0$
  - using $x$.
- show $-(\root n \of y ^ n) = y \ \text{if} \ x - 1 < y \ \text{and} \ y < 0 \ \text{for} \ y$

**proof** $-$

- have $\root n (-y) ^ n = -y$
  - using that $0 < n$ by simp
- with real-root-minus and (even $n$)
- show $-(\root n \of y ^ n) = y$ by simp

**qed**

**show** $\text{DERIV} (\lambda x. \ -(x ^ n)) (\root n \of x) x :> - \ \text{real} \ n \ \ast \ \root n \of x ^ (n - \text{Suc} \ 0)$

- by (auto intro!: derivative-eq-intros)
- show $\text{real} \ n \ \ast \ \root n \of x ^ (n - \text{Suc} \ 0) \neq 0$
  - using $n x$ by simp
- show $\text{isCont} \ (\root n) x$
  - by (rule isCont-real-root)

**qed**

**lemma** $\text{DERIV-real-root-generic}$:

- assumes $0 < n$
  - and $x \neq 0$
  - and even $n \ \Longrightarrow \ 0 < x \ \Longrightarrow \ D = \text{inverse} \ (\text{real} \ n \ \ast \ \root n \of x ^ (n - \text{Suc} \ 0))$
  - and even $n \ \Longrightarrow \ x < 0 \ \Longrightarrow \ D = - \ \text{inverse} \ (\text{real} \ n \ \ast \ \root n \of x ^ (n - \text{Suc} \ 0))$
  - and odd $n \ \Longrightarrow \ D = \text{inverse} \ (\text{real} \ n \ \ast \ \root n \of x ^ (n - \text{Suc} \ 0))$
- shows $\text{DERIV} (\root n \of x) x :> D$

**using** assms

- by (cases even $n$, cases $0 < x$)
  - (auto intro: DERIV-real-root[THEN DERIV-cong]
    DERIV-odd-real-root[THEN DERIV-cong]
    DERIV-even-real-root[THEN DERIV-cong])

**qed**
lemma power-tendsto-0-iff [simp]:
  fixes f :: 'a ⇒ real
  assumes n > 0
  shows ((λx. f x ^ n) −−→ 0) F ⇔ (f −−→ 0) F
proof –
  have ((λx. |root n (f x ^ n)|) −−→ 0) F ⇒ (f −−→ 0) F
    by (auto simp: assms root-abs-power tendsto-rabs-zero-iff)
  then have ((λx. f x ^ n) −−→ 0) F ⇒ (f −−→ 0) F
    by (metis tendsto-real-root-abs-0 real-root-zero tendsto-rabs)
  with assms show ?thesis
    by (auto simp: tendsto-null-power)
qed

109.3 Square Root

definition sqrt :: real ⇒ real
  where sqrt = root 2

lemma pos2: 0 < (2::nat)
  by simp

lemma real-sqrt-unique: y^2 = x ⇒ 0 ≤ y ⇒ sqrt x = y
  unfolding sqrt-def by (rule real-root-pos-unique [OF pos2])

lemma real-sqrt-abs [simp]: sqrt (x^2) = |x|
  by (metis power2-abs abs-ge-zero real-sqrt-unique)

lemma real-sqrt-pow2 [simp]: 0 ≤ x ⇒ (sqrt x)^2 = x
  unfolding sqrt-def by (rule real-root-pow-pos2 [OF pos2])

lemma real-sqrt-pow2-iff [simp]: (sqrt x)^2 = x ⇔ 0 ≤ x
  by (metis real-sqrt-pow2-zero-le-power2)

lemma real-sqrt-zero [simp]: sqrt 0 = 0
  unfolding sqrt-def by (rule real-root-zero)

lemma real-sqrt-one [simp]: sqrt 1 = 1
  unfolding sqrt-def by (rule real-root-one [OF pos2])

lemma real-sqrt-four [simp]: sqrt 4 = 2
  using real-sqrt-abs[of 2] by simp

lemma real-sqrt-minus: sqrt (−x) = −sqrt x
  unfolding sqrt-def by (rule real-root-minus)

lemma real-sqrt-mult: sqrt (x * y) = sqrt x * sqrt y
  unfolding sqrt-def by (rule real-root-mult)
lemma real-sqrt-mult-self [simp]: \( \sqrt{a} \cdot \sqrt{a} = |a| \)
using real-sqrt-apply[of a] unfolding power2_eq_square real-sqrt-mult .

lemma real-sqrt-inverse: \( \sqrt{\frac{1}{x}} = \frac{1}{\sqrt{x}} \)
unfolding sqrt-def by (rule real-root-inverse)

lemma real-sqrt-divide: \( \sqrt{x / y} = \sqrt{x} / \sqrt{y} \)
unfolding sqrt-def by (rule real-root-divide)

lemma real-sqrt-power: \( \sqrt{x^k} = \sqrt{x}^k \)
unfolding sqrt-def by (rule real-root-power[OF pos2])

lemma real-sqrt-gt-zero: \( 0 < x \Rightarrow 0 < \sqrt{x} \)
unfolding sqrt-def by (rule real-root-gt-zero[OF pos2])

lemma real-sqrt-ge-zero: \( 0 \leq x \Rightarrow 0 \leq \sqrt{x} \)
unfolding sqrt-def by (rule real-root-ge-zero)

lemma real-sqrt-less-mono: \( x < y \Rightarrow \sqrt{x} < \sqrt{y} \)
unfolding sqrt-def by (rule real-root-less-mono[OF pos2])

lemma real-sqrt-le-mono: \( x \leq y \Rightarrow \sqrt{x} \leq \sqrt{y} \)
unfolding sqrt-def by (rule real-root-le-mono[OF pos2])

lemma real-sqrt-less-iff [simp]: \( \sqrt{x} < \sqrt{y} \iff x < y \)
unfolding sqrt-def by (rule real-root-less-iff[OF pos2])

lemma real-sqrt-le-iff [simp]: \( \sqrt{x} \leq \sqrt{y} \iff x \leq y \)
unfolding sqrt-def by (rule real-root-le-iff[OF pos2])

lemma real-sqrt-eq-iff [simp]: \( \sqrt{x} = \sqrt{y} \iff x = y \)
unfolding sqrt-def by (rule real-root-eq-iff[OF pos2])

lemma real-less-lsqrt: \( 0 \leq x \Rightarrow 0 \leq y \Rightarrow x < y^2 \Rightarrow \sqrt{x} < y \)
using real-sqrt-less-iff[of x y2] by simp

lemma real-le-lsqrt: \( 0 \leq x \Rightarrow 0 \leq y \Rightarrow x \leq y^2 \Rightarrow \sqrt{x} \leq y \)
using real-sqrt-le-iff[of x y2] by simp

lemma real-sqrt-power-even: assumes even n x \( \geq 0 \)
shows \( \sqrt{x^n} = x^{n \div 2} \)
proof –
from assms obtain k where \( n = 2 \cdot k \) by (auto elim: evenE)
with assms show ?thesis by (simp add: power-mult)
qed

lemma sqrt-le-D: \( \sqrt{x} \leq y \implies x \leq y^2 \)
by (meson not-le real-less-rsqrt)

lemma sqrt-ge-absD: \(|x| \leq \sqrt{y} \implies x^2 \leq y \)
using real-sqrt-le-iff[of x^2] by simp

lemma sqrt-even-pow2:
assumes \( n: \text{even} n \)
shows \( \sqrt{2^n} = 2^{\frac{n}{2}} \)
proof
from n obtain m where \( m: n = 2 \cdot m \)
from m have \( \sqrt{2^n} = \sqrt{(2^m)^2} \)
by (simp only: power-mult[symmetric] mult.commute)
then show ?thesis
using m by simp
qed

lemmas real-sqrt-gt-0-iff [simp] = real-sqrt-less-iff [where x=0, unfolded real-sqrt-zero]
lemmas real-sqrt-lt-0-iff [simp] = real-sqrt-less-iff [where y=0, unfolded real-sqrt-zero]
lemmas real-sqrt-ge-0-iff [simp] = real-sqrt-le-iff [where x=0, unfolded real-sqrt-zero]
lemmas real-sqrt-le-0-iff [simp] = real-sqrt-le-iff [where y=0, unfolded real-sqrt-zero]
lemmas real-sqrt-eq-0-iff [simp] = real-sqrt-eq-iff [where y=0, unfolded real-sqrt-one]
lemmas real-sqrt-gt-1-iff [simp] = real-sqrt-less-iff [where x=1, unfolded real-sqrt-zero]
lemmas real-sqrt-lt-1-iff [simp] = real-sqrt-less-iff [where y=1, unfolded real-sqrt-one]
lemmas real-sqrt-ge-1-iff [simp] = real-sqrt-le-iff [where x=1, unfolded real-sqrt-zero]
lemmas real-sqrt-le-1-iff [simp] = real-sqrt-le-iff [where y=1, unfolded real-sqrt-one]
lemmas real-sqrt-eq-1-iff [simp] = real-sqrt-eq-iff [where y=1, unfolded real-sqrt-one]

lemma sqrt-add-le-add-sqrt:
assumes \( 0 \leq x \leq y \)
shows \( \sqrt{x + y} \leq \sqrt{x} + \sqrt{y} \)
by (rule power2-le-imp-le) (simp-all add: power2-sum assms)

lemma isCont-real-sqrt: isCont \( \sqrt{x} \)
unfolding sqrt-def by (rule isCont-real-root)

lemma tendsto-real-sqrt [tendsto-intros]:
\( (f \longrightarrow x) \ F \implies ((\lambda x. \sqrt{f x}) \longrightarrow \sqrt{x}) \ F \)
unfolding sqrt-def by (rule tendsto-real-root)

lemma continuous-real-sqrt [continuous-intros]:
continuous \( F f \implies \text{continuous} F (\lambda x. \sqrt{f x}) \)
unfolding sqrt-def by (rule continuous-real-root)
lemma continuous-on-real-sqrt [continuous-intros]:
continuous-on s f \implies continuous-on s (\lambda x. sqrt (f x))
utfolding sqrt-def by (rule continuous-on-real-root)

lemma DERIV-real-sqrt-generic:
assumes x \neq 0
  and x > 0 \implies D = inverse (sqrt x) / 2
  and x < 0 \implies D = - inverse (sqrt x) / 2
shows DERIV sqrt x :> D
using assms unfolding sqrt-def
by (auto intro!: DERIV-real-root-generic)

lemma DERIV-real-sqrt: 0 < x \implies DERIV sqrt x :> inverse (sqrt x) / 2
using DERIV-real-sqrt-generic by simp

declare 
DERIV-real-sqrt-generic[THEN DERIV-chain2, derivative-intros]
DERIV-real-root-generic[THEN DERIV-chain2, derivative-intros]

lemmas has-derivative-real-sqrt[derivative-intros] = DERIV-real-sqrt[THEN DERIV-compose-FDERIV]

lemma not-real-square-gt-zero [simp]: \neg 0 < x * x \iff x = 0
for x :: real
  apply auto
  using linorder-less-linear [where x = x and y = 0]
  apply (simp add: zero-less-mult-iff)
done

lemma real-sqrt-abs2 [simp]: sqrt (x * x) = |x|
  apply (subst power2-eq-square [symmetric])
  apply (rule real-sqrt-abs)
done

lemma real-inv-sqrt-pow2: 0 < x \implies (inverse (sqrt x))^2 = inverse x
by (simp add: power-inverse)

lemma real-sqrt-eq-zero-cancel: 0 \leq x \implies sqrt x = 0 \implies x = 0
by simp

lemma real-sqrt-ge-one: 1 \leq x \implies 1 \leq sqrt x
by simp

lemma sqrt-divide-self-eq:
assumes nneg: 0 \leq x
shows sqrt x / x = inverse (sqrt x)
proof (cases x = 0)
  case True
  then show ?thesis by simp
next
case False
then have pos: 0 < x
  using nneg by arith
show ?thesis
proof (rule right-inverse-eq [THEN iffD1, symmetric])
  show sqrt x / x ≠ 0
    by (simp add: divide-inverse nneg False)
  show inverse (sqrt x) / (sqrt x / x) = 1
    by (simp add: divide-inverse mult_assoc [symmetric]
        power2-eq-square [symmetric] real-inv-sqrt-pow2 pos False)
qed

lemma real-div-sqrt: 0 ≤ x ⇒ x / sqrt x = sqrt x
  by (cases x = 0) (simp-all add: sqrt-divide-self-eq
        of x field-simps)

lemma real-divide-square-eq [simp]: (r * a) / (r * r) = a / r
  for a r :: real
  by (cases r = 0) (simp-all add: divide-inverse ac-simps)

lemma lemma-real-divide-sqrt-less: 0 < u ⇒ u / sqrt 2 < u
  by (simp add: divide-less-eq)

lemma four-x-squared: 4 * x^2 = (2 * x)^2
  for x :: real
  by (simp add: power2-eq-square)

lemma sqrt-at-top: LIM x at-top. sqrt x :: real ⇒ at-top
  by (rule filterlim-at-top-at-top[where Q=λx. True and P=λx. 0 < x and
g=power2])
      (auto intro: eventually-gt-at-top)

109.4 Square Root of Sum of Squares

lemma sum-squares-bound: 2 * x * y ≤ x^2 + y^2
  for x y :: 'a::linordered-field
proof –
  have (x - y)^2 = x * x - 2 * x * y + y * y
    by algebra
  then have 0 ≤ x^2 - 2 * x * y + y^2
    by (metis sum-power2-ge-zero zero-le-double-add-iff-zero-le-single-add power2-eq-square)
  then show ?thesis
    by arith
qed

lemma arith-geo-mean:
  fixes u :: 'a::linordered-field
  assumes u^2 = x * y x ≥ 0 y ≥ 0
  shows u ≤ (x + y)/2
apply (rule power2-le-imp-le)
using sum-squares-bound assms
apply (auto simp: zero-le-mult-iff)
apply (auto simp: algebra-simps power2-eq-square)
done

lemma arith-geo-mean-sqrt:
  fixes x :: real
  assumes x ≥ 0 y ≥ 0
  shows sqrt (x * y) ≤ (x + y) / 2
  apply (rule arith-geo-mean)
  using assms
  apply (auto simp: zero-le-mult-iff)
done

lemma real-sqrt-sum-squares-mult-ge-zero [simp]: 0 ≤ sqrt (x^2 + y^2) * (x * y)
  by (metis real-sqrt-ge-0-iff split-mult-pos-le sum-power2-ge-zero)

lemma real-sqrt-sum-squares-mult-squared-eq: (sqrt (x^2 + y^2) * (x * y))^2 = x^2 + y^2 * (x * y)
  by (simp add: zero-le-mult-iff)

lemma real-sqrt-sum-squares-eq-cancel: sqrt (x^2 + y^2) = x =⇒ y = 0
  by (drule arg-cong [where f = λx. x^2]) simp

lemma real-sqrt-sum-squares-eq-cancel2: sqrt (x^2 + y^2) = y =⇒ x = 0
  by (drule arg-cong [where f = λx. x^2]) simp

lemma real-sqrt-ge-abs1 [simp]: |x| ≤ sqrt (x^2 + y^2)
  by (rule power2-le-imp-le simp-all)

lemma real-sqrt-ge-abs2 [simp]: |y| ≤ sqrt (x^2 + y^2)
  by (rule power2-le-imp-le simp-all)

lemma le-real-sqrt-sumsq [simp]: x ≤ sqrt (x * x + y * y)
  by (simp add: power2-eq-square [symmetric])

lemma sqrt-sum-squares-le-sum:
  [0 ≤ x; 0 ≤ y] =⇒ sqrt (x^2 + y^2) ≤ x + y
  by (rule power2-le-imp-le) (simp-all add: power2-sum)

lemma L2-set-mult-ineq-lemma:
fixes $a\ b\ c\ d :: \text{real}$
shows $2 \ast (a \ast c) \ast (b \ast d) \leq a^2 \ast d^2 + b^2 \ast c^2$
proof
   have $0 \leq (a \ast d - b \ast c)^2$ by simp
   also have $\ldots = a^2 \ast d^2 + b^2 \ast c^2 - 2 \ast (a \ast d) \ast (b \ast c)$
      by (simp only: power2-diff power-mult-distrib)
   also have $\ldots = a^2 \ast d^2 + b^2 \ast c^2 - 2 \ast (a \ast c) \ast (b \ast d)$
      by simp
   finally show $2 \ast (a \ast c) \ast (b \ast d) \leq a^2 \ast d^2 + b^2 \ast c^2$
      by simp
qed

lemma sqrt-sum-squares-le-sum-abs: $\sqrt{x^2 + y^2} \leq |x| + |y|
by (rule power2-le-imp-le) (simp-all add: power2-sum)

lemma real-sqrt-sum-squares-triangle-ineq:
$sqrt ((a + c)^2 + (b + d)^2) \leq sqrt (a^2 + b^2) + sqrt (c^2 + d^2)$
proof
   have $(a + c + b + d) \leq (sqrt (a^2 + b^2) \ast sqrt (c^2 + d^2))$
      by (rule power2-le-imp-le) (simp-all add: power2-sum power-mult-distrib ring-distribs
L2-set-mult-ineq-lemma add.commute)
   then have $(a + c)^2 + (b + d)^2 \leq (sqrt (a^2 + b^2) + sqrt (c^2 + d^2))^2$
      by (simp add: power2-sum)
   then show ?thesis
      by (auto intro: power2-le-imp-le)
qed

lemma real-sqrt-sum-squares-less: $|x| < u / sqrt 2 \implies |y| < u / sqrt 2 \implies sqrt (x^2 + y^2) < u$
apply (rule power2-less-imp-less)
apply simp
apply (drule power-strict-mono [OF - abs-zero pos2])
apply (drule power-strict-mono [OF - abs-zero pos2])
apply (simp add: power-divide)
apply (drule order-le-less-trans [OF abs-ge-zero])
apply (simp add: zero-less-divide-iff)
done

lemma sqrt2-less-2: $sqrt 2 < (2::real)$
by (metis not-less not-less-iff-gr-or-eq numeral-less-iff real-sqrt-four
real-sqrt-le-iff semiring-norm(75) semiring-norm(78) semiring-norm(85))

lemma sqrt-sum-squares-half-less:
$x < u/2 \implies y < u/2 \implies 0 \leq x \implies 0 \leq y \implies sqrt (x^2 + y^2) < u$
apply (rule real-sqrt-sum-squares-less)
apply (auto simp add: abs-if field-simps)
apply (rule le-less-trans [where y = x+2])
using less-eq-real-def sqrt2-less-2 apply force
apply assumption
apply (rule le-less-trans [where y = y\*2])
using less-eq-real-def sqrt2-less-2 mult-le-cancel-left
apply auto
done

lemma LIMSEQ-root: (\lambda n. root n n) \longrightarrow 1
proof –
define x where x n = root n n - 1 for n
have x \longrightarrow sqrt 0
proof (rule tendsto-sandwich[OF - - tendsto-const])
  show (\lambda x. sqrt (2 / x)) \longrightarrow sqrt 0
    by (intro tendsto-intros tendsto-divide-0[OF tendsto-const]
        filterlim-mono[OF filterlim-real-sequentially])
Mit also have ... \leq (\sum k \in {0, 2}. of-nat (n choose k) + x n \cdot k)
  by (simp add: x-def)
also have ... \leq (\sum k \leq n. of-nat (n choose k) \cdot x n \cdot k)
  using (2 < n)
  by (intro sum-mono2) (auto intro!: mult-nonneg-nonneg zero-le-power simp: x-def le-diff-eq)
also have ... = (x n + 1) ^ n
  by (simp add: binomial-ring)
also have ... = n
  using (2 < n) by (simp add: x-def)
finally have real (n - 1) * (real n / 2 * (x n)^2) \leq real (n - 1) * 1
  by simp
then have (x n)^2 \leq 2 / real n
  using (2 < n) unfolding mult-le-cancel-left by (simp add: field-simps)
from real-sqrt-le-mono[OF this] show ?thesis
  by simp
qed

then show eventually (\lambda n. x n \leq sqrt (2 / real n)) sequentially
  by (auto intro!: extI[of - 3] simp: eventually-sequentially)
show eventually (\lambda n. sqrt 0 \leq x n) sequentially
  by (auto intro!: extI[of - 1] simp: eventually-sequentially le-diff-eq x-def)
qed

from tendsto-add[OF this tendsto-const[of 1]] show ?thesis
  by (simp add: x-def)
qed

lemma LIMSEQ-root-const:
  assumes 0 < c
  shows (\lambda n. root n n c) \longrightarrow 1
proof –
have ge-1: (\lambda n. root n n c) \longrightarrow 1 if I \leq c for c :: real
proof

define \( x \) where \( x \cdot n = \sqrt[n]{c} - 1 \) for \( n \)

have \( x \longrightarrow 0 \)

proof

show \( (\lambda n. \frac{c}{n}) \longrightarrow 0 \)

by (intro tendsto-divide-0[OF tendsto-const] filterlim-mono[OF filterlim-real-sequentially])

(simp-all add: at-infinity-eq-at-top-bot)

have \( x \cdot n \leq c / n \) if \( 1 < n \) for \( n :: \text{nat} \)

proof

have \( 1 + x \cdot n \cdot n = 1 + \text{of-nat} (n \cdot \text{choose} 1) \cdot x \cdot n^{-1} \)

by (simp add: choose-one)

also have \( \ldots \leq (\sum k \in \{0, 1\}. \text{of-nat} (n \cdot \text{choose} k) \cdot x \cdot n^{-k}) \)

by (simp add: x-def)

also have \( \ldots \leq (\sum k \leq n. \text{of-nat} (n \cdot \text{choose} k) \cdot x \cdot n^{-k}) \)

using \( (1 < n) \cdot (1 \leq c) \)

by (intro sum-mono2)

(auto intro!: mult-nonneg-nonneg zero-le-power simp: x-def diff-eq)

also have \( \ldots = (x \cdot n + 1) \cdot n \)

by (simp add: binomial-ring)

also have \( \ldots = c \)

using \( (1 < n) \cdot (1 \leq c) \cdot (1 < c) \) by (simp add: field-simps)

finally show \( \text{thesis} \)

using \( (1 \leq c) \cdot (1 < n) \) by (simp add: field-simps)

qed

then show eventually \( (\lambda n. x \cdot n \leq c / n) \) sequentially

by (auto intro!: exI[of - 3] simp: eventually-sequentially)

show eventually \( (\lambda n. 0 \leq x \cdot n) \) sequentially

using \( (1 \leq c) \)

by (auto intro!: exI[of - 1] simp: eventually-sequentially le-diff-eq x-def)

qed

from tendsto-add[OF this tendsto-const[of 1]] show \( \text{thesis} \)

by (simp add: x-def)

qed

show \( \text{thesis} \)

proof (cases \( 1 \leq c \))

case True

with ge-1 show \( \text{thesis} \) by blast

next
case False

with \( \langle 0 < c \rangle \) have \( 1 \leq 1 / c \)

by simp

then have \( (\lambda n. 1 / \sqrt[n]{c}) \longrightarrow 1 / 1 \)

by (intro tendsto-divide tendsto-const ge-1 \( 1 \leq 1 / c \) one-neg-zero)

then show \( \text{thesis} \)

by (rule filterlim-cong[THEN iffD1, rotated 3])

(auto intro!: exI[of - 1] simp: eventually-sequentially real-root-divide)

qed

qed

Legacy theorem names:
THEORY "Transcendental"

lemmas real-root-pos2 = real-root-power-cancel
lemmas real-root-pos-pos = real-root-gt-zero [THEN order-less-imp-le]
lemmas real-root-pos-pos-le = real-root-ge-zero
lemmas real-sqrt-eq-zero-cancel-iff = real-sqrt-eq-0-iff
end

110 Power Series, Transcendental Functions etc.

theory Transcendental
imports Series Deriv NthRoot
begin

A theorem about the factorial function on the reals.

lemma square-fact-le-2-fact: fact n * fact n ≤ (fact (2 * n) :: real)
proof (induct n)
case 0
then show ?case by simp
next
case (Suc n)
have (fact (Suc n)) * (fact (Suc n)) = of-nat (Suc n) * of-nat (Suc n) * (fact n * fact n :: real)
by (simp add: field-simps)
also have ... ≤ of-nat (Suc n) * of-nat (Suc n) * fact (2 * n)
by (rule mult-left-mono [OF Suc]) simp
also have ... ≤ of-nat (Suc (Suc (2 * n))) * of-nat (Suc (2 * n)) * fact (2 * n)
by (rule mult-right-mono)+ (auto simp: field-simps)
also have ... = fact (2 * Suc n) by (simp add: field-simps)
finally show ?case .
qed

lemma fact-in-Reals: fact n ∈ ℝ
by (induction n) auto

lemma of-real-fact [simp]: of-real (fact n) = fact n
by (metis of-nat-fact of-real-of-nat-eq)

lemma pochhammer-of-real: pochhammer (of-real x) n = of-real (pochhammer x n)
by (simp add: pochhammer-prod)

lemma norm-fact [simp]: norm (fact n :: 'a::real_normed_algebra_1) = fact n
proof
have (fact n :: 'a) = of-real (fact n)
by simp
also have norm ... = fact n
by (subst norm-of-real) simp
finally show ?thesis .
qed

lemma root-test-convergence:
fixes f :: nat ⇒ 'a::banach
assumes f: (λn. root n (norm (f n))) −→ x — could be weakened to lim sup and x < 1
shows summable f
proof −
have 0 ≤ x
  by (rule LIMSEQ-le[OF tendsto-const f]) (auto intro!: exI[of - 1])
from (x < 1) obtain z where z: x < z z < 1
  by (metis dense)
from f (x < z) have eventually (λn. root n (norm (f n)) < z) sequentially
  by (rule order-tendstoD)
then have eventually (λn. norm (f n) ≤ zˆn) sequentially
  using eventually-ge-at-top
proof eventually-elim
  fix n
  assume less: root n (norm (f n)) < z and n: 1 ≤ n
  from power-strict-mono[OF less, of n] n show norm (f n) ≤ z ˆ n
    by simp
qed
then show summable f
  unfolding eventually-sequentially
  using z ⟨0 ≤ x⟩ by (auto intro!: summable-comparison-test[OF - summable-geometric])
qed

110.1 More facts about binomial coefficients

These facts could have been proven before, but having real numbers makes the proofs a lot easier.

lemma central-binomial-odd:
odd n ⇒ n choose (Suc (n div 2)) = n choose (n div 2)
proof −
assume odd n
hence Suc (n div 2) ≤ n by presburger
hence n choose (Suc (n div 2)) = n choose (n - Suc (n div 2))
  by (rule binomial-symmetric)
also from ⟨odd n⟩ have n - Suc (n div 2) = n div 2 by presburger
finally show ?thesis .
qed

lemma binomial-less-binomial-Suc:
assumes k: k < n div 2
shows n choose k < n choose (Suc k)
proof −
from k have k': k ≤ n Suc k ≤ n by simp-all
from k' have real (n choose k) = fact n / (fact k * fact (n - k))
  by (simp add: binomial-fact)
also from $k'$ have $n - k = \text{Suc} \ (n - \text{Suc} \ k)$ by simp
also from $k'$ have fact ... = $(\text{real} \ n - \text{real} \ k) \ast \text{fact} \ (n - \text{Suc} \ k)$
    by (subst fact-Suc) (simp-all add: af-nat-diff)
also from $k$ have fact $k = \text{fact} \ (\text{Suc} \ k) / (\text{real} \ k + 1)$ by (simp add: field-simps)
also have fact $n / (\text{fact} \ (\text{Suc} \ k) / (\text{real} \ k + 1) \ast ((\text{real} \ n - \text{real} \ k) \ast \text{fact} \ (n - \text{Suc} \ k))) =$
    $(n \choose \text{Suc} \ k) \ast ((\text{real} \ k + 1) / (\text{real} \ n - \text{real} \ k))$
    using $k$ by (simp add: field-split-simps binomial-fact)
also from assms have $(\text{real} \ k + 1) / (\text{real} \ n - \text{real} \ k) < 1$ by simp
finally show $?thesis$ using $k$ by (simp add: mult-less-cancel-left)
qed

lemma binomial-strict-mono:
assumes $k < k' \ 2 \ast k' \leq n$
shows $n \choose k < n \choose k'$
proof -
  from assms have $k \leq k' - 1$ by simp
  thus $?thesis$
proof (induction rule: inc-induct)
case base
  with assms binomial-less-binomial-Suc[of $k' - 1 \ n$]
  show $?case$ by simp
next
case (step $k$)
  from step.prems step.hyps assms have $n \choose k < n \choose (\text{Suc} \ k)$
    by (intro binomial-less-binomial-Suc) simp-all
  also have $\ldots < n \choose k'$ by (rule step.IH)
  finally show $?case$.
qed

defined

lemma binomial-mono:
assumes $k \leq k' \ 2 \ast k' \leq n$
shows $n \choose k \leq n \choose k'$
proof (cases $k = k'$) simp-all

lemma binomial-strict-antimono:
assumes $k < k' \ 2 \ast k \geq n \ k' \leq n$
shows $n \choose k > n \choose k'$
proof -
  from assms have $(n - k) > n \choose (n - k')$
    by (intro binomial-strict-mono) (simp-all add: algebra-simps)
  with assms show $?thesis$ by (simp add: binomial-symmetric [symmetric])
qed

lemma binomial-antimono:
assumes $k \leq k' \ k \geq n \div 2 \ k' \leq n$
shows $n \choose k \geq n \choose k'$
proof (cases $k = k'$)
case False
note not-eq = False
show ?thesis
proof (cases k = n div 2 ∧ odd n)
  case False
  with assms(2) have 2 * k ≥ n by presburger
  with not-eq assms binomial-strict-antimono[of k k' n]
  show ?thesis by simp
next
  case True
  have n choose k' ≤ n choose (Suc (n div 2))
  proof (cases k' = Suc (n div 2))
    case False
    with assms True not-eq have Suc (n div 2) < k' by simp
    with assms binomial-strict-antimono[of Suc (n div 2) k' n] True
    show ?thesis by auto
  qed simp-all
  also from True have ... = n choose k by (simp add: central-binomial-odd)
  finally show ?thesis .
qed

lemma binomial-maximum: n choose k ≤ n choose (n div 2)
proof -
  have k ≤ n div 2 ⟷ 2 * k ≤ n by linarith
  consider 2 * k ≤ n | 2 * k ≥ n | k ≤ n | k > n by linarith
  thus ?thesis
  proof cases
    case 1
    thus ?thesis by (intro binomial-mono) linarith+
  next
    case 2
    thus ?thesis by (intro binomial-antimono) simp-all
  qed (simp-all add: binomial-eq-0)
qed

lemma binomial-maximum': (2 * n) choose k ≤ (2 * n) choose n
using binomial-maximum[of 2 * n] by simp

lemma central-binomial-lower-bound:
assumes n > 0
shows 4^n / (2 * real n) ≤ real ((2 * n) choose n)
proof -
  from binomial[of 1 1 2 * n]
  have 4^n = (∑ k≤2*n. (2*n) choose k)
  by (simp add: power-mult power2-eq-square One-nat-def [symmetric] del: One-nat-def)
  also have {..2*n} = {0..<2*n} ∪ {0,2*n} by auto
  also have (∑ k∈... (2*n) choose k) =
           (∑ k∈{0..<2*n}. (2*n) choose k) + (∑ k∈{0,2*n}. (2*n) choose k)
by (subst sum.union-disjoint) auto
also have \((\sum k \in \{0,2\cdot n\}. (2\cdot n) \text{ choose } k) \leq (\sum k \leq 1. (n \text{ choose } k)^2)\)
  by (cases n) simp-all
also from assms have \(\ldots \leq (\sum k \leq n. (n \text{ choose } k)^2)\)
  by (intro sum-mono2) auto
also have \(\ldots = (2\cdot n) \text{ choose } n\) by (rule choose-square-sum)
also have \((\sum k \in \{0\ldots,2\cdot n\}. (2\cdot n) \text{ choose } k) \leq (\sum k \in \{0\ldots,2\cdot n\}. (2\cdot n) \text{ choose } n)\)
  by (intro sum-mono binomial-maximum')
also have \(\ldots = \text{ card } \{0\ldots,2\cdot n\} \cdot ((2\cdot n) \text{ choose } n)\) by simp
also have \(\text{ card } \{0\ldots,2\cdot n\} \leq 2\cdot n - 1\) by (cases n) simp-all
also have \((2 \cdot n - 1) \cdot (2 \cdot n \text{ choose } n) + (2 \cdot n \text{ choose } n) = ((2\cdot n) \text{ choose } n) \cdot (2\cdot n)\)
  using assms by (simp add: algebra-simps)
finally have \(4 \cdot n \leq (2 \cdot n \text{ choose } n) \cdot (2 \cdot n)\) by simp-all
hence \(\text{ real } (4 \cdot n) \leq \text{ real } ((2 \cdot n \text{ choose } n) \cdot (2 \cdot n))\)
  by (subst of-nat-le-iff)
with assms show \(?\text{thesis}\) by (simp add: field-simps)
qed

110.2 Properties of Power Series

lemma powser-zero [simp]: \((\sum n. f n \cdot 0 \cdot n) = f 0\)
  for \(f :: \text{nat} \Rightarrow 'a::real-normed-algebra-1\)
proof –
  have \((\sum n < 1. f n \cdot 0 \cdot n) = (\sum n. f n \cdot 0 \cdot n)\)
    by (subst suminf-finite[where \(N = \{0\}\)] (auto simp: power-0-left))
  then show \(?\text{thesis}\) by simp
qed

lemma powser-sums-zero: \((\lambda n. a n \cdot 0 \cdot n) \text{ sums } a 0\)
  for \(a :: \text{nat} \Rightarrow 'a::real-normed-div-algebra\)
  using sums-finite [of \(\{0\}\) \(\lambda n. a n \cdot 0 \cdot n\)]
  by simp

lemma powser-sums-zero-iff [simp]: \((\lambda n. a n \cdot 0 \cdot n) \text{ sums } x \leftrightarrow a 0 = x\)
  for \(a :: \text{nat} \Rightarrow 'a::real-normed-div-algebra\)
  using powser-sums-zero sums-unique2 by blast

Power series has a circle or radius of convergence: if it sums for \(x\), then it sums absolutely for \(z\) with \(|z| < |x|\).

lemma powser-insidea:
  fixes \(x\) \(z:: 'a::real-normed-div-algebra\)
  assumes 1: \(\text{summable } (\lambda n. f n \cdot x \cdot n)\)
    and 2: \(\text{norm } z < \text{ norm } x\)
  shows \(\text{summable } (\lambda n. \text{norm } (f n \cdot z \cdot n))\)
proof –
  from 2 have \(x \cdot \text{-neg-0}: x \neq 0\) by clarsimp
  from 1 have \((\lambda n. f n \cdot x \cdot n) \rightarrow 0\)
by (rule summable-LIMSEQ-zero)
then have convergent (λn. f n * x^n)
  by (rule convergentI)
then have Cauchy (λn. f n * x^n)
  by (rule convergent-Cauchy)
then have Bseq (λn. f n * x^n)
  by (rule Cauchy-Bseq)
then obtain K where 3: 0 < K and 4: ∀n. norm (f n * x^n) ≤ K
  by (auto simp: Bseq_def)
have ∃N. ∀n≥N. norm (norm (f n * z^n)) ≤ K * norm (z^n) * inverse (norm (x^n))
proof (intro exI allI impI)
  fix n :: nat
  assume 0 ≤ n
  have norm (norm (f n * z^n)) * norm (x^n) =
    norm (f n * x^n) * norm (z^n)
    by (simp add: norm-mult abs-mult)
  also have ... ≤ K * norm (z^n)
    by (simp only: mult-right-mono 4 norm-ge-zero)
  also have ... = K * norm (z^n) * (inverse (norm (x^n)) * norm (z^n))
    by (simp add: x-neq-0)
  also have ... = K * norm (z^n) * inverse (norm (x^n)) * norm (x^n)
    by (simp only: mult_assoc)
  finally show norm (norm (f n * z^n)) ≤ K * norm (z^n) * inverse (norm (x^n))
    by (simp add: mult-le-cancel-right x-neq-0)
qed
moreover have summable (λn. K * norm (z^n) * inverse (norm (x^n)))
proof –
  from 2 have norm (norm (z * inverse x)) < 1
    using x-neq-0
    by (simp add: norm-mult nonzero-norm-inverse divide-inverse [where 'a=real, symmetric])
then have summable (λn. norm (z * inverse x) * n)
  by (rule summable-geometric)
then have summable (λn. K * norm (z * inverse x) * n)
  by (rule summable-mult)
then show summable (λn. K * norm (z * inverse x) * inverse (norm (x^n)))
  using x-neq-0
  by (simp add: norm-mult nonzero-norm-inverse power-mul-distrib
       power-inverse norm-power mult_assoc)
qed
ultimately show summable (λn. norm (f n * z^n))
  by (rule summable-comparison-test)
qed

lemma powser-inside:
  fixes f :: nat ⇒ 'a::{real-normed-div-algebra,banach}
  shows
THEORY "Transcendental"

summable \((\lambda n. f n \cdot (x^n))\) \implies \text{norm} z < \text{norm} x \implies
summable \((\lambda n. f n \cdot (z^n))\)
by (rule powser-insidea [THEN summable-norm-cancel])

lemma powser-times-n-limit-0:
fixes \(x :: 'a::\{real-normed-div-algebra,banach\}\)
assumes \(\text{norm} x < 1\)
shows \((\lambda n. \text{of-nat} n \cdot x^n) \longrightarrow 0\)
proof
  have \(\text{norm} x / (1 - \text{norm} x) \geq 0\)
  using assms by (auto simp: field-split-simps)
  moreover obtain \(N\) where \(N: \text{norm} x / (1 - \text{norm} x) < \text{of-int} N\)
  using ex-le-of-int by (meson ex-less-of-int)
  ultimately have \(*\): \(\text{real-of-int} (N + 1) \cdot \text{norm} x / \text{real-of-int} N < 1\)
  by auto
  then have \(*\): \(\text{real-of-int} (N + 1) \cdot \text{norm} x / \text{real-of-int} N < 1\)
  using assms by (auto simp: field-simps)
  have \(**\): \(\text{real-of-int} N \cdot \text{norm} x \cdot \text{real-of-nat} (\text{Suc} n) \cdot \text{norm} (x^n)\) \leq \(\text{real-of-int} n \cdot (\text{norm} x \cdot ((1 + N) \cdot \text{norm} (x^n)))\) if \(N \leq \text{int} n\) for \(n :: \text{nat}\)
  proof
    from that have \(\text{real-of-int} N \cdot \text{real-of-nat} (\text{Suc} n) \leq \text{real-of-int} n \cdot \text{real-of-int} (1 + N)\)
    by (simp add: algebra-simps)
    then have \(*\): \(\text{real-of-int} (N + 1) \cdot \text{norm} x / \text{real-of-int} N < 1\)
    using \(N\) by (auto simp: field-simps)
    have \(**\): \(\text{real-of-int} N \cdot \text{norm} x \cdot \text{real-of-nat} (\text{Suc} n) \cdot \text{norm} (x^n)\) \leq \(\text{real-of-int} n \cdot (1 + N) \cdot \text{norm} x \cdot \text{norm} (x^n)\)
    using \(N0\) mult-mono by fastforce
    then show \(?thesis\)
    by (simp add: algebra-simps)
  qed
  show \(?thesis\) using *
  by (rule summable-LIMSEQ-zero [OF summable-ratio-test, where \(N1=\text{nat} N\)])
  (simp add: \(N0\) norm-mult field-simps ** del: of-nat-Suc of-int-add)
qed

corollary lim-n-over-pown:
fixes \(x :: 'a::\{real-normed-field,banach\}\)
shows \(1 < \text{norm} x \implies ((\lambda n. \text{of-nat} n / x^n) \longrightarrow 0)\) sequentially
using powser-times-n-limit-0 [of \(\text{inverse} x\)]
by (simp add: norm-divide field-split-simps)

lemma sum-split-even-odd:
fixes \(f :: \text{nat} \Rightarrow \text{real}\)
shows \((\sum i < 2 \cdot n. \text{if even } i \text{ then } f i \text{ else } g i) = (\sum i < \text{n}. f (2 \cdot i)) + (\sum i < \text{n}. g (2 \cdot i + 1))\)
proof (induct \(n\))
  case 0
  then show \(?case\) by simp
next
case (Suc n)
  have \((\sum_{i<2 \cdot Suc n} \text{if even } i \text{ then } f i \text{ else } g i) = (\sum_{i<n} f (2 \cdot i)) + (\sum_{i<n} g (2 \cdot i + 1)) + (f (2 \cdot n) + g (2 \cdot n + 1))\)
    using Suc.hyps unfolding One-nat-def by auto
  also have \(\ldots = (\sum_{i<Suc n} f (2 \cdot i)) + (\sum_{i<Suc n} g (2 \cdot i + 1))\)
    by auto
finally show \(\text{?case} .\)
Qed

lemma sums-if':
  fixes g :: nat \Rightarrow real
  assumes g sums x
  shows \((\lambda n. \text{if even } n \text{ then } 0 \text{ else } g ((n - 1) \div 2)) \text{ sums } x\)
  unfolding sums-def
proof (rule LIMSEQ-I)
  fix r :: real
  assume \(0 < r\)
  from \((g \text{ sums } x) [\text{unfolded sums-def}, \text{THEN } \text{LIMSEQ-D, OF this}]\)
  obtain no where no-eq: \(\forall n. n \geq no \implies (\text{norm } (\sum g \{..<n\} - x) < r)\)
    by blast
  let \(?SUM = \lambda m. \sum_{i<m} \text{if even } i \text{ then } 0 \text{ else } g ((i - 1) \div 2)\)
  have \((\text{norm } (?SUM m - x) < r) \text{ if } m \geq 2 \cdot no \text{ for } m\)
  proof
    from that have \(m \div 2 \geq no\) by auto
    have sam-eq: \(?SUM (2 \cdot (m \div 2)) = \sum g \{..<m \div 2\}\)
      using sum-split-even-odd by auto
    then have \((\text{norm } (?SUM (2 \cdot (m \div 2)) - x) < r)\)
      using no-eq unfolding sum-eq using \(\langle m \div 2 \geq no \rangle\) by auto
  moreover
  have \(?SUM (2 \cdot (m \div 2)) = ?SUM m\)
  proof (cases even m)
    case True
    then show \(\text{?thesis}\)
      by (auto simp: even-two-times-div-two)
  next
    case False
    then have eq: \(Suc (2 \cdot (m \div 2)) = m\) by simp
    then have even \(\langle 2 \cdot (m \div 2) \rangle\) using odd m by auto
    have \(?SUM m = \langle ?SUM (Suc (2 \cdot (m \div 2))) \rangle\) unfolding eq ..
    also have \(\ldots = ?SUM (2 \cdot (m \div 2)) \text{ using } \langle \text{even } (2 \cdot (m \div 2)) \rangle\) by auto
    finally show \(\text{?thesis}\) by auto
  qed
  ultimately show \(\text{?thesis}\) by auto
  qed
  then show \(\exists no. \forall m \geq no. \text{norm } (?SUM m - x) < r\)
    by blast
  qed
lemma sums-if:
fixes $g : \mathbb{N} \rightarrow \mathbb{R}$
assumes $g \sum x$ and $f \sum y$
shows $(\lambda n. \text{if even } n \text{ then } f \left(\frac{n}{2}\right) \text{ else } g \left(\frac{n-1}{2}\right)) \sum (x + y)$
proof
  let $?s = (\lambda n. \text{if even } n \text{ then } 0 \text{ else } f \left(\frac{n-1}{2}\right))$
  have if-sum: $(\lambda n. \text{if } B \text{ then } 0 :: \mathbb{R} \text{ else } E) + (\lambda n. \text{if } \neg B \text{ then } T \text{ else } 0) = (\lambda n. \text{if } B \text{ then } E \text{ else } T)$
    using sums-def, THEN LIMSEQ-Suc
    show $f \sum x$ using sums-if $\langle g \sum x \rangle$.
  have if-eq: $(\lambda n. \text{if } \neg B \text{ then } T \text{ else } E) = (\lambda n. \text{if } B \text{ then } E \text{ else } T)$
    using sums-def cong del: if-weak-cong
    have $?s \sum y$ using sums-if $\langle f \sum y \rangle$.
    from this unfolded sums-def, THEN LIMSEQ-Suc
    have $(\lambda n. \text{if even } n \text{ then } f \left(\frac{n}{2}\right) \text{ else } 0) \sum y$
      by simp add: lessThan-Suc-eq-insert-0 sum atLeast1-atMost-eq image-Suc-lessThan if-eq sums-def cong del: if-weak-cong
    from sums-add $\langle g \sum y \rangle$ show $?\text{thesis}$
    by simp only: if-sum
qed

110.3 Alternating series test / Leibniz formula

lemma sums-alternating-upper-lower:
fixes a :: \mathbb{N} \Rightarrow \mathbb{R}
assumes mono: $\forall n. a \left(Suc \ n\right) \leq a \ n$
  and a-pos: $\forall n. 0 \leq a \ n$
  and a $\longrightarrow 0$
shows $\exists l. ((\forall n. \sum i < 2 \ast n. (-1) \ast i \ast a \ i) \leq l) \land (\lambda n. \sum i < 2 \ast n. (-1) \ast i \ast a \ i) \longrightarrow l) \land (\forall n. l \leq (\sum i < 2 \ast n + 1. (-1) \ast i \ast a \ i) \land (\lambda n. \sum i < 2 \ast n + 1. (-1) \ast i \ast a \ i) \longrightarrow l)$
(is $\exists l. ((\forall n. \text{if } n \leq l \text{ then } \text{if } \neg g \ n \text{ then } a \ (2 \ast n) \text{ else } a \ (2 \ast n) \text{ by auto}$
proof (rule nested-sequence-unique)
  have fg-diff: $\forall n. \text{if } n \leq \text{if } (Suc \ n)$
proof
  show $\text{if } n \leq \text{if } (Suc \ n)$ for $n$
    using mono[of $2 \ast n$] by auto
qed
show $\forall n. \text{if } n \leq \text{if } (Suc \ n)$
proof
  show $\text{if } n \leq \text{if } (Suc \ n)$ for $n$
    using mono[of $\text{Suc } (2 \ast n)$] by auto
qed
show $\forall n. \text{if } n \leq \text{if } g \ n$

proof
  show \( \forall n \leq ?g n \) for \( n \)
  using fg-diff a-pos by auto
qed

show \((\lambda n. \ ?f n - \ ?g n) \rightarrow \theta\)
unfolding fg-diff
proof (rule LIMSEQ-I)
  fix \( r : \mathbb{R} \)
  assume \( 0 < r \)
  with \( (a \rightarrow 0) \) [THEN LIMSEQ-D] obtain \( N \) where \( \forall n \geq N \Rightarrow \)norm \((a n - 0)\) < \( r \)
  by auto
  then have \( \forall n \geq N. \) norm \((- a \cdot (2 * n) - 0)\) < \( r \)
  by auto
  qed
qed

lemma summable-Leibniz':
  fixes \( a : \mathbb{N} \Rightarrow \mathbb{R} \)
  assumes a-zero: \( a \rightarrow 0 \)
  and a-pos: \( \forall n. \ 0 \leq a n \)
  and a-monotone: \( \forall n. \ a (Suc n) \leq a n \)
  shows summable: \( \forall n. \ (\sum_{i=0}^{2n} (-1)^i a i) \leq (\sum_{i=0}^{2n+1} (-1)^i a i) \)
  and \( \forall n. \ (\sum_{i=0}^{2n} (-1)^i a i) \rightarrow (\sum_{i=0}^{n} (-1)^i a i) \)
  and \( \forall n. \ (\sum_{i=0}^{2n+1} (-1)^i a i) \rightarrow (\sum_{i=0}^{n+1} (-1)^i a i) \)
proof
  let \( ?S = \lambda n. (-1)^n a n \)
  let \( ?P = \lambda n. \sum_{i=0}^{n} ?S i \)
  let \( ?f = \lambda n. \ ?P (2 * n) \)
  let \( ?g = \lambda n. \ ?P (2 * n + 1) \)
  obtain \( l : \mathbb{R} \)
  where below-l: \( \forall n. \ ?f n \leq l \)
  and \( ?f \rightarrow l \)
  and above-l: \( \forall n. \ l \leq ?g n \)
  and \( ?g \rightarrow l \)
  using sums-alternating-upper-lower [OF a-monotone a-pos a-zero] by blast

  let \( ?Sa = \lambda m. \sum_{n=m}^{l} ?S n \)
  have \( ?Sa \rightarrow l \)
proof (rule LIMSEQ-I)
  fix \( r : \mathbb{R} \)
  assume \( 0 < r \)
  with \( (\forall f \rightarrow l) \) [THEN LIMSEQ-D]
  obtain f-no where \( f: \forall n \geq f-no \Rightarrow \)norm \((?f n - l)\) < \( r \)
  by auto
from \(0 < r\) \(\langle ?g \longrightarrow l\rangle\)\ [THEN LIMSEQ-D]

obtain \(\text{g-no}\) where \(g: \forall n. n \geq \text{g-no} \implies \text{norm} \ (\text{?g n - l}) < r\)

by auto

have \(\text{norm} \ (\text{?Sa n - l}) < r\) if \(n \geq (\max (2 \ast \text{f-no}) (2 \ast \text{g-no}))\) for \(n\)

proof –

from that have \(n \geq 2 \ast \text{f-no}\) and \(n \geq 2 \ast \text{g-no}\) by auto

show \(?\text{thesis}\)

proof (cases even \(n\))

case True

then have \(\text{n-eq}: 2 \ast (n \div 2) = n\)

by (simp add: even-two-times-div-two)

with \(\langle n \geq 2 \ast \text{f-no}\rangle\) have \(n \div 2 \geq \text{f-no}\)

by auto

from \(f[\text{OF this}]\) show \(?\text{thesis}\)

unfolding \(n\)-eq \(\text{atLeastLessThanSuc-atLeastAtMost}\).

next

case False

then have \(\text{even} \ (n - 1)\) by simp

then have \(\text{n-eq}: 2 \ast ((n - 1) \div 2) = n - 1\)

by (simp add: even-two-times-div-two)

then have \(\text{range-eq}: n - 1 + 1 = n\)

using \text{odd-pos}[OF False] by auto

from \(\text{n-eq} \ (n \geq 2 \ast \text{g-no})\) have \(n \div 2 \geq \text{g-no}\)

by auto

from \(g[\text{OF this}]\) show \(?\text{thesis}\)

by (simp only: \(n\)-eq range-eq)

qed

qed

then show \(\exists \text{no}. \ \forall n \geq \text{no}. \ \text{norm} \ (\text{?Sa n - l}) < r\) by blast

qed

then have \(\text{sams-l}: (\lambda i. (-1)^i \ast a \ i) \ \text{sums} \ l\)

by (simp only: \text{sums-def})

then show \(\text{summable} \ ?S\)

by (auto simp: \text{summable-def})

have \(l = \text{suminf} \ ?S\) by (rule \text{sums-unique}[OF \text{sams-l}])

fix \(n\)

show \(\text{suminf} \ ?S \leq \ ?g \ n\)

unfolding \text{sums-unique}[OF \text{sams-l}, \text{symmetric}] using \text{above-l} by auto

show \(?f \ n \leq \text{suminf} \ ?S\)

unfolding \text{sums-unique}[OF \text{sams-l}, \text{symmetric}] using \text{below-l} by auto

show \(?g \longrightarrow \text{suminf} \ ?S\)

using \(?g \longrightarrow l\; l = \text{suminf} \ ?S\) by auto

show \(?f \longrightarrow \text{suminf} \ ?S\)

using \(?f \longrightarrow l\; l = \text{suminf} \ ?S\) by auto

qed

theorem \text{summable-Leibniz}:
THEORY "Transcendental"

proof

assumes

have

this

summable

|−?

summable

note

then have

(1)

with

let

and

by

also

and

case

have

by

note

Leibniz = summable-lemma [OF a as m] = sums

from

unfolding

ends

have

next

let

and

proof

shows

and

a x 0

assumes

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and

and
by auto

have \( ?\pos \) using \((0 \leq ?a 0)\) by auto
moreover have \(?\neg\)
  using leibniz(2,4)
  unfolding mult-minus-right sum-negf move-minus neg-le-iff-le
  by auto
moreover have \(?f\) and \(?g\)
  using leibniz(3,5)
  unfolding mult-minus-right sum-negf move-minus

  THEN tendsto-minus-cancel
  by auto
ultimately show \(?\thesis\)
  by auto
qed

then show \(?\summable\) and \(?\pos\) and \(?\neg\) and \(?f\) and \(?g\)
  by safe
qed

110.4 Term-by-Term Differentiability of Power Series

definition diffs :: \((\text{nat} \Rightarrow 'a::\text{ring-1}) \Rightarrow \text{nat} \Rightarrow 'a\)
  where diffs \(c\) = \((\lambda n. \text{of-nat} (\text{Suc} n) * c \text{ (Suc } n))\)

Lemma about distributing negation over it.

lemma diffs-minus: diffs (\(\lambda n. - c n\)) = (\(\lambda n. - \text{diffs } c \text{ } n\))
  by (simp add: diffs-def)

lemma diffs-equiv:
  fixes \(x::'a::\{\text{real-normed-vector,ring-1}\}\)
  shows summable (\(\lambda n. \text{diffs } c \text{ } n * x^{'n}\)) \(\Rightarrow\)
  (\(\lambda n. \text{of-nat } n * c n * x^{(n - \text{Suc } 0)}\)) \(\sum\)s \((\sum n. \text{diffs } c \text{ } n * x^{'n})\)
  unfolding diffs-def
  by (simp add: summable-sums sums-Suc-imp)

lemma lemma-termdiff1:
  fixes \(z::'a::\{\text{monoid-mult,comm-ring}\}\)
  shows (\(\sum p<\text{Suc } m. (((z + h) ^ (m - p)) * (z ^ p)) - (z ^ m)) =\)
  (\(\sum p<\text{Suc } m. (z ^ p) * (((z + h) ^ (m - p)) - (z ^ (m - p))))\))
  by (auto simp: algebra-simps power-add [symmetric])

lemma sumr-diff-mult-const2: sum f \(\{..<n\} - \text{of-nat } n * r = (\sum i<n. f i - r)\)
  for \(r::'a::\text{ring-1}\)
  by (simp add: sum-subtractf)

lemma lemma-realspow-rev-sumr:
  (\(\sum p<\text{Suc } n. (x ^ p) * (y ^ (n - p))\)) = (\(\sum p<\text{Suc } n. (x ^ (n - p)) * (y ^ p)\))
  by (subst \sum.\nat\-diff\-reindex[\text{symmetric}]| simp)

lemma lemma-termdiff2:
  fixes \(h::'a::\text{field}\)
assumes \( h \neq 0 \)

shows \(
\frac{(z + h)^n - z^n}{h - \text{of-nat } n \times z^{(n - \text{Suc } 0)}} = h \times \left( \sum_{p < n - \text{Suc } 0} p \times (z + h)^q \times z^{(n - 2 - q)} \right)
\)

(is lhs = rhs)

proof (cases n)

case (Suc n)

have \( \forall x. \left( \sum_{n < \text{Suc } k} h \times ((x - (k - n) \times (h + z)^n)) = \sum_{j < \text{Suc } k} (h + z)^j \cdot z^{(x + k - j)} \right) \)

by (simp add: power-add [symmetric] mult.commute)

have \( \left( \sum_{i < n} z^i \cdot ((z + h)^{(n - i) - z} \cdot (n - i)) \right) = \sum_{i < n} \sum_{j < n - i} h \times ((z + h)^j \cdot z^{(n - \text{Suc } j)}) \)

apply (rule sum.cong [OF refl])

apply (clarsimp simp add: less-iff-Suc-add sum-distrib-left diff-power-eq-sum ac-simps 0 simp del: sum.lessThan-Suc power-Suc)

done

have \( h \times \text{lhs} = h \times \text{rhs} \)

apply (simp add: right-diff-distrib diff-divide-distrib h mult.assoc [symmetric])

using Suc

apply (simp add: diff-power-eq-sum h right-diff-distrib [symmetric] mult.assoc del: power-Suc sum.lessThan-Suc of-nat-Suc)

apply (subst lemma-realpow-rev-sumr)

apply (subst sumr-diff-mult-const2)

apply (simp add: lemma-termdiff1 sum-distrib-left *)

done

then show \( \text{thesis} \)

by (simp add: h)

qed auto

lemma real-sum-nat-ivl-bounded2:

fixes \( K :: 'a::{linordered-semidom} \)

assumes \( f : \\forall p :: \text{nat}. \ p < n \implies f \ p \leq K \)

and \( K : 0 \leq K \)

shows \( \text{sum } f \{..n-k\} \leq \text{of-nat } n \times K \)

apply (rule order-trans [OF sum-mono [OF f]])

apply (auto simp: mult-right-mono K)

done

lemma lemma-termdiff3:

fixes \( h \ z :: 'a::{real-normed-field} \)

assumes \( I : h \neq 0 \)

and \( 2: \text{norm } z \leq K \)

and \( 3: \text{norm } (z + h) \leq K \)

shows \( \text{norm } ((z + h)^n - z^n) / h - \text{of-nat } n \times z^{(n - \text{Suc } 0)} \leq \text{of-nat } n \times \text{of-nat } (n - \text{Suc } 0) \times K^{(n - 2)} \times \text{norm } h \)

proof –
have \( \text{norm} \left( \frac{(z + h) \cdot n - z \cdot n}{h} - \text{of-nat} \ n \ast z \cdot (n - \text{Suc} \ 0) \right) \) =
\( \text{norm} \left( \sum \ p < n - \text{Suc} \ 0 \ . \ sum \ q < n - \text{Suc} \ 0 - \ p \ . \ (z + h) \cdot q \ast z \cdot (n - 2 - q) \right) \)
\* \text{norm} \ h
by (metis (lifting, no-types) lemma-termdiff2 \[ \text{OF} \ 1 \] \text{mult.commute} \ text{norm-mult})
also have \ldots \leq \text{of-nat} \ n \ast (\text{of-nat} \ (n - \text{Suc} \ 0) \ast K \cdot (n - 2)) \ast \text{norm} \ h
proof (rule \text{mult-right-mono} \ [\text{OF} - \text{norm-ge-zero}] )
from \text{norm-ge-zero} \ 2 \ have \ K: 0 \leq K
by (rule \text{order-trans})
have le-Kn: \( \forall i \ j \ n. \ i + j = n \imp \text{norm} \ ((z + h) \cdot i \ast z \cdot j) \leq K \cdot n \)
apply (erule subst)
apply (simp only: \text{norm-mult} \ \text{norm-power} \ \text{power-add})
apply (intro \text{mult-mono} \ \text{power-mono} \ 2 \ 3 \ \text{norm-ge-zero} \ \text{zero-le-power} \ K)
done
show \( \text{norm} \left( \sum \ p < n - \text{Suc} \ 0. \ sum \ q < n - \text{Suc} \ 0 - \ p. \ (z + h) \cdot q \ast z \cdot (n - 2 - q) \right) \)
\leq \( \text{of-nat} \ n \ast (\text{of-nat} \ (n - \text{Suc} \ 0) \ast K \cdot (n - 2)) \)
apply (intro \text{order-trans} \ [\text{OF} \ \text{norm-sum}]
\text{real-sum-nat-ivl-bounded2}
\text{mult-nonneg-nonneg}
\text{of-nat-0-le-iff}
\text{zero-le-power} \ K)
apply (rule le-Kn, simp)
done
qed
also have \ldots = \text{of-nat} \ n \ast \text{of-nat} \ (n - \text{Suc} \ 0) \ast K \cdot (n - 2) \ast \text{norm} \ h
by (simp only: \text{mult.assoc})
finally show \( \? \text{thesis} \).
qed

lemma lemma-termdiff4:
fixes \( f : 'a::\text{real-normed-vector} \imp 'b::\text{real-normed-vector} \)
and \( k :: \text{real} \)
assumes \( k : 0 < k \)
and le: \( \forall h. \ h \neq 0 \imp \text{norm} \ h < k \imp \text{norm} \ (f \ h) \leq K \ast \text{norm} \ h \)
shows \( f \rightarrow 0 \)
proof (rule \text{tendsto-norm-zero-cancel})
show (\( \lambda h. \ \text{norm} \ (f \ h) \)) \rightarrow 0
proof (rule \text{real-tendsto-cancel})
show eventually (\( \lambda h. \ 0 \leq \text{norm} \ (f \ h) \)) (at 0)
by simp
show eventually (\( \lambda h. \ \text{norm} \ (f \ h) \leq K \ast \text{norm} \ h \)) (at 0)
using \( k \) by (auto simp: eventually-at dist-norm le)
show (\( \lambda h. \ 0 \)) \rightarrow (\( 0::'a \))
by (rule \text{tendsto-const})
have (\( \lambda h. \ K \ast \text{norm} \ h \)) \rightarrow (\( 0::'a \)) \rightarrow K \ast \text{norm} \ (0::'a)
by (intro \text{tendsto-intros})
then show (\( \lambda h. \ K \ast \text{norm} \ h \)) \rightarrow (\( 0::'a \)) \rightarrow 0
by simp
qed

lemma lemma-termdiff5:
  fixes g :: 'a::{real-normed-vector} ⇒ nat ⇒ 'b::banach
  and k :: real
  assumes k: 0 < k
  and le: ∃h n. h ≠ 0 ⇒ norm h < k ⇒ norm (g h n) ≤ f n * norm h
  shows (λh. suminf (g h)) -0 → 0
proof (rule lemma-termdiff4 [OF k])
  fix h :: 'a
  assume h ≠ 0 and norm h < k
  then have 1: ∀ n. norm (g h n) ≤ f n * norm h
    by (simp add: le)
  then have ∃ N. ∀ n≥N. norm (g h n) ≤ f n * norm h
    by simp
  moreover from f have 2: summable (λn. f n * norm h)
    by (rule summable-mult2)
  ultimately have 3: summable (λn. norm (g h n))
    by (rule summable-comparison-test)
  then have norm (suminf (g h)) ≤ (∑ n. norm (g h n))
    by (rule summable-norm)
  also from 1 3 2 have (∑ n. norm (g h n)) ≤ (∑ n. f n * norm h)
    by (rule suminf-le)
  also from f have (∑ n. f n * norm h) = suminf f * norm h
    by (rule suminf-mult2 [symmetric])
  finally show norm (suminf (g h)) ≤ suminf f * norm h.
qed

lemma termdiffs-aux:
  fixes x :: 'a::{real-normed-field,banach}
  assumes 1: summable (λn. diffs (diffs c) n * K ^ n)
  and 2: norm x < norm K
  shows (λh. (∑ n. c n * (((x + h) ^ n - x ^ n) / h - of-nat n * x ^ (n - Suc 0)))) -0 → 0
proof –
  from dense [OF 2] obtain r where r1: norm x < r and r2: r < norm K
  by fast
  from norm-ge-zero r1 have r: 0 < r
    by (rule order-le-less-trans)
  then have r-neq-0: r ≠ 0 by simp
  show ?thesis
proof (rule lemma-termdiff5)
  show 0 < r - norm x
    using r1 by simp
from $r \ r 2$ have $\text{norm (of-real } r :: 'a) < \text{norm } K$
    by simp
with $I$ have summable $(\lambda n. \text{norm (diffs (diffs } c) n * (of-real } r ^ n))$
    by (rule powser-insidea)
them have summable $(\lambda n. \text{diffs (diffs (\lambda n. \text{norm } (c n))} n * r ^ n)$
    using $r$ by (simp add: diffs-def norm-mult norm-power del: of-nat-Suc)
them have summable $(\lambda n. \text{of-nat } n * \text{diffs (\lambda n. \text{norm } (c n))} n * r ^ (n - Suc 0))$
    by (rule diffs-equiv [THEN sums-summable])
also have $(\lambda n. \text{of-nat } n * \text{diffs (\lambda n. \text{norm } (c n))} n * r ^ (n - Suc 0) = $
    $(\lambda n. \text{diffs (\lambda n. \text{of-nat } (m - Suc 0) * \text{norm } (c m) * \text{inverse } r) n * (r ^ n)))$
apply (rule ext)
apply (case-tac $n$
apply (simp-all add: diffs-def r-neq-0)
done
finally have summable $(\lambda n. \text{of-nat } n * \text{of-nat } n * \text{of-nat } (n - Suc 0) * r ^ (n - 2))$
    by (rule diffs-equiv [THEN sums-summable])
also have $(\lambda n. \text{of-nat } n * \text{of-nat } n * \text{of-nat } (n - Suc 0) * r ^ (n - 2))$
    apply (rule ext)
    apply (case-tac $n$, simp)
    apply (rename-tac nat)
    apply (case-tac nat, simp)
    apply (simp add: r-neq-0)
done
finally show summable $(\lambda n. \text{norm } (c n) * \text{of-nat } n * \text{of-nat } (n - Suc 0) * r ^ (n - 2))$.
next
fix $h :: 'a$
fix $n :: \text{nat}$
assume $h: h \neq 0$
assume $\text{norm } h < r - \text{norm } x$
then have $\text{norm } x + \text{norm } h < r$ by simp
with $\text{norm-triangle-ineq}$ have $\text{xh: norm } (x + h) < r$
    by (rule order-le-less-trans)
show $\text{norm } (c n * ((x + h) ^ n - x ^ n) / h - \text{of-nat } n * x ^ (n - Suc 0))) \leq $
    $\text{norm } (c n) * \text{of-nat } n * \text{of-nat } (n - Suc 0) * r ^ (n - 2) * \text{norm } h$
apply (simp only: norm-mul mult_assoc)
apply (rule mult-left-mono [OF - norm_ge_zero])
apply (simp add: mult_assoc [symmetric])
apply (metis h lemma-termdiff3 less_eq_real_def r1 xh)
done
qed
qed
lemma termdiffs:
  fixes K x :: 'a::{real-normed-field,banach}
  assumes 1: summable (λn. c n * K ^ n)
    and 2: summable (λn. (diffs c) n * K ^ n)
    and 3: summable (λn. (diffs (diffs c)) n * K ^ n)
    and 4: norm x < norm K
  shows DERIV (λx. ∑ n. c n * x ^ n) x :> (∑ n. (diffs c) n * x ^ n)
proof (rule LIM-zero-cancel)
  show (λh. (suminf (λn. c n * (x + h) ^ n) - suminf (λn. c n * x ^ n)) / h
    - suminf (λn. diffs (λn. c n * x ^ n))) -0→ 0
  using 4 by (simp add: less-diff-eq)
next
fix h :: 'a
assume norm (h - 0) < norm K - norm x
then have norm x + norm h < norm K by simp
then have 5: norm (x + h) < norm K
  by (rule norm-triangle-ineq [THEN order-le-less-trans])
  have summable (λn. c n * x ^ n)
    and summable (λn. c n * (x + h) ^ n)
    and summable (λn. diffs c n * x ^ n)
  using 1 2 4 5 by (auto elim: powser-inside)
then have (((∑ n. c n * (x + h) ^ n) - (∑ n. c n * x ^ n)) / h - (∑ n. diffs c n * x ^ n)) =
  (((∑ n. c n * (x + h) ^ n - c n * x ^ n) / h - of-nat n * c n * x ^ (n - Suc 0)))
  by (intro sums-unique sums-diff sums-divide diffs-equiv summable-sums)
then show (((∑ n. c n * (x + h) ^ n) - (∑ n. c n * x ^ n)) / h - (∑ n. diffs c n * x ^ n)) =
  (((∑ n. c n * (((x + h) ^ n - x ^ n) / h - of-nat n * x ^ (n - Suc 0)))
  by (simp add: algebra-simps)
next
  show (λh. ∑ n. c n * (((x + h) ^ n - x ^ n) / h - of-nat n * x ^ (n - Suc 0))) -0→ 0
  by (rule termdiffs-aux [OF 3 4])
qed

