The Supplemental Isabelle/HOL Library

October 8, 2017

Contents

1 Implementation of Association Lists **18**
- 1.1 *update* and *updates* . . . 18
- 1.2 *delete* . . . 20
- 1.3 *update-with-aux* and *delete-aux* . . . 21
- 1.4 *restrict* . . . 23
- 1.5 *clearjunk* . . . 24
- 1.6 *map-ran* . . . 25
- 1.7 *merge* . . . 26
- 1.8 *compose* . . . 27
- 1.9 *map-entry* . . . 28
- 1.10 *map-default* . . . 29

2 Pointwise instantiation of functions to algebra type classes **29**

3 Algebraic operations on sets **33**

4 Big O notation **39**
- 4.1 Definitions . . . 40
- 4.2 Sum . . . 44
- 4.3 Misc useful stuff . . . 44
- 4.4 Less than or equal to . . . 45

5 The Field of Integers mod 2 **46**
- 5.1 Bits as a datatype . . . 46
- 5.2 Type *bit* forms a field . . . 47
- 5.3 Numerals at type *bit* . . . 48
- 5.4 Conversion from *bit* . . . 48

6 Axiomatic Declaration of Bounded Natural Functors **49**

7 Generalized Corecursor Sugar (corec and friends) **49**
- 7.1 Coinduction . . . 50

8 Boolean Algebras **53**
8.1 Complement . . . 54
8.2 Conjunction . . . 54
8.3 Disjunction . . . 55
8.4 De Morgan's Laws . . . 55
8.5 Symmetric Difference . . . 55

9 A general "while" combinator **57**
9.1 Partial version . . . 57
9.2 Total version . . . 58

10 The Bourbaki-Witt tower construction for transfinite iteration **61**
10.1 Connect with the while combinator for executability on chain-finite lattices. . . . 64

11 Order on characters **66**
11.1 YXML encoding for *term* . . . 67
11.2 Test engine and drivers . . . 70

12 Pretty syntax for lattice operations **70**

13 A combinator to build partial equivalence relations from a predicate and an equivalence relation **71**

14 Formalisation of chain-complete partial orders, continuity and admissibility **72**
14.1 Continuity . . . 74
14.1.1 Theorem collection *cont-intro* . . . 75
14.2 Admissibility . . . 81
14.3 *op* = as order . . . 85
14.4 ccpo for products . . . 85
14.5 Complete lattices as ccpo . . . 90
14.6 Parallel fixpoint induction . . . 93

15 Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums **96**
15.1 The datatype universe . . . 96
15.2 Freeness: Distinctness of Constructors . . . 98
15.3 Set Constructions . . . 101

16 Bijections between natural numbers and other types **105**
16.1 Type *nat* × *nat* . . . 105
16.2 Type *nat* + *nat* . . . 107
16.3 Type *int* . . . 107

16.4 Type *nat list* . . . 108
16.5 Finite sets of naturals . . . 109
16.5.1 Preliminaries . . . 109
16.5.2 From sets to naturals . . . 110
16.5.3 From naturals to sets . . . 110
16.5.4 Proof of isomorphism . . . 110

17 Encoding (almost) everything into natural numbers 111
17.1 The class of countable types . . . 111
17.2 Conversion functions . . . 111
17.3 Finite types are countable . . . 112
17.4 Automatically proving countability of old-style datatypes . . . 112
17.5 Automatically proving countability of datatypes . . . 113
17.6 More Countable types . . . 113
17.7 The rationals are countably infinite . . . 114

18 Infinite Sets and Related Concepts 114
18.1 The set of natural numbers is infinite . . . 115
18.2 The set of integers is also infinite . . . 115
18.3 Infinitely Many and Almost All . . . 116
18.4 Enumeration of an Infinite Set . . . 118

19 Countable sets 120
19.1 Predicate for countable sets . . . 120
19.2 Enumerate a countable set . . . 120
19.3 Closure properties of countability . . . 123
19.4 Misc lemmas . . . 125
19.5 Uncountable . . . 126

20 Countable Complete Lattices 126
20.0.1 Instances of countable complete lattices . . . 131

21 Cardinal Notations 132

22 Type of (at Most) Countable Sets 132
22.1 Cardinal stuff . . . 132
22.2 The type of countable sets . . . 133
22.3 Additional lemmas . . . 140
22.3.1 *cempty* . . . 140
22.3.2 *cinsert* . . . 140
22.3.3 *cimage* . . . 140
22.3.4 bounded quantification . . . 140
22.3.5 *cUnion* . . . 141
22.4 Setup for Lifting/Transfer . . . 141
22.4.1 Relator and predicator properties . . . 141

22.4.2 Transfer rules for the Transfer package 141
22.5 Registration as BNF 142

23 Debugging facilities for code generated towards Isabelle/ML 143

24 Sequence of Properties on Subsequences 144

25 Partitions and Disjoint Sets 146
25.1 Set of Disjoint Sets 146
25.1.1 Family of Disjoint Sets 147
25.2 Construct Disjoint Sequences 148
25.3 Partitions 149
25.4 Constructions of partitions 149
25.5 Finiteness of partitions 150
25.6 Equivalence of partitions and equivalence classes 150

26 Lists with elements distinct as canonical example for datatype invariants 151
26.1 The type of distinct lists 151
26.2 Executable version obeying invariant 152
26.3 Induction principle and case distinction 153
26.4 Functorial structure 154
26.5 Quickcheck generators 154
26.6 BNF instance 154

27 Continuity and iterations 160
27.1 Continuity for complete lattices 160
27.1.1 Least fixed points in countable complete lattices . . . 163

28 Extended natural numbers (i.e. with infinity) 163
28.1 Type definition 164
28.2 Constructors and numbers 165
28.3 Addition 166
28.4 Multiplication 167
28.5 Numerals 167
28.6 Subtraction 168
28.7 Ordering 169
28.8 Cancellation simprocs 172
28.9 Well-ordering 172
28.10 Complete Lattice 173
28.11 Traditional theorem names 173

29 Liminf and Limsup on conditionally complete lattices 174
29.0.1 *Liminf* and *Limsup* 175
29.1 More Limits 179

30 Extended real number line **180**
30.1 Definition and basic properties . . . 182
30.1.1 Addition . . . 184
30.1.2 Linear order on *ereal* . . . 186
30.1.3 Multiplication . . . 192
30.1.4 Power . . . 197
30.1.5 Subtraction . . . 197
30.1.6 Division . . . 201
30.2 Complete lattice . . . 205
30.2.1 Topological space . . . 206
30.3 Relation to *enat* . . . 212
30.4 Limits on *ereal* . . . 213
30.4.1 Convergent sequences . . . 214
30.4.2 Sums . . . 219
30.4.3 Continuity . . . 226
30.4.4 liminf and limsup . . . 228
30.4.5 Tests for code generator . . . 231

31 Indicator Function **231**
31.1 The type of non-negative extended real numbers . . . 234
31.2 Defining the extended non-negative reals . . . 237
31.3 Cancellation simprocs . . . 241
31.4 Order with top . . . 241
31.5 Arithmetic . . . 243
31.6 Coercion from *real* to *ennreal* . . . 247
31.7 Coercion from *ennreal* to *real* . . . 251
31.8 Coercion from *enat* to *ennreal* . . . 251
31.9 Topology on *ennreal* . . . 252
31.10 Approximation lemmas . . . 258
31.11 *ennreal* theorems . . . 260

32 Type of finite sets defined as a subtype of sets **263**
32.1 Definition of the type . . . 263
32.2 Basic operations and type class instantiations . . . 263
32.3 Other operations . . . 266
32.4 Transferred lemmas from Set.thy . . . 267
32.5 Additional lemmas . . . 272
32.5.1 *ffUnion* . . . 272
32.5.2 *fbind* . . . 272
32.5.3 *fsingleton* . . . 272
32.5.4 *femepty* . . . 273
32.5.5 *fset* . . . 273
32.5.6 *ffilter* . . . 273
32.5.7 *fset-of-list* . . . 273

32.5.8 *finsert* . . . 274
32.5.9 *fimage* . . . 274
32.5.10 bounded quantification . . . 274
32.5.11 *fcard* . . . 275
32.5.12 *ffold* . . . 276
32.5.13 Group operations . . . 278
32.5.14 Semilattice operations . . . 278
32.6 Choice in fsets . . . 280
32.7 Induction and Cases rules for fsets . . . 280
32.8 Setup for Lifting/Transfer . . . 281
32.8.1 Relator and predicator properties . . . 281
32.8.2 Transfer rules for the Transfer package . . . 281
32.9 BNF setup . . . 283
32.10 Size setup . . . 284
32.11 Advanced relator customization . . . 285
32.11.1 Countability . . . 285
32.12 Quickcheck setup . . . 285

33 Type of finite maps defined as a subtype of maps 286
33.1 Auxiliary constants and lemmas over *map* . . . 287
33.2 Abstract characterisation . . . 289
33.3 Operations . . . 289
33.4 BNF setup . . . 299
33.5 *size* setup . . . 302
33.6 Additional operations . . . 303
33.7 Lifting/transfer setup . . . 304
33.8 View as datatype . . . 304
33.9 Code setup . . . 305
33.10 Instances . . . 306

34 Logarithm of Natural Numbers 306

35 Various algebraic structures combined with a lattice 308
35.1 Positive Part, Negative Part, Absolute Value . . . 309

36 Floating-Point Numbers 313
36.1 Real operations preserving the representation as floating point number . . . 314
36.2 Arithmetic operations on floating point numbers . . . 316
36.3 Quickcheck . . . 318
36.4 Represent floats as unique mantissa and exponent . . . 318
36.5 Compute arithmetic operations . . . 320
36.6 Lemmas for types *real*, *nat*, *int* . . . 321
36.7 Rounding Real Numbers . . . 321

36.8 Rounding Floats . . . 323
36.9 Truncating Real Numbers . . . 324
36.10Truncating Floats . . . 326
36.11Approximation of positive rationals . . . 327
36.12Division . . . 329
36.13Approximate Power . . . 329
36.14Approximate Addition . . . 331
36.15Lemmas needed by Approximate . . . 333

37 Pi and Function Sets **337**
37.1 Basic Properties of *Pi* . . . 338
37.2 Composition With a Restricted Domain: *compose* . . . 340
37.3 Bounded Abstraction: *restrict* . . . 340
37.4 Bijections Between Sets . . . 341
37.5 Extensionality . . . 342
37.6 Cardinality . . . 343
37.7 Extensional Function Spaces . . . 343
37.7.1 Injective Extensional Function Spaces . . . 345
37.7.2 Cardinality . . . 346

38 Pointwise instantiation of functions to division **346**
38.1 Syntactic with division . . . 346

39 Preorders with explicit equivalence relation **347**

40 Common discrete functions **348**
40.1 Discrete logarithm . . . 348
40.2 Discrete square root . . . 350

41 Comparing growth of functions on natural numbers by a preorder relation **351**
41.1 Motivation . . . 352
41.2 Model . . . 352
41.3 The $\lesssim$ relation . . . 352
41.4 The $\approx$ relation, the equivalence relation induced by $\lesssim$ . . . 353
41.5 The $\prec$ relation, the strict part of $\lesssim$ . . . 353
41.6 $\lesssim$ is a preorder . . . 354
41.7 Simple examples . . . 355

42 Lexical order on functions **356**

43 The 'going_to' filter **356**

44 Big sum and product over function bodies **359**
44.1 Abstract product . . . 359
44.2 Concrete sum . . . 361
44.3 Concrete product . . . 362

45 Immutable Arrays with Code Generation **363**
45.1 Code Generation . . . 364
45.2 Values extended by a bottom element . . . 366
45.3 Values extended by a top element . . . 368
45.4 Values extended by a top and a bottom element . . . 370

46 Infinite Streams **373**
46.1 prepend list to stream . . . 373
46.2 set of streams with elements in some fixed set . . . 374
46.3 nth, take, drop for streams . . . 375
46.4 unary predicates lifted to streams . . . 378
46.5 recurring stream out of a list . . . 378
46.6 iterated application of a function . . . 379
46.7 stream repeating a single element . . . 379
46.8 stream of natural numbers . . . 380
46.9 flatten a stream of lists . . . 380
46.10 merge a stream of streams . . . 381
46.11 product of two streams . . . 381
46.12 interleave two streams . . . 381
46.13 zip . . . 381
46.14 zip via function . . . 382

47 List prefixes, suffixes, and homeomorphic embedding **383**
47.1 Prefix order on lists . . . 383
47.2 Basic properties of prefixes . . . 384
47.3 Prefixes . . . 386
47.4 Longest Common Prefix . . . 387
47.5 Parallel lists . . . 388
47.6 Suffix order on lists . . . 389
47.7 Suffixes . . . 392
47.8 Homeomorphic embedding on lists . . . 394
47.9 Subsequences (special case of homeomorphic embedding) . . . 395
47.10 Appending elements . . . 396
47.11 Relation to standard list operations . . . 397
47.12 Contiguous sublists . . . 397
47.13 Parametricity . . . 400

48 Linear Temporal Logic on Streams **401**

49 Preliminaries **401**

50 Linear temporal logic **402**

51 Lists as vectors **412**
51.1 + and − . . . 412
51.2 Inner product . . . 413

52 Definitions of Least Upper Bounds and Greatest Lower Bounds 414
52.1 Rules for the Relations ∗<= and <=∗ . . . 414
52.2 Rules about the Operators *leastP*, *ub* and *lub* . . . 415
52.3 Rules about the Operators *greatestP*, *isLb* and *isGlb* . . . 416

53 An abstract view on maps for code generation. **419**
53.1 Parametricity transfer rules . . . 419
53.2 Type definition and primitive operations . . . 421
53.3 Functorial structure . . . 422
53.4 Derived operations . . . 422
53.5 Properties . . . 423
53.6 Code generator setup . . . 431

54 Adhoc overloading of constants based on their types **431**

55 Monad notation for arbitrary types **431**

56 Less common functions on lists **433**
56.1 Simproc Set-Up . . . 439

57 (Finite) Multisets **439**
57.1 The type of multisets . . . 439
57.2 Representing multisets . . . 440
57.3 Basic operations . . . 442
57.3.1 Conversion to set and membership . . . 442
57.3.2 Union . . . 444
57.3.3 Difference . . . 444
57.3.4 Min and Max . . . 446
57.3.5 Equality of multisets . . . 446
57.3.6 Pointwise ordering induced by count . . . 448
57.3.7 Intersection and bounded union . . . 451
57.3.8 Additional intersection facts . . . 452
57.3.9 Additional bounded union facts . . . 454
57.4 Replicate and repeat operations . . . 455
57.4.1 Simprocs . . . 456
57.4.2 Conditionally complete lattice . . . 457
57.4.3 Filter (with comprehension syntax) . . . 459

57.4.4 Size . . . 460
57.5 Induction and case splits . . . 462
57.5.1 Strong induction and subset induction for multisets . 463
57.6 The fold combinator . . . 463
57.7 Image . . . 464
57.8 Further conversions . . . 466
57.9 More properties of the replicate and repeat operations . . . 471
57.10 Big operators . . . 471
57.11 Alternative representations . . . 477
57.11.1 Lists . . . 477
57.12 The multiset order . . . 479
57.12.1 Well-foundedness . . . 479
57.12.2 Closure-free presentation . . . 480
57.13 The multiset extension is cancellative for multiset union . . . 480
57.14 Quasi-executable version of the multiset extension . . . 481
57.14.1 Partial-order properties . . . 481
57.14.2 Monotonicity of multiset union . . . 482
57.14.3 Termination proofs with multiset orders . . . 482
57.15 Legacy theorem bindings . . . 483
57.16 Naive implementation using lists . . . 485
57.17 BNF setup . . . 487
57.18 Size setup . . . 489
57.19 Lemmas about Size . . . 490

58 More Theorems about the Multiset Order 490
58.1 Alternative Characterizations . . . 491
58.2 Simprocs . . . 493
58.3 Additional facts and instantiations . . . 494

59 Permutations, both general and specifically on finite sets. 495
59.1 Transpositions . . . 495
59.2 Basic consequences of the definition . . . 496
59.3 Group properties . . . 497
59.4 Mapping permutations with bijections . . . 498
59.5 The number of permutations on a finite set . . . 498
59.6 Permutations of index set for iterated operations . . . 499
59.7 Various combinations of transpositions with 2, 1 and 0 common elements . . . 499
59.8 Permutations as transposition sequences . . . 499
59.9 Some closure properties of the set of permutations, with lengths 499
59.10 The identity map only has even transposition sequences . . . 500
59.11 Therefore we have a welldefined notion of parity . . . 501
59.12 And it has the expected composition properties . . . 501
59.13 A more abstract characterization of permutations . . . 502

59.14Relation to *permutes* . . . 502
59.15Hence a sort of induction principle composing by swaps . . . 503
59.16Sign of a permutation as a real number . . . 503
59.17Permuting a list . . . 503
59.18More lemmas about permutations . . . 504
59.19Sum over a set of permutations (could generalize to iteration) 506
59.20Constructing permutations from association lists . . . 506

60 Permutations of a Multiset 508
60.1 Permutations of a multiset . . . 509
60.2 Cardinality of permutations . . . 510
60.3 Permutations of a set . . . 511
60.4 Code generation . . . 513

61 Non-negative, non-positive integers and reals 515
61.1 Non-positive integers . . . 515
61.2 Non-negative reals . . . 517
61.3 Non-positive reals . . . 519

62 A generic phantom type 520

63 Cardinality of types 521
63.1 Preliminary lemmas . . . 521
63.2 Cardinalities of types . . . 521
63.3 Classes with at least 1 and 2 . . . 522
63.4 A type class for deciding finiteness of types . . . 523
63.5 A type class for computing the cardinality of types . . . 523
63.6 Instantiations for *card-UNIV* . . . 523
63.7 Code setup for sets . . . 527

64 Numeral Syntax for Types 529
64.1 Numeral Types . . . 529
64.2 Locales for for modular arithmetic subtypes . . . 530
64.3 Ring class instances . . . 532
64.4 Order instances . . . 533
64.5 Code setup and type classes for code generation . . . 534
64.6 Syntax . . . 536
64.7 Examples . . . 536

65 ω-words 537
65.1 Type declaration and elementary operations . . . 537
65.2 Subsequence, Prefix, and Suffix . . . 538
65.3 Prepending . . . 540
65.4 The limit set of an ω-word . . . 541
65.5 Index sequences and piecewise definitions . . . 543

66 Combinator syntax for generic, open state monads (single-threaded monads) 545
66.1 Motivation . . . 545
66.2 State transformations and combinators . . . 545
66.3 Monad laws . . . 546
66.4 Do-syntax . . . 547

67 Canonical order on option type 547

68 Futures and parallel lists for code generated towards Isabelle/ML 552
68.1 Futures . . . 552
68.2 Parallel lists . . . 552

69 Input syntax for pattern aliases (or "as-patterns" in Haskell) 553
69.1 Definition . . . 554
69.2 Usage . . . 554

70 Periodic Functions 555

71 Permutations as abstract type 557
71.1 Abstract type of permutations . . . 557
71.2 Identity, composition and inversion . . . 558
71.3 Orbit and order of elements . . . 560
71.4 Swaps . . . 563
71.5 Permutations specified by cycles . . . 564
71.6 Syntax . . . 564

72 Permutations 565
72.1 Some examples of rule induction on permutations . . . 565
72.2 Ways of making new permutations . . . 565
72.3 Further results . . . 566
72.4 Removing elements . . . 566

73 Additive group operations on product types 567
73.1 Operations . . . 568
73.2 Class instances . . . 569

74 Roots of real quadratics 570

75 Pretty syntax for Quotient operations 571

76 Quotient infrastructure for the set type 572
76.1 Contravariant set map (vimage) and set relator, rules for the Quotient package . . . 572

77 Quotient infrastructure for the product type **573**
77.1 Rules for the Quotient package 574

78 Quotient infrastructure for the option type **575**
78.1 Rules for the Quotient package 576

79 Quotient infrastructure for the list type **577**
79.1 Rules for the Quotient package 577

80 Quotient infrastructure for the sum type **580**
80.1 Rules for the Quotient package 580

81 Quotient types **582**
81.1 Equivalence relations and quotient types 582
81.2 Equality on quotients 583
81.3 Picking representing elements 583

82 Ramsey's Theorem **584**
82.1 Finite Ramsey theorem(s) 584
82.2 Preliminaries 584
82.2.1 "Axiom" of Dependent Choice 584
82.2.2 Partitions of a Set 585
82.3 Ramsey's Theorem: Infinitary Version 585
82.4 Disjunctive Well-Foundedness 586

83 Generic reflection and reification **586**

84 Assigning lengths to types by type classes **587**

85 Saturated arithmetic **588**
85.1 The type of saturated naturals 588

86 State monad **592**

87 Stirling numbers of first and second kind **596**
87.1 Stirling numbers of the second kind 596
87.2 Stirling numbers of the first kind 596
87.2.1 Efficient code 598

88 A decision procedure for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming **599**

89 A table-based implementation of the reflexive transitive closure **599**

90 Binary Tree **601**
90.1 *map-tree* . . . 602
90.2 *size* . . . 602
90.3 *set-tree* . . . 603
90.4 *subtrees* . . . 603
90.5 *height* and *min-height* . . . 604
90.6 *complete* . . . 604
90.7 *balanced* . . . 605
90.8 *wbalanced* . . . 605
90.9 *ipl* . . . 605
90.10List of entries . . . 605
90.11Binary Search Tree . . . 606
90.12*heap* . . . 607
90.13*mirror* . . . 607

91 Multiset of Elements of Binary Tree **607**

92 Unordered pairs **609**

93 Pointwise order on product types **613**
93.1 Pointwise ordering . . . 613
93.2 Binary infimum and supremum . . . 613
93.3 Top and bottom elements . . . 614
93.4 Complete lattice operations . . . 615
93.5 Complete distributive lattices . . . 617
93.6 Bekic's Theorem . . . 617
93.7 Finite Distributive Lattices . . . 620
93.8 Linear Orders . . . 620
93.9 Finite Linear Orders . . . 621

94 Lexicographic order on lists **622**

95 Prefix order on lists as order class instance **623**

96 Lexicographic order on product types **624**

97 Subsequence Ordering **626**
97.1 Definitions and basic lemmas . . . 626

98 Implementation of mappings with Association Lists **627**

99 Avoidance of pattern matching on natural numbers **629**
99.1 Case analysis . . . 629
99.2 Preprocessors . . . 629

100 Implementation of natural numbers as binary numerals 630
100.1 Representation . . . 630
100.2 Basic arithmetic . . . 630
100.3 Conversions . . . 632

101 Code generation of pretty characters (and strings) 633

102 Code generation of prolog programs 635

103 Setup for Numerals 635

104 Implementation of integer numbers by target-language integers 635

105 Implementation of natural numbers by target-language integers 642
105.1 Implementation for *nat* . . . 642

106 Implementation of natural and integer numbers by target-language integers 645

107 Abstract type of association lists with unique keys 646
107.1 Preliminaries . . . 646
107.2 Type (*'key*, *'value*) *alist* . . . 646
107.3 Primitive operations . . . 646
107.4 Abstract operation properties . . . 647
107.5 Further operations . . . 647
107.5.1 Equality . . . 647
107.5.2 Size . . . 648
107.6 Quickcheck generators . . . 648

108 alist is a BNF 650

109 Multisets partially implemented by association lists 650

110 Implementation of Red-Black Trees 655
110.1 Datatype of RB trees . . . 655
110.2 Tree properties . . . 655
110.2.1 Content of a tree . . . 655
110.2.2 Search tree properties . . . 656
110.2.3 Tree lookup . . . 657
110.2.4 Red-black properties . . . 659
110.3 Insertion . . . 659
110.4 Deletion . . . 663
110.5 Modifying existing entries . . . 668

110.6 Mapping all entries . . . 669
110.7 Folding over entries . . . 670
110.8 Bulkloading a tree . . . 670
110.9 Building a RBT from a sorted list . . . 671
110.10 Union and intersection of sorted associative lists . . . 677
110.11 Code generator setup . . . 681

111 Abstract type of RBT trees 683
111.1 Type definition . . . 683
111.2 Primitive operations . . . 683
111.3 Derived operations . . . 684
111.4 Abstract lookup properties . . . 684
111.5 Quickcheck generators . . . 687
111.6 Hide implementation details . . . 687

112 Implementation of mappings with Red-Black Trees 688
112.1 Data type and invariant . . . 688
112.2 Operations . . . 688
112.3 Invariant preservation . . . 689
112.4 Map Semantics . . . 689

113 Implementation of sets using RBT trees 690

114 Definition of code datatype constructors 690

115 Deletion of already existing code equations 690

116 Lemmas 690
116.1 Auxiliary lemmas . . . 690
116.2 fold and filter . . . 690
116.3 foldi and Ball . . . 691
116.4 foldi and Bex . . . 691
116.5 folding over non empty trees and selecting the minimal and maximal element . . . 692

117 Code equations 695

118 Common constants 700

119 Pairs 701

120 Filters 701

121 Bounded quantifiers 701

122 Operations on Predicates 701

123 Setup for Numerals **701**

124 Arithmetic operations **701**
124.1 Arithmetic on naturals and integers 701
124.2 Inductive definitions for ordering on naturals 702

125 Alternative list definitions **702**
125.1 Alternative rules for *length* 702
125.2 Alternative rules for *list-all2* 703
125.3 Alternative rules for membership in lists 703

126 Setup for String.literal **703**

127 Simplification rules for optimisation **703**

128 A Prototype of Quickcheck based on the Predicate Compiler **704**

129 TFL: recursive function definitions **704**
129.1 Lemmas for TFL 704
129.2 Rule setup 705

130 Refute **706**

1 Implementation of Association Lists

theory *AList*
imports *Main*
begin

context
begin

The operations preserve distinctness of keys and function *clearjunk* distributes over them. Since *clearjunk* enforces distinctness of keys it can be used to establish the invariant, e.g. for inductive proofs.