110.5 The Derivative of a Power Series Has the Same Radius of Convergence

lemma termdiff-converges:
  fixes x :: 'a::{real-normed-field,banach}
  assumes K: norm x < K
    and sm: ∃x. norm x < K ⇒ summable(λn. c n * x ^ n)
  shows summable (λn. diffs c n * x ^ n)
proof (cases x = 0)
THEORY "Transcendental"

case True
then show ?thesis
using powser-sums-zero sums-summable by auto

next
case False
then have K > 0
using K less-trans zero-less-norm-iff by blast
then obtain r :: real where r: norm x < norm r norm r < K r > 0
using K False
by (auto simp: field-simps abs-less-iff add-pos-pos intro: that [of (norm x + K) / 2])

have to0: (λn. of-nat n * (x / of-real r) ^ n) ----> 0
using r by (simp add: norm-divide powser-times-n-limit-0 [of x / of-real r])

obtain N where N: ∃n. n ≥ N → real-of-real n * norm x ^ n < r ^ n
using r LIMSEQ-D [OF to0, of 1]
by (auto simp: norm-divide norm-mult norm-power field-simps)

have summable (λn. of-nat n * c n) * x ^ n)
proof (rule summable-comparison-test)
show summable (λn. norm (c n * of-real r ^ n))
apply (rule powser-insidea [OF sm [of of-real ((r+K)/2)]]
using N r norm-of-real [of r + K, where 'a = 'a] by auto
show ∃n. N ≤ n → norm (of-nat n * c n * x ^ n) ≤ norm (c n * of-real r ^ n)
using N r by (fastforce simp add: norm-mult norm-power less-eq-real-def)

qed

then have summable (λn. of-nat (Suc n) * c(Suc n)) * x ^ Suc n)
using summable-iff-shift [of λn. of-nat n * c n * x ^ n 1]
by simp

then have summable (λn. (of-nat (Suc n) * c(Suc n)) * x ^ n)
using False summable-mult2 [of λn. (of-nat (Suc n) * c(Suc n) * x ^ n) * x
inverse x]
by (simp add: mult.assoc) (auto simp: ac-simps)

then show ?thesis
by (simp add: diffs-def)

qed

lemma termdiff-converges-all:
fixes x :: 'a::{real-normed-field,banach}
assumes ∃x. summable (λn. c n * x^n)
shows summable (λn. diffs c n * x^n)
by (rule summable-converges [where K = 1 + norm x]) (use assms in auto)

lemma termdiffs-strong:
fixes K x :: 'a::{real-normed-field,banach}
assumes sm: summable (λn. c n * K ^ n)
and K: norm x < norm K
shows DERIV (λx. ∑n. c n * x ^ n) x := (∑n. diffs c n * x ^ n)
proof –
have K2: norm ((of-real (norm K) + of-real (norm x)) / 2 :: 'a) < norm K
using $K$
apply (auto simp: norm-divide field-simps)
apply (rule le-less-trans [of - (norm $x$) + of-real (norm $K$)])
apply (auto simp: mult-2-right norm-triangle-mono)
done
then have [simp]: norm ((of-real (norm $K$) + of-real (norm $x$)) :: 'a) < norm $K$ * 2
  by (simp)
have summable $(\lambda n. c n * (of-real (norm x + norm K) / 2) ^ n)$
  by (metis K2 summable-norm-cancel [OF powser-insidea [OF sm]] add.commute of-real-add)
moreover have $\forall x. norm x < norm K \Rightarrow$ summable $(\lambda n. diffs c n * x ^ n)$
  by (blast intro: sm termdiff-converges powser-inside)
moreover have $\forall x. norm x < norm K \Rightarrow$ summable $(\lambda n. diffs(diffs c n) * x ^ n)$
  by (blast intro: sm termdiff-converges powser-inside)
ultimately show ?thesis
apply (rule termdiffs [where $K = of-real (norm x + norm K) / 2$])
using $K$
apply (auto simp: field-simps)
done
qed

lemma termdiffs-strong-converges-wherever:
  fixes $K x :: 'a::{real-normed-field,banach}$
  assumes $\forall y. summable (\lambda n. c n * y ^ n)$
  shows $(\lambda x. sum n. c n * x ^ n)$ has-field-derivative $(\sum n. diffs c n * x ^ n)) (at x)$
using termdiffs-strong [OF assms [of of-real (norm $x$ + norm $K$)]]
by (force simp del: of-real-add)

lemma termdiffs-strong-':
  fixes $z :: 'a::{real-normed-field,banach}$
  assumes $\forall z. norm z < K \Rightarrow$ summable $(\lambda n. c n * z ^ n)$
  assumes norm $z < K$
  shows $(\lambda z. sum n. c n * z ^ n)$ has-field-derivative $(\sum n. diffs c n * z ^ n)) (at z)$
proof (rule termdiffs-strong)
define $L :: real$ where $L = (norm x + K) / 2$
have $0 \leq norm z$ by simp
also note $\langle norm z < K \rangle$
finally have $K \geq 0$ by simp
from assms $K$ have $L \geq 0$ norm $z < L \wedge K$ by (simp-all add: L-def)
from $L$ show norm $z < norm (of-real L :: 'a)$ by simp
from $L$ show summable $(\lambda n. c n * of-real L ^ n)$ by (intro assms(1)) simp-all
qed

lemma termdiffs-sums-strong:
  fixes $z :: 'a::{banach,real-normed-field}$
  assumes sums: $\forall z. norm z < K \Rightarrow$ $(\lambda n. c n * z ^ n)$ sums $f z$
assumes deriv: \((f \text{ has-field-derivative } f')\) (at \(z\))
assumes norm: \(\text{norm } z < K\)
shows \((\lambda n. \text{diffs } c \cdot n \cdot z \cdot n) \text{ sums } f'\)

proof –
  have summable: summable \((\lambda n. \text{diffs } c \cdot n \cdot z \cdot n)\)
    by (intro termdiffs-strong \([\text{OF norm}] \text{ sums-sammlable} [\text{OF sums}]\))
from norm have eventually \((\lambda z. z \in \text{norm} \rightarrow \ldots < K)\) (nhds \(z\))
  by (intro eventually-nhds-in-open open-vimage)
  (simp-all add: continuous-on-norm)
  hence eq: eventually \((\lambda z. (\sum n. c \cdot n \cdot z \cdot n) = f z)\) (nhds \(z\))
    by eventually-elim (insert sums, simp add: sums-iff)

  have \((\lambda z. \sum n. c \cdot n \cdot z \cdot n) \text{ has-field-derivative} (\sum n. \text{diffs } c \cdot n \cdot z \cdot n)\) (at \(z\))
    by (intro termdiffs-strong \([\text{OF norm}] \text{ sums-sammlable} [\text{OF sums}]\))
  hence \((f \text{ has-field-derivative} (\sum n. \text{diffs } c \cdot n \cdot z \cdot n))\) (at \(z\))
    by (subst (asm) DERIV-cong-ev [OF refl eq refl])
  from this and deriv have \((\sum n. \text{diffs } c \cdot n \cdot z \cdot n) = f'\)
    by (rule DERIV-unique)
  with summable show \(?\text{thesis}\) by (simp add: sums-iff)
  qed

lemma isCont-powser:
  fixes \(K \cdot x \cdot a\): \{real-normed-field, banach\}
  assumes summable: \((\lambda n. c \cdot n \cdot K \cdot n)\)
  assumes norm x < norm K
  shows isCont \((\lambda x. \sum n. c \cdot n \cdot x \cdot n)\) \(x\)
  using termdiffs-strong [OF assms] by (blast intro!: DERIV-isCont)

lemmas isCont-powser' = isCont-o2 [OF - isCont-powser]

lemma isCont-powser-converges-everywhere:
  fixes \(K \cdot x \cdot a\): \{real-normed-field, banach\}
  assumes \(y\): summable \((\lambda n. c \cdot n \cdot y \cdot n)\)
  shows isCont \((\lambda x. \sum n. c \cdot n \cdot x \cdot n)\) \(x\)
  using termdiffs-strong [OF assms [OF of-real [of norm \(x\) + 1]], of \(x\)]
    by (force intro!: DERIV-isCont simp del: of-real-add)

lemma powser-limit-0:
  fixes \(a\): \{real-normed-field, banach\}
  assumes \(s: 0 < s\)
  and sm: \(\forall x. \text{norm } x < s \rightarrow (\lambda n. a \cdot n \cdot x \cdot n) \text{ sums } (f x)\)
  shows \((f \longrightarrow a \cdot 0)\) (at \(0\))
  proof –
  have norm \((\text{of-real } s / 2: 'a) < s\)
    using \(s\) by (auto simp: norm-divide)
  then have summable \((\lambda n. a \cdot n \cdot (\text{of-real } s / 2) \cdot n)\)
    by (rule sums-sammlable [OF sm])
  then have \((\lambda x. \sum n. a \cdot n \cdot x \cdot n) \text{ has-field-derivative} (\sum n. \text{diffs } a \cdot n \cdot 0 \cdot n)\)
    (at \(0\))
    by (rule termdiffs-strong) (use \(s\) in (auto simp: norm-divide))
THEORY “Transcendental”

then have isCont \((\lambda x. \sum n. a_n * x^n) \cdot 0\)
  by (blast intro: DERIV-continuous)
then have \(((\lambda x. \sum n. a_n * x^n) \to a \cdot 0) \at 0\)
  by (simp add: continuous-within)
then show ?thesis
  apply (rule Lim-transform)
  apply (clarsimp simp: LIM-eq)
  apply (rule_tac x=s in exI)
  using s sm sums-unique by fastforce
qed

lemma powser-limit-0-strong:
  fixes a :: nat ⇒ './a::{real-normed-field,banach}
  assumes s: \(0 < s\)
  and sm: \(\forall x. x \neq 0 \Rightarrow \text{norm } x < s \Rightarrow (\lambda n. a_n * x^n)\) sums (f x)
  shows (f \to a \cdot 0) \at 0\)
proof
  have *: \((\lambda x. \text{if } x = 0 \text{ then } a_0 \text{ else } f x) \to a \cdot 0) \at 0\)
    by (rule powser-limit-0 [OF s]) (auto simp: powser-sums-zero sm)
  show ?thesis
    apply (subst LIM-equal [where g = \((\lambda x. \text{if } x = 0 \text{ then } a_0 \text{ else } f x)\)])
    apply simp
    done
qed

110.6 Derivability of power series

lemma DERIV-series':
  fixes f :: real ⇒ nat ⇒ real
  assumes DERIV-f: \(\forall n. \text{DERIV } (\lambda x. f x n) x0 : (f' x0 n)\)
  and allf-summable: \(\forall x. x \in {a <..< b} \Rightarrow \text{summable } (f x)\)
  and x0-in-I: \(x0 \in {a <..< b}\)
  and summable (f' x0)
  and summable L
  and L-def: \(\forall x y. x \in {a <..< b} \Rightarrow y \in {a <..< b} \Rightarrow |f x n - f y n| \leq L \cdot |x - y|\)
  shows \(\text{DERIV } (\lambda x. \text{suminf } (f x)) x0 : (\text{suminf } (f' x0))\)
unfolding DERIV-def
proof (rule LIM-I)
  fix r :: real
  assume 0 < r then have 0 < r/3 by auto
  obtain N-L where N-L: \(\forall n. N-L \leq n \Rightarrow |\sum i. L (i + n)| < r/3\)
    using suminf-exist-split[OF 0 < r/3] (summable L) by auto
  obtain N-f' where N-f': \(\forall n. N-f' \leq n \Rightarrow |\sum i. f' x0 (i + n)| < r/3\)
    using suminf-exist-split[OF 0 < r/3] (summable (f' x0)) by auto
  let ?N = Suc (max N-L N-f')
THEORY "Transcendental"

have \( \sum_i i \cdot f'(x_0) (i + \frac{1}{2}N) < r/3 \) (is \( \frac{1}{2}f'-\text{part} < r/3 \))
and \( L\)-estimate: \( \sum_i L(i + \frac{1}{2}N) < r/3 \)
using \( N-L[\text{of } \frac{1}{2}N] \) and \( N-f'[\text{of } \frac{1}{2}N] \) by auto

let \(?\text{diff} = \lambda x. (f(x_0 + x) - f(x_0)) / x\)

let \(?r = r / (3 \cdot \text{real } \frac{1}{2}N)\)
from \( (0 < r) \) have \( 0 < ?r \) by simp

let \(?s = \lambda n. \text{SOME } s. 0 < s \land (\forall x \cdot x \neq 0 \land |x| < s \rightarrow |?\text{diff} n x - f'(x_0)| < \frac{r}{3}\)\)
define \(S'\) where \(S' = \text{Min}(\{?s \cdot \{..<\frac{1}{2}N\}\})\)

have \( 0 < ?s n \)
unfolding \(S\)-def
proof (rule iffD2[\text{OF } \text{Min-gr-iff}])
show \( \forall x \in (\{?s \cdot \{..<\frac{1}{2}N\}\}), 0 < x \)
proof
fix \( x \)
assume \( x \in (\{?s \cdot \{..<\frac{1}{2}N\}\}) \)
then obtain \( n \) where \( x = ?s n \) and \( n \in \{..<\frac{1}{2}N\} \)
using image-iff[\text{THEN } \text{iffD1}] by blast
from \text{DERIV-D}[\text{OF } \text{DERIV-f}[\text{where } n=n], \text{THEN } \text{LIM-D, OF } (0 < ?r), \text{unfolding } \text{real-norm-def}]\]
obtain \( s \) where \( s\)-bound: \( 0 < s \land (\forall x \cdot x \neq 0 \land |x| < s \rightarrow |?\text{diff} n x - f'(x_0)| < \frac{r}{3}\)\)
\( n \mid < ?r)\)
by auto
have \( 0 < ?s n \)
by (rule someI2[\text{where } a=s]) (auto simp: \( s\)-bound simp del: of-nat-Suc)
then show \( 0 < x \) by (simp only: \( x = ?s n \))
qed
qed auto

define \(S\) where \( S = \text{min}(\text{min}(x_0 - a) (b - x_0)) S' \)
then have \( 0 < S \) and \( S-a: S \leq x_0 - a \) and \( S-b: S \leq b - x_0 \)
and \( S \leq S' \) using \( x_0\)-in-I and \( (0 < S') \)
by auto

have \( |(\sum_{i} f(x_0 + x) - \sum_{i} f(x_0)) / x - \sum_{i} f'(x_0)| < r \)
if \( x \neq 0 \) and \( |x| < S \) for \( x \)
proof -
from that have \( x\)-in-I: \( x_0 + x \in \{a <..< b\} \)
using \( S-a \) \( S-b \) by auto

\textbf{note} \( \text{diff-smbl} = \text{summable-diff}[\text{OF } \text{all-f-summable}[\text{OF } x_0\text{-in-I}]] \) \( \text{all-f-summable}[\text{OF } x_0\text{-in-I}]\)
\textbf{note} \( \text{div-smbl} = \text{summable-divide}[\text{OF } \text{diff-smbl}] \)
\textbf{note} \( \text{all-smbl} = \text{summable-diff}[\text{OF } \text{div-smbl } \text{summable } (f'(x_0))] \)
\textbf{note} \( \text{ign} = \text{summable-ignore-initial-segment}[\text{where } k=\frac{1}{2}N] \)
note diff-shift-smbl = summable-diff[OF ign[OF allf-summable[OF x-in-I]]
ign[OF allf-summable[OF x0-in-I]]]

note div-shift-smbl = summable-divide[OF diff-shift-smbl]

note all-shift-smbl = summable-diff[OF div-smbl ign[OF (summable (f' x0))]]

have 1: \(\|?\text{diff} (n + ?N) x)\| \leq L \(n + ?N\) for n

proof
- have \(\|?\text{diff} (n + ?N) x\| \leq L \(n + ?N\) * \(|x0 + x - x0| / |x|\)
  using divide-right-mono[OF L-def[OF x-in-I x0-in-I] abs-ge-zero]
  by (simp only: abs-divide)
  with \((x \neq 0)\) show \(?\text{thesis}\) by auto

qed

note div 2 = summable-rabs-comparison-test[OF - ign[OF (summable L)]]
from 1 have \(\sum i. \|?\text{diff} (i + ?N) x\| \leq \(\sum i. L (i + ?N)\)\)
by (metis (lifting) abs-idempotent)

order-trans[OF summable-rabs[OF 2] suminf-le[OF - 2 ign[OF (summable L)]]]
then have \(\sum i. \|?\text{diff} (i + ?N) x\| \leq r / 3\) (is ?L-part \(\leq r/3\))
using L-estimate by auto

have \(\|?\text{diff} n x - f' x0 n\| \leq \(\sum n<\?N. \|?\text{diff} n x - f' x0 n\|\)\)
also have \(\ldots < \(\sum n<\?N. \|?r\)\)
proof (rule sum-strict-mono)
fix n
assume \(n \in \{..<\?N\}\)
have \(|x| < S\) using \(|x| < S\).
also have \(S \leq S'\) using \(|S \leq S|\).
also have \(S' \leq \?s n\)
unfolding S'-def
proof (rule Min-le-iff[THEN iffD2])
have \(\?s n \in \{\?s \cdot \{..<\?N\}\} \wedge \?s n \leq \?s n\)
using \(n \in \{..<\?N\}\) by auto
then show \(\exists a \in \{\?s \cdot \{..<\?N\}\}, a \leq \?s n\)
by blast

qed auto
finally have \(|x| < \?s n\).

from DERIV-D[OF DERIV-f[where n=n], THEN LIM-D, OF \(\langle 0 < \?r,\)
unfolded real-norm-def diff-0-right, unfolded some-eq-ex[ symmetric], THEN
conjunct2]
have \(\forall x. x \neq 0 \wedge |x| < \?s n \rightarrow |?\text{diff} n x - f' x0 n| < \?r\).
with \((x \neq 0)\) and \(|x| < \?s n\) show \(|?\text{diff} n x - f' x0 n| < \?r\)
by blast

qed auto
also have \(\ldots = \text{af-nat (card \{..<\?N\})} \ast \?r\)
by (rule sum-constant)
also have \(\ldots = \text{real ?N} \ast \?r\)
by simp
also have \(\ldots = r/3\)
THEORY "Transcendental"

by (auto simp del: af-nat-Suc)
finally have |∑ n<?N. ?diff x z < f’ x0 n | < r / 3 (is ?diff-part < r / 3).

from suminf-diff[OF allf-sammable[OF x-in-I] allf-sammable[OF x0-in-I]]
have |(suminf (f (x0 + x)) - (suminf (f x0))) / x - suminf (f’ x0)| =
|∑ n. ?diff x0 n x - f’ x0 n |
unfolding suminf-diff[OF div-smbl (summable (f’ x0)), symmetric]
using suminf-divide[OF diff-smbl, symmetric] by auto
also have ... ≤ ?diff-part + |(∑ n. ?diff (n + ?N) x) - (∑ n. f’ x0 (n + ?N))|
unfolding suminf-split-initial-segment[OF all-smbl, where k=?N]
unfolding suminf-diff[OF div-shf-smbl ign[OF summable (f’ x0)]]]
apply (simp only: add.commute)
using abs-triangle-ineq by blast
also have ... ≤ ?diff-part + ?L-part + ?f’-part
using abs-triangle-ineq4 by auto
also have ... < r / 3 + r/3 + r / 3
using (?diff-part < r/3) (?L-part ≤ r/3) and (?f’-part < r/3)
by (rule add-strict-mono[OF add-less-le-mono])
finally show ?thesis
by auto
qed

then show ∃s > 0. ∀ x. x ≠ 0 ∧ norm (x - 0) < s ---
  norm |(∑ n. f (x0 + x) n) - (∑ n. f x0 n)| / x - (∑ n. f’ x0 n) < r
using |0 < S' by auto
qed

lemma DERIV-power-series':
fixes f :: nat ⇒ real
assumes converges: ∃x. x ∈ {-R <..< R} --- summable (λn. f n * real (Suc n))
and x0-in-I: x0 ∈ {-R <..< R}
and 0 < R
shows DERIV (λx. (∑ n. f n * x’(Suc n))) x0 : (∑ n. f n * real (Suc n) * x0’ n)
(is DERIV (λx. suminf (?f x)) x0 := suminf (?f’ x0))
proof --
  have for-subinterval: DERIV (λx. suminf (?f x)) x0 : suminf (?f’ x0)
  if 0 < R' and R' < R and -R' < x0 and x0 < R' for R'
proof --
  from that have x0 ∈ {-R’ <..< R’} and R’ ∈ {-R <..< R} and x0 ∈ {-R
  <..< R} by auto
show ?thesis
proof (rule DERIV-series')
show summable (λn. f n * real (Suc n) * R’ ’ n)
  proof --
  have (R’ + R) / 2 < R and 0 < (R’ + R) / 2
  using |0 < R' ∧ 0 < R' by (auto simp: field-simps)
then have in-Rball: \((R' + R) / 2 \in \{-R' <..< R\}\)
using \(\langle R' < R' \rangle \) by auto
have norm \(R' < \text{norm} \((R' + R) / 2\)\)
using \(\langle 0 < R' \rangle, 0 < R' : R' < R' \rangle \) by (auto simp: field-simps)
from powser-insidea[OF converges[OF in-Rball]] this show \(?thesis\)
by auto
qed

next
fix \(n x y\)
assume \(x \in \{-R' <..< R'\} \) and \(y \in \{-R' <..< R'\}\)
show \(\vert f n x - ?y n \vert \leq \vert f n \times \text{real} \, (\text{Suc} n) \times R' \times n \times \vert x - y \vert \)
proof
  have \([f n \times x \times (\text{Suc} n) - f n \times y \times (\text{Suc} n)] =\)
  \([(f n \times \vert x - y \vert) \times \sum_{p<\text{Suc} n} x \times p \times y \times (n - p)]\)
  unfolding right-diff-distrib["symmetric"] diff-power-eq-sum abs-mult
  by auto
  also have \(\ldots \leq ([f n \times \vert x - y \vert] \times (\text{real} \, (\text{Suc} n) \times R' \times n))\)
  proof (rule mult-left-mono)
    have \(\sum_{p<\text{Suc} n} x \times p \times y \times (n - p) \leq \sum_{p<\text{Suc} n} \vert x \times p \times y \times (n - p) \)\)
    by (rule sum-abs)
    also have \(\ldots \leq (\sum_{p<\text{Suc} n} R' \times n)\)
    proof (rule rule mono)
      fix \(p\)
      assume \(p \in \{..<\text{Suc} n\}\)
      then have \(p \leq n\) by auto
      have \(\vert x \times n \vert \leq R' \times n\) if \(x \in \{-R' <..< R'\}\) for \(n\) and \(x:: \text{real}\)
      proof
        from that have \(\vert x \vert \leq R'\) by auto
        then show \(?thesis\)
        unfolding power-abs by (rule rule mono)
      qed
      from mult-mono[OF this[OF \(\langle x \in \{-R' <..< R'\}, \, of \, p\rangle\)] this[OF \(\langle y \in \{-R' <..< R'\}, \, of \, n - p\rangle\)]
      and \(\langle 0 < R'\rangle\)
      have \(\vert x \times p \times y \times (n - p) \vert \leq R' \times p \times R' \times (n - p)\)
      unfolding abs-mult by auto
      then show \(\vert x \times p \times y \times (n - p) \vert \leq R' \times n \)
      unfolding power-add["symmetric"] using \(\langle p \leq n\rangle\) by auto
      qed
      also have \(\ldots = \text{real} \, (\text{Suc} n) \times R' \times n\)
      unfolding sum-constant card-atLeastLessThan by auto
      finally show \(\sum_{p<\text{Suc} n} \vert x \times p \times y \times (n - p) \vert \leq \vert \text{real} \, (\text{Suc} n) \times R' \times n \)
      unfolding abs-of-nonneg[OF less-imp-le[OF \(\langle 0 < R'\rangle\)]]
      by linarith
      show \(\theta \leq \vert f n \times \vert x - y \vert\)
      unfolding abs-mult["symmetric"] by auto
      qed
      also have \(\ldots = \vert f n \times \text{real} \, (\text{Suc} n) \times R' \times n \times \vert x - y \vert\)
unfolding \texttt{abs-mul} \texttt{mult.assoc}[\texttt{symmetric}] by \texttt{algebra}

finally show \texttt{?thesis}.

qed

next

show \texttt{DERIV} \((\lambda x. \ ?f x n)\) \(x_0 \mapsto \ ?f' x_0 n\) \texttt{for} \(n\)

by (auto intro!: \texttt{derivative-eq-intros} simp del: \texttt{power-Suc})

next

fix \(x\)

assume \(x \in \{-R' <..< R'\}\)

then have \(R' \in \{-R <..< R\}\) \texttt{and} \(\text{norm } x < \text{norm } R'\)

using \texttt{assms} \(R' < R\) by auto

have \texttt{summable} \((\lambda n. f n * x ^ n)\)

proof (rule \texttt{summable-comparison-test}, intro \texttt{exI allI impI})

fix \(n\)

have \(\text{le} : |f n| \cdot 1 \leq |f n| \cdot \text{real} (\text{Suc } n)\)

by (rule \texttt{mult-left-mono}) auto

show \(\text{norm} (f n * x ^ n) \leq \text{norm} (f n * \text{real} (\text{Suc } n) * x ^ n)\)

unfolding \texttt{real-norm-def} \texttt{abs-mul}

using \texttt{le mult-right-mono} by \texttt{fastforce}

qed (rule \texttt{powser-insidea}[\texttt{OF converges[\texttt{OF} \langle x_0 \in \{-R <..< R\}\rangle]} (\texttt{norm } x < \text{norm } R')])

from this\texttt{THEN} \texttt{summable-mult2[where} \texttt{c=x]}, \texttt{simplified} \texttt{mult.assoc, simplified} \texttt{mult.commute}]

show \texttt{summable} \((\lambda x. \ ?f x)\) by auto

next

show \texttt{summable} \((\lambda x. \ ?f' x)\)

using \texttt{converges[\texttt{OF} \langle x_0 \in \{-R' <..< R'\}\rangle]} .

show \(x_0 \in \{-R' <..< R'\}\)

using \(x_0 \in \{-R' <..< R'\}\) .

qed

qed

let \(?R = (R + \text{abs } x_0) / 2\)

have \(\text{abs } x_0 < ?R\)

using \texttt{assms} by (auto simp: \texttt{field-simps})

then have \(- ?R < x_0\)

proof (cases \texttt{x0 < 0})

case \texttt{True}

then have \(- x_0 < ?R\)

using \(|x_0| < ?R\) by auto

then show \texttt{?thesis}.

unfolding \texttt{neg-less_iff_less[\texttt{symmetric, of} - x_0]} by auto

next

case \texttt{False}

have \(- ?R < 0\) using \texttt{assms} by auto

also have \(\ldots \leq x_0\) using \texttt{False} by auto

finally show \texttt{?thesis}.

qed

then have \(0 < ?R \ ?R < R - ?R < x_0\) \texttt{and} \(x_0 < ?R\)

using \texttt{assms} by (auto simp: \texttt{field-simps})
from for-subinterval[OF this] show ?thesis .
qed

lemma geometric-deriv-sums:
  fixes z :: 'a :: {real-normed-field,banach}
  assumes norm z < 1
  shows \((\lambda n. of-nat (Suc n) * z ^ n) \sums (1 / (1 - z) ^ 2))\)
proof -
  have \((\lambda n. \text{diffs} (\lambda n. 1) n * z ^ n) \sums (1 / (1 - z) ^ 2))\)
  proof (rule termdiffs-sums-strong)
    fix z :: 'a assume norm z < 1
    thus \((\lambda n. 1 * z ^ n) \sums (1 / (1 - z))\) by (simp add: geometric-sums)
  qed (insert assms, auto intro!: derivative-eq-intros simp: power2_eq_square)
  thus ?thesis unfolding diffs-def by simp
qed

lemma isCont-pochhammer [continuous-intros]:
  isCont (\(\lambda z. \text{pochhammer} z n\)) z
  for z :: 'a::real-normed-field
  by (induct n) (auto simp: pochhammer-rec)

lemma continuous-on-pochhammer [continuous-intros]:
  continuous-on A (\(\lambda z. \text{pochhammer} z n\))
  for A :: 'a::real-normed-field set
  by (intro continuous-at-imp-continuous-on ballI isCont-pochhammer)

lemmas continuous-on-pochhammer' [continuous-intros] =
  continuous-on-compose2[OF continuous-on-pochhammer - subset_UNIV]

110.7 Exponential Function

definition exp :: 'a \Rightarrow 'a::{real-normed-algebra-1,banach}
  where exp = \((\lambda x. \sum n. x ^ n / R \text{fact n})\)

lemma summable-exp-generic:
  fixes x :: 'a::{real-normed-algebra-1,banach}
  defines S-def: \(S \equiv \lambda n. x ^ n / R \text{fact n}\)
  shows summable S
proof -
  have S-Suc: \(\forall n. S (Suc n) = (x * S n) / R (Suc n)\)
    unfolding S-def by (simp del: mult-Suc)
  obtain r :: real where r0: \(0 < r\) and r1: \(r < 1\)
    using dense[of zero-less-one] by fast
  obtain N :: nat where N: \(\text{norm} x < \text{real} N * r\)
    using ex-less-of-nat-mult r0 by auto
  from r1 show ?thesis
  proof (rule summable-ratio-test [rule-format])
    fix n :: nat
    assume n: \(N \leq n\)
    have \(\text{norm} x \leq \text{real} N * r\)
  qed
using $N$ by (rule order-less-imp-le) 
also have real $N \times r \leq$ real $(Suc \ n) \times r$
using $\forall \ n$ by (simp add: mult-right-mono)
finally have norm $x \times \text{norm} \ (S \ n) \leq$ real $(Suc \ n) \times r \times \text{norm} \ (S \ n)$
using norm-ge-zero by (rule mult-right-mono)
then have norm $(x \times S \ n) \leq$ real $(Suc \ n) \times r \times \text{norm} \ (S \ n)$
by (rule order-less-imp-le)
then have norm $(x \times S \ n) /$ real $(Suc \ n) \leq r \times \text{norm} \ (S \ n)$
by (simp add: pos-divide-le-eq ac-simps)
then show norm $(S \ (Suc \ n)) \leq r \times \text{norm} \ (S \ n)$
by (simp add: S-Suc inverse-eq-divide)

qed

lemma summable-norm-exp: summable $(\lambda n. \text{norm} \ (x \ ^{\ n} / R \ \text{fact} \ n))$
for $x :: 'a::{real-normed-algebra-1,banach}$
proof (rule summable-norm-comparison-test [OF exI, rule-format])
show summable $(\lambda n. \text{norm} \ (x \ ^{\ n} / R \ \text{fact} \ n))$
by (rule summable-exp-generic)
show norm $(x \ ^{\ n} / R \ \text{fact} \ n) \leq$ norm $(x \ ^{\ n} / R \ \text{fact} \ n)$
for $n$
by (simp add: norm-power-ineq)

qed

lemma summable-exp: summable $(\lambda n. \text{inverse} \ (\text{fact} \ n) \times x \ ^{\ n})$
for $x :: 'a::{real-normed-field,banach}$
using summable-exp-generic [where $x=x$]
by (simp add: scaleR-conv-of-real nonzero-of-real-inverse)

lemma exp-converges: $(\lambda n. \text{inverse} \ (\text{fact} \ n) \times x \ ^{\ n})$ sums $\exp \ x$
unfolding $\exp-def$ by (rule summable-exp-generic [THEN summable-sums])

lemma exp-fdiffs:
diffs $(\lambda n. \text{inverse} \ (\text{fact} \ n)) = (\lambda n. \text{inverse} \ (\text{fact} \ n) :: 'a::{real-normed-field,banach}))$
by (simp add: diffs-def mult-ac nonzero-inverse-mult-distrib nonzero-of-real-inverse
del: mult-Suc of-nat-Suc)

lemma diffs-of-real: diffs $(\lambda n. \text{of-real} \ (f \ n)) = (\lambda n. \text{of-real} \ (\diffs f \ n))$
by (simp add: diffs-def)

lemma DERIV-exp [simp]: DERIV $\exp \ x \Rightarrow \exp \ x$
unfolding $\exp-def$ scaleR-conv-of-real
proof (rule DERIV-cong)
have $\sinv$: summable $(\lambda n. \text{of-real} \ (\text{inverse} \ (\text{fact} \ n)) \times x \ ^{\ n})$ for $x::'a$
by (rule exp-converges [THEN sums-summable, unfolded scaleR-conv-of-real])

note $xx = exp-converges$ [THEN sums-summable, unfolded scaleR-conv-of-real]

show $(\lambda x. \sum n. \text{of-real} \ (\text{inverse} \ (\text{fact} \ n)) \times x \ ^{\ n})$ has-field-derivative
$(\sum n. \text{diffs} \ (\lambda n. \text{of-real} \ (\text{inverse} \ (\text{fact} \ n)) \times x \ ^{\ n}))$ (at $x$)
by (rule termadiffs [where $K=of-real \ (1 + norm \ x))$] (simp-all only: diffs-of-real
exp-fdiffs sinv norm-of-real)
show \((\sum n \cdot \text{diffs}(\lambda n \cdot \text{of-real}(\text{inverse}(\text{fact} n)))) \cdot x^n = (\sum n \cdot \text{of-real}(\text{inverse}(\text{fact} n)) \cdot x^n)\)
by (simp add: diffs-of-real exp-fdiffs)

qed

declare DERIV-exp[THEN DERIV-chain2, derivative-intros]
and DERIV-exp[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]

lemmas has-derivative-exp[derivative-intros] = DERIV-exp[THEN DERIV-compose-FDERIV]

lemma norm-exp: \(\text{norm}(\exp x) \leq \exp(\text{norm } x)\)
proof
  from summable-norm[OF summable-norm-exp, of x]
  have \(\text{norm}(\exp x) \leq (\sum n \cdot \text{inverse}(\text{fact } n) \cdot \text{norm}(x^n))\)
  by (simp add: exp-def)
  also have \(\ldots \leq \exp(\text{norm } x)\)
  using summable-exp-generic[of norm x] summable-norm-exp[OF x]
  by (auto simp: exp-def intro!: suminf_le norm-power-ineq)
finally show \(?thesis\).
qed

lemma isCont-exp: isCont \(\exp x\)
for \(x::'a::\{\text{real-normed-field},\text{banach}\}\)
by (rule DERIV-exp[THEN DERIV-isCont])

lemma isCont-exp'[simp]: \(\text{isCont } f\ a \Longrightarrow \text{isCont } (\lambda x. \exp(f x))\ a\)
for \(f::'a::\{\text{real-normed-field},\text{banach}\}\)
by (rule isCont-o2[OF - isCont-exp])

lemma tendsto-exp [tendsto-intros]: \(\lim_{x\to a} F = \lim_{x\to \exp(f x)}\ \exp(\lambda x. \exp(f x))\)
for \(f::'a::\{\text{real-normed-field},\text{banach}\}\)
by (rule isCont-tendsto-compose[OF isCont-exp])

lemma continuous-exp [continuous-intros]: \(\text{continuous } F\ f \Longrightarrow \text{continuous } F\ (\lambda x. \exp(f x))\)
for \(f::'a::\{\text{real-normed-field},\text{banach}\}\)
unfolding continuous-def by (rule tendsto-exp)

lemma continuous-on-exp [continuous-intros]: \(\text{continuous-on } s\ f \Longrightarrow \text{continuous-on } s\ (\lambda x. \exp(f x))\)
for \(f::'a::\{\text{real-normed-field},\text{banach}\}\)
unfolding continuous-on-def by (auto intro: tendsto-exp)

110.7.1 Properties of the Exponential Function

lemma exp-zero [simp]: \(\exp 0 = 1\)
unfolding exp-def by (simp add: scaleR-conv-of-real)
lemma exp-series-add-commuting:
  fixes x y :: 'a::{real-normed-algebra-1,banach}
  defines S-def: S ≡ λx n. x^n / R fact n
  assumes comm: x * y = y * x
  shows S (x + y) n = (∑ i≤n. S x i * S y (n - i))
proof (induct n)
  case 0
  show ?case
    unfolding S-def by simp
next
  case (Suc n)
  have S-Suc: (∀ x n. S x (Suc n) = (x * S x n) / R real (Suc n))
    unfolding S-def by (simp del: mult-Suc)
  then have times-S: ∀ x n. x * S x n = real (Suc n) * R S x (Suc n)
    by simp
  have S-comm: ∀ n. S x n * y = y * S x n
    by (simp add: power-commuting-commutes comm S-def)
  have real (Suc n) * R S (x + y) (Suc n) = (x + y) * S (x + y) n
    by (simp only: times-S)
  also have ... = (x + y) * (∑ i≤n. S x i * S y (n - i))
    by (simp only: Suc)
  also have ... = x * (∑ i≤n. S x i * S y (n - i)) + y * (∑ i≤n. S x i * S y (n - i))
    by (rule distrib-right)
  also have ... = (∑ i≤n. x * S x i * S y (n - i)) + (∑ i≤n. S x i * y * S y (n - i))
    by (simp add: sum-distrib-left ac-simps S-comm)
  also have ... = (∑ i≤n. x * S x i * S y (n - i)) + (∑ i≤n. S x i * (y * S y (n - i)))
    by (simp add: ac-simps)
  also have ... = (∑ i≤n. real (Suc i) * R (S x (Suc i) * S y (n - i))) +
               (∑ i≤n. real (Suc n - i) * R (S x i * S y (Suc n - i)))
    by (simp add: times-S Suc-diff-le)
  also have (∑ i≤n. real (Suc i) * R (S x (Suc i) * S y (n - i))) =
               (∑ i≤Suc n. real i * R (S x i * S y (Suc n - i)))
    by (subst sum.atMost-Suc-shift) simp
  also have (∑ i≤n. real (Suc n - i) * R (S x i * S y (Suc n - i))) =
               (∑ i≤Suc n. real (Suc n - i) * R (S x i * S y (Suc n - i)))
    by simp
  also have (∑ i≤Suc n. real i * R (S x i * S y (Suc n - i))) +
               (∑ i≤Suc n. real (Suc n - i) * R (S x i * S y (Suc n - i))) =
               (∑ i≤Suc n. real (Suc n) * R (S x i * S y (Suc n - i)))
    by (simp only: sum.distrib[symmetric] scaleR-left-distrib[symmetric] of_nat_add[symmetric]) simp
  also have ... = real (Suc n) * R (∑ i≤Suc n. S x i * S y (Suc n - i))
    by (simp only: scaleR-right.sum)
finally show S (x + y) (Suc n) = (∑ i≤Suc n. S x i * S y (Suc n - i))
  by (simp del: sum.cl_ivl-Suc)
qed

lemma `exp-add-commuting`: `x * y = y * x` → `exp (x + y) = exp x * exp y`
by (simp only: exp-def Cauchy-product summable-norm-exp exp-series-add-commuting)

lemma `exp-times-arg-commute`: `exp A * A = A * exp A`
by (simp add: exp-def suminf-mult[symmetric] summable-exp-generic power-commutes suminf-mult2)

lemma `exp-add`: `exp (x + y) = exp x * exp y`
for `x y` :: 'a::{real_normed_field,banach}
by (rule exp-add-commuting) (simp add: ac-simps)

lemma `exp-double`: `exp (2 * z) = exp z ^ 2`
by (simp add: exp-add-commuting mult-2 power2-eq-square)

lemmas `mult-exp-exp` = exp-add [symmetric]

lemma `exp-of-real`: `exp (of-real x) = of-real (exp x)`
unfolding exp-def
apply (subst suminf-of-real [OF summable-exp-generic])
apply (simp add: scaleR-conv-of-real)
done

lemmas `of-real-exp` = exp-of-real [symmetric]

corollary `exp-in-Reals [simp]`: `z ∈ ℝ` → `exp z ∈ ℝ`
by (metis Reals-cases Reals-of-real exp-of-real)

lemma `exp-not-eq-zero [simp]`: `exp x ≠ 0`
proof
have `exp x * exp (− x) = 1`
  by (simp add: exp-add-commuting[symmetric])
also assume `exp x = 0`
finally show `False` by simp
qed

lemma `exp-minus-inverse`: `exp x * exp (− x) = 1`
by (simp add: exp-add-commuting[symmetric])

lemma `exp-minus`: `exp (− x) = inverse (exp x)`
for `x` :: 'a::{real_normed_field,banach}
by (intro inverse-unique [symmetric] exp-minus-inverse)

lemma `exp-diff`: `exp (x − y) = exp x / exp y`
for `x` :: 'a::{real_normed_field,banach}
using `exp-add [of x − y]` by (simp add: exp-minus divide-inverse)

lemma `exp-of-nat-mult`: `exp (of-nat n * x) = exp x ^ n`
for $x :: 'a::{real-normed-field, banach}$
by (induct n) (auto simp: distrib-left exp-add mult.commute)

**corollary** exp-of-nat2-mult: $\exp (x \ast \text{of-nat } n) = \exp x \ast n$
for $x :: 'a::{real-normed-field, banach}$
by (metis exp-of-nat-mult mult-of-nat-commute)

**lemma** exp-sum: finite I $\implies \exp (\sum f I) = \prod (\lambda x. \exp (f x)) I$
by (induct I rule: finite-induct) (auto simp: exp-add-commuting mult.commute)

**lemma** exp-divide-power-eq:
fixes $x :: 'a::{real-normed-field, banach}$
assumes $n > 0$
shows $\exp (x / \text{of-nat } n) \ast n = \exp x$
using assms
proof (induction n arbitrary: x)
case (Suc n)
show ?case
proof (cases n = 0)
case True
then show ?thesis by simp
next
case False
have [simp]: $1 + (\text{of-nat } n \ast \text{of-nat } n + \text{of-nat } n \ast 2) \neq (0::'a)$
using of-nat-eq-iff [of $1 + n \ast n + n \ast 2 0$]
by simp
from False have [simp]: $x \ast \text{of-nat } n / (1 + \text{of-nat } n) / \text{of-nat } n = x / (1 + \text{of-nat } n)$
by simp
have [simp]: $x / (1 + \text{of-nat } n) + x \ast \text{of-nat } n / (1 + \text{of-nat } n) = x$
using of-nat-neq-0
by (auto simp add: field-split-simps)
show ?thesis
using Suc.IH [of $x \ast \text{of-nat } n / (1 + \text{of-nat } n)$] False
by (simp add: exp-add [symmetric])
qed simp

**110.7.2 Properties of the Exponential Function on Reals**

Comparisons of $\exp x$ with zero.

Proof: because every exponential can be seen as a square.

**lemma** exp-ge-zero [simp]: $0 \leq \exp x$
for $x :: \text{real}$
proof
have $0 \leq \exp (x/2) \ast \exp (x/2)$
  by simp
then show ?thesis
by (simp add: exp-add [symmetric])
qed

lemma exp-gt-zero [simp]: \(0 < \exp x\)
  for \(x :: \text{real}\)
  by (simp add: order-less-le)

lemma not-exp-less-zero [simp]: \(\neg \exp x < 0\)
  for \(x :: \text{real}\)
  by (simp add: not-less)

lemma not-exp-le-zero [simp]: \(\neg \exp x \leq 0\)
  for \(x :: \text{real}\)
  by (simp add: not-le)

lemma abs-exp-cancel [simp]: \(|\exp x| = \exp x\)
  for \(x :: \text{real}\)
  by simp

Strict monotonicity of exponential.

lemma exp-ge-add-one-self-aux: 
  fixes \(x :: \text{real}\)
  assumes \(0 \leq x\)
  shows \(1 + x \leq \exp x\)
  using order-le-imp-less-or-eq [OF assms]
proof 
  assume \(0 < x\)
  have \(1 + x \leq (\sum\ n<2. \text{inverse} (\text{fact} n) * x^n)\)
    by (auto simp: numeral-2-eq-2)
  also have \(\ldots \leq (\sum\ n. \text{inverse} (\text{fact} n) * x^n)\)
    apply (rule sum-le-suminf [OF summable-exp])
    using \(0 < x\)
    apply (auto simp add: zero-le-mult-iff)
  done
  finally show \(1 + x \leq \exp x\).
  qed auto

lemma exp-gt-one: \(0 < x \Longrightarrow 1 < \exp x\)
  for \(x :: \text{real}\)
proof 
  assume \(x: 0 < x\)
  then have \(1 < 1 + x\) by simp
  also from \(x\) have \(1 + x \leq \exp x\)
    by (simp add: exp-ge-add-one-self-aux)
  finally show \(?thesis\).
  qed

lemma exp-less-mono:
fixes $x \cdot y :: real$
assumes $x < y$
shows $exp \ x < exp \ y$
proof
  from $(x < y)$ have $0 < y - x$ by simp
  then have $1 < exp \ (y - x)$ by (rule exp-gt-one)
  then have $1 < exp \ y / exp \ x$ by (simp only: exp-diff)
  then show $exp \ x < exp \ y$ by simp
qed

lemma exp-less-cancel: $exp \ x < exp \ y \Rightarrow x < y$
  for $x \cdot y :: real$
  unfolding linorder-not-le [symmetric]
  by (auto simp: order-le-less exp-less-mono)

lemma exp-less-cancel-iff [iff]: $exp \ x < exp \ y \iff x < y$
  for $x \cdot y :: real$
  by (auto intro: exp-less-mono exp-less-cancel)

lemma exp-le-cancel-iff [iff]: $exp \ x \leq exp \ y \iff x \leq y$
  for $x \cdot y :: real$
  by (auto simp: linorder-not-less [symmetric])

lemma exp-inj-iff [iff]: $exp \ x = exp \ y \iff x = y$
  for $x \cdot y :: real$
  by (simp add: order-eq-iff)

Comparisons of $exp \ x$ with one.

lemma one-less-exp-iff [simp]: $1 < exp \ x \iff 0 < x$
  for $x :: real$
  using exp-less-cancel-iff [where $x = 0$ and $y = x$] by simp

lemma exp-less-one-iff [simp]: $exp \ x < 1 \iff x < 0$
  for $x :: real$
  using exp-less-cancel-iff [where $x = x$ and $y = 0$] by simp

lemma one-le-exp-iff [simp]: $1 \leq exp \ x \iff 0 \leq x$
  for $x :: real$
  using exp-le-cancel-iff [where $x = 0$ and $y = x$] by simp

lemma exp-le-one-iff [simp]: $exp \ x \leq 1 \iff x \leq 0$
  for $x :: real$
  using exp-le-cancel-iff [where $x = x$ and $y = 0$] by simp

lemma exp-eq-one-iff [simp]: $exp \ x = 1 \iff x = 0$
  for $x :: real$
  using exp-inj-iff [where $x = x$ and $y = 0$] by simp

lemma lemma-exp-total: $1 \leq y \Rightarrow \exists x. \ 0 \leq x \land x \leq y - 1 \land exp \ x = y$
for \( y :: \text{real} \)

proof (rule IVT)

assume \( 1 \leq y \)

then have \( 0 \leq y - 1 \) by simp

then have \( 1 + (y - 1) \leq \exp (y - 1) \)

by (rule exp-ge-add-one-self-aux)

then show \( y \leq \exp (y - 1) \) by simp

qed (simp-all add: le-diff-eq)

lemma exp-total: \( 0 < y \implies \exists x. \exp x = y \)

for \( y :: \text{real} \)

proof (rule linorder-le-cases [of \( y \)])

assume \( 1 \leq y \)

then show \( \exists x. \exp x = y \)

by (fast dest: lemma-exp-total)

next

assume \( 0 < y \) and \( y \leq 1 \)

then have \( 1 \leq \text{inverse } y \)

by (simp add: one-le-inverse-iff)

then obtain \( x \) where \( \exp x = \text{inverse } y \)

by (fast dest: lemma-exp-total)

then have \( \exp (-x) = y \)

by (simp add: exp-minus)

then show \( \exists x. \exp x = y \)

qed

110.8 Natural Logarithm

class \( \ln = \text{real-normed-algebra-1} + \text{banach} + \)

fixes \( \ln :: 'a \Rightarrow 'a \)

assumes \( \ln-one \) [simp]: \( \ln 1 = 0 \)

definition powr :: 'a => 'a => 'a :: ln (infixr powr 80)

— exponentiation via \( \ln \) and \( \exp \)

where \( x \text{ powr } a \equiv \text{if } x = 0 \text{ then } 0 \text{ else } \exp (a * \ln x) \)

lemma powr-0 [simp]: \( 0 \text{ powr } z = 0 \)

by (simp add: powr-def)

instantiation \( \text{real} :: \ln \)

begin

definition ln-real :: real => real

where \( \ln-real x = (\text{THE } u. \exp u = x) \)

instance

by intra-classes (simp add: ln-real-def)
end

lemma powr-eq-0-iff [simp]: \( w \powr z = 0 \iff w = 0 \)
  by (simp add: powr-def)

lemma ln-exp [simp]: \( \ln (\exp x) = x \)
  for \( x :: \text{real} \)
  by (simp add: ln-real-def)

lemma exp-ln [simp]: \( 0 < x \implies \exp (\ln x) = x \)
  for \( x :: \text{real} \)
  by (auto dest: exp-total)

lemma exp-ln-iff [simp]: \( \exp (\ln x) = x \iff 0 < x \)
  for \( x :: \text{real} \)
  by (metis exp-gt-zero exp-ln)

lemma ln-unique: \( \exp y = x \implies \ln x = y \)
  for \( x :: \text{real} \)
  by (erule subst) (rule ln-exp)

lemma ln-mult: \( 0 < x \implies 0 < y \implies \ln (x \cdot y) = \ln x + \ln y \)
  for \( x :: \text{real} \)
  by (rule ln-unique) (simp add: exp-add)

lemma ln-prod: finite \( I \implies (\forall i \in I \implies f i > 0) \implies \ln (\prod f I) = \sum (\lambda x. \ln(f x)) I \)
  for \( f :: 'a \Rightarrow \text{real} \)
  by (induct I rule: finite-induct) (auto simp: ln-mult prod-pos)

lemma ln-inverse: \( 0 < x \implies \ln (\inverse x) = -\ln x \)
  for \( x :: \text{real} \)
  by (rule ln-unique) (simp add: exp-minus)

lemma ln-div: \( 0 < x \implies 0 < y \implies \ln (x / y) = \ln x - \ln y \)
  for \( x :: \text{real} \)
  by (rule ln-unique) (simp add: exp-diff)

lemma ln-realpow: \( 0 < x \implies \ln (x^n) = n \cdot \ln x \)
  by (rule ln-unique) (simp add: exp-of-nat-mult)

lemma ln-less-cancel-iff [simp]: \( 0 < x \implies 0 < y \implies \ln x < \ln y \iff x < y \)
  for \( x :: \text{real} \)
  by (subst exp-less-cancel-iff [symmetric]) simp

lemma ln-le-cancel-iff [simp]: \( 0 < x \implies 0 < y \implies \ln x \leq \ln y \iff x \leq y \)
  for \( x :: \text{real} \)
  by (simp add: linorder-not-less [symmetric])
lemma ln-inj-iff [simp]: \(0 < x \implies 0 < y \implies \ln x = \ln y \iff x = y\)
  for \(x :: \text{real}\)
  by (simp add: order-eq-iff)

lemma ln-add-one-self-le-self: \(0 \leq x \implies \ln (1 + x) \leq x\)
  for \(x :: \text{real}\)
  by (rule exp-le-cancel-iff [THEN iffD1]) (simp add: exp-ge-add-one-self-aux)

lemma ln-less-self [simp]: \(0 < x \implies \ln x < x\)
  for \(x :: \text{real}\)
  by (rule order-less-le-trans [where \(y = \ln (1 + x)\)] (simp-all add: ln-add-one-self-le-self)

lemma ln-ge-iff: \(\forall x :: \text{real}. 0 < x \implies y \leq \ln x \iff \exp y \leq x\)
  using exp-le-cancel-iff exp-total by force

lemma ln-ge-zero [simp]: \(1 \leq x \implies 0 \leq \ln x\)
  for \(x :: \text{real}\)
  using ln-le-cancel-iff [of 1 x] by simp

lemma ln-ge-zero-imp-ge-one: \(0 \leq \ln x \implies 0 < x \implies 1 \leq x\)
  for \(x :: \text{real}\)
  using ln-le-cancel-iff [of 1 x] by simp

lemma ln-ge-zero-iff [simp]: \(0 < x \implies 0 \leq \ln x \iff 1 \leq x\)
  for \(x :: \text{real}\)
  using ln-le-cancel-iff [of 1 x] by simp

lemma ln-less-zero-iff [simp]: \(0 < x \implies \ln x < 0 \iff x < 1\)
  for \(x :: \text{real}\)
  using ln-less-cancel-iff [of x 1] by simp

lemma ln-le-zero-iff [simp]: \(0 < x \implies \ln x \leq 0 \iff x \leq 1\)
  for \(x :: \text{real}\)
  by (metis less-numeral-extra (1) ln-le-cancel-iff ln-one)

lemma ln-gt-zero: \(1 < x \implies 0 < \ln x\)
  for \(x :: \text{real}\)
  using ln-less-cancel-iff [of 1 x] by simp

lemma ln-gt-zero-imp-gt-one: \(0 < \ln x \implies 0 < x \implies 1 < x\)
  for \(x :: \text{real}\)
  using ln-less-cancel-iff [of 1 x] by simp

lemma ln-gt-zero-iff [simp]: \(0 < x \implies 0 < \ln x \iff 1 < x\)
  for \(x :: \text{real}\)
  using ln-less-cancel-iff [of 1 x] by simp

lemma ln-eq-zero-iff [simp]: \(0 < x \implies \ln x = 0 \iff x = 1\)
  for \(x :: \text{real}\)
using \text{ln-inj-iff} [of x 1] by simp

lemma \text{ln-less-zero}: \( 0 < x \implies x < 1 \implies \ln x < 0 \)
for \( x :: \text{real} \)
by simp

lemma \text{ln-neg-is-const}: \( x \leq 0 \implies \ln x = (\text{THE} \ x. \ False) \)
for \( x :: \text{real} \)
by (auto simp: \text{ln-real-def} intro: \text{arg-cong}[where \ f = \text{The}])

lemma \text{powr-eq-one-iff} [simp]:
\( a \ powr x = 1 \iff x = 0 \)
if \( a > 1 \)
for \( a, x :: \text{real} \)
using that by (auto simp: \text{powr-def} split: if-splits)

lemma \text{isCont-ln}:
fixes \( x :: \text{real} \)
assumes \( x \neq 0 \)
shows \( \text{isCont} \ \ln x \)
proof (cases \( 0 < x \))
  case True
  then have \( \text{isCont} \ \ln (\exp (\ln x)) \)
  by (intro \text{isCont-inverse-function}[where \ d = \mid x \mid \text{and} \ f = \exp]\) auto
  with True show \( ?\text{thesis} \)
  by simp
next
  case False
  with \( (x \neq 0) \) show \( \text{isCont} \ \ln x \)
unfolding \text{continuous-def}
  by (subst \text{filterlim-cong}[OF \ refl, of nhds (\ln 0) - \lambda -. \ ln 0])
    (auto simp: \text{ln-neg-is-const} \text{not-less} \text{eventually-at} \text{dist-real-def}
    intro!: \text{exI}[of - \mid x \mid])
qed

lemma \text{tendsto-ln} [tendsto-intros]: \( (f \longrightarrow a) \ F \implies a \neq 0 \implies ((\lambda x. \ln (f x)) \longrightarrow \ln a) \ F \)
for \( a :: \text{real} \)
by (rule \text{isCont-tendsto-compose} [OF \text{isCont-ln}])

lemma \text{continuous-ln}:
\( \text{continuous} \ \ F \ f \implies f (\text{Lim} \ F (\lambda x. x)) \neq 0 \implies \text{continuous} \ F (\lambda x. \ln (f x :: \text{real})) \)
unfolding \text{continuous-def} by (rule \text{tendsto-ln})

lemma \text{isCont-ln}’ [continuous-intros]:
\( \text{continuous} (\text{at} \ x) \ f \implies f x \neq 0 \implies \text{continuous} (\text{at} \ x) (\lambda x. \ln (f x :: \text{real})) \)
unfolding \text{continuous-at} by (rule \text{tendsto-ln})

lemma \text{continuous-within-ln} [continuous-intros]:
\( \text{continuous} (\text{at} \ x \ \text{within} \ s) \ f \implies f x \neq 0 \implies \text{continuous} (\text{at} \ x \ \text{within} \ s) (\lambda x. \ln (f x :: \text{real})) \)
unfolding continuous-within by (rule tendsto-ln)

lemma continuous-on-ln [continuous-intros]:
continuous-on s f \Rightarrow (\forall x \in s. f x \neq 0) \Rightarrow continuous-on s (\lambda x. \ln (f x :: real))
unfolding continuous-on-def by (auto intro: tendsto-ln)

lemma DERIV-ln: 0 < x \Rightarrow DERIV ln x :> inverse x
for x :: real
by (rule DERIV-inverse-function [where f=exp and a=0 and b=x+1])
(auto intro: DERIV-cong [OF DERIV-exp exp-ln] isCont-ln)

lemma DERIV-ln-divide: 0 < x \Rightarrow DERIV ln x :: 1 / x
for x :: real
by (rule DERIV-ln[THEN DERIV-cong]) (simp-all add: divide-inverse)

declare DERIV-ln-divide[THEN DERIV-chain2, derivative-intros]
and DERIV-ln-divide[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]

lemmas has-derivative-ln[derivative-intros] = DERIV-ln[THEN DERIV-compose-FDERIV]

lemma ln-series:
assumes 0 < x and x < 2
shows \ln x = (\sum n. (-1)^n * (1 / real (n + 1)) * (x - 1)^n) (Suc n)
(auto intro: DERIV-isconst3 [where x = x])

proof
assume x \in {0 <..< 2}
then have 0 < x and x < 2 by auto
have norm (1 - x) < 1
using (0 < x) and (x < 2) by auto
have 1 / x = 1 / (1 - (1 - x)) by auto
also have \ldots = (\sum n. (1 - x)^n)
using geometric-sums[OF \norm (1 - x) < 1] by (rule sums-unique)
also have \ldots = suminf (?f' x)
unfolding power-mult-distrib[symmetric]
by (rule arg-cong[where f=suminf], rule arg-cong[where f=(\cdot)], auto)
finally have DERIV ln x :: suminf (?f' x)
using DERIV-ln[OF \(0 < x\)] unfolding divide-inverse by auto
moreover
have repos: \\ h x :: real. h - 1 + x = h + x - 1 by auto
have DERIV (\lambda x. suminf (?f x) (x - 1)) :-
(\sum n. (-1)^n * (1 / real (n + 1)) * real (Suc n) * (x - 1)^n)
proof (rule DERIV-power-series')
show x - 1 \in \{- 1<..<1\} and (0 :: real) < 1
using \(0 < x \land x < 2\) by auto

next
fix \(x::\text{real}\)
assume \(x \in \{-1 <..< 1\}\)
then have \(\text{norm}(-x) < 1\) by auto

show \(\text{summable}(\lambda n. (-1)^n * (1/\text{real}(n+1)) * \text{real}(\text{Suc} n) * x^n)\)
unfolding \(\text{One-nat-def}\)
by (auto simp: \(\text{power-mult-distrib}\) \(\text{symmetric}\) \(\text{summable-geometric}\) \(\text{OF} \ \\text{norm}(-x) < 1\))
qed

then have \(\text{DERIV}(\lambda x. \text{suminf}(\lambda f. x)) (x-1) > \text{suminf}(\lambda f' x)\)
unfolding \(\text{DERIV-def repos}\)
then have \(\text{DERIV}(\lambda x. \text{suminf}(\lambda f. x)) x > \text{suminf}(\lambda f' x)\)
unfolding \(\text{DERIV-def repos}\)
ultimately have \(\text{DERIV}(\lambda x. \text{ln} x - \text{suminf}(\lambda f. x - 1)) x > 0\) by auto
then show \(?\text{thesis}\) by auto

qed

lemma \(\text{exp-first-terms}\):
fixes \(x::\text{a::\{real-normed-algebra-1,banach}\}\)
shows \(\exp x = (\sum n<k. \text{inverse}(\text{fact} n) \ast_R (x^n)) + (\sum n. \text{inverse}(\text{fact}(n+k)) \ast_R (x^n (n+k)))\)
proof -
have \(\exp x = \text{suminf}(\lambda n. \text{inverse}(\text{fact} n) \ast_R (x^n))\)
by (simp add: \(\text{exp-def}\))
also from \(\text{summable-exp-generic}\) have \(\ldots = (\sum n. \text{inverse}(\text{fact}(n+k)) \ast_R (x^n (n+k))) +\)
(\(\sum n::\text{nat}<k. \text{inverse}(\text{fact} n) \ast_R (x^n)\)) \(\text{is} = - + ?a\)
by (rule \(\text{suminf-split-initial-segment}\))
finally show \(?\text{thesis}\) by simp
qed

lemma \(\text{exp-first-term}: \exp x = 1 + (\sum n. \text{inverse}(\text{fact}(n)) \ast_R (x^n))\)
for \(x::\text{a::\{real-normed-algebra-1,banach}\}\)
using \(\text{exp-first-terms}[\text{of} \ x \ 1]\) by simp

lemma \(\text{exp-first-two-terms}: \exp x = 1 + x + (\sum n. \text{inverse}(\text{fact}(n+2)) \ast_R (x^n (n+2)))\)
for \(x::\text{a::\{real-normed-algebra-1,banach}\}\)
using \(\text{exp-first-terms}[\text{of} \ x \ 2]\) by (simp add: \(\text{eval-nat-numeral}\))

lemma \(\text{exp-bound}:\)
fixes \(x::\text{real}\)
assumes \(a:0 \leq x\)
and \(b:x \leq 1\)
shows \( \exp x \leq 1 + x + x^2 \)

proof –

have suminf \((\lambda n. \text{inverse}(\text{fact} (n+2)) \cdot (x ^ (n + 2))) \leq x^2\)

proof –

have \((\lambda n. x^2 / 2 \cdot (1 / 2) ^ n) \text{ sums} (x^2 / 2 \cdot (1 / (1 - 1 / 2)))\)
  by \((\text{intro sums-mult geometric-sums})\) simp
then have sumsx: \((\lambda n. x^2 / 2 \cdot (1 / 2) ^ n) \text{ sums} x^2\)
  by simp
have suminf \((\lambda n. \text{inverse}(\text{fact} (n+2)) \cdot (x ^ (n + 2))) \leq \text{suminf} \((\lambda n. (x^2/2)^{((1/2) ^ n)})\)
  \((\text{intro suminf-le allI})\)
show inverse \((\text{fact} (n + 2)) \cdot x ^ (n + 2) \leq (x^2/2) * ((1/2) ^ n)\) \text{ for } n :: nat
  proof –
  have \((2::\text{nat}) \cdot 2 ^ n \leq \text{fact} (n + 2)\)
    by \((\text{induct n})\) simp-all
  then have real \(((2::\text{nat}) \cdot 2 ^ n) \leq \text{real-of-nat} (\text{fact} (n + 2))\)
    by \((\text{simp only: of-nat-le-iff})\)
  then have \(((2::\text{real}) \cdot 2 ^ n) \leq \text{fact} (n + 2)\)
    unfolding \(\text{of-nat-fact}\) by simp
  then have inverse \((\text{fact} (n + 2)) \leq \text{inverse} ((2::\text{real}) \cdot 2 ^ n)\)
    by \((\text{rule le-imp-inverse-le})\) simp
  then have inverse \((\text{fact} (n + 2)) \leq 1/(2::\text{real}) * (1/2) ^ n\)
    by \((\text{simp add: power-inverse \([\text{symmetric}]\)})\)
  then have inverse \((\text{fact} (n + 2)) \cdot (x ^ n \cdot x^2) \leq 1/2 * (1/2) ^ n * (1 * x^2)\)
    by \((\text{rule mult-mono})\) \((\text{rule mult-mono, simp-all add: power-le-one a b})\)
then show \?thesis
  unfolding \(\text{power-add}\) by \((\text{simp add: ac-simps del: fact-Suc})\)
qed
show summable \((\lambda n. \text{inverse}(\text{fact} (n+2)) \cdot x ^ (n + 2))\)
  by \((\text{rule summable-exp \([\text{THEN summable-ignore-initial-segment}]\)})\)
show summable \((\lambda n. x^2 / 2 \cdot (1 / 2) ^ n)\)
  by \((\text{rule sums-summable \([\text{OF sumsx}]\)})\)
qed
also have ... = \(x^2\)
  by \((\text{rule sums-unique \([\text{THEN sym}]\)})\) \((\text{rule sumsx})\)
finally show \?thesis .
qed
then show \?thesis
  unfolding \(\text{exp-first-two-terms}\) by auto
qed

\textbf{corollary} \(\exp(1/2) \leq (2::\text{real})\)

using \(\exp-bound \([1/2]\)\)
  by \((\text{simp add: field-simps})\)

\textbf{corollary} \(\exp 1 \leq (3::\text{real})\)

using \(\exp-bound \([1]\)\)
THEORY "Transcendental"

by (simp add: field-simps)

lemma exp-bound-half: \( \text{norm } z \leq 1/2 \implies \text{norm } (\exp z) \leq 2 \)
by (blast intro: order-trans intro: exp-half-le2 norm-exp)

lemma exp-bound-lemma:
assumes \( \text{norm } z \leq 1/2 \)
shows \( \text{norm } (\exp z) \leq 1 + 2 * \text{norm } z \)
proof
have \( \ast: (\text{norm } z)^2 \leq \text{norm } z * 1 \)
  unfolding power2-eq-square
  by (rule mult-left-mono) (use assms in auto)
have \( \text{norm } (\exp z) \leq \exp (\text{norm } z) \)
  by (rule norm-exp)
also have \( \ldots \leq 1 + (\text{norm } z) + (\text{norm } z)^2 \)
  using assms exp-bound by auto
also have \( \ldots \leq 1 + 2 * \text{norm } z \)
  using \( \ast \) by auto
finally show \( \astheorem \).
qed

lemma real-exp-bound-lemma: \( 0 \leq x \implies x \leq 1/2 \implies \exp x \leq 1 + 2 * x \)
for \( x :: \text{real} \)
using exp-bound-lemma [of \( x \)] by simp

lemma ln-one-minus-pos-upper-bound:
fixes \( x :: \text{real} \)
assumes \( a: 0 \leq x \) and \( b: x < 1 \)
shows \( \ln (1 - x) \leq -x \)
proof
have \( (1 - x) * (1 + x + x^2) = 1 - x^3 \)
  by (simp add: algebra-simps power2-eq-square power3-eq-cube)
also have \( \ldots \leq 1 \)
  by (auto simp: a)
finally have \( (1 - x) * (1 + x + x^2) \leq 1 \).
moreover have \( c: 0 < 1 + x + x^2 \)
  by (simp add: add-pos-nonneg a)
ultimately have \( 1 - x \leq 1 / (1 + x + x^2) \)
  by (elim mult-imp-le-div-pos)
also have \( \ldots \leq 1 / \exp x \)
  by (metis a abs-one b exp-bound exp-gt-zero frac-le less-eq-real-def real-sqrt-iff real-sqrt-power)
also have \( \ldots = \exp (-x) \)
  by (auto simp: exp-minus divide-inverse)
finally have \( 1 - x \leq \exp (-x) \).
also have \( 1 - x = \exp (\ln (1 - x)) \)
  by (metis b diff-0 exp-ln-iff less-iff-diff-less-0 minus-diff-eq)
finally have \( \exp (\ln (1 - x)) \leq \exp (-x) \).
then show \( \astheorem \).
by (auto simp only: exp-le-cancel-iff)

qed

lemma exp-ge-add-one-self [simp]: \(1 + x \leq \exp x\)
for \(x :: \text{real}\)
proof (cases \(0 \leq x \vee x \leq -1\))
case True
then show \(?thesis\)
apply (rule disjE)
apply (simp add: exp-ge-add-one-self-aux)
using exp-ge-zero order-trans real-add-le-0-iff by blast
next
case False
then have \(\ln 1 : \ln (1 + x) \leq x\)
using ln-one-minus-pos-upper-bound \([of -x]\) by simp
have \(1 + x = \exp (\ln (1 + x))\)
using False by auto
also have \(?\) \(\leq \exp x\)
by (simp add: ln1)
finally show \(?thesis\).

qed

lemma ln-one-plus-pos-lower-bound:
fixes \(x :: \text{real}\)
assumes \(a : 0 \leq x \\text{ and } b : x \leq 1\)
shows \(x - x^2 \leq \ln (1 + x)\)
proof -
have \(\exp (x - x^2) = \exp x / \exp (x^2)\)
by (rule exp-diff)
also have \(\ldots \leq (1 + x + x^2) / \exp (x^2)\)
by (metis a b divide-right-mono exp-bound exp-ge-zero)
also have \(\ldots \leq (1 + x + x^2) / (1 + x^2)\)
by (simp add: a divide-left-mono add-pos-nonneg)
also from \(a \) have \(\ldots \leq 1 + x\)
by (simp add: field-simps add-strict-increasing zero-le-mult-iff)
finally have \(\exp (x - x^2) \leq 1 + x\).
also have \(\ldots = \exp (\ln (1 + x))\)
proof -
from \(a \) have \(0 < 1 + x \) by auto
then show \(?thesis\)
by (auto simp only: exp-ln-iff [THEN sym])
qed
finally have \(\exp (x - x^2) \leq \exp (\ln (1 + x))\).
then show \(?thesis\)
by (metis exp-le-cancel-iff)

qed

lemma ln-one-minus-pos-lower-bound:
fixes \(x :: \text{real}\)
assumes \( a: 0 \leq x \) and \( b: x \leq 1 / 2 \)
shows \(-x - 2 * x^2 \leq \ln (1 - x)\)

proof –
  from \( b \) have \( c: x < 1 \) by auto
  then have \( \ln (1 - x) = - \ln (1 + x / (1 - x)) \)
  by (auto simp: ln-inverse \[symmetric\] field-simps intro: arg-cong \[where f=ln\])
  also have \(- (x / (1 - x)) \leq \ldots\)

proof –
  have \( \ln (1 + x / (1 - x)) \leq x / (1 - x) \)
  using \( a \) by (intro ln-add-one-self-le-self) auto
  then show \(?thesis\)
  by auto

qed

also have \(- (x / (1 - x)) = - x / (1 - x)\)
by auto

finally have \( d: -x / (1 - x) \leq \ln (1 - x)\).

have \( 0 < 1 - x \) using \( a \) by simp
then have \( c: -x - 2 * x^2 \leq -x / (1 - x) \)
using \( mult-right-le-one-le \[of x * x 2 * x\] a b \)
by (simp add: field-simps power2-eq-square)

from \( e \) \( d \) show \(-x - 2 * x^2 \leq \ln (1 - x)\)
by (rule order-trans)

qed

lemma \( \text{ln-add-one-self-le-self}\): 
fixes \( x :: \text{real} \)
shows \(-1 < x \Longrightarrow \ln (1 + x) \leq x\)
by (metis diff-gt-0-iff-gt diff-minus-eq-add exp-ge-add-one-self exp-le-cancel-iff exp-ln minus-less-iff)

lemma \( \text{abs-ln-one-plus-x-minus-x-bound-nonneg}\):
fixes \( x :: \text{real} \)
assumes \( x: 0 \leq x \) and \( x1: x \leq 1 \)
shows \(|\ln (1 + x) - x| \leq x^2\)

proof –
  from \( x \) have \( \ln (1 + x) \leq x \)
  by (rule ln-add-one-self-le-self)
  then have \( \ln (1 + x) - x \leq 0 \)
  by simp
  then have \(|\ln (1 + x) - x| = -(\ln (1 + x) - x)\)
  by (rule abs-of-nonpos)
  also have \ldots \leq x - \ln (1 + x)
  by simp
  also have \ldots \leq x^2
  proof –
    from \( x \) \( x1 \) have \( x - x^2 \leq \ln (1 + x) \)
    by (intro ln-one-plus-pos-lower-bound)
    then show \(?thesis\)
    by simp
finally show \( \text{thesis} \).

**lemma abs-ln-one-plus-x-minus-x-bound-nonpos:**
- fixes \( x :: \text{real} \)
- assumes \( a : -(1 / 2) \leq x \) and \( b : x \leq 0 \)
- shows \( |\ln (1 + x) - x| \leq 2 \cdot x^2 \)
- proof –
  - have \( \*: -(-x) - 2 \cdot (-x)^2 \leq \ln (1 - (-x)) \)
    - by (metis a b diff-zero ln-one-minus-pos-lower-bound minus-diff-eq neg-le-iff-le)
  - have \( |\ln (1 + x) - x| = x - \ln (1 - (-x)) \)
    - using a ln-add-one-self-le-self2 [of \( x \)] by (simp add: abs-if)
    - also have \( \ldots \leq 2 \cdot x^2 \)
      - using \( \* \) by (simp add: algebra-simps)
    - finally show \( \text{thesis} \).

**lemma abs-ln-one-plus-x-minus-x-bound:**
- fixes \( x :: \text{real} \)
- assumes \( |x| \leq 1 / 2 \)
- shows \( |\ln (1 + x) - x| \leq 2 \cdot x^2 \)
- proof (cases \( 0 \leq x \))
  - case True
    - then show \( \text{thesis} \)
      - using abs-ln-one-plus-x-minus-x-bound-nonneg assms by fastforce
  - case False
    - then show \( \text{thesis} \)
      - using abs-ln-one-plus-x-minus-x-bound-nonpos assms by auto

**lemma ln-x-over-x-mono:**
- fixes \( x :: \text{real} \)
- assumes \( x : \exp 1 \leq x \leq y \)
- shows \( \ln y / y \leq \ln x / x \)
- proof –
  - note \( x \)
  - moreover have \( 0 < \exp (1 :: \text{real}) \) by simp
  - ultimately have \( a : 0 < x \) and \( b : 0 < y \)
    - by (fast intro: less-le-trans order-trans)+
  - have \( x \cdot \ln y - x \cdot \ln x = x \cdot (\ln y - \ln x) \)
    - by (simp add: algebra-simps)
  - also have \( \ldots = x \cdot \ln (y / x) \)
    - by (simp only: ln-div a b)
  - also have \( y / x = (x + (y - x)) / x \)
    - by simp
  - also have \( \ldots = 1 + (y - x) / x \)
using 

also have 

using 
by 
also have 
using 
by 
also have 
using 
by 
also have 
using 
by 
also have 
using 
by 
also have 
using 
by 
also have 
using 
by 
also have 
using 
by 
also have 
using 
by 
also have 
using 
by 
also have 
using 
by 
also have 
using 
by 
also have 
using 
by 
also have 
using 
by 
also have 
using 
by 
also have 
using 
by 
finally have 
by 
then have 
by 
finally show 
using 
by 

lemma ln-le-minus-one: 
for 
using 

corollary ln-diff-le: 
for 
by 

lemma ln-eq-minus-one: 
fixes 
assumes 
shows 
proof -- 
let 
have 
by 
show 
proof 
assume 
from 
from 
proof 
fix 
assume 
with 
have 
by 
with 
show 
by 
qed
also have \( \ldots \leq 0 \)
using \( \ln\leq-\text{minus-one} \) \( \langle 0 < x \rangle \langle x < a \rangle \) by (auto simp: field-simps)
finally show \( x = 1 \) using assms by auto

next
assume \( 1 < x \)
from dense[OF this] obtain \( a \) where \( 1 < a \ a < x \) by blast
proof (rule DERIV-neg-imp-decreasing)
fix \( y \)
assume \( a \leq y \ y \leq x \)
with \( \langle 1 < a \rangle \) have \( 1 / y - 1 < 0 \ 0 < y \)
by (auto simp: field-simps)
with \( D \) show \( \exists z. \text{DERIV } \not\exists y : z \wedge z < 0 \)
by blast
qed
also have \( \ldots \leq 0 \)
using \( \ln\leq-\text{minus-one} \) \( \langle 1 < a \rangle \) by (auto simp: field-simps)
finally show \( x = 1 \) using assms by auto

next
assume \( x = 1 \)
then show \( ?\text{thesis} \) by simp

qed

lemma \( \ln\times-over-x-tendsto-0 \): ((\( \lambda x :: \text{real} \). \ln x / x) \longrightarrow 0) at-top
proof (rule lhospital-at-top-at-top[where \( f' = \text{inverse} \) and \( g' = \lambda -. 1 \)])
from eventually-gt-at-top[where \( \not\exists 0 :: \text{real} \)]
show \( \forall F \ x \in \text{at-top}. (\ln \text{has-real-derivative } \text{inverse } x) (\text{at } x) \)
by eventually-elim (auto intro: derivative-eq-intros simp: field-simps)
qed (use tendsto-inverse-0 in \( \langle \text{auto simp: filterlim-ident dest!: tendsto-mono[OF at-top-le-at-infinity]} \rangle \))

lemma \( \exp\geq\text{-one-plus-x-over-n-power-n} \):
assumes \( x \geq - \text{real } n \ n > 0 \)
shows \( (1 + x / \text{of-nat } n) ^ n \leq \exp x \)
proof (cases \( x = - \text{of-nat } n \))
  case False
  from assms False have \( (1 + x / \text{of-nat } n) ^ n = \exp (\text{of-nat } n * \ln (1 + x / \text{of-nat } n)) \)
by (subst exp-of-nat-mult, subst exp-ln) (simp-all add: field-simps)
also from assms False have \( \ln (1 + x / \text{real } n) \leq x / \text{real } n \)
by (intro ln-add-one-self-le-self2) (simp-all add: field-simps)
with assms have \( \exp (\text{of-nat } n * \ln (1 + x / \text{of-nat } n)) \leq \exp x \)
by (simp add: field-simps)
finally show \( ?\text{thesis} \).

next
  case True
  then show \( ?\text{thesis} \) by (simp add: zero-power)
qed
lemma \text{exp-ge-one-minus-x-over-n-power-n}:  
\text{assumes } x \leq \text{real n n > 0}  
\text{shows } \left(1 - \frac{x}{\text{of-nat n}}\right)^n \leq \exp(-x)  
\text{using } \text{exp-ge-one-plus-x-over-n-power-n[of n - x]} \text{ assms by simp}

lemma \text{exp-at-bot}: \left(\exp \longrightarrow (\text{0::real})\right) \text{ at-bot}  
\text{unfolding tendsto-Zfun-iff}  
\text{proof (rule ZfunI, simp add: eventually-at-bot-dense)}  
\text{fix } r :: \text{real}  
\text{assume } 0 < r  
\text{have } \exp x < r \text{ if } x < \ln r \text{ for } x  
\text{ by (metis } 0 < r \text{ exp-less-mono exp-ln that)}  
\text{then show } \exists k. \forall n < k. \exp n < r \text{ by auto}  
\text{qed}

lemma \text{exp-at-top}: \text{LIM x at-top. } \exp x :: \text{real}  
\text{by (rule filterlim-at-top-at-top[where } Q = \lambda x. \text{True and } P = \lambda x. 0 < x \text{ and } g = \ln]\}  
\text{ (auto intro: eventually-gt-at-top)}

lemma \text{lim-exp-minus-1}: \left(\lambda z :: \text{'}a. \frac{(\exp(z) - 1)}{z} \longrightarrow 1\right) \text{ (at 0)}  
\text{for } x :: \text{'}a::\{\text{real-normed-field,banach}\}  
\text{proof (intro derivative-eq-intros | simp)+}  
\text{then show } \text{thesis}  
\text{by (simp add: Deriv.has-field-derivative-iff)}  
\text{qed}

lemma \text{ln-at-0}: \text{LIM x at-right 0. } \ln x :: \text{real} \Rightarrow \text{at-bot}  
\text{by (rule filterlim-at-bot-at-right[where } Q = \lambda x. \text{0 < x and } P = \lambda x. \text{True and } g = \exp]\}  
\text{ (auto simp: eventually-at-filter)}

lemma \text{ln-at-top}: \text{LIM x at-top. } \ln x :: \text{real} \Rightarrow \text{at-top}  
\text{by (rule filterlim-at-top-at-top[where } Q = \lambda x. \text{0 < x and } P = \lambda x. \text{True and } g = \exp]\}  
\text{ (auto intro: eventually-gt-at-top)}

lemma \text{filtermap-ln-at-top}: \text{filtermap } \ln :: \text{real \Rightarrow real} \text{ at-top = at-top}  
\text{by (intro filtermap-fun-inverse[of exp] exp-at-top ln-at-top) auto}

lemma \text{filtermap-exp-at-top}: \text{filtermap } \exp :: \text{real \Rightarrow real} \text{ at-top = at-top}  
\text{by (intro filtermap-fun-inverse[of exp] exp-at-top ln-at-top) auto}

\text{lemma filtermap-ln-at-right}: \text{filtermap } \ln \text{ (at-right } (0::real)) \text{ at-bot}  
\text{by (auto intro: filtermap-fun-inverse[where } g = \lambda x. \exp x]\text{ ln-at-0 simp: filterlim-at-exp-at-bot)
lemma tendsto-power-div-exp-0: \((\lambda x. x ^ k / \exp x) \longrightarrow (0::real)\) at-top
proof (induct k)
case 0
  show \((\lambda x. x ^ 0 / \exp x) \longrightarrow (0::real)\) at-top
  by (simp add: inverse-eq-divide[symmetric])
  (metis filterlim-compose[OF tendsto-inverse-0] exp-at-top filterlim-mono
   at-top-le-at-infinity order-refl)
ext
case (Suc k)
  show ?case
proof (rule lhospital-at-top-at-top)
  show eventually \((\lambda x. \DERIV (\lambda x. x ^ \text{Suc} k) x :> (\text{real} (\text{Suc} k) * x ^ k))\) at-top
  by eventually-elim (intro derivative-eq-intros, auto)
  show eventually \((\lambda x. \DERIV \exp x x :> \exp x)\) at-top
  by eventually-elim auto
  show eventually \((\lambda x. \exp x \neq 0)\) at-top
  by auto
  from tendsto-mult[OF tendsto-const Suc, of real (Suc k)]
  show \((\lambda x. \text{real} (\text{Suc} k) * x ^ \text{Suc} k / \exp x) \longrightarrow 0)\) at-top
  by simp
qed (rule exp-at-top)
qed

110.8.1 A couple of simple bounds

lemma exp-plus-inverse-exp:
  fixes x::real
  shows \(2 \leq \exp x + \inverse (\exp x)\)
proof
  have \(2 \leq \exp x + \exp (-x)\)
  using exp-plus-inverse-exp by (fastforce intro: derivative-eq-intros DERIV-nonneg-imp-nondecreasing[OF that])
  show ?thesis
  using \(\exp x + \inverse (\exp x) \leq 2\) by simp
qed

lemma real-le-x-sinh:
  fixes x::real
  assumes \(0 \leq x\)
  shows \(x \leq (\exp x - \inverse (\exp x)) / 2\)
proof
  have \(*: \exp a - \inverse (\exp a) - 2*a \leq \exp b - \inverse (\exp b) - 2*b \) if \(a \leq b\) for \(a b::real\)
  using exp-plus-inverse-exp
  by (fastforce intro: derivative-eq-intros DERIV-nonneg-imp-nondecreasing[OF that])
  show ?thesis
  using \(*\) by simp
lemma real-le-abs-sinh:
  fixes x :: real
  shows abs x \leq abs \left( (exp x - inverse(exp x)) / 2 \right)
proof (cases 0 \leq x)
  case True
  show ?thesis using real-le-x-sinh [OF True] True by (simp add: abs-if)
next
  case False
  have \(-x \leq (exp(-x) - inverse(exp(-x))) / 2\)
  by (meson False linear neg-le-0-iff real-le-x-sinh)
  also have \(-x \leq \left| (exp x - inverse(exp x)) / 2 \right|\)
  by (metis (no-types, hide-lams) abs-divide abs-le-iff abs-minus-cancel add.inverse-inverse exp-minus minus-diff-eq order-refl)
  finally show ?thesis using False by linarith
qed

110.9 The general logarithm

definition log :: real \Rightarrow real \Rightarrow real
  — logarithm of \(x\) to base \(a\)
  where \(\log a x = \ln x / \ln a\)

lemma tendsto-log [tendsto-intros]:
  \((f \longrightarrow a) \Longrightarrow (g \longrightarrow b) \Longrightarrow 0 < a \Longrightarrow a \neq 1 \Longrightarrow 0 < b \Longrightarrow (\lambda x. \log (f x) (g x)) \longrightarrow \log a b) \Longrightarrow\)
unfolding log-def by (intro tendsto-intros) auto

lemma continuous-log:
  assumes continuous F f
  and continuous F g
  and 0 < f (Lim F (\lambda x. x))
  and f (Lim F (\lambda x. x)) \neq 1
  and 0 < g (Lim F (\lambda x. x))
  shows continuous F (\lambda x. log (f x) (g x))
using assms unfolding continuous-def by (rule tendsto-log)

lemma continuous-at-within-log[continuous-intros]:
  assumes continuous (at a within s) f
  and continuous (at a within s) g
  and 0 < f a
  and f a \neq 1
  and 0 < g a
  shows continuous (at a within s) (\lambda x. log (f x) (g x))
using assms unfolding continuous-within by (rule tendsto-log)
lemma isCont-log [continuous-intros, simp]:
assumes isCont f a isCont g a 0 < f a f a ≠ 1 0 < g a
shows isCont (λx. log (f x) (g x)) a
using assms unfolding continuous-at by (rule tendsto-log)

lemma continuous-on-log [continuous-intros]:
assumes continuous-on s f continuous-on s g
and ∀ x ∈ s. 0 < f x ∀ x ∈ s. f x ≠ 1 ∀ x ∈ s. 0 < g x
shows continuous-on s (λx. log (f x) (g x))
using assms unfolding continuous-on-def by (fast intro: tendsto-log)

lemma powr-one-eq-one [simp]: 1 powr a = 1
by (simp add: powr-def)

lemma powr-zero-eq-one [simp]: x powr 0 = (if x = 0 then 0 else 1)
by (simp add: powr-def)

lemma powr-one-gt-zero-iff [simp]: x powr 1 = x ←→ 0 ≤ x
for x :: real
by (auto simp: powr-def)

declare powr-one-gt-zero-iff [THEN iffD2, simp]

lemma powr-mult [0 ≤ x =⇒ 0 ≤ y =⇒ (x * y) powr a = (x powr a) * (y powr a)]
for a x y :: real
by (simp add: powr-def exp-add [symmetric] ln-mult distrib-left)

lemma powr-ge-pzero [simp]: 0 ≤ x powr y
for x y :: real
by (simp add: powr-def)

lemma powr-non-neq [simp]: ¬ a powr x < 0 for a x::real
using powr-ge-pzero[of a x] by arith

lemma powr-divide [0 ≤ x; 0 ≤ y] =⇒ (x / y) powr a = (x powr a) / (y powr a)
for a b x :: real
apply (simp add: divide-inverse positive-imp-inverse-positive powr-mult)
done

lemma powr-add: x powr (a + b) = (x powr a) * (x powr b)
for a b x :: 'a::{ln,real-normed-field}
by (simp add: powr-def exp-add [symmetric] distrib-right)
lemma pour-mult-base: 0 ≤ x ⟹ x * x powr y = x powr (1 + y)
  for x :: real
  by (auto simp: powr-add)

lemma pour-pour: (x powr a) powr b = x powr (a * b)
  for a b x :: real
  by (simp add: powr-def)

lemma pour-pour-swap: (x powr a) powr b = (x powr b) powr a
  for a b x :: real
  by (simp add: powr-mult.commute)

lemma pour-minus: x powr (− a) = inverse (x powr a)
  for a x :: 'a::{ln,real-normed-field}
  by (simp add: powr-def exp-minus [symmetric])

lemma pour-minus-divide: x powr (− a) = 1/(x powr a)
  for a x :: 'a::{ln,real-normed-field}
  by (simp add: divide-inverse powr-minus)

lemma divide-powr-uminus: a / b powr c = a * b powr (− c)
  for a b c :: real
  by (simp add: powr-minus-divide)

lemma pour-less-mono: a < b ⟹ 1 < x ⟹ x powr a < x powr b
  for a b x :: real
  by (simp add: powr-def)

lemma pour-less-cancel: x powr a < x powr b ⟹ 1 < x ⟹ a < b
  for a b x :: real
  by (simp add: powr-def)

lemma pour-less-cancel-iff [simp]: 1 < x ⟹ x powr a < x powr b ⟷ a < b
  for a b x :: real
  by (blast intro: pour-less-cancel pour-less-mono)

lemma pour-le-cancel-iff [simp]: 1 < x ⟹ x powr a ≤ x powr b ⟷ a ≤ b
  for a b x :: real
  by (simp add: linorder-not-less [symmetric])

lemma pour-realpow: 0 < x ⟹ x powr (real n) = x ^ n
  by (induction n) (simp-all add: ac-simps pour-add)

lemma log-ln: ln x = log (exp(1)) x
  by (simp add: log-def)

lemma DERIV-log:
  assumes x > 0
shows DERIV \((\lambda y. \log b y) x \mapsto 1 / (\ln b * x)\)

proof –
  define \(lb\) where \(lb = 1 / \ln b\)
  moreover have DERIV \((\lambda y. lb * \ln y) x \mapsto lb / x\)
    using \(x > 0\) by (auto intro: derivative-eq-intros)
  ultimately show \(\theta\)
    by (simp add: log-def)

qed

lemmas DERIV-log[THEN DERIV-chain2, derivative-intros]
  and DERIV-log[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]

lemma powr-log-cancel [simp]: \(0 < a \Longrightarrow a \neq 1 \Longrightarrow 0 < x \Longrightarrow a \operatorname{powr} (\log a x) = x\)
  by (simp add: powr-def log-def)

lemma log-powr-cancel [simp]: \(0 < a \Longrightarrow a \neq 1 \Longrightarrow \log a (a \operatorname{powr} y) = y\)
  by (simp add: log-def powr-def)

lemma log-mult:
  \(0 < a \Longrightarrow a \neq 1 \Longrightarrow 0 < x \Longrightarrow 0 < y \Longrightarrow\)
  \(\log a (x * y) = \log a x + \log a y\)
  by (simp add: log-def ln-mult divide-inverse distrib-right)

lemma log-eq-div-ln-mult-log:
  \(0 < a \Longrightarrow a \neq 1 \Longrightarrow 0 < b \Longrightarrow b \neq 1 \Longrightarrow 0 < x \Longrightarrow\)
  \(\log a x = (\ln b / \ln a) * \log b x\)
  by (simp add: log-def divide-inverse)

Base 10 logarithms

lemma log-base-10-eq1: \(0 < x \Longrightarrow \log 10 x = (\ln \exp 1 / \ln 10) * \ln x\)
  by (simp add: log-def)

lemma log-base-10-eq2: \(0 < x \Longrightarrow \log 10 x = (\log 10 \exp 1) * \ln x\)
  by (simp add: log-def)

lemma log-one [simp]: \(\log a 1 = 0\)
  by (simp add: log-def)

lemma log-eq-one [simp]: \(0 < a \Longrightarrow a \neq 1 \Longrightarrow \log a a = 1\)
  by (simp add: log-def)

lemma log-inverse: \(0 < a \Longrightarrow a \neq 1 \Longrightarrow 0 < x \Longrightarrow \log a (\operatorname{inverse} x) = - \log a x\)
  using \(\ln\)-inverse log-def by auto

lemma log-divide: \(0 < a \Longrightarrow a \neq 1 \Longrightarrow 0 < x \Longrightarrow 0 < y \Longrightarrow \log a (x/y) = \log a x - \log a y\)
  by (simp add: log-mult divide-inverse log-inverse)
lemma powr-gt-zero [simp]: \(0 < x \text{ powr } a \iff x \neq 0\) 
for \(a, x::\text{real}\)
by (simp add: powr-def)

lemma powr-nonneg-iff [simp]: \(a \text{ powr } x \leq 0 \iff a = 0\)
for \(a, x::\text{real}\)
by (meson not-less powr-gt-zero)

lemma log-add-eq-powr: \(0 < b \Longrightarrow b \neq 1 \Longrightarrow 0 < x \Longrightarrow \log b x + y = \log b (x * b \text{ powr } y)\)
and add-log-evq-powr: \(0 < b \Longrightarrow b \neq 1 \Longrightarrow 0 < x \Longrightarrow y + \log b x = \log b (b \text{ powr } y * x)\)
and log-minus-eq-powr: \(0 < b \Longrightarrow b \neq 1 \Longrightarrow 0 < x \Longrightarrow \log b x - y = \log b (x * b \text{ powr } -y)\)
and minus-log-evq-powr: \(0 < b \Longrightarrow b \neq 1 \Longrightarrow 0 < x \Longrightarrow y - \log b x = \log b (b \text{ powr } y / x)\)
by (simp-all add: log-mult log-divide)

lemma log-less-cancel-iff [simp]: \(1 < a \Longrightarrow 0 < x \Longrightarrow 0 < y \Longrightarrow \log a x < \log a y \iff x < y\)
by (metis less-eq-real-def less-trans not-le zero-less-one)

lemma log-inj:
assumes \(1 < b\)
shows inj-on (log b) {0 <..}
proof (rule inj-on1, simp)
fix \(x, y\)
assume pos: \(0 < x, 0 < y\) and \(*: \log b x = \log b y\)
show \(x = y\)
proof (cases rule: linorder-cases)
  assume \(x = y\)
  then show \(\text{thesis}\) by simp
next
  assume \(x < y\)
  then have \(\log b x < \log b y\)
  using log-less-cancel-iff [OF \(1 < b\)] pos by simp
  then show \(\text{thesis}\) using \(*\) by simp
next
  assume \(y < x\)
  then have \(\log b y < \log b x\)
  using log-less-cancel-iff [OF \(1 < b\)] pos by simp
  then show \(\text{thesis}\) using \(*\) by simp
qed
qed

lemma log-le-cancel-iff [simp]: \(1 < a \Longrightarrow 0 < x \Longrightarrow 0 < y \Longrightarrow \log a x \leq \log a y \iff x \leq y\)
by (simp add: linorder-not-less [symmetric])

lemma zero-less-log-cancel-iff[simp]: \( 1 < a \implies 0 < x \implies 0 < \log a x \leftrightarrow 1 < x \)
  using log-less-cancel-iff[of a 1 x] by simp

lemma zero-le-log-cancel-iff[simp]: \( 1 < a \implies 0 < x \implies 0 \leq \log a x \leftrightarrow 1 \leq x \)
  using log-le-cancel-iff[of a 1 x] by simp

lemma log-less-zero-cancel-iff[simp]: \( 1 < a \implies 0 < x \implies \log a x < 0 \leftrightarrow x < 1 \)
  using log-less-cancel-iff[of a x 1] by simp

lemma log-le-zero-cancel-iff[simp]: \( 1 < a \implies 0 < x \implies \log a x \leq 0 \leftrightarrow x \leq 1 \)
  using log-le-cancel-iff[of a x 1] by simp

lemma one-less-log-cancel-iff[simp]: \( 1 < a \implies 0 < x \implies 1 < \log a x \leftrightarrow a < x \)
  using log-less-cancel-iff[of a a x] by simp

lemma one-le-log-cancel-iff[simp]: \( 1 < a \implies 0 < x \implies 1 \leq \log a x \leftrightarrow a \leq x \)
  using log-le-cancel-iff[of a a x] by simp

lemma log-less-one-cancel-iff[simp]: \( 1 < a \implies 0 < x \implies \log a x < 1 \leftrightarrow x < a \)
  using log-less-cancel-iff[of a x a] by simp

lemma log-le-one-cancel-iff[simp]: \( 1 < a \implies 0 < x \implies \log a x \leq 1 \leftrightarrow x \leq a \)
  using log-le-cancel-iff[of a x a] by simp

lemma le-log-iff:
  fixes b x y :: real
  assumes \( 1 < b \cdot x > 0 \)
  shows \( y \leq \log b x \leftrightarrow b \powr y \leq x \)
  using assms
  by (metis less-irrefl less-trans pour-le-cancel-iff pour-log-cancel zero-less-one)

lemma less-log-iff:
  assumes \( 1 < b x > 0 \)
  shows \( y < \log b x \leftrightarrow b \powr y < x \)
  by (metis assms dual-order.strict-trans less-irrefl pour-less-cancel-iff
    pour-log-cancel zero-less-one)

lemma
  assumes \( 1 < b x > 0 \)
  shows log-less-iff: \( \log b x < y \leftrightarrow x < b \powr y \)
    and log-le-iff: \( \log b x \leq y \leftrightarrow x \leq b \powr y \)
  using le-log-iff[OF assms, of y] less-log-iff[OF assms, of y]
  by auto
lemmas powr-le-iff = le-log-iff[symmetric]
  and powr-less-iff = less-log-iff[symmetric]
  and less-powr-iff = log-less-iff[symmetric]
  and le-powr-iff = log-le-iff[symmetric]

lemma le-log-of-power:
  assumes b ^ n ≤ m 1 < b
  shows n ≤ log b m
proof –
  from assms have 0 < m by (metis less-trans zero-less-power less-le-trans zero-less-one)
  thus ?thesis using assms by (simp add: le-log-iff powr-realpow)
qed

lemma le-log2-of-power: 2 ^ n ≤ m ⇒ n ≤ log 2 m for m n :: nat
using le-log-of-power[of 2] by simp

lemma log-of-power-le: [ m ≤ b ^ n; b > 1; m > 0 ] ⇒ log b (real m) ≤ n
by (simp add: log-le-iff powr-realpow)

lemma log2-of-power-le: [ m ≤ 2 ^ n; m > 0 ] ⇒ log 2 m ≤ n for m n :: nat
using log-of-power-le[of - 2] by simp

lemma less-log-of-power:
  assumes b ^ n < m 1 < b
  shows n < log b m
proof –
  have 0 < m by (metis assms less-trans zero-less-power zero-less-one)
  thus ?thesis using assms by (simp add: less-log-iff powr-realpow)
qed

lemma less-log2-of-power: 2 ^ n < m ⇒ n < log 2 m for m n :: nat
using less-log-of-power[of 2] by simp

lemma gr-one-powr[simp]:
  fixes x y :: real shows [ x > 1; y > 0 ] ⇒ 1 < x powr y
by(simp add: less-powr-iff)

lemma log-pow-cancel [simp]:
  a > 0 ⇒ a ≠ 1 ⇒ log a (a ^ b) = b
by (simp add: ln-realpow log-def)

lemma floor-log-eq-power-iff: x > 0 ⇒ b > 1 ⇒ \lfloor log b x \rfloor = k ⇐ b powr k ≤
THEORY "Transcendental"

\[ x \land x < b \powr (k + 1) \]
by (auto simp: floor-eq-iff powr-le-iff less-powr-iff)

lemma floor-log-nat-eq-powr-iff: fixes b n k :: nat
shows \[ b \geq 2; k > 0 \implies \floor{(\log b (\real k))} = n \land b^n \leq k \land k < b^{(n+1)} \]

lemma floor-log-nat-eq-if: fixes b n k :: nat
assumes \[ b^n \leq k \land k < b^{(n+1)} \land b \geq 2 \]
shows \[ \floor{(\log b (\real k))} = n \]
proof -
  have \[ k \geq 1 \]
  using assms(1,3) one-le-power[of b n] by linarith
  with assms show \?thesis by (simp add: floor-log-nat-eq-powr-iff)
qed

lemma ceiling-log-eq-powr-iff: \[ x > 0; b > 1 \implies \lceil(\log b x)\rceil = \int n + 1 \iff b \powr k < x \land x \leq b \powr (k + 1) \]
by (auto simp: ceiling-eq-iff powr-less-iff le-powr-iff)

lemma ceiling-log-nat-eq-powr-iff: fixes b n k :: nat
shows \[ b \geq 2; k > 0 \implies \ceiling{(\log b (\real k))} = \int n + 1 \iff (b^n < k \land k \leq b^{(n+1)}) \]
using ceiling-log-eq-powr-iff

lemma ceiling-log-nat-eq-if: fixes b n k :: nat
assumes \[ b^n < k \land k \leq b^{(n+1)} \land b \geq 2 \]
shows \[ \ceiling{(\log b (\real k))} = \int n + 1 \]
proof -
  have \[ k \geq 1 \]
  using assms(1,3) one-le-power[of b n] by linarith
  with assms show \?thesis by (simp add: ceiling-log-nat-eq-powr-iff)
qed

lemma floor-log2-div2: fixes n :: nat assumes \[ n \geq 2 \]
shows \[ \floor{(\log 2 n)} = \floor{(\log 2 (\text{n div 2}))} + 1 \]
proof cases
  assume \[ \text{n=2} \]
  thus \?thesis by simp
next
  let \[ ?m = \text{n div 2} \]
  assume \[ \text{n\neq 2} \]
  hence \[ 1 \leq ?m \]
  using assms by arith
  then obtain \[ i \mid i : 2 ^ i \leq ?m \land ?m < 2 ^ (i + 1) \]
  using ex-power-ivl1[of 2 \?m] by auto
  have \[ 2 ^ (i+1) \leq 2 \* \?m \]
  using \[ i(1) \]
  by simp
also have \(2^*m \leq n\) by arith

finally have \(*: 2^-(i+1) \leq \ldots\ .\)

have \(n < 2^-(i+1)\) using \(i(2)\) by simp

from floor-log-nat-\textit{eq-if}\[OF \ this]\ floor-log-nat-\textit{eq-if}[OF i]

show \(?thesis by simp\)

qed

lemma ceiling-log2-div2: assumes \(n \geq 2\)

shows \(ceiling(\log 2 (real\ n)) = ceiling(\log 2 ((n-1) \ div 2 + 1)) + 1\)

proof cases

assume \(n=2\) thus \(?thesis by simp\)

next

let \(?m = (n-1) \ div 2 + 1\)

assume \(n\neq 2\)

hence \(2 \leq \ ?m\) using assms by arith

then obtain \(i\) where \(i: 2^i \leq \ ?m \leq 2^((i+1)+1)\)

using ex-power-\textit{ivl2}[of \ ?m]\ by auto

have \(n \leq 2^\ ?m\) by arith

also have \(2^*m \leq 2^*((i+1)+1)\) using \(i(2)\) by simp

finally have \(*: n \leq \ldots\ .\)

have \(2^((i+1)) < n\) using \(i(1)\) by (auto simp: less-Suc-\textit{eq-0-disj})

from ceiling-log-nat-\textit{eq-if}[OF this \(*]\ ceiling-log-nat-\textit{eq-if}[OF i]

show \(?thesis by simp\)

qed

lemma powr-real-of-int: \(x > 0 \Longrightarrow x \ powr\ real-of-int\ n = (\text{if \(n \geq 0\ then \ x \ ^*\ nat\ n\ else\ inverse\ (x \ ^*\ nat\ (-n)))}\)

using powr-\textit{realpow}[of \ x\ nat\ n]\ powr-\textit{realpow}[of \ x\ nat\ (-n)]

by (auto simp: field-simps powr-minus)

lemma powr-numeral [simp]: \(0 \leq x \Longrightarrow x \ powr\ (\text{numeral\ n::real}) = x \ ^*\ (\text{numeral\ n})\)

by (metis less-le power-zero-numeral powr-0 of-nat-numeral powr-\textit{realpow})

lemma powr-int: \(\text{assumes } x > 0\)

shows \(x \ powr\ i = (\text{if \(i \geq 0\ then \ x \ ^*\ nat\ i\ else\ 1 / x \ ^*\ nat\ (-i)))\)

proof (cases \(i < 0\))

case True

have \(r: x \ powr\ i = 1 / x \ powr\ (-i)\)

by (simp add: powr-minus field-simps)

show \(?thesis using \(i < 0\) \(x > 0\)\)

by (simp add: r field-simps powr-\textit{realpow}[symmetric])

next

case False

then show \(?thesis\)

by (simp add: assms powr-\textit{realpow}[symmetric])

qed
definition powr-real :: real ⇒ real ⇒ real
where [code-abbrev, simp]: powr-real = Transcendental.powr

lemma compute-powr-real [code]:
  powr-real b i = 
  (if b ≤ 0 then Code.abort (STR "powr-real with nonpositive base") (λ-. powr-real b i)
   else if ⌊i⌋ = i then (if 0 ≤ i then b ^ nat ⌊i⌋ else 1 / b ^ nat ⌊− i⌋)
   else Code.abort (STR "powr-real with non-integer exponent") (λ-. powr-real b i))
  for b i :: real
by (auto simp: powr-int)

lemma powr-one: 0 ≤ x ⇒ x powr 1 = x
for x :: real
using powr-realpow [of x 1] by simp

lemma powr-neg-one: 0 < x ⇒ x powr -1 = 1 / x
for x :: real
using powr-int [of x -1] by simp

lemma powr-neg-numeral: 0 < x ⇒ x powr -numeral n = 1 / x ^ numeral n
for x :: real
using powr-int [of x -numeral n] by simp

lemma root-powr-inverse: 0 < n ⇒ 0 < x ⇒ root n x = x powr (1/n)
by (rule real-root-pos-unique) (auto simp: powr-realpow[symmetric] powr-powr)

lemma ln-powr: x ≠ 0 ⇒ ln (x powr y) = y * ln x
for x :: real
by (simp add: powr-def)

lemma ln-root: n > 0 ⇒ b > 0 ⇒ ln (root n b) = ln b / n
by (simp add: root-powr-inverse ln-powr)

lemma ln-sqrt: 0 < x ⇒ ln (sqrt x) = ln x / 2
by (simp add: ln-powr ln-powr[symmetric] mult.commute)

lemma log-root: n > 0 ⇒ a > 0 ⇒ log b (root n a) = log b a / n
by (simp add: log-def ln-root)

lemma log-powr: x ≠ 0 ⇒ log b (x powr y) = y * log b x
by (simp add: log-def ln-powr)

lemma log-nat-power: 0 < x ⇒ log b (x ^ n) = real n * log b x
by (simp add: log-powr powr-realpow [symmetric])
THEORY "Transcendental"  

lemma log-of-power-eq:
assumes \( m = b \cdot n \) \( b > 1 \)
shows \( n = \log b (\text{real } m) \)
proof
  have \( n = \log b (b \cdot n) \) using assms(2) by (simp add: log-nat-power)
  also have \( \ldots = \log b m \) using assms by simp
finally show \( \text{thesis} \).
qed

lemma log2-of-power-eq: \( m = 2 \cdot n \Rightarrow n = \log 2 m \) for \( m, n :: \text{nat} \)
using log-of-power-eq[of - 2] by simp

lemma log-base-change: \( 0 < a \Rightarrow a \neq 1 \Rightarrow \log b x = \frac{\log a x}{\log a b} \)
by (simp add: log-def)

lemma log-base-pow: \( 0 < a \Rightarrow \log (a \cdot n) x = \log a x / n \)
by (simp add: log-def ln-realpow)

lemma log-base-powr: \( a \neq 0 \Rightarrow \log (\text{powr } b) x = \log a x / b \)
by (simp add: log-def ln-powr)

lemma log-base-root: \( n > 0 \Rightarrow b > 0 \Rightarrow \log (\text{root } n b) x = n \cdot (\log b x) \)
by (simp add: log-def ln-root)

lemma ln-bound: \( 0 < x \Rightarrow \ln x \leq x \) for \( x :: \text{real} \)
using ln-le-minus-one by force

lemma powr-mono:
fixes \( x :: \text{real} \)
assumes \( a \leq b \) and \( 1 \leq x \)
shows \( x \text{ powr } a \leq x \text{ powr } b \)
using assms less-eq-real-def by auto

lemma ge-one-powr-ge-zero: \( 1 \leq x \Rightarrow 0 \leq a \Rightarrow 1 \leq x \text{ powr } a \)
for \( x :: \text{real} \)
using powr-mono by fastforce

lemma powr-less-mono2: \( 0 < a \Rightarrow 0 \leq x \Rightarrow x < y \Rightarrow x \text{ powr } a < y \text{ powr } a \)
for \( x :: \text{real} \)
by (simp add: powr-def)

lemma powr-less-mono2-neg: \( a < 0 \Rightarrow 0 < x \Rightarrow x < y \Rightarrow y \text{ powr } a < x \text{ powr } a \)
for \( x :: \text{real} \)
by (simp add: powr-def)

lemma powr-mono2: \( x \text{ powr } a \leq y \text{ powr } a \) if \( 0 \leq a \) \( 0 \leq x \leq y \)
for \( x :: \text{real} \)
using less-eq-real-def powr-less-mono2 that by auto
lemma `powr-le1`: 0 ≤ a ⇒ 0 ≤ x ⇒ x ≤ 1 ⇒ x powr a ≤ 1
  for x :: real
  using `powr-mono2` by `fastforce`

lemma `powr-mono2`:
  fixes a x y :: real
  assumes a ≤ 0 x > 0 x ≤ y
  shows x powr a ≥ y powr a
  proof
    from assms have x powr a ≤ y powr a
      by intro `powr-mono2` simp-all
    with assms show thesis
      by (auto simp: `powr-minus` field-simps)
  qed

lemma `powr-mono-both`:
  fixes x :: real
  assumes 0 ≤ a a ≤ b 1 ≤ x x ≤ y
  shows x powr a ≤ y powr b
  by (meson assms order.trans `powr-mono` `powr-mono2` `zero-le-one`)

lemma `powr-inj`:
  0 < a ⇒ a ≠ 1 ⇒ a powr x = a powr y ←→ x = y
  for x :: real
  unfolding `powr-def` `exp-inj-iff` by simp

lemma `powr-half-sqrt`:
  0 ≤ x ⇒ x powr (1/2) = sqrt x
  by (simp add: `powr-def` `root-powr-inverse` `sqrt-def`)

lemma `square-powr-half` [simp]:
  fixes x :: real
  shows x powr (1/2) = |x|
  by (simp add: `powr-half-sqrt`)

lemma `ln-powr-bound`: 1 ≤ x ⇒ 0 < a ⇒ ln x ≤ (x powr a) / a
  for x :: real
  by (metis `exp-gt-zero` `linear` `ln-eq-zero-iff` `ln-exp` `ln-less-self` `ln-powr`
      `mult.commute` `mult-imp-le-div-pos` `not-less` `powr-gt-zero`)

lemma `ln-powr-bound2`:
  fixes x :: real
  assumes 1 < x and 0 < a
  shows (ln x) powr a ≤ (a powr a) * x
  proof
    from assms have ln x ≤ (x powr (1 / a)) / (1 / a)
      by (metis `less-eq-real-def` `ln-powr-bound` `zero-less-divide-1-iff`
          `ln-powr-bound` `powr-bound`
          `ln-powr-bound2` `powr-bound`
          `powr-bound`
          `powr-bound`
          `powr-bound`
          `powr-bound`)
    also have ... = a * (x powr (1 / a))
      by simp
    finally have (ln x) powr a ≤ (a * (x powr (1 / a))) powr a
      by (metis assms `less-imp-le` `ln-gt-zero` `powr-mono2`)
    also have ... = (a powr a) * ((x powr (1 / a)) powr a)
using assms powr-mult by auto
also have \((x \ powr (1/a)) \ powr a = x \ powr ((1/a) * a)\)
by (rule powr-powr)
also have \(\ldots = x\) using assms
by auto
finally show \(?thesis\).
qed

lemma tendsto-powr:
fixes \(a\) \(b\) :: real
assumes \(f\) \((f \longrightarrow a)\) \(F\)
and \(g\) \((g \longrightarrow b)\) \(F\)
and \(a\) : \(a \neq 0\)
shows \(((\lambda x. f x \ powr g x) \longrightarrow a \ powr b)\) \(F\)
unfolding powr-def
proof (rule filterlim-If)
from \(f\) show \(((\lambda x. 0) \longrightarrow (if a = 0 then 0 else exp (b * ln a)))\) \(\inf F\) \(\{x. f x = 0\}\)
by simp (auto simp: filterlim-iff eventually-inf-principal elim: eventually-mono dest: t1-space-nhds)
from \(f\) \(g\) \(a\) show \(((\lambda x. \exp (g x * \ln (f x))) \longrightarrow (if a = 0 then 0 else exp (b * ln a)))\)
\(\inf F\) \(\{x. f x \neq 0\}\)
by (auto intro!: tendsto-intros intro: tendsto-mono inf-le1)
qed

lemma tendsto-powr'[tendsto-intros]:
fixes \(a\) :: real
assumes \(f\) \((f \longrightarrow a)\) \(F\)
and \(g\) \((g \longrightarrow b)\) \(F\)
and \(a\) : \(a \neq 0 \lor (b > 0 \land \text{eventually} \ (\lambda x. f x \geq 0)\) \(F)\)
shows \(((\lambda x. f x \ powr g x) \longrightarrow a \ powr b)\) \(F\)
proof
from \(a\) consider \(a \neq 0 \mid a = 0\) \(b > 0\) \(\text{eventually} \ (\lambda x. f x \geq 0)\) \(F\)
by auto
then show \(?thesis\)
proof cases
  case 1
  with \(f\) \(g\) show \(?thesis\) by (rule tendsto-powr)
next
case 2
have \(((\lambda x. \text{if} f x = 0 \text{ then} 0 \text{ else} \exp (g x * \ln (f x))) \longrightarrow 0)\) \(F\)
proof (intro filterlim-If)
  have \(\text{filterlim} f \ \{0<..<\\} \ \{\text{inf} F\} \ \{z. f z \neq 0\}\)
  using \(\\text{eventually} \ (\lambda x. f x \geq 0)\) \(F)\)
  by (auto simp: filterlim-iff eventually-inf-principal eventually-principal elim: eventually-mono)
  moreover have \(\text{filterlim} f \ \{\\text{nhds} a\} \ \{\text{inf} F\} \ \{z. f z \neq 0\}\)
  by (rule tendsto-mono)\(\{\text{OF} - f\}\) simp-all
ultimately have \( f : \text{filterlim } f \ (\text{at-right } 0) \ (\inf F \ (\text{principal} \ \{x . \ f x \neq 0\})) \)
by (simp add: at-within-def filterlim-inf \((a = 0)\))

have \( g : (g \longrightarrow b) \ (\inf F \ (\text{principal} \ \{z . \ f z \neq 0\})) \)
by (rule tendsto-mono[OF - g]) simp-all

show \(((\lambda x. \exp (g x * \ln (f x))) \longrightarrow 0) \ (\inf F \ (\text{principal} \ \{x . \ f x \neq 0\}))\)
by (rule filterlim-compose[OF exp-at-bot] filterlim-tendsto-pos-mul-at-bot filterlim-compose[OF ln-at-0] f g \((b > 0)\))

qed simp-all with \((a = 0)\) show \(?thesis\)
by (simp add: powr-def)

qed

lemma continuous-powr:
assumes continuous \( F f \)
and continuous \( F g \)
and \( f \ (\text{Lim } F \ (\lambda x. \ x)) \neq 0 \)
shows continuous \( F \ (\lambda x. \ (f x) \text{ powr } (g x :: \text{real})) \)
using assms unfolding continuous-def by (rule tendsto-powr)

lemma continuous-at-within-powr[continuous-intros]:
fixes \( f g :: - \Rightarrow \text{real} \)
assumes continuous \((at a within s) f \)
and continuous \((at a within s) g \)
and \( f a \neq 0 \)
shows continuous \((at a within s) (\lambda x. \ (f x) \text{ powr } (g x)) \)
using assms unfolding continuous-within by (rule tendsto-powr)

lemma isCont-powr[continuous-intros, simp]:
fixes \( f g :: - \Rightarrow \text{real} \)
assumes isCont \( f a \) isCont \( g a \) \( f a \neq 0 \)
shows isCont \((\lambda x. \ (f x) \text{ powr } (g x)) \ a \)
using assms unfolding continuous-at by (rule tendsto-powr)

lemma continuous-on-powr[continuous-intros]:
fixes \( f g :: - \Rightarrow \text{real} \)
assumes continuous-on \( s f \) continuous-on \( s g \) and \( \forall x \in s. \ f x \neq 0 \)
shows continuous-on \( s \ (\lambda x. \ (f x) \text{ powr } (g x)) \)
using assms unfolding continuous-on-def by (fast intro: tendsto-powr)

lemma tendsto-powr2:
fixes \( a :: \text{real} \)
assumes \( f : (f \longrightarrow a) F \)
and \( g : (g \longrightarrow b) F \)
and \( \forall F x \in F. \ 0 \leq f x \)
and \( b : 0 < b \)
shows \(((\lambda x. \ f x \text{ powr } g x) \longrightarrow a \text{ powr } b) F \)
using tendsto-powr"[of \( f \ a \ F b \)]" assms by auto
lemma has-derivative-powr[derivative-intros]:
assumes g[derivative-intros]: (g has-derivative g') (at x within X)
and f[derivative-intros]: (f has-derivative f') (at x within X)
assumes pos: 0 < g x and x ∈ X
shows ((λx. g x powr f x::real) has-derivative (λh. (g x powr f x) * (f' h * ln (g x) + g' h * f x / g x))) (at x within X)
proof –
have ∀x in at x within X. g x > 0
  by (rule order-tendstoD[OF - pos])
then obtain d where d > 0 and pos': ∃x'. x' ∈ X ⇒ dist x' x < d ⇒ 0 < g x'
  using pos unfolding eventually-at by force
have ((λx. exp (f x * ln (g x))) has-derivative
  (λh. (g x powr f x) * (f' h * ln (g x) + g' h * f x / g x))) (at x within X)
  using pos
  by (auto intro!: derivative-eq-intros ext simp: has-field-derivative-def algebra-simps)
then show ?thesis
  by (rule has-derivative-transform-within[OF - (d > 0) (x ∈ X)]) (auto simp: powr-def dest: pos')
qed

lemma DERIV-powr:
fixes r :: real
assumes g: DERIV g x :> m
and pos: g x > 0
and f: DERIV f x :> r
shows DERIV (λx. g x powr f x) x :> (g x powr f x) * (r * ln (g x) + m * f x / g x)
  using assms
  by (auto intro!: derivative-eq-intros ext simp: has-field-derivative-def algebra-simps)

lemma DERIV-fun-powr:
fixes r :: real
assumes g: DERIV g x :> m
and pos: g x > 0
shows DERIV (λx. (g x) powr r) x :> r * (g x) powr (r − of-nat 1) * m
  using DERIV-powr[OF g pos DERIV-const, of r] pos
  by (simp add: powr-diff field-simps)

lemma has-real-derivative-powr:
assumes z > 0
shows ((λz. z powr r) has-real-derivative r * z powr (r − 1)) (at z)
proof (subst DERIV-cong-ev[OF refl - refl])
  from assms have eventually (λz. z ≠ 0) (nhds z)
    by (intro t1-space-nhds auto)
  then show eventually (λz. z powr r = exp (r * ln z)) (nhds z)
    unfolding powr-def by eventually-elim simp
  from assms show ((λz. exp (r * ln z)) has-real-derivative r * z powr (r − 1))
THEORY "Transcendental"

\[(at \, z)\]

by (auto intro!: derivative-eq-intros simp: powr-def field-simps exp-diff)

qed

declare has-real-derivative-powr[THEN DERIV_chain2, derivative-intros]

lemma tendsto-zero-powrI:
  assumes \((f \to (0::\text{real})) \land (g \to b) \land 0 \leq f \, x \lt b\)
  shows \((\lambda x. f \, x \, \text{powr} \, g \, x) \to 0\)
  using tendsto-powr2[OF assms] by simp

lemma continuous-on-powr':
  fixes \(f, g::\text{-} \Rightarrow \text{real}\)
  assumes \(\text{continuous-on } s \, f, \text{continuous-on } s \, g\) \land \(\forall \, x \in s. \, f \, x \geq 0 \land (f \, x = 0 \to g \, x > 0)\)
  shows \(\text{continuous-on } s \, (\lambda x. f \, x \, \text{powr} \, g \, x)\)
  unfolding continuous-on_def
  proof
    fix \(x\)
    assume \(x: \, x \in s\)
    from assms \(x\) show \((\lambda x. f \, x \, \text{powr} \, g \, x) \to f \, x \, \text{powr} \, g \, x\) \((\text{at } \, x \, \text{within } s)\)
    proof (cases \(f \, x = 0\))
      case True
      from assms(3) have eventually \((\lambda x. f \, x \geq 0) \land (f \, x = 0 \to g \, x > 0)\) \((\text{at } \, x \, \text{within } s)\)
      by (auto simp: at-within_def eventually-inf_principal)
      with True \(x\) assms show \(?thesis\)
      by (auto intro!: tendsto-zero-powrI[of \(g \, g \, x\)] simp: continuous-on_def)
      next
      case False
      with assms \(x\) show \(?thesis\)
      by (auto intro!: tendsto-powr'[simp: continuous-on_def])
    qed
  qed

lemma tendsto-exp-limit-at-right:
  \((\lambda y. \, (1 + x \, y) \, \text{powr} \, (1 / y)) \to \exp \, x\)
THEORY "Transcendental"

(at-right 0)
for x :: real
proof (cases x = 0)
case True
then show ?thesis by simp
next
case False
have ((λy. ln (1 + x * y) :: real) has-real-derivative 1 * x) (at 0)
by (auto intro!: derivative-eq-intros)
then have ((λy. ln (1 + x * y) / y) ----> x) (at 0)
by (auto simp: has-field-derivative-def field-has-derivative-at)
then have *: ((λy. exp (ln (1 + x * y) / y)) ----> exp x) (at 0)
by (rule tendsto-intros)
then show ?thesis proof
(rule filterlim-mono-eventually)
show eventually (λx. exp (ln (1 + x * x) / x) = (1 + x * x) powr (1 / x))
(at-right 0)
unfolding eventually-at-right[OF zero-less-one]
using False
by (intro exI[of - 1 / |x|]) (auto simp: field-simps powr-def abs-if add-nonneg-eq-0-iff)
qed (simp-all add: at-eq-sup-left-right)
qed

lemma tendsto-exp-limit-at-top: ((λy. (1 + x / y) powr y) ----> exp x) at-top
for x :: real
by (simp add: filterlim-at-top-to-right inverse-eq-divide tendsto-exp-limit-at-right)

lemma tendsto-exp-limit-sequentially: (λn. (1 + x / n) ^ n) ----> exp x
for x :: real
proof (rule filterlim-mono-eventually)
from reals-Archimedean2[of |x|] obtain n :: nat where *: real n > |x| ..
then have eventually (λn :: nat. 0 < 1 + x / real n) at-top
by (intro eventually-sequentiallyI[of n]) (auto simp: field-split-simps)
then show eventually (λn. (1 + x / n) powr n = (1 + x / n) ^ n) at-top
by (rule eventually-mono) (erule powr-realpow)
show (λn. (1 + x / real n) powr real n) ----> exp x
by (rule filterlim-compose [OF tendsto-exp-limit-at-top filterlim-real-sequentially])
qed auto