1.1 *update* and *updates*

qualified primrec *update* :: $'key \Rightarrow 'val \Rightarrow ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$
where
$update\ k\ v\ [] = [(k, v)]$
| $update\ k\ v\ (p \# ps) = (if\ fst\ p = k\ then\ (k, v) \# ps\ else\ p \# update\ k\ v\ ps)$

lemma *update-conv'*: $map\text{-}of\ (update\ k\ v\ al) = (map\text{-}of\ al)(k \mapsto v)$
⟨*proof*⟩

corollary *update-conv*: $map\text{-}of\ (update\ k\ v\ al)\ k' = ((map\text{-}of\ al)(k \mapsto v))\ k'$
⟨*proof*⟩

lemma *dom-update*: $fst\ `\ set\ (update\ k\ v\ al) = \{k\} \cup fst\ `\ set\ al$
⟨*proof*⟩

lemma *update-keys*:
$map\ fst\ (update\ k\ v\ al) =$
$(if\ k \in set\ (map\ fst\ al)\ then\ map\ fst\ al\ else\ map\ fst\ al\ @\ [k])$
⟨*proof*⟩

lemma *distinct-update*:
assumes $distinct\ (map\ fst\ al)$
shows $distinct\ (map\ fst\ (update\ k\ v\ al))$
⟨*proof*⟩

lemma *update-filter*:
$a \neq k \Longrightarrow update\ k\ v\ [q \leftarrow ps.\ fst\ q \neq a] = [q \leftarrow update\ k\ v\ ps.\ fst\ q \neq a]$
⟨*proof*⟩

lemma *update-triv*: $map\text{-}of\ al\ k = Some\ v \Longrightarrow update\ k\ v\ al = al$
⟨*proof*⟩

lemma *update-nonempty* [*simp*]: $update\ k\ v\ al \neq []$
⟨*proof*⟩

lemma *update-eqD*: *update k v al* = *update k v' al'* ⟹ *v* = *v'*
⟨*proof*⟩

lemma *update-last* [*simp*]: *update k v* (*update k v' al*) = *update k v al*
⟨*proof*⟩

Note that the lists are not necessarily the same: *update k v* (*update k' v'* []) = [(*k'*, *v'*), (*k*, *v*)] and *update k' v'* (*update k v* []) = [(*k*, *v*), (*k'*, *v'*)].

lemma *update-swap*:
k ≠ *k'* ⟹ *map-of* (*update k v* (*update k' v' al*)) = *map-of* (*update k' v'* (*update k v al*))
⟨*proof*⟩

lemma *update-Some-unfold*:
map-of (*update k v al*) *x* = *Some y* ⟷
x = *k* ∧ *v* = *y* ∨ *x* ≠ *k* ∧ *map-of al x* = *Some y*
⟨*proof*⟩

lemma *image-update* [*simp*]: *x* ∉ *A* ⟹ *map-of* (*update x y al*) ‘ *A* = *map-of al* ‘ *A*
⟨*proof*⟩ **definition** *updates* ::
'key list ⇒ *'val list* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where *updates ks vs* = *fold* (*case-prod update*) (*zip ks vs*)

lemma *updates-simps* [*simp*]:
updates [] *vs ps* = *ps*
updates ks [] *ps* = *ps*
updates (*k*#*ks*) (*v*#*vs*) *ps* = *updates ks vs* (*update k v ps*)
⟨*proof*⟩

lemma *updates-key-simp* [*simp*]:
updates (*k* # *ks*) *vs ps* =
(*case vs of* [] ⇒ *ps* | *v* # *vs* ⇒ *updates ks vs* (*update k v ps*))
⟨*proof*⟩

lemma *updates-conv'*: *map-of* (*updates ks vs al*) = (*map-of al*)(*ks*[↦]*vs*)
⟨*proof*⟩

lemma *updates-conv*: *map-of* (*updates ks vs al*) *k* = ((*map-of al*)(*ks*[↦]*vs*)) *k*
⟨*proof*⟩

lemma *distinct-updates*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*updates ks vs al*))
⟨*proof*⟩

lemma *updates-append1*[*simp*]: *size ks* < *size vs* ⟹
updates (*ks*@[*k*]) *vs al* = *update k* (*vs*!*size ks*) (*updates ks vs al*)
⟨*proof*⟩

lemma *updates-list-update-drop*[*simp*]:
size ks ≤ *i* ⟹ *i* < *size vs* ⟹
updates ks (*vs*[*i*:=*v*]) *al* = *updates ks vs al*
⟨*proof*⟩

lemma *update-updates-conv-if*:
map-of (*updates xs ys* (*update x y al*)) =
map-of
(*if x* ∈ *set* (*take* (*length ys*) *xs*)
then updates xs ys al
else (*update x y* (*updates xs ys al*)))
⟨*proof*⟩

lemma *updates-twist* [*simp*]:
k ∉ *set ks* ⟹
map-of (*updates ks vs* (*update k v al*)) = *map-of* (*update k v* (*updates ks vs al*))
⟨*proof*⟩

lemma *updates-apply-notin* [*simp*]:
k ∉ *set ks* ⟹ *map-of* (*updates ks vs al*) *k* = *map-of al k*
⟨*proof*⟩

lemma *updates-append-drop* [*simp*]:
size xs = *size ys* ⟹ *updates* (*xs* @ *zs*) *ys al* = *updates xs ys al*
⟨*proof*⟩

lemma *updates-append2-drop* [*simp*]:
size xs = *size ys* ⟹ *updates xs* (*ys* @ *zs*) *al* = *updates xs ys al*
⟨*proof*⟩

1.2 *delete*

qualified definition *delete* :: *'key* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where *delete-eq*: *delete k* = *filter* (λ(*k'*, -). *k* ≠ *k'*)

lemma *delete-simps* [*simp*]:
delete k [] = []
delete k (*p* # *ps*) = (*if fst p* = *k then delete k ps else p* # *delete k ps*)
⟨*proof*⟩

lemma *delete-conv'*: *map-of* (*delete k al*) = (*map-of al*)(*k* := *None*)
⟨*proof*⟩

corollary *delete-conv*: *map-of* (*delete k al*) *k'* = ((*map-of al*)(*k* := *None*)) *k'*
⟨*proof*⟩

lemma *delete-keys*: *map fst* (*delete k al*) = *removeAll k* (*map fst al*)
⟨*proof*⟩

lemma *distinct-delete*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*delete k al*))
⟨*proof*⟩

lemma *delete-id* [*simp*]: $k \notin fst\ `\ set\ al \Longrightarrow delete\ k\ al = al$
⟨*proof*⟩

lemma *delete-idem*: *delete k* (*delete k al*) = *delete k al*
⟨*proof*⟩

lemma *map-of-delete* [*simp*]: $k' \neq k \Longrightarrow map\text{-}of\ (delete\ k\ al)\ k' = map\text{-}of\ al\ k'$
⟨*proof*⟩

lemma *delete-notin-dom*: $k \notin fst\ `\ set\ (delete\ k\ al)$
⟨*proof*⟩

lemma *dom-delete-subset*: $fst\ `\ set\ (delete\ k\ al) \subseteq fst\ `\ set\ al$
⟨*proof*⟩

lemma *delete-update-same*: *delete k* (*update k v al*) = *delete k al*
⟨*proof*⟩

lemma *delete-update*: $k \neq l \Longrightarrow delete\ l\ (update\ k\ v\ al) = update\ k\ v\ (delete\ l\ al)$
⟨*proof*⟩

lemma *delete-twist*: *delete x* (*delete y al*) = *delete y* (*delete x al*)
⟨*proof*⟩

lemma *length-delete-le*: *length* (*delete k al*) ≤ *length al*
⟨*proof*⟩

1.3 *update-with-aux* and *delete-aux*

qualified primrec *update-with-aux* ::
$'val \Rightarrow 'key \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$
where
update-with-aux v k f [] = [(*k*, *f v*)]
| *update-with-aux v k f* (*p* # *ps*) =
(*if* (*fst p* = *k*) *then* (*k*, *f* (*snd p*)) # *ps else p* # *update-with-aux v k f ps*)

The above *delete* traverses all the list even if it has found the key. This one does not have to keep going because is assumes the invariant that keys are distinct.

qualified fun *delete-aux* :: $'key \Rightarrow ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$
where
delete-aux k [] = []
| *delete-aux k* ((*k'*, *v*) # *xs*) = (*if k* = *k' then xs else* (*k'*, *v*) # *delete-aux k xs*)

lemma *map-of-update-with-aux'*:
map-of (*update-with-aux v k f ps*) *k'* =
((*map-of ps*)(*k* ↦ (*case map-of ps k of None* ⇒ *f v* | *Some v* ⇒ *f v*))) *k'*
⟨*proof*⟩

lemma *map-of-update-with-aux*:
map-of (*update-with-aux v k f ps*) =
(*map-of ps*)(*k* ↦ (*case map-of ps k of None* ⇒ *f v* | *Some v* ⇒ *f v*))
⟨*proof*⟩

lemma *dom-update-with-aux*: *fst* ‘ *set* (*update-with-aux v k f ps*) = {*k*} ∪ *fst* ‘ *set ps*
⟨*proof*⟩

lemma *distinct-update-with-aux* [*simp*]:
distinct (*map fst* (*update-with-aux v k f ps*)) = *distinct* (*map fst ps*)
⟨*proof*⟩

lemma *set-update-with-aux*:
distinct (*map fst xs*) ⟹
set (*update-with-aux v k f xs*) =
(*set xs* − {*k*} × *UNIV* ∪ {(*k*, *f* (*case map-of xs k of None* ⇒ *v* | *Some v* ⇒ *v*))})
⟨*proof*⟩

lemma *set-delete-aux*: *distinct* (*map fst xs*) ⟹ *set* (*delete-aux k xs*) = *set xs* − {*k*} × *UNIV*
⟨*proof*⟩

lemma *dom-delete-aux*: *distinct* (*map fst ps*) ⟹ *fst* ‘ *set* (*delete-aux k ps*) = *fst* ‘ *set ps* − {*k*}
⟨*proof*⟩

lemma *distinct-delete-aux* [*simp*]: *distinct* (*map fst ps*) ⟹ *distinct* (*map fst* (*delete-aux k ps*))
⟨*proof*⟩

lemma *map-of-delete-aux'*:
distinct (*map fst xs*) ⟹ *map-of* (*delete-aux k xs*) = (*map-of xs*)(*k* := *None*)
⟨*proof*⟩

lemma *map-of-delete-aux*:
distinct (*map fst xs*) ⟹ *map-of* (*delete-aux k xs*) *k'* = ((*map-of xs*)(*k* := *None*)) *k'*
⟨*proof*⟩

lemma *delete-aux-eq-Nil-conv*: *delete-aux k ts* = [] ⟷ *ts* = [] ∨ (∃ *v*. *ts* = [(*k*, *v*)])

⟨*proof*⟩

1.4 *restrict*

qualified definition *restrict* :: *'key set* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where *restrict-eq*: *restrict A* = *filter* (λ(*k*, *v*). *k* ∈ *A*)

lemma *restr-simps* [*simp*]:
restrict A [] = []
restrict A (*p*#*ps*) = (*if fst p* ∈ *A then p* # *restrict A ps else restrict A ps*)
⟨*proof*⟩

lemma *restr-conv'*: *map-of* (*restrict A al*) = ((*map-of al*)|‘ *A*)
⟨*proof*⟩

corollary *restr-conv*: *map-of* (*restrict A al*) *k* = ((*map-of al*)|‘ *A*) *k*
⟨*proof*⟩

lemma *distinct-restr*: *distinct* (*map fst al*) ⟹ *distinct* (*map fst* (*restrict A al*))
⟨*proof*⟩

lemma *restr-empty* [*simp*]:
restrict {} *al* = []
restrict A [] = []
⟨*proof*⟩

lemma *restr-in* [*simp*]: *x* ∈ *A* ⟹ *map-of* (*restrict A al*) *x* = *map-of al x*
⟨*proof*⟩

lemma *restr-out* [*simp*]: *x* ∉ *A* ⟹ *map-of* (*restrict A al*) *x* = *None*
⟨*proof*⟩

lemma *dom-restr* [*simp*]: *fst* ‘ *set* (*restrict A al*) = *fst* ‘ *set al* ∩ *A*
⟨*proof*⟩

lemma *restr-upd-same* [*simp*]: *restrict* (−{*x*}) (*update x y al*) = *restrict* (−{*x*}) *al*
⟨*proof*⟩

lemma *restr-restr* [*simp*]: *restrict A* (*restrict B al*) = *restrict* (*A*∩*B*) *al*
⟨*proof*⟩

lemma *restr-update*[*simp*]:
map-of (*restrict D* (*update x y al*)) =
map-of ((*if x* ∈ *D then* (*update x y* (*restrict* (*D*−{*x*}) *al*)) *else restrict D al*))
⟨*proof*⟩

lemma *restr-delete* [*simp*]:
delete x (*restrict D al*) = (*if x* ∈ *D then restrict* (*D* − {*x*}) *al else restrict D al*)

⟨*proof*⟩

lemma *update-restr*:
map-of (*update x y* (*restrict D al*)) = *map-of* (*update x y* (*restrict* (*D* − {*x*}) *al*))
⟨*proof*⟩

lemma *update-restr-conv* [*simp*]:
x ∈ *D* ⟹
map-of (*update x y* (*restrict D al*)) = *map-of* (*update x y* (*restrict* (*D* − {*x*}) *al*))
⟨*proof*⟩

lemma *restr-updates* [*simp*]:
length xs = *length ys* ⟹ *set xs* ⊆ *D* ⟹
map-of (*restrict D* (*updates xs ys al*)) =
map-of (*updates xs ys* (*restrict* (*D* − *set xs*) *al*))
⟨*proof*⟩

lemma *restr-delete-twist*: (*restrict A* (*delete a ps*)) = *delete a* (*restrict A ps*)
⟨*proof*⟩

1.5 *clearjunk*

qualified function *clearjunk* :: ('*key* × '*val*) *list* ⇒ ('*key* × '*val*) *list*
where
clearjunk [] = []
| *clearjunk* (*p*#*ps*) = *p* # *clearjunk* (*delete* (*fst p*) *ps*)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *map-of-clearjunk*: *map-of* (*clearjunk al*) = *map-of al*
⟨*proof*⟩

lemma *clearjunk-keys-set*: *set* (*map fst* (*clearjunk al*)) = *set* (*map fst al*)
⟨*proof*⟩

lemma *dom-clearjunk*: *fst* ‘ *set* (*clearjunk al*) = *fst* ‘ *set al*
⟨*proof*⟩

lemma *distinct-clearjunk* [*simp*]: *distinct* (*map fst* (*clearjunk al*))
⟨*proof*⟩

lemma *ran-clearjunk*: *ran* (*map-of* (*clearjunk al*)) = *ran* (*map-of al*)
⟨*proof*⟩

lemma *ran-map-of*: *ran* (*map-of al*) = *snd* ‘ *set* (*clearjunk al*)
⟨*proof*⟩

lemma *clearjunk-update*: *clearjunk* (*update k v al*) = *update k v* (*clearjunk al*)
⟨*proof*⟩

lemma *clearjunk-updates*: *clearjunk* (*updates ks vs al*) = *updates ks vs* (*clearjunk al*)
⟨*proof*⟩

lemma *clearjunk-delete*: *clearjunk* (*delete x al*) = *delete x* (*clearjunk al*)
⟨*proof*⟩

lemma *clearjunk-restrict*: *clearjunk* (*restrict A al*) = *restrict A* (*clearjunk al*)
⟨*proof*⟩

lemma *distinct-clearjunk-id* [*simp*]: *distinct* (*map fst al*) ⟹ *clearjunk al* = *al*
⟨*proof*⟩

lemma *clearjunk-idem*: *clearjunk* (*clearjunk al*) = *clearjunk al*
⟨*proof*⟩

lemma *length-clearjunk*: *length* (*clearjunk al*) ≤ *length al*
⟨*proof*⟩

lemma *delete-map*:
assumes ⋀*kv*. *fst* (*f kv*) = *fst kv*
shows *delete k* (*map f ps*) = *map f* (*delete k ps*)
⟨*proof*⟩

lemma *clearjunk-map*:
assumes ⋀*kv*. *fst* (*f kv*) = *fst kv*
shows *clearjunk* (*map f ps*) = *map f* (*clearjunk ps*)
⟨*proof*⟩

1.6 *map-ran*

definition *map-ran* :: ('*key* ⇒ '*val* ⇒ '*val*) ⇒ ('*key* × '*val*) *list* ⇒ ('*key* × '*val*) *list*
where *map-ran f* = *map* (λ(*k*, *v*). (*k*, *f k v*))

lemma *map-ran-simps* [*simp*]:
map-ran f [] = []
map-ran f ((*k*, *v*) # *ps*) = (*k*, *f k v*) # *map-ran f ps*
⟨*proof*⟩

lemma *dom-map-ran*: *fst* ‘ *set* (*map-ran f al*) = *fst* ‘ *set al*
⟨*proof*⟩

lemma *map-ran-conv*: *map-of* (*map-ran f al*) *k* = *map-option* (*f k*) (*map-of al k*)
⟨*proof*⟩

lemma *distinct-map-ran*: *distinct* (*map fst al*) $\Longrightarrow$ *distinct* (*map fst* (*map-ran f al*))
⟨*proof*⟩

lemma *map-ran-filter*: *map-ran f* [*p*←*ps*. *fst p* $\neq$ *a*] = [*p*←*map-ran f ps*. *fst p* $\neq$ *a*]
⟨*proof*⟩

lemma *clearjunk-map-ran*: *clearjunk* (*map-ran f al*) = *map-ran f* (*clearjunk al*)
⟨*proof*⟩

1.7 *merge*

qualified definition *merge* :: (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where *merge qs ps* = *foldr* (λ(*k*, *v*). *update k v*) *ps qs*

lemma *merge-simps* [*simp*]:
merge qs [] = *qs*
merge qs (*p*#*ps*) = *update* (*fst p*) (*snd p*) (*merge qs ps*)
⟨*proof*⟩

lemma *merge-updates*: *merge qs ps* = *updates* (*rev* (*map fst ps*)) (*rev* (*map snd ps*)) *qs*
⟨*proof*⟩

lemma *dom-merge*: *fst* ‘ *set* (*merge xs ys*) = *fst* ‘ *set xs* ∪ *fst* ‘ *set ys*
⟨*proof*⟩

lemma *distinct-merge*: *distinct* (*map fst xs*) $\Longrightarrow$ *distinct* (*map fst* (*merge xs ys*))
⟨*proof*⟩

lemma *clearjunk-merge*: *clearjunk* (*merge xs ys*) = *merge* (*clearjunk xs*) *ys*
⟨*proof*⟩

lemma *merge-conv'*: *map-of* (*merge xs ys*) = *map-of xs* ++ *map-of ys*
⟨*proof*⟩

corollary *merge-conv*: *map-of* (*merge xs ys*) *k* = (*map-of xs* ++ *map-of ys*) *k*
⟨*proof*⟩

lemma *merge-empty*: *map-of* (*merge* [] *ys*) = *map-of ys*
⟨*proof*⟩

lemma *merge-assoc* [*simp*]: *map-of* (*merge m1* (*merge m2 m3*)) = *map-of* (*merge* (*merge m1 m2*) *m3*)
⟨*proof*⟩

lemma *merge-Some-iff*:
map-of (*merge m n*) *k* = *Some x* ⟷
map-of n k = *Some x* ∨ *map-of n k* = *None* ∧ *map-of m k* = *Some x*
⟨*proof*⟩

lemmas *merge-SomeD* [*dest*!] = *merge-Some-iff* [*THEN iffD1*]

lemma *merge-find-right* [*simp*]: *map-of n k* = *Some v* ⟹ *map-of* (*merge m n*) *k* = *Some v*
⟨*proof*⟩

lemma *merge-None* [*iff*]: (*map-of* (*merge m n*) *k* = *None*) = (*map-of n k* = *None* ∧ *map-of m k* = *None*)
⟨*proof*⟩

lemma *merge-upd* [*simp*]: *map-of* (*merge m* (*update k v n*)) = *map-of* (*update k v* (*merge m n*))
⟨*proof*⟩

lemma *merge-updatess* [*simp*]:
map-of (*merge m* (*updates xs ys n*)) = *map-of* (*updates xs ys* (*merge m n*))
⟨*proof*⟩

lemma *merge-append*: *map-of* (*xs* @ *ys*) = *map-of* (*merge ys xs*)
⟨*proof*⟩

1.8 *compose*

qualified function *compose* :: ('*key* × '*a*) *list* ⇒ ('*a* × '*b*) *list* ⇒ ('*key* × '*b*) *list*
where
compose [] *ys* = []
| *compose* (*x* # *xs*) *ys* =
(*case map-of ys* (*snd x*) *of*
None ⇒ *compose* (*delete* (*fst x*) *xs*) *ys*
| *Some v* ⇒ (*fst x*, *v*) # *compose xs ys*)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *compose-first-None* [*simp*]: *map-of xs k* = *None* ⟹ *map-of* (*compose xs ys*) *k* = *None*
⟨*proof*⟩

lemma *compose-conv*: *map-of* (*compose xs ys*) *k* = (*map-of ys* ∘$_m$ *map-of xs*) *k*
⟨*proof*⟩

lemma *compose-conv'*: *map-of* (*compose xs ys*) = (*map-of ys* ∘$_m$ *map-of xs*)
⟨*proof*⟩

lemma *compose-first-Some* [*simp*]: *map-of xs k = Some v ⟹ map-of (compose xs ys) k = map-of ys v*
⟨*proof*⟩

lemma *dom-compose*: *fst ‘ set (compose xs ys) ⊆ fst ‘ set xs*
⟨*proof*⟩

lemma *distinct-compose*:
assumes *distinct (map fst xs)*
shows *distinct (map fst (compose xs ys))*
⟨*proof*⟩

lemma *compose-delete-twist*: *compose (delete k xs) ys = delete k (compose xs ys)*
⟨*proof*⟩

lemma *compose-clearjunk*: *compose xs (clearjunk ys) = compose xs ys*
⟨*proof*⟩

lemma *clearjunk-compose*: *clearjunk (compose xs ys) = compose (clearjunk xs) ys*
⟨*proof*⟩

lemma *compose-empty* [*simp*]: *compose xs [] = []*
⟨*proof*⟩

lemma *compose-Some-iff*:
(map-of (compose xs ys) k = Some v) ⟷
(∃ k′. map-of xs k = Some k′ ∧ map-of ys k′ = Some v)
⟨*proof*⟩

lemma *map-comp-None-iff*:
map-of (compose xs ys) k = None ⟷
(map-of xs k = None ∨ (∃ k′. map-of xs k = Some k′ ∧ map-of ys k′ = None))
⟨*proof*⟩

1.9 *map-entry*

qualified fun *map-entry* :: *′key ⇒ (′val ⇒ ′val) ⇒ (′key × ′val) list ⇒ (′key × ′val) list*
where
map-entry k f [] = []
| *map-entry k f (p # ps) =*
(if fst p = k then (k, f (snd p)) # ps else p # map-entry k f ps)

lemma *map-of-map-entry*:
map-of (map-entry k f xs) =
(map-of xs)(k := case map-of xs k of None ⇒ None | Some v′ ⇒ Some (f v′))
⟨*proof*⟩

lemma *dom-map-entry*: *fst ‘ set (map-entry k f xs) = fst ‘ set xs*

⟨*proof*⟩

lemma *distinct-map-entry*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*map-entry k f xs*))
⟨*proof*⟩

1.10 *map-default*

fun *map-default* :: *'key* ⇒ *'val* ⇒ (*'val* ⇒ *'val*) ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where
map-default k v f [] = [(*k*, *v*)]
| *map-default k v f* (*p* # *ps*) =
(*if fst p* = *k then* (*k*, *f* (*snd p*)) # *ps else p* # *map-default k v f ps*)

lemma *map-of-map-default*:
map-of (*map-default k v f xs*) =
(*map-of xs*)(*k* := *case map-of xs k of None* ⇒ *Some v* | *Some v'* ⇒ *Some* (*f v'*))
⟨*proof*⟩

lemma *dom-map-default*: *fst* ‘ *set* (*map-default k v f xs*) = *insert k* (*fst* ‘ *set xs*)
⟨*proof*⟩

lemma *distinct-map-default*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*map-default k v f xs*))
⟨*proof*⟩

end

end

2 Pointwise instantiation of functions to algebra type classes

theory *Function-Algebras*
imports *Main*
begin

Pointwise operations

instantiation *fun* :: (*type*, *plus*) *plus*
begin

definition *f* + *g* = (λ*x*. *f x* + *g x*)
instance ⟨*proof*⟩

end

lemma *plus-fun-apply* [*simp*]:
$(f + g)\ x = f\ x + g\ x$
⟨*proof*⟩

instantiation *fun* :: (*type*, *zero*) *zero*
begin

definition $0 = (\lambda x.\ 0)$
instance ⟨*proof*⟩

end

lemma *zero-fun-apply* [*simp*]:
$0\ x = 0$
⟨*proof*⟩

instantiation *fun* :: (*type*, *times*) *times*
begin

definition $f * g = (\lambda x.\ f\ x * g\ x)$
instance ⟨*proof*⟩

end

lemma *times-fun-apply* [*simp*]:
$(f * g)\ x = f\ x * g\ x$
⟨*proof*⟩

instantiation *fun* :: (*type*, *one*) *one*
begin

definition $1 = (\lambda x.\ 1)$
instance ⟨*proof*⟩

end

lemma *one-fun-apply* [*simp*]:
$1\ x = 1$
⟨*proof*⟩

Additive structures

instance *fun* :: (*type*, *semigroup-add*) *semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *cancel-semigroup-add*) *cancel-semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *ab-semigroup-add*) *ab-semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *monoid-add*) *monoid-add*
⟨*proof*⟩

instance *fun* :: (*type*, *comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instance *fun* :: (*type*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add* ⟨*proof*⟩

instance *fun* :: (*type*, *group-add*) *group-add*
⟨*proof*⟩

instance *fun* :: (*type*, *ab-group-add*) *ab-group-add*
⟨*proof*⟩

Multiplicative structures

instance *fun* :: (*type*, *semigroup-mult*) *semigroup-mult*
⟨*proof*⟩

instance *fun* :: (*type*, *ab-semigroup-mult*) *ab-semigroup-mult*
⟨*proof*⟩

instance *fun* :: (*type*, *monoid-mult*) *monoid-mult*
⟨*proof*⟩

instance *fun* :: (*type*, *comm-monoid-mult*) *comm-monoid-mult*
⟨*proof*⟩

Misc

instance *fun* :: (*type*, *Rings.dvd*) *Rings.dvd* ⟨*proof*⟩

instance *fun* :: (*type*, *mult-zero*) *mult-zero*
⟨*proof*⟩

instance *fun* :: (*type*, *zero-neq-one*) *zero-neq-one*
⟨*proof*⟩

Ring structures

instance *fun* :: (*type*, *semiring*) *semiring*
⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring*) *comm-semiring*
⟨*proof*⟩

instance *fun* :: (*type*, *semiring-0*) *semiring-0* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring-0*) *comm-semiring-0* ⟨*proof*⟩

instance *fun* :: (*type*, *semiring-0-cancel*) *semiring-0-cancel* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring-0-cancel*) *comm-semiring-0-cancel* ⟨*proof*⟩

instance *fun* :: (*type*, *semiring-1*) *semiring-1* ⟨*proof*⟩

lemma *of-nat-fun*: *of-nat n* = (λ*x*::′*a*. *of-nat n*)
⟨*proof*⟩

lemma *of-nat-fun-apply* [*simp*]:
of-nat n x = *of-nat n*
⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring-1*) *comm-semiring-1* ⟨*proof*⟩

instance *fun* :: (*type*, *semiring-1-cancel*) *semiring-1-cancel* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-semiring-1-cancel*) *comm-semiring-1-cancel*
⟨*proof*⟩

instance *fun* :: (*type*, *semiring-char-0*) *semiring-char-0*
⟨*proof*⟩

instance *fun* :: (*type*, *ring*) *ring* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-ring*) *comm-ring* ⟨*proof*⟩

instance *fun* :: (*type*, *ring-1*) *ring-1* ⟨*proof*⟩

instance *fun* :: (*type*, *comm-ring-1*) *comm-ring-1* ⟨*proof*⟩

instance *fun* :: (*type*, *ring-char-0*) *ring-char-0* ⟨*proof*⟩

Ordered structures

instance *fun* :: (*type*, *ordered-ab-semigroup-add*) *ordered-ab-semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *ordered-cancel-ab-semigroup-add*) *ordered-cancel-ab-semigroup-add*
⟨*proof*⟩

instance *fun* :: (*type*, *ordered-ab-semigroup-add-imp-le*) *ordered-ab-semigroup-add-imp-le*
⟨*proof*⟩

instance *fun* :: (*type*, *ordered-comm-monoid-add*) *ordered-comm-monoid-add* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-cancel-comm-monoid-add*) *ordered-cancel-comm-monoid-add* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-ab-group-add*) *ordered-ab-group-add* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-semiring*) *ordered-semiring* ⟨*proof*⟩

instance *fun* :: (*type*, *dioid*) *dioid* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-comm-semiring*) *ordered-comm-semiring* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-cancel-semiring*) *ordered-cancel-semiring* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-cancel-comm-semiring*) *ordered-cancel-comm-semiring* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-ring*) *ordered-ring* ⟨*proof*⟩

instance *fun* :: (*type*, *ordered-comm-ring*) *ordered-comm-ring* ⟨*proof*⟩

lemmas *func-plus* = *plus-fun-def*
lemmas *func-zero* = *zero-fun-def*
lemmas *func-times* = *times-fun-def*
lemmas *func-one* = *one-fun-def*

end

3 Algebraic operations on sets

theory *Set-Algebras*
imports *Main*
begin

This library lifts operations like addition and multiplication to sets. It was designed to support asymptotic calculations. See the comments at the top of `BigO.thy`.