110.10  Sine and Cosine

definition sin-coeff :: nat ⇒ real
where sin-coeff = (λn. if even n then 0 else (- 1) ^ ((n - Suc 0) div 2) / (fact n))

definition cos-coeff :: nat ⇒ real
where cos-coeff = (λn. if even n then ((- 1) ^ (n div 2)) / (fact n) else 0)

definition sin :: 'a ⇒ 'a::{real-normed-algebra-1,banach}
where \( \sin = (\lambda x. \sum n. \sin\text{-coeff } n \ast_R x^\cdot n) \)

definition \( \cos :: 'a \Rightarrow 'a::\{real-normed-algebra-1, banach\} \)
\[ \cos = (\lambda x. \sum n. \cos\text{-coeff } n \ast_R x^\cdot n) \]

lemma \( \sin\text{-coeff} 0 \) [simp]: \( \sin\text{-coeff } 0 = 0 \)
unfolding \( \sin\text{-coeff-def} \) by simp

lemma \( \cos\text{-coeff} 0 \) [simp]: \( \cos\text{-coeff } 0 = 1 \)
unfolding \( \cos\text{-coeff-def} \) by simp

lemma \( \sin\text{-coeff-Suc} \) : \( \sin\text{-coeff } (\text{Suc } n) = \cos\text{-coeff } n / \text{real } (\text{Suc } n) \)
unfolding \( \cos\text{-coeff-def} \) \( \sin\text{-coeff-def} \)
by (simp del: mult-Suc)

lemma \( \cos\text{-coeff-Suc} \) : \( \cos\text{-coeff } (\text{Suc } n) = -\sin\text{-coeff } n / \text{real } (\text{Suc } n) \)
unfolding \( \cos\text{-coeff-def} \) \( \sin\text{-coeff-def} \)
by (simp del: mult-Suc) (auto elim: oddE)

lemma \( \text{summable-norm-sin} \) : \( \text{summable } (\lambda n. \text{norm } (\sin\text{-coeff } n \ast_R x^\cdot n)) \)
for \( x :: 'a::\{real-normed-algebra-1, banach\} \)
unfolding \( \sin\text{-coeff-def} \)
apply (rule summable-comparison-test [OF - summable-norm-exp [where \( x=x\)])
apply (auto simp: divide-inverse abs-mult power-abs [symmetric] zero-le-mult-iff)
done

lemma \( \text{summable-norm-cos} \) : \( \text{summable } (\lambda n. \text{norm } (\cos\text{-coeff } n \ast_R x^\cdot n)) \)
for \( x :: 'a::\{real-normed-algebra-1, banach\} \)
unfolding \( \cos\text{-coeff-def} \)
apply (rule summable-comparison-test [OF - summable-norm-exp [where \( x=x\)])
apply (auto simp: divide-inverse abs-mult power-abs [symmetric] zero-le-mult-iff)
done

lemma \( \sin\text{-converges} \) : \( (\lambda n. \sin\text{-coeff } n \ast_R x^\cdot n) \) sums \( \sin x \)
unfolding \( \sin\text{-def} \)
by (metis (full-types) summable-norm-cancel summable-norm-sin summable-sums)

lemma \( \cos\text{-converges} \) : \( (\lambda n. \cos\text{-coeff } n \ast_R x^\cdot n) \) sums \( \cos x \)
unfolding \( \cos\text{-def} \)
by (metis (full-types) summable-norm-cancel summable-norm-cos summable-sums)

lemma \( \sin\text{-of-real} \) : \( \sin \) (of-real \( x \)) = of-real \( \sin x \)
for \( x :: \text{real} \)
proof
have \( (\lambda n. \text{of-real } (\sin\text{-coeff } n \ast_R x^\cdot n)) = (\lambda n. \sin\text{-coeff } n \ast_R (\text{of-real } x)^\cdot n) \)
proof
show \( \text{of-real } (\sin\text{-coeff } n \ast_R x^\cdot n) = \sin\text{-coeff } n \ast_R \text{of-real } x^\cdot n \) for \( n \)
by (simp add: scaleR-conv-of-real)
qed
also have ... sums (\sin (\text{of-real } x))
  by (rule sin-converges)
finally have (\lambda n. \text{of-real } (\text{sin-coeff } n * R x^n)) sums (\sin (\text{of-real } x)).
then show \(\text{thesis}\)
  using sums-unique2 sums-of-real [OF sin-converges]
  by blast
qed

corollary sin-in-Reals [simp]: \(z \in \mathbb{R} \Rightarrow \sin z \in \mathbb{R}\)
  by (metis Reals-cases Reals-of-real sin-of-real)

lemma cos-of-real: \(\cos (\text{of-real } x) = \text{of-real } (\cos x)\)
  for \(x :: \text{real}\)
proof –
  have (\lambda n. \text{of-real } (\text{cos-coeff } n * R x^n)) = (\lambda n. \text{cos-coeff } n * R (\text{of-real } x) ^ n)
  proof
    show \(\text{of-real } (\text{cos-coeff } n * R x^n) = \text{cos-coeff } n * R \text{of-real } x^n\) for \(n\)
    by (simp add: scaleR-conv-of-real)
  qed
also have ... sums (\cos (\text{of-real } x))
  by (rule cos-converges)
finally have (\lambda n. \text{of-real } (\text{cos-coeff } n * R x^n)) sums (\cos (\text{of-real } x)).
then show \(\text{thesis}\)
  using sums-unique2 sums-of-real [OF cos-converges]
  by blast
qed

corollary cos-in-Reals [simp]: \(z \in \mathbb{R} \Rightarrow \cos z \in \mathbb{R}\)
  by (metis Reals-cases Reals-of-real cos-of-real)

lemma diffs-sin-coeff: \(\text{diffs } \sin-coeff = \cos-coeff\)
  by (simp add: diffs-def sin-coeff-Suc del: of-nat-Suc)

lemma diffs-cos-coeff: \(\text{diffs } \cos-coeff = (\lambda n. - \sin-coeff n)\)
  by (simp add: diffs-def cos-coeff-Suc del: of-nat-Suc)

lemma sin-int-times-real: \(\sin (\text{of-int } m * \text{of-real } x) = \text{of-real } (\sin (\text{of-int } m * x))\)
  by (metis sin-of-real of-real-mult of-real-of-int-eq)

lemma cos-int-times-real: \(\cos (\text{of-int } m * \text{of-real } x) = \text{of-real } (\cos (\text{of-int } m * x))\)
  by (metis cos-of-real of-real-mult of-real-of-int-eq)

Now at last we can get the derivatives of \(\exp, \sin\) and \(\cos\).

lemma DERIV-sin [simp]: \(\DERIV \sin x :> \cos x\)
  for \(x :: 'a::{\text{real-normed-field,banach}}\)
  unfolding sin-def cos-def scaleR-conv-of-real
  apply (rule DERIV-cong)
  apply (rule termdiffs [where \(K=\text{of-real } (\text{norm } x) + 1 :: 'a\)])
    apply (simp-all add: norm-less-p1 diffs-of-real diffs-sin-coeff diffs-cos-coeff)
summable-minus-iff scaleR-conv-of-real [symmetric]
summable-norm-sin [THEN summable-norm-cancel]
summable-norm-cos [THEN summable-norm-cancel])
done

declare DERIV-sin[THEN DERIV-chain2, derivative-intros]
and DERIV-sin[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]

lemmas has-derivative-sin[derivative-intros] = DERIV-sin[THEN DERIV-compose-FDERIV]

lemma DERIV-cos [simp]: DERIV cos x :- sin x
for x :: 'a::{real-normed-field,banach}
unfolding sin-def cos-def scaleR-conv-of-real
apply (rule DERIV-cong)
apply (rule termdiffs [where K=of-real (norm x) + 1 :: 'a])
apply (simp-all add: norm-less-p1 diffs-of-real diffs-minus suminf-minus
diffs-sin-coeff diffs-cos-coeff
summable-minus-iff scaleR-conv-of-real [symmetric]
summable-norm-sin [THEN summable-norm-cancel]
summable-norm-cos [THEN summable-norm-cancel])
done

declare DERIV-cos[THEN DERIV-chain2, derivative-intros]
and DERIV-cos[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]

lemmas has-derivative-cos[derivative-intros] = DERIV-cos[THEN DERIV-compose-FDERIV]

lemma isCont-sin: isCont sin x
for x :: 'a::{real-normed-field,banach}
by (rule DERIV-sin[THEN DERIV-isCont])

lemma continuous-on-sin-real: continuous-on {a..b} sin for a::real
using continuous-at-imp-continuous-on isCont-sin by blast

lemma isCont-cos: isCont cos x
for x :: 'a::{real-normed-field,banach}
by (rule DERIV-cos[THEN DERIV-isCont])

lemma continuous-on-cos-real: continuous-on {a..b} cos for a::real
using continuous-at-imp-continuous-on isCont-cos by blast

lemma isCont-sin' [simp]: isCont f a :- isCont (λx. sin (f x)) a
for f :: - :- 'a::{real-normed-field,banach}
by (rule isCont-o2 [OF - isCont-sin])

lemma isCont-cos' [simp]: isCont f a :- isCont (λx. cos (f x)) a
for f :: - :- 'a::{real-normed-field,banach}
by (rule isCont-o2 [OF - isCont-cos])

lemma tendsto-sin [tendsto-intros]: (f ----> a) F ==> ((λx. sin (f x)) ----> sin a) F
  for f :: - ==> 'a::{real-normed-field,banach}
  by (rule isCont-tendsto-compose [OF isCont-sin])

lemma tendsto-cos [tendsto-intros]: (f ----> a) F ==> ((λx. cos (f x)) ----> cos a) F
  for f :: - ==> 'a::{real-normed-field,banach}
  by (rule isCont-tendsto-compose [OF isCont-cos])

lemma continuous-sin [continuous-intros]: continuous F f ==> continuous F (λx. sin (f x))
  for f :: - ==> 'a::{real-normed-field,banach}
  unfolding continuous-def by (rule tendsto-sin)

lemma continuous-on-sin [continuous-intros]: continuous-on s f ==> continuous-on s (λx. sin (f x))
  for f :: - ==> 'a::{real-normed-field,banach}
  unfolding continuous-on-def by (auto intro: tendsto-sin)

lemma continuous-within-sin: continuous (at z within s) sin
  for z :: 'a::{real-normed-field,banach}
  by (simp add: continuous-within tendsto-sin)

lemma continuous-cos [continuous-intros]: continuous F f ==> continuous F (λx. cos (f x))
  for f :: - ==> 'a::{real-normed-field,banach}
  unfolding continuous-def by (rule tendsto-cos)

lemma continuous-on-cos [continuous-intros]: continuous-on s f ==> continuous-on s (λx. cos (f x))
  for f :: - ==> 'a::{real-normed-field,banach}
  unfolding continuous-on-def by (auto intro: tendsto-cos)

lemma continuous-within-cos: continuous (at z within s) cos
  for z :: 'a::{real-normed-field,banach}
  by (simp add: continuous-within tendsto-cos)

110.11 Properties of Sine and Cosine

lemma sin-zero [simp]: sin 0 = 0
  by (simp add: sin-def sin-coeff-def scaleR-conv-of-real)

lemma cos-zero [simp]: cos 0 = 1
  by (simp add: cos-def cos-coeff-def scaleR-conv-of-real)

lemma DERIV-fun-sin: DERIV g x ::> m ==> DERIV (λx. sin (g x)) x ::> cos
THEORY "Transcendental" 2177

\((gx) \ast m\)
by (auto intro!: derivative-intros)

lemma DERIV-fun-cos: \(\text{DERIV } g \ x \ : \ m \ \Longrightarrow \ \text{DERIV } (\lambda x. \cos(g \ x)) \ x \ : \ -\sin\)
\((gx) \ast m\)
by (auto intro!: derivative-eq-intros)

110.12 Deriving the Addition Formulas

The product of two cosine series.

lemma cos-x-cos-y:
fixes \(x::'a::{\text{real-normed-field,banach}}\)
shows
\((\lambda p. \sum n \leq p. \ 0)\)
proof –
have \((\text{cos-coeff } n \ast \text{cos-coeff } (p - n)) \ast_R (x^\cdot n \ast y^\cdot(p - n)) =\)
\((0)\)
if \(n \leq p \ \text{for} \ n \ p :: \text{nat}\)
proof –
from that have \(*: \text{even } n \ \Longrightarrow \ \text{even } p \ \Longrightarrow \ (-1) ^ \cdot (p \ \text{div} \ 2) \ast (-1) ^ \cdot ((p - n) \ \text{div} \ 2) = (-1 :: \text{real}) ^ \cdot (p \ \text{div} \ 2)\)
by (metis div-add power-add le-odd-diff-inverse odd-add)
with that show \(?\text{thesis}\)
by (auto simp: algebra-simps cos-coeff-def binomial-fact)
qed
then have \((\lambda p. \sum n \leq p. \ 0)\)
proof –
from that have \(*: \text{even } n \ \Longrightarrow \ \text{even } p \ \Longrightarrow \ (-1) ^ \cdot (p \ \text{div} \ 2) \ast (p \ \text{choose} \ n) / (\text{fact } p) \ast_R (x^\cdot n) \ast y^\cdot(p-n)\)
else \(0\)
proof –
also have \(...) = \((\lambda p. \sum n \leq p. \ (\text{cos-coeff } n \ast \text{cos-coeff } (p - n)) \ast_R (x^\cdot n) \ast (\text{cos-coeff } (p - n) \ast_R y^\cdot(p-n)))\)
by (simp add: algebra-simps)
also have \(...) \ast_R (x^\cdot n) \ast (\text{cos-coeff } (p - n) \ast_R y^\cdot(p-n))\)
by (auto simp: cos-def scaleR-cone-of-real intro!: Cauchy-product-sums)
finally show \(?\text{thesis}\).
qed

The product of two sine series.

lemma sin-x-sin-y:
fixes \(x::'a::{\text{real-normed-field,banach}}\)
shows
\[(\lambda p. \sum_{n \leq p.} \text{if even } p \land \text{odd } n \text{ then } -((-1) \cdot (p \text{ div } 2) \ast (p \text{ choose } n) / (\text{fact } p)) \ast_R (x^n) \ast y^{(p-n)} \text{ else } 0) \text{ sums } (\sin x \ast \sin y)\]

**proof** –

\[
(\sin-coeff n \ast \sin-coeff (p - n)) \ast_R (x^n \ast y^{(p-n)}) = \\
(\text{if even } p \land \text{odd } n \text{ then } -((-1) \cdot (p \text{ div } 2) \ast (p \text{ choose } n) / (\text{fact } p)) \ast_R (x^n) \ast y^{(p-n)} \text{ else } 0) \\
\text{if } n \leq p \text{ for } n \in \text{nats}
\]

**proof** –

\[
(\sin-coeff n \ast \sin-coeff (p - n)) \ast_R (x^n) \ast y^{(p-n)}
\]

**lemma sums-cos-x-plus-y:**

**fixes** \(x :: 'a::\{\text{real-normed-field,banach}\}\)

**shows**

\[(\lambda p. \sum_{n \leq p.} \text{if even } p \text{ then } 0 \text{ else sums } (\sin x \ast \sin y) \ast_R (x^n \ast y^{(p-n)}) \text{ for } n \in \text{nats})\]

**proof** –

\[
\text{if even } p \land \text{odd } n \text{ then } -((-1) \cdot (p \text{ div } 2) \ast (p \text{ choose } n) / (\text{fact } p)) \ast_R (x^n) \ast y^{(p-n)} \text{ else } 0
\]

**lemma sums-cos-x-plus-y:**

**fixes** \(x :: 'a::\{\text{real-normed-field,banach}\}\)

**shows**

\[(\lambda p. \sum_{n \leq p.} \text{if even } p \text{ then } 0 \text{ else sums } (\sin x \ast \sin y) \ast_R (x^n \ast y^{(p-n)}) \text{ for } n \in \text{nats})\]
then \((-1)^{\lfloor \frac{p}{2} \rfloor} \binom{p}{n} R(x^n) y^{(p-n)} \)
else 0

sums \(\cos (x + y)\)

**proof**

- **have**

  \[(\sum_{n \leq p} \text{if even } p \text{ then } (-1)^{\lfloor \frac{p}{2} \rfloor} \binom{p}{n} R(x^n) y^{(p-n)} \text{ else 0} = \cos-coeff p \times ((x + y)^p) R(x^n) y^{(p-n)} \]

- **for** \(p : \text{nat}\)

**proof**

- **have**

  \[(\sum_{n \leq p} \text{if even } p \text{ then } (-1)^{\lfloor \frac{p}{2} \rfloor} \binom{p}{n} R(x^n) y^{(p-n)} \text{ else 0} = \cos-coeff p \times ((x + y)^p) R(x^n) y^{(p-n)} \]

- **by** `simp`

- **also have** ...

- **by** `auto simp: sum-distrib-left field-simps scaleR-conv-of-real nonzero-of-real-divide`

- **also have** ...

- **by** `simp add: cos-coeff-def binomial-ring [of x y] scaleR-conv-of-real atLeast0AtMost`

- **finally show** ?thesis.

**qed**

**then have**

\[(\lambda p. \sum_{n \leq p} \text{if even } p \text{ then } (-1)^{\lfloor \frac{p}{2} \rfloor} \binom{p}{n} R(x^n) y^{(p-n)} \text{ else 0} = \cos-coeff p \times ((x + y)^p) R(x^n) y^{(p-n)}) \]

- **by** `simp`

- **also have** ...

- **by** `rule cos-converges`

- **finally show** ?thesis.

**qed**

**theorem** `cos-add`:

**fixes** \(x : 'a::\{real-normed-field,banach\}\)

**shows** \(\cos (x + y) = \cos x \times \cos y - \sin x \times \sin y\)

**proof**

- **have**

  (if even \(p \wedge \text{even } n\)
  \then \((-1)^{\lfloor \frac{p}{2} \rfloor} \binom{p}{n} R(x^n) y^{(p-n)} \)
  else 0) =

  (if even \(p \wedge \text{odd } n\)
  \then \((-1)^{\lfloor \frac{p}{2} \rfloor} \binom{p}{n} R(x^n) y^{(p-n)} \)
  else 0) =

  (if even \(p\)
  \then \((-1)^{\lfloor \frac{p}{2} \rfloor} \binom{p}{n} R(x^n) y^{(p-n)} \)
  else 0) =
THEORY "Transcendental"

else 0)
if n ≤ p for n p :: nat
  by simp
then have
  (λp. ∑ n≤p. (if even p then (−1) ^ (p div 2) * (p choose n) / (fact p)) *R
  (x^n) * y ^ (p−n) else 0))
  using sums-diff [OF cos-x-cos-y [of x y] sin-x-sin-y [of x y]]
  by (simp add: sum-subtractf [symmetric])
then show ?thesis
  by (blast intro: sums-cos-x-plus-y sums-unique2)
qed

lemma sin-minus-converges: (λn. − (sin-coeff n *R (−x) ^ n)) sums sin x
proof –
  have [simp]: ∀n. − (sin-coeff n *R (−x) ^ n) = (sin-coeff n *R x ^ n)
    by (auto simp: sin-coeff-def elim!: oddE)
  show ?thesis
    by (simp add: sin-def summable-norm-sin [THEN summable-norm-cancel,
      THEN summable-sums])
qed

lemma cos-minus-converges: (λn. (cos-coeff n *R (−x) ^ n)) sums cos x
proof –
  have [simp]: ∀n. (cos-coeff n *R (−x) ^ n) = (cos-coeff n *R x ^ n)
    by (auto simp: Transcendental.cos-coeff-def elim!: evenE)
  show ?thesis
    by (simp add: cos-def summable-norm-cos [THEN summable-norm-cancel,
      THEN summable-sums])
qed

lemma cos-minus [simp]: cos (−x) = cos x
for x :: 'a::{real-normed-algebra-1,banach}
using cos-minus-converges [of x]
by (auto simp: cos-def summable-norm-cos [THEN summable-norm-cancel,
      equation-minus-iff])

lemma sin-cos-squared-add [simp]: (sin x)^2 + (cos x)^2 = 1
for x :: 'a::{real-normed-field,banach}
using cos-add [of x − x]
by (simp add: power2-eq-square algebra-simps)

lemma sin-cos-squared-add2 [simp]: (cos x)^2 + (sin x)^2 = 1
THEORY "Transcendental"

for $x :: 'a::\{\text{real-normed-field, banach}\}$
by (subst add.commute, rule sin-cos-squared-add)

lemma sin-cos-squared-add3 [simp]: $\cos x * \cos x + \sin x * \sin x = 1$
for $x :: 'a::\{\text{real-normed-field, banach}\}$
using sin-cos-squared-add2 [unfolded power2-eq-square] .

lemma sin-squared-eq: $(\sin x)^2 = 1 - (\cos x)^2$
for $x :: 'a::\{\text{real-normed-field, banach}\}$
unfolding eq-diff-eq by (rule sin-cos-squared-add)

lemma cos-squared-eq: $(\cos x)^2 = 1 - (\sin x)^2$
for $x :: 'a::\{\text{real-normed-field, banach}\}$
unfolding eq-diff-eq by (rule sin-cos-squared-add2)

lemma abs-sin-le-one [simp]: $|\sin x| \leq 1$
for $x :: \text{real}$
by (rule power2-le-imp-le) (simp-all add: sin-squared-eq)

lemma sin-ge-minus-one [simp]: $-1 \leq \sin x$
for $x :: \text{real}$
using abs-sin-le-one [of $x$] by (simp add: abs-le-iff)

lemma sin-le-one [simp]: $\sin x \leq 1$
for $x :: \text{real}$
using abs-sin-le-one [of $x$] by (simp add: abs-le-iff)

lemma abs-cos-le-one [simp]: $|\cos x| \leq 1$
for $x :: \text{real}$
by (rule power2-le-imp-le) (simp-all add: cos-squared-eq)

lemma cos-ge-minus-one [simp]: $-1 \leq \cos x$
for $x :: \text{real}$
using abs-cos-le-one [of $x$] by (simp add: abs-le-iff)

lemma cos-le-one [simp]: $\cos x \leq 1$
for $x :: \text{real}$
using abs-cos-le-one [of $x$] by (simp add: abs-le-iff)

lemma cos-diff: $\cos (x - y) = \cos x * \cos y + \sin x * \sin y$
for $x :: 'a::\{\text{real-normed-field, banach}\}$
using cos-add [of $x - y$] by simp

lemma cos-double: $\cos(2*x) = (\cos x)^2 - (\sin x)^2$
for $x :: 'a::\{\text{real-normed-field, banach}\}$
using cos-add [where $x=x$ and $y=x$] by (simp add: power2-eq-square)

lemma sin-cos-le1: $|\sin x * \sin y + \cos x * \cos y| \leq 1$
for $x :: \text{real}$
using cos-diff [of x y] by (metis abs-cos-le-one add.commute)

lemma DERIV-fun-pow: DERIV g x :: m ===> DERIV (λx. (g x) ^ n) x :: real
  n * (g x) ^ (n - 1) * m
  by (auto intro!: derivative-eq-intros simp)

lemma DERIV-fun-exp: DERIV g x :: m ===> DERIV (λx. exp (g x)) x :: exp
  (g x) * m
  by (auto intro!: derivative-intros)

110.13 The Constant Pi

definition pi :: real
  where pi = 2 * (THE x. 0 ≤ x ∧ x ≤ 2 ∧ cos x = 0)

Show that there’s a least positive x with cos x = (0::'a); hence define pi.

lemma sin-paired: (λn. (- 1) ^ n / (fact (2 * n + 1)) * x ^ (2 * n + 1)) sums sin x
  for x :: real
  proof -
    have (λn. ∑ k = n*2..<n*2+2. sin-coeff k * x ^ k) sums sin x
      by (rule sums-group) (use sin-converges [of x, unfolded scaleR-conv-of-real] in auto)
    then show ?thesis
      by (simp add: sin-coeff-def ac-simps)
  qed

lemma sin-gt-zero-02:
  fixes x :: real
  assumes 0 < x and x < 2
  shows 0 < sin x
  proof -
    let ?f = λn::nat. ∑ k = n*2..<n*2+2. (- 1) ^ k / (fact (2*k+1)) * x ^ (2*k+1)
    have pos: ∀ n. 0 < ?f n
      proof
        fix n :: nat
        let ?k2 = real (Suc (Suc (4 * n)))
        let ?k3 = real (Suc (Suc (Suc (Suc (4 * n)))))
        have x * x < ?k2 * ?k3
          using assms by (intro mult-strict-mono', simp-all)
        then have x * x * x * x ^ (n * 4) < ?k2 * ?k3 * x * x ^ (n * 4)
          by (intro mult-strict-right-mono zero-less-power :0 < x)
        then show 0 < ?f n
          by (simp add: ac-simps divide-less-eq)
      qed
    have sums: ?f sums sin x
      by (rule sin-paired [THEN sums-group]) simp
    show 0 < sin x
      unfolding sums-unique [OF sums]
using sums-summable [OF sums] pos
by (rule suminf-pos)
qed

lemma cos-double-less-one: \( 0 < x \implies x < 2 \implies \cos(2 \cdot x) < 1 \)
for \( x :: \text{real} \)
using sin-gt-zero-02 [where \( x = x \)] by (auto simp: cos-squared-eq cos-double)

lemma cos-paired: \( \lambda n. (-1) \cdot n / (\text{fact}(2 \cdot n)) \cdot (2 \cdot n) \) sums \( \cos x \)
for \( x :: \text{real} \)
proof
have \( \lambda n. \sum k = n * 2..<n * 2 + 2 \cdot \cos\text{-coeff } k \cdot x \cdot k \) sums \( \cos x \)
by (rule sums-group) (use cos-converges [of \( x \)], unfolded scaleR-conv-of-real in auto)
then show \(?thesis\)
by (simp add: cos-coeff-def ac-simps)
qed

lemma sum-pos-lt-pair:
fixes \( f :: \text{nat} \Rightarrow \text{real} \)
assumes \( f :: \text{summable} f \) and \( fplus \)
shows \( \sum f \{..<k\} < \text{suminf } f \)
proof
have \( \lambda n. \sum n = n * \text{Suc } (\text{Suc } 0) ..<n * \text{Suc } (\text{Suc } 0) + \text{Suc } (\text{Suc } 0). f (n + k) \)
sums \( \sum n. f (n + k) \)
proof (rule sums-group)
show \( \lambda n. f (n + k) \) sums \( \sum n. f (n + k) \)
by (simp add: f summable-iff-shift summable-sums)
qed auto
with \( fplus \) have \( 0 < (\sum n. f (n + k)) \)
apply (simp add: add.commute)
apply (metis (no-types, lifting) suminf-pos summable-def sums-unique)
done
then show \(?thesis\)
by (simp add: f suminf-minus-initial-segment)
qed

lemma cos-two-less-zero [simp]: \( \cos 2 < (0::\text{real}) \)
proof
note fact-Suc [simp del]
from sums-minus [OF cos-paired]
have \( \lambda n. -((-1) \cdot n * 2 \cdot (2 \cdot n) / (\text{fact}(2 \cdot n))) \) sums \( -\cos(2::\text{real}) \)
by simp
then have sm: summable \( \lambda n. -((-1::\text{real}) \cdot n * 2 \cdot (2 \cdot n) / (\text{fact}(2 \cdot n))) \)
by (rule sums-summable)
have \( 0 < (\sum n<\text{Suc } (\text{Suc } 0)). -((-1::\text{real}) \cdot n * 2 \cdot (2 \cdot n) / (\text{fact}(2 \cdot n))) \)
by (rule sums-summable)
THEORY "Transcendental"

2184

proof

qed

ultimately have \( 0 < (\sum n. - ((- 1 :: real) * n * 2 ^ (2 * n) / (\text{fact} (2 * n)))) \)

by (rule order-less-trans)

moreover from \( \exists x :: real. \quad 0 < x \land x \leq 2 \land \cos x = 0 \)

have \( \forall x :: real. \quad 0 \leq x \land x \leq 2 \land \cos x = 0 \)

proof (rule ex-ex1I)

show \( \forall x :: real. \quad 0 \leq x \land x \leq 2 \land \cos x = 0 \)

by (rule IVT2) simp-all

next

fix a b :: real

assume \( a \\ b : real. \quad 0 \leq a \land a \leq 2 \land \cos a = 0 \land 0 \leq b \land b \leq 2 \land \cos b = 0 \)

have cosd: \( \forall x :: real. \quad \cos \text{ differentiable} \quad (\text{at} x) \)

unfolding \( \text{real-differentiable-def} \) by (auto intro: DERIV-cos)

show \( a = b \)

proof (cases a b rule: linorder-cases)

case less

then obtain \( z \quad \text{where} \quad a < z < b \quad \text{(\( \cos \text{ has-real-derivative} \quad 0 \)) \quad (\text{at} z) \quad} \)

using Rolle by (metis cosd continuous-on-cos-reald)

then have \( \sin z = 0 \)

using DERIV-cos DERIV-unique neg-equal-0-iff-equal by blast

then show \( \text{?thesis} \)
by (metis ⟨a < z⟩ ⟨z < b⟩ ab order-less-le-trans less-le sin-gt-zero-02)
next
  case greater
  then obtain z where b < z z < a (cos has-real-derivative 0) (at z)
    using Rolle by (metis cosd continuous-on-cos-real ab)
  then have sin z = 0
    using DERIV-cos DERIV-unique neg-equal-0-iff-equal by blast
  then show ?thesis
    by (metis ⟨b < z⟩ ⟨z < a⟩ ab order-less-le-trans less-le sin-gt-zero-02)
qed auto

proof −
  have 0 ≤ pi/2
    by (simp add: pi-half cos-is-zero [THEN theI'])
  then show ?thesis
    by (metis cos-pi-half cos-zero less-eq-real-def one-neq-zero)
qed

lemmas pi-half-neq-zero [simp] = pi-half-gt-zero [THEN less-imp-neq, symmetric]
lemmas pi-half-ge-zero [simp] = pi-half-gt-zero [THEN order-less-imp-le]

proof −
  have pi/2 ≤ 2
    by (simp add: pi-half cos-is-zero [THEN theI'])
  then show ?thesis
    by (metis cos-pi-half cos-two-neq-zero le-less)
qed

lemmas pi-half-neq-two [simp] = pi-half-less-two [THEN less-imp-neq]
lemmas pi-half-le-two [simp] = pi-half-less-two [THEN order-less-imp-le]

proof −
  have 0 < pi
    using pi-gt-zero [simp] by simp

proof −
  have pi/2 < 2
    by (simp add: pi-half cos-is-zero [THEN theI'])
  then show ?thesis
    by (metis cos-pi-half cos-two-neq-zero le-less)
qed
by (rule pi-gt-zero [THEN order-less-imp-le])

lemma pi-neq-zero [simp]: \(\pi \neq 0\)
  by (rule pi-gt-zero [THEN less-imp-neq, symmetric])

lemma pi-not-less-zero [simp]: \(\neg \pi < 0\)
  by (simp add: linorder-not-less)

lemma minus-pi-half-less-zero:
  \(-\left(\frac{\pi}{2}\right) < 0\)
  by simp

lemma m2pi-less-pi:
  \(-2\pi < \pi\)
  by simp

lemma sin-pi-half [simp]: \(\sin\left(\frac{\pi}{2}\right) = 1\)
  using sin-cos-squared-add2 [where \(x = \frac{\pi}{2}\)]
  using sin-gt-zero-02 [OF pi-half-gt-zero pi-half-less-two]
  by (simp add: power2-eq-1-iff)

lemma sin-of-real-pi-half [simp]: \(\sin\left(\frac{\text{of-real} \pi}{2}\right) = 1\)
  if SORT-CONSTRAINT (\(\text{'a}::\{\text{real-field,banach,real-normed-algebra-1}\}\))
  using sin-pi-half
  by (metis sin-pi-half eq-numeral-simps (4) nonzero-of-real-divide of-real-1 of-real-numeral sin-of-real)

lemma sin-cos-eq: \(\sin x = \cos\left(\frac{\pi}{2} - x\right)\)
  for \(x::\{'a::\{\text{real-normed-field,banach}\}\}\)
  by (simp add: cos-diff)

lemma minus-sin-cos-eq: \(-\sin x = \cos\left(x + \frac{\pi}{2}\right)\)
  for \(x::\{'a::\{\text{real-normed-field,banach}\}\}\)
  by (simp add: cos-add nonzero-of-real-divide)

lemma cos-sin-eq: \(\cos x = \sin\left(\frac{\pi}{2} - x\right)\)
  for \(x::\{'a::\{\text{real-normed-field,banach}\}\}\)
  using sin-cos-eq [of of-real pi/2 - x] by simp

lemma sin-add: \(\sin (x + y) = \sin x * \cos y + \cos x * \sin y\)
  for \(x::\{'a::\{\text{real-normed-field,banach}\}\}\)
  using cos-add [of of-real pi/2 - x - y]
  by (simp add: cos-sin-eq) (simp add: sin-cos-eq)

lemma sin-diff: \(\sin (x - y) = \sin x * \cos y - \cos x * \sin y\)
  for \(x::\{'a::\{\text{real-normed-field,banach}\}\}\)
  using sin-add [of x - y] by simp

lemma sin-double: \(\sin(2 * x) = 2 * \sin x * \cos x\)
  for \(x::\{'a::\{\text{real-normed-field,banach}\}\}\)
  using sin-add [where \(x\text{=}x\) and \(y\text{=}x\)] by simp
lemma cos-of-real-pi [simp]: cos (of-real pi) = −1
using cos-add [where x = pi/2 and y = pi/2] by (simp add: cos-of-real)

lemma sin-of-real-pi [simp]: sin (of-real pi) = 0
using sin-add [where x = pi/2 and y = pi/2] by (simp add: sin-of-real)

lemma cos-pi [simp]: cos pi = −1
using cos-add [where x = pi/2 and y = pi/2] by simp

lemma sin-pi [simp]: sin pi = 0
using sin-add [where x = pi/2 and y = pi/2] by simp

lemma sin-periodic-pi [simp]: sin (x + pi) = − sin x
by (simp add: sin-add)

lemma sin-periodic-pi2 [simp]: sin (pi + x) = − sin x
by (simp add: sin-add)

lemma cos-periodic-pi [simp]: cos (x + pi) = − cos x
by (simp add: cos-add)

lemma cos-periodic-pi2 [simp]: cos (pi + x) = − cos x
by (simp add: cos-add)

lemma sin-periodic [simp]: sin (x + 2 * pi) = sin x
by (simp add: sin-add sin-double cos-double)

lemma cos-periodic [simp]: cos (x + 2 * pi) = cos x
by (simp add: cos-add sin-double cos-double)

lemma cos-npi [simp]: cos (real n * pi) = (−1) ^ n
by (induct n) (auto simp: distrib-right)

lemma cos-npi2 [simp]: cos (pi * real n) = (−1) ^ n
by (metis cos-npi mult.commute)

lemma sin-npi [simp]: sin (real n * pi) = 0
for n :: nat
by (induct n) (auto simp: distrib-right)

lemma sin-npi2 [simp]: sin (pi * real n) = 0
for n :: nat
by (simp add: mult.commute [of pi])

lemma cos-two-pi [simp]: cos (2 * pi) = 1
by (simp add: cos-double)
lemma sin-two-pi [simp]: sin (2 * pi) = 0
by (simp add: sin-double)

lemma sin-times-sin: sin w * sin z = (cos (w - z) - cos (w + z)) / 2
for w :: 'a::{real-normed-field, banach}
by (simp add: cos-diff cos-add)

lemma sin-times-cos: sin w * cos z = (sin (w + z) + sin (w - z)) / 2
for w :: 'a::{real-normed-field, banach}
by (simp add: sin-diff sin-add)

lemma cos-times-sin: cos w * sin z = (sin (w + z) - sin (w - z)) / 2
for w :: 'a::{real-normed-field, banach}
by (simp add: sin-diff sin-add)

lemma cos-times-cos: cos w * cos z = (cos (w - z) + cos (w + z)) / 2
for w :: 'a::{real-normed-field, banach, field}
apply (simp add: mult.assoc cos-times-cos)
apply (simp add: field-simps)
done

lemma sin-plus-sin: sin w + sin z = 2 * sin ((w + z) / 2) * cos ((w - z) / 2)
for w :: 'a::{real-normed-field, banach}
apply (simp add: mult.assoc sin-times-cos)
apply (simp add: field-simps)
done

lemma sin-diff-sin: sin w - sin z = 2 * sin ((w - z) / 2) * cos ((w + z) / 2)
for w :: 'a::{real-normed-field, banach}
apply (simp add: mult.assoc sin-times-cos)
apply (simp add: field-simps)
done

lemma cos-plus-cos: cos w + cos z = 2 * cos ((w + z) / 2) * cos ((w - z) / 2)
for w :: 'a::{real-normed-field, banach, field}
apply (simp add: mult.assoc cos-times-cos)
apply (simp add: field-simps)
done

lemma cos-diff-cos: cos w - cos z = 2 * sin ((w + z) / 2) * sin ((z - w) / 2)
for w :: 'a::{real-normed-field, banach, field}
apply (simp add: mult.assoc sin-times-sin)
apply (simp add: field-simps)
done

lemma cos-double-cos: cos (2 * z) = 2 * cos z ^ 2 - 1
for z :: 'a::{real-normed-field, banach}
by (simp add: cos-double sin-squared-eq)

lemma cos-double-sin: cos (2 * z) = 1 - 2 * sin z ^ 2
for z :: 'a::{real-normed-field,banach}
by (simp add: cos-double sin-squared-eq)

lemma sin-pi-minus [simp]: \( \sin (\pi - x) = \sin x \)
by (metis sin-minus sin-periodic-pi minus-minus uminus-add-conv-diff)

lemma cos-pi-minus [simp]: \( \cos (\pi - x) = - (\cos x) \)
by (metis cos-minus cos-periodic-pi uminus-add-conv-diff)

lemma sin-minus-pi [simp]: \( \sin (x - \pi) = - (\sin x) \)
by (simp add: sin-diff)

lemma cos-minus-pi [simp]: \( \cos (x - \pi) = - (\cos x) \)
by (simp add: cos-diff)

lemma sin-2pi-minus [simp]: \( \sin (2\pi - x) = - (\sin x) \)
by (metis sin-periodic-pi2 add-diff-eq mult-2 sin-pi-minus)

lemma cos-2pi-minus [simp]: \( \cos (2\pi - x) = \cos x \)
by (metis no-types hide-lams cos-add cos-minus cos-two-pi sin-minus sin-two-pi
diff-0-right minus-diff-eq mult-1 mult-zero-left uminus-add-conv-diff)

lemma sin-gt-zero2: \( 0 < x \implies x < \pi/2 \implies 0 < \sin x \)
by (metis sin-gt-zero-02 order-less-trans pi-half-less-two)

lemma sin-less-zero: assumes \(-\pi/2 < x \land x < 0\) shows \( \sin x < 0 \)
proof -
  have \( 0 < \sin (-x) \)
  using assms by (simp only: sin-gt-zero2)
  then show \(?thesis \) by simp
qed

lemma pi-less-4: \( \pi < 4 \)
using pi-half-less-two by auto

lemma cos-gt-zero: \( 0 < x \implies x < \pi/2 \implies 0 < \cos x \)
by (simp add: cos-sin-eq sin-gt-zero2)

lemma cos-gt-zero-pi: \(-\pi/2 < x \implies x < \pi/2 \implies 0 < \cos x \)
using cos-gt-zero [of \( x \)] cos-gt-zero [of \(-x\)]
by (cases rule: linorder-cases [of \( x \)] \( 0 \))\) auto

lemma cos-ge-zero: \( -\pi/2 \leq x \implies x \leq \pi/2 \implies 0 \leq \cos x \)
by (auto simp: order-le-less cos-gt-zero-pi
  (metis cos-pi-half eq-divide-eq eq-numeral-simps(4)))

lemma sin-gt-zero: \( 0 < x \implies x < \pi \implies 0 < \sin x \)
by (simp add: sin-cos-eq cos-gt-zero-pi)

lemma sin-lt-zero: \( \pi < x \Rightarrow x < 2 \pi \Rightarrow \sin x < 0 \)
  using sin-gt-zero [of \( x - \pi \)]
by (simp add: sin-diff)

lemma pi-ge-two: \( 2 \leq \pi \)
proof (rule ccontr)
  assume \( \neg \, \text{thesis} \)
  then have \( \pi < 2 \) by auto
  have \( \exists y. y > \pi \land y < 2 \pi \)
  proof (cases \( 2 < 2 \pi \))
    case True
    with dense [OF \( \pi < 2 \)] show \( \text{thesis} \) by auto
  next
    case False
    have \( \pi < 2 \pi \) by auto
    from dense [OF this and False] show \( \text{thesis} \) by auto
  qed
then obtain \( y \) where \( \pi < y \) and \( y < 2 \) and \( y < 2 \pi \)
  by blast
then have \( 0 < \sin y \)
  using sin-gt-zero-02 by auto
moreover have \( \sin y < 0 \)
  using sin-gt-zero [of \( y - \pi \)] \( \pi < y \) and \( y < 2 \pi \)
  sin-periodic-pi [of \( y - \pi \)]
  by auto
ultimately show \( \text{False} \) by auto
qed

lemma sin-ge-zero: \( 0 \leq x \Rightarrow x \leq \pi \Rightarrow 0 \leq \sin x \)
by (auto simp: order-le-less sin-gt-zero)

lemma sin-le-zero: \( \pi \leq x \Rightarrow x < 2 \pi \Rightarrow \sin x \leq 0 \)
using sin-ge-zero [of \( x - \pi \)] by (simp add: sin-diff)

lemma sin-pi-divide-n-ge-0 [simp]:
  assumes \( n \neq 0 \)
  shows \( 0 \leq \sin (\pi / \text{real } n) \)
  by (rule sin-ge-zero) (use assms in (simp-all add: field-split-simps))

lemma sin-pi-divide-n-gt-0:
  assumes \( 2 \leq n \)
  shows \( 0 < \sin (\pi / \text{real } n) \)
  by (rule sin-gt-zero) (use assms in (simp-all add: field-split-simps))

Proof resembles that of cos-is-zero but with \( \pi \) for the upper bound

lemma cos-total:
  assumes \( -1 \leq y \leq 1 \)
THEORY “Transcendental”

shows \( \exists x. \ 0 \leq x \wedge x \leq \pi \wedge \cos x = y \)

proof (rule ex-ex1I)
  show \( \exists x::\mathtt{real}. \ 0 \leq x \wedge x \leq \pi \wedge \cos x = y \)
    by (rule IVT2) (simp-all add: y)

next

fix a b :: \mathtt{real}
assume ab: \( 0 \leq a \wedge a \leq \pi \wedge \cos a = y \) \( 0 \leq b \wedge b \leq \pi \wedge \cos b = y \)
have cosd: \( \forall x::\mathtt{real}. \ \cos \ \text{differentiable} \ (\text{at} \ x) \)
  unfolding real-differentiable-def by (auto intro: DERIV-cos)
show a = b
  proof (cases a b rule: linorder-cases)
    case less
    then obtain z where a < z \( z < b \) (cos has-real-derivative 0) (at z)
    using Rolle by (metis cosd continuous-on-cos-real ab)
    then have sin z = 0
      using DERIV-cos DERIV-unique neg-equal-0-iff-equal by blast
    then show \( \text{thesis} \)
      by (metis (a < z) (z < b) ab order-less-le-trans less-le sin-gt-zero)
  next
    case greater
    then obtain z where b < z \( z < a \) (cos has-real-derivative 0) (at z)
    using Rolle by (metis cosd continuous-on-cos-real ab)
    then have sin z = 0
      using DERIV-cos DERIV-unique neg-equal-0-iff-equal by blast
    then show \( \text{thesis} \)
      by (metis (b < z) (z < a) ab order-less-le-trans less-le sin-gt-zero)
  qed

qed auto

lemma sin-total:
  assumes y: \( -1 \leq y \leq 1 \)
  shows \( \exists x. \ -(\pi/2) \leq x \wedge x \leq \pi/2 \wedge \sin x = y \)
proof -
  from cos-total \([OF y]\)
  obtain x where x: \( 0 \leq x \leq \pi \cos x = y \)
    and uniq: \( \forall x'. \ 0 \leq x' \implies x' \leq \pi \implies \cos x' = y \implies x' = x \)
  by blast
  show \( \text{thesis} \)
    unfolding sin-cos-eq
  proof (rule ex1I [where a=\pi/2 - x])
    show \( -(\pi/2) \leq z \wedge z \leq \pi/2 \wedge \cos (\text{of-real} \ \pi/2 - z) = y \implies z = \pi/2 - x \) for z
      using uniq [of \pi/2 - z] by auto
  qed (use x in auto)

qed

lemma cos-zero-lemma:
  assumes 0 \( \leq x \cos x = 0 \)
  shows \( \exists n. \ \text{odd} \ n \wedge x = \text{of-nat} \ n \ast (\pi/2) \wedge n > 0 \)
proof
  have xle: $x < (1 + \text{real-of-int } \lfloor x / \pi \rfloor) \times \pi$
  using floor-correct [of $x / \pi$]
  by (simp add: add.commute divide-less-eq)

obtain $n$ where $\text{real } n \times \pi \leq x < \text{real } (\text{Suc } n) \times \pi$
  proof
    show $\text{real } \lfloor x / \pi \rfloor \times \pi \leq x$
      using assms floor-divide-lower [of $\pi x$] by auto
    show $x < \text{real } (\text{Suc } \lfloor x / \pi \rfloor) \times \pi$
      using assms floor-divide-upper [of $\pi x$] by (simp add: xle)
  qed

  then have $0 \leq x - n \times \pi$
    using $x$ by blast
  moreover have $\pi / 2 = \vartheta$
    using $\pi$-half-ge-zero uniq by fastforce
  ultimately show ?thesis
    by (rule-tac $x = \text{Suc } (2 \times n)$ in exI) (simp add: algebra-simps)

qed

lemma sin-zero-lemma: $0 \leq x \Longrightarrow \sin x = 0 \Longrightarrow \exists n :: \text{nat}. \text{even } n \land x = \text{real } n \times (\pi / 2)$
  using cos-zero-lemma [of $x + \pi / 2$]
  apply (clarsimp simp add: cos-add)
  apply (rule-tac $x = n - 1$ in exI)
  apply (simp add: algebra-simps of-nat-diff)
  done

lemma cos-zero-iff:
  \(\cos x = 0 \Longleftrightarrow (\exists n. \text{odd } n \land x = \text{real } n \times (\pi / 2)) \lor (\exists n. \text{odd } n \land x = - (\text{real } n \times (\pi / 2)))\)
(is ?lhs = ?rhs)

proof
  have \(\ast: \cos (\text{real } n \times \pi / 2) = 0\) if odd $n$ for $n :: \text{nat}$
  proof
    from that obtain $m$ where $n = 2 \times m + 1$ ..
    then show ?thesis
      by (simp add: field-simps) (simp add: cos-add add-divide-distrib)
  qed

show ?thesis
proof
  show ?rhs if ?lhs
    using that cos-zero-lemma [of $x$] cos-zero-lemma [of $-x$] by force
show ?lhs if ?rhs using (auto dest: simp del: eq-divide-eq-numeral1)
qed

lemma sin-zero-iff:
sin x = 0 ↔ (∃ n. even n ∧ x = real n * (pi/2)) ∨ (∃ n. even n ∧ x = −(real n * (pi/2)))
is ?lhs = ?rhs
proof
show ?rhs if ?lhs using that sin-zero-lemma[of x] sin-zero-lemma[of −x] by force
show ?lhs if ?rhs using that by (auto elim: evenE)
qed

lemma sin-zero-pi-iff:
fixes x :: real
assumes |x| < pi
shows sin x = 0 ↔ x = 0
proof
show x = 0 if sin x = 0 using that assms by (auto simp: sin-zero-iff)
qed auto

lemma cos-zero-iff-int: cos x = 0 ↔ (∃ n. odd n ∧ x = of-int n * (pi/2))
proof
have 1: ∀ n. odd n −⇒ ∃ i. odd i ∧ real n = real-of-int i
by (metis even-of-nat of-int-of-nat-eq)
have 2: ∀ n. odd n −⇒ ∃ i. odd i ∧ −(real n * pi) = real-of-int i * pi
by (metis even-minus even-of-nat mult.commute mult-minus-right of-int-minus of-int-of-nat-eq)
have 3: [odd i; ∀ n. even n ∨ real-of-int i ≠ −(real n)]
−⇒ ∃ n. odd n ∧ real-of-int i = real n for i
by (cases i rule: int-cases2) auto
show ?thesis
by (force simp: cos-zero-iff intro!: 1 2 3)
qed

lemma sin-zero-iff-int: sin x = 0 ↔ (∃ n. even n ∧ x = of-int n * (pi/2))
proof safe
assume sin x = 0
then show ∃ n. even n ∧ x = of-int n * (pi/2)
apply (simp add: sin-zero-iff, safe)
apply (metis even-of-nat of-int-of-nat-eq)
apply (rule_tac x=−(int n) in exI)
apply simp
done
next
THEORY “Transcendental”

fix i :: int
assume even i
then show $\sin (\text{of-int } i \ast (\pi/2)) = 0$
  by (cases i rule: int-cases2, simp-all add: sin-zero-iff)
qed

lemma sin-zero-iff-int2: $\sin x = 0 \iff (\exists n::\text{int}. x = \text{of-int } n \ast \pi)$
apply (simp only: sin-zero-iff-int)
apply (safe elim!: evenE)
apply (simp-all add: field-simps)
using dvd-triv-left
apply fastforce
done

lemma sin-npi-int [simp]: $\sin (\pi \ast \text{of-int } n) = 0$
by (simp add: sin-zero-iff-int2)

lemma cos-monotone-0-pi:
  assumes $0 \leq y$ and $y < x$ and $x \leq \pi$
  shows $\cos x < \cos y$
proof –
  have $-(x - y) < 0$ using assms by auto
  from MVT2[OF $y < x$ DERIV-cos]
  obtain z where $y < z$ and $z < x$ and cos-diff: $\cos x - \cos y = (x - y) \ast -\sin z$
    by auto
  then have $0 < z$ and $z < \pi$
    using assms by auto
  then have $0 < \sin z$
    using sin-gt-zero by auto
  then have $\cos x - \cos y < 0$
    unfolding cos-diff minus-mult-commute[symmetric]
    using $(x - y) < 0$ by (rule mult-pos-neg2)
  then show ?thesis by auto
qed

lemma cos-monotone-0-pi-le:
  assumes $0 \leq y$ and $y \leq x$ and $x \leq \pi$
  shows $\cos x \leq \cos y$
proof (cases $y < x$)
  case True
  show ?thesis
    using cos-monotone-0-pi[OF $0 \leq y$ True $x \leq \pi$] by auto
next
  case False
  then have $y = x$ using $(y \leq x)$ by auto
  then show ?thesis by auto
qed

lemma cos-monotone-minus-pi-0:
assumes $-\pi \leq y$ and $y < x$ and $x \leq 0$
shows $\cos y < \cos x$
proof - 
  have $0 \leq -x$ and $-x < -y$ and $-y \leq \pi$
  using assms by auto
from $\cos$-monotone-$0$-$\pi$[OF this] show $\theta$thesis
  unfolding $\cos$-$\cos$-eq.
qed

lemma $\cos$-monotone-minus-$\pi$-$0$):
  assumes $-\pi \leq y$ and $y \leq x$ and $x \leq 0$
  shows $\cos y \leq \cos x$
proof (cases $y < x$)
  case True
  show $\theta$thesis using $\cos$-monotone-minus-$\pi$-$0$[OF $(-\pi \leq y)$ True ($x \leq 0$)]
    by auto
next
  case False
  then have $y = x$ using $(y \leq x)$ by auto
  then show $\theta$thesis by auto
qed

lemma $\sin$-monotone-$2\pi$:
  assumes $-(\pi/2) \leq y$ and $y < x$ and $x \leq \pi/2$
  shows $\sin y < \sin x$
unfolding $\sin$-$\cos$-eq
  using assms by (auto intro: $\cos$-monotone-$0$-$\pi$)

lemma $\sin$-monotone-$2\pi$-$le$:
  assumes $-(\pi/2) \leq y$ and $y \leq x$ and $x \leq \pi/2$
  shows $\sin y \leq \sin x$
  by (metis assms le-less $\sin$-monotone-$2\pi$)

lemma $\sin$-$x$-$le$-$x$:
  fixes $x$ :: real
  assumes $x \geq 0$
  shows $\sin x \leq x$
proof - 
  let $\theta f = \lambda x. x - \sin x$
  from $x$ have $\theta f x \geq \theta f 0$
    apply (rule DERIV-nonneg-imp-nondecreasing)
    apply (intro allI impI exI[of -$1 - \cos x$ for $x$])
    apply (auto intro!: derivative-eq-intros simp: field-simps)
  done
  then show $\sin x \leq x$ by simp
qed

lemma $\sin$-$x$-$ge$-$neg$-$x$:
  fixes $x$ :: real
assumes $x: x \geq 0$
shows $\sin x \geq -x$
proof
let $\dot{f} = \lambda x. x + \sin x$
from $x$ have $\dot{f} x \geq \dot{f} 0$
apply (rule DERIV-nonneg-imp-nondecreasing)
apply (intro allI impI exI [of $-1 + \cos x$ for $x$])
apply (auto intro!: deriv-eq-intros simp: field-simps real-0-le-add-iff)
done
then show $\sin x \geq -x$ by simp
qed

lemma abs-sin-x-le-abs-x: $|\sin x| \leq |x|
for $x :: \text{real}$
using sin-x-ge-neg-x [of $x$] sin-x-le-x [of $x$] sin-x-ge-neg-x [of $-x$] sin-x-le-x [of $-x$]
by (auto simp: abs-real-def)

110.14 More Corollaries about Sine and Cosine

lemma sin-cos-npi [simp]: $\sin \left( \text{real} \left( \text{Suc} \left( 2 * n \right) \right) * \pi / 2 \right) = (-1)^n$
proof
have $\sin \left( \left( \text{real} \left( n + 1 / 2 \right) \right) * \pi \right) = \cos \left( \text{real} \left( n * \pi \right) \right)$
  by (auto simp: algebra-simps sin-add)
then show $\text{thesis}$
  by (simp add: distrib-right add-divide-distrib add.commute mult.commute [of $\pi$])
qed

lemma cos-2npi [simp]: $\cos \left( 2 * \text{real} \left( n * \pi \right) \right) = 1$
for $n :: \text{nat}$
by (cases even $n$) (simp-all add: cos-double mult.assoc)

lemma cos-3over2-pi [simp]: $\cos \left( 3 / 2 \pi \right) = 0$
proof
have $\cos \left( 3 / 2 \pi \right) = \cos \left( \pi + \pi / 2 \right)$
  by simp
also have $\ldots = 0$
  by (subst cos-add, simp)
finally show $\text{thesis}$. 
qed

lemma sin-2npi [simp]: $\sin \left( 2 * \text{real} \left( n * \pi \right) \right) = 0$
for $n :: \text{nat}$
by (auto simp: mult.assoc sin-double)

lemma sin-3over2-pi [simp]: $\sin \left( 3 / 2 \pi \right) = -1$
proof
have $\sin \left( 3 / 2 \pi \right) = \sin \left( \pi + \pi / 2 \right)$
by simp
also have ... = −1
  by (subst sin-add simp)
finally show thesis .
qed

lemma cos-pi-eq-zero [simp]: cos (pi * real (Suc (2 * m)) / 2) = 0
  by (simp only: cos-add sin-add of-nat-Suc distrib-right distrib-left add-divide-distrib auto)

lemma DERIV-cos-add [simp]: DERIV (λx. cos (x + k)) xa > − sin (xa + k)
  by (auto intro!: derivative-eq-intros)

lemma sin-zero-abs-cos-one: sin x = 0 =⇒ |cos x| = (1::real)
  using sin-zero-norm-cos-one by fastforce

lemma cos-one-2pi: cos x = 1 =⇒ (∃ n::nat. x = n * 2 * pi) ∨ (∃ n::nat. x = -(n * 2 * pi))
  (is ?lhs = ?rhs)
proof
assume ?lhs
  then have sin x = 0
    by (simp add: cos-one-sin-zero)
then show ?rhs
proof (simp only: sin-zero-iff, elim exE disjE conjE)
fix n :: nat
  assume n: even n x = real n * (pi/2)
then obtain m where n: n = 2 * m
    using dvdE by blast
  then have me: even m using (?lhs) n
    by (auto simp: field-simps) (metis one-neq-neg-one power-minus-odd power-one)
show ?rhs
using m me n
by (auto simp: field-simps elim!: evenE)

next
fix n :: nat
assume n: even n x = − (real n * (pi/2))
then obtain m where m: n = 2 * m
using dvdE by blast
then have me: even m using ⟨lhs⟩
by (auto simp: field-simps)
(show ?rhs
using m me n
by (auto simp: field-simps elim!: evenE)
qed

next
assume ?rhs
then show cos x = 1
by (metis cos-2npi cos-minus mult_assoc mult-commute)
qed

lemma cos-one-2pi-int: cos x = 1 \iff (\exists n::int. x = n * 2 * pi) (is ?lhs = ?rhs)
proof
assume cos x = 1
then show ?rhs
by (metis cos-2npi cos-minus mult_assoc mult-commute)
next
assume ?rhs
then show cos x = 1
by (clarsimp simp add: cos-one-2pi) (metis mult-commute)
qed

lemma cos-npi-int: fixes n :: int shows cos (pi * of_int n) = (if even n then 1 else -1)
by (auto simp: algebra-simps cos-one-2pi-int elim!: oddE evenE)

lemma sin-cos-sqrt: 0 \leq sin x \implies sin x = sqrt (1 - (cos (x ^ 2)))
using sin-squared-eq real-sqrt-unique by fastforce

lemma sin-eq-0-pi: - pi < x \implies x < pi \implies sin x = 0 \implies x = 0
by (metis sin-gl-zero sin-minus-less-iff neg-0-less_iff_less not-less_iff_gr_or_eq)

lemma cos-treble-cos: cos (3 * x) = 4 * cos x ^ 3 - 3 * cos x
for x :: 'a::{real-normed-field,banach}
proof
have *: (sin x * (sin x * 3)) = 3 - (cos x * (cos x * 3))
by (simp add: mult_assoc [symmetric] sin-squared-eq [unfolded power2-eq-square])
have cos(3 * x) = cos(2*x + x)
by simp
also have \ldots = 4 * cos x ^ 3 - 3 * cos x
apply (simp only: cos-add cos-double sin-double)
apply (simp add: * field-simps power2-eq-square power3-eq-cube)
done
finally show ?thesis.
qed

lemma cos-45: cos (pi / 4) = sqrt (2 / 2)
proof –
  let ?c = cos (pi / 4)
  let ?s = sin (pi / 4)
  have nonneg: 0 ≤ ?c
    by (simp add: cos-ge-zero)
  have 0 = cos (pi / 4 + pi / 4)
    by simp
  also have cos (pi / 4 + pi / 4) = ?c^2 - ?s^2
    by (simp only: cos-add power2-eq-square)
  also have ... = 2 * ?c^2 - 1
    by (simp add: sin-squared-eq)
  finally have ?c^2 = (sqrt (2 / 2))^2
    by (simp add: power-divide)
  then show ?thesis
    using nonneg by (rule power2-eq-imp-eq) simp
qed

lemma cos-30: cos (pi / 6) = sqrt (3 / 2)
proof –
  let ?c = cos (pi / 6)
  let ?s = sin (pi / 6)
  have pos-c: 0 < ?c
    by (rule cos-gt-zero) simp-all
  have 0 = cos (pi / 6 + pi / 6 + pi / 6)
    by simp
  also have ... = (?c * ?c - ?s * ?s) * ?c - (?s * ?c + ?c * ?s) * ?s
    by (simp only: cos-add sin-add)
  also have ... = ?c * (?c^2 - 3 * ?s^2)
    by (simp add: algebra-simps power2-eq-square)
  finally have ?c^2 = (sqrt (3 / 2))^2
    using pos-c by (simp add: sin-squared-eq power-divide)
  then show ?thesis
    using pos-c [THEN order-less-imp-le]
    by (rule power2-eq-imp-eq) simp
qed

lemma sin-45: sin (pi / 4) = sqrt (2 / 2)
  by (simp add: sin-cos-eq cos-45)

lemma sin-60: sin (pi / 3) = sqrt (3 / 2)
  by (simp add: sin-cos-eq cos-30)

lemma cos-60: cos (pi / 3) = 1 / 2
proof –
  have 0 ≤ cos (pi / 3)
    by (rule cos-ge-zero) (use pi-half-ge-zero in linarith+)
  then show ?thesis
    by (simp add: cos-squared-eq sin-60 power-divide power2-eq-imp-eq)
qed

lemma sin-30: sin (pi / 6) = 1 / 2
  by (simp add: sin-cos-eq cos-60)

lemma cos-integer-2pi: n ∈ ℤ ⇒ cos (2 * pi * n) = 1
  by (metis Ints-cases cos-one-2pi-int mult.assoc mult.commute)

lemma sin-integer-2pi: n ∈ ℤ ⇒ sin (2 * pi * n) = 0
  by (metis sin-two-pi Ints-mult mult.assoc mult.commute sin-times-pi-eq-0)

lemma sincos-principal-value: ∃ y. (− pi < y ∧ y ≤ pi) ∧ (sin y = sin x ∧ cos y = cos x)
  apply (rule exI [where x = pi − (2 * pi) * frac ((pi − x) / (2 * pi))])
  apply (auto simp: field-simps frac-lt-1)
  apply (simp-all add: frac-def field-simps)
  apply (simp-all add: add-divide-distrib diff-divide-distrib)
  apply (simp-all add: sin-add cos-add mult.assoc [symmetric])
  done

110.15 Tangent

definition tan :: 'a ⇒ 'a::{real-normed-field,banach}
where tan = (λx. sin x / cos x)

lemma tan-of-real: of-real (tan x) = (tan (of-real x) :: 'a::{real-normed-field,banach})
  by (simp add: tan-def sin-of-real cos-of-real)

lemma tan-in-Reals [simp]: z ∈ ℝ ⇒ tan z ∈ ℝ
  for z :: 'a::{real-normed-field,banach}
  by (simp add: tan-def)

lemma tan-zero [simp]: tan 0 = 0
  by (simp add: tan-def)

lemma tan-pi [simp]: tan pi = 0
  by (simp add: tan-def)
lemma tan-npi [simp]: \( \tan (\text{real } n \cdot \pi) = 0 \)
  for \( n :: \text{nat} \)
  by (simp add: tan-def)

lemma tan-minus [simp]: \( \tan (-x) = -\tan x \)
  by (simp add: tan-def)

lemma tan-periodic [simp]: \( \tan (x + 2 \cdot \pi) = \tan x \)
  by (simp add: tan-def)

lemma lemma-tan-add1: \( \cos x \neq 0 = \Rightarrow \cos y \neq 0 = \Rightarrow 1 - \tan x \cdot \tan y = \cos(x + y)/(\cos x \cdot \cos y) \)
  by (simp add: tan-def cos-add field-simps)

lemma add-tan-eq: \( \cos x \neq 0 = \Rightarrow \cos y \neq 0 = \Rightarrow \tan x + \tan y = \sin(x + y)/(\cos x \cdot \cos y) \)
  for \( x :: 'a::\{\text{real-normed-field},\text{banach}\} \)
  by (simp add: tan-def sin-add field-simps)

lemma tan-add: \( \cos x \neq 0 = \Rightarrow \cos y \neq 0 = \Rightarrow \cos(x + y) = (\tan x + \tan y)/((1 - \tan x \cdot \tan y)\)
  for \( x :: 'a::\{\text{real-normed-field},\text{banach}\} \)
  by (simp add: add-tan-eq lemma-tan-add1 field-simps) (simp add: tan-def)

lemma tan-double: \( \cos x \neq 0 = \Rightarrow \cos (2 \cdot x) \neq 0 = \Rightarrow \tan (2 \cdot x) = (2 \cdot \tan x)/((1 - (\tan x)^2)\)
  for \( x :: 'a::\{\text{real-normed-field},\text{banach}\} \)
  using tan-add [of x x] by (simp add: power2-eq-square)

lemma tan-gt-zero: \( 0 < x = \Rightarrow x < \pi/2 = \Rightarrow 0 < \tan x \)
  by (simp add: tan-def zero-less-divide-iff sin-gt-zero2 cos-gt-zero-pi)

lemma tan-less-zero:
  assumes \( -\pi/2 < x \) and \( x < 0 \)
  shows \( \tan x < 0 \)
proof --
  have \( 0 < \tan (-x) \)
    using assms by (simp only: tan-gt-zero)
  then show \?thesis by simp
qed

lemma tan-half: \( \tan x = \sin(2 \cdot x)/((\cos(2 \cdot x) + 1)\)
  for \( x :: 'a::\{\text{real-normed-field},\text{banach},\text{field}\} \)
  unfolding tan-def sin-double cos-double sin-squared-eq
  by (simp add: power2-eq-square)

lemma tan-30: \( \tan (\pi / 6) = 1/\sqrt{3} \)
  unfolding tan-def by (simp add: sin-30 cos-30)
lemma `tan-45`: \(\tan \left(\frac{\pi}{4}\right) = 1\)
  unfolding `tan-def` by (simp add: `sin-45 cos-45`)

lemma `tan-60`: \(\tan \left(\frac{\pi}{3}\right) = \sqrt{3}\)
  unfolding `tan-def` by (simp add: `sin-60 cos-60`)

lemma `DERIV-tan` [simp]: \(\cos x \neq 0 \Rightarrow \text{DERIV \(\tan x\)} : >\)
  for \(x :: 'a::{\text{real-normed-field,banach}}\)
  unfolding `tan-def`
  by (auto intro!: `derivative-eq-intros`, simp add: `divide-inverse power2-eq-square`)

declare `DERIV-tan` [THEN `DERIV-chain2`, `derivative-intros`]
and `DERIV-tan` [THEN `DERIV-chain2`, unfolded `has-field-derivative-def`, `derivative-intros`]
lemmas `has-derivative-tan` [`derivative-intros`] = `DERIV-tan` [THEN `DERIV-compose-FDERIV`]

lemma `isCont-tan`: \(\cos x \neq 0 \Rightarrow \text{isCont \(\tan x\)}\)
  for \(x :: 'a::{\text{real-normed-field,banach}}\)
  by (rule `DERIV-tan` [THEN `DERIV-isCont`])

lemma `isCont-tan` [simp,continuous-intros]:
  fixes \(a :: 'a::{\text{real-normed-field,banach}}\) and \(f :: 'a \Rightarrow 'a\)
  shows `isCont f a \Rightarrow \cos (f a) \neq 0 \Rightarrow \text{isCont} (\lambda x. \tan (f x)) a`
  by (rule `isCont-o2 [OF - isCont-tan]`)

lemma `tendsto-tan` [tendsto-intros]:
  fixes \(f :: 'a \Rightarrow 'a::{\text{real-normed-field,banach}}\)
  shows \((f \longrightarrow a) F \Rightarrow \cos a \neq 0 \Rightarrow (\lambda x. \tan (f x)) \longrightarrow \tan a) F\)
  by (rule `isCont-tendsto-compose [OF isCont-tan]`)

lemma `continuous-tan`:
  fixes \(f :: 'a \Rightarrow 'a::{\text{real-normed-field,banach}}\)
  shows `continuous F f \Rightarrow \cos (f (\text{Lim} F (\lambda x. x))) \neq 0 \Rightarrow \text{continuous} F (\lambda x. \tan (f x))`
  unfolding `continuous-def` by (rule `tendsto-tan`)

lemma `continuous-on-tan` [continuous-intros]:
  fixes \(f :: 'a \Rightarrow 'a::{\text{real-normed-field,banach}}\)
  shows `continuous-on s f \Rightarrow (\forall x \in s. \cos (f x) \neq 0) \Rightarrow \text{continuous-on} s (\lambda x. \tan (f x))`
  unfolding `continuous-on-def` by (auto intro: `tendsto-tan`)

lemma `continuous-within-tan` [continuous-intros]:
  fixes \(f :: 'a \Rightarrow 'a::{\text{real-normed-field,banach}}\)
  shows `continuous (at x within s) f \Rightarrow \cos (f x) \neq 0 \Rightarrow \text{continuous (at x within s)} (\lambda x. \tan (f x))`
  unfolding `continuous-within` by (rule `tendsto-tan`)
lemma LIM-cos-div-sin: $(\lambda x. \cos(x)/\sin(x)) - \pi/2 \to 0$
by (rule tendsto-cong-limit, (rule tendsto-intros)+, simp-all)

lemma lemma-tan-total:
assumes $0 < y$
shows $\exists x. 0 < x \land x < \pi/2 \land y < \tan x$
proof
obtain $s$ where $0 < s$
and $s: \{x. [x \neq \pi/2; \text{norm} (x - \pi/2) < s] \implies \text{norm} (\cos x / \sin x - 0) < \text{inverse } y\}$
using LIM-D [OF LIM-cos-div-sin, of inverse y] that assms by force
obtain $e$ where $0 < e < e < \pi/2$
using $\langle 0 < s \rangle$ field-lbound-gt-zero pi-half-gt-zero by blast
show ?thesis
proof (intro exI conjI)
  have $0 < \sin e 0 < \cos e$
  using $e$ by (auto intro: cos-gt-zero sin-gt-zero2 simp: mult.commute)
  then show $y < \tan (\pi/2 - e)$
  using $s [of \pi/2 - e]$ e assms by (simp add: tan-def sin-diff cos-diff) (simp add: field-simps split: if-split-asm)
qed (use e in auto)
qed

lemma tan-total-pos:
assumes $0 \leq y$
shows $\exists x. 0 \leq x \land x < \pi/2 \land \tan x = y$
proof (cases y = 0)
  case True
  then show ?thesis
  using pi-half-gt-zero tan-zero by blast
next
  case False
  with assms have $y > 0$
  by linarith
obtain $x$ where $x: 0 < x x < \pi/2 y < \tan x$
using lemma-tan-total $(0 < y)$ by blast
have $\exists u > 0. u \leq x \land \tan u = y$
proof (intro IVT allI impI)
  show isCont $\tan u$ if $0 \leq u \land u \leq x$ for $u$
  proof
    have $\cos u \neq 0$
    using antisym-conv2 cos-gt-zero that $x(2)$ by fastforce
    with assms show ?thesis
    proof
      by (auto intro!: DERIV-tan [THEN DERIV-isCont])
    qed
    qed (use assms x in auto)
  then show ?thesis
  using $x(2)$ by auto
  qed
THEORY "Transcendental"

lemma lemma-tan-total1: \( \exists x. - (\pi/2) < x \land x < (\pi/2) \land \tan x = y \)
proof (cases 0::<real y rule: le-cases)
  case le
  then show \( \text{thesis} \) by (meson less-le-trans minus-pi-half-less-zero tan-total-pos)
next
  case ge
  with tan-total-pos [of \(-y\)] obtain \( x \) where \( 0 \leq x < \pi / 2 \tan x = - y \)
  by force
  then show \( \text{thesis} \) by (rule-tac \( x = -x \) in exI) auto
qed

proposition tan-total: \( \exists! x. - (\pi/2) < x \land x < (\pi/2) \land \tan x = y \)
proof
  have \( u = v \) if \( u: - (\pi / 2) < u u < \pi / 2 \) and \( v: - (\pi / 2) < v v < \pi / 2 \)
  and eq: \( \tan u = \tan v \) for \( u \) and \( v \)
proof (cases \( u \) \( v \) rule: linorder-cases)
  case less
  have \( \forall x. u \leq x \land x \leq v \longrightarrow \text{isCont} \tan x \)
  by (metis cos-gt-zero-pi isCont-tan le-less-trans less-irrefl less-le-trans \( u \) \( v \))
  then have continuous-on \( \{ u .. v \} \tan \)
  by (simp add: continuous-at-imp-continuous-on)
  moreover have \( \forall x. u < x \land x < v \longrightarrow \tan \text{differentiable} \ (at \ x) \)
  by (metis DERIV-tan cos-gt-zero-pi real-differentiable-def less-numeral-extra(3) order.strict-trans \( u(1) \) \( v(2) \))
  ultimately obtain \( z \) where \( u < z z < v \) DERIV \( \tan z := 0 \)
  by (metis less Rolle eq)
  moreover have \( \cos z \neq 0 \)
  by (metis (no-types) \( u < z \) \( z < v \) cos-gt-zero-pi less-le-trans linorder-not-less)
  ultimately show \( \text{thesis} \)
  using DERIV-unique [OF - DERIV-tan] by fastforce
next
  case greater
  have \( \forall x. v < x \land x < u \longrightarrow \text{isCont} \tan x \)
  by (metis cos-gt-zero-pi isCont-tan le-less-trans less-irrefl less-le-trans \( u(2) \) \( v(1) \))
  then have continuous-on \( \{ v .. u \} \tan \)
  by (simp add: continuous-at-imp-continuous-on)
  moreover have \( \forall x. v < x \land x < u \longrightarrow \tan \text{differentiable} \ (at \ x) \)
  by (metis DERIV-tan cos-gt-zero-pi real-differentiable-def less-numeral-extra(3) order.strict-trans \( u(2) \), \( v(1) \))
  ultimately obtain \( z \) where \( v < z z < u \) DERIV \( \tan z := 0 \)
  by (metis greater Rolle eq)
  moreover have \( \cos z \neq 0 \)
  by (metis \( v < z \) \( z < u \) cos-gt-zero-pi less-eq-real-def less-le-trans order-less-irrefl \( u(2) \), \( v(1) \))
ultimately show thesis
  using DERIV-unique [OF - DERIV-tan] by fastforce
qed auto
then have ∃x. − (pi / 2) < x ∧ x < pi / 2 ∧ tan x = y
  if x: −(pi / 2) < x x < pi / 2 tan x = y for x
  using that by auto
then show thesis
  using lemma-tan-total1 [where y = y]
  by auto
qed

lemma tan-monotone:
  assumes − (pi/2) < y and y < x and x < pi/2
  shows tan y < tan x
proof
  have DERIV tan x' := inverse ((cos x')^2) if y ≤ x' x' ≤ x for x'
  proof
    have −(pi/2) < x' and x' < pi/2
      using that assms by auto
    with cos-gt-zero-pi have cos x' ≠ 0 by force
    then show DERIV tan x' := inverse ((cos x')^2)
      by (rule DERIV-tan)
  qed
from MVT2[OF ⟨y < x⟩ this]
obtain z where y < z and z < x
  and tan-diff: tan x − tan y = (x − y) * inverse ((cos z)^2) by auto
then have − (pi/2) < z and z < pi/2
  using assms by auto
then have 0 < cos z
  using cos-gt-zero-pi by auto
then have inv-pos: 0 < inverse ((cos z)^2)
  by auto
have 0 < x − y using ⟨y < x⟩ by auto
with inv-pos have 0 < tan x − tan y
  unfolding tan-diff by auto
then show thesis by auto
qed

lemma tan-monotone':
  assumes − (pi/2) < y
  and y < pi/2
  and − (pi/2) < x
  and x < pi/2
  shows y < x −→ tan y < tan x
proof
  assume y < x
  then show tan y < tan x
    using tan-monotone and ⟨− (pi/2) < y⟩ and ⟨x < pi/2⟩ by auto
next
assume $\tan y < \tan x$
show $y < x$
proof (rule ccontr)
  assume $\neg \text{thesis}$
  then have $x \leq y$ by auto
  then have $\tan x \leq \tan y$
proof (cases $x = y$)
  case True
  then show $\text{thesis}$ by auto
next
  case False
  then have $x < y$ using $x \leq y$ by auto
  from $\tan$-monotone[OF $(- (\pi/2)) < x$] this $y < \pi/2$] show $\text{thesis}$
  by auto
qed
then show False
  using $(\tan y < \tan x)$ by auto
qed
qed

lemma $\tan$-inverse: $1 / (\tan y) = \tan (\pi/2 - y)$
unfolding $\tan$-def $\sin$-$\cos$-eq[of $y$] $\cos$-$\sin$-eq[of $y$] by auto

lemma $\tan$-periodic-pi[simp]: $\tan (x + \pi) = \tan x$
  by (simp add: $\tan$-def)

lemma $\tan$-periodic-nat[simp]: $\tan (x + \text{real } n * \pi) = \tan x$
  for $n$ :: nat
proof (induct $n$ arbitrary: $x$)
  case 0
  then show $\text{case}$ by simp
next
  case (Suc $n$)
  have split-pi-off: $x + \text{real } (\text{Suc } n) * \pi = (x + \text{real } n * \pi) + \pi$
  unfolding Suc-eq-plus1 of-nat-add distrib-right by auto
  show $\text{case}$
    unfolding split-pi-off using Suc by auto
qed

lemma $\tan$-periodic-int[simp]: $\tan (x + \text{of-int } i * \pi) = \tan x$
proof (cases $0 \leq i$)
  case True
  then have $\text{i-nat}: \text{of-int } i = \text{of-int } (\text{nat } i)$ by auto
  show $\text{thesis}$ unfolding i-nat
    by (metis of-int-of-nat-eq $\tan$-periodic-nat)
next
  case False
  then have $\text{i-nat}: \text{of-int } i = - \text{of-int } (\text{nat } (- i))$ by auto
  have $\tan x = \tan (x + \text{of-int } i * \pi - \text{of-int } i * \pi)$
by auto
also have \[= \tan \left( x + \text{of-int } i \times \pi \right) \]
unfolding i-nat mult-minus-left diff-minus-eq-add
by (metis of-int-of-nat-eq tan-periodic-nat)
finally show ?thesis by auto
qed

lemma tan-periodic-n simp: \( \tan \left( x + \text{numeral } n \times \pi \right) = \tan x \)
using tan-periodic-int [of \( \text{numeral } n \)] by simp

lemma tan-minus-45: \( \tan \left( -\left( \pi/4 \right) \right) = -1 \)
unfolding tan-def by (simp add: sin-45 cos-45)

lemma tan-diff:
\[\cos x \neq 0 \implies \cos y \neq 0 \implies \cos (x - y) = (\tan x - \tan y)/(1 + \tan x \times \tan y)\]
for \( x :: 'a::{real_normed_field,banach} \)
using tan-add [of \( x - y \)] by simp

lemma tan-pos-pi2-le:
\[0 \leq x \implies x < \pi/2 \implies 0 \leq \tan x\]
using less-eq-real-def tan-gt-zero by auto

lemma cos-tan:
\[|x| < \pi/2 \implies \cos x = 1/\sqrt{(1 + \tan x \times 2)}\]
using cos-gt-zero-pi [of \( x \)] by (simp add: field-split-simps tan-def real-sqrt-divide abs-if split: if-split-asm)

lemma sin-tan:
\[|x| < \pi/2 \implies \sin x = \tan x / \sqrt{(1 + \tan x \times 2)}\]
using cos-gt-zero [of \( x \)] cos-gt-zero [of \( -x \)]
by (force simp: field-split-simps tan-def real-sqrt-divide abs-if split: if-split-asm)

lemma tan-mono-le:
\[-\left( \pi/2 \right) < x \implies x < y \implies y < \pi/2 \implies \tan x \leq \tan y\]
using less-eq-real-def tan-monotone by auto

lemma tan-mono-lt-eq:
\[-\left( \pi/2 \right) < x \implies x < \pi/2 \implies -\left( \pi/2 \right) < y \implies y < \pi/2 \implies \tan x < \tan y\]
exchange \( x < y \)
using tan-monotone’ by blast

lemma tan-mono-le-eq:
\[-\left( \pi/2 \right) < x \implies x < \pi/2 \implies -\left( \pi/2 \right) < y \implies y < \pi/2 \implies \tan x \leq \tan y\]
exchange \( x \leq y \)
by (meson tan-mono-le not-le tan-monotone)

lemma tan-bound-pi2: \(|x| < \pi/4 \implies |\tan x| < 1\)
using tan-45 tan-monotone [of \( x \pi/4 \)] tan-monotone [of \( -x \pi/4 \)]
by (auto simp: abs-if split: if-split-asm)

lemma tan-cot: \(\tan(\pi/2 - x) = \text{inverse}(\tan x)\)
by (simp add: tan-def sin-diff cos-diff)
110.16 Cotangent

definition cot :: 'a ⇒ 'a::{real-normed-field,banach}
  where cot = (λx. cos x / sin x)

lemma cot-of-real: of-real (cot x) = (cot (of-real x) :: 'a::{real-normed-field,banach})
  by (simp add: cot-def sin-of-real cos-of-real)

lemma cot-in-Reals [simp]: z ∈ ℝ ⇒ cot z ∈ ℝ
  for z :: 'a::{real-normed-field,banach}
  by (simp add: cot-def)

lemma cot-zero [simp]: cot 0 = 0
  by (simp add: cot-def)

lemma cot-pi [simp]: cot pi = 0
  by (simp add: cot-def)

lemma cot-api [simp]: cot (real n * pi) = 0
  for n :: nat
  by (simp add: cot-def)

lemma cot-minus [simp]: cot (− x) = − cot x
  by (simp add: cot-def)

lemma cot-periodic [simp]: cot (x + 2 * pi) = cot x
  by (simp add: cot-def)

lemma cot-altdef: cot x = inverse (tan x)
  by (simp add: cot-def tan-def)

lemma tan-altdef: tan x = inverse (cot x)
  by (simp add: cot-def tan-def)

lemma tan-cot': tan (pi/2 − x) = cot x
  by (simp add: tan-cot cot-altdef)

lemma cot-gt-zero: 0 < x ⇒ x < pi/2 ⇒ 0 < cot x
  by (simp add: cot-def zero-less-divide-iff sin-gt-zero2 cos-gt-zero-pi)

lemma cot-less-zero:
  assumes lb: − pi/2 < x and x < 0
  shows cot x < 0
proof −
  have 0 < cot (− x)
    using assms by (simp only: cot-gt-zero)
  then show ?thesis by simp
qed

lemma DERIV-cot [simp]: sin x ≠ 0 ⇒ DERIV cot x :> − inverse ((sin x)²)
for $x :: 'a::{real-normed-field,banach}$

unfolding cot-def using cos-squared-eq[of $x$]
by (auto intro!: derivative-eq-intros) (simp add: divide-inverse power2-eq-square)

lemma isCont-cot: $\sin x \neq 0 \Rightarrow \text{isCont } \cot x$
for $x :: 'a::{real-normed-field,banach}$
by (rule DERIV-cot [THEN DERIV-isCont])

lemma isCont-cot': [simp,continuous-intros]:
$\text{isCont } f a \Rightarrow \sin f a \neq 0 \Rightarrow \text{isCont } (\lambda x. \cot (f x)) a$
for $a :: 'a::{real-normed-field,banach}$ and $f :: 'a \Rightarrow 'a$
by (rule isCont-o2 [OF - isCont-cot])

lemma tendsto-cot [tendsto-intros]: $(f \longmapsto a) F \Rightarrow \sin a \neq 0 \Rightarrow ((\lambda x. \cot (f x)) \longmapsto \cot a) F$
for $f :: 'a \Rightarrow 'a::{real-normed-field,banach}$
by (rule isCont-tendsto-compose [OF isCont-cot])

lemma continuous-cot:
$\text{continuous } F f \Rightarrow \sin (\lim F (\lambda x. x)) \neq 0 \Rightarrow \text{continuous } (\lambda x. \cot (f x))$
for $f :: 'a \Rightarrow 'a::{real-normed-field,banach}$
unfolding continuous-def by (rule tendsto-cot)

lemma continuous-on-cot [continuous-intros]:
fixes $f :: 'a \Rightarrow 'a::{real-normed-field,banach}$
shows $\text{continuous-on } s f \Rightarrow (\forall x \in s. \sin (f x) \neq 0) \Rightarrow \text{continuous-on } s (\lambda x. \cot (f x))$
unfolding continuous-on-def by (auto intro: tendsto-cot)

lemma continuous-within-cot [continuous-intros]:
fixes $f :: 'a \Rightarrow 'a::{real-normed-field,banach}$
shows $\text{continuous } (\text{at } x \text{ within } s) f \Rightarrow \sin (f x) \neq 0 \Rightarrow \text{continuous } (\text{at } x \text{ within } s) (\lambda x. \cot (f x))$
unfolding continuous-within by (rule tendsto-cot)

110.17 Inverse Trigonometric Functions

definition arcsin :: $\text{real } \Rightarrow \text{real}$
where $\text{arcsin } y = (\text{THE } x. -(\pi/2) \leq x \land x \leq \pi/2 \land \sin x = y)$

definition arccos :: $\text{real } \Rightarrow \text{real}$
where $\text{arccos } y = (\text{THE } x. 0 \leq x \land x \leq \pi \land \cos x = y)$

definition arctan :: $\text{real } \Rightarrow \text{real}$
where $\text{arctan } y = (\text{THE } x. -(\pi/2) < x \land x < \pi/2 \land \tan x = y)$

lemma arcsin: $-1 \leq y \Rightarrow y \leq 1 \Rightarrow -(\pi/2) \leq \text{arcsin } y \land \text{arcsin } y \leq \pi/2$
$\land \sin (\text{arcsin } y) = y$
unfolding arcsin-def by (rule the1’ [OF sin-total])

lemma arcsin-pi: \(-1 \leq y \implies y \leq 1 \implies -(\pi/2) \leq \arcsin y \land \arcsin y \leq \pi\)
  by (drule (1) arcsin) (force intro: order-trans)

lemma sin-arcsin [simp]: \(-1 \leq y \implies y \leq 1 \implies \sin (\arcsin y) = y\)
  by (blast dest: arcsin)

lemma arcsin-bounded: \(-1 \leq y \implies y \leq 1 \implies -(\pi/2) \leq \arcsin y \land \arcsin y \leq \pi/2\)
  by (blast dest: arcsin)

lemma arcsin-lbound: \(-1 \leq y \implies y \leq 1 \implies -(\pi/2) \leq \arcsin y\)
  by (blast dest: arcsin)

lemma arcsin-ubound: \(-1 \leq y \implies y \leq 1 \implies \arcsin y \leq \pi/2\)
  by (blast dest: arcsin)

lemma arcsin-lt-bounded:
  assumes \(-1 < y < 1\)
  shows \(-\pi/2 < \arcsin y \land \arcsin y < \pi/2\)
  proof –
    have \(\arcsin y \neq \pi/2\)
      by (metis arcsin assms not-less not-less-iff-or-eq sin-pi-half)
    moreover have \(\arcsin y \neq -\pi/2\)
      by (metis arcsin assms minus-divide-left not-less not-less-iff-or-eq sin-minus
          sin-pi-half)
    ultimately show \(?\thesis) using arcsin-bounded [of y] assms by auto
  qed

lemma arcsin-sin: \(-\pi/2 \leq x \implies x \leq \pi/2 \implies \arcsin (\sin x) = x\)
  unfolding arcsin-def
  using the1-equality [OF sin-total] by simp

lemma arcsin-0 [simp]: \(\arcsin 0 = 0\)
  using arcsin-sin [of 0] by simp

lemma arcsin-1 [simp]: \(\arcsin 1 = \pi/2\)
  using arcsin-sin [of \(\pi/2\)] by simp

lemma arcsin-minus-1 [simp]: \(\arcsin (-1) = -(\pi/2)\)
  using arcsin-sin [of \(-\pi/2\)] by simp

lemma arcsin-minus: \(-1 \leq x \implies x \leq 1 \implies \arcsin (-x) = -\arcsin x\)
  by (metis (no-types, hide-lams) arcsin arcsin-sin minus-minus neg-le-iff-le sin-minus)

lemma arcsin-eq-iff: \(\exists x \leq 1 \implies |y| \leq 1 \implies \arcsin x = \arcsin y \iff x = y\)
THEORY "Transcendental"

by (metis abs-le-iff arcsin minus-le-iff)

lemma cos-arcsin-nonzero: \(-1 < x \iff x < 1 \implies \cos (\arcsin x) \neq 0\)
using arcsin-lt-bounded cos-gt-zero-pi by force

lemma arccos: \(-1 \leq y \iff y \leq 1 \implies 0 \leq \arccos y \land \arccos y \leq \pi \land \cos (\arccos y) = y\)
unfolding arccos-def by (rule theI' [OF cos-total])

lemma cos-arccos [simp]: \(-1 \leq y \iff y \leq 1 \implies \cos (\arccos y) = y\)
by (blast dest: arccos)

lemma arccos-bounded: \(-1 \leq y \iff y \leq 1 \implies 0 \leq \arccos y \land \arccos y \leq \pi\)
by (blast dest: arccos)

lemma arccos-bound: \(-1 \leq y \implies y \leq 1 \implies 0 \leq \arccos y\)
by (blast dest: arccos)

lemma arccos-ubound: \(-1 \leq y \implies y \leq 1 \implies \arccos y \leq \pi\)
by (blast dest: arccos)

lemma arccos-lbound: \(-1 \leq y \implies y \leq 1 \implies 0 \leq \arccos y\)
by (blast dest: arccos)

lemma arccos-lt-bounded:
assumes \(-1 < y \land y < 1\)
shows \(0 < \arccos y \land \arccos y < \pi\)
proof
  have \(\arccos y \neq 0\)
  by (metis (no-types) arccos assms(1) assms(2) cos-zero less-eq-real-def less-irrefl)
  moreover have \(\arccos y \neq -\pi\)
  by (metis arccos assms(1) assms(2) cos-minus cos-pi not-less not-less-iff-gr-or-eq)
  ultimately show \(\text{thesis}\)
  using arccos-bounded [of y] assms
  by (metis arccos cos-pi not-less not-less-iff-gr-or-eq)
qed

lemma arccos-cos: \(0 \leq x \implies x \leq \pi \implies \arccos (\cos x) = x\)
by (auto simp: arccos-def intro!: theI-equality cos-total)

lemma arccos-cos2: \(x \leq 0 \implies -\pi \leq x \implies \arccos (\cos x) = -x\)
by (auto simp: arccos-def intro!: theI-equality cos-total)

lemma cos-arcsin:
assumes \(-1 \leq x \land x \leq 1\)
shows \(\cos (\arcsin x) = \sqrt{1 - x^2}\)
proof (rule power2-eq-imp-eq)
  show \((\cos (\arcsin x))^2 = (\sqrt{1 - x^2})^2\)
  by (simp add: square-le-1 assms cos-squared-eq)
  show \(0 \leq \cos (\arcsin x)\)
  using arcsin assms cos-ge-zero by blast
  show \(0 \leq \sqrt{1 - x^2}\)
by (simp add: square-le-1 assms)

qed

lemma sin-arccos:
  assumes $-1 \leq x \leq 1$
  shows $\sin (\arccos x) = \sqrt{1 - x^2}$
proof (rule power2-eq-imp-eq)
  show $(\sin (\arccos x))^2 = (\sqrt{1 - x^2})^2$
    by (simp add: square-le-1 assms sin-squared-eq)
  show $0 \leq \sin (\arccos x)$
    by (simp add: arccos-bounded assms sin-ge-zero)
  show $0 \leq \sqrt{1 - x^2}$
    by (simp add: square-le-1 assms)
qed

lemma arccos-0 [simp]: $\arccos 0 = \pi/2$
  by (metis arccos-cos cos-gt-zero cos-pi cos-pi-half pi-gt-zero
       pi-half-ge-zero not-le not-zero-less-neg-numeral numeral-One)

lemma arccos-1 [simp]: $\arccos 1 = 0$
  using arccos-cos by force

lemma arccos-minus-1 [simp]: $\arccos (-1) = \pi$
  by (metis arccos-cos cos-pi order-refl pi-ge-zero)

lemma arccos-minus: $-1 \leq x \Rightarrow x \leq 1 \Rightarrow \arccos (-x) = \pi - \arccos x$
  by (metis arccos-cos arccos-cos2 cos-minus-pi-total diff-le-0-iff-le le-add-same-cancel1
       minus-diff-eq uminus-add-conv-diff)

corollary arccos-minus-abs:
  assumes $|x| \leq 1$
  shows $\arccos (-x) = \pi - \arccos x$
  using assms by (simp add: arccos-minus)

lemma sin-arccos-nonzero: $-1 < x \Rightarrow x < 1 \Rightarrow \sin (\arccos x) \neq 0$
  using arccos-lt-bounded sin-gt-zero by force

lemma arctan: $-(\pi/2) < \arctan y \land \arctan y < \pi/2 \land \tan (\arctan y) = y$
  unfolding arctan-def by (rule theI' [OF tan-total])

lemma tan-arctan: $\tan (\arctan y) = y$
  by (simp add: arctan)

lemma arctan-bounded: $-(\pi/2) < \arctan y \land \arctan y < \pi/2$
  by (auto simp only: arctan)

lemma arctan-lbound: $-(\pi/2) < \arctan y$
  by (simp add: arctan)
lemma arctan-ubound: arctan y < pi/2
by (auto simp only: arctan)

lemma arctan-unique:
assumes -(pi/2) < x
and x < pi/2
and tan x = y
shows arctan y = x
using assms arctan [of y] tan-total [of y] by (fast elim: cx1E)

lemma arctan-tan: -(pi/2) < x \implies x < pi/2 \implies arctan (tan x) = x
by (rule arctan-unique) simp-all

lemma arctan-zero-zero [simp]: arctan 0 = 0
by (rule arctan-unique) simp-all

lemma arctan-minus: arctan (- x) = - arctan x
using arctan [of x] by (auto simp: arctan-unique)

lemma cos-arctan-not-zero [simp]: cos (arctan x) \neq 0
by (intro less-imp-neq [symmetric] cos-gt-zero-pi arctan-lbound arctan-ubound)

lemma cos-arctan: cos (arctan x) = 1 / sqrt (1 + x^2)
proof (rule power2-eq-imp-eq)
  have 0 < 1 + x^2 by (simp add: add-pos-nonneg)
  show 0 \leq 1 / sqrt (1 + x^2) by simp
  show 0 \leq cos (arctan x)
    by (intro less-imp-le cos-gt-zero-pi arctan-lbound arctan-ubound)
  have (cos (arctan x))^2 * (1 + (tan (arctan x))^2) = 1
    unfolding tan-def by (simp add: distrib-left power-divide)
  then show (cos (arctan x))^2 = (1 / sqrt (1 + x^2))^2
    using 0 < 1 + x^2 by (simp add: arctan power-divide eq-divide-eq)
qed

lemma sin-arctan: sin (arctan x) = x / sqrt (1 + x^2)
using add-pos-nonneg [OF zero-less-one zero-le-power2 [of x]]
using tan-arctan [of x] unfolding tan-def cos-arctan
by (simp add: eq-divide-eq)

lemma tan-sec: cos x \neq 0 \implies 1 + (tan x)^2 = (inverse (cos x))^2
for x :: 'a::{real-normed-field,banach.field}
by (simp add: add-divide-eq iff inverse-eq-divide power2-eq-square tan-def)

lemma arctan-less-iff: arctan x < arctan y \iff x < y
by (metis tan-monotone' arctan-lbound arctan-ubound tan-arctan)

lemma arctan-le-iff: arctan x \leq arctan y \iff x \leq y
by (simp only: not-less [symmetric] arctan-less-iff)
lemma arctan-eq-iff: \( \arctan x = \arctan y \iff x = y \)
  by (simp only: eq-iff [where 'a=real] arctan-le-iff)

lemma zero-less-arctan-iff [simp]: \( 0 < \arctan x \iff 0 < x \)
  using arctan-less-iff [of 0 x] by simp

lemma arctan-less-zero-iff [simp]: \( \arctan x < 0 \iff x < 0 \)
  using arctan-less-iff [of x 0] by simp

lemma zero-le-arctan-iff [simp]: \( 0 \leq \arctan x \iff 0 \leq x \)
  using arctan-le-iff [of 0 x] by simp

lemma arctan-le-zero-iff [simp]: \( \arctan x \leq 0 \iff x \leq 0 \)
  using arctan-le-iff [of x 0] by simp

lemma arctan-eq-zero-iff [simp]: \( \arctan x = 0 \iff x = 0 \)
  using arctan-eq-iff [of x 0] by simp

lemma continuous-on-arcsin': continuous-on \{-1 .. 1\} arcsin
  proof -
    have continuous-on (sin ' {-\pi/2 .. \pi/2}) arcsin
      by (rule continuous-on-inv) (auto intro: continuous-intros simp: arcsin-sin)
    also have sin ' {-\pi/2 .. \pi/2} = {-\pi/2 .. \pi/2}
      proof safe
        fix x :: real
        assume x \in \{-\pi/2 .. \pi/2\}
        then show x \in {-\pi/2 .. \pi/2}
          using arcsin-lbound arcsin-ubound
              by (intro image-eqI [where x=arcsin x]) auto
      qed simp
    finally show ?thesis .
  qed

lemma continuous-on-arccos': continuous-on \{-1 .. 1\} arccos
  proof -
    have continuous-on (cos ' {0 .. \pi}) arccos
      by (rule continuous-on-inv) (auto intro: continuous-intros simp: arccos-cos)
    also have cos ' {0 .. \pi} = {-\pi .. \pi}
      proof safe
      qed
THEORY "Transcendental"

fix x :: real
assume x ∈ {-1..1}
thен show x ∈ cos' {0..pi}
using arccos-lbound arccos-ubound
by (intro image-eqI [where x = arccos x]) auto
qed simp
finally show thesis.
qed

lemma continuous-on-arccos [continuous-intros]:
continuous-on s f ⇒ (∀ x ∈ s. -1 ≤ f x ∧ f x ≤ 1) ⇒ continuous-on s (λx. arccos (f x))
using continuous-on-compose [of s f, OF - continuous-on-subset[OF continuous-on-arccos']]
by (auto simp: comp-def subset-eq)

lemma isCont-arccos: -1 < x ⇒ x < 1 ⇒ isCont arccos x
using continuous-on-arccos"[THEN continuous-on-subset, of { -1 <..< 1 }]
by (auto simp: continuous-on-eq-continuous-at subset-eq)

lemma isCont-arctan: isCont arctan x
proof -
obtain u where u: - (pi / 2) < u u < arctan x
  by (meson arctan arctan-less-iff linordered-field-no-lb)
obtain v where v: arctan x < v v < pi / 2
  by (meson arctan-less-iff arctan-ubound linordered-field-no-ub)
have isCont arctan (tan (arctan x))
proof (rule isCont-inverse-function2 [of u arctan x v])
  show ∀z. [u ≤ z; z ≤ v] → arctan (tan z) = z
    using arctan-unique u(1) v(2) by auto
  then show ∀z. [u ≤ z; z ≤ v] → isCont tan z
    by (metis arctan cos-gt-zero-pi isCont-tan less-irrefl)
qed (use u v in auto)
then show thesis
  by (simp add: arctan)
qed

lemma tendsto-arctan [tendsto-intros]: (f ---\ x) F ⇒ ((λx. arctan (f x)) ---\ arctan x) F'
by (rule isCont-tendsto-compose [OF isCont-arccos])

lemma continuous-arctan [continuous-intros]: continuous F f ⇒ continuous F (λx. arctan (f x))
unfolding continuous-def by (rule tendsto-arctan)

lemma continuous-on-arctan [continuous-intros]:
continuous-on s f ⇒ continuous-on s (λx. arctan (f x))
unfolding continuous-on-def by (auto intro: tendsto-arctan)

lemma DERIV-arcsin:
assumes $-1 < x < 1$
shows $\text{DERIV} \arcsin x \Rightarrow \text{inverse} \left(\sqrt{1 - x^2}\right)$
proof
(rule $\text{DERIV-inverse-function}$)
show $(\sin \text{ has-real-derivative } \sqrt{1 - x^2}) \ (\text{at } \arcsin x))$
  by (rule $\text{derivative-eq-intros} \ | \ \text{use assms cos-arcsin in force})$+
show $\sqrt{1 - x^2} \neq 0$
  using $\text{abs-square-eq-1} \ \text{assms by force}$
qed (use assms $\text{isCont-arcsin in auto}$)

lemma $\text{DERIV-arccos}$:
assumes $-1 < x < 1$
shows $\text{DERIV} \arccos x \Rightarrow \text{inverse} \left(-\sqrt{1 - x^2}\right)$
proof
(rule $\text{DERIV-inverse-function}$)
show $(\cos \text{ has-real-derivative } -\sqrt{1 - x^2}) \ (\text{at } \arccos x)$
  by (rule $\text{derivative-eq-intros} \ | \ \text{use assms sin-arccos in force})$+
show $-\sqrt{1 - x^2} \neq 0$
  using $\text{abs-square-eq-1} \ \text{assms by force}$
qed (use assms $\text{isCont-arccos in auto}$)

lemma $\text{DERIV-arctan}$: $\text{DERIV} \arctan x \Rightarrow \text{inverse} \left(1 + x^2\right)$
proof
(rule $\text{DERIV-inverse-function} \ [\text{where } f=\tan \ \text{and } a=x - 1 \ \text{and } b=x + 1])$
show $(\tan \text{ has-real-derivative } 1 + x^2) \ (\text{at } \arctan x))$
  apply (rule $\text{derivative-eq-intros} \ | \ \text{simp})$+
show $1 + x^2 \neq 0$
  by (metis $\text{power-one} \ \text{sum-power2-eq-zero-iff} \ \text{zero-neq-one}$)
qed (use $\text{isCont-arctan in auto}$)

declare
$\text{DERIV-arcsin[THEN DERIV-chain2, derivative-intros]}$
$\text{DERIV-arcsin[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]}$
$\text{DERIV-arccos[THEN DERIV-chain2, derivative-intros]}$
$\text{DERIV-arccos[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]}$
$\text{DERIV-arctan[THEN DERIV-chain2, derivative-intros]}$
$\text{DERIV-arctan[THEN DERIV-chain2, unfolded has-field-derivative-def, derivative-intros]}$

lemmas $\text{has-derivative-arctan[derivative-intros]} = \text{DERIV-arctan[THEN DERIV-compose-FDERIV]}$
and $\text{has-derivative-arccos[derivative-intros]} = \text{DERIV-arccos[THEN DERIV-compose-FDERIV]}$
and $\text{has-derivative-arcsin[derivative-intros]} = \text{DERIV-arcsin[THEN DERIV-compose-FDERIV]}$

lemma $\text{filterlim-tan-at-right}$: $\text{filterlim} \tan \ (\text{at-right } (- (\pi/2)))$
  by (rule $\text{filterlim-at-bot-at-right} [\text{where } Q=\lambda x. -\pi/2 < x ∧ x < \pi/2 \ \text{and } P=\lambda x. \text{True} \ \text{and } g=\arctan])$
  (auto simp: $\text{arctan le-less eventually-at dist-real-def simp del: less-divide-eq-natural1}$
    intro!: $\text{tan-monotone exl[of -\pi/2]}$)
lemma filterlim-tan-at-left: \(\lim_{x \to \frac{\pi}{2}} \tan x = \tan \frac{\pi}{2}\)
  by (rule filterlim-at-top-at-left[where \(Q=\lambda x. -\frac{\pi}{2} < x < \frac{\pi}{2}\) and \(P=\lambda x. \text{True}\) and \(g=\arctan\)])
  (auto simp: arctan le-less eventually-at dist-real-def simp del: less-divide-eq-numeral1 intro!: tan-monotone exI[of - \frac{\pi}{2}])

lemma tendsto-arctan-at-top: \(\lim_{x \to \frac{\pi}{2}} \arctan x = \frac{\pi}{2}\)
proof (rule tendstoI)
  fix \(e\) :: real
  assume \(0 < e\)
  define \(y\) where \(y = \frac{\pi}{2} - \min(\frac{\pi}{2}, e)\)
  then have \(y > 0\) \(\land y < \frac{\pi}{2}\)
  using \(0 < e\) by auto
  show \(\lim_{x \to \frac{\pi}{2}} \text{dist}(\arctan x, \frac{\pi}{2}) < e\)
  proof (intro eventually-at-top-dense[THEN iffD2] exI allI impI)
    fix \(x\)
    assume \(\tan y < x\)
    then have \(\arctan(\tan y) < \arctan x\)
      by (simp add: arctan-less-iff)
    with \(y < \arctan x\)
    by (subst (asm) arctan-tan) simp-all
    with arctan-ubound[of \(x\), arith] \(0 < e\)
    show \(\text{dist}(\arctan x, \frac{\pi}{2}) < e\)
      by (simp add: dist-real-def)
  qed
qed

lemma tendsto-arctan-at-bot: \(\lim_{x \to -\frac{\pi}{2}} \arctan x = -\frac{\pi}{2}\)
unfolding filterlim-at-bot-mirror arctan-minus
by (intro tendsto-minus tendsto-arctan-at-top)

110.18 Prove Totality of the Trigonometric Functions

lemma cos-arccos-abs: \(|y| \leq 1 \implies \cos(\arccos y) = y\)
  by (simp add: abs-le-iff)

lemma sin-arccos-abs: \(|y| \leq 1 \implies \sin(\arccos y) = \sqrt{1 - y^2}\)
  by (simp add: sin-arccos abs-le-iff)

lemma sin-mono-less-eq:
  \(- \frac{\pi}{2} \leq x \implies x \leq \frac{\pi}{2} \implies - \frac{\pi}{2} \leq y \implies y \leq \frac{\pi}{2} \implies \sin x < \sin y\)
\(\iff x < y\)
  by (metis not-less-iff-gr-or-eq sin-monotone-2pi)

lemma sin-mono-le-eq:
  \(- \frac{\pi}{2} \leq x \implies x \leq \frac{\pi}{2} \implies - \frac{\pi}{2} \leq y \implies y \leq \frac{\pi}{2} \implies \sin x \leq \sin y\)
\(\iff x \leq y\)
  by (meson leD le-less-linear sin-monotone-2pi sin-monotone-2pi-le)
lemma \textit{sin-inj-pi}:
\[ -\left(\frac{\pi}{2}\right) \leq x \implies x \leq \frac{\pi}{2} \implies -\left(\frac{\pi}{2}\right) \leq y \implies y \leq \frac{\pi}{2} \implies \sin x = \sin y \]
\[ \implies x = y \]
\by (metis \text{arcsin-sin})

lemma \textit{arcsin-le-iff}:
\textbf{assumes} x \geq -1 \quad x \leq 1 \quad y \geq -\frac{\pi}{2} \quad y \leq \frac{\pi}{2} 
\textbf{shows} \quad \arcsin x \leq y \iff x \leq \sin y 
\textbf{proof} – 
\begin{itemize}
\item have \( \arcsin x \leq y \iff \sin (\arcsin x) \leq \sin y \)
\item using \text{arcsin-bounded[of x]} \asms \by (subst \text{sin-monotone-le-eq}) \text{auto}
\end{itemize}
\text{also from asms have} \quad \sin (\arcsin x) = x \by \text{simp}
\finally show \text{thesis}.

qed

lemma \textit{le-arcsin-iff}:
\textbf{assumes} x \geq -1 \quad x \leq 1 \quad y \geq -\frac{\pi}{2} \quad y \leq \frac{\pi}{2} 
\textbf{shows} \quad \arcsin x \geq y \iff x \geq \sin y 
\textbf{proof} – 
\begin{itemize}
\item have \( \arcsin x \geq y \iff \sin (\arcsin x) \geq \sin y \)
\item using \text{arcsin-bounded[of x]} \asms \by (subst \text{sin-monotone-le-eq}) \text{auto}
\end{itemize}
\text{also from asms have} \quad \sin (\arcsin x) = x \by \text{simp}
\finally show \text{thesis}.

qed

lemma \textit{cos-mono-less-eq}:
\[ 0 \leq x \implies x \leq \pi \implies 0 \leq y \implies y \leq \pi \implies \cos x < \cos y \iff y < x \]
\by (meson \text{cos-monotone-0-pi} \cos-monotone-0-pi-le \text{leD} \text{le-less-linear})

lemma \textit{cos-mono-le-eq}:
\[ 0 \leq x \implies x \leq \pi \implies 0 \leq y \implies y \leq \pi \implies \cos x \leq \cos y \iff y \leq x \]
\by (metis \text{arccos-cos} \cos-monotone-0-pi-le \text{eq-iff} \text{linear})

lemma \textit{cos-inj-pi}:
\[ 0 \leq x \implies x \leq \pi \implies 0 \leq y \implies y \leq \pi \implies \cos x = \cos y \]
\[ \implies x = y \]
\by (metis \text{arccos-cos})

lemma \textit{arccos-le-pi2}:
\[ [0 \leq y; y \leq 1] \implies \arccos y \leq \pi/2 \]
\by (metis (mono-claqs) \text{arccos-0} \text{arccos cos-le-one} \cos-monotone-0-pi-le 
\quad \cos-pi \cos-pi-half \pi-half-ge-zero \text{antisym conv} \text{less-eq-neg-nonpos} \text{linear minus-minus}
\quad \text{order.trans order-refl})

lemma \textit{sincos-total-pi-half}:
\textbf{assumes} 0 \leq x \quad 0 \leq y \quad x^2 + y^2 = 1 
\textbf{shows} \exists t. \quad 0 \leq t \wedge t \leq \pi/2 \wedge x = \cos t \wedge y = \sin t 
\textbf{proof} – 
\begin{itemize}
\item have \textit{x1}: \quad x \leq 1
\item using \textit{asms} \by (metis \text{le-add-cancel1} \text{power2-le-imp-le power-one zero-le-power2})
\end{itemize}
\textbf{with asms} \textit{have} \*: \quad 0 \leq \arccos x \cos (\arccos x) = x
by (auto simp: arccos)

from assms have y = sqrt (1 - x^2)
  by (metis abs-of-nonneg add.commute add-diff-cancel real-sqrt-abs)

with x1 * assms arccos-le-pi2 [of x] show ?thesis
  by (rule_tac x=arccos x in exI) (auto simp: sin-arccos)

qed

lemma sincos-total-pi:
  assumes 0 ≤ y x^2 + y^2 = 1
  shows ∃t. 0 ≤ t ∧ t ≤ pi ∧ x = cos t ∧ y = sin t

proof (cases rule: le-cases [of 0 x])
  case le
  from sincos-total-pi-half [OF le] show ?thesis
    by (metis assms le-add-same-cancel1 mult.commute mult-2-right order.trans)

next
  case ge
  then have 0 ≤ -x
    by simp
  then obtain t where t: (≥0 t ≤ pi/2 -x = cos t y = sin t
    using sincos-total-pi-half assms
  by auto (metis ⟨0 ≤ -x⟩ power2-minus)

  show ?thesis
  by (rule exI [where x = pi -t]) (use t in auto)

qed

lemma sincos-total-2pi-le:
  assumes x^2 + y^2 = 1
  shows ∃t. 0 ≤ t ∧ t ≤ 2 * pi ∧ x = cos t ∧ y = sin t

proof (cases rule: le-cases [of 0 y])
  case le
  from sincos-total-pi [OF le] show ?thesis
    by (metis assms le-add-same-cancel1 mult.commute mult-2-right order.trans)

next
  case ge
  then have 0 ≤ -y
    by simp
  then obtain t where t: (≥0 t ≤ pi x = cos t -y = sin t
    using sincos-total-pi assms
  by auto (metis ⟨0 ≤ -y⟩ power2-minus)

  show ?thesis
  by (rule exI [where x = 2 * pi - t]) (use t in auto)

qed

lemma sincos-total-2pi:
  assumes x^2 + y^2 = 1
  obtains t where 0 ≤ t t < 2*pi x = cos t y = sin t

proof
  from sincos-total-2pi-le [OF assms]
obtain $t$ where $t: 0 \leq t \leq 2 \pi \ x = \cos t \ y = \sin t$
   by blast
show $\text{thesis}$
   by (cases $t = 2 \pi$) (use $t$ that in (force+))
qed

lemma $\text{arcsin-less-mono}: |x| \leq 1 \implies |y| \leq 1 \implies \arcsin x < \arcsin y \iff x < y$
   by (rule trans [OF sin-mono-less-eq [symmetric]]) (use arcsin-ubound arcsin-lbound in auto)

lemma $\text{arcsin-le-mono}: |x| \leq 1 \implies |y| \leq 1 \implies \arcsin x \leq \arcsin y \iff x \leq y$
   using arcsin-less-mono not-le by blast

lemma $\text{arccos-less-arcsin}: -1 \leq x \implies x < y \implies y < 1 \implies \arccos x < \arcsin y$
   using arcsin-less-mono by auto

lemma $\text{arccos-le-arcsin}: -1 \leq x \implies x \leq y \implies y \leq 1 \implies \arccos x \leq \arcsin y$
   using arcsin-less-arcsin of $y$ $x$ by (simp add: not-le [symmetric])

lemma $\text{arccos-less-arccos}: -1 \leq x \implies x < y \implies y \leq 1 \implies \arccos y < \arccos x$
   using arccos-less-mono by auto

lemma $\text{arccos-le-arccos}: -1 \leq x \implies x \leq y \implies y \leq 1 \implies \arccos y \leq \arccos x$
   using arccos-less-arccos by auto

lemma $\text{arccos-eq-iff}: |x| \leq 1 \wedge |y| \leq 1 \implies \arccos x = \arccos y \iff x = y$
   using cos-arccos-abs by fastforce

lemma $\text{arccos-cos-eq-abs}$:
   assumes $|\vartheta| \leq \pi$
   shows $\arccos (\cos \vartheta) = |\vartheta|$
   unfolding arccos-def
   proof (intro the-equality conjI; clarify?)
   show $\cos |\vartheta| = \cos \vartheta$
      by (simp add: abs-real-def)
   show $x = |\vartheta|$ if $\cos x = \cos \vartheta \ 0 \leq x x \leq \pi$ for $x$
      by (simp add: $\langle$cos $|\vartheta| = \cos \vartheta$ $\rangle$ assms cos-inj-pi that)
   qed (use assms in auto)

lemma $\text{arccos-cos-eq-abs-2pi}$:
   obtains $k$ where $\arccos (\cos \vartheta) = |\vartheta - \text{of-int} \ k \ast (2 \ast \pi)|$
THEORY "Transcendental"

proof –
  define k where k ≡ \[(\vartheta + \pi) / (2 * \pi)\]
  have lepi: \(|\vartheta - \text{of-int} \ k * (2 * \pi)| \leq \pi
    using floor-divide-lower [of 2*pi \vartheta + pi] floor-divide-upper [of 2*pi \vartheta + pi]
    by (auto simp: k-def abs_if algebra-simps)
  have arccos (cos \vartheta) = arccos (cos (\vartheta - of-int \ k * (2 * \pi)))
    using cos-int-2pin sin-int-2pin
    by (simp add: cos-diff mult.commute)
  also have \(\ldots \approx |\vartheta - \text{of-int} \ k * (2 * \pi)|
    using arccos-cos-eq-abs lepi
    by blast
  finally show ?thesis
    using that by metis
  qed

lemma cos-limit-1:
  assumes \(\lambda j. \cos (\vartheta j) \longrightarrow 1\)
  shows \(\exists k. (\lambda j. \vartheta j - \text{of-int} (k \ j) * (2 * \pi)) \longrightarrow 0\)
proof –
  have \(\forall j. j \in \text{sequentially}. \cos (\vartheta j) \in \{-1..1\}\)
    by auto
  then have \(\lambda j. \arccos (\cos (\vartheta j)) \longrightarrow \arccos 1\)
    using continuous-on-tendsto-compose [OF continuous-on-arccos assms] by auto
  moreover have \(\lambda j. \arccos (\cos (\vartheta j)) = |\vartheta j - \text{of-int} (k \ j) * (2 * \pi)|\)
    by (rule tendsto-abs)
  then have \(\lambda j. \arccos (\cos (\vartheta j)) = |\vartheta j - \text{of-int} (k \ j) * (2 * \pi)|\)
    by (simp add: tendsto-rabs-zero-iff)
  ultimately have \(\lambda j. |\vartheta j - \text{of-int} (k \ j) * (2 * \pi)| \longrightarrow 0\)
    by auto
  then show ?thesis
    by (simp add: tendsto-rabs-zero-iff)
  qed

lemma cos-diff-limit-1:
  assumes \(\lambda j. \cos (\vartheta j - \Theta) \longrightarrow 1\)
  obtains k where \(\lambda j. \vartheta j - \text{of-int} (k \ j) * (2 * \pi) \longrightarrow \Theta\)
proof –
  obtain k where \(\lambda j. (\vartheta j - \Theta) - \text{of-int} (k \ j) * (2 * \pi) \longrightarrow 0\)
    using cos-limit-1 [OF assms] by auto
  then have \(\lambda j. \Theta + ((\vartheta j - \Theta) - \text{of-int} (k \ j) * (2 * \pi)) \longrightarrow \Theta + 0\)
    by (rule tendsto-add [OF tendsto-const])
  with that show ?thesis
    by auto
  qed

110.19 Machin’s formula

lemma arctan-one: \(\arctan 1 = \pi / 4\)
  by (rule arctan-unique) (simp-all add: tan-45 m2pi-less-pi)
lemma \textit{tan-total-pi4}:  
assumes \(|x| < 1\)  
shows \(\exists z. -\left(\frac{\pi}{4}\right) < z < z < \frac{\pi}{4} \land \tan z = x\)
proof
  show \(-\left(\frac{\pi}{4}\right) < \arctan x < \frac{\pi}{4} \land \tan (\arctan x) = x\)
  unfolding arctan-one [symmetric] arctan-minus [symmetric]
  unfolding arctan-less-iff
  using assms by (auto simp: arctan)
qed

lemma \textit{arctan-add}:  
assumes \(|x| \leq 1\) \(|y| < 1\)  
shows \(\arctan x + \arctan y = \arctan \left(\frac{x + y}{1 - x \cdot y}\right)\)
proof (rule arctan-unique [symmetric])
  have \(-\left(\frac{\pi}{4}\right) \leq \arctan x - \left(\frac{\pi}{4}\right) < \arctan y\)
  unfolding arctan-one [symmetric] arctan-minus [symmetric]
  unfolding arctan-le-iff arctan-less-iff
  using assms by auto
  from add-le-less-mono [OF this] show 1: \(-\left(\frac{\pi}{2}\right) < \arctan x + \arctan y\)
  by simp
  have \(\arctan x \leq \frac{\pi}{4} \land \arctan y < \frac{\pi}{4}\)
  unfolding arctan-one [symmetric]
  unfolding arctan-le-iff arctan-less-iff
  using assms by auto
  from add-le-less-mono [OF this] show 2: \(\arctan x + \arctan y < \frac{\pi}{2}\)
  by simp
  show \(\tan (\arctan x + \arctan y) = \frac{x + y}{1 - x \cdot y}\)
  using cos-gt-zero-pi [OF 1 2] by (simp add: arctan tan-add)
qed

lemma \textit{arctan-double}: \(|x| < 1 \implies 2 \cdot \arctan x = \arctan \left(\frac{2 \cdot x}{1 - x^2}\right)\)
by (metis arctan-add linear mult-2 not-less power2-eq-square)

theorem \textit{machin}: \(\pi / 4 = 4 \cdot \arctan (1 / 5) - \arctan (1 / 239)\)
proof
  have \(1 / 5 < (1 :: \text{real})\)
  by auto
  from arctan-add[OF less-imp-le[OF this] this] have 2: \(2 \cdot \arctan (1 / 5) = \arctan (5 / 12)\)
  by auto
  moreover have \(5 / 12 < (1 :: \text{real})\)
  by auto
  from arctan-add[OF less-imp-le[OF this] this] have 2: \(2 \cdot \arctan (5 / 12) = \arctan (120 / 119)\)
  by auto
  moreover have \(1 \leq (1 :: \text{real})\) and \(1 / 239 < (1 :: \text{real})\)
  by auto

from arctan-add[OF this] have arctan 1 + arctan (1 / 239) = arctan (120 / 119)
  by auto
ultimately have arctan 1 + arctan (1 / 239) = 4 * arctan (1 / 5)
  by auto
then show ?thesis
  unfolding arctan-one by algebra
qed

lemma machin-Euler: 5 * arctan (1 / 7) + 2 * arctan (3 / 79) = pi / 4
proof –
  have 17: |1 / 7| < (1 :: real) by auto
  with arctan-double have 2 * arctan (1 / 7) = arctan (7 / 24)
    by simp (simp add: field-simps)
moreover
  have [7 / 24] < (1 :: real) by auto
  with arctan-double have 2 * arctan (7 / 24) = arctan (336 / 527)
    by simp (simp add: field-simps)
moreover
  have [336 / 527] < (1 :: real) by auto
  from arctan-add[OF less-imp-le[OF 17 this]]
  have arctan(1/7) + arctan (336 / 527) = arctan (2879 / 3353)
    by auto
ultimately have I: 5 * arctan (1 / 7) = arctan (2879 / 3353) by auto
have 379: |3 / 79| < (1 :: real) by auto
with arctan-double have II: 2 * arctan (3 / 79) = arctan (237 / 3116)
  by simp (simp add: field-simps)
have *: |2879 / 3353| < (1 :: real) by auto
have |237 / 3116| < (1 :: real) by auto
from arctan-add[OF less-imp-le[OF * this]] have arctan (2879/3353) + arctan
  (237/3116) = pi/4
  by (simp add: arctan-one)
with I II show ?thesis by auto
qed

110.20 Introducing the inverse tangent power series

lemma monoseq-arctan-series:
  fixes x :: real
  assumes |x| ≤ 1
  shows monoseq (λn. 1 / real (n * 2 + 1) * x pow (n * 2 + 1))
    (is monoseq ?a)
proof (cases x = 0)
case True
  then show ?thesis by (auto simp: monoseq-def)
next
case False
  have norm x ≤ 1 and x ≤ 1 and −1 ≤ x
    using assms by auto
show monoseq ?a
proof –
  have mono: 1 / real (Suc (Suc n * 2)) * x ^ Suc (Suc n * 2) ≤
    1 / real (Suc (n * 2)) * x ^ Suc (n * 2)
  if 0 ≤ x and x ≤ 1 for n and x :: real
proof (rule mult-mono)
  show 1 / real (Suc (Suc n * 2)) ≤ 1 / real (Suc (n * 2))
    by (rule frac-le simp-all)
  show 0 ≤ 1 / real (Suc (n * 2))
    by auto
  show x ^ Suc (Suc n * 2) ≤ x ^ Suc (n * 2)
    by (rule power-decreasing simp add: ⟨0 ≤ x ⟩ ⟨x ≤ 1 ⟩)
  show 0 ≤ x ^ Suc (Suc n * 2)
    by (rule zero-le-power simp add: ⟨0 ≤ x ⟩)
  show ?thesis
proof (cases 0 ≤ x)
  case True
  then have 0 ≤ −x and −x ≤ 1
    using ⟨−1 ≤ x ⟩ by auto
  from mono[OF this] show ?thesis
    unfolding Suc-eq-plus1[symmetric] by (rule mono-SucI2)
next
  case False
  have norm x ≤ 1 and x ≤ 1 and −1 ≤ x
    using assms by auto
  show ?a −→ 0
  proof (cases |x| < 1)
  qed
  qed
  qed

lemma zeroseq-arctan-series:
  fixes x :: real
  assumes |x| ≤ 1
  shows (λn. 1 / real (n * 2 + 1) * x^(n * 2 + 1)) −→ 0
  (is ?a −→ 0)
proof (cases x = 0)
  case True
  then show ?thesis by simp
next
  case False
  have norm x ≤ 1 and x ≤ 1 and −1 ≤ x
    using assms by auto
  show ?a −→ 0
proof (cases |x| < 1)
THEORY "Transcendental"

case True
then have norm x < 1 by auto
from tendsto-mult[OF LIMSEQ-inverse-real-of-nat LIMSEQ-power-zero[OF \( \langle \text{norm } x < 1 \rangle, \text{THEN LIMSEQ-Suc} \)]
have \( \langle \lambda n. 1 / \text{real } (n + 1) * x \cdot (n + 1) \rangle \longrightarrow 0 \)
unfolding inverse-eq-divide Suc-eq-plus1 by simp
then show \(?\text{thesis}\)
using pos2 by (rule LIMSEQ-linear)

next
case False
then have x = -1 \lor x = 1
using \( \langle |x| \leq 1 \rangle \) by auto
then have n-eq: \( \forall n. x \cdot (n * 2 + 1) = x \)
unfolding One-nat-def by auto
from tendsto-mult[OF LIMSEQ-inverse-real-of-nat[THEN LIMSEQ-linear, OF pos2, unfolded inverse-eq-divide] tendsto-const[of x]]
show \(?\text{thesis}\)
unfolding n-eq Suc-eq-plus1 by auto
qed

lemma summable-arctan-series:
fixes n :: nat
assumes \(|x| \leq 1\)
sows summable \( \langle \lambda k. (-1)^k \cdot (1 / \text{real } (k*2+1) * x \cdot (k*2+1)) \rangle \)
(is summable (?c x))
by (rule summable-Leibniz(1),
rule zeroseq-arctan-series[OF assms],
rule monoseq-arctan-series[OF assms])

lemma DERIV-arctan-series:
assumes \(|x| < 1\)
sows DERIV \( \langle \lambda x'. \Sigma k. (-1)^k \cdot (1 / \text{real } (k * 2 + 1) * x' \cdot (k * 2 + 1)) \rangle \)
x :=
(\( \Sigma k. (-1)^k \cdot x'(k * 2) \))
(is DERIV ?arctan - :- ?Int)
proof –
let \(?f = \lambda n. \text{if even } n \text{ then } (-1)^{\langle n \text{ div } 2 \rangle} * 1 / \text{real } (\text{Suc } n) \text{ else } 0\)

have n-even: even n \Longrightarrow 2 * (n div 2) = n for n :: nat
by presburger
then have if-eq: \text{if } n * \text{real } (\text{Suc } n) * x' \cdot n =
(\text{if even } n \text{ then } (-1)^{\langle n \text{ div } 2 \rangle} * x''(2 * (n \text{ div } 2)) \text{ else } 0)
for n x'
by auto

have summable-Integral: summable \( \langle \lambda n. (-1)^n \cdot x'(2 * n) \rangle \) if \(|x| < 1\)
for x :: real
proof –
from that have \( x^2 < 1 \)
by (simp add: abs-square-less-1)
have summable (\( \lambda n. (-1) ^ n * (x^2) ^ n \))
by (rule summable-Leibniz(1))
(auto intro!: LIMSEQ-realpow-zero monoseq-realpow \( x^2 < 1 \); order-less-imp-le[OF \( x^2 < 1 \)])
then show ?thesis
by (simp only: power-mult)
qed

have sums-even: (sums) \( f = \sum \lambda n. \text{if even } n \text{ then } f \left( \frac{n}{2} \right) \text{ else } 0 \) for \( f : \mathbb{N} \rightarrow \mathbb{R} \)
proof -
have \( f \) sums \( x = (\lambda n. \text{if even } n \text{ then } f \left( \frac{n}{2} \right) \text{ else } 0) \) sums \( x \) for \( x : \mathbb{R} \)
proof
assume \( f \) sums \( x \)
from sums-if[OF OF sums-zero this] show (\( \lambda n. \text{if even } n \text{ then } f \left( \frac{n}{2} \right) \text{ else } 0 \)) sums \( x \)
  by auto
next
assume (\( \lambda n. \text{if even } n \text{ then } f \left( \frac{n}{2} \right) \text{ else } 0 \)) sums \( x \)
from LIMSEQ-linear[OF this] simplified sums-def pos2, simplified sum-split-even-odd[simplified mult.commute]
show \( f \) sums \( x \)
  unfolding sums-def by auto
qed
then show ?thesis ..
qed

have Int-eq: \( \sum n. \text{if } n \text{ real } (\text{Suc } n) \cdot x^n = \text{Int} \)
unfolding if-eq mult.commute[of - 2]
suminf-def sums-even[of \( \lambda n. (-1) ^ n * x ^ 2 \cdot (2 * n), \text{symmetric} \)]
by auto

have arctan-eq: \( \sum n. \text{if } n \cdot x \cdot (\text{Suc } n) = \text{arctan } x \) for \( x \)
proof -
have if-eq': \( \forall n. \text{if even } n \text{ then } (-1) ^ n \cdot (n \div 2) \cdot 1 \cdot \text{real } (\text{Suc } n) \text{ else } 0 \) * \( x \cdot \text{Suc } n = \)
  \( \text{if even } n \text{ then } (-1) ^ n \cdot (n \div 2) \cdot (1 \cdot \text{real } (2 * (n \div 2))) \cdot x \cdot \text{Suc } (2 * (n \div 2)) \text{ else } 0 \)
  using n-even by auto
have idz-eq: \( \forall n. n + 2 + 1 = \text{Suc } (2 * n) \)
  by auto
then show ?thesis
  unfolding if-eq' idz-eq suminf-def
  sums-even[of \( \lambda n. (-1) ^ n \cdot (1 \cdot \text{real } (2 * (n \div 2))) \cdot x \cdot \text{Suc } (2 * n)), \text{symmetric}]
  by auto
qed
have DERIV \((\lambda x. \sum n. ?f n \cdot x^{-n}(\text{Suc } n)) \cdot x :> (\sum n. ?f n \cdot \text{real } (\text{Suc } n) \cdot x^{-n})\)

proof (rule DERIV-power-series')
show \(x \in \{-1 <..<1\}\)
using \(|x| < 1\) by auto
show summable \((\lambda n. ?f n \cdot \text{real } (\text{Suc } n) \cdot x^{-n})\)
if \(x^{-n}\)-bounds: \(x^{-n} \in \{-1 <..<1\}\) for \(x^{-n} : \text{real}\)
proof −
from that have \(|x^{-n}| < 1\) by auto
then show \(?\text{thesis}\)
using that sums-summable sums-if \[\text{OF sums-0 [of } \lambda x. 0 \text{] summable-sums OF summable-Integral}]\by (auto simp add: if-distrib [of \(\lambda x. x \cdot y\) for \(y\)] cong: if-cong)
qed
qed
then show \(?\text{thesis}\)
by (simp only: Int-eq arctan-eq)
qed

lemma arctan-series:
assumes \(|x| \leq 1\)
shows \(\arctan x = (\sum k. (-1)^k \cdot (1 / \text{real } (k \cdot 2 + 1)) \cdot x^{-k}) (k \cdot 2 + 1))\)
(is = = suminf \((\lambda n. ?c x n)\))
proof −
let \(?c' = \lambda x n. (-1)^n \cdot x^{-n}(n \cdot 2)\)
have DERIV-arctan-suminf: DERIV \((\lambda x. \text{suminf } (?c x)) \cdot x :> (\text{suminf } (?c' x))\)
if \(0 < r \text{ and } r < 1 \text{ and } |x| < r\) for \(r x : \text{real}\)
proof (rule DERIV-arctan-series)
from that show \(|x| < 1\)
using \((r < 1) \text{ and } (|x| < r)\) by auto
qed