instantiation *set* :: (*plus*) *plus*
begin

definition *plus-set* :: *'a*::*plus set* ⇒ *'a set* ⇒ *'a set*
where *set-plus-def*: $A + B = \{c.\ \exists\, a{\in}A.\ \exists\, b{\in}B.\ c = a + b\}$

instance ⟨*proof*⟩

end

instantiation *set* :: (*times*) *times*
begin

definition *times-set* :: *'a::times set* ⇒ *'a set* ⇒ *'a set*
  where *set-times-def*: $A * B = \{c.\ \exists\, a{\in}A.\ \exists\, b{\in}B.\ c = a * b\}$

instance ⟨*proof*⟩

end

instantiation *set* :: (*zero*) *zero*
begin

definition *set-zero*[*simp*]: $(0{::}'a{::}zero\ set) = \{0\}$

instance ⟨*proof*⟩

end

instantiation *set* :: (*one*) *one*
begin

definition *set-one*[*simp*]: $(1{::}'a{::}one\ set) = \{1\}$

instance ⟨*proof*⟩

end

definition *elt-set-plus* :: *'a::plus* ⇒ *'a set* ⇒ *'a set* (**infixl** $+o$ *70*)
  where $a +o B = \{c.\ \exists\, b{\in}B.\ c = a + b\}$

definition *elt-set-times* :: *'a::times* ⇒ *'a set* ⇒ *'a set* (**infixl** $*o$ *80*)
  where $a *o B = \{c.\ \exists\, b{\in}B.\ c = a * b\}$

abbreviation (*input*) *elt-set-eq* :: *'a* ⇒ *'a set* ⇒ *bool* (**infix** $=o$ *50*)
  where $x =o A \equiv x \in A$

instance *set* :: (*semigroup-add*) *semigroup-add*
  ⟨*proof*⟩

instance *set* :: (*ab-semigroup-add*) *ab-semigroup-add*
  ⟨*proof*⟩

instance *set* :: (*monoid-add*) *monoid-add*
  ⟨*proof*⟩

instance *set* :: (*comm-monoid-add*) *comm-monoid-add*
  ⟨*proof*⟩

instance *set* :: (*semigroup-mult*) *semigroup-mult*
⟨*proof*⟩

instance *set* :: (*ab-semigroup-mult*) *ab-semigroup-mult*
⟨*proof*⟩

instance *set* :: (*monoid-mult*) *monoid-mult*
⟨*proof*⟩

instance *set* :: (*comm-monoid-mult*) *comm-monoid-mult*
⟨*proof*⟩

lemma *set-plus-intro* [*intro*]: $a \in C \Longrightarrow b \in D \Longrightarrow a + b \in C + D$
⟨*proof*⟩

lemma *set-plus-elim*:
assumes $x \in A + B$
obtains a b **where** $x = a + b$ **and** $a \in A$ **and** $b \in B$
⟨*proof*⟩

lemma *set-plus-intro2* [*intro*]: $b \in C \Longrightarrow a + b \in a +o C$
⟨*proof*⟩

lemma *set-plus-rearrange*: $(a +o C) + (b +o D) = (a + b) +o (C + D)$
for a b :: *'a*::*comm-monoid-add*
⟨*proof*⟩

lemma *set-plus-rearrange2*: $a +o (b +o C) = (a + b) +o C$
for a b :: *'a*::*semigroup-add*
⟨*proof*⟩

lemma *set-plus-rearrange3*: $(a +o B) + C = a +o (B + C)$
for a :: *'a*::*semigroup-add*
⟨*proof*⟩

theorem *set-plus-rearrange4*: $C + (a +o D) = a +o (C + D)$
for a :: *'a*::*comm-monoid-add*
⟨*proof*⟩

lemmas *set-plus-rearranges* = *set-plus-rearrange set-plus-rearrange2*
set-plus-rearrange3 set-plus-rearrange4

lemma *set-plus-mono* [*intro*!]: $C \subseteq D \Longrightarrow a +o C \subseteq a +o D$
⟨*proof*⟩

lemma *set-plus-mono2* [*intro*]: $C \subseteq D \Longrightarrow E \subseteq F \Longrightarrow C + E \subseteq D + F$
for C D E F :: *'a*::*plus set*
⟨*proof*⟩

lemma *set-plus-mono3* [*intro*]: $a \in C \Longrightarrow a +o D \subseteq C + D$
⟨*proof*⟩

lemma *set-plus-mono4* [*intro*]: $a \in C \Longrightarrow a +o D \subseteq D + C$
for $a :: 'a$::*comm-monoid-add*
⟨*proof*⟩

lemma *set-plus-mono5*: $a \in C \Longrightarrow B \subseteq D \Longrightarrow a +o B \subseteq C + D$
⟨*proof*⟩

lemma *set-plus-mono-b*: $C \subseteq D \Longrightarrow x \in a +o C \Longrightarrow x \in a +o D$
⟨*proof*⟩

lemma *set-plus-mono2-b*: $C \subseteq D \Longrightarrow E \subseteq F \Longrightarrow x \in C + E \Longrightarrow x \in D + F$
⟨*proof*⟩

lemma *set-plus-mono3-b*: $a \in C \Longrightarrow x \in a +o D \Longrightarrow x \in C + D$
⟨*proof*⟩

lemma *set-plus-mono4-b*: $a \in C \Longrightarrow x \in a +o D \Longrightarrow x \in D + C$
for $a\ x :: 'a$::*comm-monoid-add*
⟨*proof*⟩

lemma *set-zero-plus* [*simp*]: $0 +o C = C$
for $C :: 'a$::*comm-monoid-add set*
⟨*proof*⟩

lemma *set-zero-plus2*: $0 \in A \Longrightarrow B \subseteq A + B$
for $A\ B :: 'a$::*comm-monoid-add set*
⟨*proof*⟩

lemma *set-plus-imp-minus*: $a \in b +o C \Longrightarrow a - b \in C$
for $a\ b :: 'a$::*ab-group-add*
⟨*proof*⟩

lemma *set-minus-imp-plus*: $a - b \in C \Longrightarrow a \in b +o C$
for $a\ b :: 'a$::*ab-group-add*
⟨*proof*⟩

lemma *set-minus-plus*: $a - b \in C \longleftrightarrow a \in b +o C$
for $a\ b :: 'a$::*ab-group-add*
⟨*proof*⟩

lemma *set-times-intro* [*intro*]: $a \in C \Longrightarrow b \in D \Longrightarrow a * b \in C * D$
⟨*proof*⟩

lemma *set-times-elim*:
assumes $x \in A * B$

obtains a b **where** $x = a * b$ **and** $a \in A$ **and** $b \in B$
⟨*proof*⟩

lemma *set-times-intro2* [*intro*!]: $b \in C \Longrightarrow a * b \in a *o C$
⟨*proof*⟩

lemma *set-times-rearrange*: $(a *o C) * (b *o D) = (a * b) *o (C * D)$
for a b :: $'a$::*comm-monoid-mult*
⟨*proof*⟩

lemma *set-times-rearrange2*: $a *o (b *o C) = (a * b) *o C$
for a b :: $'a$::*semigroup-mult*
⟨*proof*⟩

lemma *set-times-rearrange3*: $(a *o B) * C = a *o (B * C)$
for a :: $'a$::*semigroup-mult*
⟨*proof*⟩

theorem *set-times-rearrange4*: $C * (a *o D) = a *o (C * D)$
for a :: $'a$::*comm-monoid-mult*
⟨*proof*⟩

lemmas *set-times-rearranges = set-times-rearrange set-times-rearrange2*
set-times-rearrange3 set-times-rearrange4

lemma *set-times-mono* [*intro*]: $C \subseteq D \Longrightarrow a *o C \subseteq a *o D$
⟨*proof*⟩

lemma *set-times-mono2* [*intro*]: $C \subseteq D \Longrightarrow E \subseteq F \Longrightarrow C * E \subseteq D * F$
for C D E F :: $'a$::*times set*
⟨*proof*⟩

lemma *set-times-mono3* [*intro*]: $a \in C \Longrightarrow a *o D \subseteq C * D$
⟨*proof*⟩

lemma *set-times-mono4* [*intro*]: $a \in C \Longrightarrow a *o D \subseteq D * C$
for a :: $'a$::*comm-monoid-mult*
⟨*proof*⟩

lemma *set-times-mono5*: $a \in C \Longrightarrow B \subseteq D \Longrightarrow a *o B \subseteq C * D$
⟨*proof*⟩

lemma *set-times-mono-b*: $C \subseteq D \Longrightarrow x \in a *o C \Longrightarrow x \in a *o D$
⟨*proof*⟩

lemma *set-times-mono2-b*: $C \subseteq D \Longrightarrow E \subseteq F \Longrightarrow x \in C * E \Longrightarrow x \in D * F$
⟨*proof*⟩

lemma *set-times-mono3-b*: $a \in C \Longrightarrow x \in a *o D \Longrightarrow x \in C * D$

⟨*proof*⟩

lemma *set-times-mono4-b*: $a \in C \Longrightarrow x \in a *o D \Longrightarrow x \in D * C$
for $a\ x$ :: *'a*::*comm-monoid-mult*
⟨*proof*⟩

lemma *set-one-times* [*simp*]: $1 *o C = C$
for C :: *'a*::*comm-monoid-mult set*
⟨*proof*⟩

lemma *set-times-plus-distrib*: $a *o (b +o C) = (a * b) +o (a *o C)$
for $a\ b$ :: *'a*::*semiring*
⟨*proof*⟩

lemma *set-times-plus-distrib2*: $a *o (B + C) = (a *o B) + (a *o C)$
for a :: *'a*::*semiring*
⟨*proof*⟩

lemma *set-times-plus-distrib3*: $(a +o C) * D \subseteq a *o D + C * D$
for a :: *'a*::*semiring*
⟨*proof*⟩

lemmas *set-times-plus-distribs* =
set-times-plus-distrib
set-times-plus-distrib2

lemma *set-neg-intro*: $a \in (-\ 1) *o C \Longrightarrow -\ a \in C$
for a :: *'a*::*ring-1*
⟨*proof*⟩

lemma *set-neg-intro2*: $a \in C \Longrightarrow -\ a \in (-\ 1) *o C$
for a :: *'a*::*ring-1*
⟨*proof*⟩

lemma *set-plus-image*: $S + T = (\lambda(x,\ y).\ x + y)\ `\ (S \times T)$
⟨*proof*⟩

lemma *set-times-image*: $S * T = (\lambda(x,\ y).\ x * y)\ `\ (S \times T)$
⟨*proof*⟩

lemma *finite-set-plus*: *finite* $s \Longrightarrow$ *finite* $t \Longrightarrow$ *finite* $(s + t)$
⟨*proof*⟩

lemma *finite-set-times*: *finite* $s \Longrightarrow$ *finite* $t \Longrightarrow$ *finite* $(s * t)$
⟨*proof*⟩

lemma *set-sum-alt*:
assumes *fin*: *finite* I
shows *sum* $S\ I = \{sum\ s\ I\ |s.\ \forall i{\in}I.\ s\ i \in S\ i\}$

(**is** - = *?sum I*)
⟨*proof*⟩

lemma *sum-set-cond-linear*:
fixes *f* :: *'a*::*comm-monoid-add set* ⇒ *'b*::*comm-monoid-add set*
assumes [*intro*!]: ⋀*A B*. *P A* ⟹ *P B* ⟹ *P* (*A* + *B*) *P* {*0*}
and *f*: ⋀*A B*. *P A* ⟹ *P B* ⟹ *f* (*A* + *B*) = *f A* + *f B f* {*0*} = {*0*}
assumes *all*: ⋀*i*. *i* ∈ *I* ⟹ *P* (*S i*)
shows *f* (*sum S I*) = *sum* (*f* ∘ *S*) *I*
⟨*proof*⟩

lemma *sum-set-linear*:
fixes *f* :: *'a*::*comm-monoid-add set* ⇒ *'b*::*comm-monoid-add set*
assumes ⋀*A B*. *f*(*A*) + *f*(*B*) = *f*(*A* + *B*) *f* {*0*} = {*0*}
shows *f* (*sum S I*) = *sum* (*f* ∘ *S*) *I*
⟨*proof*⟩

lemma *set-times-Un-distrib*:
A ∗ (*B* ∪ *C*) = *A* ∗ *B* ∪ *A* ∗ *C*
(*A* ∪ *B*) ∗ *C* = *A* ∗ *C* ∪ *B* ∗ *C*
⟨*proof*⟩

lemma *set-times-UNION-distrib*:
A ∗ *UNION I M* = (⋃*i*∈*I*. *A* ∗ *M i*)
UNION I M ∗ *A* = (⋃*i*∈*I*. *M i* ∗ *A*)
⟨*proof*⟩

end

4 Big O notation

theory *BigO*
imports
Complex-Main
Function-Algebras
Set-Algebras
begin

This library is designed to support asymptotic "big O" calculations, i.e. reasoning with expressions of the form $f = O(g)$ and $f = g + O(h)$. An earlier version of this library is described in detail in [1].

The main changes in this version are as follows:

- We have eliminated the O operator on sets. (Most uses of this seem to be inessential.)
- We no longer use + as output syntax for $+o$
- Lemmas involving *sumr* have been replaced by more general lemmas involving '*sum*.

- The library has been expanded, with e.g. support for expressions of the form $f < g + O(h)$.

Note also since the Big O library includes rules that demonstrate set inclusion, to use the automated reasoners effectively with the library one should redeclare the theorem *subsetI* as an intro rule, rather than as an *intro*! rule, for example, using **declare** *subsetI* [*del*, *intro*].

4.1 Definitions

definition *bigo* :: $('a \Rightarrow 'b{::}\textit{linordered-idom}) \Rightarrow ('a \Rightarrow 'b)\ \textit{set}$ $((1O'(\text{-}')))$
where $O(f{::}\ 'a \Rightarrow 'b) = \{h.\ \exists\, c.\ \forall\, x.\ |h\ x| \leq c * |f\ x|\}$

lemma *bigo-pos-const*:
$(\exists\, c{::}'a{::}\textit{linordered-idom}.\ \forall\, x.\ |h\ x| \leq c * |f\ x|) \longleftrightarrow$
$(\exists\, c.\ 0 < c \wedge (\forall\, x.\ |h\ x| \leq c * |f\ x|))$
⟨*proof*⟩

lemma *bigo-alt-def*: $O(f) = \{h.\ \exists\, c.\ 0 < c \wedge (\forall\, x.\ |h\ x| \leq c * |f\ x|)\}$
⟨*proof*⟩

lemma *bigo-elt-subset* [*intro*]: $f \in O(g) \Longrightarrow O(f) \leq O(g)$
⟨*proof*⟩

lemma *bigo-refl* [*intro*]: $f \in O(f)$
⟨*proof*⟩

lemma *bigo-zero*: $0 \in O(g)$
⟨*proof*⟩

lemma *bigo-zero2*: $O(\lambda x.\ 0) = \{\lambda x.\ 0\}$
⟨*proof*⟩

lemma *bigo-plus-self-subset* [*intro*]: $O(f) + O(f) \subseteq O(f)$
⟨*proof*⟩

lemma *bigo-plus-idemp* [*simp*]: $O(f) + O(f) = O(f)$
⟨*proof*⟩

lemma *bigo-plus-subset* [*intro*]: $O(f + g) \subseteq O(f) + O(g)$
⟨*proof*⟩

lemma *bigo-plus-subset2* [*intro*]: $A \subseteq O(f) \Longrightarrow B \subseteq O(f) \Longrightarrow A + B \subseteq O(f)$
⟨*proof*⟩

lemma *bigo-plus-eq*: $\forall\, x.\ 0 \leq f\ x \Longrightarrow \forall\, x.\ 0 \leq g\ x \Longrightarrow O(f + g) = O(f) + O(g)$
⟨*proof*⟩

lemma *bigo-bounded-alt*: $\forall\, x.\ 0 \leq f\ x \Longrightarrow \forall\, x.\ f\ x \leq c * g\ x \Longrightarrow f \in O(g)$

⟨*proof*⟩

lemma *bigo-bounded*: $\forall x.\ 0 \leq f\ x \Longrightarrow \forall x.\ f\ x \leq g\ x \Longrightarrow f \in O(g)$
⟨*proof*⟩

lemma *bigo-bounded2*: $\forall x.\ lb\ x \leq f\ x \Longrightarrow \forall x.\ f\ x \leq lb\ x + g\ x \Longrightarrow f \in lb +o O(g)$
⟨*proof*⟩

lemma *bigo-abs*: $(\lambda x.\ |f\ x|) =o\ O(f)$
⟨*proof*⟩

lemma *bigo-abs2*: $f =o\ O(\lambda x.\ |f\ x|)$
⟨*proof*⟩

lemma *bigo-abs3*: $O(f) = O(\lambda x.\ |f\ x|)$
⟨*proof*⟩

lemma *bigo-abs4*: $f =o\ g +o\ O(h) \Longrightarrow (\lambda x.\ |f\ x|) =o\ (\lambda x.\ |g\ x|) +o\ O(h)$
⟨*proof*⟩

lemma *bigo-abs5*: $f =o\ O(g) \Longrightarrow (\lambda x.\ |f\ x|) =o\ O(g)$
⟨*proof*⟩

lemma *bigo-elt-subset2* [*intro*]:
assumes $*$: $f \in g +o\ O(h)$
shows $O(f) \subseteq O(g) + O(h)$
⟨*proof*⟩

lemma *bigo-mult* [*intro*]: $O(f)*O(g) \subseteq O(f * g)$
⟨*proof*⟩

lemma *bigo-mult2* [*intro*]: $f *o\ O(g) \subseteq O(f * g)$
⟨*proof*⟩

lemma *bigo-mult3*: $f \in O(h) \Longrightarrow g \in O(j) \Longrightarrow f * g \in O(h * j)$
⟨*proof*⟩

lemma *bigo-mult4* [*intro*]: $f \in k +o\ O(h) \Longrightarrow g * f \in (g * k) +o\ O(g * h)$
⟨*proof*⟩

lemma *bigo-mult5*:
fixes $f :: 'a \Rightarrow 'b$::*linordered-field*
assumes $\forall x.\ f\ x \neq 0$
shows $O(f * g) \subseteq f *o\ O(g)$
⟨*proof*⟩

lemma *bigo-mult6*: $\forall x.\ f\ x \neq 0 \Longrightarrow O(f * g) = f *o\ O(g)$
for $f :: 'a \Rightarrow 'b$::*linordered-field*

⟨*proof*⟩

lemma *bigo-mult7*: $\forall x.\ f\ x \neq 0 \Longrightarrow O(f * g) \subseteq O(f) * O(g)$
for $f :: 'a \Rightarrow 'b$::*linordered-field*
⟨*proof*⟩

lemma *bigo-mult8*: $\forall x.\ f\ x \neq 0 \Longrightarrow O(f * g) = O(f) * O(g)$
for $f :: 'a \Rightarrow 'b$::*linordered-field*
⟨*proof*⟩

lemma *bigo-minus* [*intro*]: $f \in O(g) \Longrightarrow - f \in O(g)$
⟨*proof*⟩

lemma *bigo-minus2*: $f \in g +o O(h) \Longrightarrow - f \in -g +o O(h)$
⟨*proof*⟩

lemma *bigo-minus3*: $O(- f) = O(f)$
⟨*proof*⟩

lemma *bigo-plus-absorb-lemma1*:
assumes $*$: $f \in O(g)$
shows $f +o O(g) \subseteq O(g)$
⟨*proof*⟩

lemma *bigo-plus-absorb-lemma2*:
assumes $*$: $f \in O(g)$
shows $O(g) \subseteq f +o O(g)$
⟨*proof*⟩

lemma *bigo-plus-absorb* [*simp*]: $f \in O(g) \Longrightarrow f +o O(g) = O(g)$
⟨*proof*⟩

lemma *bigo-plus-absorb2* [*intro*]: $f \in O(g) \Longrightarrow A \subseteq O(g) \Longrightarrow f +o A \subseteq O(g)$
⟨*proof*⟩

lemma *bigo-add-commute-imp*: $f \in g +o O(h) \Longrightarrow g \in f +o O(h)$
⟨*proof*⟩

lemma *bigo-add-commute*: $f \in g +o O(h) \longleftrightarrow g \in f +o O(h)$
⟨*proof*⟩

lemma *bigo-const1*: $(\lambda x.\ c) \in O(\lambda x.\ 1)$
⟨*proof*⟩

lemma *bigo-const2* [*intro*]: $O(\lambda x.\ c) \subseteq O(\lambda x.\ 1)$
⟨*proof*⟩

lemma *bigo-const3*: $c \neq 0 \Longrightarrow (\lambda x.\ 1) \in O(\lambda x.\ c)$
for $c :: 'a$::*linordered-field*

⟨*proof*⟩

lemma *bigo-const4*: $c \neq 0 \Longrightarrow O(\lambda x.\ 1) \subseteq O(\lambda x.\ c)$
for $c :: 'a$::*linordered-field*
⟨*proof*⟩

lemma *bigo-const* [*simp*]: $c \neq 0 \Longrightarrow O(\lambda x.\ c) = O(\lambda x.\ 1)$
for $c :: 'a$::*linordered-field*
⟨*proof*⟩

lemma *bigo-const-mult1*: $(\lambda x.\ c * f\ x) \in O(f)$
⟨*proof*⟩

lemma *bigo-const-mult2*: $O(\lambda x.\ c * f\ x) \subseteq O(f)$
⟨*proof*⟩

lemma *bigo-const-mult3*: $c \neq 0 \Longrightarrow f \in O(\lambda x.\ c * f\ x)$
for $c :: 'a$::*linordered-field*
⟨*proof*⟩

lemma *bigo-const-mult4*: $c \neq 0 \Longrightarrow O(f) \subseteq O(\lambda x.\ c * f\ x)$
for $c :: 'a$::*linordered-field*
⟨*proof*⟩

lemma *bigo-const-mult* [*simp*]: $c \neq 0 \Longrightarrow O(\lambda x.\ c * f\ x) = O(f)$
for $c :: 'a$::*linordered-field*
⟨*proof*⟩

lemma *bigo-const-mult5* [*simp*]: $c \neq 0 \Longrightarrow (\lambda x.\ c) *o\ O(f) = O(f)$
for $c :: 'a$::*linordered-field*
⟨*proof*⟩

lemma *bigo-const-mult6* [*intro*]: $(\lambda x.\ c) *o\ O(f) \subseteq O(f)$
⟨*proof*⟩

lemma *bigo-const-mult7* [*intro*]:
assumes $*$: $f =o\ O(g)$
shows $(\lambda x.\ c * f\ x) =o\ O(g)$
⟨*proof*⟩

lemma *bigo-compose1*: $f =o\ O(g) \Longrightarrow (\lambda x.\ f\ (k\ x)) =o\ O(\lambda x.\ g\ (k\ x))$
⟨*proof*⟩

lemma *bigo-compose2*: $f =o\ g +o\ O(h) \Longrightarrow (\lambda x.\ f\ (k\ x)) =o\ (\lambda x.\ g\ (k\ x)) +o$
$O(\lambda x.\ h(k\ x))$
⟨*proof*⟩