\{
fix \(x : \text{real}\)
assume \(|x| \leq 1\)
note summable-Leibniz[OF zeroseq-arctan-series[OF this] monoseq-arctan-series[OF this]]
\} note arctan-series-borders = this

have when-less-one: \(\arctan x = (\sum k. ?c x k)\) if \(|x| < 1\) for \(x : \text{real}\)
proof −
obtain \(r\) where \(|x| < r\) and \(r < 1\)
using dense[OF \(|x| < 1\)] by blast
then have \(0 < r \text{ and } -r < x \text{ and } x < r\) by auto
have suminf-eq-arctan-bounded: \(\text{suminf } (?c x) = \arctan x = \text{suminf } (?c a) -\)
arctan a

if \( -r < a \) and \( b < r \) and \( a < b \) and \( a \leq x \) and \( x \leq b \) for \( x \a b \)

proof

from that have \( |x| < r \) by auto

show \( \text{suminf} (\?c \, x) - \text{arctan} \, x = \text{suminf} (\?c \, a) - \text{arctan} \, a \)

proof (rule DERIV-is-const2[of \( \?a \, b \)])

show \( a < b \) and \( a \leq x \) and \( x \leq b \)

using \( (a < b) \, (a \leq x) \, (x \leq b) \) by auto

have \( \forall \, x. \, -r < x \land x < r \, \implies \, \text{DERIV} \, (\lambda \, x. \, \text{suminf} \, (\?c \, x) - \text{arctan} \, x) \, x \)

\( :> 0 \)

proof (rule allI, rule \( \text{impl} \))

fix \( x \)

assume \( -r < x \land x < r \)

then have \( |x| < r \) by auto

with \( (r < 1) \) have \( |x| < 1 \) by auto

have \( \lfloor (x^2) \rfloor < 1 \) using \( \text{abs-square-less-1} \, (|x| < 1) \) by auto

then have \( (\lambda n. \, (- \lfloor (x^2) \rfloor) \cdot n) \, \text{sums} \, ((1 \, / \, (1 - (- \lfloor (x^2) \rfloor)))) \)

unfolding \( \text{real-norm-def} [\text{symmetric}] \) by (rule \( \text{geometric-sums} \))

then have \( \lfloor (x') \rfloor \, \text{sums} \, ((1 \, / \, (1 - (- \lfloor (x^2) \rfloor)))) \)

unfolding \( \text{power-mult-def}[\text{symmetric}] \, \text{power-mult} \, \text{mult-commute}[\text{of -}] \) by auto

then have \( \text{suminf-c'-eq-geom} \, \text{inverse} \, (1 \, + \, x^2) = \text{suminf} \, (\lfloor x' \rfloor) \)

using \( \text{sums-unique} \, \text{unfolding} \, \text{inverse-\text{eq-divide}} \) by auto

have \( \text{DERIV} \, (\lambda x. \, \text{suminf} \, (\?c \, x)) \, x \, :> \, (\text{inverse} \, (1 \, + \, x^2)) \)

unfolding \( \text{suminf-c'-eq-geom} \)

by (rule \( \text{DERIV-\text{arctan}} \, \text{suminf} \, [\text{OF} \, \text{0 < r} \, \land \, r < 1 \, \land \, |x| < r]) \)

from \( \text{DERIV-\text{diff}} \, \text{[OF this DERIV-\text{arctan}] show} \, \text{DERIV} \, (\lambda x. \, \text{suminf} \, (\?c \, x) - \text{arctan} \, x) \, x \, :> \, 0 \)

by auto

qed

then have \( \text{DERIV-\text{in-rball}} \, (\forall y. \, a \leq y \land y \leq b \, \implies \, \text{DERIV} \, (\lambda x. \, \text{suminf} \, (\?c \, x) - \text{arctan} \, x) \, y \, :> \, 0 \)

using \( (-r < a) \, (b < r) \) by auto

then show \( (\forall y. \, (a < y \land y < b) \, \implies \, \text{DERIV} \, (\lambda x. \, \text{suminf} \, (\?c \, x) - \text{arctan} \, x) \, y \, :> \, 0 \)

using \( (|x| < r) \) by auto

show \( \text{continuous-on} \, \{a.\, b\} \, (\lambda x. \, \text{suminf} \, (\?c \, x) - \text{arctan} \, x) \)

using \( \text{DERIV-\text{in-rball}} \, \text{DERIV-\text{atLeastAtMost-imp-continuous-on}} \) by blast

qed

qed

have \( \text{suminf-\text{arctan-zero}} \, \text{suminf} \, (\?c \, 0) - \text{arctan} \, 0 = 0 \)

unfolding \( \text{Suc-\text{eq-plus1}[symmetric]} \, \text{power-Suc2\, mult-zero-right} \, \text{arctan-zero-zero} \)

unfolding \( \text{suminf-zero} \)

by auto

have \( \text{suminf} \, (\?c \, x) - \text{arctan} \, x = 0 \)

proof (cases \( x = 0 \))

case \( \text{True} \)
then show ?thesis
using suminf-arctan-zero by auto
next
case False
then have $0 < |x|$ and $-|x| < |x|$ by auto
have suminf (?c (− |x|)) − arctan (− |x|) = suminf (?c 0) − arctan 0
by (rule suminf-eq-arctan-bounded[where $x1=0$ and $a1=−|x|$ and $b1=|x|$, symmetric])
(simp-all only: $|x| < r$) $(-|x| < |x|)$ neg-less-iff-less
moreover
have suminf (?c x) − arctan x = suminf (?c (− |x|)) − arctan (− |x|)
by (rule suminf-eq-arctan-bounded[where $x1=x$ and $a1=−|x|$ and $b1=|x|$])
(simp-all only: $(|x| < r)$ $(-|x| < |x|)$ neg-less-iff-less)
ultimately show ?thesis
using suminf-arctan-zero by auto
qed
then show ?thesis by auto
qed

show arctan x = suminf ($\lambda n. ?c x n$)
proof (cases $|x| < 1$)
case True
then show ?thesis by (rule when-less-one)
next
case False
then have $|x| = 1$ using $|x| \leq 1$ by auto
let ?a = $\lambda x n. |1 / \text{real} (n * 2 + 1) * x^\langle n * 2 + 1 \rangle|
let ?diff = $\lambda x n. \text{arctan } x - \big(\sum_{i<n} ?c x i\big)$
have ?diff $1 n \leq ?a 1 n$ for $n :: \text{nat}$
proof –
have $0 < (1 :: \text{real})$ by auto
moreover
have ?diff $x n \leq ?a x n$ if $0 < x$ and $x < 1$ for $x :: \text{real}$
proof –
from that have $|x| \leq 1$ and $|x| < 1$
by auto
from ($0 < x$) have $0 < 1 / \text{real} (0 * 2 + 1 :: \text{nat}) * x^\langle 0 * 2 + 1 \rangle$
by auto
note bounds = mp[OF arctan-series-borders(2)][OF $|x| \leq 1$] this, unfolded
when-less-one[OF $|x| < 1$, symmetric], THEN spec
have $0 < 1 / \text{real} (n*2+1) * x^\langle n*2+1 \rangle$
by (rule mult-pos-pos) (simp-all only: zero-less-power[OF $0 < x$], auto)
then have a-pos: ?a $x n = 1 / \text{real} (n*2+1) * x^\langle n*2+1 \rangle$
by (rule abs-of-pos)
show ?thesis
proof (cases even n)
case True
then have sgn-pos: $(-1)^n = (1::real)$ by auto
from (even n) obtain m where n = 2 * m ..
then have 2 * m = n ..
from bounds[of m, unfolded this atLeastAtMost-iff]
have \(|\arctan x - (\sum i< n. (?c x i))| \leq (\sum i< n+1. (?c x i)) - (\sum i< n. (?c x i))\)
by auto
also have ... = ?c x n by auto
also have ... = ?a x n unfolding sgn-pos a-pos by auto
finally show ?thesis .
next
case False
then have sgn-neg: \((-1)^n = (-1::real)\) by auto
from (odd n) obtain m where n = 2 * m + 1 ..
then have m-def: 2 * m + 1 = n ..
then have m-plus: 2 * (m + 1) = n + 1 by auto
from bounds[of m + 1, unfolded this atLeastAtMost-iff, THEN conjunct1]
bounds[of m, unfolded m-def atLeastAtMost-iff, THEN conjunct2]
have \(|\arctan x - (\sum i< n. (?c x i))| \leq (\sum i< n. (?c x i)) - (\sum i< n+1. (?c x i))\)
by auto
also have ... = - ?c x n by auto
also have ... = ?a x n unfolding sgn-neg a-pos by auto
finally show ?thesis .
qed

hence \(\forall x \in \{ 0 <..< 1 \} . \ 0 \leq ?a x n - ?diff x n\) by auto
moreover have isCont (\(\lambda x. ?a x n - ?diff x n\)) x for x
unfolding diff-conv-add-uminus divide-inverse
by (auto intro!: isCont-add isCont-rabs continuous-ident isCont-minus isCont-arctan
continuous-at-within-inverse isCont-mult isCont-power continuous-const
isCont-sum
simp del: add-uminus-conv-diff)
ultimately have \(0 \leq ?a 1 n - ?diff 1 n\)
by (rule LIM-less-bound)
then show ?thesis by auto
qed

have \(?a 1 \rightarrow 0\)
unfolding tendsto-rabs-zero-iff power-one divide-inverse One-nat-def
by (auto intro!: tendsto-mult LIMSEQ-linear LIMSEQ-inverse-real-of-nat simp
del: of-nat-Suc)

have \(?diff 1 \rightarrow 0\)
proof (rule LIMSEQ-1)
fix r :: real
assume \(0 < r\)

obtain N :: nat where N-I: \(N \leq n \Longrightarrow ?a 1 n < r\) for n
using LIMSEQ-D[OF (?a 1 \rightarrow 0); \(0 < r\)] by auto
have norm (?diff 1 n - 0) < r if N \(\leq n\) for n
using (?diff 1 n \leq ?a 1 n) N-I[OF that] by auto
then show \(\exists N. \forall n \geq N. \ \text{norm} (\?diff 1 n - 0) < r\) by blast
THEORY "Transcendental"

qed
from this [unfolded tendsto-rabs-zero-iff, THEN tendsto-add [OF - tendsto-const],
of - arctan 1, THEN tendsto-minus]

have (?c 1) sums (arctan 1) unfolding sums-def by auto
then have arctan 1 = (∑ i. ?c 1 i) by (rule sums-unique)

show ?thesis
proof (cases x = 1)
case True
then show ?thesis by (simp add: ⟨arctan 1 = (∑ i. ?c 1 i)⟩)
next
case False
then have x = -1 using ⟨|x| = 1⟩ by auto

have - (pi/2) < 0 using pi-gt-zero by auto
have - (2 * pi) < 0 using pi-gt-zero by auto

have c-minus-minus: ?c (- 1) i = - ?c 1 i for i by auto

have arctan (- 1) = arctan (tan (- (pi / 4))) unfolding tan-45 tan-minus ..
also have ... = - (pi / 4) by (rule arctan-tan) (auto simp: order-less-trans[OF (- (pi/2) < 0) pi-gt-zero])
also have ... = - (arctan (tan (pi / 4))) unfolding neg-equal-iff-equal by (rule arctan-tan[symmetric]) (auto simp: order-less-trans[OF (- (2 * pi) < 0) pi-gt-zero])
also have ... = - (arctan 1) unfolding tan-45 ..
also have ... = - (∑ i. ?c 1 i)
using ⟨arctan 1 = (∑ i. ?c 1 i)⟩ by auto
also have ... = (∑ i. ?c (- 1) i)
using suminf-minus[OF sums-summable[OF ⟨(?c 1) sums (arctan 1)⟩]] unfolding c-minus-minus by auto
finally show ?thesis using ⟨x = -1⟩ by auto
qed
qed

lemma arctan-half: arctan x = 2 * arctan (x / (1 + sqrt(1 + x^2)))
for x :: real
proof
obtain y where low: - (pi/2) < y and high: y < pi/2 and y-eq: tan y = x
using tan-total by blast
then have low2: - (pi/2) < y and high2: y / 2 < pi/2 by auto

have 0 < cos y by (rule cos-gt-zero-pi[OF low high])
then have \( \cos y \neq 0 \) and \( \cos-sqrt: \sqrt{((\cos y)^2)} = \cos y \)
by auto

have \( 1 + (\tan y)^2 = 1 + (\sin y)^2 / (\cos y)^2 \)
unfolding tan-def power-divide ..
also have \( \ldots = (\cos y)^2 / (\cos y)^2 + (\sin y)^2 / (\cos y)^2 \)
using \( \langle \cos y \neq 0 \rangle \) by auto
also have \( \ldots = 1 / (\cos y)^2 \)
unfolding add-divide-distrib[ symmetric ] sin-cos-squared-add2 ..
finally have \( 1 + (\tan y)^2 = 1 / (\cos y)^2 \).

have \( \sin y / (\cos y + 1) = \tan y / ((\cos y + 1) / \cos y) \)
unfolding tan-def using \( \langle \cos y \neq 0 \rangle \) by simp add: field-simps
also have \( \ldots = \tan y / (1 + 1 / \cos y) \)
using \( \langle \cos y \neq 0 \rangle \) unfolding add-divide-distrib by auto
also have \( \ldots = \tan y / (1 + \sqrt{1 / (\cos y)^2}) \)
unfolding cos-sqrt ..
also have \( \ldots = \tan y / (1 + \sqrt{(1 / (\cos y)^2)}) \)
unfolding real-sqrt-divide by auto
finally have eq: \( \sin y / (\cos y + 1) = \tan y / (1 + \sqrt{(1 + (\tan y)^2)}) \)
unfolding \( 1 + (\tan y)^2 = 1 / (\cos y)^2 \).

have \( \arctan x = y \)
using \( \arctan-tan \) low high y-eq by auto
also have \( \ldots = 2 * (\arctan (\tan (y / 2))) \)
using \( \arctan-tan(OF low2 high2) \) by auto
also have \( \ldots = 2 * (\arctan (\sin y / (\cos y + 1))) \)
unfolding tan-half by auto
finally show \?thesis
unfolding eq (\( \tan y = x \)).
qued

lemma arctan-monotone: \( x < y \implies \arctan x < \arctan y \)
by (simp only: arctan-less-iff)

lemma arctan-monotone': \( x < y \implies \arctan x \leq \arctan y \)
by (simp only: arctan-le-iff)

lemma arctan-inverse:
assumes \( x \neq 0 \)
shows \( \arctan (1 / x) = \sgn x * \pi/2 - \arctan x \)
proof (rule arctan-unique)
show \( (\pi/2) < \sgn x * \pi/2 - \arctan x \)
using arctan-bounded [of x] assms
unfolding sgn-real-def
apply (auto simp: arctan algebra-simps)
apply (drule zero-less-arctan-iff [THEN iffD2], arith)
done
show \( \sgn x * \pi/2 - \arctan x < \pi/2 \)
using arctan-bounded [of \(- x\)] assms
unfolding sgn-real-def arctan-minus
by (auto simp: algebra-simps)
show \(\tan (\sgn x \ast \pi/2 - \arctan x) = 1 / x\)
unfolding tan-inverse [of arctan x, unfolded tan-arctan]
unfolding sgn-real-def
by (simp add: tan-def cos-arctan sin-arctan sin-diff cos-diff)
qed

theorem pi-series: \(\pi/4 = (\sum k \cdot (-1)^k \ast x / (k \ast 2 + 1))\)
(is - = ?SUM)
proof
  have \(\pi/4 = \arctan 1\)
  using arctan-one by auto
  also have \(\ldots = ?SUM\)
  using arctan-series [of 1] by auto
  finally show ?thesis by auto
qed

110.21 Existence of Polar Coordinates

lemma cos-x-y-le-one: \(|x / \sqrt{x^2 + y^2}| \leq 1\)
by (rule power2-le-imp-le [OF - zero-le-one])
  (simp add: power-divide divide-le-eq not-sum-power2-lt-zero)
lemmas cos-arccos-lemma1 = cos-arccos-abs [OF cos-x-y-le-one]
lemmas sin-arccos-lemma1 = sin-arccos-abs [OF cos-x-y-le-one]

lemma polar-Ex: \(\exists r::\text{real}. \exists a. x = r \ast \cos a \land y = r \ast \sin a\)
proof
  have polar-ex1: \(0 < y \Longrightarrow \exists r a. x = r \ast \cos a \land y = r \ast \sin a \) for y
  apply (rule exI [where x = \sqrt{x^2 + y^2}])
  apply (rule exI [where x = \arccos (x / \sqrt{x^2 + y^2})])
  apply (simp add: cos-arccos-lemma1 sin-arccos-lemma1 power-divide
    real-sqrt-mult [symmetric] right-diff-distrib)
  done
show ?thesis
proof (cases 0::real y rule: linorder-cases)
  case less
  then show ?thesis
  by (rule polar-ex1)
next
  case equal
  then show ?thesis
  by (force simp: intro!: cos-zero sin-zero)
next
  case greater
  with polar-ex1 [where y = \(- y\)] show ?thesis
110.22 Basics about polynomial functions: products, extremal behaviour and root counts

lemma pairs-le-eq-Sigma: \{(i, j). \ i + j \leq m\} = \text{Sigma} (\text{atMost} m) (\lambda r. \text{atMost} (m - r))
for \(m::\text{nat}\)
by auto

lemma sum-up-index-split: \((\sum_{k \leq m + n}. \ f \ k) = (\sum_{k \leq m}. \ f \ k) + (\sum k = \text{Suc} m..m + n. \ f \ k)\)
by (metis atLeast0AtMost Suc-\text{eq}\_plus1 \text{le}\_0 \text{sum\_ub}\_\text{add\_nat})

lemma Sigma-interval-disjoint: \((\text{SIGMA} i:\text{A}. \{..i\}) \cap (\text{SIGMA} i:\text{A}.\{v \leq w\})\) = \{}
for \(w::'a::'order\)
by auto

lemma product-atMost-\text{eq}\_\text{Un}: A \times \{..m\} = \text{SIGMA} i:\text{A}.\{..m - i\} \cup \text{SIGMA} i:\text{A}.\{m - i <..m\})
for \(m::\text{nat}\)
by auto

lemma polynomial-product:
fixes \(x::'a::\text{idom}\)
assumes \(m::\\{i. i > m \Longrightarrow a \ i = 0\}\)
and \(n::\\{j. j > n \Longrightarrow b \ j = 0\}\)
shows \((\sum_{i \leq m}. (a \ i) \times x \ ^{-} i) \times (\sum_{j \leq n}. (b \ j) \times x \ ^{-} j) =\)
\((\sum_{r \leq m + n}. (a \ k) \times (b (r - k))) \times x \ ^{-} r)\)
proof –
have \((\sum_{i \leq m}. (a \ i) \times x \ ^{-} i) \times (\sum_{j \leq n}. (b \ j) \times x \ ^{-} j) = (\sum i \leq m. \sum j \leq n. (a \ i \times x \ ^{-} i) \times (b \ j \times x \ ^{-} j))\)
by (rule \text{sum\_product})
also have \(\ldots = (\sum_{i \leq m + n}. (a \ i) \times x \ ^{-} i) \times (\sum_{k \leq m + n - r. (a \ r) \times x \ ^{-} r) \times (b \ j \times x \ ^{-} j))\)
using \text{assms} by (auto \text{simp}: \text{sum\_up\_index\_split})
also have \(\ldots = (\sum_{r \leq m + n}. (\sum_{k \leq m + n - r. (a \ k) \times (b (r - k))) \times x \ ^{-} r)\)
apply (simp \text{add\_ac} \text{sum\_Sigma product\_atMost\_eq\_Un})
apply (clarsimp simp \text{add\_UnSigma\_interval\_disjoint intro!}: \text{sum\_neutral})
apply (metis \text{add\_diff\_assoc2} \text{add\_commute} \text{add\_less\_D1} \text{le\_D} m n \text{nat\_le\_linear} \text{neg\_E})
done
also have \(\ldots = (\sum (i,j)\in\{(i,j). i+j \leq m+n\}. (a \ i \times x \ ^{-} i) \times (b \ j \times x \ ^{-} j))\)
by (auto \text{simp}: \text{pairs\_le\_eq\_Sigma} \text{sum\_Sigma})
also have \(\ldots = (\sum_{r \leq m + n. (\sum_{k \leq r. (a \ k) \times (b (r - k))) \times x \ ^{-} r})\)
apply (subst \text{sum\_triangle\_reindex\_eq})
apply (auto \text{simp}: \text{algebra\_simps} \text{sum\_distrib\_left intro!}: \text{sum\_cong})
apply (metis le-add-diff-inverse power-add)
done
finally show ?thesis .
qed
have \((\sum_{i=Suc\ j..n.\ a\ i\ *\ y^i(i-j-1)}) = (\sum_{k<n-j.\ a(j+k+1)\ *\ y^k})\)
if \(j < n\) for \(j :: nat\)

proof 
  have \(h\): \(bij-betw\ (\lambda i.\ i - (j + 1))\) \{Suc\ j..n\} \(\text{lessThan}\ (n-j)\)
  apply \((auto\ simp: bij-betw-def inj-on-def)\)
  apply \((\text{rule-tac}\ x=x + Suc\ j\ \text{in} image-eqI,\ auto)\)
  done
then show \(?thesis\)
by \((auto\ simp\ add: \text{bij-betw-def inj-on-def})\)

next

  case False
  have \((\exists b.\ \forall z.\ (\sum_{i\leq n.\ c\ i\ *\ z^i}) = (z - a) * (\sum_{i<n.\ b\ i\ *\ z^i}) + (\sum_{i\leq n.\ c\ i\ *\ a^i})\)
  proof \((\text{cases}\ n = 0)\)
    case True then show \(?thesis\)
    by \(simp\)
  next
    case False
    have \((\exists b.\ \forall z.\ (\sum_{i\leq n.\ c\ i\ *\ z^i}) = (z - a) * (\sum_{i<n.\ b\ i\ *\ z^i}) + (\sum_{i\leq n.\ c\ i\ *\ a^i})\)
    proof
      \(\text{by}\ (simp\ add: \text{algebra-simps})\)
    also have \(\ldots\ \longleftrightarrow\ \exists b.\ \forall z.\ (\sum_{i\leq n.\ c\ i\ *\ z^i}) - (\sum_{i\leq n.\ c\ i\ *\ a^i}) = (z - a) * (\sum_{i<n.\ b\ i\ *\ z^i})\)
      using False by \(\text{simp\ add: polyfun-diff}\)
    also have \(\ldots = True\ by\ auto\)
    finally show \(?thesis\)
    by \(simp\)
  qed

lemma polyfun-linear-factor-root:
  fixes \(a :: 'a::idom\)
  assumes \((\sum_{i\leq n.\ c\ i\ *\ a^i}) = 0\)
  obtains \(b\) where \(\forall z.\ (\sum_{i\leq n.\ c\ i\ *\ z^i}) = (z - a) * (\sum_{i<n.\ b\ i\ *\ z^i})\)
  using polyfun-linear-factor \([\text{of}\ c\ n\ a\ \text{assms}]\ by\ auto\)

lemma isCont-polynom: isCont \((\lambda w.\ \sum_{i\leq n.\ c\ i\ *\ w^i})\) \(a\)
  for \(c :: nat\ \Rightarrow \ 'a::real-normed-div-algebra\)
  by \(simp\)
lemma zero-polynom-imp-zero-coeffs:
fixes \( c \) :: \( \text{nat} \) ⇒ \( \mathbb{a} \cdot \{\text{ab-semigroup-mult,real-normed-div-algebra}\} \)
assumes \( \bigwedge w. (\sum i \leq n. c \cdot w^i) = 0 \) \( k \leq n \)
shows \( c \cdot k = 0 \)
using \( \text{assms} \)
proof (induction \( n \) arbitrary: \( c \cdot k \))
case \( 0 \)
then show \( \text{?case} \)
by simp
next
case \((\text{Suc} \ m \ c \cdot k)\)
have \( \text{simp}: c \cdot 0 = 0 \) using \( \text{Suc.prems(1)} \) [of \( 0 \)]
by simp
have \( (\sum i \leq \text{Suc} \ n. c \cdot w^i) = w \cdot (\sum i \leq n. c \cdot (\text{Suc} \ i) \cdot w^i) \) for \( w \)
proof
have \( (\sum i \leq \text{Suc} \ n. c \cdot w^i) = (\sum i \leq n. c \cdot (\text{Suc} \ i) \cdot w^\text{Suc} \ i) \)
  unfolding \( \text{Set-Interval.sum.atMost-Suc-shift} \)
by simp
also have \( \ldots = w \cdot (\sum i \leq n. c \cdot (\text{Suc} \ i) \cdot w^i) \)
by \( \text{(simp add: sum-distrib-left ac-simps)} \)
finally show \( \text{?thesis} \).
qed
then have \( w. \bigwedge w. w \neq 0 \implies (\sum i \leq n. c \cdot (\text{Suc} \ i) \cdot w^i) = 0 \)
using \( \text{Suc} \) by auto
then have \( (\lambda h. (\sum i \leq n. c \cdot (\text{Suc} \ i) \cdot h^i)) = 0 \)
  unfolding \( \text{LIM-cong} \)
by simp
then have \( (\sum i \leq n. c \cdot (\text{Suc} \ i) \cdot 0^i) = 0 \)
using \( \text{isCont-polynom} \) [of \( 0 \) \( \lambda i. c \cdot (\text{Suc} \ i) \) \( n \)] \( \text{LIM-unique} \)
by \( \text{(force simp: Limits.isCont-iff)} \)
then have \( \bigwedge w. (\sum i \leq n. c \cdot (\text{Suc} \ i) \cdot w^i) = 0 \)
using \( w \) by metis
then have \( \bigwedge i. i \leq n \implies c \cdot (\text{Suc} \ i) = 0 \)
using \( \text{Suc.IH} \) [of \( \lambda i. c \cdot (\text{Suc} \ i) \)] by blast
then show \( \text{?case using} \ (k \leq \text{Suc} \ n) \)
by \( \text{(cases \( k \)) auto} \)
qed

lemma polyfun-rootbound:
fixes \( c \) :: \( \text{nat} \) ⇒ \( \mathbb{a} \cdot \{\text{idom,real-normed-div-algebra}\} \)
assumes \( c \cdot k \neq 0 \) \( k \leq n \)
shows \( \text{finite \{z. } (\sum i \leq n. c(i) \cdot z^i) = 0 \} \land \text{card \{z. } (\sum i \leq n. c(i) \cdot z^i) = 0 \} \)
\( \leq n \)
using \( \text{assms} \)
proof (induction \( n \) arbitrary: \( c \cdot k \))
case \( 0 \)
then show \( \text{?case} \)
by simp
next
case \((\text{Suc} \ m \ c \cdot k)\)
let ?succase = ?case

show ?case

proof (cases \{z. (∑ i≤Suc m. c(i) * z^i) = 0\} = \{\})

  case True
  then show ?succase
  by simp

next

  case False
  then obtain z0 where z0: (∑ i≤Suc m. c(i) * z0^i) = 0
  by blast

  then have eq: \{z. (∑ i≤Suc m. c(i) * z^i) = 0\} = insert z0 \{z. (∑ i≤m. b i * z^i) = 0\}
  by auto

  have ¬ (∀ k≤m. b k = 0)
  proof
    assume [simp]: ∀ k≤m. b k = 0
    then have (∑ w. (∑ i≤m. b i * w^i) = 0)
    by simp

    then have (∑ w. (∑ i≤Suc m. c(i) * w^i) = 0)
    by simp

    then have (∑ k. k ≤ Suc m =⇒ c k = 0)
    using zero-polynom-imp-zero-coeffs by blast

    then show False using Suc.prems by blast
  qed

  then obtain k′ where bk′: b k′ ≠ 0 k′ ≤ m
  by blast

  show ?succase
  using Suc.IH [of b k′] bk′
  by (simp add: eq card-insert-if del: sum.atMost-Suc)

qed


lemma

fixes c :: nat ⇒ 'a::{idom,real-normed-div-algebra}

assumes c k ≠ 0 k≤n

shows polyfun-roots-finite: finite \{z. (∑ i≤n. c(i) * z^i) = 0\}
  and polyfun-roots-card: card \{z. (∑ i≤n. c(i) * z^i) = 0\} ≤ n

using polyfun-rootbound assms by auto


lemma polyfun-finite-roots:

fixes c :: nat ⇒ 'a::{idom,real-normed-div-algebra}

shows finite \{x. (∑ i≤n. c(i) * x^i) = 0\} ←→ (∃ i≤n. c i ≠ 0)
  (is ?lhs = ?rhs)

proof
  assume ?lhs


moreover have ¬ finite \{ x. (∑ i≤n. c i * x^i) = 0 \} if ∀ i≤n. c i = 0
proof –
  from that have \( ∃ x. (∑ i≤n. c i * x^i) = 0 \)
  by simp
  then show ?thesis
    using ex-new-if-finite [OF infinite-UNIV-char-0 [where 'a='a]]
    by auto
qed
ultimately show ?rhs by metis
next
assume ?rhs
with polyfun-rootbound show ?lhs by blast
qed

lemma polyfun-eq-0: (∀ x. (∑ i≤n. c i * x^i) = 0) ↔ (∀ i≤n. c i = 0)
for c :: nat ⇒ 'a::{idom,real-normed-div-algebra}
using zero-polynom-imp-zero-coeffs by auto

lemma polyfun-eq-coeffs: (∀ x. (∑ i≤n. c i * x^i) = (∑ i≤n. d i * x^i)) ↔ (∀ i≤n. c i = d i)
for c :: nat ⇒ 'a::{idom,real-normed-div-algebra}
proof –
  have (∀ x. (∑ i≤n. c i * x^i) = (∑ i≤n. d i * x^i)) ↔ (∀ x. (∑ i≤n. (c i - d i) * x^i) = 0)
    by (simp add: left-diff-distrib Groups-Big.sum-subtractf)
  also have ... ↔ (∀ i≤n. c i - d i = 0)
    by (rule polyfun-eq-0)
  finally show ?thesis
    by simp
qed

lemma polyfun-eq-const:
  fixes c :: nat ⇒ 'a::{idom,real-normed-div-algebra}
  shows (∀ x. (∑ i≤n. c i * x^i) = k) ↔ c 0 = k ∧ (∀ i ∈ {1..n}. c i = 0)
    (is ?lhs = ?rhs)
proof –
  have *: ∀ x. (∑ i≤n. (if i=0 then k else 0) * x^i) = k
    by (induct n) auto
  show ?thesis
  proof
    assume ?lhs
    with * have (∀ i≤n. c i = (if i=0 then k else 0))
      by (simp add: polyfun-eq-coeffs [symmetric])
    then show ?rhs by simp
  next
    assume ?rhs
    then show ?lhs by (induct n) auto
  qed
qed

lemma root-polyfun:
  fixes z :: 'a::idom
  assumes 1 ≤ n
  shows zⁿ = a =⇒ (∑ i≤n. (if i = 0 then −a else if i=n then 1 else 0) * zⁱ) = 0
  using assms by (cases n) (simp-all add: sum.atLeast-Suc-atMost atLeast0AtMost [symmetric])

lemma assumes SORT-CONSTRAINT('a::{idom,real-normed-div-algebra})
  and 1 ≤ n
  shows finite-roots-unity: finite {z::'a. zⁿ = 1}
  and card-roots-unity: card {z::'a. zⁿ = 1} ≤ n
  using polyfun-rootbound [of λ i. if i = 0 then −1 else if i=n then 1 else 0 n n]
  assms(2)
  by (auto simp: root-polyfun [OF assms(2)])

110.23 Hyperbolic functions

definition sinh :: 'a :: {banach, real-normed-algebra-1} ⇒ 'a where
  sinh x = (exp x − exp (−x)) / R 2

definition cosh :: 'a :: {banach, real-normed-algebra-1} ⇒ 'a where
  cosh x = (exp x + exp (−x)) / R 2

definition tanh :: 'a :: {banach, real-normed-field} ⇒ 'a where
  tanh x = sinh x / cosh x

definition arsinh :: 'a :: {banach, real-normed-algebra-1, ln} ⇒ 'a where
  arsinh x = ln (x + (x² + 1) powr of-real (1/2))

definition arcosh :: 'a :: {banach, real-normed-algebra-1, ln} ⇒ 'a where
  arcosh x = ln (x + (x² − 1) powr of-real (1/2))

definition artanh :: 'a :: {banach, real-normed-field, ln} ⇒ 'a where
  artanh x = ln ((1 + x) / (1 − x)) / 2

lemma arsinh-0 [simp]: arsinh 0 = 0
  by (simp add: arsinh-def)

lemma arcosh-1 [simp]: arcosh 1 = 0
  by (simp add: arcosh-def)

lemma artanh-0 [simp]: artanh 0 = 0
  by (simp add: artanh-def)

lemma tanh-altdef:
tanh x = (exp x - exp (-x)) / (exp x + exp (-x))

proof
  have tanh x = (2 *R sinh x) / (2 *R cosh x)
    by (simp add: tanh-def scaleR-conv-of-real)
  also have 2 *R sinh x = exp x - exp (-x)
    by (simp add: sinh-def)
  also have 2 *R cosh x = exp x + exp (-x)
    by (simp add: cosh-def)
  finally show ?thesis .
qed

lemma tanh-real-altdef: tanh (x::real) = (1 - exp (- 2 * x)) / (1 + exp (- 2 * x))
proof
  have [simp]: exp (2 * x) = exp x * exp x exp (x * 2) = exp x * exp x
    by (subst exp-add [symmetric]; simp)+
  have tanh x = (2 * exp (-x) * sinh x) / (2 * exp (-x) * cosh x)
    by (simp add: tanh-def)
  also have 2 * exp (-x) * sinh x = 1 - exp (-2*x)
    by (simp add: exp-minus field-simps sinh-def)
  also have 2 * exp (-x) * cosh x = 1 + exp (-2*x)
    by (simp add: exp-minus field-simps cosh-def)
  finally show ?thesis .
qed

lemma sinh-converges: (λn. if even n then 0 else x ^ n /R fact n) sums sinh x
proof
  have (λn. (x ^ n /R fact n - (-x) ^ n /R fact n) /R 2) sums sinh x
    unfolding sinh-def by (intro sums-scaleR-right sums-diff exp-converges)
  also have (λn. (x ^ n /R fact n - (-x) ^ n /R fact n) /R 2) =
    (λn. if even n then 0 else x ^ n /R fact n) by auto
  finally show ?thesis .
qed

lemma cosh-converges: (λn. if even n then x ^ n /R fact n else 0) sums cosh x
proof
  have (λn. (x ^ n /R fact n + (-x) ^ n /R fact n) /R 2) sums cosh x
    unfolding cosh-def by (intro sums-scaleR-right sums-add exp-converges)
  also have (λn. (x ^ n /R fact n + (-x) ^ n /R fact n) /R 2) =
    (λn. if even n then x ^ n /R fact n else 0) by auto
  finally show ?thesis .
qed

lemma sinh-0 [simp]: sinh 0 = 0
  by (simp add: sinh-def)

lemma cosh-0 [simp]: cosh 0 = 1
proof
have $cosh 0 = (1/2) \ast_R (1 + 1)$ by (simp add: cosh-def)
also have $\ldots = 1$ by (rule scaleR-half-double)
finally show ?thesis.

qed

lemma tanh-0 [simp]: $tanh 0 = 0$
by (simp add: tanh-def)

lemma sinh-minus [simp]: $sinh (-x) = -sinh x$
by (simp add: sinh-def algebra-simps)

lemma cosh-minus [simp]: $cosh (-x) = cosh x$
by (simp add: cosh-def algebra-simps)

lemma tanh-minus [simp]: $tanh (-x) = -tanh x$
by (simp add: tanh-def)

lemma sinh-ln-real: $x > 0 \Rightarrow sinh (ln x :: real) = (x - inverse x) \div 2$
by (simp add: sinh-def exp-minus)

lemma cosh-ln-real: $x > 0 \Rightarrow cosh (ln x :: real) = (x + inverse x) \div 2$
by (simp add: cosh-def exp-minus)

lemma tanh-ln-real:
$tanh (ln x :: real) = (x ^ 2 - 1) \div (x ^ 2 + 1)$ if $x > 0$
proof
from that have $(x \ast 2 - inverse x \ast 2) \ast (x^2 + 1) = (x^2 - 1) \ast (2 \ast x + 2 \ast inverse x)$
by (simp add: field-simps power2-eq-square)
moreover have $x^2 + 1 > 0$
using that by (simp add: ac-simps add-pos-nonneg)
moreover have $2 \ast x + 2 \ast inverse x > 0$
using that by (simp add: add-pos-pos)
ultimately have $(x \ast 2 - inverse x \ast 2) \div (2 \ast x + 2 \ast inverse x) = (x^2 - 1) \div (x^2 + 1)$
by (simp add: frac-eq-eq)
with that show ?thesis
by (simp add: tanh-def sinh-ln-real cosh-ln-real)
qed

lemma has-field-derivative-scaleR-right [derivative-intros]:
$(f has-field-derivative D) F \Rightarrow ((\lambda x. c \ast_R f x) has-field-derivative (c \ast_R D)) F$
unfolding has-field-derivative-def
using has-derivative-scaleR-right[of f \lambda x. D \ast x F c]
by (simp add: mult-scaleR-left [symmetric] del: mult-scaleR-left)

lemma has-field-derivative-sinh [THEN DERIV-chain2, derivative-intros]:
$(sinh has-field-derivative \cosh x) (at (x :: 'a :: {banach, real-normed-field})))$
THEORY “Transcendental” 2243

unfolding sinh-def cosh-def by (auto intro!: derivative-eq-intros)

lemma has-field-derivative-cosh [THEN DERIV-chain2, derivative-intros]:
(cosh has-field-derivative sinh x) (at (x :: 'a :: {banach, real-normed-field}))

unfolding sinh-def cosh-def by (auto intro!: derivative-eq-intros)

lemma has-field-derivative-tanh [THEN DERIV-chain2, derivative-intros]:
cosh x ≠ 0 ⨯ (tanh has-field-derivative 1 − tanh x · 2)
(at (x :: 'a :: {banach, real-normed-field}))

unfolding tanh-def by (auto intro!: derivative-eq-intros simp: power2-eq-square field-split-simps)

lemma has-derivative-sinh [derivative-intros]:
fixes g :: 'a ⇒ ('a :: {banach, real-normed-field})
assumes (g has-derivative (λx. Db * x)) (at x within s)
shows ((λx. sinh (g x)) has-derivative (λy. (cosh (g x) * Db) * y)) (at x within s)
proof –
have ((λx. − g x) has-derivative (λy. −(Db * y))) (at x within s)
using assms by (intro derivative-intros)
also have (λy. −(Db * y)) = (λx. −(Db) * x) by (simp add: fun-eq-iff)
finally have ((λx. sinh (g x)) has-derivative
(λy. (exp (g x) * Db * y − exp (−g x) * (−Db) * y) / R 2)) (at x within s)
unfolding sinh-def by (intro derivative-intros assms)
also have (λy. (exp (g x) * Db * y − exp (−g x) * (−Db) * y) / R 2) = (λy.
(cosh (g x) * Db) * y)
by (simp add: fun-eq-iff cosh-def algebra-simps)
finally show ?thesis .
qed

lemma has-derivative-cosh [derivative-intros]:
fixes g :: 'a ⇒ ('a :: {banach, real-normed-field})
assumes (g has-derivative (λy. Db * y)) (at x within s)
shows ((λx. cosh (g x)) has-derivative (λy. (sinh (g x) * Db) * y)) (at x within s)
proof –
have ((λx. − g x) has-derivative (λy. −(Db * y))) (at x within s)
using assms by (intro derivative-intros)
also have (λy. −(Db * y)) = (λy. −(Db) * y) by (simp add: fun-eq-iff)
finally have ((λx. cosh (g x)) has-derivative
(λy. (exp (g x) * Db * y + exp (−g x) * (−Db) * y) / R 2)) (at x within s)
unfolding cosh-def by (intro derivative-intros assms)
also have (λy. (exp (g x) * Db * y + exp (−g x) * (−Db) * y) / R 2) = (λy.
(sinh (g x) * Db) * y)
by (simp add: fun-eq-iff sinh-def algebra-simps)
finally show ?thesis .
qed

lemma sinh-plus-cosh: sinh x + cosh x = exp x
proof
  have \( \sinh x + \cosh x = (1/2) \ast_R (\exp x + \exp x) \)
    by (simp add: sinh-def cosh-def algebra-simps)
  also have \( \ldots = \exp x \) by (rule scaleR-half-double)
  finally show \(?thesis\).
qed

lemma cosh-plus-sinh: \( \cosh x + \sinh x = \exp x \)
  by (subst add.commute) (rule sinh-plus-cosh)

lemma cosh-minus-sinh: \( \cosh x - \sinh x = \exp (-x) \)
proof
  have \( \cosh x - \sinh x = (1/2) \ast_R (\exp (-x) + \exp (-x)) \)
    by (simp add: sinh-def cosh-def algebra-simps)
  also have \( \ldots = \exp (-x) \) by (rule scaleR-half-double)
  finally show \(?thesis\).
qed

lemma sinh-minus-cosh: \( \sinh x - \cosh x = -\exp (-x) \)
using cosh-minus-sinh[of \( x \)] by (simp add: algebra-simps)

context
  fixes \( x :: \{\text{real-normed-field, banach}\} \)
begin

lemma sinh-zero-iff: \( \sinh x = 0 \iff \exp x \in \{1, -1\} \)
  by (auto simp: sinh-def field-simps exp-minus power2-eq-square square-eq-1-iff)

lemma cosh-zero-iff: \( \cosh x = 0 \iff \exp x ^ 2 = -1 \)
  by (auto simp: cosh-def exp-minus field-simps power2-eq-square eq-neg-iff-add-eq-0)

lemma cosh-square-eq: \( \cosh x ^ 2 = \sinh x ^ 2 + 1 \)
  by (simp add: cosh-def sinh-def algebra-simps power2-eq-square exp-add [symmetric]
    scaleR-conv-of-real)

lemma sinh-square-eq: \( \sinh x ^ 2 = \cosh x ^ 2 - 1 \)
  by (simp add: cosh-square-eq)

lemma hyperbolic-pythagoras: \( \cosh x ^ 2 - \sinh x ^ 2 = 1 \)
  by (simp add: cosh-square-eq)

lemma sinh-add: \( \sinh (x + y) = \sinh x \ast \cosh y + \cosh x \ast \sinh y \)
  by (simp add: sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add [symmetric])

lemma sinh-diff: \( \sinh (x - y) = \sinh x \ast \cosh y - \cosh x \ast \sinh y \)
  by (simp add: sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add [symmetric])

lemma cosh-add: \( \cosh (x + y) = \cosh x \ast \cosh y + \sinh x \ast \sinh y \)
THEORY “Transcendental”

by (simp add: sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add [symmetric])

lemma cosh-diff: \[\cosh(x - y) = \cosh x \ast \cosh y - \sinh x \ast \sinh y\]
by (simp add: sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add [symmetric])

lemma tanh-add:
\[\tanh(x + y) = (\tanh x + \tanh y) / (1 + \tanh x \ast \tanh y)\]
if \(\cosh x \neq 0\) cosh \(y \neq 0\)
proof –
have \((\sinh x \ast \cosh y + \cosh x \ast \sinh y) \ast (1 + \sinh x \ast \sinh y / (\cosh x \ast \cosh y)) =\)
\((\cosh x \ast \cosh y + \sinh x \ast \sinh y) \ast ((\sinh x \ast \cosh y + \sinh y \ast \cosh x) / (\cosh y \ast \cosh x))\)
using that by (simp add: field-split-simps)
also have \((\sinh x \ast \cosh y + \sinh y \ast \cosh x) / (\cosh y \ast \cosh x) = \sinh x / \cosh x + \sinh y / \cosh y\)
using that by (simp add: field-split-simps)
finally have \((\sinh x \ast \cosh y + \cosh x \ast \sinh y) \ast (1 + \sinh x \ast \sinh y / (\cosh x \ast \cosh y)) =\)
\((\sinh x / \cosh x + \sinh y / \cosh y) \ast (\cosh x \ast \cosh y + \sinh x \ast \sinh y)\)
by simp
then show \(?\thesis\)
using that by (auto simp add: tanh-def sinh-add cosh-add eq-divide-eq)
(simp-all add: field-split-simps)
qed

lemma sinh-double: \[\sinh(2 \ast x) = 2 \ast \sinh x \ast \cosh x\]
using sinh-add[of \(x\)] by simp

lemma cosh-double: \[\cosh(2 \ast x) = \cosh x \ast 2 + \sinh x \ast 2\]
using cosh-add[of \(x\)] by (simp add: power2-eq-square)

end

lemma sinh-field-def: \[\sinh z = (\exp z - \exp (-z)) / (2 :: 'a :: \{banach, real-normed-field\})\]
by (simp add: sinh-def scaleR-conv-of-real)

lemma cosh-field-def: \[\cosh z = (\exp z + \exp (-z)) / (2 :: 'a :: \{banach, real-normed-field\})\]
by (simp add: cosh-def scaleR-conv-of-real)

110.23.1 More specific properties of the real functions

lemma sinh-real-zero-iff [simp]: \(\sinh (x::real) = 0 \iff x = 0\)
proof –
have \((-1 :: real) < 0\) by simp
also have \(0 < \exp x\) by simp
finally have \(\exp x \neq -1\) by (intro notI) simp
thus \(?\thesis\) by (subst sinh-zero-iff) simp
qed
lemma plus-inverse-ge-2:
  fixes x :: real
  assumes x > 0
  shows x + inverse x ≥ 2
proof -
  have 0 ≤ (x - 1) ^ 2 by simp
  also have ... = x^2 - 2*x + 1 by (simp add: power2-eq-square algebra-simps)
  finally show ?thesis using assms by (simp add: field-simps power2-eq-square)
qed

lemma sinh-real-nonneg-iff [simp]: sinh (x :: real) ≥ 0 ←→ x ≥ 0
  by (simp add: sinh-def)

lemma sinh-real-pos-iff [simp]: sinh (x :: real) > 0 ←→ x > 0
  by (simp add: sinh-def)

lemma sinh-real-nonpos-iff [simp]: sinh (x :: real) ≤ 0 ←→ x ≤ 0
  by (simp add: sinh-def)

lemma sinh-real-neg-iff [simp]: sinh (x :: real) < 0 ←→ x < 0
  by (simp add: sinh-def)

lemma cosh-real-ge-1: cosh (x :: real) ≥ 1
  using plus-inverse-ge-2[of exp x] by (simp add: cosh-def exp-minus)

lemma cosh-real-pos [simp]: cosh (x :: real) > 0
  using cosh-real-ge-1[of x] by simp

lemma cosh-real-nonneg [simp]: cosh (x :: real) ≥ 0
  using cosh-real-ge-1[of x] by simp

lemma cosh-real-nonzero [simp]: cosh (x :: real) ≠ 0
  using cosh-real-ge-1[of x] by simp

lemma tanh-real-nonneg-iff [simp]: tanh (x :: real) ≥ 0 ←→ x ≥ 0
  by (simp add: tanh-def field-simps)

lemma tanh-real-pos-iff [simp]: tanh (x :: real) > 0 ←→ x > 0
  by (simp add: tanh-def field-simps)

lemma tanh-real-nonpos-iff [simp]: tanh (x :: real) ≤ 0 ←→ x ≤ 0
  by (simp add: tanh-def field-simps)

lemma tanh-real-neg-iff [simp]: tanh (x :: real) < 0 ←→ x < 0
  by (simp add: tanh-def field-simps)

lemma tanh-real-zero-iff [simp]: tanh (x :: real) = 0 ←→ x = 0
  by (simp add: tanh-def field-simps)
lemma arsinh-real-def: \( \text{arsinh} (x \colon \text{real}) = \ln (x + \sqrt{x^2 + 1}) \)
by (simp add: arsinh-def powr-half-sqrt)

lemma arccosh-real-def: \( x \geq 1 \implies \text{arccosh} (x \colon \text{real}) = \ln (x + \sqrt{x^2 - 1}) \)
by (simp add: arccosh-def powr-half-sqrt)

lemma arsinh-real-aux: \( 0 < x + \sqrt{x^2 + 1} \colon \text{real} \)
proof (cases \( x < 0 \))
case True
have \( -x ^2 = x ^2 \) by simp
also have \( x ^2 < x ^2 + 1 \) by simp
finally have \( \sqrt{(-x) ^2} < \sqrt{x ^2 + 1} \)
by (rule real-sqrt-less-mono)
thus \?thesis using True by simp
qed (auto simp: add-nonneg-pos)

lemma arsinh-minus-real [simp]: \( \text{arsinh} (-x \colon \text{real}) = -\text{arsinh} x \)
proof
have \( \text{arsinh} (-x) = \ln (\sqrt{x^2 + 1} - x) \)
by (simp add: arsinh-real-def)
also have \( \sqrt{x^2 + 1} - x = \text{inverse} (\sqrt{x^2 + 1} + x) \)
using arsinh-real-aux[of \( x \)] by (simp add: field-split-simps algebra-simps power2-eq-square)
also have \( \ln \ldots = -\text{arsinh} x \)
using arsinh-real-aux[of \( x \)] by (simp add: arsinh-real-def ln-inverse)
finally show \?thesis .
qed

lemma artanh-minus-real [simp]:
assumes abs \( x \) < 1
shows \( \text{artanh} (-x \colon \text{real}) = -\text{artanh} x \)
using assms by (simp add: artanh-def ln-div field-simps)

lemma sinh-less-cosh-real: \( \text{sinh} (x \colon \text{real}) < \text{cosh} x \)
by (simp add: sinh-def cosh-def)

lemma sinh-le-cosh-real: \( \text{sinh} (x \colon \text{real}) \leq \text{cosh} x \)
by (simp add: sinh-def cosh-def)

lemma tanh-real-lt-1: \( \text{tanh} (x \colon \text{real}) < 1 \)
by (simp add: tanh-def sinh-less-cosh-real)

lemma tanh-real-gt-neg1: \( \text{tanh} (x \colon \text{real}) > -1 \)
proof
have \( -\text{cosh} x < \text{sinh} x \) by (simp add: sinh-def cosh-def field-split-simps)
thus \?thesis by (simp add: tanh-def field-simps)
qed

lemma tanh-real-bounds: \( \text{tanh} (x \colon \text{real}) \in \{-1<..<1\} \)
using \( \text{tanh-real-lt-1} \ text{tanh-real-gt-neg1} \) by simp

context
fixes \( x \) :: real
begin

lemma \( \text{arsinh-sinh-real} \): \( \text{arsinh} (\sinh x) = x \) by (simp add: \( \text{arsinh-real-def} \ text{powr-def} \ text{sinh-square-eq} \ text{sinh-plus-cosh} \))

lemma \( \text{arcosh-cosh-real} \): \( x \geq 0 \Rightarrow \text{arcosh} (\cosh x) = x \) by (simp add: \( \text{arcosh-real-def} \ text{powr-def} \ text{cosh-square-eq} \ text{cosh-real-ge-1} \ text{cosh-plus-sinh} \))

lemma \( \text{artanh-tanh-real} \): \( \text{artanh} (\tanh x) = x \)
proof
  have \( \text{artanh} (\tanh x) = \ln (\cosh x * (\cosh x + \sinh x) / (\cosh x * (\cosh x - \sinh x))) / 2 \) by (simp add: \( \text{artanh-def} \ text{tanh-def} \ text{field-split-simps} \))
  also have \( \cosh x * (\cosh x + \sinh x) / (\cosh x * (\cosh x - \sinh x)) = (\cosh x + \sinh x) / (\cosh x - \sinh x) \) by simp
  also have \( \ldots = (\exp x)^2 \)
by (simp add: \( \text{cosh-plus-sinh} \ text{cosh-minus-sinh} \ exp-minus \ text{field-simps} \ text{power2-eq-square} \))
  also have \( \ln ((\exp x)^2) / 2 = x \) by (simp add: \( \text{ln-realpow} \))
  finally show \( \text{thesis} \).
qed

end

lemma \( \text{sinh-real-strict-mono} \): \text{strict-mono} (\( \text{sinh} :: \text{real} \Rightarrow \text{real} \)) by (rule pos-deriv-imp-strict-mono derivative-intros)

lemma \( \text{cosh-real-strict-mono} \):
assumes \( 0 \leq x \) and \( x < (y :: \text{real}) \)
shows \( \cosh x < \cosh y \)
proof
  from \( \text{assms} \) have \( \exists z > x. \ z < y \land \cosh y - \cosh x = (y - x) * \sinh z \)
  by (intro \( \text{MVT2} \) (auto dest: \( \text{connectedD-interval} \) \( \text{intro} !: \text{derivative-eq-intros} \)))
  then obtain \( z \) where \( z > x \) \( z < y \) \( \cosh y - \cosh x = (y - x) * \sinh z \) by blast
  note \( \cosh y - \cosh x = (y - x) * \sinh z \)
  also from \( z > x \) and \( \text{assms} \) have \( (y - x) * \sinh z > 0 \) by (intro \( \text{mult-pos-pos} \))
  auto
  finally show \( \cosh x < \cosh y \) by simp
qed

lemma \( \text{tanh-real-strict-mono} \): \text{strict-mono} (\( \text{tanh} :: \text{real} \Rightarrow \text{real} \))
proof
  have \( \text{tanh x} ^ 2 < 1 \) for \( x :: \text{real} \)
  using \( \text{tanh-real-bounds[of x]} \) by (simp add: \( \text{abs-square-less-1} \ text{abs-if} \))
  show \( \text{thesis} \)

qed
by (rule pos-deriv-imp-strict-mono) (insert *, auto intro!: derivative-intros)

qed

lemma sinh-real-abs [simp]: sinh (abs x :: real) = abs (sinh x)
  by (simp add: abs-if)

lemma cosh-real-abs [simp]: cosh (abs x :: real) = cosh x
  by (simp add: abs-if)

lemma tanh-real-abs [simp]: tanh (abs x :: real) = abs (tanh x)
  by (auto simp: abs-if)

lemma sinh-real-eq-iff [simp]: sinh x = sinh y ←→ x = (y :: real)
proof
  have cosh x = cosh y ⟷ x = y if x ≥ 0 y ≥ 0 for x y :: real
    using cosh-real-strict-mono[of x y] cosh-real-strict-mono[of y x] that
    by (cases x y rule: linorder-cases) auto
  from this[of abs x abs y] show ?thesis by simp
qed

lemma sinh-real-le-iff [simp]: sinh x ≤ sinh y ←→ x ≤ (y :: real)
using sinh-real-strict-mono by (simp add: strict-mono-less-eq)

lemma cosh-real-nonneg-le-iff: x ≥ 0 ⟹ y ≥ 0 ⟹ cosh x ≤ cosh y ⟷ x ≤ (y :: real)
  using cosh-real-strict-mono[of x y] cosh-real-strict-mono[of y x]
  by (cases x y rule: linorder-cases) auto

lemma cosh-real-nonpos-le-iff: x ≤ 0 ⟹ y ≤ 0 ⟹ cosh x ≤ cosh y ⟷ x ≥ (y :: real)
  using cosh-real-nonneg-le-iff[of −x −y] by simp

lemma tanh-real-le-iff [simp]: tanh x ≤ tanh y ⟷ x ≤ (y :: real)
using tanh-real-strict-mono by (simp add: strict-mono-less-eq)

lemma sinh-real-less-iff [simp]: sinh x < sinh y ⟷ x < (y :: real)
using sinh-real-strict-mono by (simp add: strict-mono-less)

lemma cosh-real-nonneg-less-iff: x ≥ 0 ⟹ y ≥ 0 ⟹ cosh x < cosh y ⟷ x < (y :: real)
  using cosh-real-strict-mono[of x y] cosh-real-strict-mono[of y x]
  by (cases x y rule: linorder-cases) auto
lemma \( \text{cosh-real-nonpos-less-iff} \): \( x \leq 0 \implies y \leq 0 \implies \cosh x < \cosh y \iff x > y \) (\( y::\text{real} \))
using \( \text{cosh-real-nonneg-less-iff[of } -x -y \) by simp

lemma \( \text{tanh-real-less-iff} \) [simp]: \( \tanh x < \tanh y \iff x < (y::\text{real}) \)
using \( \text{tanh-real-strict-mono} \) by (simp add: strict-mono-less)

110.23.2 Limits

lemma \( \text{sinh-real-at-top} \): \( \text{filterlim (sinh :: real } \Rightarrow \text{ real) at-top at-top} \)
proof –
  have \( \ast \): \( ((\lambda x. - \exp (-x)) \to (-0::\text{real})) \) at-top
  by (intro tendsto-minus filterlim-compose[OF exp-at-bot filterlim-uminus-at-bot-at-top])
  have \( \text{filterlim (Ax. (1 / 2) \ast (-exp (-x) + exp x) :: real) at-top at-top} \)
  by (rule filterlim-tendsto-pos-mult-at-top[OF \( \ast \)] tendsto-const)+ (auto simp: exp-at-top)
  also have \( \lambda x. (1 / 2) \ast (-exp (-x) + exp x) :: real \) = sinh
  by (simp add: fun-eq-iff sinh-def)
finally show \( \text{thesis} \).
qed

lemma \( \text{sinh-real-at-bot} \): \( \text{filterlim (sinh :: real } \Rightarrow \text{ real) at-bot at-bot} \)
proof –
  have \( \text{filterlim (Ax. -sinh x :: real) at-bot at-top} \)
  by (simp add: filterlim-uminus-at-top [symmetric] sinh-real-at-top)
  also have \( \lambda x. -\sinh x :: real = (\lambda x. \sinh ( -x)) \) by simp
finally show \( \text{thesis} \)
by (subst filterlim-at-bot-mirror)
qed

lemma \( \text{cosh-real-at-top} \): \( \text{filterlim (cosh :: real } \Rightarrow \text{ real) at-top at-top} \)
proof –
  have \( \ast \): \( ((\lambda x. \exp (-x)) \to (0::\text{real})) \) at-top
  by (intro filterlim-compose[OF exp-at-bot filterlim-uminus-at-bot-at-top])
  have \( \text{filterlim (Ax. (1 / 2) \ast (exp (-x) + exp x) :: real) at-top at-top} \)
  by (rule filterlim-tendsto-pos-mult-at-top[OF \( \ast \)] tendsto-const)+ (auto simp: exp-at-top)
  also have \( \lambda x. (1 / 2) \ast (exp (-x) + exp x) :: real \) = cosh
  by (simp add: fun-eq-iff cosh-def)
finally show \( \text{thesis} \).
qed

lemma \( \text{cosh-real-at-bot} \): \( \text{filterlim (cosh :: real } \Rightarrow \text{ real) at-top at-bot} \)
proof –
  have \( \text{filterlim (Ax. \cosh (-x) :: real) at-top at-top} \)
  by (simp add: cosh-real-at-top)
thus \( \text{thesis} \)
by (subst filterlim-at-bot-mirror)
THEORY "Transcendental"

qed

lemma tanh-real-at-top: (tanh ----> (1::real)) at-top
proof -
  have ((λx::real. (1 - exp (- 2 * x)) / (1 + exp (- 2 * x))) ----> (1 - 0) / (1 + 0)) at-top
    by (intro tendsto-intros filterlim-compose[OF exp-at-bot]
        filterlim-tendsto-neg-mult-at-bot[OF tendsto-const] filterlim-ident) auto
  also have (λx::real. (1 - exp (- 2 * x)) / (1 + exp (- 2 * x))) = tanh
    by (rule ext) (simp add: tanh-real-altdef)
  finally show ?thesis by simp
qed

lemma tanh-real-at-bot: (tanh ----> (-1::real)) at-bot
proof -
  have ((λx::real. -tanh x) ----> -1) at-top
    by (intro tendsto-minus tanh-real-at-top)
  also have (λx. -tanh x :: real) = (λx. tanh (-x)) by simp
  finally show ?thesis by (subst filterlim-at-bot-mirror)
qed

110.23.3 Properties of the inverse hyperbolic functions

lemma isCont-sinh: isCont sinh (x :: 'a :: {real-normed-field, banach})
  unfolding sinh-def [abs-def] by (auto intro!: continuous-intros)

lemma isCont-cosh: isCont cosh (x :: 'a :: {real-normed-field, banach})
  unfolding cosh-def [abs-def] by (auto intro!: continuous-intros)

lemma isCont-tanh: cosh x ≠ 0 ----> isCont tanh (x :: 'a :: {real-normed-field, banach})
  unfolding tanh-def [abs-def]
  by (auto intro!: continuous-intros isCont-divide isCont-sinh isCont-cosh)

lemma continuous-on-sinh [continuous-intros]:
  fixes f :: - ⇒ 'a::{real-normed-field,banach}
  assumes continuous-on A f
  shows continuous-on A (λx. sinh (f x))
  unfolding sinh-def using assms by (intro continuous-intros)

lemma continuous-on-cosh [continuous-intros]:
  fixes f :: - ⇒ 'a::{real-normed-field,banach}
  assumes continuous-on A f
  shows continuous-on A (λx. cosh (f x))
  unfolding cosh-def using assms by (intro continuous-intros)

lemma continuous-sinh [continuous-intros]:
  fixes f :: - ⇒ 'a::{real-normed-field,banach}
  assumes continuous F f
shows \( \text{continuous } F (\lambda x. \sinh(f x)) \)

unfolding \( \sinh\)-def using assms by (intro continuous-intros)

**lemma** continuous-cosh [continuous-intros]:
fixes \( f :: - \Rightarrow 'a :: \{\text{real-normed-field,banach}\} \)
assumes \( \text{continuous } F f \)
shows \( \text{continuous } F (\lambda x. \cosh(f x)) \)
unfolding \( \cosh\)-def using assms by (intro continuous-intros)

**lemma** continuous-on-tanh [continuous-intros]:
fixes \( f :: - \Rightarrow 'a :: \{\text{real-normed-field,banach}\} \)
assumes \( \text{continuous-on } A f \land x \in A \implies \cosh(f x) \neq 0 \)
shows \( \text{continuous-on } A (\lambda x. \tanh(f x)) \)
unfolding \( \tanh\)-def using assms by (intro continuous-intros continuous-divide) auto

**lemma** continuous-at-within-tanh [continuous-intros]:
fixes \( f :: - \Rightarrow 'a :: \{\text{real-normed-field,banach}\} \)
assumes \( \text{continuous (at } x \text{ within } A) f \cosh(f x) \neq 0 \)
shows \( \text{continuous (at } x \text{ within } A) (\lambda x. \tanh(f x)) \)
unfolding \( \tanh\)-def using assms by (intro continuous-intros continuous-divide) auto

**lemma** tendsto-sinh [tendsto-intros]:
fixes \( f :: - \Rightarrow 'a :: \{\text{real-normed-field,banach}\} \)
shows \( (f \rightarrow a) F \implies ((\lambda x. \sinh(f x)) \rightarrow \sinh a) F \)
by (rule isCont-tendsto-compose [OF isCont-sinh])

**lemma** tendsto-cosh [tendsto-intros]:
fixes \( f :: - \Rightarrow 'a :: \{\text{real-normed-field,banach}\} \)
shows \( (f \rightarrow a) F \implies ((\lambda x. \cosh(f x)) \rightarrow \cosh a) F \)
by (rule isCont-tendsto-compose [OF isCont-cosh])

**lemma** tendsto-tanh [tendsto-intros]:
fixes \( f :: - \Rightarrow 'a :: \{\text{real-normed-field,banach}\} \)
shows \( (f \rightarrow a) F \implies (\lambda x. \tanh(f x)) \rightarrow \tanh a) F \)
by (rule isCont-tendsto-compose [OF isCont-tanh])

**lemma** arsinh-real-has-field-derivative [derivative-intros]:
fixes \( x :: \text{real} \)
shows \( (\text{arsinh has-field-derivative } (1 / (\sqrt{x^2 + 1}))) \text{ (at } x \text{ within } A) \)
proof –
THEORY “Transcendental”

have pos: \(1 + x \cdot 2 > 0\) by (intro add-pos-nonneg) auto
from pos arsinh-real-aux[of x] show ?thesis unfolding arsinh-def [abs-def]
  by (auto intro!: derivative-eq-intros simp: powr-minus powr-half-sqrt field-split-simps)
qed

lemma arcosh-real-has-field-derivative [derivative-intros]:
  fixes x :: real
  assumes x > 1
  shows \(\text{arcosh has-field-derivative} (1 / (\sqrt{x^2 - 1}))\) (at x within A)
proof –
  from assms have \(x + \sqrt{x^2 - 1} > 0\) by (simp add: add-pos-pos)
  thus ?thesis using assms unfolding arcosh-def [abs-def]
  by (auto intro!: derivative-eq-intros simp: powr-minus powr-half-sqrt field-split-simps power2-eq-1-iff)
qed

lemma artanh-real-has-field-derivative [derivative-intros]:
  (artanh has-field-derivative \((1 / (1 - x^2))\)) (at x within A) if |x| < 1 for x :: real
proof –
  from that have \(-1 < x < 1\) by linarith+
  hence \(\text{artanh has-field-derivative} \((4 - 4 \cdot x) / ((1 + x) \cdot (1 - x) \cdot (1 - x) \cdot 4)\)\) (at x within A) unfolding artanh-def [abs-def]
  by (auto intro!: derivative-eq-intros simp: powr-minus powr-half-sqrt)
  also have \((4 - 4 \cdot x) / ((1 + x) \cdot (1 - x) \cdot (1 - x) \cdot 4) = 1 / ((1 + x) \cdot (1 - x))\)
    using \((-1 < x) \land (x < 1)\) by (simp add: frac-eq-eq)
  also have \((1 + x) \cdot (1 - x) = 1 - x^2\)
    by (simp add: algebra-simps power2-eq-square)
  finally show ?thesis
qed

lemma continuous-on-arsinh [continuous-intros]: continuous-on A (arsinh :: real \Rightarrow real)
  by (rule DERIV-continuous-on derivative-intros)+

lemma continuous-on-arcosh [continuous-intros]:
  assumes A \subseteq \{1..\}
  shows continuous-on A (arcosh :: real \Rightarrow real)
proof –
  have pos: \(x + \sqrt{x^2 - 1} > 0\) if \(x \geq 1\) for x
    using that by (intro add-pos-nonneg) auto
  show ?thesis unfolding arcosh-def [abs-def]
  by (intro continuous-on-subset [OF - assms] continuous-on-ln continuous-on-add_continuous-on-id continuous-on-powr')
    (auto dest: pos simp: powr-half-sqrt intro!: continuous-intros)
qed
lemma continuous-on-artanh [continuous-intros]:
assumes $A \subseteq \{-1 <..< 1\}$
shows $\text{continuous-on } A \ (\text{artanh} :: \ 	ext{real} \Rightarrow \text{real})$

unfolding artanh-def [abs-def]
by (intro continuous-on-subset [OF - assms]) (auto intro!: continuous-intros)

lemma continuous-on-arsinh' [continuous-intros]:
fixes $f :: \text{real} \Rightarrow \text{real}$
assumes $\text{continuous-on } A \ f$
shows $\text{continuous-on } A \ (\lambda x. \text{arsinh} \ (f \ x))$
by (rule continuous-on-compose2 [OF continuous-on-arsinh assms])
(auto)

lemma continuous-on-arcosh' [continuous-intros]:
fixes $f :: \text{real} \Rightarrow \text{real}$
assumes $x > 1$
shows $\text{continuous-on } A \ (\lambda x. \text{arcosh} \ (f \ x))$
by (rule continuous-on-compose2 [OF continuous-on-arcosh assms(1) order.refl])
(auto)

lemma isCont-arsinh [continuous-intros]: isCont $\text{arsinh} \ (x :: \text{real})$
using continuous-on-arsinh[of UNIV] by (auto simp: continuous-on-eq-continuous-at)

lemma isCont-arcosh [continuous-intros]:
assumes $x > 1$
shows $\text{isCont } \text{arcosh} \ (x :: \text{real})$
proof –
have $\text{continuous-on } \{1::\text{real}<..\} \ \text{arcosh}$
by (rule continuous-on-arcosh)
(auto)
with assms show ?thesis by (auto simp: continuous-on-eq-continuous-at)
qed

lemma isCont-artanh [continuous-intros]:
assumes $x > -1 \ x < 1$
shows $\text{isCont } \text{artanh} \ (x :: \text{real})$
proof –
have $\text{continuous-on } \{-1<..<(1::\text{real})\} \ \text{artanh}$
by (rule continuous-on-artanh)
(auto)
with assms show ?thesis by (auto simp: continuous-on-eq-continuous-at)
qed

lemma tendsto-arsinh [tendsto-intros]: $(f \longrightarrow a) \ F \Rightarrow ((\lambda x. \text{arsinh} \ (f \ x)) \ \longrightarrow$
arsinh a) F
for f :: - ⇒ real
by (rule isCont-tendsto-compose [OF isCont-arsinh])

lemma tendsto-arcosh-strong [tendsto-intros]:
  fixes f :: - ⇒ real
  assumes (f −−−→ a) F a ≥ 1 eventually (λx. f x ≥ 1) F
  shows ((λx. arccosh (f x)) −−−→ arccosh a) F
  by (rule continuous-on-tendsto-compose[OF continuous-on-arcosh[OF order.refl]])
    (use assms in auto)

lemma tendsto-arcosh:
  fixes f :: - ⇒ real
  assumes (f −−−→ a) F a > -1 a < 1
  shows ((λx. arccosh (f x)) −−−→ arccosh a) F
  by (rule isCont-tendsto-compose [OF isCont-arcosh]) (use assms in auto)

lemma tendsto-arcosh-at-left-1: (arcosh −−−→ 0) (at-right (1::real))
  proof –
    have (arcosh −−−→ arcosh 1) (at-right (1::real))
      by (rule tendsto-arcosh-strong) (auto simp: eventually-at intro: exI[of - 1])
    thus ?thesis by simp
  qed

lemma tendsto-artanh [tendsto-intros]:
  fixes f :: 'a ⇒ real
  assumes (f −−−→ a) F a > -1 a < 1
  shows ((λx. artanh (f x)) −−−→ artanh a) F
  by (rule isCont-tendsto-compose [OF isCont-artanh]) (use assms in auto)

lemma continuous-arsinh [continuous-intros]:
  continuous F f ⇒ continuous F (λx. arsinh (f x :: real))
  unfolding continuous_def by (rule tendsto-arsinh)

lemma continuous-arcosh-strong [continuous-intros]:
  assumes continuous F f eventually (λx. f x ≥ 1) F
  shows continuous F (λx. arccosh (f x :: real))
  proof (cases F = bot)
    case False
    show ?thesis
      unfolding continuous_def
      proof (intro tendsto-arcosh-strong)
        show 1 ≤ f (Lim F (λx. x))
          using assms False unfolding continuous_def by (rule tendsto-lowerbound)
      qed
    qed
  qed auto

lemma continuous-arcosh:
continuous $F \ f = \Lim F (\lambda x. x) > 1 \Rightarrow$ continuous $F \ (\lambda x. \ arccosh (f \ x :: \ real))$

unfolding continuous-def by (rule tendsto-arcosh) auto

lemma continuous-artanh [continuous-intros]:
continuous $F \ f = \Lim F (\lambda x. x) \in \{-1 <..< 1\} \Rightarrow$ continuous $F \ (\lambda x. \ artanh (f \ x :: \ real))$

unfolding continuous-def by (rule tendsto-artanh) auto

lemma arsinh-real-at-top:
filterlim (arsinh :: real ⇒ real) at-top at-top
proof (subst filterlim-cong [OF refl refl])
  show filterlim (λx. ln (x + sqrt (1 + x^2))) at-top at-top
  by (intro filterlim-compose [OF ln-at-top filterlim-at-top-add-at-top] filterlim-ident)
      filterlim-compose [OF sqrt-at-top] filterlim-tendsto-add-at-top [OF tendsto-const]
    filterlim-pow-at-top auto
qed (auto intro!: eventually-mono [OF eventually-ge-at-top [of 1]] simp: arsinh-real-def add-ac)

lemma arsinh-real-at-bot:
filterlim (arsinh :: real ⇒ real) at-bot at-bot
proof –
  have filterlim (λx::real. -arsinh x) at-bot at-top
    by (subst filterlim-uminus-at-top [symmetric]) (rule arsinh-real-at-top)
  also have (λx::real. -arsinh x) = (λx. arsinh (-x)) by simp
  finally show ?thesis
    by (subst filterlim-at-bot-mirror)
qed

lemma arccosh-real-at-top:
filterlim (arccosh :: real ⇒ real) at-top at-top
proof (subst filterlim-cong [OF refl refl])
  show filterlim (λx. ln (x + sqrt (-1 + x^2))) at-top at-top
  by (intro filterlim-compose [OF ln-at-top filterlim-at-top-add-at-top] filterlim-ident)
      filterlim-compose [OF sqrt-at-top] filterlim-tendsto-add-at-top [OF tendsto-const]
    filterlim-pow-at-top auto
qed (auto intro!: eventually-mono [OF eventually-ge-at-top [of 1]] simp: arccosh-real-def)

lemma artanh-real-at-left-1:
filterlim (artanh :: real ⇒ real) at-top (at-left 1)
proof –
  have *: filterlim (λx::real. (1 + x) / (1 - x)) at-top (at-left 1)
    by (rule LIM-at-top-divide)
      (auto intro!: tendsto-eq-intros eventually-mono [OF eventually-at-left-real [of 0]])
  have filterlim (λx::real. (1/2) * ln ((1 + x) / (1 - x))) at-top (at-left 1)
    by (intro filterlim-tendsto-pos-mult-at-top [OF tendsto-const] *)
filterlim-compose \text{OF \ ln-at-top}) \ auto
also have $(\lambda x :: \text{real}. (1/2) \ast \ln((1 + x) / (1 - x))) = \text{arctanh}$
by \text{(simp add: \text{arctanh-def \ [abs-def]})}
finally show \text{?thesis}.
qed

lemma \text{arctanh-real-at-right-1}:
filterlim (\text{arctanh} :: \text{real} \Rightarrow \text{real}) \text{at-bot} \text{(at-right \ (-1))}
proof –
have \text{?thesis} \longleftrightarrow \text{filterlim} (\lambda x :: \text{real}. -\text{arctanh} \ x) \text{at-top} \text{(at-right \ (-1))}
by \text{(simp add: \text{filterlim-uminus-at-bot})}
also have \ldots \longleftrightarrow \text{filterlim} (\lambda x :: \text{real}. \text{arctanh} \ (-x)) \text{at-top} \text{(at-right \ (-1))}
by \text{(intro \text{filterlim-cong refl \ eventually-mono[OF \ eventually-at-right-real[of \ (-1) \ I]]) \ auto})
also have \ldots \ by \text{(rule \text{arctanh-real-at-left-1})}
finally show \text{?thesis}.
qed

110.24 Simprocs for root and power literals

lemma \text{numeral-powr-numeral-real [simp]}:
\text{numeral} m \ powr \text{numeral} n = (\text{numeral} m \ ˆ \text{numeral} n :: \text{real})
by \text{(simp add: \text{powr-numeral})}
context
begin

private lemma \text{sqrt-numeral-simproc-aux}:
assumes \text{m \ast \text{m} \equiv \text{n}}
shows \text{sqrt} (\text{numeral} \ n :: \text{real}) \equiv \text{numeral} m
proof –
have \text{numeral} n \equiv \text{numeral} m \ast (\text{numeral} m :: \text{real}) \ by \text{(simp add: \text{assms \ symmetric})}
moreover have \text{sqrt} \ldots \equiv \text{numeral} m \ by \text{(subst \text{real-sqrt-abs2) simp}}
ultimately show \text{sqrt} (\text{numeral} \ n :: \text{real}) \equiv \text{numeral} m \ by \text{simp}
qed

private lemma \text{root-numeral-simproc-aux}:
assumes \text{Num.\pow m n \equiv x}
shows \text{root} (\text{numeral} \ n :: \text{real}) \equiv \text{numeral} m
by \text{(subst \text{assms \ symmetric}, subst 	ext{numeral-pow}, subst \text{real-root-pos2 \ simp-all})}

private lemma \text{powr-numeral-simproc-aux}:
assumes \text{Num.\pow y n \equiv x}
shows \text{numeral} x \ powr (m / \text{numeral} n :: \text{real}) \equiv \text{numeral y \ powr} m
by \text{(subst \text{assms \ symmetric}, subst 	ext{numeral-pow}, subst \text{powr-numeral \ symmetric})}
\text{(simp, subst \text{powr-pow, simp-all})}
private lemma numeral-powr-inverse-eq:
  numeral x powr (inverse (numeral n)) = numeral x powr (1 / numeral n :: real)
  by simp

ML :

signature ROOT-NUMERAL-SIMPROC = sig

  val sqrt : int option => int => int option
  val sqrt' : int option => int => int option
  val nth-root : int option => int => int => int option
  val nth-root' : int option => int => int => int option
  val sqrt-simproc : Proof.context => cterm => thm option
  val root-simproc : int * int => Proof.context => cterm => thm option
  val powr-simproc : int * int => Proof.context => cterm => thm option

end

structure Root-Numeral-Simproc : ROOT-NUMERAL-SIMPROC = struct

  fun iterate NONE p f x = let
    fun go x = if p x then x else go (f x)
    in
      SOME (go x)
    end |
  iterate (SOME threshold) p f x = let
    fun go (threshold, x) = if p x then SOME x else if threshold = 0 then NONE else go (threshold - 1, f x)
    in
      go (threshold, x)
    end

fun nth-root - 1 x = SOME x |
  nth-root - - 0 = SOME 0 |
  nth-root - - 1 = SOME 1 |
  nth-root threshold n x = let
    fun newton-step y = ((n - 1) * y + x div Integer.pow (n - 1) y) div n
    fun is-root y = Integer.pow n y <= x andalso x < Integer.pow n (y + 1)
    in
      if x < n then
        SOME 1
      else if x < Integer.pow n 2 then
SOME 1
else
let
  val y = Real.floor (Math.pow (Real.fromInt x, Real.fromInt 1 / Real.fromInt n))
in
  if is-root y then
    SOME y
  else
    iterate threshold is-root newton-step ((x + n - 1) div n)
end
end

fun nth-root' 1 x = SOME x
| nth-root' - 0 = SOME 0
| nth-root' - 1 = SOME 1
| nth-root' threshold n x = if x < n then NONE else if x < Integer.pow n 2 then
  NONE else
    case nth-root threshold n x of
      NONE => NONE
    | SOME y => if Integer.pow n y = x then SOME y else NONE
end

fun sqrt - 0 = SOME 0
| sqrt - 1 = SOME 1
| sqrt threshold n =
  let
    fun aux (a, b) = if n >= b * b then aux (b, b * b) else (a, b)
    val (lower-root, lower-n) = aux (1, 2)
    fun newton-step x = (x + n div x) div 2
    fun is-sqrt r = r * r <= n andalso n < (r+1)*(r+1)
    val y = Real.floor (Math.sqrt (Real.fromInt n))
in
    if is-sqrt y then
      SOME y
    else
      Option.mapPartial (iterate threshold is-root newton-step o fn x => x * lower-root)
    (sqrt threshold (n div lower-n))
  end

fun sqrt' threshold x =
  case sqrt threshold x of
    NONE => NONE
  | SOME y => if y * y = x then SOME y else NONE

fun sqrt-simproc ctxt ct =
  let
    val n = ct |> Thm.term-of |> dest-comb |> snd |> dest-comb |> snd |> HOLogic.dest-numeral
fun root-simproc (threshold1, threshold2) ctxt ct =
  let
    val [n, x] =
      snd (Thm.term-of (Thm.dest-equals-rhs (Thm.cprop-of ct)))
    val ct = Thm.dest-comb (Thm.term-of ct)
  in
    if n > threshold1 orelse x > threshold2 then NONE else
      case nth-root' (SOME 100) n x of
        NONE => NONE
      | SOME m =>
        SOME (Thm.instantiate' [] (map (SOME o Thm.cterm-of ctxt o HO-Logic.mk-numeral) [m, n]))
          @{thm root-numeral-simproc-aux}
    end
  handle TERM - => NONE
end

fun powr-simproc (threshold1, threshold2) ctxt ct =
  let
    val eq-thm = Conv.try_conv (Conv.rewr Conv.rewr @{thm numeral-powr-inverse-eq})
    val ct = Thm.dest-equals-rhs (Thm.cprop-of eq-thm)
    val (-, [x, t]) = strip-comb (Thm.term-of ct)
    val (-, [m, n]) = strip-comb t
    val [x, n] = map (dest-comb #> snd #> HO-Logic.dest-numeral) [x, n]
    val thm = Thm.instantiate' [] (map (SOME o Thm.cterm-of ctxt) [y, n, x])
    val [y, n, x] = map HO-Logic.mk-numeral [y, n, x]
  in
    if n > threshold1 orelse x > threshold2 then NONE else
      case nth-root' (SOME 100) n x of
        NONE => NONE
      | SOME y =>
        let
          val [y, n, x] = map HO-Logic.mk-numeral [y, n, x]
          val thm = Thm.instantiate' [] (map (SOME o Thm.cterm-of ctxt) [y, n, x])
        in
          SOME (@{thm transitive} OF [eq-thm, thm])
        end
end
THEORY “Complex”

handle TERM - => NONE 
| Match => NONE

end

end

simproc-setup sqrt-numeral (sqrt (numeral n)) =
  ⟨K Root-Numeral-Simproc.sqrt-simproc⟩

simproc-setup root-numeral (root (numeral n) (numeral x)) =
  ⟨K (Root-Numeral-Simproc.root-simproc (200, Integer.pow 200 2))⟩

simproc-setup powr-divide-numeral
  (numeral x powr (m / numeral n :: real) | numeral x powr (inverse (numeral n) :: real)) =
  ⟨K (Root-Numeral-Simproc.powr-simproc (200, Integer.pow 200 2))⟩

lemma root 100 1267650600228229401496703205376 = 2
  by simp

lemma sqrt 196 = 14
  by simp

lemma 256 powr (7 / 4 :: real) = 16384
  by simp

lemma 27 powr (inverse 3) = (3::real)
  by simp

end

111 Complex Numbers: Rectangular and Polar Representations

theory Complex
imports Transcendental
begin

We use the codatatype command to define the type of complex numbers. This allows us to use primcorec to define complex functions by defining their real and imaginary result separately.

codatatype complex = Complex (Re: real) (Im: real)

lemma complex-surj: Complex (Re z) (Im z) = z
  by (rule complex.collapse)
lemma complex-eqI [intro?]: \( \text{Re } x = \text{Re } y \rightarrow \text{Im } x = \text{Im } y \rightarrow x = y \)
by (rule complex.expand) simp

lemma complex-eq-iff: \( x = y \iff \text{Re } x = \text{Re } y \land \text{Im } x = \text{Im } y \)
by (auto intro: complex.expand)

111.1 Addition and Subtraction

instantiation complex :: ab-group-add
begin

primcorec zero-complex
where
\( \text{Re } 0 = 0 \)
| \( \text{Im } 0 = 0 \)

primcorec plus-complex
where
\( \text{Re } (x + y) = \text{Re } x + \text{Re } y \)
| \( \text{Im } (x + y) = \text{Im } x + \text{Im } y \)

primcorec uminus-complex
where
\( \text{Re } (-x) = -\text{Re } x \)
| \( \text{Im } (-x) = -\text{Im } x \)

primcorec minus-complex
where
\( \text{Re } (x - y) = \text{Re } x - \text{Re } y \)
| \( \text{Im } (x - y) = \text{Im } x - \text{Im } y \)

instance
by standard (simp-all add: complex-eq-iff)
end

111.2 Multiplication and Division

instantiation complex :: field
begin

primcorec one-complex
where
\( \text{Re } 1 = 1 \)
| \( \text{Im } 1 = 0 \)

primcorec times-complex
where
\( \text{Re } (x \ast y) = \text{Re } x \ast \text{Re } y - \text{Im } x \ast \text{Im } y \)
\[ | \text{Im} (x \times y) = \text{Re} x \times \text{Im} y + \text{Im} x \times \text{Re} y \]

**primcorec** inverse-complex

**where**

\[ \text{Re} \ (\text{inverse} \ x) = \frac{\text{Re} x}{((\text{Re} x)^2 + (\text{Im} x)^2)} \]

\[ | \text{Im} \ (\text{inverse} \ x) = -\frac{\text{Im} x}{((\text{Re} x)^2 + (\text{Im} x)^2)} \]

**definition** \( x \div y = x \times \text{inverse} y \) for \( x \ y :: \text{complex} \)

**instance**

by standard

(simp all add: complex-eq-iff divide-complex-def distrib-left distrib-right right-diff-distrib left-diff-distrib power2-eq-square add-divide-distrib [symmetric])

end

**lemma** Re-divide: \( \text{Re} \ (x \div y) = \frac{(\text{Re} x \times \text{Re} y + \text{Im} x \times \text{Im} y)}{((\text{Re} y)^2 + (\text{Im} y)^2)} \)

by (simp add: divide-complex-def add-divide-distrib)

**lemma** Im-divide: \( \text{Im} \ (x \div y) = \frac{(\text{Im} x \times \text{Re} y - \text{Re} x \times \text{Im} y)}{((\text{Re} y)^2 + (\text{Im} y)^2)} \)

by (simp add: divide-complex-def diff-divide-distrib)

**lemma** Complex-divide: \( (x \div y) = \text{Complex} \left( \frac{(\text{Re} x \times \text{Re} y + \text{Im} x \times \text{Im} y)}{((\text{Re} y)^2 + (\text{Im} y)^2)} \right) \)

by (metis Im-divide Re-divide complex-surj)

**lemma** Re-power2: \( \text{Re} \ (x \cdot 2) = (\text{Re} x) \cdot 2 - (\text{Im} x) \cdot 2 \)

by (simp add: power2-eq-square)

**lemma** Im-power2: \( \text{Im} \ (x \cdot 2) = 2 \times \text{Re} x \times \text{Im} x \)

by (simp add: power2-eq-square)

**lemma** Re-power-real [simp]: \( \text{Im} x = 0 \implies \text{Re} (x \cdot n) = \text{Re} x \cdot n \)

by (induct n) simp-all

**lemma** Im-power-real [simp]: \( \text{Im} x = 0 \implies \text{Im} (x \cdot n) = 0 \)

by (induct n) simp-all

**111.3 Scalar Multiplication**

**instantiation** complex :: real-field

begin

**primcorec** scaleR-complex

where
instance
proof
  fix a b :: real and x y :: complex
  show scaleR a (x + y) = scaleR a x + scaleR a y
    by (simp add: complex-eq-iff distrib-left)
  show scaleR (a + b) x = scaleR a x + scaleR b x
    by (simp add: complex-eq-iff distrib-right)
  show scaleR a (scaleR b x) = scaleR (a * b) x
    by (simp add: complex-eq-iff mult_assoc)
  show scaleR 1 x = x
    by (simp add: complex-eq-iff)
  show scaleR a x * y = scaleR a (x * y)
    by (simp add: complex-eq-iff algebra-simps)
  show x * scaleR a y = scaleR a (x * y)
    by (simp add: complex-eq-iff algebra-simps)
qed
end

111.4 Numerals, Arithmetic, and Embedding from R

abbreviation complex-of-real :: real ⇒ complex
  where complex-of-real ≡ of-real

declare [[coercion of-real :: real ⇒ complex]]
declare [[coercion of-rat :: rat ⇒ complex]]
declare [[coercion of-int :: int ⇒ complex]]
declare [[coercion of-nat :: nat ⇒ complex]]

lemma complex-Re-of-nat [simp]: Re (of-nat n) = of-nat n
  by (induct n) simp-all

lemma complex-Im-of-nat [simp]: Im (of-nat n) = 0
  by (induct n) simp-all

lemma complex-Re-of-int [simp]: Re (of-int z) = of-int z
  by (cases z rule: int-diff-cases) simp

lemma complex-Im-of-int [simp]: Im (of-int z) = 0
  by (cases z rule: int-diff-cases) simp

lemma complex-Re-numeral [simp]: Re (numeral v) = numeral v
  using complex-Re-of-int [of numeral v] by simp

lemma complex-Im-numeral [simp]: Im (numeral v) = 0
  using complex-Im-of-int [of numeral v] by simp
lemma Re-complex-of-real [simp]: \( \text{Re} (\text{complex-of-real} \ z) = z \)
by (simp add: of-real-def)

lemma Im-complex-of-real [simp]: \( \text{Im} (\text{complex-of-real} \ z) = 0 \)
by (simp add: of-real-def)

lemma Re-divide-numeral [simp]: \( \text{Re} (z / \text{numeral} \ w) = \text{Re} z / \text{numeral} \ w \)
by (simp add: Re-divide sqr-conv-mult)

lemma Im-divide-numeral [simp]: \( \text{Im} (z / \text{numeral} \ w) = \text{Im} z / \text{numeral} \ w \)
by (simp add: Im-divide sqr-conv-mult)

lemma Re-divide-of-nat [simp]: \( \text{Re} (z / \text{of-nat} \ n) = \text{Re} z / \text{of-nat} \ n \)
by (cases n) (simp-all add: Re-divide field-split-simps power2-eq-square del: of-nat-Suc)

lemma Im-divide-of-nat [simp]: \( \text{Im} (z / \text{of-nat} \ n) = \text{Im} z / \text{of-nat} \ n \)
by (cases n) (simp-all add: Im-divide field-split-simps power2-eq-square del: of-nat-Suc)

lemma of-real-Re [simp]: \( z \in \mathbb{R} \implies \text{of-real} (\text{Re} z) = z \)
by (auto simp: Reals-def)

lemma complex-Re-fact [simp]: \( \text{Re} (\text{fact} \ n) = \text{fact} \ n \)
proof
  have (\text{fact} \ n :: \text{complex}) = \text{of-real} (\text{fact} \ n)
    by simp
  also have \text{Re} \ldots = \text{fact} \ n
    by (subst Re-complex-of-real) simp-all
  finally show ?thesis.
qed

lemma complex-Im-fact [simp]: \( \text{Im} (\text{fact} \ n) = 0 \)
by (subst of-nat-fact [symmetric]) (simp only: complex-Im-of-nat)

lemma Re-prod-Reals: \( \bigwedge x. x \in A \implies f x \in \mathbb{R} \implies \text{Re} (\text{prod} \ f \ A) = \text{prod} (\lambda x. \text{Re} (f x)) \ A \)
proof (induction A rule: infinite-finite-induct)
case (insert x A)
  hence \( \text{Re} (\text{prod} f (\text{insert} x A)) = \text{Re} (f x) * \text{Re} (\text{prod} f A) - \text{Im} (f x) * \text{Im} (\text{prod} f A) \)
    by simp
  also from insert.prems have \( f x \in \mathbb{R} \) by simp
  hence \( \text{Im} (f x) = 0 \) by (auto elim!: Reals-cases)
  also have \( \text{Re} (\text{prod} f A) = (\prod x \in A. \text{Re} (f x)) \)
    by (intro insert.IH insert.prems) auto
  finally show ?case using insert.hyps by simp
qed auto
111.5 The Complex Number $i$

primcorec imaginary-unit :: complex (i)
where
  $\text{Re } i = 0$
  $\text{Im } i = 1$

lemma Complex-eq: $\text{Complex } a \ b = a + i \ b$
  by (simp add: complex-eq-iff)

lemma complex-eq: $a = \text{Re } a + i \ \text{Im } a$
  by (simp add: complex-eq-iff)

lemma fun-complex-eq: $f = (\lambda x. \text{Re } (f x) + i \ \text{Im } (f x))$
  by (simp add: fun-eq-iff complex-eq)

lemma i-squared [simp]: $i \ast i = -1$
  by (simp add: complex-eq-iff)

lemma power2-i [simp]: $i^2 = -1$
  by (simp add: power2-eq-square)

lemma inverse-i [simp]: $\text{inverse } i = -i$
  by (rule inverse-unique simp)

lemma divide-i [simp]: $x / i = -i * x$
  by (simp add: divide-complex-def)

lemma complex-i-mult-minus [simp]: $i * (i * x) = -x$
  by (simp add: mult.assoc [symmetric])

lemma complex-i-not-zero [simp]: $i \neq 0$
  by (simp add: complex-eq-iff)

lemma complex-i-not-one [simp]: $i \neq 1$
  by (simp add: complex-eq-iff)

lemma complex-i-not-numeral [simp]: $i \neq \text{numeral } w$
  by (simp add: complex-eq-iff)

lemma complex-i-not-neg-numeral [simp]: $i \neq -\text{numeral } w$
  by (simp add: complex-eq-iff)

lemma complex-split-polar: $\exists r a. \ z = \text{complex-of-real } r * (\cos a + i * \sin a)$
  by (simp add: complex-eq-iff polar-Ex)

lemma i-even-power [simp]: $i ^{n * 2} = (-1) ^{n}$
  by (metis mult.commute power2-i power-mult)

lemma Re-i-times [simp]: $\text{Re } (i * z) = -\text{Im } z$
by simp

lemma Im-i-times [simp]: Im (i * z) = Re z
  by simp

lemma i-times-eq-iff: i * w = z ←→ w = - (i * z)
  by auto

lemma divide-numeral-i [simp]: z / (numeral n * i) = -(i * z) / numeral n
  by (metis divide-divide-eq-left divide-i mult.commute mult-minus-right)

lemma imaginary-eq-real-iff [simp]:
  assumes y ∈ Reals x ∈ Reals
  shows i * y = x ←→ x=0 ∧ y=0
    using assms
    unfolding Reals-def
    apply clarify
    apply (rule iffI)
    apply (metis Im-complex-of-real Im-i-times Re-complex-of-real mult-eq-0-iff of-real-0)
  by simp

lemma real-eq-imaginary-iff [simp]:
  assumes y ∈ Reals x ∈ Reals
  shows x = i * y ←→ x=0 ∧ y=0
    using assms imaginary-eq-real-iff
    by fastforce

111.6 Vector Norm

instantiation complex :: real-normed-field
begin

definition norm z = sqrt ((Re z)^2 + (Im z)^2)

abbreviation cmod :: complex ⇒ real
  where cmod ≡ norm

definition complex-sgn-def: sgn x = x / _cmod x

definition dist-complex-def: dist x y = cmod (x - y)

definition uniformity-complex-def [code del]:
  (uniformity :: (complex × complex) filter) = (INF e∈{0 <..}. principal {(x, y). dist x y < e})

definition open-complex-def [code del]:
  open (U :: complex set) ←→ (∀x∈U. eventually (λ(x’, y). x’ = x → y ∈ U) uniformity)
instance
proof
  fix r :: real and x y :: complex and S :: complex set
  show (norm x = 0) = (x = 0)
    by (simp add: norm-complex-def complex-eq-iff)
  show norm (x + y) ≤ norm x + norm y
    by (simp add: norm-complex-def complex-eq-iff real-sqrt-sum-squares-triangle-ineq)
  show norm (scaleR r x) = |r| * norm x
    by (simp add: norm-complex-def complex-eq-iff power-mult-distrib distrib-left [symmetric] real-sqrt-mult)
  show norm (x * y) = norm x * norm y
    by (simp add: norm-complex-def complex-eq-iff real-sqrt-mult [symmetric] power2-eq-square algebra-simps)
qed (rule complex-sgn-def dist-complex-def open-complex-def uniformity-complex-def)

end

declare uniformity-Abort [where 'a = complex, code]

lemma norm-ii [simp]: norm i = 1
  by (simp add: norm-complex-def)

lemma cmod-unit-one: cmod (cos a + i * sin a) = 1
  by (simp add: norm-complex-def)

lemma cmod-complex-polar: cmod (r * (cos a + i * sin a)) = |r|
  by (simp add: norm-mult cmod-unit-one)

lemma complex-Re-le-cmod: Re x ≤ cmod x
  unfolding norm-complex-def by (rule real-sqrt-sum-squares-ge1)

lemma complex-mod-minus-le-complex-mod: - cmod x ≤ cmod x
  by (rule order-trans [OF - norm-ge-zero]) simp

lemma complex-mod-triangle-ineq2: cmod (b + a) - cmod b ≤ cmod a
  by (rule ord-le-eq-trans [OF norm-triangle-ineq2]) simp

lemma abs-Re-le-cmod: |Re x| ≤ cmod x
  by (simp add: norm-complex-def)

lemma abs-Im-le-cmod: |Im x| ≤ cmod x
  by (simp add: norm-complex-def)

lemma cmod-le: cmod z ≤ |Re z| + |Im z|
  apply (subst complex-eq)
  apply (rule order-trans)
  apply (rule norm-triangle-ineq)
  apply (simp add: norm-mult)
done

lemma cmod-eq-Re: \( Im \, z = 0 \implies cmod \, z = |Re \, z| \)
by (simp add: norm-complex-def)

lemma cmod-eq-Im: \( Re \, z = 0 \implies cmod \, z = |Im \, z| \)
by (simp add: norm-complex-def)

lemma cmod-power2: \((cmod \, z)^2 = (Re \, z)^2 + (Im \, z)^2\)
by (simp add: norm-complex-def)

lemma cmod-plus-Re-le-0-iff: \( cmod \, z + Re \, z \leq 0 \iff Re \, z = -cmod \, z \)
using abs-Re-le-cmod[of z] by auto

lemma cmod-Re-le-iff: \( Im \, x = Im \, y = \Rightarrow cmod \, x \leq cmod \, y \iff |Im \, x| \leq |Im \, y| \)
by (metis add-le-cancel-left norm-complex-def real-sqrt-abs real-sqrt-le-iff)

lemma cmod-Im-le-iff: \( Re \, x = Re \, y = \Rightarrow cmod \, x \leq cmod \, y \iff |Im \, x| \leq |Im \, y| \)
by (metis add-le-cancel-left norm-complex-def real-sqrt-abs real-sqrt-le-iff)

lemma Im-eq-0: \( |Re \, z| = cmod \, z = \Rightarrow Im \, z = 0 \)
by (subst (asm) power-eq-iff-eq-base[symmetric, where n=2]) (auto simp add: norm-complex-def)

lemma abs-sqrt-wlog: \((\forall x. x \geq 0 \implies P \, x \, (x^2)) \implies P \, |x| \, (x^2)\)
for x::linordered-idom
by (metis abs-ge-zero power2-abs)

lemma complex-abs-le-norm: \(|Re \, z| + |Im \, z| \leq sqrt \, 2 * norm \, z\)
unfolding norm-complex-def
apply (rule abs-sqrt-wlog [where x=Re z])
apply (rule abs-sqrt-wlog [where x=Im z])
apply (rule power2-le-imp-le)
apply (simp-all add: power2-sum add.commute sum-squares-bound real-sqrt-mult [symmetric])
done

lemma complex-unit-circle: \( z \neq 0 \implies (Re \, z / cmod \, z)^2 + (Im \, z / cmod \, z)^2 = 1 \)
by (simp add: norm-complex-def complex-eq-iff power2-eq-square add-divide-distrib [symmetric])

Properties of complex signum.

lemma sgn-eq: \( sgn \, z = z / complex-of-real \, (cmod \, z) \)
by (simp add: sgn-div-norm divide-inverse scaleR-conv-of-real mult.commute)

lemma Re-sgn [simp]: \( Re(sgn \, z) = Re(z)/cmod \, z \)
by (simp add: complex-sgn-def divide-inverse)

lemma Im-sgn [simp]: \( Im(sgn \, z) = Im(z)/cmod \, z \)
by (simp add: complex-sgn-def divide-inverse)

111.7 Absolute value

instantiation complex :: field-abs-sgn

begin

definition abs-complex :: complex ⇒ complex
  where abs-complex = of-real ◦ norm

instance
  apply standard
  apply (auto simp add: abs-complex-def complex-sgn-def norm-mult)
  apply (auto simp add: scaleR-conv-of-real field-simps)
  done

end

111.8 Completeness of the Complexes

lemma bounded-linear-Re: bounded-linear Re
  by (rule bounded-linear-intro [where K=1]) (simp-all add: norm-complex-def)

lemma bounded-linear-Im: bounded-linear Im
  by (rule bounded-linear-intro [where K=1]) (simp-all add: norm-complex-def)

lemmas Cauchy-Re = bounded-linear.Cauchy [OF bounded-linear-Re]
lemmas Cauchy-Im = bounded-linear.Cauchy [OF bounded-linear-Im]
lemmas tendsto-Re [tendsto-intros] = bounded-linear.tendsto [OF bounded-linear-Re]
lemmas tendsto-Im [tendsto-intros] = bounded-linear.tendsto [OF bounded-linear-Im]
lemmas isCont-Re [simp] = bounded-linear.isCont [OF bounded-linear-Re]
lemmas isCont-Im [simp] = bounded-linear.isCont [OF bounded-linear-Im]
lemmas continuous-Re [simp] = bounded-linear.continuous [OF bounded-linear-Re]
lemmas continuous-Im [simp] = bounded-linear.continuous [OF bounded-linear-Im]
lemmas continuous-on-Re [continuous-intros] = bounded-linear.continuous-on[OF bounded-linear-Re]
lemmas continuous-on-Im [continuous-intros] = bounded-linear.continuous-on[OF bounded-linear-Im]
lemmas has-derivative-Re [derivative-intros] = bounded-linear.has-derivative[OF bounded-linear-Re]
lemmas has-derivative-Im [derivative-intros] = bounded-linear.has-derivative[OF bounded-linear-Im]
lemmas sums-Re = bounded-linear.sums [OF bounded-linear-Re]
lemmas sums-Im = bounded-linear.sums [OF bounded-linear-Im]

lemma tendsto-Complex [tendsto-intros];
  (f −→ a) F ⇒ (g −→ b) F ⇒ ((λx. Complex (f x) (g x)) −→ Complex a b) F
  unfolding Complex-eq by (auto intro!: tendsto-intros)
lemma tendsto-complex-iff:
\[
(f \to x) F \iff ((\lambda x. \operatorname{Re} (f x)) \to \operatorname{Re} x) F \land ((\lambda x. \operatorname{Im} (f x)) \to \operatorname{Im} x) F
\]

proof safe
  assume \((\lambda x. \operatorname{Re} (f x)) \to \operatorname{Re} x) F \land ((\lambda x. \operatorname{Im} (f x)) \to \operatorname{Im} x) F

  from tendsto-Complex[OF this] show \((f \to x) F\)
    unfolding complex.collapse .
qed (auto intro: tendsto-intros)

lemma continuous-complex-iff:
\[
\text{continuous } F f \iff \text{continuous } F (\lambda x. \operatorname{Re} (f x)) \land \text{continuous } F (\lambda x. \operatorname{Im} (f x))
\]

by (simp only: continuous-def tendsto-complex-iff)

lemma continuous-on-of-real-o-iff [simp]:
\[
\text{continuous-on } S (\lambda x. \operatorname{complex-of-real} (g x)) = \text{continuous-on } S g
\]

using continuous-on-Re continuous-on-of-real by fastforce

lemma continuous-on-of-real-id [simp]:
\[
\text{continuous-on } S (\operatorname{of-real} :: \text{real-normed-algebra-1})
\]

by (rule continuous-on-of-real [OF continuous-on-id])

lemma has-vector-derivative-complex-iff: \((f \text{ has-vector-derivative } x) F \iff ((\lambda x. \operatorname{Re} (f x)) \text{ has-field-derivative } (\operatorname{Re} x)) F \land ((\lambda x. \operatorname{Im} (f x)) \text{ has-field-derivative } (\operatorname{Im} x)) F\)

by (simp add: has-vector-derivative-def has-field-derivative-def has-derivative-def tendsto-complex-iff algebra-simps bounded-linear-scaleR-left bounded-linear-mult-right)

lemma has-field-derivative-Re[derivative-intros]:
\[(f \text{ has-vector-derivative } D) F \implies ((\lambda x. \operatorname{Re} (f x)) \text{ has-field-derivative } (\operatorname{Re} D)) F\]

unfolding has-vector-derivative-complex-iff by safe

lemma has-field-derivative-Im[derivative-intros]:
\[(f \text{ has-vector-derivative } D) F \implies ((\lambda x. \operatorname{Im} (f x)) \text{ has-field-derivative } (\operatorname{Im} D)) F\]

unfolding has-vector-derivative-complex-iff by safe

instance complex :: banach

proof
  fix \(X :: \text{ nat} \Rightarrow \text{ complex}\)
  assume \(X: \text{ Cauchy } X\)
  then have \((\lambda n. \text{ Complex } (\operatorname{Re} (X n)) (\operatorname{Im} (X n))) \to \text{ Complex } (\lim (\lambda n. \operatorname{Re} (X n))) (\lim (\lambda n. \operatorname{Im} (X n)))\)
    by (intro tendsto-Complex convergent-LIMSEQ-iff[OF this, THEN iffD1]
      Cauchy-convergent-iff[THEN iffD1] Cauchy-Re Cauchy-Im)
  then show \(\text{ convergent } X\)
    unfolding complex.collapse by (rule convergentI)
qed

declare DERIV-power[where 'a=complex, unfolded of-nat-def[ symmetric], derivative-intros]
111.9 Complex Conjugation

\texttt{primcorec \texttt{cnj :: complex ⇒ complex}}

\texttt{where}

\begin{align*}
\text{Re} (\text{cnj} \: z) &= \text{Re} \: z \\
\text{Im} (\text{cnj} \: z) &= -\text{Im} \: z
\end{align*}

\texttt{lemma complex-cnj-cancel-iff \: [simp]: cnj \: x = cnj \: y ↔ x = y}

\texttt{by (simp\ add: complex-eq-iff)}

\texttt{lemma complex-cnj-cnj \: [simp]: cnj (cnj \: z) = z}

\texttt{by (simp\ add: complex-eq-iff)}

\texttt{lemma complex-cnj-zero \: [simp]: cnj \: 0 = 0}

\texttt{by (simp\ add: complex-eq-iff)}

\texttt{lemma complex-cnj-zero-iff \: [iff]: cnj \: z = 0 ↔ z = 0}

\texttt{by (simp\ add: complex-eq-iff)}

\texttt{lemma complex-cnj-one-iff \: [simp]: cnj \: z = 1 ↔ z = 1}

\texttt{by (simp\ add: complex-eq-iff)}

\texttt{lemma complex-cnj-add \: [simp]: cnj \: (x + y) = cnj \: x + cnj \: y}

\texttt{by (simp\ add: complex-eq-iff)}

\texttt{lemma cnj-sum \: [simp]: cnj \: (\sum f \: s) = (\sum x \in s. \: cnj \: (f \: x))}

\texttt{by (induct \: s \: rule: infinite-finite-induct)\ auto}

\texttt{lemma complex-cnj-diff \: [simp]: cnj \: (x - y) = cnj \: x - cnj \: y}

\texttt{by (simp\ add: complex-eq-iff)}

\texttt{lemma complex-cnj-minus \: [simp]: cnj \: (- \: x) = - \: cnj \: x}

\texttt{by (simp\ add: complex-eq-iff)}

\texttt{lemma complex-cnj-one \: [simp]: cnj \: 1 = 1}

\texttt{by (simp\ add: complex-eq-iff)}

\texttt{lemma complex-cnj-mult \: [simp]: cnj \: (x * y) = cnj \: x * cnj \: y}

\texttt{by (simp\ add: complex-eq-iff)}

\texttt{lemma cnj-prod \: [simp]: cnj \: (\prod f \: s) = (\prod x \in s. \: cnj \: (f \: x))}

\texttt{by (induct \: s \: rule: infinite-finite-induct)\ auto}

\texttt{lemma complex-cnj-inverse \: [simp]: cnj \: \text{inverse} \: x = \text{inverse} \: (cnj \: x)}

\texttt{by (simp\ add: complex-eq-iff)}

\texttt{lemma complex-cnj-divide \: [simp]: cnj \: (x / y) = cnj \: x / cnj \: y}

\texttt{by (simp\ add: divide-complex-def)}

\texttt{lemma complex-cnj-power \: [simp]: cnj \: (x ^ n) = cnj \: x ^ n}
by (induct n) simp-all

lemma complex-cnj-of-nat [simp]: cnj (of-nat n) = of-nat n
  by (simp add: complex-eq-iff)

lemma complex-cnj-of-int [simp]: cnj (of-int z) = of-int z
  by (simp add: complex-eq-iff)

lemma complex-cnj-numeral [simp]: cnj (numeral w) = numeral w
  by (simp add: complex-eq-iff)

lemma complex-cnj-neg-numeral [simp]: cnj (−numeral w) = −numeral w
  by (simp add: complex-eq-iff)

lemma complex-cnj-scaleR [simp]: cnj (scaleR r x) = scaleR r (cnj x)
  by (simp add: complex-eq-iff)

lemma complex-mod-cnj [simp]: cmod (cnj z) = cmod z
  by (simp add: norm-complex-def)

lemma complex-cnj-complex-of-real [simp]: cnj (of-real x) = of-real x
  by (simp add: complex-eq-iff)

lemma complex-cnj-i [simp]: cnj i = −i
  by (simp add: complex-eq-iff)

lemma complex-add-cnj: z + cnj z = complex-of-real (2 * Re z)
  by (simp add: complex-eq-iff)

lemma complex-diff-cnj: z − cnj z = complex-of-real (2 * Im z) * i
  by (simp add: complex-eq-iff)

lemma complex-mult-cnj: z * cnj z = complex-of-real ((Re z)^2 + (Im z)^2)
  by (simp add: complex-eq-iff power2-eq-square)

lemma cnj-add-mult-cnj-Real: z * cnj w + cnj z * w = 2 * Re (z * cnj w)
  by (rule complex-eql) auto

lemma complex-mod-mult-cnj: cmod (z * cnj z) = (cmod z)^2
  by (simp add: norm-mult power2-eq-square)

lemma complex-mod-sqrt-Re-mult-cnj: cmod z = sqrt (Re (z * cnj z))
  by (simp add: norm-complex-def power2-eq-square)

lemma complex-In-mult-cnj-zero [simp]: Im (z * cnj z) = 0
  by simp

lemma complex-cnj-fact [simp]: cnj (fact n) = fact n
  by (subst of-nat-fact [symmetric], subst complex-cnj-of-nat) simp
lemma complex-cnj-pochhammer [simp]: \( \text{cnj} (\text{pochhammer} \ z \ n) = \text{pochhammer} (\text{cnj} \ z) \ n \)
  by (induct n arbitrary: z) (simp-all add: pochhammer-rec)

lemma bounded-linear-cnj: bounded-linear cnj
  using complex-cnj-add complex-cnj-scaleR by (rule bounded-linear-intro [where K=1]) simp

lemma linear-cnj: linear cnj
  using bounded-linear.
  linear [OF bounded-linear-cnj].

lemmas tendsto-cnj [tendsto-intros] = bounded-linear.tendsto [OF bounded-linear-cnj]
  and isCont-cnj [simp] = bounded-linear.isCont [OF bounded-linear-cnj]
  and continuous-cnj [simp, continuous-intros] = bounded-linear.continuous [OF bounded-linear-cnj]
  and continuous-on-cnj [simp, continuous-intros] = bounded-linear.continuous-on [OF bounded-linear-cnj]
  and has-derivative-cnj [simp, derivative-intros] = bounded-linear.has-derivative [OF bounded-linear-cnj]

lemma lim-cnj: \( (\lambda x. \text{cnj}(f \ x)) \to \text{cnj} \ l \) \( F \leftrightarrow (f \to l) \)

lemma sums-cnj: \( (\lambda x. \text{cnj}(f \ x)) \to \text{cnj} \ l \) \( \leftrightarrow (f \to l) \)
  by (simp add: sums-def lim-cnj cnj-sum [symmetric] del: cnj-sum)

lemma differentiable-cnj-iff:
  \( (\lambda x. \text{cnj} (f \ x)) \) differentiable at x within \( A \) \( \leftrightarrow f \) differentiable at x within \( A \)
proof
  assume \( (\lambda x. \text{cnj} (f \ x)) \) differentiable at x within \( A \)
  then obtain D where \( ((\lambda x. \text{cnj} (f \ x)) \) has-derivative \( D \) \( \) at \( x \) within \( A \)
    by (auto simp: differentiable-def)
  from has-derivative-cnj [OF this] show \( f \) differentiable at x within \( A \)
    by (auto simp: differentiable-def)
next
  assume \( f \) differentiable at x within \( A \)
  then obtain D where \( (f \) has-derivative \( D \) \( \) at \( x \) within \( A \)
    by (auto simp: differentiable-def)
  from has-derivative-cnj [OF this] show \( (\lambda x. \text{cnj} (f \ x)) \) differentiable at x within \( A \)
    by (auto simp: differentiable-def)
qed

lemma has-vector-derivative-cnj [derivative-intros]:
  assumes \( f \) has-vector-derivative \( f' \) \( \) at \( z \) within \( A \)
  shows \( ((\lambda x. \text{cnj} (f \ x)) \) has-vector-derivative \( \text{cnj} f' \) \( \) at \( z \) within \( A \)
  using assms by (auto simp: has-vector-derivative-complex-iff intro: derivative-intros)
111.10 Basic Lemmas

lemma complex-eq-0: \( z = 0 \) \(\iff\) \((Re z)^2 + (Im z)^2 = 0\)
    by (metis zero-complex sel complex-eqI sum-power2-eq-zero-iff)

lemma complex-neq-0: \( z \neq 0 \) \(\iff\) \((Re z)^2 + (Im z)^2 > 0\)
    by (metis complex-eq-0 less-numeral-extra(3) sum-power2-gt-zero-iff)

lemma complex-norm-square: of-real \(((\text{norm } z)^2) = z \ast \text{cnj } z\)
    by (cases z)
    (auto simp complex-eq-iff norm-complex-def power2-eq-square
        [symmetric] of-real-power [symmetric] simp del: of-real-power)

lemma complex-div-cnj: \( a / b = (a \ast \text{cnj } b) / (\text{norm } b)^2\)
    using complex-norm-square by auto

lemma Re-complex-div-eq-0: \( Re (a / b) = 0 \) \(\iff\) \( Re (a \ast \text{cnj } b) = 0\)
    by (auto simp add: Re-divide)

lemma Im-complex-div-eq-0: \( Im (a / b) = 0 \) \(\iff\) \( Im (a \ast \text{cnj } b) = 0\)
    by (auto simp add: Im-divide)

lemma complex-div-gt-0: \( Re (a / b) > 0 \) \(\iff\) \( Re (a \ast \text{cnj } b) > 0\) \(\wedge\) \( Im (a / b) > 0 \) \(\iff\) \( Im (a \ast \text{cnj } b) > 0\)
    proof (cases b = 0)
        case True
        then show \{thesis\} by auto
    next
        case False
        then have \( 0 < (Re b)^2 + (Im b)^2\)
            by (simp add: complex-eq-iff sum-power2-gt-zero-iff)
        then show \{thesis\}
            by (simp add: Re-divide Im-divide zero-less-divide-iff)
    qed

lemma Re-complex-div-gt-0: \( Re (a / b) > 0 \) \(\iff\) \( Re (a \ast \text{cnj } b) > 0\)
    and \( Im-complex-div-gt-0: Im (a / b) > 0 \) \(\iff\) \( Im (a \ast \text{cnj } b) > 0\)
    using complex-div-gt-0 by auto

lemma Re-complex-div-le-0: \( Re (a / b) \geq 0 \) \(\iff\) \( Re (a \ast \text{cnj } b) \geq 0\)
    by (metis le-less Re-complex-div-eq-0 Re-complex-div-gt-0)

lemma Im-complex-div-le-0: \( Im (a / b) \geq 0 \) \(\iff\) \( Im (a \ast \text{cnj } b) \geq 0\)
    by (metis Im-complex-div-eq-0 Im-complex-div-gt-0 le-less)

lemma Re-complex-div-le-0: \( Re (a / b) < 0 \) \(\iff\) \( Re (a \ast \text{cnj } b) < 0\)
    by (metis less-asym neq-iff Re-complex-div-eq-0 Re-complex-div-gt-0)

lemma Im-complex-div-le-0: \( Im (a / b) < 0 \) \(\iff\) \( Im (a \ast \text{cnj } b) < 0\)
    by (metis Im-complex-div-eq-0 Im-complex-div-gt-0 less-asym neq-iff)
lemma \( \text{Re-complex-div-le-0} \): \( \text{Re} \left( \frac{a}{b} \right) \leq 0 \iff \text{Re} \left( \frac{a \ast \text{cnj } b}{\text{cnj } b} \right) \leq 0 \)
by (metis not-le Re-complex-div-gt-0)

lemma \( \text{Im-complex-div-le-0} \): \( \text{Im} \left( \frac{a}{b} \right) \leq 0 \iff \text{Im} \left( \frac{a \ast \text{cnj } b}{\text{cnj } b} \right) \leq 0 \)
by (metis Im-complex-div-gt-0 not-le)

lemma \( \text{Re-divide-of-real} \) [simp]: \( \text{Re} \left( \frac{z}{\text{of-real } r} \right) = \text{Re} z \ast \frac{1}{r} \)
by (simp add: Re-divide power2-eq-square)

lemma \( \text{Im-divide-of-real} \) [simp]: \( \text{Im} \left( \frac{z}{\text{of-real } r} \right) = \text{Im} z \ast \frac{1}{r} \)
by (simp add: Im-divide power2-eq-square)

lemma \( \text{Re-divide-Reals} \) [simp]: \( r \in \mathbb{R} \iff \text{Re} \left( \frac{z}{r} \right) = \text{Re} z \ast \frac{1}{\text{Re } r} \)
by (metis Re-divide-of-real of-real-Re)

lemma \( \text{Im-divide-Reals} \) [simp]: \( r \in \mathbb{R} \iff \text{Im} \left( \frac{z}{r} \right) = \text{Im} z \ast \frac{1}{\text{Re } r} \)
by (metis Im-divide-of-real of-real-Re)

lemma \( \text{Re-sum} \) [simp]: \( \text{Re} \left( \sum f \in s. f \right) = \sum_{x \in s.} \text{Re } f x \)
by (induct s rule: infinite-finite-induct) auto

lemma \( \text{Im-sum} \) [simp]: \( \text{Im} \left( \sum f \in s. f \right) = \sum_{x \in s.} \text{Im } f x \)
by (induct s rule: infinite-finite-induct) auto

lemma \( \text{sums-complex-iff} \): \( f \text{ sums } x \iff (\lambda x. \text{Re } f x) \text{ sums } \text{Re } x \land (\lambda x. \text{Im } f x) \text{ sums } \text{Im } x \)
unfolding sums-def tendsto-complex-iff Im-sum Re-sum ..

lemma \( \text{summable-complex-iff} \): \( \text{summable } f \iff \text{summable } (\lambda x. \text{Re } f x) \land \text{summable } (\lambda x. \text{Im } f x) \)
unfolding summable-def sums-complex-iff[abs-def] by (metis complex.sel)

lemma \( \text{summable-complex-of-real} \) [simp]: \( \text{summable } (\lambda n. \text{complex-of-real } f n) \iff \text{summable } f \)
unfolding summable-complex-iff by simp

lemma \( \text{summable-Re} \): \( \text{summable } f \implies \text{summable } (\lambda x. \text{Re } f x) \)
unfolding summable-complex-iff by blast

lemma \( \text{summable-Im} \): \( \text{summable } f \implies \text{summable } (\lambda x. \text{Im } f x) \)
unfolding summable-complex-iff by blast

lemma \( \text{complex-is-Nat-iff} \): \( z \in \mathbb{N} \iff \text{Im } z = 0 \land (\exists i. \text{Re } z = \text{of-nat } i) \)
by (auto simp: Nats-def complex-eq-iff)

lemma \( \text{complex-is-Int-iff} \): \( z \in \mathbb{Z} \iff \text{Im } z = 0 \land (\exists i. \text{Re } z = \text{of-int } i) \)
by (auto simp: Ints-def complex-eq-iff)
lemma complex-is-Real-iff: \( z \in \mathbb{R} \iff \text{Im} z = 0 \)

by (auto simp: complex-eq-iff)

lemma Reals-cnj-iff: \( z \in \mathbb{R} \iff \text{cnj} z = z \)

by (auto simp: complex-is-Real-iff complex-eq-iff)

lemma in-Real-norm: \( z \in \mathbb{R} = \rightarrow \|z\| = |\text{Re} z| \)

by (simp add: complex-is-Real-iff norm-complex-def)

lemma Re-Reals-divide: \( r \in \mathbb{R} = \rightarrow \text{Re} \left(\frac{r}{z}\right) = \frac{\text{Re} r \times \text{Re} z}{\|z\|^2} \)

by (simp add: Re-divide complex-is-Real-iff cmod-power2)

lemma Im-Reals-divide: \( r \in \mathbb{R} = \rightarrow \text{Im} \left(\frac{r}{z}\right) = -\frac{\text{Re} r \times \text{Im} z}{\|z\|^2} \)

by (simp add: Im-divide complex-is-Real-iff cmod-power2)

lemma series-comparison-complex:
fixes f :: nat \Rightarrow 'a::banach

assumes sg: summable g

\( \wedge n. g n \in \mathbb{R} \wedge n. \text{Re} (g n) \geq 0 \)

and fg: \( \wedge n. n \geq N \rightarrow \|f n\| \leq \|g n\| \)

shows summable f

proof

have g: \( \wedge n. \text{cmod} (g n) = \text{Re} (g n) \)

using assms by (metis abs-of-nonneg in-Real-norm)

show ?thesis

apply (rule summable-comparison-test' [where g = \( \lambda n. \text{norm} (g n) \) and N=N])

using sg

apply (auto simp: summable-def)

apply (rule-tac x = \( \text{Re} s \) in exI)

apply (auto simp: g sums-Re)

apply (metis fg g)

done

qed

111.11 Polar Form for Complex Numbers

lemma complex-unimodular-polar:

assumes norm z = 1

obtains t where \( 0 \leq t t < 2 \pi z = \text{Complex} (\cos t) (\sin t) \)

by (metis cmod-power2 one-power2 complex-surj sincos-total-2pi [of Re z Im z] assms)

111.11.1 \( \cos \theta + i \sin \theta \)

primcorec cis :: real \Rightarrow complex

where

\( \text{Re} (\text{cis} a) = \cos a \)

\| \text{Im} (\text{cis} a) = \sin a \)

lemma cis-zero [simp]: \( \text{cis} 0 = 1 \)
by (simp add: complex-eq-iff)

lemma norm-cis [simp]: norm (cis a) = 1
  by (simp add: norm-complex-def)

lemma sgn-cis [simp]: sgn (cis a) = cis a
  by (simp add: sgn-div-norm)

lemma cis-2pi [simp]: cis (2 * pi) = 1
  by (simp add: cis_mult)

lemma cis-neq-zero [simp]: cis a ≠ 0
  by (metis norm-cis norm-zero zero-neq-one)

lemma cis-cnj: cnj (cis t) = cis (-t)
  by (simp add: complex-eq-iff)

lemma cis-mult: cis a * cis b = cis (a + b)
  by (simp add: complex-eq-iff cos-add sin-add)

lemma DeMoivre: (cis a) ^ n = cis (real n * a)
  by (induct n) (simp-all add: algebra-simps cis-mult)

lemma cis-inverse [simp]: inverse (cis a) = cis (- a)
  by (simp add: complex-eq-iff)

lemma cis-divide: cis a / cis b = cis (a - b)
  by (simp add: divide-complex-def cis-mult)

lemma cos-n-Re-cis-pow-n: cos (real n * a) = Re (cis a ^ n)
  by (auto simp add: DeMoivre)

lemma sin-n-Im-cis-pow-n: sin (real n * a) = Im (cis a ^ n)
  by (auto simp add: DeMoivre)

lemma cis-pi [simp]: cis pi = -1
  by (simp add: complex-eq-iff)

lemma cis-pi-half [simp]: cis (pi / 2) = i
  by (simp add: cis_ctr complex-eq-iff)

lemma cis-minus-pi-half [simp]: cis (-ipi / 2) = -i
  by (simp add: cis_ctr complex-eq-iff)

lemma cis-multiple-2pi [simp]: n ∈ ℤ → cis (2 * pi * n) = 1
  by (auto elim!: Ints-cases simp: cis_ctr one-complex_ctr)
111.11.2 \( r(\cos \theta + i \sin \theta) \)

**definition** \( rcis :: \text{real} \Rightarrow \text{real} \Rightarrow \text{complex} \)

**where** \( rcis \ n \ a = \text{complex-of-real} \ n * \text{cis} \ a \)

**lemma** \( \text{Re-rcis} \ [\text{simp}]: \text{Re}(rcis \ n \ a) = n * \cos a \)

**by** (simp add: rcis-def)

**lemma** \( \text{Im-rcis} \ [\text{simp}]: \text{Im}(rcis \ n \ a) = n * \sin a \)

**by** (simp add: rcis-def)

**lemma** \( \text{rcis-Ex} \): \( \exists \ n \ a. \ z = rcis \ n \ a \)

**by** (simp add: complex-eq-iff polar-Ex)

**lemma** \( \text{complex-mod-rcis} \ [\text{simp}]: \text{cmod} \ (rcis \ n \ a) = |n| \)

**by** (simp add: rcis-def norm-mult)

**lemma** \( \text{cis-rcis-eq} \): \( \text{cis} \ a = rcis \ 1 \ a \)

**by** (simp add: rcis-def)

**lemma** \( \text{rcis-mult} \): \( rcis \ n1 \ a * rcis \ n2 \ b = rcis \ (n1 * n2) \ (a + b) \)

**by** (simp add: rcis-def cis-mult)

**lemma** \( \text{rcis-zero-mod} \ [\text{simp}]: \text{rcis} \ 0 \ a = 0 \)

**by** (simp add: rcis-def)

**lemma** \( \text{rcis-zero-arg} \ [\text{simp}]: \text{rcis} \ n \ 0 = \text{complex-of-real} \ n \)

**by** (simp add: rcis-def)

**lemma** \( \text{rcis-eq-zero-iff} \ [\text{simp}]: \text{rcis} \ n \ a = 0 \longleftrightarrow \ n = 0 \)

**by** (simp add: rcis-def)

**lemma** \( \text{DeMoivre2} \): \( (rcis \ n \ a) ^ n = rcis \ (n ^ a) \ (\text{real} \ n * a) \)

**by** (simp add: rcis-def power-mult-distrib DeMoivre)

**lemma** \( \text{rcis-inverse} \): \( \text{inverse}(rcis \ n \ a) = rcis \ (1 / n) \ (- \ a) \)

**by** (simp add: divide-inverse rcis-def)

**lemma** \( \text{rcis-divide} \): \( rcis \ n1 \ a / rcis \ n2 \ b = rcis \ (n1 / n2) \ (a - b) \)

**by** (simp add: rcis-def cis-divide [symmetric])

111.11.3 Complex exponential

**lemma** \( \text{exp-Reals-eq} \):

**assumes** \( z \in \mathbb{R} \)

**shows** \( \text{exp} \ z = \text{of-real} \ (\text{exp} \ (\text{Re} \ z)) \)

**using assms by** (auto elim!: Reals-cases simp: exp-of-real)

**lemma** \( \text{cis-conv-exp} \): \( \text{cis} \ b = \text{exp} \ (i * b) \)

**proof** –
have (i * complex-of-real b) ^ n \mid_R \text{fact } n =
  of-real (\cos-coeff n * b \cdot n) + i * of-real (\sin-coeff n * b \cdot n)
for n :: nat

proof -
  have i ^ n = \text{fact } n *_R (\cos-coeff n + i * \sin-coeff n)
  by (induct n)
    (simp-all add: \sin-coeff-Suc \cos-coeff-Suc complex-eq-iff Re-divide Im-divide
     field-simps
     power2-eq-square add-nonneg-0-iff)
  then show ?thesis
    by (simp add: field-simps)
qed

then show ?thesis
  using \sin-converges [of b] \cos-converges [of b]
  by (auto simp add: Complex-eq cis.ctr exp-def simp del: of-real-mult
       intro!; sums-unique sums-add sums-mult sums-of-real)
qed

lemma exp-eq-polar:
  exp z = exp (Re z) * cis (Im z)
unfolding cis-conv-exp exp-of-real [symmetric] mul-exp-exp
by (cases z) (simp add: Complex-eq)

lemma Re-exp:
  Re (exp z) = exp (Re z) * cos (Im z)
unfolding exp-eq-polar by simp

lemma Im-exp:
  Im (exp z) = exp (Re z) * sin (Im z)
unfolding exp-eq-polar by simp

lemma norm-cos-sin [simp]: norm (Complex (cos t) (sin t)) = 1
by (simp add: norm-complex-def)

lemma norm-exp-eq-Re [simp]: norm (exp z) = exp (Re z)
by (simp add: cis.code cmod-complex-polar exp-eq-polar Complex-eq)

lemma complex-exp-exists: \exists a r. z = complex-of-real r * exp a
apply (insert rcis-Ex [of z])
apply (auto simp add: exp-eq-polar rcis-def mult.assoc [symmetric])
apply (rule_tac x = i * complex-of-real a in exI)
apply auto
done

lemma exp-pi-i [simp]: exp (of-real pi * i) = -1
by (metis cis-conv-exp cis-pi mult.commute)

lemma exp-pi-i' [simp]: exp (i * of-real pi) = -1
using cis-conv-exp cis-pi by auto

lemma exp-two-pi-i [simp]: exp (2 * of-real pi * i) = 1
by (simp add: exp-eq-polar complex-eq-iff)
lemma \texttt{exp-two-pi-i} \texttt{[simp]}: \( \exp (i \cdot (\text{of-real} \ \pi \cdot 2)) = 1 \)
by (metis \texttt{exp-two-pi-i} mult.commute)

lemma \texttt{continuous-on-cis} \texttt{[continuous-intros]}:
\[ \text{continuous-on } A \ f \implies \text{continuous-on } A \ (\lambda x. \text{cis} \ (f \ x)) \]
by (auto simp: cis-conv-exp intro: continuous-intros)