4.2 Sum

lemma *bigo-sum-main*: $\forall x.\ \forall y \in A\ x.\ 0 \leq h\ x\ y \Longrightarrow$
$\exists c.\ \forall x.\ \forall y \in A\ x.\ |f\ x\ y| \leq c * h\ x\ y \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ f\ x\ y) =o\ O(\lambda x.\ \sum y \in A\ x.\ h\ x\ y)$
⟨*proof*⟩

lemma *bigo-sum1*: $\forall x\ y.\ 0 \leq h\ x\ y \Longrightarrow$
$\exists c.\ \forall x\ y.\ |f\ x\ y| \leq c * h\ x\ y \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ f\ x\ y) =o\ O(\lambda x.\ \sum y \in A\ x.\ h\ x\ y)$
⟨*proof*⟩

lemma *bigo-sum2*: $\forall y.\ 0 \leq h\ y \Longrightarrow$
$\exists c.\ \forall y.\ |f\ y| \leq c * (h\ y) \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ f\ y) =o\ O(\lambda x.\ \sum y \in A\ x.\ h\ y)$
⟨*proof*⟩

lemma *bigo-sum3*: $f =o\ O(h) \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * f\ (k\ x\ y)) =o\ O(\lambda x.\ \sum y \in A\ x.\ |l\ x\ y * h\ (k\ x\ y)|)$
⟨*proof*⟩

lemma *bigo-sum4*: $f =o\ g +o\ O(h) \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * f\ (k\ x\ y)) =o$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * g\ (k\ x\ y)) +o$
$O(\lambda x.\ \sum y \in A\ x.\ |l\ x\ y * h\ (k\ x\ y)|)$
⟨*proof*⟩

lemma *bigo-sum5*: $f =o\ O(h) \Longrightarrow \forall x\ y.\ 0 \leq l\ x\ y \Longrightarrow$
$\forall x.\ 0 \leq h\ x \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * f\ (k\ x\ y)) =o$
$O(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * h\ (k\ x\ y))$
⟨*proof*⟩

lemma *bigo-sum6*: $f =o\ g +o\ O(h) \Longrightarrow \forall x\ y.\ 0 \leq l\ x\ y \Longrightarrow$
$\forall x.\ 0 \leq h\ x \Longrightarrow$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * f\ (k\ x\ y)) =o$
$(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * g\ (k\ x\ y)) +o$
$O(\lambda x.\ \sum y \in A\ x.\ l\ x\ y * h\ (k\ x\ y))$
⟨*proof*⟩

4.3 Misc useful stuff

lemma *bigo-useful-intro*: $A \subseteq O(f) \Longrightarrow B \subseteq O(f) \Longrightarrow A + B \subseteq O(f)$
⟨*proof*⟩

lemma *bigo-useful-add*: $f =o\ O(h) \Longrightarrow g =o\ O(h) \Longrightarrow f + g =o\ O(h)$
⟨*proof*⟩

lemma *bigo-useful-const-mult*: $c \neq 0 \Longrightarrow (\lambda x.\ c) * f =o\ O(h) \Longrightarrow f =o\ O(h)$
for c :: *'a*::*linordered-field*

⟨*proof*⟩

lemma *bigo-fix*: $(\lambda x{::}nat.\ f\ (x + 1)) =o\ O(\lambda x.\ h\ (x + 1)) \Longrightarrow f\ 0 = 0 \Longrightarrow f =o\ O(h)$
⟨*proof*⟩

lemma *bigo-fix2*:
$(\lambda x.\ f\ ((x{::}nat) + 1)) =o\ (\lambda x.\ g(x + 1)) +o\ O(\lambda x.\ h(x + 1)) \Longrightarrow$
$f\ 0 = g\ 0 \Longrightarrow f =o\ g +o\ O(h)$
⟨*proof*⟩

4.4 Less than or equal to

definition *lesso* :: $('a \Rightarrow 'b{::}linordered\text{-}idom) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixl** $<o$ *70*)
where $f <o g = (\lambda x.\ max\ (f\ x - g\ x)\ 0)$

lemma *bigo-lesseq1*: $f =o\ O(h) \Longrightarrow \forall x.\ |g\ x| \leq |f\ x| \Longrightarrow g =o\ O(h)$
⟨*proof*⟩

lemma *bigo-lesseq2*: $f =o\ O(h) \Longrightarrow \forall x.\ |g\ x| \leq f\ x \Longrightarrow g =o\ O(h)$
⟨*proof*⟩

lemma *bigo-lesseq3*: $f =o\ O(h) \Longrightarrow \forall x.\ 0 \leq g\ x \Longrightarrow \forall x.\ g\ x \leq f\ x \Longrightarrow g =o\ O(h)$
⟨*proof*⟩

lemma *bigo-lesseq4*: $f =o\ O(h) \Longrightarrow$
$\forall x.\ 0 \leq g\ x \Longrightarrow \forall x.\ g\ x \leq |f\ x| \Longrightarrow g =o\ O(h)$
⟨*proof*⟩

lemma *bigo-lesso1*: $\forall x.\ f\ x \leq g\ x \Longrightarrow f <o g =o\ O(h)$
⟨*proof*⟩

lemma *bigo-lesso2*: $f =o\ g +o\ O(h) \Longrightarrow \forall x.\ 0 \leq k\ x \Longrightarrow \forall x.\ k\ x \leq f\ x \Longrightarrow k <o g =o\ O(h)$
⟨*proof*⟩

lemma *bigo-lesso3*: $f =o\ g +o\ O(h) \Longrightarrow \forall x.\ 0 \leq k\ x \Longrightarrow \forall x.\ g\ x \leq k\ x \Longrightarrow f <o k =o\ O(h)$
⟨*proof*⟩

lemma *bigo-lesso4*: $f <o g =o\ O(k) \Longrightarrow g =o\ h +o\ O(k) \Longrightarrow f <o h =o\ O(k)$
for $k :: 'a \Rightarrow 'b{::}linordered\text{-}field$
⟨*proof*⟩

lemma *bigo-lesso5*: $f <o g =o\ O(h) \Longrightarrow \exists C.\ \forall x.\ f\ x \leq g\ x + C * |h\ x|$
⟨*proof*⟩

lemma *lesso-add*: $f <o g =o O(h) \Longrightarrow k <o l =o O(h) \Longrightarrow (f + k) <o (g + l) =o O(h)$
⟨*proof*⟩

lemma *bigo-LIMSEQ1*: $f =o O(g) \Longrightarrow g \longrightarrow 0 \Longrightarrow f \longrightarrow 0$
for *f g* :: *nat* ⇒ *real*
⟨*proof*⟩

lemma *bigo-LIMSEQ2*: $f =o g +o O(h) \Longrightarrow h \longrightarrow 0 \Longrightarrow f \longrightarrow a \Longrightarrow g \longrightarrow a$
for *f g h* :: *nat* ⇒ *real*
⟨*proof*⟩

end

5 The Field of Integers mod 2

theory *Bit*
imports *Main*
begin

5.1 Bits as a datatype

typedef *bit* = *UNIV* :: *bool set*
morphisms *set Bit* ⟨*proof*⟩

instantiation *bit* :: {*zero*, *one*}
begin

definition *zero-bit-def*: *0* = *Bit False*

definition *one-bit-def*: *1* = *Bit True*

instance ⟨*proof*⟩

end

old-rep-datatype *0*::*bit* *1*::*bit*
⟨*proof*⟩

lemma *Bit-set-eq* [*simp*]: *Bit* (*set b*) = *b*
⟨*proof*⟩

lemma *set-Bit-eq* [*simp*]: *set* (*Bit P*) = *P*
⟨*proof*⟩

lemma *bit-eq-iff*: *x* = *y* ⟷ (*set x* ⟷ *set y*)
⟨*proof*⟩

lemma *Bit-inject* [*simp*]: *Bit P* = *Bit Q* ⟷ (*P* ⟷ *Q*)
⟨*proof*⟩

lemma *set* [*iff*]:
¬ *set 0*
set 1
⟨*proof*⟩

lemma [*code*]:
set 0 ⟷ *False*
set 1 ⟷ *True*
⟨*proof*⟩

lemma *set-iff*: *set b* ⟷ *b* = *1*
⟨*proof*⟩

lemma *bit-eq-iff-set*:
b = *0* ⟷ ¬ *set b*
b = *1* ⟷ *set b*
⟨*proof*⟩

lemma *Bit* [*simp*, *code*]:
Bit False = *0*
Bit True = *1*
⟨*proof*⟩

lemma *bit-not-0-iff* [*iff*]: *x* ≠ *0* ⟷ *x* = *1* **for** *x* :: *bit*
⟨*proof*⟩

lemma *bit-not-1-iff* [*iff*]: *x* ≠ *1* ⟷ *x* = *0* **for** *x* :: *bit*
⟨*proof*⟩

lemma [*code*]:
HOL.equal 0 b ⟷ ¬ *set b*
HOL.equal 1 b ⟷ *set b*
⟨*proof*⟩

5.2 Type *bit* forms a field

instantiation *bit* :: *field*
begin

definition *plus-bit-def*: *x* + *y* = *case-bit y* (*case-bit 1 0 y*) *x*

definition *times-bit-def*: *x* ∗ *y* = *case-bit 0 y x*

definition *uminus-bit-def* [*simp*]: − *x* = *x* **for** *x* :: *bit*

definition *minus-bit-def* [*simp*]: *x* − *y* = *x* + *y* **for** *x y* :: *bit*

definition *inverse-bit-def* [*simp*]: *inverse x = x* **for** *x* :: *bit*

definition *divide-bit-def* [*simp*]: *x div y = x * y* **for** *x y* :: *bit*

lemmas *field-bit-defs* =
plus-bit-def times-bit-def minus-bit-def uminus-bit-def
divide-bit-def inverse-bit-def

instance
⟨*proof*⟩

end

lemma *bit-add-self*: *x + x = 0* **for** *x* :: *bit*
⟨*proof*⟩

lemma *bit-mult-eq-1-iff* [*simp*]: *x * y = 1 ⟷ x = 1 ∧ y = 1* **for** *x y* :: *bit*
⟨*proof*⟩

Not sure whether the next two should be simp rules.

lemma *bit-add-eq-0-iff*: *x + y = 0 ⟷ x = y* **for** *x y* :: *bit*
⟨*proof*⟩

lemma *bit-add-eq-1-iff*: *x + y = 1 ⟷ x ≠ y* **for** *x y* :: *bit*
⟨*proof*⟩

5.3 Numerals at type *bit*

All numerals reduce to either 0 or 1.

lemma *bit-minus1* [*simp*]: *− 1 = (1 :: bit)*
⟨*proof*⟩

lemma *bit-neg-numeral* [*simp*]: *(− numeral w :: bit) = numeral w*
⟨*proof*⟩

lemma *bit-numeral-even* [*simp*]: *numeral (Num.Bit0 w) = (0 :: bit)*
⟨*proof*⟩

lemma *bit-numeral-odd* [*simp*]: *numeral (Num.Bit1 w) = (1 :: bit)*
⟨*proof*⟩

5.4 Conversion from *bit*

context *zero-neq-one*
begin

definition *of-bit* :: *bit ⇒ 'a*
where *of-bit b = case-bit 0 1 b*

lemma *of-bit-eq* [*simp*, *code*]:
of-bit 0 = 0
of-bit 1 = 1
⟨*proof*⟩

lemma *of-bit-eq-iff*: *of-bit x = of-bit y ⟷ x = y*
⟨*proof*⟩

end

lemma (**in** *semiring-1*) *of-nat-of-bit-eq*: *of-nat* (*of-bit b*) = *of-bit b*
⟨*proof*⟩

lemma (**in** *ring-1*) *of-int-of-bit-eq*: *of-int* (*of-bit b*) = *of-bit b*
⟨*proof*⟩

hide-const (**open**) *set*

end

6 Axiomatic Declaration of Bounded Natural Functors

theory *BNF-Axiomatization*
imports *Main*
keywords
bnf-axiomatization :: *thy-decl*
begin

⟨*ML*⟩

end

7 Generalized Corecursor Sugar (corec and friends)

theory *BNF-Corec*
imports *Main*
keywords
corec :: *thy-decl* **and**
corecursive :: *thy-goal* **and**
friend-of-corec :: *thy-goal* **and**
coinduction-upto :: *thy-decl*
begin

lemma *obj-distinct-prems*: *P ⟶ P ⟶ Q ⟹ P ⟹ Q*
⟨*proof*⟩

lemma *inject-refine*: $g\ (f\ x) = x \Longrightarrow g\ (f\ y) = y \Longrightarrow f\ x = f\ y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *convol-apply*: *BNF-Def.convol f g x* = $(f\ x,\ g\ x)$
⟨*proof*⟩

lemma *Grp-UNIV-id*: *BNF-Def.Grp UNIV id* = $(op =)$
⟨*proof*⟩

lemma *sum-comp-cases*:
assumes *f o Inl* = *g o Inl* **and** *f o Inr* = *g o Inr*
shows *f* = *g*
⟨*proof*⟩

lemma *case-sum-Inl-Inr-L*: *case-sum* $(f\ o\ Inl)\ (f\ o\ Inr) = f$
⟨*proof*⟩

lemma *eq-o-InrI*: $[\![g\ o\ Inl = h;\ case\text{-}sum\ h\ f = g]\!] \Longrightarrow f = g\ o\ Inr$
⟨*proof*⟩

lemma *id-bnf-o*: *BNF-Composition.id-bnf* $\circ\ f = f$
⟨*proof*⟩

lemma *o-id-bnf*: $f \circ$ *BNF-Composition.id-bnf* $= f$
⟨*proof*⟩

lemma *if-True-False*:
$(if\ P\ then\ True\ else\ Q) \longleftrightarrow P \vee Q$
$(if\ P\ then\ False\ else\ Q) \longleftrightarrow \neg\ P \wedge Q$
$(if\ P\ then\ Q\ else\ True) \longleftrightarrow \neg\ P \vee Q$
$(if\ P\ then\ Q\ else\ False) \longleftrightarrow P \wedge Q$
⟨*proof*⟩

lemma *if-distrib-fun*: $(if\ c\ then\ f\ else\ g)\ x = (if\ c\ then\ f\ x\ else\ g\ x)$
⟨*proof*⟩

7.1 Coinduction

lemma *eq-comp-compI*: $a\ o\ b = f\ o\ x \Longrightarrow x\ o\ c = id \Longrightarrow f = a\ o\ (b\ o\ c)$
⟨*proof*⟩

lemma *self-bounded-weaken-left*: $(a :: 'a :: semilattice\text{-}inf) \leq inf\ a\ b \Longrightarrow a \leq b$
⟨*proof*⟩

lemma *self-bounded-weaken-right*: $(a :: 'a :: semilattice\text{-}inf) \leq inf\ b\ a \Longrightarrow a \leq b$
⟨*proof*⟩

lemma *symp-iff*: *symp R* $\longleftrightarrow R = R\hat{}\!\!-\!\!-1$
⟨*proof*⟩

lemma *equivp-inf*: $[\![equivp\ R;\ equivp\ S]\!] \Longrightarrow equivp\ (inf\ R\ S)$
⟨*proof*⟩

lemma *vimage2p-rel-prod*:
$(\lambda x\ y.\ rel\text{-}prod\ R\ S\ (BNF\text{-}Def.convol\ f1\ g1\ x)\ (BNF\text{-}Def.convol\ f2\ g2\ y)) =$
$(inf\ (BNF\text{-}Def.vimage2p\ f1\ f2\ R)\ (BNF\text{-}Def.vimage2p\ g1\ g2\ S))$
⟨*proof*⟩

lemma *predicate2I-obj*: $(\forall x\ y.\ P\ x\ y \longrightarrow Q\ x\ y) \Longrightarrow P \leq Q$
⟨*proof*⟩

lemma *predicate2D-obj*: $P \leq Q \Longrightarrow P\ x\ y \longrightarrow Q\ x\ y$
⟨*proof*⟩

locale *cong* =
fixes *rel* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool)$
and *eval* :: $'b \Rightarrow 'a$
and *retr* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool)$
assumes *rel-mono*: $\bigwedge R\ S.\ R \leq S \Longrightarrow rel\ R \leq rel\ S$
and *equivp-retr*: $\bigwedge R.\ equivp\ R \Longrightarrow equivp\ (retr\ R)$
and *retr-eval*: $\bigwedge R\ x\ y.\ [\![(rel\text{-}fun\ (rel\ R)\ R)\ eval\ eval;\ rel\ (inf\ R\ (retr\ R))\ x\ y]\!]$
$\Longrightarrow$
$retr\ R\ (eval\ x)\ (eval\ y)$
begin

definition *cong* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
$cong\ R \equiv equivp\ R \wedge (rel\text{-}fun\ (rel\ R)\ R)\ eval\ eval$

lemma *cong-retr*: $cong\ R \Longrightarrow cong\ (inf\ R\ (retr\ R))$
⟨*proof*⟩

lemma *cong-equivp*: $cong\ R \Longrightarrow equivp\ R$
⟨*proof*⟩

definition *gen-cong* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
$gen\text{-}cong\ R\ j1\ j2 \equiv \forall R'.\ R \leq R' \wedge cong\ R' \longrightarrow R'\ j1\ j2$

lemma *gen-cong-reflp*[*intro*, *simp*]: $x = y \Longrightarrow gen\text{-}cong\ R\ x\ y$
⟨*proof*⟩

lemma *gen-cong-symp*[*intro*]: $gen\text{-}cong\ R\ x\ y \Longrightarrow gen\text{-}cong\ R\ y\ x$
⟨*proof*⟩

lemma *gen-cong-transp*[*intro*]: $gen\text{-}cong\ R\ x\ y \Longrightarrow gen\text{-}cong\ R\ y\ z \Longrightarrow gen\text{-}cong\ R\ x\ z$
⟨*proof*⟩

lemma *equivp-gen-cong*: $equivp\ (gen\text{-}cong\ R)$

⟨*proof*⟩

lemma *leq-gen-cong*: $R \leq$ *gen-cong* R
⟨*proof*⟩

lemmas *imp-gen-cong*[*intro*] = *predicate2D*[*OF leq-gen-cong*]

lemma *gen-cong-minimal*: $[\![R \leq R'; \textit{cong}\ R']\!] \Longrightarrow \textit{gen-cong}\ R \leq R'$
⟨*proof*⟩

lemma *congdd-base-gen-congdd-base-aux*:
rel (*gen-cong* R) $x\ y \Longrightarrow R \leq R' \Longrightarrow \textit{cong}\ R' \Longrightarrow R'$ (*eval* x) (*eval* y)
⟨*proof*⟩

lemma *cong-gen-cong*: *cong* (*gen-cong* R)
⟨*proof*⟩

lemma *gen-cong-eval-rel-fun*:
(*rel-fun* (*rel* (*gen-cong* R)) (*gen-cong* R)) *eval eval*
⟨*proof*⟩

lemma *gen-cong-eval*:
rel (*gen-cong* R) $x\ y \Longrightarrow$ *gen-cong* R (*eval* x) (*eval* y)
⟨*proof*⟩

lemma *gen-cong-idem*: *gen-cong* (*gen-cong* R) = *gen-cong* R
⟨*proof*⟩

lemma *gen-cong-rho*:
ϱ = *eval o* $f \Longrightarrow$ *rel* (*gen-cong* R) ($f\ x$) ($f\ y$) $\Longrightarrow$ *gen-cong* R ($\varrho\ x$) ($\varrho\ y$)
⟨*proof*⟩
lemma *coinduction*:
assumes *coind*: $\forall R.\ R \leq \textit{retr}\ R \longrightarrow R \leq \textit{op}\ =$
assumes *cih*: $R \leq$ *retr* (*gen-cong* R)
shows $R \leq$ *op* =
⟨*proof*⟩

end

lemma *rel-sum-case-sum*:
rel-fun (*rel-sum* $R\ S$) T (*case-sum f1 g1*) (*case-sum f2 g2*) = (*rel-fun* $R\ T$ *f1 f2*
$\wedge$ *rel-fun* $S\ T$ *g1 g2*)
⟨*proof*⟩

context
fixes *rel eval rel′ eval′ retr emb*
assumes *base*: *cong rel eval retr*
and *step*: *cong rel′ eval′ retr*
and *emb*: *eval′ o emb* = *eval*

and *emb-transfer*: *rel-fun* (*rel R*) (*rel′ R*) *emb emb*
begin

interpretation *base*: *cong rel eval retr* ⟨*proof*⟩
interpretation *step*: *cong rel′ eval′ retr* ⟨*proof*⟩

lemma *gen-cong-emb*: *base.gen-cong R* ≤ *step.gen-cong R*
⟨*proof*⟩

end

named-theorems *friend-of-corec-simps*

⟨*ML*⟩

end

8 Boolean Algebras

theory *Boolean-Algebra*
imports *Main*
begin

locale *boolean* =
fixes *conj* :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** ⊓ *70*)
and *disj* :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** ⊔ *65*)
and *compl* :: *'a* ⇒ *'a* (∼ - [*81*] *80*)
and *zero* :: *'a* (**0**)
and *one* :: *'a* (**1**)
assumes *conj-assoc*: (*x* ⊓ *y*) ⊓ *z* = *x* ⊓ (*y* ⊓ *z*)
and *disj-assoc*: (*x* ⊔ *y*) ⊔ *z* = *x* ⊔ (*y* ⊔ *z*)
and *conj-commute*: *x* ⊓ *y* = *y* ⊓ *x*
and *disj-commute*: *x* ⊔ *y* = *y* ⊔ *x*
and *conj-disj-distrib*: *x* ⊓ (*y* ⊔ *z*) = (*x* ⊓ *y*) ⊔ (*x* ⊓ *z*)
and *disj-conj-distrib*: *x* ⊔ (*y* ⊓ *z*) = (*x* ⊔ *y*) ⊓ (*x* ⊔ *z*)
and *conj-one-right* [*simp*]: *x* ⊓ **1** = *x*
and *disj-zero-right* [*simp*]: *x* ⊔ **0** = *x*
and *conj-cancel-right* [*simp*]: *x* ⊓ ∼ *x* = **0**
and *disj-cancel-right* [*simp*]: *x* ⊔ ∼ *x* = **1**
begin

sublocale *conj*: *abel-semigroup conj*
⟨*proof*⟩

sublocale *disj*: *abel-semigroup disj*
⟨*proof*⟩

lemmas *conj-left-commute* = *conj.left-commute*
lemmas *disj-left-commute* = *disj.left-commute*

lemmas *conj-ac* = *conj.assoc conj.commute conj.left-commute*
lemmas *disj-ac* = *disj.assoc disj.commute disj.left-commute*

lemma *dual*: *boolean disj conj compl one zero*
⟨*proof*⟩

8.1 Complement

lemma *complement-unique*:
assumes *1*: $a \sqcap x = \mathbf{0}$
assumes *2*: $a \sqcup x = \mathbf{1}$
assumes *3*: $a \sqcap y = \mathbf{0}$
assumes *4*: $a \sqcup y = \mathbf{1}$
shows $x = y$
⟨*proof*⟩

lemma *compl-unique*: $x \sqcap y = \mathbf{0} \Longrightarrow x \sqcup y = \mathbf{1} \Longrightarrow \sim x = y$
⟨*proof*⟩

lemma *double-compl* [*simp*]: $\sim (\sim x) = x$
⟨*proof*⟩

lemma *compl-eq-compl-iff* [*simp*]: $\sim x = \sim y \longleftrightarrow x = y$
⟨*proof*⟩

8.2 Conjunction

lemma *conj-absorb* [*simp*]: $x \sqcap x = x$
⟨*proof*⟩

lemma *conj-zero-right* [*simp*]: $x \sqcap \mathbf{0} = \mathbf{0}$
⟨*proof*⟩

lemma *compl-one* [*simp*]: $\sim \mathbf{1} = \mathbf{0}$
⟨*proof*⟩

lemma *conj-zero-left* [*simp*]: $\mathbf{0} \sqcap x = \mathbf{0}$
⟨*proof*⟩

lemma *conj-one-left* [*simp*]: $\mathbf{1} \sqcap x = x$
⟨*proof*⟩

lemma *conj-cancel-left* [*simp*]: $\sim x \sqcap x = \mathbf{0}$
⟨*proof*⟩

lemma *conj-left-absorb* [*simp*]: $x \sqcap (x \sqcap y) = x \sqcap y$
⟨*proof*⟩

lemma *conj-disj-distrib2*: $(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$