111.11.4 Complex argument

definition \texttt{arg} :: \texttt{complex} \Rightarrow \texttt{real}
where \texttt{arg} \ z = (if \ z = 0 \ then \ 0 \ else \ (\text{SOME} \ a. \ \text{sgn} \ z = \text{cis} \ a \ \land \ - \pi < a \ \land \ a \leq \pi))

lemma \texttt{arg-zero}: \texttt{arg} \ 0 = 0
by (simp add: \texttt{arg-def})

lemma \texttt{arg-unique}:
assumes \texttt{sgn} \ z = \texttt{cis} \ x \ \texttt{and} \ - \pi < x \ \texttt{and} \ x \leq \pi
shows \texttt{arg} \ z = x
proof
fix \ a
define \texttt{d} where \texttt{d} = \texttt{a} – \texttt{x}
assume \texttt{a}: \texttt{sgn} \ z = \texttt{cis} \ a \ \texttt{and} \ - \pi < a \ \texttt{and} \ a \leq \pi
from \texttt{a} \ 	exttt{assms} \ have \ \texttt{–} (2*\pi) < \texttt{d} \ \texttt{and} \ \texttt{d} < 2*\pi
unfolding \texttt{d-def} \ by \ simp
moreover from \texttt{a} \ 	exttt{assms} \ have \ \texttt{cos} \ a = \texttt{cos} \ x \ \texttt{and} \ \texttt{sin} \ a = \texttt{sin} \ x
by (simp-all add: complex-eq-iff)
then have \texttt{cos}: \texttt{cos} \ d = 1
by (simp add: \texttt{d-def} \cos-diff)
moreover from \texttt{cos} \ have \ \texttt{sin} \ d = 0
by (rule cos-one-sin-zero)
ultimately have \texttt{d} = 0
by (auto simp: sin-zero-iff elim!: evenE dest!: less-2-cases)
then show \texttt{a} = \texttt{x}
by (simp add: \texttt{d-def})
qed (simp add: \texttt{assms del: Re-sgn Im-sgn})

with \texttt{(z \ \texttt{\neq} \ 0)} \ show \ \texttt{arg} \ z = \texttt{x}
by (simp add: \texttt{arg-def})

qed

lemma \texttt{arg-correct}:
assumes \texttt{z \ \texttt{\neq} \ 0}
shows \texttt{sgn} \ z = \texttt{cis} (\texttt{arg} \ z) \ \texttt{and} \ - \pi < \texttt{arg} \ z \ \texttt{and} \ \texttt{arg} \ z \leq \pi
proof (simp add: \texttt{arg-def} \assms, rule someE-ex)
obtain \( r \ a \) where \( z = \text{rcis} \ r \ a \)
using \( \text{rcis-Ex} \) by fast
with \( \text{assms} \) have \( r \neq 0 \) by auto
define \( b \) where \( b = (\text{if} \ 0 < r \ \text{then} \ a \ \text{else} \ a + \pi) \)
have \( b \): \( \text{sgn} \ z = \text{cis} \ b \)
using \( \langle r \neq 0 \rangle \) by (simp add: \( z \ \text{b-def} \ \text{rcis-def} \ \text{of-real-def} \ \text{sgn-scaleR} \ \text{sgn-if complex-eq-iff} \))

have \( \text{cis-2pi-nat}: \text{cis} \ (2 * \pi * \text{real-of-nat} \ n) = 1 \) for \( n \)
by (induct \( n \)) (simp-all add: distrib-left cis-mult [symmetric] complex-eq-iff)

have \( \text{cis-2pi-int}: \text{cis} \ (2 * \pi * \text{real-of-int} \ x) = 1 \) for \( x \)
by (cases \( x \) rule: int-diff-cases) (simp add: right-diff-distrib cis-divide [symmetric] cis-2pi-nat)

define \( c \) where \( c = b - 2 * \pi * \text{of-int} \ \lceil (b - \pi) / (2 * \pi) \rceil \)
have \( \text{sgn} \ z = \text{cis} \ c \)
by (simp add: \( b \ \text{c-def} \ \text{cis-divide} \ [\text{symmetric}] \ \text{cis-2pi-int} \))
moreover have \( - \pi < c \land c \leq \pi \)
using ceiling-correct [of \( (b - \pi) / (2 * \pi) \)]
by (simp add: \( \text{c-def} \ \text{less-divide-eq} \ \text{divide-le-eq} \ \text{algebra-simps del: le-of-int-ceiling} \))

ultimately show \( \exists \ a. \ \text{sgn} \ z = \text{cis} \ a \land -\pi < a \land a \leq \pi \)
by fast

qed

lemma \( \text{arg-bounded}: -\pi < \text{arg} \ z \land \text{arg} \ z \leq \pi \)
by (cases \( z = 0 \)) (simp-all add: arg-zero arg-correct)

lemma \( \text{cis-arg}: z \neq 0 \implies \text{cis} \ (\text{arg} \ z) = \text{sgn} \ z \)
by (simp add: arg-correct)

lemma \( \text{rcis-cmod-arg}: \text{rcis} \ (\text{cmod} \ z) \ (\text{arg} \ z) = z \)
by (cases \( z = 0 \)) (simp-all add: rcis-def cis-arg sgn-div-norm of-real-def)

lemma \( \text{cos-arg-i-mult-zero} \ [\text{simp}]: y \neq 0 \implies \Re y = 0 \implies \cos \ (\text{arg} \ y) = 0 \)
using cis-arg [of \( y \)] by (simp add: complex-eq-iff)

111.12 Complex n-th roots

lemma \( \text{bij-betw-roots-unity}: \)
assumes \( n > 0 \)
shows \( \text{bij-betw} \ \langle \lambda k. \ \text{cis} \ (2 * \pi * \text{real} \ k / \text{real} \ n) \rangle \ \{..<n\} \ \{z. \ z ^ n = 1\} \)
(is \( \text{bij-betw} \ ?f - - \))
unfolding \( \text{bij-betw-def} \)
proof (intro conjI)
show inj: inj-on \( ?f \ [..<n] \) unfolding inj-on-def
proof (safe, goal-cases)
case \( (1 \ k \ l) \)
hence \( k < n \ l < n \) by simp-all
from \( l \) have \( 1 = \ ?f \ k \ /
?f \ l \) by simp
also have \( \ldots = \text{cis} \ (2 * \pi * (\text{real} \ k - \text{real} \ l) / n) \)
using \( \text{assms} \) by (simp add: field-simps cis-divide)
finally have \( \cos (2\pi (\text{real } k - \text{real } l) / \text{real } n) = 1 \)
  by (simp add: complex-eq-iff)
then obtain \( m :: \text{int} \) where \( 2 \cdot \pi \cdot (\text{real } k - \text{real } l) / \text{real } n = \text{real-of-int } m \)
  \( \ast \cdot 2 \cdot \pi \)
  by (subst (asm) cos-one-2pi-int) blast
hence \( \text{real-of-int } (\text{int } k - \text{int } l) = \text{real-of-int } (m \ast \text{int } n) \)
  unfolding of-int-diff of-int-mult using assms
  by (simp add: nonzero-divide-eq-eq)
also note of-int-eq-iff
finally have \( \ast :: \text{abs } m \ast \text{abs } n = \text{abs } (\text{int } k - \text{int } l) \)
  by (simp add: abs-mult)
also have \( \ldots < \text{int } n \) using kl by linarith
finally have \( m = 0 \)
  using assms by simp
with \( \ast \) show \( k = l \) by simp
qed

have subset: \( ?f ' \{..<n\} \subseteq \{z::\text{complex}. z \ast n = 1\} \)
proof safe
  fix \( k :: \text{nat} \)
  have \( \text{cis } (2 \cdot \pi \cdot \text{real } k / \text{real } n) \ast n = \text{cis } (2 \cdot \pi) \ast k \)
    unfolding of-real-power using n by (simp add: c' -def power-mult-distrib)
  also have \( \text{cis } (2 \cdot \pi) = 1 \) by (simp add: complex-eq-iff)
  finally show \( ?f k \ast n = 1 \) by simp
qed

have \( n = \text{card } \{..<n\} \)
  by simp
also from assms and subset have \( \ldots \leq \text{card } \{z::\text{complex}. z \ast n = 1\} \)
  by (intro card-inj-on-le[OF inj]) (auto simp: finite-roots-unity)
finally have \( \text{card } \{z::\text{complex}. z \ast n = 1\} = n \)
  using assms by (intro antisym card-roots-unity) auto

have \( \text{card } (?f ' \{..<n\}) = \text{card } \{z::\text{complex}. z \ast n = 1\} \)
  using card inj by (subst card-image) auto
with subset and assms show \( ?f ' \{..<n\} = \{z::\text{complex}. z \ast n = 1\} \)
  by (intro card-subset-eq finite-roots-unity) auto
qed

lemma card-roots-unity-eq:
assumes \( n > 0 \)
shows \( \text{card } \{z::\text{complex}. z \ast n = 1\} = n \)
using bij-btw-same-card [OF bij-btw-roots-unity [OF assms]] by simp

lemma bij-btw-nth-root-unity:
fixes \( c :: \text{complex} \) and \( n :: \text{nat} \)
assumes \( c :: c \neq 0 \) and \( n :: n > 0 \)
defines \( c' :: \text{root } n (\text{norm } c) \ast \text{cis } (\text{arg } c / n) \)
sshows bij-btw \( \lambda z. c' \ast z \) \( \{z::\text{complex}. z \ast n = 1\} \) \( \{z::\text{complex}. z \ast n = c\} \)
proof -
  have \( c' \ast n = \text{of-real } (\text{root } n (\text{norm } c) \ast n) \ast \text{cis } (\text{arg } c) \)
    unfolding of-real-power using n by (simp add: c'-def power-mult-distrib)
DeMoivre
also from \( n \) have \( \sqrt[n]{\text{norm } c} = \text{norm } c \) by simp
also from \( c \) have of-real \( \ldots * \) cis \( \arg c = c \) by (simp add: cis-arg Complex.sgn-eq)
finally have \([\text{simp}]: c' \cdot n = c \).

show \(?\text{thesis}\) unfolding bij-betw-def inj-on-def
proof safe
fix \( z :: \text{complex} \)
assume \( z^n = 1 \)
hence \( (c' \cdot z)^n = c' \cdot n \) by (simp add: power-mult-distrib)
also have \( c' \cdot n = \text{of-real} \left( \sqrt[n]{\text{norm } c} \right)^n \) * cis \( \arg c \)
unfolding of-real-power using \( n \) by (simp add: c'-def power-mult-distrib)

DeMoivre
also from \( n \) have \( \sqrt[n]{\text{norm } c} = \text{norm } c \) by simp
also from \( c \) have \( \ldots * \) cis \( \arg c = c \) by (simp add: cis-arg Complex.sgn-eq)
finally show \( (c' \cdot z)^n = c \).

next
fix \( z \)
assume \( z = c = z^n \)
define \( z' \) where \( z' = z / c' \)
from \( c \) and \( n \) have \( c' \neq 0 \) by (auto simp: c'-def)
with \( n \) have \( z = c' \cdot z' \) and \( z' \cdot n = 1 \)
by (auto simp: z'-def power-divide z)
thus \( z \in (\lambda z. c' \cdot z)^{-1} \{ z. z^n = 1 \} \) by blast
qed (insert \( c \), \( n \), auto simp: c'-def)

lemma finite-nth-roots [intro]:
assumes \( n > 0 \)
shows \( \{ z :: \text{complex}. \ z^n = c \} \) finite
proof (cases \( c = 0 \))
case True
with assms have \( \{ z :: \text{complex}. \ z^n = c \} = \{ 0 \} \) by auto
thus \(?\text{thesis}\) by simp
next
case False
from assms have \( \{ z :: \text{complex}. \ z^n = 1 \} \) by (intro finite-roots-unity)
simp-all
also have \(?\text{this} \iff ?\text{thesis}\)
by (rule bij-betw-finite, rule bij-betw-nth-root-unity) fact+
finally show \(?\text{thesis}\.\)
qed

lemma card-nth-roots:
assumes \( c \neq 0 \) \( n > 0 \)
shows \( \{ z :: \text{complex}. \ z^n = c \} \) card \( n \)
proof
have \( \{ z. z^n = c \} = \{ z :: \text{complex}. \ z^n = 1 \} \)
by (rule sym, rule bij-betw-same-card, rule bij-betw-nth-root-unity) fact+
also have \( \ldots = n \) by (rule card-roots-unity-eq) fact+
finally show \( \text{thesis} \).

\text{qed}

\text{lemma sum-roots-unity:}

assumes \( n > 1 \)

shows \( \sum \{ z :: \text{complex}. \ z ^ \ n = 1 \} = 0 \)

\text{proof –}

define \( \omega \) where \( \omega = \text{cis} (2 \ast \pi / \text{real } n) \)

have \( \text{simp}: \omega \neq 1 \)

\text{proof}

assume \( \omega = 1 \)

with \( \text{assms} \) obtain \( k :: \text{int} \) where \( 2 \ast \pi / \text{real } n = 2 \ast \pi \ast \text{of-int } k \)

by \( (\text{auto simp: } \omega\text{-def complex-eq-iff cos-one-2pi-int}) \)

with \( \text{assms} \) have \( \text{real } n \ast \text{of-int } k = \text{of-int} (\text{int } n \ast k) \) by \( \text{simp} \)

also have \( 1 = (\text{of-int } 1 :: \text{real}) \) by \( \text{simp} \)

also note \( \text{of-int-eq-iff} \)

finally show \( False \) using \( \text{assms} \) by \( (\text{auto simp: } z\text{mult-eq-1-iff}) \)

\text{qed}

\text{have} \( \sum z \mid z ^ \ n = 1. \ z :: \text{complex} = (\sum k < n. \ \text{cis} (2 \ast \pi \ast \text{real } k / \text{real } n)) \)

\text{using} \( \text{assms} \) by \( (\text{intro sum.reindex-bij-betw [symmetric] bij-betw-roots-unity}) \)

\text{auto}

also have \( \ldots = (\sum k < n. \ \omega ^ k) \)

by \( (\text{intro sum.cong refl}) (\text{auto simp: } \omega\text{-def DeMoivre mult-ac}) \)

also have \( \ldots = (\omega ^ n - 1) / (\omega - 1) \)

by \( (\text{subst geometric-sum}) \)

also have \( \omega ^ n - 1 = \text{cis} (2 \ast \pi) - 1 \) using \( \text{assms} \) by \( (\text{auto simp: } \omega\text{-def DeMoivre}) \)

also have \( \ldots = 0 \) by \( (\text{simp add: complex-eq-iff}) \)

finally show \( \text{thesis by simp} \)

\text{qed}

\text{lemma sum-nth-roots:}

assumes \( n > 1 \)

shows \( \sum \{ z :: \text{complex}. \ z ^ \ n = c \} = 0 \)

\text{proof (cases } c = 0 \text{)}

\text{case} \( True \)

with \( \text{assms} \) have \( \{ z :: \text{complex}. \ z ^ \ n = c \} = \{ 0 \} \) by \( \text{auto} \)

also have \( \sum \ldots = 0 \) by \( \text{simp} \)

finally show \( \text{thesis} \).

\text{next}

\text{case} \( False \)

define \( c' \) where \( c' = \text{root } n \ (\text{norm } c) \ast \text{cis} (\text{arg } c / n) \)

from \( False \) and \( \text{assms} \) have \( (\sum \{ z. \ z ^ \ n = c \}) = (\sum z \mid z ^ \ n = 1. \ c' \ast z) \)

by \( (\text{subst sum.reindex-bij-betw [OF bij-betw-nth-root-unity, symmetric]}) \)

(auto simp: sum-distrib-left finite-roots-unity c'-def)

also from \( \text{assms} \) have \( \ldots = 0 \)

by \( (\text{simp add: sum-distrib-left [symmetric] sum-roots-unity}) \)
finally show ?thesis.

qed

111.13 Square root of complex numbers

primcorec csqrt :: complex ⇒ complex
  where
    Re (csqrt z) = sqrt ((cmod z + Re z) / 2)
    | Im (csqrt z) = (if Im z = 0 then 1 else sgn (Im z)) * sqrt ((cmod z - Re z) / 2)

lemma csqrt-of-real-nonneg [simp]: Im x = 0 ⇒ Re x ≥ 0 ⇒ csqrt x = sqrt (Re x)
  by (simp add: complex-eq-iff norm-complex-def)

lemma csqrt-of-real-nonpos [simp]: Im x = 0 ⇒ Re x ≤ 0 ⇒ csqrt x = i * sqrt (Re x)
  by (simp add: complex-eq-iff norm-complex-def)

lemma of-real-sqrt: x ≥ 0 ⇒ of-real (sqrt x) = csqrt (of-real x)
  by (simp add: complex-eq-iff norm-complex-def)

lemma csqrt-0 [simp]: csqrt 0 = 0
  by simp

lemma csqrt-1 [simp]: csqrt 1 = 1
  by simp

lemma csqrt-ii [simp]: csqrt i = (1 + i) / sqrt 2
  by (simp add: complex-eq-iff norm-complex-def)

lemma power2-csqrt [simp, algebra]: (csqrt z)^2 = z
  proof (cases Im z = 0)
    case True
    then show ?thesis
      using real-sqrt-pow2[of Re z] real-sqrt-pow2[of - Re z]
      by (cases 0 :: real Re z rule: linorder-cases)
        (simp-all add: complex-eq-iff Re-power2 Im-power2 power2-eq-square cmod-eq-Re)
  next
    case False
    moreover have cmod z * cmod z - Re z * Re z = Im z * Im z
      by (simp add: norm-complex-def power2-eq-square)
    moreover have |Re z| ≤ cmod z
      by (simp add: norm-complex-def)
    ultimately show ?thesis
      by (simp add: Re-power2 Im-power2 complex-eq-iff real-sgn-eq
          field-simps real-sqrt-mult[symmetric] real-sqrt-divide)
  qed
lemma `csqrt-eq-0` [simp]: `csqrt z = 0 ←→ z = 0`  
  by auto (metis `power2-csqrt` `power-eq-0-iff`) 

lemma `csqrt-eq-1` [simp]: `csqrt z = 1 ←→ z = 1`  
  by auto (metis `power2-csqrt` `power2-eq-1-iff`) 

lemma `csqrt-principal`: `0 < Re (csqrt z) ∨ Re (csqrt z) = 0 ∧ 0 ≤ Im (csqrt z)`  
  by (auto simp add: `not-less cmod-plus-Re-le-0-iff` `Im-eq-0`) 

lemma `Re-csqrt`: `0 ≤ Re (csqrt z)`  
  by (metis `csqrt-principal` `le-less`) 

lemma `csqrt-square`:  
  assumes `0 < Re b ∨ (Re b = 0 ∧ 0 ≤ Im b)`  
  shows `csqrt (b^2) = b`  
  proof (auto simp add: `power2-eq-iff`)  
    have `csqrt (b^2) = b ∨ csqrt (b^2) = -b`  
      by (simp add: `power2-eq-iff[ symmetric]`)  
    moreover have `csqrt (b^2) ≠ -b ∨ b = 0`  
      using `csqrt-principal[of b^2]` assms  
      by (intro disjCI notI) (auto simp: `complex-eq-iff`)  
    ultimately show `?thesis` by auto  
  qed  

lemma `csqrt-unique`: `w^2 = z ⇒ 0 < Re w ∨ Re w = 0 ∧ 0 ≤ Im w ⇒ csqrt z = w`  
  by (auto simp: `csqrt-square`) 

lemma `csqrt-minus` [simp]:  
  assumes `Im x < 0 ∨ (Im x = 0 ∧ 0 ≤ Re x)`  
  shows `csqrt (−x) = i * csqrt x`  
  proof  
    have `csqrt ((i * csqrt x)^2) = i * csqrt x`  
      proof (rule `csqrt-square`)  
        have `Im (csqrt x) ≤ 0`  
          using assms by (auto simp add: `cmod-eq-Re mult-le-0-iff field-simps complex-Re-le-cmod`)  
        then show `0 < Re (i * csqrt x) ∨ Re (i * csqrt x) = 0 ∧ 0 ≤ Im (i * csqrt x)`  
          by (auto simp add: `Re-csqrt simp del: csqrt.simps`)  
      qed  
    also have `(i * csqrt x)^2 = −x`  
      by (simp add: `power-mult-distrib`)  
    finally show `?thesis` .  
  qed  

Legacy theorem names  
lemmas `cmod-def` = `norm-complex-def` 

lemma `legacy-Complex-simps`:
shows Complex-eq-0: Complex \( a \ b = 0 \) \( \iff \) \( a = 0 \land b = 0 \)
and complex-add: Complex \( a \ b + \) Complex \( c \ d = \) Complex \( (a + c) \ (b + d) \)
and complex-minus: \(-\) (Complex \( a \ b) = \) Complex \( (-a) \ (-b) \)
and complex-diff: Complex \( a \ b - \) Complex \( c \ d = \) Complex \( (a - c) \ (b - d) \)
and Complex-eq-1: Complex \( a \ b = 1 \) \( \iff \) \( a = 1 \land b = 0 \)
and Complex-eq-neg-1: Complex \( a \ b = -1 \) \( \iff \) \( a = -1 \land b = 0 \)
and complex-mult: Complex \( a \ b \star \) Complex \( c \ d = \) Complex \( (a \star c - b \star d) \ (a \star d + b \star c) \)
and complex-inverse: inverse (Complex \( a \ b) = \) Complex \( (a / (a^2 + b^2)) \ (-b / (a^2 + b^2)) \)
and Complex-eq-numeral: Complex \( a \ b = \) numeral \( w \) \( \iff \) \( a = \) numeral \( w \land b = 0 \)
and Complex-eq-neg-numeral: Complex \( a \ b = -\) numeral \( w \) \( \iff \) \( a = -\) numeral \( w \land b = 0 \)
and complex-scaleR: scaleR \( r \) (Complex \( a \ b) = \) Complex \( (r \star a) \ (r \star b) \)
and i-mult-Complex: i \( \star \) Complex \( a \ b = \) Complex \( (-b) \ a \)
and Complex-mult-i: Complex \( a \ b \star i = \) Complex \( (-b) \ a \)
and i-complex-of-real: i \( \star \) complex-of-real \( r = \) Complex \( 0 \ r \)
and complex-of-real-i: complex-of-real \( r \star i = \) Complex \( 0 \ r \)
and Complex-add-complex-of-real: Complex \( x \ y + \) complex-of-real \( r = \) Complex \( (x+r) \ y \)
and complex-of-real-add-Complex: complex-of-real \( r + \) Complex \( x \ y = \) Complex \( (r+x) \ y \)
and Complex-mult-complex-of-real: Complex \( x \ y \star \) complex-of-real \( r = \) Complex \( (x \star r) \ (y \star r) \)
and complex-of-real-mult-Complex: complex-of-real \( r \star \) Complex \( x \ y = \) Complex \( (r \star x) \ (r \star y) \)
and complex-eq-cancel-iff2: (Complex \( x \ y = \) complex-of-real \( xa) = (x = xa \land y = 0 \))
and complex-cnj: cnj (Complex \( a \ b) = \) Complex \( a \ (-b) \)
and Complex-sum': sum (\( \lambda x. \) Complex \( (f \ x) \ 0) \ s = \) Complex \( (\sum f \ s) \ 0 \)
and Complex-sum: Complex \( (\sum f \ s) \ 0 = \) sum (\( \lambda x. \) Complex \( (f \ x) \ 0) \ s \)
and complex-of-real-def: complex-of-real \( r = \) Complex \( r \ 0 \)
and complex-norm: cmod (Complex \( x \ y) = \sqrt{ (x^2 + y^2) } \)
by (simp-all add: norm-complex-def field-simps complex-eq-iff Re-divide Im-divide)

lemma Complex-in-Reals: Complex \( x \) \( 0 \in \mathbb{R} \)
by (metis Reals-of-real complex-of-real-def)

end

112 MacLaurin and Taylor Series

theory MacLaurin
imports Transcendental
begin
112.1 Maclaurin’s Theorem with Lagrange Form of Remainder

This is a very long, messy proof even now that it’s been broken down into lemmas.

**lemma Maclaurin-lemma:**

\[ \theta < h \implies \exists B: \text{real}. f h = \left( \sum_{m<n} (j m / (\text{fact } m)) \ast (h \cdot m) \right) + (B \ast ((h \cdot n) / (\text{fact } n))) \]

by (rule exI [where \(x = (f h - (\sum_{m<n} (j m / (\text{fact } m)) \ast h \cdot m)) \ast (\text{fact } n) / (h \cdot n))]) simp

**lemma eq-diff-eq':** \(x = y - z \iff y = x + z\) for \(x y z \:: \text{real}\) by arith

**lemma fact-diff-Suc:** \(n < \text{Suc } m \implies \text{fact } (\text{Suc } m - n) = (\text{Suc } m - n) \ast \text{fact } (m - n)\) by (subst fact-reduce) auto

**lemma Maclaurin-lemma2:**

fixes \(B\)

assumes \(\text{DERIV}: \forall m. t. m < n \land 0 \leq t \land t \leq h \rightarrow \text{DERIV } (\text{diff } m) t :\geq \text{diff} (\text{Suc } m) t\)

and \(\text{INIT}: n = \text{Suc } k\)

defines \(\text{difg }\equiv\)

\(((\sum_{p<n-m} \text{diff } (m + p) 0 / \text{fact } p \ast t ^ p) + B \ast (t ^ (n - m) / \text{fact } (n - m)))\)

(is \(\text{difg }\equiv (\lambda m. t. \text{diff } m t = ?\text{difg } m t))\)

shows \(\forall m t. m < n \land 0 \leq t \land t \leq h \rightarrow \text{DERIV } (\text{difg } m) t :\geq \text{difg } (\text{Suc } m) t\)

**proof (rule allI impI)+**

fix \(m t\)

assume \(\text{INIT2}: m < n \land 0 \leq t \land t \leq h\)

have \(\text{DERIV } (\text{difg } m) t :\geq \text{diff } (\text{Suc } m) t - ((\sum_{x<n-m} \text{real } x \ast t ^ (x - \text{Suc } 0) \ast \text{diff } (m + x) 0 / \text{fact } x) + \text{real } (n - m) \ast t ^ (n - \text{Suc } m) \ast B / \text{fact } (n - m))\)

by (auto simp: difg-def intro!: derivative-eq-intros DERIV[rule-format, OF INIT2])

moreover

from \(\text{INIT2}\) have intub: \(\{..<n - m\} = \text{insert } 0 \{..<n - \text{Suc } m\}\) and \(0 < n - m\)

unfolding atLeast0LessThan[symmetric] by auto

have \(\sum_{x<n-m} \text{real } x \ast t ^ (x - \text{Suc } 0) \ast \text{diff } (m + x) 0 / \text{fact } x = (\sum_{x<n-m} \text{Suc } m \ast \text{real } (\text{Suc } x) \ast t ^ x \ast \text{diff } (\text{Suc } m + x) 0 / \text{fact } (\text{Suc } x))\)

unfolding intub by (subst sum.insert) (auto simp: sum.reindex)

moreover

have \(\text{fact-neq-0}: \forall x. (\text{fact } x) + \text{real } x \ast (\text{fact } x) \neq 0\)

by (metis add-pos-pos fact-gt-zero less-add-same-cancel1 less-add-same-cancel2 less-numeral-extra(3) mult-less-0-iff of-nat-less-0-iff)
have \( \forall x. \, (Suc\ x) * t \cdot x \cdot \text{diff} (Suc\ m + x) / \text{fact} (Suc\ x) = \text{diff} (Suc\ m + x) * t \cdot x / \text{fact} x \)

by (rule nonzero-divide-eq-eq[THEN iffD2]) auto

moreover

have \( (n - m) * t \cdot (n - Suc\ m) * B / \text{fact} (n - m) = B * (t \cdot (n - Suc\ m)) / \text{fact} (n - Suc\ m) \)

using \( 0 < n - m \); by (simp add: field-split-simps fact-reduce)

ultimately show \( \text{DERIV} \ (\text{difg}\ m)\ t :> \text{difg} \ (Suc\ m)\ t \)

unfolding \( \text{difg-def} \) by (simp add: mult.commute)

qed

lemma Maclaurin:

assumes \( h : 0 < h \)

and \( n : 0 < n \)

and \( \text{diff-0} : \text{diff} 0 = f \)

and \( \text{diff-Suc} : \forall m\ t.\ m < n \land 0 \leq t \land t \leq h \longrightarrow \text{DERIV} \ (\text{diff}\ m)\ t :> \text{diff} \ (Suc\ m)\ t \)

shows \( \exists t :: \text{real}.\ 0 < t \land t < h \land \)

\( f\ h = \sum (\lambda m.\ (\text{diff}\ m \cdot 0 / \text{fact}\ m) * h \cdot m) \{..<n\} + (\text{diff}\ n\ t / \text{fact}\ n) \cdot h \cdot n\)

proof

from \( n \) obtain \( m \) where \( m : n = Suc\ m \)

by (cases \( n \)) (simp add: \( n \))

from \( m \) have \( m < n \) by simp

obtain \( B \) where \( f\ h : f\ h = (\sum m < n.\ \text{diff}\ m \cdot 0 / \text{fact}\ m \cdot h \cdot m) + B * (h \cdot n / \text{fact}\ n) \)

using Maclaurin-lemma [OF \( h \)] ..

define \( g \) where \( [\text{abs-def}] : \, \text{difg}\ m\ t = \)

\( f\ t - (\sum (\lambda m.\ (\text{diff}\ m \cdot 0 / \text{fact}\ m) * t \cdot m) \{..<n\} + B * (t \cdot n / \text{fact}\ n)) \) for \( t \)

have \( g_2 : g_2 : 0 = 0 \) for \( g \) for \( h = 0 \)

by (simp-all add: \( f\ h\) \( g\)-def \( \text{lessThan-Suc-eq-insert-0} \) \( image-if \) \( \text{diff-0} \) \( \text{sum-reindex} \))

define \( \text{difg where} \ [\text{abs-def}] : \, \text{difg}\ m\ t = \)

\( \text{difg}\ m\ t - (\sum (\lambda p.\ (\text{diff}\ (m + p) \cdot 0 / \text{fact}\ p) * (t \cdot p)) \{..<n-m\} + B * (t \cdot (n - m) / \text{fact}\ (n - m))) \) for \( m\ t \)

have \( \text{difg-0} : \, \text{difg}\ 0 = g \)

by (simp add: \( \text{difg-def} \) \( g\)-def \( \text{diff-0} \))

have \( \text{difg-Suc} : \, \forall m\ t.\ m < n \land 0 \leq t \land t \leq h \longrightarrow \text{DERIV} \ (\text{difg}\ m)\ t :> \text{difg} \ (Suc\ m)\ t \)

using \( \text{diff-Suc} \) unfolding \( \text{difg-def} \) [abs-def] by (rule Maclaurin-lemma2)

have \( \text{difg-eq-0} : \forall m < n.\ \text{difg}\ m\ 0 = 0 \)

by (auto simp: \( \text{difg-def} \) \( \text{Suc-diff-le-lessThan-Suc-eq-insert-0} \) \( \text{image-if} \) \( \text{sum-reindex} \))

have \( \text{isCont-difg} : \forall m\ x.\ m < n \Longrightarrow 0 \leq x \Longrightarrow x \leq h \Longrightarrow \text{isCont} \ (\text{difg}\ m)\ x \)

by (rule DERIV-isCont [OF \( \text{difg-Suc} \) [rule-format]]) simp

have \( \text{differentiable-difg} : \forall m\ x.\ m < n \Longrightarrow 0 \leq x \Longrightarrow x \leq h \Longrightarrow \text{difg}\ m\ \text{differentiable} \ (at\ x) \)
using difg-Suc real-differentiable-def by auto
have difg-Suc-eq-0:
\[ \forall m. m < n \Rightarrow 0 \leq t \Rightarrow t \leq h \Rightarrow \text{DERIV} (\text{difg } m) t :> 0 \Rightarrow \text{difg} (\text{Suc } m) t = 0 \]
  by (rule DERIV-unique [OF difg-Suc [rule-format]]) simp

have \( \exists t. 0 < t \land t < h \land \text{DERIV} (\text{difg } m) t :> 0 \)
using (\( m < n \))
proof (induct \( m \))
case 0
show \(?case\)
proof (rule Rolle)
  show 0 < h by fact
  show difg 0 0 = difg 0 h
    by (simp add: difg-0 g2)
  show continuous-on \( \{0..h\} \) (difg 0)
    by (simp add: continuous-at-imp-continuous-on isCont-difg n)
qed (simp add: differentiable-difg n)
next
case (Suc \( m' \))
then have \( \exists t. 0 < t \land t < h \land \text{DERIV} (\text{difg } m') t :> 0 \)
  by simp
then obtain \( t \) where \( t : 0 < t t < h \land \text{DERIV} (\text{difg } m') t :> 0 \)
  by fast
have \( \exists t'. 0 < t' \land t' < t \land \text{DERIV} (\text{difg } (\text{Suc } m')) t' :> 0 \)
proof (rule Rolle)
  show 0 < t by fact
  show difg (Suc \( m' \)) 0 = difg (Suc \( m' \)) t
    using \( t : (\text{Suc } m' < n) \) by (simp add: difg-Suc-eq-0 difg-eq-0)
  have \( \forall x. 0 \leq x \land x \leq t \Rightarrow \text{isCont} (\text{difg } (\text{Suc } m')) x \)
    using \( t : (h) \) \( (\text{Suc } m' < n) \) by (simp add: isCont-difg)
  then show continuous-on \( \{0..t\} \) (difg (Suc \( m' \)))
    by (simp add: continuous-at-imp-continuous-on)
qed (use \( t : (h) \) \( (\text{Suc } m' < n) \) in (simp add: differentiable-difg n))
with \( t : (h) \) show \(?case\)
  by auto
qed
then obtain \( t \) where \( 0 < t t < h \land \text{DERIV} (\text{difg } m) t :> 0 \)
  by fast
with \( m < n \) have difg (Suc \( m \)) t = 0
  by (simp add: difg-Suc-eq-0)
show \(?thesis\)
proof (intro exI conjI)
  show 0 < t by fact
  show t < h by fact
  show \( f h = (\sum m<n. \text{diff } m 0 / (\text{fact } m) * h ^ m) + \text{diff } n t / (\text{fact } n) * h ^ n \)
    using (difg (Suc \( m \)) t = 0) by (simp add: m f-h difg-def)
qed
qed
lemma Maclaurin2:
  fixes n :: nat
  and h :: real
  assumes INIT1: 0 < h
  and INIT2: diff 0 = f
  and DERIV: \forall m t. m < n \land 0 \leq t \land t \leq h \rightarrow DERIV (diff m) t := diff (Suc m) t
  shows \exists t. 0 < t \land t \leq h \land f h = (\sum m<n. diff m 0 / (fact m) * h ^ m) + diff n t / fact n * h ^ n
proof (cases n)
  case 0
  with INIT1 INIT2 show ?thesis by fastforce
next
  case Suc
  then have n > 0 by simp
  from INIT1 this INIT2 DERIV
  have \exists t>0. t < h \land f h = (\sum m<n. diff m 0 / (fact m) * h ^ m) + diff n t / fact n * h ^ n
    by (rule Maclaurin)
  then show ?thesis by fastforce
qed

lemma Maclaurin-minus:
  fixes n :: nat and h :: real
  assumes h < 0 0 < n diff 0 = f
  and DERIV: \forall m t. m < n \land h \leq t \land t \leq 0 \rightarrow DERIV (diff m) t := diff (Suc m) t
  shows \exists t. h < t \land t < 0 \land f h = (\sum m<n. diff m 0 / (fact m) * h ^ m) + diff n t / fact n * h ^ n
proof
  Transform ABL into derivative-intros format.
  note DERIV' = DERIV-chain[OF - DERIV[rule-format], THEN DERIV-cong]
  let \?sum = \lambda t. 
    (\sum m<n. (\minus 1) ^ m * diff m (\minus 0) / (fact m) * (\minus h) ^ m) + 
    (\minus 1) ^ n * diff n (\minus t) / (fact n) * (\minus h) ^ n
  from assms have \exists t>0. t < -h \land f (\minus h) = \?sum t
    by (intro Maclaurin) (auto intro!: derivative-eq-intros DERIV')
  then obtain t where 0 < t t < - h f (\minus h) = \?sum t
    by blast
  moreover have (\minus 1) ^ n * diff n (\minus t) * (\minus h) ^ n / fact n = diff n (\minus t) * h ^ n / fact n
    by (auto simp: power-mult-distrib[symmetric])
  moreover have (\sum m<n. (\minus 1) ^ m * diff m 0 * (\minus h) ^ m / fact m) = (\sum m<n. diff m 0 * h ^ m / fact m)
    by (auto intro: sum.cong simp add: power-mult-distrib[symmetric])
  ultimately have h < - t \land - t < 0 \land
f h = (∑ m<n. diff m 0 / (fact m) * h ^ m) + diff n (- t) / (fact n) * h ^ n
by auto
then show ?thesis ..
qed

112.2 More Convenient "Bidirectional" Version.

**lemma Maclaurin-bi-le:**

fixes n :: nat and x :: real
assumes diff 0 = f
and DERIV : ∀ m t. m < n ∧ |t| ≤ |x| → DERIV (diff m) t :=> diff (Suc m) t
shows ∃ t. |t| ≤ |x| ∧ f x = (∑ m<n. diff m 0 / (fact m) * x ^ m) + diff n t / (fact n) * x ^ n
(is ∃ t. - ∧ f x = ?f x t)
proof (cases n = 0)
case True
with (diff 0 = f) show ?thesis by force
next
case False
show ?thesis proof (cases rule: linorder-cases)
assume x = 0
with (n ≠ 0) (diff 0 = f) DERIV have |0| ≤ |x| ∧ f x = ?f x 0
by auto
then show ?thesis ..
next
assume x < 0
with (n ≠ 0) DERIV have ∃ t>x. t < 0 ∧ diff 0 x = ?f x t
by (intro Maclaurin-minus) auto
then obtain t where x < t t < 0
diff 0 x = (∑ m<n. diff m 0 / (fact m) * x ^ m) + diff n t / (fact n) * x ^ n
by blast
with (x < 0) (diff 0 = f) have |t| ≤ |x| ∧ f x = ?f x t
by simp
then show ?thesis ..
next
assume x > 0
with (n ≠ 0) (diff 0 = f) DERIV have ∃ t>0. t < x ∧ diff 0 x = ?f x t
by (intro Maclaurin) auto
then obtain t where 0 < t t < x
diff 0 x = (∑ m<n. diff m 0 / (fact m) * x ^ m) + diff n t / (fact n) * x ^ n
by blast
with (x > 0) (diff 0 = f) have |t| ≤ |x| ∧ f x = ?f x t by simp
then show ?thesis ..
qed
qed

**lemma Maclaurin-all-lt:**
fixes $x :: \text{real}$
assumes $\text{INIT1}: \text{diff 0} = f$
and $\text{INIT2}: 0 < n$ and $\text{INIT3}: x \neq 0$
and $\text{DERIV}: \forall m. \text{DERIV (diff m) x} :> \text{diff (Suc m) x}$
shows $\exists t. 0 < |t| \land |t| < |x| \land f x = (\sum_{m<n} (\text{diff m 0} / \text{fact m}) \ast x ^ m) + (\text{diff n t} / \text{fact n}) \ast x ^ n$
(proof (cases rule: linorder-cases))
assume $x = 0$
with $\text{INIT3}$ show $\text{thesis}$ ..
next
assume $x < 0$
with $\text{assms}$ have $\exists t. t > x \land f x = ?f x t$
by (intro Maclaurin-minus) auto
then obtain $t$ where $t > x \land t < 0 \land f x = ?f x t$
by blast
with $(x < 0)$ have $0 < |t| \land |t| < |x| \land f x = ?f x t$
by simp
then show $\text{thesis}$ ..
next
assume $x > 0$
with $\text{assms}$ have $\exists t. t > 0 \land t < x \land f x = ?f x t$
by (intro Maclaurin) auto
then obtain $t$ where $t > 0 \land t < x \land f x = ?f x t$
by blast
with $(x > 0)$ have $0 < |t| \land |t| < |x| \land f x = ?f x t$
by simp
then show $\text{thesis}$ ..
qed

lemma Maclaurin-zero: $x = 0 \implies n \neq 0 \implies (\sum_{m<n} (\text{diff m 0} / \text{fact m}) \ast x ^ m) = \text{diff 0 0}$
for $x :: \text{real}$ and $n :: \text{nat}$
by simp

lemma Maclaurin-all-le:
fixes $x :: \text{real}$ and $n :: \text{nat}$
assumes $\text{INIT}: \text{diff 0} = f$
and $\text{DERIV}: \forall m. \text{DERIV (diff m) x} :> \text{diff (Suc m) x}$
shows $\exists t. |t| \leq |x| \land f x = (\sum_{m<n} (\text{diff m 0} / \text{fact m}) \ast x ^ m) + (\text{diff n t} / \text{fact n}) \ast x ^ n$
(is $\exists t. - \neq f x t$)
(proof (cases $n = 0$))
case True with $\text{INIT}$ show $\text{thesis}$ by force
next
case False
show ?thesis
proof (cases $x = 0$)
  case True
  with ($n \neq 0$) have ($\sum m < n$. $\text{diff } m \ 0 / (\text{fact } m) * x ^ m = \text{diff } 0 \ 0$)
    by (intro Maclaurin-zero) auto
  with INIT ($x = 0$) ($n \neq 0$) have $|0| \leq |x| \land f \ x = \ ?f \ x \ 0$
    by force
  then show ?thesis ..
next
  case False
  with INIT ($n \neq 0$) DERIV have $\exists t. \ 0 < |t| \land |t| < |x| \land f \ x = \ ?f \ x \ t$
    by (intro Maclaurin-all-lt) auto
  then obtain $t$ where $0 < |t| \land |t| < |x| \land f \ x = \ ?f \ x \ t$
    by simp
  then show ?thesis ..
qed
qed

lemma Maclaurin-all-le-objl:
  $\text{diff } 0 = f \land (\forall m \ x. \ \text{DERIV } (\text{diff } m) \ x :> \text{diff } (\text{Suc } m) \ x)$ $\longrightarrow$
  $(\exists t :: \text{real}. \ |t| \leq |x| \land f \ x = (\sum m < n. \ (\text{diff } m \ 0 / \text{fact } m) * x ^ m) + (\text{diff } n \ t / \text{fact } n) * x ^ n)$
  for $x :: \text{real}$ and $n :: \text{nat}$
  by (blast intro: Maclaurin-all-le)

112.3 Version for Exponential Function

lemma Maclaurin-exp-lt:
  fixes $x :: \text{real}$ and $n :: \text{nat}$
  shows $x \neq 0 \implies n > 0 \implies$
  $(\exists t :: \text{real}. \ |t| \leq |x| \land exp \ x = (\sum m < n. \ (x ^ m) / \text{fact } m) + (\exp t / \text{fact } n) * x ^ n)$
  using Maclaurin-all-lt [where $\text{diff } = \lambda n. \ \exp$ and $f = \exp$ and $x = x$ and $n = n$] by auto

lemma Maclaurin-exp-le:
  fixes $x :: \text{real}$ and $n :: \text{nat}$
  shows $\exists t :: |t| \leq |x| \land exp \ x = (\sum m < n. \ (x ^ m) / \text{fact } m) + (\exp t / \text{fact } n) * x ^ n$
  using Maclaurin-all-le-objl [where $\text{diff } = \lambda n. \ \exp$ and $f = \exp$ and $x = x$ and $n = n$] by auto

corollary exp-lower-Taylor-quadratic: $0 \leq x \implies 1 + x + x^2 / 2 \leq \exp x$
  for $x :: \text{real}$
  using Maclaurin-exp-le [of $x \ 3$] by (auto simp: numeral-3-eq-3 power2-eq-square)

corollary ln-2-less-1: $\ln 2 < (1 :: \text{real})$
proof –
  have 2 < 5/(2::real) by simp
  also have 5/2 ≤ exp (1::real) using exp-lower-Taylor-quadratic[of 1, simplified]
  by simp
  finally have exp (ln 2) < exp (1::real) by simp
  thus ln 2 < (1::real) by (subst (asm) exp-less-cancel-iff) simp
qed

112.4 Version for Sine Function

lemma mod-exhaust-less-4: m mod 4 = 0 ∨ m mod 4 = 1 ∨ m mod 4 = 2 ∨ m mod 4 = 3
  for m :: nat
  by auto

It is unclear why so many variant results are needed.

lemma sin-expansion-lemma: sin (x + real (Suc m) * pi / 2) = cos (x + real m * pi / 2)
  by (auto simp: cos-add sin-add add-divide-distrib distrib-right)

lemma Maclaurin-sin-expansion2:
  ∃ t. |t| ≤ |x| ∧ |t| < |x| ∧ sin x = (∑ m<n. sin-coeff m * x ^ m) + (sin (t + 1/2 * real n * pi) / fact n) * x ^ n
proof (cases n = 0 ∨ x = 0)
  case False
  let ?diff = λn x. sin (x + 1/2 * real n * pi)
  have ∃ t. 0 < |t| ∧ |t| < |x| ∧ sin x =
    (∑ m<n. (?diff 0 m / fact m) * x ^ m) + (?diff n t / fact n) * x ^ n
  proof (rule Maclaurin-all-lt)
    show ∀ m. ((λt. sin (t + 1/2 * real m * pi)) has-real-derivative
      sin (x + 1/2 * real (Suc m) * pi)) (at x)
      by (rule allI derivative-eq-intros | use sin-expansion-lemma in force)+
  qed (use False in auto)
  then show ?thesis
    apply (rule ex-forward, simp)
    apply (rule sum.cong[OF refl])
    apply (auto simp: sin-coeff-def sin-zero-iff elim: oddE simp del: of-nat-Suc)
    done
  qed auto

lemma Maclaurin-sin-expansion:
  ∃ t. sin x = (∑ m<n. sin-coeff m * x ^ m) + (sin (t + 1/2 * real n * pi) / fact n) * x ^ n
  using Maclaurin-sin-expansion2 [of x n] by blast

lemma Maclaurin-sin-expansion3:
  assumes n > 0 x > 0
  shows ∃ t. 0 < t ∧ t < x ∧
\[ \sin x = \left( \sum_{m<n} \sin-coeff m \times x^m \right) + \left( \sin \left( t + \frac{1}{2} \times \text{real } n \times \pi \right) / \text{fact } n \right) \times x^n \]

**proof**

\begin{itemize}
  \item [\text{let}] \( \text{?diff} = \lambda n. \sin (x + \frac{1}{2} \times \text{real } n \times \pi) \)
  \item [\text{have}] \( \exists t. \ 0 < t \land t < x \land \sin x = \left( \sum_{m<n} \text{?diff } m \ / \ (\text{fact } m) \times x^m \right) + \text{?diff } n \times t / \text{fact } n \times x^n \)
\end{itemize}

**proof** (rule Maclaurin)

\begin{itemize}
  \item [\text{show}] \forall m \cdot m < n \land 0 \leq t \land t \leq x \rightarrow \ ((\lambda u. \sin (u + \frac{1}{2} \times \text{real } m \times \pi)) \ \text{has-real-derivative} \sin (t + \frac{1}{2} \times \text{real } (\text{Suc } m) \times \pi)) \ (\text{at } t)
  \item [\text{apply}] (simp add: sin-expansion-lemma del: of-nat-Suc)
  \item [\text{apply}] (force intro!: derivative-eq-intros)
  \item [\text{done}]
\end{itemize}

**qed** (use assms in auto)

**then show** \(?thesis

\begin{itemize}
  \item [\text{apply}] (rule ex-forward, simp)
  \item [\text{apply}] (rule sum.cong[of refl])
  \item [\text{apply}] (auto simp: sin-coeff-def sin-zero-iff elim: oddE simp del: of-nat-Suc)
  \item [\text{done}]
\end{itemize}

**qed**

**lemma** Maclaurin-sin-expansion4:

**assumes** \( 0 < x \)

**shows** \( \exists t. \ 0 < t \land t \leq x \land \sin x = \left( \sum_{m<n} \text{?diff } m \ / \ (\text{fact } m) \times x^m \right) + \text{?diff } n \times t / \text{fact } n \times x^n \)

**proof**

\begin{itemize}
  \item [\text{let}] \( \text{?diff} = \lambda n. \sin (x + \frac{1}{2} \times \text{real } n \times \pi) \)
  \item [\text{have}] \( \exists t. \ 0 < t \land t \leq x \land \sin x = \left( \sum_{m<n} \text{?diff } m \ / \ (\text{fact } m) \times x^m \right) + \text{?diff } n \times t / \text{fact } n \times x^n \)
\end{itemize}

**proof** (rule Maclaurin2)

\begin{itemize}
  \item [\text{show}] \forall m \cdot m < n \land 0 \leq t \land t \leq x \rightarrow \ ((\lambda u. \sin (u + \frac{1}{2} \times \text{real } m \times \pi)) \ \text{has-real-derivative} \sin (t + \frac{1}{2} \times \text{real } (\text{Suc } m) \times \pi)) \ (\text{at } t)
  \item [\text{apply}] (simp add: sin-expansion-lemma del: of-nat-Suc)
  \item [\text{apply}] (force intro!: derivative-eq-intros)
  \item [\text{done}]
\end{itemize}

**qed** (use assms in auto)

**then show** \(?thesis

\begin{itemize}
  \item [\text{apply}] (rule ex-forward, simp)
  \item [\text{apply}] (rule sum.cong[of refl])
  \item [\text{apply}] (auto simp: sin-coeff-def sin-zero-iff elim: oddE simp del: of-nat-Suc)
  \item [\text{done}]
\end{itemize}

**qed**

**112.5 Maclaurin Expansion for Cosine Function**

**lemma** sumr-cos-zero-one [simp]: \( \sum_{m<Suc n} \cos-coeff m \times 0^m = 1 \)

\begin{itemize}
  \item [by] (induct n) auto
\end{itemize}
lemma cos-expansion-lemma: \( \cos (x + \text{real} \ (\text{Suc} \ m) * \pi / 2) = -\sin (x + \text{real} \ m * \pi / 2) \)
by (auto simp: cos-add sin-add distrib-right add-divide-distrib)

lemma Maclaurin-cos-expansion:
\( \exists t :: \text{real}. \ |t| \leq |x| \wedge \cos x = (\sum_{m<n} \cos\text{-coeff} \ m * x ^ m) + (\cos(t + 1/2 * \text{real} \ n * \pi) / \text{fact} \ n) * x ^ n \)
proof (cases \(n = 0 \vee x = 0\))
case False
let \(?diff = \lambda n. \cos (x + 1/2 * \text{real} \ n * \pi)\)
have \(\exists t. \ 0 < |t| \wedge |t| < |x| \wedge \cos x = (\sum_{m<n} (?diff \ m 0 / \text{fact} \ m) * x ^ m) + (?diff \ n \ t / \text{fact} \ n) * x ^ n\)
proof (rule Maclaurin-all-lt)
apply (rule \(\forall \ m \ x. \ (\lambda t. \cos (t + 1/2 * \text{real} \ m * \pi)) \text{ has-real-derivative}\)
\(\cos (x + 1/2 * \text{real} \ (\text{Suc} \ m) * \pi)) \ (at \ t)\)
apply (rule allI derivative-eq-intros | simp)+
using cos-expansion-lemma by force
qed (use False in auto)
then show \(?thesis\)
apply (rule ex-forward, simp)
apply (rule sum.cong[OF refl])
apply (auto simp: cos-coeff-def cos-zero-iff elim: evenE simp del: of-nat-Suc)
done
qed auto

lemma Maclaurin-cos-expansion2:
assumes \(x > 0 \ n > 0\)
shows \(\exists t. \ 0 < t \wedge t < x \wedge \cos x = (\sum_{m<n} \cos\text{-coeff} \ m * x ^ m) + (\cos(t + 1/2 * \text{real} \ n * \pi) / \text{fact} \ n) * x ^ n\)
proof –
let \(?diff = \lambda n. \cos (x + 1/2 * \text{real} \ n * \pi)\)
have \(\exists t. \ 0 < t \wedge t < x \wedge \cos x = (\sum_{m<n} (?diff \ m 0 / \text{fact} \ m) * x ^ m) + (?diff \ n \ t / \text{fact} \ n) * x ^ n\)
proof (rule Maclaurin)
show \(\forall \ m \ t. \ m < n \wedge 0 \leq t \wedge t \leq x \rightarrow ((\lambda u. \cos (u + 1/2 * \text{real} \ m * \pi)) \text{ has-real-derivative}\)
\(\cos (t + 1/2 * \text{real} \ (\text{Suc} \ m) * \pi)) \ (at \ t)\)
by (simp add: cos-expansion-lemma del: of-nat-Suc)
qed (use assms in auto)
then show \(?thesis\)
apply (rule ex-forward, simp)
apply (rule sum.cong[OF refl])
apply (auto simp: cos-coeff-def cos-zero-iff elim: evenE)
done
qed

lemma Maclaurin-minus-cos-expansion:
assumes \( n > 0 \) \( x < 0 \)
shows \( \exists t. \ x < t \wedge t < 0 \wedge \cos x = (\sum_{m<n.} \cos\text{-coeff } m \times x^{-m}) + ((\cos (t + 1/2 \times \text{real } n \times \pi) / \text{fact } n) \times x^{-n}) \)

proof –
let \(?\text{diff } = \lambda n \cdot \cos (x + 1/2 \times \text{real } n \times \pi)\)

have \( \exists t. \ x < t \wedge t < 0 \wedge \cos x = (\sum_{m<n.} \ ?\text{diff } m \ 0 / (\text{fact } m) \times x^{-m}) + \ ?\text{diff } n t / (\text{fact } n) \times x^{-n} \)

proof (rule Maclaurin-minus)

show \( \forall m \ t. \ m < n \wedge x \leq t \wedge t \leq 0 \rightarrow ((\lambda u. \cos (u + 1/2 \times \text{real } m \times \pi)) \text{ has-real-derivative} \cos (t + 1/2 \times \text{real } \text{Suc } m \times \pi)) (at t) \)

by (simp add: cos-expansion-lemma del: of-nat-Suc)

qed (use assms in auto)

then show \( \text{?thesis} \)

apply (rule ex-forward, simp)

apply (rule sum.cong[OF refl])

apply (auto simp: cos-coeff-def cos-zero-iff elim: evenE)

done

qed

lemma \( \text{sin-bound-lemma: } x = y \implies |u| \leq v \implies |(x + u) - y| \leq v \)

for \( x \ y \ u \ v :: \text{real} \)

by auto

lemma Maclaurin-sin-bound: \( |\sin x - (\sum_{m<n.} \sin\text{-coeff } m \times x^{-m})| \leq \text{inverse} (\text{fact } n) \times |x|^{-n} \)

proof –

have \( \text{est: } x \leq 1 \implies 0 \leq y \implies x \times y \leq 1 \times y \) \( \text{for } x \ y :: \text{real} \)

by (rule mult-right-mono) simp-all

let \(?\text{diff } = \lambda n::\text{nat}: (x::\text{real})\).

if \( n \text{ mod } 4 = 0 \) then \( \sin x \)
else if \( n \text{ mod } 4 = 1 \) then \( \cos x \)
else if \( n \text{ mod } 4 = 2 \) then \( -\sin x \)
else \( -\cos x \)

have \( \text{diff-0: } ?\text{diff } 0 = \sin \) \( \text{by simp} \)

have \( \text{DERIV (}\ ?\text{diff } m \ x ) = ?\text{diff (Suc } m \ x) \) \( \text{for } m \text{ and } x \)

using \( \text{mod-exhaust-less-4 [of } m \] \)

by (auto simp: mod-Suc_intro: derivative-eq-intros)

then have \( \text{DERIV-diff: } \forall m \ x. \ \text{DERIV (}\ ?\text{diff } m \ x = ?\text{diff (Suc } m \ x) \)

by blast

from \( \text{Maclaurin-all-le [OF diff-0 DERIV-diff]} \)

obtain \( t \) where \( \text{tl: } |t| \leq |x| \)

and \( \text{tl2: } \sin x = (\sum_{m<n.} \ ?\text{diff } m \ 0 / (\text{fact } m) \times x^{-m}) + \ ?\text{diff } n t / (\text{fact } n) \times x^{-n} \)

* x ~ n
by fast
have diff-m-0: \( \text{diff} m \, 0 = (\text{if even } m \text{ then } 0 \text{ else } (-1)^{\frac{m - Suc \, 0}{2}}) \)
for \( m \)
using mod-exhaust-less-4 [of m]
by (auto simp: minus-one-power-iff even-even-mod-4-iff [of m] dest: even-mod-4-div-2 odd-mod-4-div-2)
show ?thesis
unfolding sin-coeff-def
apply (subst t2)
apply (rule sin-bound-lemma)
apply (rule sum.cong [OF refl])
apply (subst diff-m-0, simp)
using est
apply (auto intro: mult-right-mono [where b=1, simplified] mult-right-mono simp: ac-simps divide-inverse power-abs [symmetric] abs-mult)
done
qed

113 Taylor series

We use MacLaurin and the translation of the expansion point \( c \) to 0 to prove Taylor’s theorem.

lemma Taylor-ap:
assumes INIT: \( n > 0 \, \text{diff} \, 0 = f \)
and DERIV: \( \forall m \, t. \, m < n \wedge a \leq t \wedge t \leq b \longrightarrow \text{DERIV} (\text{diff} \, m) \, t \, > (\text{diff} \, (\text{Suc} \, m) \, t) \)
and INTERV: \( a \leq c \, c < b \)
shows \( \exists t::\text{real}. \, c < t \wedge t < b \wedge f \, b = (\sum_{m<n. (\text{diff} \, m \, c / \text{fact} \, m) * (b - c) ^ m} + (\text{diff} \, n \, t / \text{fact} \, n) * (b - c) ^ n ) \)
proof –
from INTERV have \( 0 < b - c \) by arith
moreover from INIT have \( n > 0 \, (\lambda \, m \, x. \, \text{diff} \, m \, (x + c)) \, 0 = (\lambda x. \, f \, (x + c)) \)
by auto
moreover have \( \forall m \, t. \, m < n \wedge 0 \leq t \wedge t \leq b - c \longrightarrow \text{DERIV} (\lambda x. \, \text{diff} \, m \, (x + c)) \, t \, > \text{diff} \, (\text{Suc} \, m) \, (t + c) \)
proof (intro strip)
fix \( m \, t \)
assume \( m < n \wedge 0 \leq t \wedge t \leq b - c \)
with DERIV and INTERV have DERIV (\text{diff} \, m) \, (t + c) \, > \text{diff} \, (\text{Suc} \, m) \, (t + c)
by auto
moreover from DERIV-ident and DERIV-const have DERIV (\lambda x. \, x + c) \, t \, > 1 + 0
by (rule DERIV-add)
ultimately have DERIV (\lambda x. \, \text{diff} \, m \, (x + c)) \, t \, > \text{diff} \, (\text{Suc} \, m) \, (t + c) \, * \, (1 + 0) \)
by (rule DERIV-chain2)
then show \( \text{DERIV} (\lambda x. \text{diff} m (x + c)) t :> \text{diff} (\text{Suc} m) (t + c) \)
  by simp
qed
ultimately obtain \( x \) where
\[
0 < x \land x < b - c \land 
\frac{f (b - c + c)}{\text{fact } n} + \text{diff} n (x + c) / \text{fact } n \ast (b - c) \ast n
\]
by (rule Maclaurin [THEN exE])
then have \( c < x + c \land x + c < b \land f b = \)
\[
(\sum_{m < n.} \text{diff} m c / \text{fact } m \ast (b - c) \ast m) + \text{diff} n (x + c) / \text{fact } n \ast (b - c) \ast n
\]
by fastforce
then show \( \text{thesis} \) by fastforce
qed

lemma \( \text{Taylor-down} \):
fixes \( a :: \text{real} \) and \( n :: \text{nat} \)
assumes \( \text{INIT}: n > 0 \land \text{diff} 0 = f \)
and \( \text{DERIV}: (\forall m t. m < n \land a \leq t \land t \leq b \rightarrow \text{DERIV} (\text{diff} m) t :> \text{diff} (\text{Suc} m) t) \)
and \( \text{INTERV}: a < c \land c \leq b \)
shows \( \exists t. a < t \land t < c \land 
\frac{f a}{\text{fact } n} (\sum_{m < n.} \text{diff} m c / \text{fact } m) \ast (a - c) \ast m + \text{diff} n t / \text{fact } n \ast (a - c) \ast n
\)
proof -
from \( \text{INTERV} \) have \( a - c < 0 \) by arith
moreover from \( \text{INIT} \) have \( n > 0 \land (\lambda m x. \text{diff} m (x + c)) 0 = (\lambda x. f (x + c)) \)
  by auto
moreover
have \( \forall m t. m < n \land a - c \leq t \land t \leq 0 \rightarrow \text{DERIV} (\lambda x. \text{diff} m (x + c)) t :> \text{diff} (\text{Suc} m) (t + c) \)
proof (rule allI impl)+
  fix \( m t \)
  assume \( m < n \land a - c \leq t \land t \leq 0 \)
  with \( \text{DERIV} \land \text{INTERV} \) have \( \text{DERIV} (\text{diff} m) (t + c) :> \text{diff} (\text{Suc} m) (t + c) \)
  by auto
moreover from \( \text{DERIV-ident} \land \text{DERIV-const} \) have \( \text{DERIV} (\lambda x. x + c) t :> 1 + 0 \)
  by (rule DERIV-add)
ultimately have \( \text{DERIV} (\lambda x. \text{diff} m (x + c)) t :> \text{diff} (\text{Suc} m) (t + c) \ast (1 + 0) \)
  by (rule DERIV-chain2)
then show \( \text{DERIV} (\lambda x. \text{diff} m (x + c)) t :> \text{diff} (\text{Suc} m) (t + c) \)
  by simp
qed
ultimately obtain \( x \) where
theory MacLaurin

assumes INIT: n > 0 \ diff 0 = f

and DERIV: \forall m t. m < n \land a \leq t \land t \leq b \rightarrow \text{DERIV} (\text{diff} m) t :> \text{diff} (\text{Suc} m) t

and INTERV: a \leq c \land c \leq b \land a \leq x \land x \leq b \land x \neq c

shows \exists t.
\begin{align*}
& (\text{if } x < c \text{ then } x < t \land t < c \text{ else } c < t \land t < x) \land \\
& f x = \left(\sum m<n. \text{diff} m c / (\text{fact} m) \ast (x - c) \ast m\right) + \text{diff} n t / (\text{fact} n) \ast (x - c) \ast n
\end{align*}

proof (cases x < c)

case True

moreover have \forall m t. m < n \land x \leq t \land t \leq b \rightarrow \text{DERIV} (\text{diff} m) t :> \text{diff} (\text{Suc} m) t

using DERIV and INTERV by fastforce

moreover note True

moreover from INTERV have c \leq b

by simp

ultimately have \exists t>x. t < c \land f x =
\begin{align*}
& \left(\sum m<n. \text{diff} m c / (\text{fact} m) \ast (x - c) \ast m\right) + \text{diff} n t / (\text{fact} n) \ast (x - c) \ast n
\end{align*}

by (rule Taylor-down)

with True show ?thesis by simp

next

case False

note INIT

moreover have \forall m t. m < n \land a \leq t \land t \leq x \rightarrow \text{DERIV} (\text{diff} m) t :> \text{diff} (\text{Suc} m) t

using DERIV and INTERV by fastforce

moreover from INTERV have a \leq c

by arith

moreover from False and INTERV have c < x

by arith

ultimately have \exists t>c. t < x \land f x =
\begin{align*}
& \left(\sum m<n. \text{diff} m c / (\text{fact} m) \ast (x - c) \ast m\right) + \text{diff} n t / (\text{fact} n) \ast (x - c) \ast n
\end{align*}

by (rule Taylor-up)

with False show ?thesis by simp
114 Comprehensive Complex Theory

theory Complex-Main
imports
  Complex
  MacLaurin
begin
end

References