⟨*proof*⟩

lemmas *conj-disj-distribs* = *conj-disj-distrib conj-disj-distrib2*

8.3 Disjunction

lemma *disj-absorb* [*simp*]: $x \sqcup x = x$
⟨*proof*⟩

lemma *disj-one-right* [*simp*]: $x \sqcup \mathbf{1} = \mathbf{1}$
⟨*proof*⟩

lemma *compl-zero* [*simp*]: $\sim \mathbf{0} = \mathbf{1}$
⟨*proof*⟩

lemma *disj-zero-left* [*simp*]: $\mathbf{0} \sqcup x = x$
⟨*proof*⟩

lemma *disj-one-left* [*simp*]: $\mathbf{1} \sqcup x = \mathbf{1}$
⟨*proof*⟩

lemma *disj-cancel-left* [*simp*]: $\sim x \sqcup x = \mathbf{1}$
⟨*proof*⟩

lemma *disj-left-absorb* [*simp*]: $x \sqcup (x \sqcup y) = x \sqcup y$
⟨*proof*⟩

lemma *disj-conj-distrib2*: $(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
⟨*proof*⟩

lemmas *disj-conj-distribs* = *disj-conj-distrib disj-conj-distrib2*

8.4 De Morgan's Laws

lemma *de-Morgan-conj* [*simp*]: $\sim (x \sqcap y) = \sim x \sqcup \sim y$
⟨*proof*⟩

lemma *de-Morgan-disj* [*simp*]: $\sim (x \sqcup y) = \sim x \sqcap \sim y$
⟨*proof*⟩

end

8.5 Symmetric Difference

locale *boolean-xor* = *boolean* +
fixes *xor* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\oplus$ *65*)
assumes *xor-def*: $x \oplus y = (x \sqcap \sim y) \sqcup (\sim x \sqcap y)$
begin

sublocale *xor*: *abel-semigroup xor*

⟨*proof*⟩

lemmas *xor-assoc* = *xor.assoc*
lemmas *xor-commute* = *xor.commute*
lemmas *xor-left-commute* = *xor.left-commute*

lemmas *xor-ac* = *xor.assoc xor.commute xor.left-commute*

lemma *xor-def2*: $x \oplus y = (x \sqcup y) \sqcap (\sim x \sqcup \sim y)$
⟨*proof*⟩

lemma *xor-zero-right* [*simp*]: $x \oplus \mathbf{0} = x$
⟨*proof*⟩

lemma *xor-zero-left* [*simp*]: $\mathbf{0} \oplus x = x$
⟨*proof*⟩

lemma *xor-one-right* [*simp*]: $x \oplus \mathbf{1} = \sim x$
⟨*proof*⟩

lemma *xor-one-left* [*simp*]: $\mathbf{1} \oplus x = \sim x$
⟨*proof*⟩

lemma *xor-self* [*simp*]: $x \oplus x = \mathbf{0}$
⟨*proof*⟩

lemma *xor-left-self* [*simp*]: $x \oplus (x \oplus y) = y$
⟨*proof*⟩

lemma *xor-compl-left* [*simp*]: $\sim x \oplus y = \sim (x \oplus y)$
⟨*proof*⟩

lemma *xor-compl-right* [*simp*]: $x \oplus \sim y = \sim (x \oplus y)$
⟨*proof*⟩

lemma *xor-cancel-right*: $x \oplus \sim x = \mathbf{1}$
⟨*proof*⟩

lemma *xor-cancel-left*: $\sim x \oplus x = \mathbf{1}$
⟨*proof*⟩

lemma *conj-xor-distrib*: $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$
⟨*proof*⟩

lemma *conj-xor-distrib2*: $(y \oplus z) \sqcap x = (y \sqcap x) \oplus (z \sqcap x)$
⟨*proof*⟩

lemmas *conj-xor-distribs* = *conj-xor-distrib conj-xor-distrib2*

end

end

9 A general "while" combinator

theory *While-Combinator*
imports *Main*
begin

9.1 Partial version

definition *while-option* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option$ **where**
while-option b c s = (*if* $(\exists k.\ \sim\ b\ ((c\ \hat{}\hat{}\ k)\ s))$
then Some $((c\ \hat{}\hat{}\ (LEAST\ k.\ \sim\ b\ ((c\ \hat{}\hat{}\ k)\ s)))\ s)$
else None)

theorem *while-option-unfold*[*code*]:
while-option b c s = (*if b s then while-option b c* (*c s*) *else Some s*)
⟨*proof*⟩

lemma *while-option-stop2*:
while-option b c s = *Some t* $\Longrightarrow$ *EX k. t* = $(c\hat{}\hat{}k)\ s \wedge \neg\ b\ t$
⟨*proof*⟩

lemma *while-option-stop*: *while-option b c s* = *Some t* $\Longrightarrow\ \sim\ b\ t$
⟨*proof*⟩

theorem *while-option-rule*:
assumes *step*: $!!s.\ P\ s\ ==>\ b\ s\ ==>\ P\ (c\ s)$
and *result*: *while-option b c s* = *Some t*
and *init*: *P s*
shows *P t*
⟨*proof*⟩

lemma *funpow-commute*:
$[\![\forall k' < k.\ f\ (c\ ((c\hat{}\hat{}k')\ s)) = c'\ (f\ ((c\hat{}\hat{}k')\ s))]\!] \Longrightarrow f\ ((c\hat{}\hat{}k)\ s) = (c'\hat{}\hat{}k)\ (f\ s)$
⟨*proof*⟩

lemma *while-option-commute-invariant*:
assumes *Invariant*: $\bigwedge s.\ P\ s \Longrightarrow b\ s \Longrightarrow P\ (c\ s)$
assumes *TestCommute*: $\bigwedge s.\ P\ s \Longrightarrow b\ s = b'\ (f\ s)$
assumes *BodyCommute*: $\bigwedge s.\ P\ s \Longrightarrow b\ s \Longrightarrow f\ (c\ s) = c'\ (f\ s)$
assumes *Initial*: *P s*
shows *map-option f* (*while-option b c s*) = *while-option b' c'* (*f s*)
⟨*proof*⟩

lemma *while-option-commute*:
assumes $\bigwedge s.\ b\ s = b'\ (f\ s)\ \bigwedge s.\ [\![b\ s]\!] \Longrightarrow f\ (c\ s) = c'\ (f\ s)$

shows *map-option f (while-option b c s) = while-option b′ c′ (f s)*
⟨*proof*⟩

9.2 Total version

definition *while* :: *('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a*
where *while b c s = the (while-option b c s)*

lemma *while-unfold* [*code*]:
while b c s = (if b s then while b c (c s) else s)
⟨*proof*⟩

lemma *def-while-unfold*:
assumes *fdef*: *f == while test do*
shows *f x = (if test x then f(do x) else x)*
⟨*proof*⟩

The proof rule for *while*, where P is the invariant.

theorem *while-rule-lemma*:
assumes *invariant*: *!!s. P s ==> b s ==> P (c s)*
and *terminate*: *!!s. P s ==> ¬ b s ==> Q s*
and *wf*: *wf {(t, s). P s ∧ b s ∧ t = c s}*
shows *P s ⟹ Q (while b c s)*
⟨*proof*⟩

theorem *while-rule*:
[| P s;
!!s. [| P s; b s |] ==> P (c s);
!!s. [| P s; ¬ b s |] ==> Q s;
wf r;
!!s. [| P s; b s |] ==> (c s, s) ∈ r |] ==>
Q (while b c s)
⟨*proof*⟩

Proving termination:

theorem *wf-while-option-Some*:
assumes *wf {(t, s). (P s ∧ b s) ∧ t = c s}*
and *!!s. P s ⟹ b s ⟹ P(c s)* **and** *P s*
shows *EX t. while-option b c s = Some t*
⟨*proof*⟩

lemma *wf-rel-while-option-Some*:
assumes *wf*: *wf R*
assumes *smaller*: *⋀s. P s ∧ b s ⟹ (c s, s) ∈ R*
assumes *inv*: *⋀s. P s ∧ b s ⟹ P(c s)*
assumes *init*: *P s*
shows *∃ t. while-option b c s = Some t*
⟨*proof*⟩

theorem *measure-while-option-Some*: **fixes** *f* :: *'s ⇒ nat*

shows $(!!s.\ P\ s \Longrightarrow b\ s \Longrightarrow P(c\ s) \wedge f(c\ s) < f\ s)$
$\Longrightarrow P\ s \Longrightarrow EX\ t.\ while\text{-}option\ b\ c\ s = Some\ t$
⟨*proof*⟩

Kleene iteration starting from the empty set and assuming some finite bounding set:

lemma *while-option-finite-subset-Some*: **fixes** $C :: 'a\ set$
assumes *mono f* **and** $!!X.\ X \subseteq C \Longrightarrow f\ X \subseteq C$ **and** *finite C*
shows $\exists P.\ while\text{-}option\ (\lambda A.\ f\ A \neq A)\ f\ \{\} = Some\ P$
⟨*proof*⟩

lemma *lfp-the-while-option*:
assumes *mono f* **and** $!!X.\ X \subseteq C \Longrightarrow f\ X \subseteq C$ **and** *finite C*
shows $lfp\ f = the(while\text{-}option\ (\lambda A.\ f\ A \neq A)\ f\ \{\})$
⟨*proof*⟩

lemma *lfp-while*:
assumes *mono f* **and** $!!X.\ X \subseteq C \Longrightarrow f\ X \subseteq C$ **and** *finite C*
shows $lfp\ f = while\ (\lambda A.\ f\ A \neq A)\ f\ \{\}$
⟨*proof*⟩

lemma *wf-finite-less*:
assumes *finite* $(C :: 'a{::}order\ set)$
shows $wf\ \{(x,\ y).\ \{x,\ y\} \subseteq C \wedge x < y\}$
⟨*proof*⟩

lemma *wf-finite-greater*:
assumes *finite* $(C :: 'a{::}order\ set)$
shows $wf\ \{(x,\ y).\ \{x,\ y\} \subseteq C \wedge y < x\}$
⟨*proof*⟩

lemma *while-option-finite-increasing-Some*:
fixes $f :: 'a{::}order \Rightarrow 'a$
assumes *mono f* **and** *finite* $(UNIV :: 'a\ set)$ **and** $s \leq f\ s$
shows $\exists P.\ while\text{-}option\ (\lambda A.\ f\ A \neq A)\ f\ s = Some\ P$
⟨*proof*⟩

lemma *lfp-the-while-option-lattice*:
fixes $f :: 'a{::}complete\text{-}lattice \Rightarrow 'a$
assumes *mono f* **and** *finite* $(UNIV :: 'a\ set)$
shows $lfp\ f = the\ (while\text{-}option\ (\lambda A.\ f\ A \neq A)\ f\ bot)$
⟨*proof*⟩

lemma *lfp-while-lattice*:
fixes $f :: 'a{::}complete\text{-}lattice \Rightarrow 'a$
assumes *mono f* **and** *finite* $(UNIV :: 'a\ set)$
shows $lfp\ f = while\ (\lambda A.\ f\ A \neq A)\ f\ bot$
⟨*proof*⟩

lemma *while-option-finite-decreasing-Some*:
fixes $f :: 'a::order \Rightarrow 'a$
assumes *mono f* **and** *finite* (*UNIV* :: $'a$ *set*) **and** $f\ s \leq s$
shows $\exists P.$ *while-option* $(\lambda A.\ f\ A \neq A)\ f\ s =$ *Some P*
⟨*proof*⟩

lemma *gfp-the-while-option-lattice*:
fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'a$
assumes *mono f* **and** *finite* (*UNIV* :: $'a$ *set*)
shows *gfp f* = *the*(*while-option* $(\lambda A.\ f\ A \neq A)\ f$ *top*)
⟨*proof*⟩

lemma *gfp-while-lattice*:
fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'a$
assumes *mono f* **and** *finite* (*UNIV* :: $'a$ *set*)
shows *gfp f* = *while* $(\lambda A.\ f\ A \neq A)\ f$ *top*
⟨*proof*⟩

Computing the reflexive, transitive closure by iterating a successor function. Stops when an element is found that dos not satisfy the test.

More refined (and hence more efficient) versions can be found in ITP 2011 paper by Nipkow (the theories are in the AFP entry Flyspeck by Nipkow) and the AFP article Executable Transitive Closures by René Thiemann.

context
fixes $p :: 'a \Rightarrow bool$
and $f :: 'a \Rightarrow 'a\ list$
and $x :: 'a$
begin

qualified fun *rtrancl-while-test* :: $'a\ list \times 'a\ set \Rightarrow bool$
where *rtrancl-while-test* $(ws,\text{-}) = (ws \neq [] \wedge p(hd\ ws))$

qualified fun *rtrancl-while-step* :: $'a\ list \times 'a\ set \Rightarrow 'a\ list \times 'a\ set$
where *rtrancl-while-step* $(ws,\ Z) =$
(*let* $x = hd\ ws$; *new* = *remdups* (*filter* $(\lambda y.\ y \notin Z)\ (f\ x)$)
in $(new\ @\ tl\ ws,\ set\ new \cup Z)$)

definition *rtrancl-while* :: $('a\ list * 'a\ set)\ option$
where *rtrancl-while* = *while-option rtrancl-while-test rtrancl-while-step* $([x],\{x\})$

qualified fun *rtrancl-while-invariant* :: $'a\ list \times 'a\ set \Rightarrow bool$
where *rtrancl-while-invariant* $(ws,\ Z) =$
$(x \in Z \wedge set\ ws \subseteq Z \wedge distinct\ ws \wedge \{(x,y).\ y \in set(f\ x)\}\ ``\ (Z - set\ ws) \subseteq Z \wedge$
$Z \subseteq \{(x,y).\ y \in set(f\ x)\}\hat{}*\ ``\ \{x\} \wedge (\forall z{\in}Z - set\ ws.\ p\ z))$

qualified lemma *rtrancl-while-invariant*:
assumes *inv*: *rtrancl-while-invariant st* **and** *test*: *rtrancl-while-test st*
shows *rtrancl-while-invariant* (*rtrancl-while-step st*)

⟨*proof*⟩

lemma *rtrancl-while-Some*: **assumes** *rtrancl-while* = *Some*(*ws*,*Z*)
shows *if ws* = []
then Z = {(*x*,*y*). *y* ∈ *set*(*f x*)} ^* `` {*x*} ∧ (∀ *z*∈*Z*. *p z*)
else ¬*p*(*hd ws*) ∧ *hd ws* ∈ {(*x*,*y*). *y* ∈ *set*(*f x*)} ^* `` {*x*}
⟨*proof*⟩

lemma *rtrancl-while-finite-Some*:
assumes *finite* ({(*x*, *y*). *y* ∈ *set* (*f x*)}* `` {*x*}) (**is** *finite ?Cl*)
shows ∃ *y*. *rtrancl-while* = *Some y*
⟨*proof*⟩

end

end

10 The Bourbaki-Witt tower construction for transfinite iteration

theory *Bourbaki-Witt-Fixpoint*
imports *While-Combinator*
begin

lemma *ChainsI* [*intro?*]:
(⋀*a b*. ⟦ *a* ∈ *Y*; *b* ∈ *Y* ⟧ ⟹ (*a*, *b*) ∈ *r* ∨ (*b*, *a*) ∈ *r*) ⟹ *Y* ∈ *Chains r*
⟨*proof*⟩

lemma *in-Chains-subset*: ⟦ *M* ∈ *Chains r*; *M'* ⊆ *M* ⟧ ⟹ *M'* ∈ *Chains r*
⟨*proof*⟩

lemma *in-ChainsD*: ⟦ *M* ∈ *Chains r*; *x* ∈ *M*; *y* ∈ *M* ⟧ ⟹ (*x*, *y*) ∈ *r* ∨ (*y*, *x*) ∈ *r*
⟨*proof*⟩

lemma *Chains-FieldD*: ⟦ *M* ∈ *Chains r*; *x* ∈ *M* ⟧ ⟹ *x* ∈ *Field r*
⟨*proof*⟩

lemma *in-Chains-conv-chain*: *M* ∈ *Chains r* ⟷ *Complete-Partial-Order.chain* (λ*x y*. (*x*, *y*) ∈ *r*) *M*
⟨*proof*⟩

lemma *partial-order-on-trans*:
⟦ *partial-order-on A r*; (*x*, *y*) ∈ *r*; (*y*, *z*) ∈ *r* ⟧ ⟹ (*x*, *z*) ∈ *r*
⟨*proof*⟩

locale *bourbaki-witt-fixpoint* =
fixes *lub* :: *'a set* ⇒ *'a*

and *leq* :: $('a \times 'a)$ *set*
and *f* :: $'a \Rightarrow 'a$
assumes *po*: *Partial-order leq*
and *lub-least*: $[\![M \in Chains\ leq;\ M \neq \{\};\ \bigwedge x.\ x \in M \Longrightarrow (x,\ z) \in leq\ ]\!] \Longrightarrow (lub\ M,\ z) \in leq$
and *lub-upper*: $[\![M \in Chains\ leq;\ x \in M\ ]\!] \Longrightarrow (x,\ lub\ M) \in leq$
and *lub-in-Field*: $[\![M \in Chains\ leq;\ M \neq \{\}\ ]\!] \Longrightarrow lub\ M \in Field\ leq$
and *increasing*: $\bigwedge x.\ x \in Field\ leq \Longrightarrow (x,\ f\ x) \in leq$
begin

lemma *leq-trans*: $[\![(x,\ y) \in leq;\ (y,\ z) \in leq\ ]\!] \Longrightarrow (x,\ z) \in leq$
⟨*proof*⟩

lemma *leq-refl*: $x \in Field\ leq \Longrightarrow (x,\ x) \in leq$
⟨*proof*⟩

lemma *leq-antisym*: $[\![(x,\ y) \in leq;\ (y,\ x) \in leq\ ]\!] \Longrightarrow x = y$
⟨*proof*⟩

inductive-set *iterates-above* :: $'a \Rightarrow 'a\ set$
for *a*
where
base: $a \in iterates\text{-}above\ a$
| *step*: $x \in iterates\text{-}above\ a \Longrightarrow f\ x \in iterates\text{-}above\ a$
| *Sup*: $[\![M \in Chains\ leq;\ M \neq \{\};\ \bigwedge x.\ x \in M \Longrightarrow x \in iterates\text{-}above\ a\ ]\!] \Longrightarrow lub\ M \in iterates\text{-}above\ a$

definition *fixp-above* :: $'a \Rightarrow 'a$
where $fixp\text{-}above\ a = (if\ a \in Field\ leq\ then\ lub\ (iterates\text{-}above\ a)\ else\ a)$

lemma *fixp-above-outside*: $a \notin Field\ leq \Longrightarrow fixp\text{-}above\ a = a$
⟨*proof*⟩

lemma *fixp-above-inside*: $a \in Field\ leq \Longrightarrow fixp\text{-}above\ a = lub\ (iterates\text{-}above\ a)$
⟨*proof*⟩

context
notes *leq-refl* [*intro*!, *simp*]
and *base* [*intro*]
and *step* [*intro*]
and *Sup* [*intro*]
and *leq-trans* [*trans*]
begin

lemma *iterates-above-le-f*: $[\![x \in iterates\text{-}above\ a;\ a \in Field\ leq\ ]\!] \Longrightarrow (x,\ f\ x) \in leq$
⟨*proof*⟩

lemma *iterates-above-Field*: $[\![x \in iterates\text{-}above\ a;\ a \in Field\ leq\ ]\!] \Longrightarrow x \in Field$

leq
⟨*proof*⟩

lemma *iterates-above-ge*:
assumes *y*: $y \in$ *iterates-above a*
and *a*: $a \in$ *Field leq*
shows $(a, y) \in$ *leq*
⟨*proof*⟩

lemma *iterates-above-lub*:
assumes *M*: $M \in$ *Chains leq*
and *nempty*: $M \neq \{\}$
and *upper*: $\bigwedge y.\ y \in M \Longrightarrow \exists z \in M.\ (y, z) \in leq \wedge z \in$ *iterates-above a*
shows *lub M* $\in$ *iterates-above a*
⟨*proof*⟩

lemma *iterates-above-successor*:
assumes *y*: $y \in$ *iterates-above a*
and *a*: $a \in$ *Field leq*
shows $y = a \vee y \in$ *iterates-above* $(f\ a)$
⟨*proof*⟩

lemma *iterates-above-Sup-aux*:
assumes *M*: $M \in$ *Chains leq M* $\neq \{\}$
and *M'*: $M' \in$ *Chains leq M'* $\neq \{\}$
and *comp*: $\bigwedge x.\ x \in M \Longrightarrow x \in$ *iterates-above* (*lub M'*) $\vee$ *lub M'* $\in$ *iterates-above x*
shows (*lub M*, *lub M'*) $\in$ *leq* $\vee$ *lub M* $\in$ *iterates-above* (*lub M'*)
⟨*proof*⟩

lemma *iterates-above-triangle*:
assumes *x*: $x \in$ *iterates-above a*
and *y*: $y \in$ *iterates-above a*
and *a*: $a \in$ *Field leq*
shows $x \in$ *iterates-above y* $\vee$ $y \in$ *iterates-above x*
⟨*proof*⟩

lemma *chain-iterates-above*:
assumes *a*: $a \in$ *Field leq*
shows *iterates-above a* $\in$ *Chains leq* (**is** *?C* $\in$ -)
⟨*proof*⟩

lemma *fixp-iterates-above*: *fixp-above a* $\in$ *iterates-above a*
⟨*proof*⟩

lemma *fixp-above-Field*: $a \in$ *Field leq* $\Longrightarrow$ *fixp-above a* $\in$ *Field leq*
⟨*proof*⟩

lemma *fixp-above-unfold*:

assumes *a*: $a \in \mathit{Field}\ \mathit{leq}$
shows *fixp-above* $a = f\ (\mathit{fixp\text{-}above}\ a)$ (**is** $?a = f\ ?a$)
⟨*proof*⟩

end

lemma *fixp-above-induct* [*case-names adm base step*]:
assumes *adm*: *ccpo.admissible lub* $(\lambda x\ y.\ (x,\ y) \in \mathit{leq})\ P$
and *base*: $P\ a$
and *step*: $\bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)$
shows P (*fixp-above a*)
⟨*proof*⟩

end

10.1 Connect with the while combinator for executability on chain-finite lattices.

context *bourbaki-witt-fixpoint* **begin**

lemma *in-Chains-finite*: — Translation from $[\![\mathit{Complete\text{-}Partial\text{-}Order.chain}\ op \leq\ ?A;\ \mathit{finite}\ ?A;\ ?A \neq \{\}]\!] \Longrightarrow \mathit{Sup}\ ?A \in ?A$.
assumes $M \in \mathit{Chains}\ \mathit{leq}$
and $M \neq \{\}$
and *finite M*
shows *lub* $M \in M$
⟨*proof*⟩

lemma *fun-pow-iterates-above*: $(f\ \hat{}\hat{}\ k)\ a \in \mathit{iterates\text{-}above}\ a$
⟨*proof*⟩

lemma *chfin-iterates-above-fun-pow*:
assumes $x \in \mathit{iterates\text{-}above}\ a$
assumes $\forall M \in \mathit{Chains}\ \mathit{leq}.\ \mathit{finite}\ M$
shows $\exists j.\ x = (f\ \hat{}\hat{}\ j)\ a$
⟨*proof*⟩

lemma *Chain-finite-iterates-above-fun-pow-iff*:
assumes $\forall M \in \mathit{Chains}\ \mathit{leq}.\ \mathit{finite}\ M$
shows $x \in \mathit{iterates\text{-}above}\ a \longleftrightarrow (\exists j.\ x = (f\ \hat{}\hat{}\ j)\ a)$
⟨*proof*⟩

lemma *fixp-above-Kleene-iter-ex*:
assumes $(\forall M \in \mathit{Chains}\ \mathit{leq}.\ \mathit{finite}\ M)$
obtains k **where** $\mathit{fixp\text{-}above}\ a = (f\ \hat{}\hat{}\ k)\ a$
⟨*proof*⟩

context fixes *a* **assumes** *a*: $a \in \mathit{Field}\ \mathit{leq}$ **begin**

lemma *funpow-Field-leq*: $(f \mathbin{\hat{}\hat{}} k)\ a \in \mathit{Field}\ \mathit{leq}$
⟨*proof*⟩

lemma *funpow-prefix*: $j < k \Longrightarrow ((f \mathbin{\hat{}\hat{}} j)\ a, (f \mathbin{\hat{}\hat{}} k)\ a) \in \mathit{leq}$
⟨*proof*⟩

lemma *funpow-suffix*: $(f \mathbin{\hat{}\hat{}} \mathit{Suc}\ k)\ a = (f \mathbin{\hat{}\hat{}} k)\ a \Longrightarrow ((f \mathbin{\hat{}\hat{}} (j + k))\ a, (f \mathbin{\hat{}\hat{}} k)\ a) \in \mathit{leq}$
⟨*proof*⟩

lemma *funpow-stability*: $(f \mathbin{\hat{}\hat{}} \mathit{Suc}\ k)\ a = (f \mathbin{\hat{}\hat{}} k)\ a \Longrightarrow ((f \mathbin{\hat{}\hat{}} j)\ a, (f \mathbin{\hat{}\hat{}} k)\ a) \in \mathit{leq}$
⟨*proof*⟩

lemma *funpow-in-Chains*: $\{(f \mathbin{\hat{}\hat{}} k)\ a \mid k.\ \mathit{True}\} \in \mathit{Chains}\ \mathit{leq}$
⟨*proof*⟩

lemma *fixp-above-Kleene-iter*:
assumes $\forall M \in \mathit{Chains}\ \mathit{leq}.\ \mathit{finite}\ M$
assumes $(f \mathbin{\hat{}\hat{}} \mathit{Suc}\ k)\ a = (f \mathbin{\hat{}\hat{}} k)\ a$
shows $\mathit{fixp\text{-}above}\ a = (f \mathbin{\hat{}\hat{}} k)\ a$
⟨*proof*⟩

context assumes *chfin*: $\forall M \in \mathit{Chains}\ \mathit{leq}.\ \mathit{finite}\ M$ **begin**

lemma *Chain-finite-wf*: $\mathit{wf}\ \{(f\ ((f \mathbin{\hat{}\hat{}} k)\ a), (f \mathbin{\hat{}\hat{}} k)\ a) \mid k.\ f\ ((f \mathbin{\hat{}\hat{}} k)\ a) \neq (f \mathbin{\hat{}\hat{}} k)\ a\}$
⟨*proof*⟩

lemma *while-option-finite-increasing*: $\exists P.\ \mathit{while\text{-}option}\ (\lambda A.\ f\ A \neq A)\ f\ a = \mathit{Some}\ P$
⟨*proof*⟩

lemma *fixp-above-the-while-option*: $\mathit{fixp\text{-}above}\ a = \mathit{the}\ (\mathit{while\text{-}option}\ (\lambda A.\ f\ A \neq A)\ f\ a)$
⟨*proof*⟩

lemma *fixp-above-conv-while*: $\mathit{fixp\text{-}above}\ a = \mathit{while}\ (\lambda A.\ f\ A \neq A)\ f\ a$
⟨*proof*⟩

end

end

end

lemma *bourbaki-witt-fixpoint-complete-latticeI*:
fixes $f :: 'a{::}\mathit{complete\text{-}lattice} \Rightarrow 'a$
assumes $\bigwedge x.\ x \leq f\ x$

shows *bourbaki-witt-fixpoint Sup* $\{(x, y).\ x \leq y\}$ *f*
⟨*proof*⟩

end

11 Order on characters

theory *Char-ord*
imports *Main*
begin

instantiation *char* :: *linorder*
begin

definition *c1* $\leq$ *c2* $\longleftrightarrow$ *nat-of-char c1* $\leq$ *nat-of-char c2*
definition *c1* $<$ *c2* $\longleftrightarrow$ *nat-of-char c1* $<$ *nat-of-char c2*

instance
⟨*proof*⟩

end

lemma *less-eq-char-simps*:
0 $\leq$ *c*
Char k $\leq$ *0* $\longleftrightarrow$ *numeral k mod 256* $=$ (*0* :: *nat*)
Char k $\leq$ *Char l* $\longleftrightarrow$ *numeral k mod 256* $\leq$ (*numeral l mod 256* :: *nat*)
for *c* :: *char*
⟨*proof*⟩

lemma *less-char-simps*:
$\neg$ *c* $<$ *0*
0 $<$ *Char k* $\longleftrightarrow$ (*0* :: *nat*) $<$ *numeral k mod 256*
Char k $<$ *Char l* $\longleftrightarrow$ *numeral k mod 256* $<$ (*numeral l mod 256* :: *nat*)
for *c* :: *char*
⟨*proof*⟩

instantiation *char* :: *distrib-lattice*
begin

definition (*inf* :: *char* $\Rightarrow$ -) $=$ *min*
definition (*sup* :: *char* $\Rightarrow$ -) $=$ *max*

instance
⟨*proof*⟩

end

instantiation *String.literal* :: *linorder*
begin

context includes *literal.lifting*
begin

lift-definition *less-literal* :: *String.literal* ⇒ *String.literal* ⇒ *bool*
  is *ord.lexordp op* < ⟨*proof*⟩

lift-definition *less-eq-literal* :: *String.literal* ⇒ *String.literal* ⇒ *bool*
  is *ord.lexordp-eq op* < ⟨*proof*⟩

instance
⟨*proof*⟩

end

end

lemma *less-literal-code* [*code*]:
  op < = (λ*xs ys*. *ord.lexordp op* < (*String.explode xs*) (*String.explode ys*))
  ⟨*proof*⟩

lemma *less-eq-literal-code* [*code*]:
  op ≤ = (λ*xs ys*. *ord.lexordp-eq op* < (*String.explode xs*) (*String.explode ys*))
  ⟨*proof*⟩

lifting-update *literal.lifting*
lifting-forget *literal.lifting*

end

theory *Code-Test*
imports *Main*
keywords *test-code* :: *diag*
begin

11.1 YXML encoding for *term*

datatype (*plugins del*: *code size quickcheck*) *yxml-of-term* = *YXML*

lemma *yot-anything*: *x* = (*y* :: *yxml-of-term*)
⟨*proof*⟩

definition *yot-empty* :: *yxml-of-term* **where** [*code del*]: *yot-empty* = *YXML*
definition *yot-literal* :: *String.literal* ⇒ *yxml-of-term*
  where [*code del*]: *yot-literal* - = *YXML*
definition *yot-append* :: *yxml-of-term* ⇒ *yxml-of-term* ⇒ *yxml-of-term*
  where [*code del*]: *yot-append* - - = *YXML*
definition *yot-concat* :: *yxml-of-term list* ⇒ *yxml-of-term*

where [*code del*]: *yot-concat* - = *YXML*

Serialise *yxml-of-term* to native string of target language

code-printing type-constructor *yxml-of-term*
⇀ (*SML*) *string*
and (*OCaml*) *string*
and (*Haskell*) *String*
and (*Scala*) *String*
| **constant** *yot-empty*
⇀ (*SML*)
and (*OCaml*)
and (*Haskell*)
and (*Scala*)
| **constant** *yot-literal*
⇀ (*SML*) -
and (*OCaml*) -
and (*Haskell*) -
and (*Scala*) -
| **constant** *yot-append*
⇀ (*SML*) *String.concat* [(-), (-)]
and (*OCaml*) *String.concat* [(-); (-)]
and (*Haskell*) **infixr** *5* ++
and (*Scala*) **infixl** *5* +
| **constant** *yot-concat*
⇀ (*SML*) *String.concat*
and (*OCaml*) *String.concat*
and (*Haskell*) *Prelude.concat*
and (*Scala*) -.*mkString*()

Stripped-down implementations of Isabelle's XML tree with YXML encoding as defined in `~~/src/Pure/PIDE/xml.ML`, `~~/src/Pure/PIDE/yxml.ML` sufficient to encode *term* as in `~~/src/Pure/term_xml.ML`.

datatype (*plugins del*: *code size quickcheck*) *xml-tree* = *XML-Tree*

lemma *xml-tree-anything*: *x* = (*y* :: *xml-tree*)
⟨*proof*⟩

context begin
⟨*ML*⟩

type-synonym *attributes* = (*String.literal* × *String.literal*) *list*
type-synonym *body* = *xml-tree list*

definition *Elem* :: *String.literal* ⇒ *attributes* ⇒ *xml-tree list* ⇒ *xml-tree*
where [*code del*]: *Elem* - - - = *XML-Tree*

definition *Text* :: *String.literal* ⇒ *xml-tree*
where [*code del*]: *Text* - = *XML-Tree*

definition *node* :: *xml-tree list* ⇒ *xml-tree*
where *node ts* = *Elem* (*STR* ''':''') [] *ts*

definition *tagged* :: *String.literal* ⇒ *String.literal option* ⇒ *xml-tree list* ⇒ *xml-tree*
where *tagged tag x ts* = *Elem tag* (*case x of None* ⇒ [] | *Some x'* ⇒ [(*STR* ''0'', *x'*)]) *ts*

definition *list* **where** *list f xs* = *map* (*node* ∘ *f*) *xs*

definition *X* :: *yxml-of-term* **where** *X* = *yot-literal* (*STR* [*Char* (*num.Bit1* (*num.Bit0 num.One*))])
definition *Y* :: *yxml-of-term* **where** *Y* = *yot-literal* (*STR* [*Char* (*num.Bit0* (*num.Bit1 num.One*))])
definition *XY* :: *yxml-of-term* **where** *XY* = *yot-append X Y*
definition *XYX* :: *yxml-of-term* **where** *XYX* = *yot-append XY X*

end

code-datatype *xml.Elem xml.Text*

definition *yxml-string-of-xml-tree* :: *xml-tree* ⇒ *yxml-of-term* ⇒ *yxml-of-term*
where [*code del*]: *yxml-string-of-xml-tree - - = YXML*

lemma *yxml-string-of-xml-tree-code* [*code*]:
yxml-string-of-xml-tree (*xml.Elem name atts ts*) *rest* =
yot-append xml.XY (
yot-append (*yot-literal name*) (
foldr (λ(*a*, *x*) *rest*.
yot-append xml.Y (
yot-append (*yot-literal a*) (
yot-append (*yot-literal* (*STR* ''='')) (
yot-append (*yot-literal x*) *rest*)))) *atts* (
foldr yxml-string-of-xml-tree ts (
yot-append xml.XYX rest))))
yxml-string-of-xml-tree (*xml.Text s*) *rest* = *yot-append* (*yot-literal s*) *rest*
⟨*proof*⟩

definition *yxml-string-of-body* :: *xml.body* ⇒ *yxml-of-term*
where *yxml-string-of-body ts* = *foldr yxml-string-of-xml-tree ts yot-empty*

Encoding *term* into XML trees as defined in `~~/src/Pure/term_xml.ML`.

definition *xml-of-typ* :: *Typerep.typerep* ⇒ *xml.body*
where [*code del*]: *xml-of-typ* - = [*XML-Tree*]

definition *xml-of-term* :: *Code-Evaluation.term* ⇒ *xml.body*
where [*code del*]: *xml-of-term* - = [*XML-Tree*]

lemma *xml-of-typ-code* [*code*]:
xml-of-typ (*typerep.Typerep t args*) = [*xml.tagged* (*STR* ''0'') (*Some t*) (*xml.list*

xml-of-typ args)]
⟨*proof*⟩

lemma *xml-of-term-code* [*code*]:
xml-of-term (*Code-Evaluation.Const x ty*) = [*xml.tagged* (*STR ''0''*) (*Some x*) (*xml-of-typ ty*)]
xml-of-term (*Code-Evaluation.App t1 t2*) = [*xml.tagged* (*STR ''5''*) *None* [*xml.node* (*xml-of-term t1*), *xml.node* (*xml-of-term t2*)]]
xml-of-term (*Code-Evaluation.Abs x ty t*) = [*xml.tagged* (*STR ''4''*) (*Some x*) [*xml.node* (*xml-of-typ ty*), *xml.node* (*xml-of-term t*)]]
— FIXME: *Code-Evaluation.Free* is used only in *Quickcheck-Narrowing* to represent uninstantiated parameters in constructors. Here, we always translate them to `Free` variables.
xml-of-term (*Code-Evaluation.Free x ty*) = [*xml.tagged* (*STR ''1''*) (*Some x*) (*xml-of-typ ty*)]
⟨*proof*⟩

definition *yxml-string-of-term* :: *Code-Evaluation.term* ⇒ *yxml-of-term*
where *yxml-string-of-term* = *yxml-string-of-body* ∘ *xml-of-term*

11.2 Test engine and drivers

⟨*ML*⟩

end

12 Pretty syntax for lattice operations

theory *Lattice-Syntax*
imports *HOL.Complete-Lattices*
begin

notation
bot (⊥) **and**
top (⊤) **and**
inf (**infixl** ⊓ *70*) **and**
sup (**infixl** ⊔ *65*) **and**
Inf (⨅- [*900*] *900*) **and**
Sup (⨆- [*900*] *900*)

syntax
-INF1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨅-./ -) [*0*, *10*] *10*)
-INF :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨅-∈-./ -) [*0*, *0*, *10*] *10*)
-SUP1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨆-./ -) [*0*, *10*] *10*)
-SUP :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨆-∈-./ -) [*0*, *0*, *10*] *10*)

end

13 A combinator to build partial equivalence relations from a predicate and an equivalence relation

theory *Combine-PER*
imports *Main Lattice-Syntax*
begin

definition *combine-per* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
where *combine-per P R* $= (\lambda x\ y.\ P\ x \wedge P\ y) \sqcap R$

lemma *combine-per-simp* [*simp*]:
combine-per P R x y $\longleftrightarrow$ *P x* $\wedge$ *P y* $\wedge$ *x* $\approx$ *y* **for** *R* (**infixl** $\approx$ *50*)
⟨*proof*⟩

lemma *combine-per-top* [*simp*]: *combine-per* $\top$ *R* = *R*
⟨*proof*⟩

lemma *combine-per-eq* [*simp*]: *combine-per P HOL.eq* = *HOL.eq* $\sqcap$ $(\lambda x\ y.\ P\ x)$
⟨*proof*⟩

lemma *symp-combine-per*: *symp R* $\Longrightarrow$ *symp* (*combine-per P R*)
⟨*proof*⟩

lemma *transp-combine-per*: *transp R* $\Longrightarrow$ *transp* (*combine-per P R*)
⟨*proof*⟩

lemma *combine-perI*: *P x* $\Longrightarrow$ *P y* $\Longrightarrow$ *x* $\approx$ *y* $\Longrightarrow$ *combine-per P R x y* **for** *R*
(**infixl** $\approx$ *50*)
⟨*proof*⟩

lemma *symp-combine-per-symp*: *symp R* $\Longrightarrow$ *symp* (*combine-per P R*)
⟨*proof*⟩

lemma *transp-combine-per-transp*: *transp R* $\Longrightarrow$ *transp* (*combine-per P R*)
⟨*proof*⟩

lemma *equivp-combine-per-part-equivp* [*intro?*]:
fixes *R* (**infixl** $\approx$ *50*)
assumes $\exists x.\ P\ x$ **and** *equivp R*
shows *part-equivp* (*combine-per P R*)
⟨*proof*⟩

end

14 Formalisation of chain-complete partial orders, continuity and admissibility

theory *Complete-Partial-Order2* **imports**
Main Lattice-Syntax
begin

lemma *chain-transfer* [*transfer-rule*]:
includes *lifting-syntax*
shows ((*A* ===> *A* ===> *op* =) ===> *rel-set A* ===> *op* =) *Complete-Partial-Order.chain Complete-Partial-Order.chain*
⟨*proof*⟩

lemma *linorder-chain* [*simp*, *intro*!]:
fixes *Y* :: - :: *linorder set*
shows *Complete-Partial-Order.chain op* ≤ *Y*
⟨*proof*⟩

lemma *fun-lub-apply*: ⋀*Sup*. *fun-lub Sup Y x* = *Sup* ((λ*f*. *f x*) ‘ *Y*)
⟨*proof*⟩

lemma *fun-lub-empty* [*simp*]: *fun-lub lub* {} = (λ-. *lub* {})
⟨*proof*⟩

lemma *chain-fun-ordD*:
assumes *Complete-Partial-Order.chain* (*fun-ord le*) *Y*
shows *Complete-Partial-Order.chain le* ((λ*f*. *f x*) ‘ *Y*)
⟨*proof*⟩

lemma *chain-Diff*:
Complete-Partial-Order.chain ord A
⟹ *Complete-Partial-Order.chain ord* (*A* − *B*)
⟨*proof*⟩

lemma *chain-rel-prodD1*:
Complete-Partial-Order.chain (*rel-prod orda ordb*) *Y*
⟹ *Complete-Partial-Order.chain orda* (*fst* ‘ *Y*)
⟨*proof*⟩

lemma *chain-rel-prodD2*:
Complete-Partial-Order.chain (*rel-prod orda ordb*) *Y*
⟹ *Complete-Partial-Order.chain ordb* (*snd* ‘ *Y*)
⟨*proof*⟩

context *ccpo* **begin**

lemma *ccpo-fun*: *class.ccpo* (*fun-lub Sup*) (*fun-ord op* ≤) (*mk-less* (*fun-ord op* ≤))
⟨*proof*⟩

lemma *ccpo-Sup-below-iff*: *Complete-Partial-Order.chain op* $\leq$ *Y* $\Longrightarrow$ *Sup Y* $\leq$ *x* $\longleftrightarrow$ $(\forall\, y{\in}Y.\ y \leq x)$
⟨*proof*⟩

lemma *Sup-minus-bot*:
assumes *chain*: *Complete-Partial-Order.chain op* $\leq$ *A*
shows $\bigsqcup(A - \{\bigsqcup\{\}\}) = \bigsqcup A$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *mono-lub*:
fixes *le-b* (**infix** $\sqsubseteq$ *60*)
assumes *chain*: *Complete-Partial-Order.chain* (*fun-ord op* $\leq$) *Y*
and *mono*: $\bigwedge f.\ f \in Y \Longrightarrow$ *monotone le-b op* $\leq f$
shows *monotone op* $\sqsubseteq$ *op* $\leq$ (*fun-lub Sup Y*)
⟨*proof*⟩

context
fixes *le-b* (**infix** $\sqsubseteq$ *60*) **and** *Y f*
assumes *chain*: *Complete-Partial-Order.chain le-b Y*
and *mono1*: $\bigwedge y.\ y \in Y \Longrightarrow$ *monotone le-b op* $\leq$ $(\lambda x.\ f\ x\ y)$
and *mono2*: $\bigwedge x\ a\ b.\ [\![\ x \in Y;\ a \sqsubseteq b;\ a \in Y;\ b \in Y\ ]\!] \Longrightarrow f\ x\ a \leq f\ x\ b$
begin

lemma *Sup-mono*:
assumes *le*: $x \sqsubseteq y$ **and** *x*: $x \in Y$ **and** *y*: $y \in Y$
shows $\bigsqcup(f\ x\ `\ Y) \leq \bigsqcup(f\ y\ `\ Y)$ (**is** *- ≤ ?rhs*)
⟨*proof*⟩

lemma *diag-Sup*: $\bigsqcup((\lambda x.\ \bigsqcup(f\ x\ `\ Y))\ `\ Y) = \bigsqcup((\lambda x.\ f\ x\ x)\ `\ Y)$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

lemma *Sup-image-mono-le*:
fixes *le-b* (**infix** $\sqsubseteq$ *60*) **and** *Sup-b* ($\bigvee$- [*900*] *900*)
assumes *ccpo*: *class.ccpo Sup-b op* $\sqsubseteq$ *lt-b*
assumes *chain*: *Complete-Partial-Order.chain op* $\sqsubseteq$ *Y*
and *mono*: $\bigwedge x\ y.\ [\![\ x \sqsubseteq y;\ x \in Y\ ]\!] \Longrightarrow f\ x \leq f\ y$
shows *Sup* $(f\ `\ Y) \leq f\ (\bigvee Y)$
⟨*proof*⟩

lemma *swap-Sup*:
fixes *le-b* (**infix** $\sqsubseteq$ *60*)
assumes *Y*: *Complete-Partial-Order.chain op* $\sqsubseteq$ *Y*
and *Z*: *Complete-Partial-Order.chain* (*fun-ord op* $\leq$) *Z*
and *mono*: $\bigwedge f.\ f \in Z \Longrightarrow$ *monotone op* $\sqsubseteq$ *op* $\leq f$
shows $\bigsqcup((\lambda x.\ \bigsqcup(x\ `\ Y))\ `\ Z) = \bigsqcup((\lambda x.\ \bigsqcup((\lambda f.\ f\ x)\ `\ Z))\ `\ Y)$

(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *fixp-mono*:
assumes *fg*: *fun-ord op* ≤ *f g*
and *f*: *monotone op* ≤ *op* ≤ *f*
and *g*: *monotone op* ≤ *op* ≤ *g*
shows *ccpo-class.fixp f* ≤ *ccpo-class.fixp g*
⟨*proof*⟩

context fixes *ordb* :: *'b* ⇒ *'b* ⇒ *bool* (**infix** ⊑ *60*) **begin**

lemma *iterates-mono*:
assumes *f*: *f* ∈ *ccpo.iterates* (*fun-lub Sup*) (*fun-ord op* ≤) *F*
and *mono*: ⋀*f*. *monotone op* ⊑ *op* ≤ *f* ⟹ *monotone op* ⊑ *op* ≤ (*F f*)
shows *monotone op* ⊑ *op* ≤ *f*
⟨*proof*⟩

lemma *fixp-preserves-mono*:
assumes *mono*: ⋀*x*. *monotone* (*fun-ord op* ≤) *op* ≤ (λ*f*. *F f x*)
and *mono2*: ⋀*f*. *monotone op* ⊑ *op* ≤ *f* ⟹ *monotone op* ⊑ *op* ≤ (*F f*)
shows *monotone op* ⊑ *op* ≤ (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord op* ≤) *F*)
(**is** *monotone - - ?fixp*)
⟨*proof*⟩

end

end

lemma *monotone2monotone*:
assumes *2*: ⋀*x*. *monotone ordb ordc* (λ*y*. *f x y*)
and *t*: *monotone orda ordb* (λ*x*. *t x*)
and *1*: ⋀*y*. *monotone orda ordc* (λ*x*. *f x y*)
and *trans*: *transp ordc*
shows *monotone orda ordc* (λ*x*. *f x* (*t x*))
⟨*proof*⟩

14.1 Continuity

definition *cont* :: (*'a set* ⇒ *'a*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'b set* ⇒ *'b*) ⇒ (*'b* ⇒ *'b* ⇒ *bool*) ⇒ (*'a* ⇒ *'b*) ⇒ *bool*
where
cont luba orda lubb ordb f ⟷
(∀ *Y*. *Complete-Partial-Order.chain orda Y* ⟶ *Y* ≠ {} ⟶ *f* (*luba Y*) = *lubb* (*f* ‘ *Y*))

definition *mcont* :: (*'a set* ⇒ *'a*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'b set* ⇒ *'b*) ⇒ (*'b* ⇒ *'b* ⇒ *bool*) ⇒ (*'a* ⇒ *'b*) ⇒ *bool*
where

mcont luba orda lubb ordb f $\longleftrightarrow$
monotone orda ordb f $\wedge$ *cont luba orda lubb ordb f*

14.1.1 Theorem collection *cont-intro*

named-theorems *cont-intro continuity and admissibility intro rules*
⟨*ML*⟩

lemmas [*cont-intro*] =
call-mono
let-mono
if-mono
option.const-mono
tailrec.const-mono
bind-mono

declare *if-mono*[*simp*]

lemma *monotone-id'* [*cont-intro*]: *monotone ord ord* ($\lambda x.\ x$)
⟨*proof*⟩

lemma *monotone-applyI*:
monotone orda ordb F $\Longrightarrow$ *monotone* (*fun-ord orda*) *ordb* ($\lambda f.\ F\ (f\ x)$)
⟨*proof*⟩

lemma *monotone-if-fun* [*partial-function-mono*]:
⟦ *monotone* (*fun-ord orda*) (*fun-ord ordb*) *F*; *monotone* (*fun-ord orda*) (*fun-ord ordb*) *G* ⟧
$\Longrightarrow$ *monotone* (*fun-ord orda*) (*fun-ord ordb*) ($\lambda f\ n.$ *if c n then F f n else G f n*)
⟨*proof*⟩

lemma *monotone-fun-apply-fun* [*partial-function-mono*]:
monotone (*fun-ord* (*fun-ord ord*)) (*fun-ord ord*) ($\lambda f\ n.\ f\ t\ (g\ n)$)
⟨*proof*⟩

lemma *monotone-fun-ord-apply*:
monotone orda (*fun-ord ordb*) *f* $\longleftrightarrow$ ($\forall x.$ *monotone orda ordb* ($\lambda y.\ f\ y\ x$))
⟨*proof*⟩

context *preorder* **begin**

lemma *transp-le* [*simp*, *cont-intro*]: *transp op* $\leq$
⟨*proof*⟩

lemma *monotone-const* [*simp*, *cont-intro*]: *monotone ord op* $\leq$ (λ-. *c*)
⟨*proof*⟩

end

lemma *transp-le* [*cont-intro*, *simp*]:
class.preorder ord (*mk-less ord*) ⟹ *transp ord*
⟨*proof*⟩

context *partial-function-definitions* **begin**

declare *const-mono* [*cont-intro*, *simp*]

lemma *transp-le* [*cont-intro*, *simp*]: *transp leq*
⟨*proof*⟩

lemma *preorder* [*cont-intro*, *simp*]: *class.preorder leq* (*mk-less leq*)
⟨*proof*⟩

declare *ccpo*[*cont-intro*, *simp*]

end

lemma *contI* [*intro?*]:
(⋀*Y*. ⟦ *Complete-Partial-Order.chain orda Y*; *Y* ≠ {} ⟧ ⟹ *f* (*luba Y*) = *lubb* (*f* ‘ *Y*))
⟹ *cont luba orda lubb ordb f*
⟨*proof*⟩

lemma *contD*:
⟦ *cont luba orda lubb ordb f*; *Complete-Partial-Order.chain orda Y*; *Y* ≠ {} ⟧
⟹ *f* (*luba Y*) = *lubb* (*f* ‘ *Y*)
⟨*proof*⟩

lemma *cont-id* [*simp*, *cont-intro*]: ⋀*Sup*. *cont Sup ord Sup ord id*
⟨*proof*⟩

lemma *cont-id′* [*simp*, *cont-intro*]: ⋀*Sup*. *cont Sup ord Sup ord* (λ*x*. *x*)
⟨*proof*⟩

lemma *cont-applyI* [*cont-intro*]:
assumes *cont*: *cont luba orda lubb ordb g*
shows *cont* (*fun-lub luba*) (*fun-ord orda*) *lubb ordb* (λ*f*. *g* (*f x*))
⟨*proof*⟩

lemma *call-cont*: *cont* (*fun-lub lub*) (*fun-ord ord*) *lub ord* (λ*f*. *f t*)
⟨*proof*⟩

lemma *cont-if* [*cont-intro*]:
⟦ *cont luba orda lubb ordb f*; *cont luba orda lubb ordb g* ⟧
⟹ *cont luba orda lubb ordb* (λ*x*. *if c then f x else g x*)
⟨*proof*⟩

lemma *mcontI* [*intro?*]:

⟦ *monotone orda ordb f*; *cont luba orda lubb ordb f* ⟧ ⟹ *mcont luba orda lubb ordb f*
⟨*proof*⟩

lemma *mcont-mono*: *mcont luba orda lubb ordb f* ⟹ *monotone orda ordb f*
⟨*proof*⟩

lemma *mcont-cont* [*simp*]: *mcont luba orda lubb ordb f* ⟹ *cont luba orda lubb ordb f*
⟨*proof*⟩

lemma *mcont-monoD*:
⟦ *mcont luba orda lubb ordb f*; *orda x y* ⟧ ⟹ *ordb* (*f x*) (*f y*)
⟨*proof*⟩

lemma *mcont-contD*:
⟦ *mcont luba orda lubb ordb f*; *Complete-Partial-Order.chain orda Y*; *Y* ≠ {} ⟧
⟹ *f* (*luba Y*) = *lubb* (*f* ‘ *Y*)
⟨*proof*⟩

lemma *mcont-call* [*cont-intro*, *simp*]:
mcont (*fun-lub lub*) (*fun-ord ord*) *lub ord* (λ*f*. *f t*)
⟨*proof*⟩

lemma *mcont-id′* [*cont-intro*, *simp*]: *mcont lub ord lub ord* (λ*x*. *x*)
⟨*proof*⟩

lemma *mcont-applyI*:
mcont luba orda lubb ordb (λ*x*. *F x*) ⟹ *mcont* (*fun-lub luba*) (*fun-ord orda*) *lubb ordb* (λ*f*. *F* (*f x*))
⟨*proof*⟩

lemma *mcont-if* [*cont-intro*, *simp*]:
⟦ *mcont luba orda lubb ordb* (λ*x*. *f x*); *mcont luba orda lubb ordb* (λ*x*. *g x*) ⟧
⟹ *mcont luba orda lubb ordb* (λ*x*. *if c then f x else g x*)
⟨*proof*⟩

lemma *cont-fun-lub-apply*:
cont luba orda (*fun-lub lubb*) (*fun-ord ordb*) *f* ⟷ (∀ *x*. *cont luba orda lubb ordb* (λ*y*. *f y x*))
⟨*proof*⟩

lemma *mcont-fun-lub-apply*:
mcont luba orda (*fun-lub lubb*) (*fun-ord ordb*) *f* ⟷ (∀ *x*. *mcont luba orda lubb ordb* (λ*y*. *f y x*))
⟨*proof*⟩

context *ccpo* **begin**

lemma *cont-const* [*simp*, *cont-intro*]: *cont luba orda Sup op* $\leq$ $(\lambda x.\ c)$
⟨*proof*⟩

lemma *mcont-const* [*cont-intro*, *simp*]:
mcont luba orda Sup op $\leq$ $(\lambda x.\ c)$
⟨*proof*⟩

lemma *cont-apply*:
assumes *2*: $\bigwedge x.$ *cont lubb ordb Sup op* $\leq$ $(\lambda y.\ f\ x\ y)$
and *t*: *cont luba orda lubb ordb* $(\lambda x.\ t\ x)$
and *1*: $\bigwedge y.$ *cont luba orda Sup op* $\leq$ $(\lambda x.\ f\ x\ y)$
and *mono*: *monotone orda ordb* $(\lambda x.\ t\ x)$
and *mono2*: $\bigwedge x.$ *monotone ordb op* $\leq$ $(\lambda y.\ f\ x\ y)$
and *mono1*: $\bigwedge y.$ *monotone orda op* $\leq$ $(\lambda x.\ f\ x\ y)$
shows *cont luba orda Sup op* $\leq$ $(\lambda x.\ f\ x\ (t\ x))$
⟨*proof*⟩

lemma *mcont2mcont'*:
⟦ $\bigwedge x.$ *mcont lub' ord' Sup op* $\leq$ $(\lambda y.\ f\ x\ y)$;
$\bigwedge y.$ *mcont lub ord Sup op* $\leq$ $(\lambda x.\ f\ x\ y)$;
mcont lub ord lub' ord' $(\lambda y.\ t\ y)$ ⟧
$\Longrightarrow$ *mcont lub ord Sup op* $\leq$ $(\lambda x.\ f\ x\ (t\ x))$
⟨*proof*⟩

lemma *mcont2mcont*:
⟦*mcont lub' ord' Sup op* $\leq$ $(\lambda x.\ f\ x)$; *mcont lub ord lub' ord'* $(\lambda x.\ t\ x)$⟧
$\Longrightarrow$ *mcont lub ord Sup op* $\leq$ $(\lambda x.\ f\ (t\ x))$
⟨*proof*⟩

context
fixes *ord* :: *'b* $\Rightarrow$ *'b* $\Rightarrow$ *bool* (**infix** $\sqsubseteq$ *60*)
and *lub* :: *'b set* $\Rightarrow$ *'b* ($\bigsqcup$- [*900*] *900*)
begin

lemma *cont-fun-lub-Sup*:
assumes *chainM*: *Complete-Partial-Order.chain* (*fun-ord op* $\leq$) *M*
and *mcont* [*rule-format*]: $\forall f{\in}M.$ *mcont lub op* $\sqsubseteq$ *Sup op* $\leq$ *f*
shows *cont lub op* $\sqsubseteq$ *Sup op* $\leq$ (*fun-lub Sup M*)
⟨*proof*⟩

lemma *mcont-fun-lub-Sup*:
⟦ *Complete-Partial-Order.chain* (*fun-ord op* $\leq$) *M*;
$\forall f{\in}M.$ *mcont lub ord Sup op* $\leq$ *f* ⟧
$\Longrightarrow$ *mcont lub op* $\sqsubseteq$ *Sup op* $\leq$ (*fun-lub Sup M*)
⟨*proof*⟩

lemma *iterates-mcont*:
assumes *f*: *f* $\in$ *ccpo.iterates* (*fun-lub Sup*) (*fun-ord op* $\leq$) *F*
and *mono*: $\bigwedge f.$ *mcont lub op* $\sqsubseteq$ *Sup op* $\leq$ *f* $\Longrightarrow$ *mcont lub op* $\sqsubseteq$ *Sup op* $\leq$ (*F f*)

shows *mcont lub op* $\sqsubseteq$ *Sup op* $\leq f$
⟨*proof*⟩

lemma *fixp-preserves-mcont*:
assumes *mono*: $\bigwedge x.$ *monotone* (*fun-ord op* $\leq$) *op* $\leq (\lambda f.\ F\ f\ x)$
and *mcont*: $\bigwedge f.$ *mcont lub op* $\sqsubseteq$ *Sup op* $\leq f \Longrightarrow$ *mcont lub op* $\sqsubseteq$ *Sup op* $\leq (F\ f)$
shows *mcont lub op* $\sqsubseteq$ *Sup op* $\leq$ (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord op* $\leq$) *F*)
(**is** *mcont - - - - ?fixp*)
⟨*proof*⟩

end

context
fixes $F :: 'c \Rightarrow 'c$ **and** $U :: 'c \Rightarrow 'b \Rightarrow 'a$ **and** $C :: ('b \Rightarrow 'a) \Rightarrow 'c$ **and** *f*
assumes *mono*: $\bigwedge x.$ *monotone* (*fun-ord op* $\leq$) *op* $\leq (\lambda f.\ U\ (F\ (C\ f))\ x)$
and *eq*: $f \equiv C$ (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord op* $\leq$) $(\lambda f.\ U\ (F\ (C\ f))))$
and *inverse*: $\bigwedge f.\ U\ (C\ f) = f$
begin

lemma *fixp-preserves-mono-uc*:
assumes *mono2*: $\bigwedge f.$ *monotone ord op* $\leq (U\ f) \Longrightarrow$ *monotone ord op* $\leq (U\ (F\ f))$
shows *monotone ord op* $\leq (U\ f)$
⟨*proof*⟩

lemma *fixp-preserves-mcont-uc*:
assumes *mcont*: $\bigwedge f.$ *mcont lubb ordb Sup op* $\leq (U\ f) \Longrightarrow$ *mcont lubb ordb Sup op* $\leq (U\ (F\ f))$
shows *mcont lubb ordb Sup op* $\leq (U\ f)$
⟨*proof*⟩

end

lemmas *fixp-preserves-mono1* = *fixp-preserves-mono-uc*[*of* $\lambda x.\ x$ - $\lambda x.\ x$, *OF* - - *refl*]
lemmas *fixp-preserves-mono2* =
fixp-preserves-mono-uc[*of case-prod* - *curry*, *unfolded case-prod-curry curry-case-prod*, *OF* - - *refl*]
lemmas *fixp-preserves-mono3* =
fixp-preserves-mono-uc[*of* $\lambda f.$ *case-prod* (*case-prod f*) - $\lambda f.$ *curry* (*curry f*), *unfolded case-prod-curry curry-case-prod*, *OF* - - *refl*]
lemmas *fixp-preserves-mono4* =
fixp-preserves-mono-uc[*of* $\lambda f.$ *case-prod* (*case-prod* (*case-prod f*)) - $\lambda f.$ *curry* (*curry* (*curry f*)), *unfolded case-prod-curry curry-case-prod*, *OF* - - *refl*]

lemmas *fixp-preserves-mcont1* = *fixp-preserves-mcont-uc*[*of* $\lambda x.\ x$ - $\lambda x.\ x$, *OF* - - *refl*]
lemmas *fixp-preserves-mcont2* =

fixp-preserves-mcont-uc[*of case-prod - curry*, *unfolded case-prod-curry curry-case-prod*, *OF - - refl*]
lemmas *fixp-preserves-mcont3* =
fixp-preserves-mcont-uc[*of* $\lambda f.$ *case-prod* (*case-prod f*) - $\lambda f.$ *curry* (*curry f*), *unfolded case-prod-curry curry-case-prod*, *OF - - refl*]
lemmas *fixp-preserves-mcont4* =
fixp-preserves-mcont-uc[*of* $\lambda f.$ *case-prod* (*case-prod* (*case-prod f*)) - $\lambda f.$ *curry* (*curry* (*curry f*)), *unfolded case-prod-curry curry-case-prod*, *OF - - refl*]

end

lemma (**in** *preorder*) *monotone-if-bot*:
fixes *bot*
assumes *mono*: $\bigwedge x\ y.\ [\![\ x \le y;\ \neg\ (x \le bound)\ ]\!] \Longrightarrow ord\ (f\ x)\ (f\ y)$
and *bot*: $\bigwedge x.\ \neg\ x \le bound \Longrightarrow ord\ bot\ (f\ x)\ ord\ bot\ bot$
shows *monotone op* $\le$ *ord* ($\lambda x.$ *if* $x \le$ *bound then bot else f x*)
⟨*proof*⟩

lemma (**in** *ccpo*) *mcont-if-bot*:
fixes *bot* **and** *lub* ($\bigvee$- [*900*] *900*) **and** *ord* (**infix** $\sqsubseteq$ *60*)
assumes *ccpo*: *class.ccpo lub op* $\sqsubseteq$ *lt*
and *mono*: $\bigwedge x\ y.\ [\![\ x \le y;\ \neg\ x \le bound\ ]\!] \Longrightarrow f\ x \sqsubseteq f\ y$
and *cont*: $\bigwedge Y.\ [\![$ *Complete-Partial-Order.chain op* $\le\ Y;\ Y \neq \{\};\ \bigwedge x.\ x \in Y \Longrightarrow \neg\ x \le bound\ ]\!] \Longrightarrow f\ (\bigsqcup Y) = \bigvee(f\ `\ Y)$
and *bot*: $\bigwedge x.\ \neg\ x \le bound \Longrightarrow bot \sqsubseteq f\ x$
shows *mcont Sup op* $\le$ *lub op* $\sqsubseteq$ ($\lambda x.$ *if* $x \le$ *bound then bot else f x*) (**is** *mcont - - - - ?g*)
⟨*proof*⟩

context *partial-function-definitions* **begin**

lemma *mcont-const* [*cont-intro*, *simp*]:
mcont luba orda lub leq ($\lambda x.$ *c*)
⟨*proof*⟩

lemmas [*cont-intro*, *simp*] =
ccpo.cont-const[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemma *mono2mono*:
assumes *monotone ordb leq* ($\lambda y.$ *f y*) *monotone orda ordb* ($\lambda x.$ *t x*)
shows *monotone orda leq* ($\lambda x.$ *f* (*t x*))
⟨*proof*⟩

lemmas *mcont2mcont'* = *ccpo.mcont2mcont'*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axiom*
lemmas *mcont2mcont* = *ccpo.mcont2mcont*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]

lemmas *fixp-preserves-mono1* = *ccpo.fixp-preserves-mono1*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mono2* = *ccpo.fixp-preserves-mono2*[*OF Partial-Function.ccpo*[*OF*

partial-function-definitions-axioms]]
lemmas *fixp-preserves-mono3* = *ccpo.fixp-preserves-mono3*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mono4* = *ccpo.fixp-preserves-mono4*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mcont1* = *ccpo.fixp-preserves-mcont1*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mcont2* = *ccpo.fixp-preserves-mcont2*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mcont3* = *ccpo.fixp-preserves-mcont3*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mcont4* = *ccpo.fixp-preserves-mcont4*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemma *monotone-if-bot*:
fixes *bot*
assumes *g*: $\bigwedge x.\ g\ x = (\mathit{if}\ \mathit{leq}\ x\ \mathit{bound}\ \mathit{then}\ \mathit{bot}\ \mathit{else}\ f\ x)$
and *mono*: $\bigwedge x\ y.\ [\![\ \mathit{leq}\ x\ y;\ \neg\ \mathit{leq}\ x\ \mathit{bound}\ ]\!] \Longrightarrow \mathit{ord}\ (f\ x)\ (f\ y)$
and *bot*: $\bigwedge x.\ \neg\ \mathit{leq}\ x\ \mathit{bound} \Longrightarrow \mathit{ord}\ \mathit{bot}\ (f\ x)\ \mathit{ord}\ \mathit{bot}\ \mathit{bot}$
shows *monotone leq ord g*
⟨*proof*⟩

lemma *mcont-if-bot*:
fixes *bot*
assumes *ccpo*: *class.ccpo lub' ord* (*mk-less ord*)
and *bot*: $\bigwedge x.\ \neg\ \mathit{leq}\ x\ \mathit{bound} \Longrightarrow \mathit{ord}\ \mathit{bot}\ (f\ x)$
and *g*: $\bigwedge x.\ g\ x = (\mathit{if}\ \mathit{leq}\ x\ \mathit{bound}\ \mathit{then}\ \mathit{bot}\ \mathit{else}\ f\ x)$
and *mono*: $\bigwedge x\ y.\ [\![\ \mathit{leq}\ x\ y;\ \neg\ \mathit{leq}\ x\ \mathit{bound}\ ]\!] \Longrightarrow \mathit{ord}\ (f\ x)\ (f\ y)$
and *cont*: $\bigwedge Y.\ [\![\ \mathit{Complete\text{-}Partial\text{-}Order.chain}\ \mathit{leq}\ Y;\ Y \neq \{\};\ \bigwedge x.\ x \in Y \Longrightarrow \neg\ \mathit{leq}\ x\ \mathit{bound}\ ]\!] \Longrightarrow f\ (\mathit{lub}\ Y) = \mathit{lub'}\ (f\ `\ Y)$
shows *mcont lub leq lub' ord g*
⟨*proof*⟩

end

14.2 Admissibility

lemma *admissible-subst*:
assumes *adm*: *ccpo.admissible luba orda* $(\lambda x.\ P\ x)$
and *mcont*: *mcont lubb ordb luba orda f*
shows *ccpo.admissible lubb ordb* $(\lambda x.\ P\ (f\ x))$
⟨*proof*⟩

lemmas [*simp*, *cont-intro*] =
admissible-all
admissible-ball
admissible-const
admissible-conj

lemma *admissible-disj'* [*simp*, *cont-intro*]:
⟦ *class.ccpo lub ord* (*mk-less ord*); *ccpo.admissible lub ord P*; *ccpo.admissible lub ord Q* ⟧
⟹ *ccpo.admissible lub ord* (λ*x*. *P x* ∨ *Q x*)
⟨*proof*⟩

lemma *admissible-imp'* [*cont-intro*]:
⟦ *class.ccpo lub ord* (*mk-less ord*);
ccpo.admissible lub ord (λ*x*. ¬ *P x*);
ccpo.admissible lub ord (λ*x*. *Q x*) ⟧
⟹ *ccpo.admissible lub ord* (λ*x*. *P x* ⟶ *Q x*)
⟨*proof*⟩

lemma *admissible-imp* [*cont-intro*]:
(*Q* ⟹ *ccpo.admissible lub ord* (λ*x*. *P x*))
⟹ *ccpo.admissible lub ord* (λ*x*. *Q* ⟶ *P x*)
⟨*proof*⟩

lemma *admissible-not-mem'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-not-mem*: *ccpo.admissible Union op* ⊆ (λ*A*. *x* ∉ *A*)
⟨*proof*⟩

lemma *admissible-eqI*:
assumes *f*: *cont luba orda lub ord* (λ*x*. *f x*)
and *g*: *cont luba orda lub ord* (λ*x*. *g x*)
shows *ccpo.admissible luba orda* (λ*x*. *f x* = *g x*)
⟨*proof*⟩

corollary *admissible-eq-mcontI* [*cont-intro*]:
⟦ *mcont luba orda lub ord* (λ*x*. *f x*);
mcont luba orda lub ord (λ*x*. *g x*) ⟧
⟹ *ccpo.admissible luba orda* (λ*x*. *f x* = *g x*)
⟨*proof*⟩

lemma *admissible-iff* [*cont-intro*, *simp*]:
⟦ *ccpo.admissible lub ord* (λ*x*. *P x* ⟶ *Q x*); *ccpo.admissible lub ord* (λ*x*. *Q x* ⟶ *P x*) ⟧
⟹ *ccpo.admissible lub ord* (λ*x*. *P x* ⟷ *Q x*)
⟨*proof*⟩

context *ccpo* **begin**

lemma *admissible-leI*:
assumes *f*: *mcont luba orda Sup op* ≤ (λ*x*. *f x*)
and *g*: *mcont luba orda Sup op* ≤ (λ*x*. *g x*)
shows *ccpo.admissible luba orda* (λ*x*. *f x* ≤ *g x*)
⟨*proof*⟩

end

lemma *admissible-leI*:
fixes *ord* (**infix** $\sqsubseteq$ *60*) **and** *lub* ($\bigsqcup$- [*900*] *900*)
assumes *class.ccpo lub op* $\sqsubseteq$ (*mk-less op* $\sqsubseteq$)
and *mcont luba orda lub op* $\sqsubseteq$ ($\lambda x.$ *f x*)
and *mcont luba orda lub op* $\sqsubseteq$ ($\lambda x.$ *g x*)
shows *ccpo.admissible luba orda* ($\lambda x.$ *f x* $\sqsubseteq$ *g x*)
⟨*proof*⟩

declare *ccpo-class.admissible-leI*[*cont-intro*]

context *ccpo* **begin**

lemma *admissible-not-below*: *ccpo.admissible Sup op* $\leq$ ($\lambda x.$ $\neg$ *op* $\leq$ *x y*)
⟨*proof*⟩

end

lemma (**in** *preorder*) *preorder* [*cont-intro*, *simp*]: *class.preorder op* $\leq$ (*mk-less op* $\leq$)
⟨*proof*⟩

context *partial-function-definitions* **begin**

lemmas [*cont-intro*, *simp*] =
admissible-leI[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
ccpo.admissible-not-below[*THEN admissible-subst*, *OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

end

⟨*ML*⟩

inductive *compact* :: (*'a set* $\Rightarrow$ *'a*) $\Rightarrow$ (*'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool*) $\Rightarrow$ *'a* $\Rightarrow$ *bool*
for *lub ord x*
where *compact*:
⟦ *ccpo.admissible lub ord* ($\lambda y.$ $\neg$ *ord x y*);
ccpo.admissible lub ord ($\lambda y.$ *x* $\neq$ *y*) ⟧
$\Longrightarrow$ *compact lub ord x*

⟨*ML*⟩

context *ccpo* **begin**

lemma *compactI*:
assumes *ccpo.admissible Sup op* $\leq$ ($\lambda y.$ $\neg$ *x* $\leq$ *y*)
shows *ccpo.compact Sup op* $\leq$ *x*
⟨*proof*⟩

lemma *compact-bot*:
assumes $x = Sup\ \{\}$
shows *ccpo.compact Sup op* $\leq x$
⟨*proof*⟩

end

lemma *admissible-compact-neq'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-compact-neq*: *ccpo.compact lub ord k* $\Longrightarrow$ *ccpo.admissible lub ord* $(\lambda x.\ k \neq x)$
⟨*proof*⟩

lemma *admissible-neq-compact'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-neq-compact*: *ccpo.compact lub ord k* $\Longrightarrow$ *ccpo.admissible lub ord* $(\lambda x.\ x \neq k)$
⟨*proof*⟩

context *partial-function-definitions* **begin**

lemmas [*cont-intro*, *simp*] = *ccpo.compact-bot*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axiom*

end

context *ccpo* **begin**

lemma *fixp-strong-induct*:
assumes [*cont-intro*]: *ccpo.admissible Sup op* $\leq P$
and *mono*: *monotone op* $\leq$ *op* $\leq f$
and *bot*: $P\ (\bigsqcup \{\})$
and *step*: $\bigwedge x.\ [\![\ x \leq$ *ccpo-class.fixp f*; $P\ x\ ]\!] \Longrightarrow P\ (f\ x)$
shows P (*ccpo-class.fixp f*)
⟨*proof*⟩

end

context *partial-function-definitions* **begin**

lemma *fixp-strong-induct-uc*:
fixes $F :: 'c \Rightarrow 'c$
and $U :: 'c \Rightarrow 'b \Rightarrow 'a$
and $C :: ('b \Rightarrow 'a) \Rightarrow 'c$
and $P :: ('b \Rightarrow 'a) \Rightarrow bool$
assumes *mono*: $\bigwedge x.$ *mono-body* $(\lambda f.\ U\ (F\ (C\ f))\ x)$
and *eq*: $f \equiv C$ (*fixp-fun* $(\lambda f.\ U\ (F\ (C\ f))))$
and *inverse*: $\bigwedge f.\ U\ (C\ f) = f$
and *adm*: *ccpo.admissible lub-fun le-fun P*
and *bot*: $P\ (\lambda\text{-}.\ lub\ \{\})$
and *step*: $\bigwedge f'.\ [\![\ P\ (U\ f')$; *le-fun* $(U\ f')\ (U\ f)\ ]\!] \Longrightarrow P\ (U\ (F\ f'))$
shows $P\ (U\ f)$

⟨*proof*⟩

end

14.3 *op* = as order

definition *lub-singleton* :: (*'a set* ⇒ *'a*) ⇒ *bool*
where *lub-singleton lub* ⟷ (∀ *a*. *lub* {*a*} = *a*)

definition *the-Sup* :: *'a set* ⇒ *'a*
where *the-Sup A* = (*THE a*. *a* ∈ *A*)

lemma *lub-singleton-the-Sup* [*cont-intro*, *simp*]: *lub-singleton the-Sup*
⟨*proof*⟩

lemma (**in** *ccpo*) *lub-singleton*: *lub-singleton Sup*
⟨*proof*⟩

lemma (**in** *partial-function-definitions*) *lub-singleton* [*cont-intro*, *simp*]: *lub-singleton lub*
⟨*proof*⟩

lemma *preorder-eq* [*cont-intro*, *simp*]:
class.preorder op = (*mk-less op* =)
⟨*proof*⟩

lemma *monotone-eqI* [*cont-intro*]:
assumes *class.preorder ord* (*mk-less ord*)
shows *monotone op* = *ord f*
⟨*proof*⟩

lemma *cont-eqI* [*cont-intro*]:
fixes *f* :: *'a* ⇒ *'b*
assumes *lub-singleton lub*
shows *cont the-Sup op* = *lub ord f*
⟨*proof*⟩

lemma *mcont-eqI* [*cont-intro*, *simp*]:
⟦ *class.preorder ord* (*mk-less ord*); *lub-singleton lub* ⟧
⟹ *mcont the-Sup op* = *lub ord f*
⟨*proof*⟩

14.4 ccpo for products

definition *prod-lub* :: (*'a set* ⇒ *'a*) ⇒ (*'b set* ⇒ *'b*) ⇒ (*'a* × *'b*) *set* ⇒ *'a* × *'b*
where *prod-lub Sup-a Sup-b Y* = (*Sup-a* (*fst* ‘ *Y*), *Sup-b* (*snd* ‘ *Y*))

lemma *lub-singleton-prod-lub* [*cont-intro*, *simp*]:
⟦ *lub-singleton luba*; *lub-singleton lubb* ⟧ ⟹ *lub-singleton* (*prod-lub luba lubb*)
⟨*proof*⟩

lemma *prod-lub-empty* [*simp*]: *prod-lub luba lubb* {} = (*luba* {}, *lubb* {})
⟨*proof*⟩

lemma *preorder-rel-prodI* [*cont-intro*, *simp*]:
assumes *class.preorder orda* (*mk-less orda*)
and *class.preorder ordb* (*mk-less ordb*)
shows *class.preorder* (*rel-prod orda ordb*) (*mk-less* (*rel-prod orda ordb*))
⟨*proof*⟩

lemma *order-rel-prodI*:
assumes *a*: *class.order orda* (*mk-less orda*)
and *b*: *class.order ordb* (*mk-less ordb*)
shows *class.order* (*rel-prod orda ordb*) (*mk-less* (*rel-prod orda ordb*))
(**is** *class.order ?ord ?ord'*)
⟨*proof*⟩

lemma *monotone-rel-prodI*:
assumes *mono2*: ⋀*a*. *monotone ordb ordc* (λ*b*. *f* (*a*, *b*))
and *mono1*: ⋀*b*. *monotone orda ordc* (λ*a*. *f* (*a*, *b*))
and *a*: *class.preorder orda* (*mk-less orda*)
and *b*: *class.preorder ordb* (*mk-less ordb*)
and *c*: *class.preorder ordc* (*mk-less ordc*)
shows *monotone* (*rel-prod orda ordb*) *ordc f*
⟨*proof*⟩

lemma *monotone-rel-prodD1*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) *ordc f*
and *preorder*: *class.preorder ordb* (*mk-less ordb*)
shows *monotone orda ordc* (λ*a*. *f* (*a*, *b*))
⟨*proof*⟩

lemma *monotone-rel-prodD2*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) *ordc f*
and *preorder*: *class.preorder orda* (*mk-less orda*)
shows *monotone ordb ordc* (λ*b*. *f* (*a*, *b*))
⟨*proof*⟩

lemma *monotone-case-prodI*:
⟦ ⋀*a*. *monotone ordb ordc* (*f a*); ⋀*b*. *monotone orda ordc* (λ*a*. *f a b*);
class.preorder orda (*mk-less orda*); *class.preorder ordb* (*mk-less ordb*);
class.preorder ordc (*mk-less ordc*) ⟧
⟹ *monotone* (*rel-prod orda ordb*) *ordc* (*case-prod f*)
⟨*proof*⟩

lemma *monotone-case-prodD1*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) *ordc* (*case-prod f*)
and *preorder*: *class.preorder ordb* (*mk-less ordb*)
shows *monotone orda ordc* (λ*a*. *f a b*)

⟨*proof*⟩

lemma *monotone-case-prodD2*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) *ordc* (*case-prod f*)
and *preorder*: *class.preorder orda* (*mk-less orda*)
shows *monotone ordb ordc* (*f a*)
⟨*proof*⟩

context
fixes *orda ordb ordc*
assumes *a*: *class.preorder orda* (*mk-less orda*)
and *b*: *class.preorder ordb* (*mk-less ordb*)
and *c*: *class.preorder ordc* (*mk-less ordc*)
begin

lemma *monotone-rel-prod-iff*:
monotone (*rel-prod orda ordb*) *ordc f* ⟷
(∀ *a*. *monotone ordb ordc* (λ*b*. *f* (*a*, *b*))) ∧
(∀ *b*. *monotone orda ordc* (λ*a*. *f* (*a*, *b*)))
⟨*proof*⟩

lemma *monotone-case-prod-iff* [*simp*]:
monotone (*rel-prod orda ordb*) *ordc* (*case-prod f*) ⟷
(∀ *a*. *monotone ordb ordc* (*f a*)) ∧ (∀ *b*. *monotone orda ordc* (λ*a*. *f a b*))
⟨*proof*⟩

end

lemma *monotone-case-prod-apply-iff*:
monotone orda ordb (λ*x*. (*case-prod f x*) *y*) ⟷ *monotone orda ordb* (*case-prod* (λ*a b*. *f a b y*))
⟨*proof*⟩

lemma *monotone-case-prod-applyD*:
monotone orda ordb (λ*x*. (*case-prod f x*) *y*)
⟹ *monotone orda ordb* (*case-prod* (λ*a b*. *f a b y*))
⟨*proof*⟩

lemma *monotone-case-prod-applyI*:
monotone orda ordb (*case-prod* (λ*a b*. *f a b y*))
⟹ *monotone orda ordb* (λ*x*. (*case-prod f x*) *y*)
⟨*proof*⟩

lemma *cont-case-prod-apply-iff*:
cont luba orda lubb ordb (λ*x*. (*case-prod f x*) *y*) ⟷ *cont luba orda lubb ordb* (*case-prod* (λ*a b*. *f a b y*))
⟨*proof*⟩

lemma *cont-case-prod-applyI*:
cont luba orda lubb ordb (*case-prod* (λ*a b*. *f a b y*))
⟹ *cont luba orda lubb ordb* (λ*x*. (*case-prod f x*) *y*)
⟨*proof*⟩

lemma *cont-case-prod-applyD*:
cont luba orda lubb ordb (λ*x*. (*case-prod f x*) *y*)
⟹ *cont luba orda lubb ordb* (*case-prod* (λ*a b*. *f a b y*))
⟨*proof*⟩

lemma *mcont-case-prod-apply-iff* [*simp*]:
mcont luba orda lubb ordb (λ*x*. (*case-prod f x*) *y*) ⟷
mcont luba orda lubb ordb (*case-prod* (λ*a b*. *f a b y*))
⟨*proof*⟩

lemma *cont-prodD1*:
assumes *cont*: *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubc ordc f*
and *class.preorder orda* (*mk-less orda*)
and *luba*: *lub-singleton luba*
shows *cont lubb ordb lubc ordc* (λ*y*. *f* (*x*, *y*))
⟨*proof*⟩

lemma *cont-prodD2*:
assumes *cont*: *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubc ordc f*
and *class.preorder ordb* (*mk-less ordb*)
and *lubb*: *lub-singleton lubb*
shows *cont luba orda lubc ordc* (λ*x*. *f* (*x*, *y*))
⟨*proof*⟩

lemma *cont-case-prodD1*:
assumes *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubc ordc* (*case-prod f*)
and *class.preorder orda* (*mk-less orda*)
and *lub-singleton luba*
shows *cont lubb ordb lubc ordc* (*f x*)
⟨*proof*⟩

lemma *cont-case-prodD2*:
assumes *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubc ordc* (*case-prod f*)
and *class.preorder ordb* (*mk-less ordb*)
and *lub-singleton lubb*
shows *cont luba orda lubc ordc* (λ*x*. *f x y*)
⟨*proof*⟩

context *ccpo* **begin**

lemma *cont-prodI*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) *op* ≤ *f*
and *cont1*: ⋀*x*. *cont lubb ordb Sup op* ≤ (λ*y*. *f* (*x*, *y*))
and *cont2*: ⋀*y*. *cont luba orda Sup op* ≤ (λ*x*. *f* (*x*, *y*))

and *class.preorder orda* (*mk-less orda*)
and *class.preorder ordb* (*mk-less ordb*)
shows *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *Sup op* ≤ *f*
⟨*proof*⟩

lemma *cont-case-prodI*:
assumes *monotone* (*rel-prod orda ordb*) *op* ≤ (*case-prod f*)
**and* ⋀*x*. *cont lubb ordb Sup op* ≤ (λ*y*. *f x y*)
**and* ⋀*y*. *cont luba orda Sup op* ≤ (λ*x*. *f x y*)
and *class.preorder orda* (*mk-less orda*)
and *class.preorder ordb* (*mk-less ordb*)
shows *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *Sup op* ≤ (*case-prod f*)
⟨*proof*⟩

lemma *cont-case-prod-iff*:
⟦ *monotone* (*rel-prod orda ordb*) *op* ≤ (*case-prod f*);
class.preorder orda (*mk-less orda*); *lub-singleton luba*;
class.preorder ordb (*mk-less ordb*); *lub-singleton lubb* ⟧
⟹ *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *Sup op* ≤ (*case-prod f*) ⟷
(∀ *x*. *cont lubb ordb Sup op* ≤ (λ*y*. *f x y*)) ∧ (∀ *y*. *cont luba orda Sup op* ≤ (λ*x*. *f x y*))
⟨*proof*⟩

end

context *partial-function-definitions* **begin**

lemma *mono2mono2*:
assumes *f*: *monotone* (*rel-prod ordb ordc*) *leq* (λ(*x*, *y*). *f x y*)
and *t*: *monotone orda ordb* (λ*x*. *t x*)
and *t'*: *monotone orda ordc* (λ*x*. *t' x*)
shows *monotone orda leq* (λ*x*. *f* (*t x*) (*t' x*))
⟨*proof*⟩

lemma *cont-case-prodI* [*cont-intro*]:
⟦ *monotone* (*rel-prod orda ordb*) *leq* (*case-prod f*);
⋀*x*. *cont lubb ordb lub leq* (λ*y*. *f x y*);
⋀*y*. *cont luba orda lub leq* (λ*x*. *f x y*);
class.preorder orda (*mk-less orda*);
class.preorder ordb (*mk-less ordb*) ⟧
⟹ *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*)
⟨*proof*⟩

lemma *cont-case-prod-iff*:
⟦ *monotone* (*rel-prod orda ordb*) *leq* (*case-prod f*);
class.preorder orda (*mk-less orda*); *lub-singleton luba*;
class.preorder ordb (*mk-less ordb*); *lub-singleton lubb* ⟧
⟹ *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*) ⟷
(∀ *x*. *cont lubb ordb lub leq* (λ*y*. *f x y*)) ∧ (∀ *y*. *cont luba orda lub leq* (λ*x*. *f x y*))

⟨*proof*⟩

lemma *mcont-case-prod-iff* [*simp*]:
⟦ *class.preorder orda* (*mk-less orda*); *lub-singleton luba*;
class.preorder ordb (*mk-less ordb*); *lub-singleton lubb* ⟧
⟹ *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*) ⟷
(∀ *x*. *mcont lubb ordb lub leq* (λ*y*. *f x y*)) ∧ (∀ *y*. *mcont luba orda lub leq* (λ*x*. *f x y*))
⟨*proof*⟩

end

lemma *mono2mono-case-prod* [*cont-intro*]:
assumes ⋀*x y*. *monotone orda ordb* (λ*f*. *pair f x y*)
shows *monotone orda ordb* (λ*f*. *case-prod* (*pair f*) *x*)
⟨*proof*⟩

14.5 Complete lattices as ccpo

context *complete-lattice* **begin**

lemma *complete-lattice-ccpo*: *class.ccpo Sup op* ≤ *op* <
⟨*proof*⟩

lemma *complete-lattice-ccpo'*: *class.ccpo Sup op* ≤ (*mk-less op* ≤)
⟨*proof*⟩

lemma *complete-lattice-partial-function-definitions*:
partial-function-definitions op ≤ *Sup*
⟨*proof*⟩

lemma *complete-lattice-partial-function-definitions-dual*:
partial-function-definitions op ≥ *Inf*
⟨*proof*⟩

lemmas [*cont-intro*, *simp*] =
Partial-Function.ccpo[*OF complete-lattice-partial-function-definitions*]
Partial-Function.ccpo[*OF complete-lattice-partial-function-definitions-dual*]

lemma *mono2mono-inf*:
assumes *f*: *monotone ord op* ≤ (λ*x*. *f x*)
and *g*: *monotone ord op* ≤ (λ*x*. *g x*)
shows *monotone ord op* ≤ (λ*x*. *f x* ⊓ *g x*)
⟨*proof*⟩

lemma *mcont-const* [*simp*]: *mcont lub ord Sup op* ≤ (λ-. *c*)
⟨*proof*⟩

lemma *mono2mono-sup*:

assumes f: *monotone ord op* $\leq (\lambda x.\ f\ x)$
and g: *monotone ord op* $\leq (\lambda x.\ g\ x)$
shows *monotone ord op* $\leq (\lambda x.\ f\ x \sqcup g\ x)$
⟨*proof*⟩

lemma *Sup-image-sup*:
assumes $Y \neq \{\}$
shows $\bigsqcup (op \sqcup\ x\ `\ Y) = x \sqcup \bigsqcup Y$
⟨*proof*⟩

lemma *mcont-sup1*: *mcont Sup op* $\leq$ *Sup op* $\leq (\lambda y.\ x \sqcup y)$
⟨*proof*⟩

lemma *mcont-sup2*: *mcont Sup op* $\leq$ *Sup op* $\leq (\lambda x.\ x \sqcup y)$
⟨*proof*⟩

lemma *mcont2mcont-sup* [*cont-intro*, *simp*]:
$[\![$ *mcont lub ord Sup op* $\leq (\lambda x.\ f\ x)$;
mcont lub ord Sup op $\leq (\lambda x.\ g\ x)$ $]\!]$
$\Longrightarrow$ *mcont lub ord Sup op* $\leq (\lambda x.\ f\ x \sqcup g\ x)$
⟨*proof*⟩

end

lemmas [*cont-intro*] = *admissible-leI*[*OF complete-lattice-ccpo'*]

context *complete-distrib-lattice* **begin**

lemma *mcont-inf1*: *mcont Sup op* $\leq$ *Sup op* $\leq (\lambda y.\ x \sqcap y)$
⟨*proof*⟩

lemma *mcont-inf2*: *mcont Sup op* $\leq$ *Sup op* $\leq (\lambda x.\ x \sqcap y)$
⟨*proof*⟩

lemma *mcont2mcont-inf* [*cont-intro*, *simp*]:
$[\![$ *mcont lub ord Sup op* $\leq (\lambda x.\ f\ x)$;
mcont lub ord Sup op $\leq (\lambda x.\ g\ x)$ $]\!]$
$\Longrightarrow$ *mcont lub ord Sup op* $\leq (\lambda x.\ f\ x \sqcap g\ x)$
⟨*proof*⟩

end

interpretation *lfp*: *partial-function-definitions op* $\leq$ :: - :: *complete-lattice* $\Rightarrow$ - *Sup*
⟨*proof*⟩

⟨*ML*⟩

interpretation *gfp*: *partial-function-definitions op* $\geq$ :: - :: *complete-lattice* $\Rightarrow$ -

Inf
⟨*proof*⟩

⟨*ML*⟩

lemma *insert-mono* [*partial-function-mono*]:
monotone (*fun-ord op* ⊆) *op* ⊆ *A* ⟹ *monotone* (*fun-ord op* ⊆) *op* ⊆ (λ*y*. *insert x* (*A y*))
⟨*proof*⟩

lemma *mono2mono-insert* [*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-insert*: *monotone op* ⊆ *op* ⊆ (*insert x*)
⟨*proof*⟩

lemma *mcont2mcont-insert*[*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-insert*: *mcont Union op* ⊆ *Union op* ⊆ (*insert x*)
⟨*proof*⟩

lemma *mono2mono-image* [*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-image*: *monotone op* ⊆ *op* ⊆ (*op* ‘ *f*)
⟨*proof*⟩

lemma *cont-image*: *cont Union op* ⊆ *Union op* ⊆ (*op* ‘ *f*)
⟨*proof*⟩

lemma *mcont2mcont-image* [*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-image*: *mcont Union op* ⊆ *Union op* ⊆ (*op* ‘ *f*)
⟨*proof*⟩

context *complete-lattice* **begin**

lemma *monotone-Sup* [*cont-intro*, *simp*]:
monotone ord op ⊆ *f* ⟹ *monotone ord op* ≤ (λ*x*. ⨆*f x*)
⟨*proof*⟩

lemma *cont-Sup*:
assumes *cont lub ord Union op* ⊆ *f*
shows *cont lub ord Sup op* ≤ (λ*x*. ⨆*f x*)
⟨*proof*⟩

lemma *mcont-Sup*: *mcont lub ord Union op* ⊆ *f* ⟹ *mcont lub ord Sup op* ≤ (λ*x*. ⨆*f x*)
⟨*proof*⟩

lemma *monotone-SUP*:
⟦ *monotone ord op* ⊆ *f*; ⋀*y*. *monotone ord op* ≤ (λ*x*. *g x y*) ⟧ ⟹ *monotone ord op* ≤ (λ*x*. ⨆*y*∈*f x*. *g x y*)
⟨*proof*⟩

lemma *monotone-SUP2*:
(⋀*y*. *y* ∈ *A* ⟹ *monotone ord op* ≤ (λ*x*. *g x y*)) ⟹ *monotone ord op* ≤ (λ*x*. ⨆ *y*∈*A*. *g x y*)
⟨*proof*⟩

lemma *cont-SUP*:
assumes *f*: *mcont lub ord Union op* ⊆ *f*
and *g*: ⋀*y*. *mcont lub ord Sup op* ≤ (λ*x*. *g x y*)
shows *cont lub ord Sup op* ≤ (λ*x*. ⨆ *y*∈*f x*. *g x y*)
⟨*proof*⟩

lemma *mcont-SUP* [*cont-intro*, *simp*]:
⟦ *mcont lub ord Union op* ⊆ *f*; ⋀*y*. *mcont lub ord Sup op* ≤ (λ*x*. *g x y*) ⟧
⟹ *mcont lub ord Sup op* ≤ (λ*x*. ⨆ *y*∈*f x*. *g x y*)
⟨*proof*⟩

end

lemma *admissible-Ball* [*cont-intro*, *simp*]:
⟦ ⋀*x*. *ccpo.admissible lub ord* (λ*A*. *P A x*);
mcont lub ord Union op ⊆ *f*;
class.ccpo lub ord (*mk-less ord*) ⟧
⟹ *ccpo.admissible lub ord* (λ*A*. ∀ *x*∈*f A*. *P A x*)
⟨*proof*⟩

lemma *admissible-Bex'*[*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-Bex*: *ccpo.admissible Union op* ⊆ (λ*A*. ∃ *x*∈*A*. *P x*)
⟨*proof*⟩

14.6 Parallel fixpoint induction

context
fixes *luba* :: *'a set* ⇒ *'a*
and *orda* :: *'a* ⇒ *'a* ⇒ *bool*
and *lubb* :: *'b set* ⇒ *'b*
and *ordb* :: *'b* ⇒ *'b* ⇒ *bool*
assumes *a*: *class.ccpo luba orda* (*mk-less orda*)
and *b*: *class.ccpo lubb ordb* (*mk-less ordb*)
begin

interpretation *a*: *ccpo luba orda mk-less orda* ⟨*proof*⟩
interpretation *b*: *ccpo lubb ordb mk-less ordb* ⟨*proof*⟩

lemma *ccpo-rel-prodI*:
class.ccpo (*prod-lub luba lubb*) (*rel-prod orda ordb*) (*mk-less* (*rel-prod orda ordb*))
(**is** *class.ccpo ?lub ?ord ?ord'*)
⟨*proof*⟩

interpretation *ab*: *ccpo prod-lub luba lubb rel-prod orda ordb mk-less* (*rel-prod orda*

ordb)
⟨*proof*⟩

lemma *monotone-map-prod* [*simp*]:
monotone (*rel-prod orda ordb*) (*rel-prod ordc ordd*) (*map-prod f g*) ⟷
monotone orda ordc f ∧ *monotone ordb ordd g*
⟨*proof*⟩

lemma *parallel-fixp-induct*:
assumes *adm*: *ccpo.admissible* (*prod-lub luba lubb*) (*rel-prod orda ordb*) (λ*x*. *P* (*fst x*) (*snd x*))
and *f*: *monotone orda orda f*
and *g*: *monotone ordb ordb g*
and *bot*: *P* (*luba* {}) (*lubb* {})
and *step*: ⋀*x y*. *P x y* ⟹ *P* (*f x*) (*g y*)
shows *P* (*ccpo.fixp luba orda f*) (*ccpo.fixp lubb ordb g*)
⟨*proof*⟩

end

lemma *parallel-fixp-induct-uc*:
assumes *a*: *partial-function-definitions orda luba*
and *b*: *partial-function-definitions ordb lubb*
and *F*: ⋀*x*. *monotone* (*fun-ord orda*) *orda* (λ*f*. *U1* (*F* (*C1 f*)) *x*)
and *G*: ⋀*y*. *monotone* (*fun-ord ordb*) *ordb* (λ*g*. *U2* (*G* (*C2 g*)) *y*)
and *eq1*: *f* ≡ *C1* (*ccpo.fixp* (*fun-lub luba*) (*fun-ord orda*) (λ*f*. *U1* (*F* (*C1 f*))))
and *eq2*: *g* ≡ *C2* (*ccpo.fixp* (*fun-lub lubb*) (*fun-ord ordb*) (λ*g*. *U2* (*G* (*C2 g*))))
and *inverse*: ⋀*f*. *U1* (*C1 f*) = *f*
and *inverse2*: ⋀*g*. *U2* (*C2 g*) = *g*
and *adm*: *ccpo.admissible* (*prod-lub* (*fun-lub luba*) (*fun-lub lubb*)) (*rel-prod* (*fun-ord orda*) (*fun-ord ordb*)) (λ*x*. *P* (*fst x*) (*snd x*))
and *bot*: *P* (λ-. *luba* {}) (λ-. *lubb* {})
and *step*: ⋀*f g*. *P* (*U1 f*) (*U2 g*) ⟹ *P* (*U1* (*F f*)) (*U2* (*G g*))
shows *P* (*U1 f*) (*U2 g*)
⟨*proof*⟩

lemmas *parallel-fixp-induct-1-1* = *parallel-fixp-induct-uc*[
of - - - - λ*x*. *x* - λ*x*. *x* λ*x*. *x* - λ*x*. *x*,
OF - - - - - - *refl refl*]

lemmas *parallel-fixp-induct-2-2* = *parallel-fixp-induct-uc*[
of - - - - *case-prod* - *curry case-prod* - *curry*,
where *P*=λ*f g*. *P* (*curry f*) (*curry g*),
unfolded case-prod-curry curry-case-prod curry-K,
OF - - - - - - *refl refl*]
for *P*

lemma *monotone-fst*: *monotone* (*rel-prod orda ordb*) *orda fst*
⟨*proof*⟩

lemma *mcont-fst*: *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *luba orda fst*
⟨*proof*⟩

lemma *mcont2mcont-fst* [*cont-intro*, *simp*]:
mcont lub ord (*prod-lub luba lubb*) (*rel-prod orda ordb*) *t*
⟹ *mcont lub ord luba orda* (λ*x*. *fst* (*t x*))
⟨*proof*⟩

lemma *monotone-snd*: *monotone* (*rel-prod orda ordb*) *ordb snd*
⟨*proof*⟩

lemma *mcont-snd*: *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubb ordb snd*
⟨*proof*⟩

lemma *mcont2mcont-snd* [*cont-intro*, *simp*]:
mcont lub ord (*prod-lub luba lubb*) (*rel-prod orda ordb*) *t*
⟹ *mcont lub ord lubb ordb* (λ*x*. *snd* (*t x*))
⟨*proof*⟩

lemma *monotone-Pair*:
⟦ *monotone ord orda f*; *monotone ord ordb g* ⟧
⟹ *monotone ord* (*rel-prod orda ordb*) (λ*x*. (*f x*, *g x*))
⟨*proof*⟩

lemma *cont-Pair*:
⟦ *cont lub ord luba orda f*; *cont lub ord lubb ordb g* ⟧
⟹ *cont lub ord* (*prod-lub luba lubb*) (*rel-prod orda ordb*) (λ*x*. (*f x*, *g x*))
⟨*proof*⟩

lemma *mcont-Pair*:
⟦ *mcont lub ord luba orda f*; *mcont lub ord lubb ordb g* ⟧
⟹ *mcont lub ord* (*prod-lub luba lubb*) (*rel-prod orda ordb*) (λ*x*. (*f x*, *g x*))
⟨*proof*⟩

context *partial-function-definitions* **begin**

Specialised versions of *mcont-call* for admissibility proofs for parallel fixpoint inductions

lemmas *mcont-call-fst* [*cont-intro*] = *mcont-call*[*THEN mcont2mcont*, *OF mcont-fst*]
lemmas *mcont-call-snd* [*cont-intro*] = *mcont-call*[*THEN mcont2mcont*, *OF mcont-snd*]
end

lemma *map-option-mono* [*partial-function-mono*]:
mono-option B ⟹ *mono-option* (λ*f*. *map-option g* (*B f*))
⟨*proof*⟩

lemma *compact-flat-lub* [*cont-intro*]: *ccpo.compact* (*flat-lub x*) (*flat-ord x*) *y*
⟨*proof*⟩

end

15 Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums

theory *Old-Datatype*
imports *Main*
keywords *old-datatype* :: *thy-decl*
begin

⟨*ML*⟩

15.1 The datatype universe

definition *Node* = {*p*. *EX f x k*. *p* = (*f* :: *nat* => *'b* + *nat*, *x* ::*'a* + *nat*) & *f k* = *Inr 0*}

typedef (*'a*, *'b*) *node* = *Node* :: ((*nat* => *'b* + *nat*) * (*'a* + *nat*)) *set*
  morphisms *Rep-Node Abs-Node*
  ⟨*proof*⟩

Datatypes will be represented by sets of type *node*

type-synonym *'a item* = (*'a*, *unit*) *node set*
type-synonym (*'a*, *'b*) *dtree* = (*'a*, *'b*) *node set*

definition *Push* :: [(*'b* + *nat*), *nat* => (*'b* + *nat*)] => (*nat* => (*'b* + *nat*))

  where *Push* == (%*b h*. *case-nat b h*)

definition *Push-Node* :: [(*'b* + *nat*), (*'a*, *'b*) *node*] => (*'a*, *'b*) *node*
  where *Push-Node* == (%*n x*. *Abs-Node* (*apfst* (*Push n*) (*Rep-Node x*)))

definition *Atom* :: (*'a* + *nat*) => (*'a*, *'b*) *dtree*
  where *Atom* == (%*x*. {*Abs-Node*((%*k*. *Inr 0*, *x*))})
definition *Scons* :: [(*'a*, *'b*) *dtree*, (*'a*, *'b*) *dtree*] => (*'a*, *'b*) *dtree*
  where *Scons M N* == (*Push-Node* (*Inr 1*) ‘ *M*) *Un* (*Push-Node* (*Inr* (*Suc 1*)) ‘ *N*)

definition *Leaf* :: *'a* => (*'a*, *'b*) *dtree*
  where *Leaf* == *Atom o Inl*
definition *Numb* :: *nat* => (*'a*, *'b*) *dtree*
  where *Numb* == *Atom o Inr*

definition *In0* :: ('a, 'b) *dtree* => ('a, 'b) *dtree*
  where *In0*(*M*) == *Scons* (*Numb* 0) *M*
definition *In1* :: ('a, 'b) *dtree* => ('a, 'b) *dtree*
  where *In1*(*M*) == *Scons* (*Numb* 1) *M*

definition *Lim* :: ('b => ('a, 'b) *dtree*) => ('a, 'b) *dtree*
  where *Lim f* == $\bigcup${*z*. *?* *x*. *z* = *Push-Node* (*Inl x*) ‘ (*f x*)}

definition *ndepth* :: ('a, 'b) *node* => *nat*
  where *ndepth*(*n*) == (%(*f*,*x*). *LEAST k*. *f k* = *Inr* 0) (*Rep-Node n*)
definition *ntrunc* :: [*nat*, ('a, 'b) *dtree*] => ('a, 'b) *dtree*
  where *ntrunc k N* == {*n*. *n*:*N* & *ndepth*(*n*)<*k*}

definition *uprod* :: [('a, 'b) *dtree set*, ('a, 'b) *dtree set*]=> ('a, 'b) *dtree set*
  where *uprod A B* == *UN x*:*A*. *UN y*:*B*. { *Scons x y* }
definition *usum* :: [('a, 'b) *dtree set*, ('a, 'b) *dtree set*]=> ('a, 'b) *dtree set*
  where *usum A B* == *In0‘A Un In1‘B*

definition *Split* :: [[('a, 'b) *dtree*, ('a, 'b) *dtree*]=>'c, ('a, 'b) *dtree*] => 'c
  where *Split c M* == *THE u*. *EX x y*. *M* = *Scons x y* & *u* = *c x y*

definition *Case* :: [[('a, 'b) *dtree*]=>'c, [('a, 'b) *dtree*]=>'c, ('a, 'b) *dtree*] => 'c
  where *Case c d M* == *THE u*. (*EX x* . *M* = *In0*(*x*) & *u* = *c*(*x*)) | (*EX y* . *M* = *In1*(*y*) & *u* = *d*(*y*))

definition *dprod* :: [(('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*, (('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*]
    => (('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*
  where *dprod r s* == *UN* (*x*,*x'*):*r*. *UN* (*y*,*y'*):*s*. {(*Scons x y*, *Scons x' y'*)}

definition *dsum* :: [(('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*, (('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*]
    => (('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*
 where *dsum r s* == (*UN* (*x*,*x'*):*r*. {(*In0*(*x*),*In0*(*x'*))}) *Un* (*UN* (*y*,*y'*):*s*. {(*In1*(*y*),*In1*(*y'*))})

lemma *apfst-convE*:
    [| *q* = *apfst f p*; !!*x y*. [| *p* = (*x*,*y*); *q* = (*f*(*x*),*y*) |] ==> *R*
    |] ==> *R*
⟨*proof*⟩

lemma *Push-inject1*: *Push i f = Push j g ==> i=j*
⟨*proof*⟩

lemma *Push-inject2*: *Push i f = Push j g ==> f=g*
⟨*proof*⟩

lemma *Push-inject*:
[| *Push i f =Push j g*; [| *i=j*; *f=g* |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *Push-neq-K0*: *Push (Inr (Suc k)) f = (%z. Inr 0) ==> P*
⟨*proof*⟩

lemmas *Abs-Node-inj = Abs-Node-inject* [*THEN* [*2*] *rev-iffD1*]

lemma *Node-K0-I*: *(%k. Inr 0, a) : Node*
⟨*proof*⟩

lemma *Node-Push-I*: *p*: *Node ==> apfst (Push i) p : Node*
⟨*proof*⟩

15.2 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [*iff*]: *Scons M N ≠ Atom(a)*
⟨*proof*⟩

lemmas *Atom-not-Scons* [*iff*] = *Scons-not-Atom* [*THEN not-sym*]

lemma *inj-Atom*: *inj(Atom)*
⟨*proof*⟩
lemmas *Atom-inject = inj-Atom* [*THEN injD*]

lemma *Atom-Atom-eq* [*iff*]: *(Atom(a)=Atom(b)) = (a=b)*
⟨*proof*⟩

lemma *inj-Leaf*: *inj(Leaf)*
⟨*proof*⟩

lemmas *Leaf-inject* [*dest*!] = *inj-Leaf* [*THEN injD*]

lemma *inj-Numb*: *inj*(*Numb*)
⟨*proof*⟩

lemmas *Numb-inject* [*dest*!] = *inj-Numb* [*THEN injD*]

lemma *Push-Node-inject*:
[| *Push-Node i m* =*Push-Node j n*; [| *i*=*j*; *m*=*n* |] ==> *P*
|] ==> *P*
⟨*proof*⟩

lemma *Scons-inject-lemma1*: *Scons M N* <= *Scons M' N'* ==> *M*<=*M'*
⟨*proof*⟩

lemma *Scons-inject-lemma2*: *Scons M N* <= *Scons M' N'* ==> *N*<=*N'*
⟨*proof*⟩

lemma *Scons-inject1*: *Scons M N* = *Scons M' N'* ==> *M*=*M'*
⟨*proof*⟩

lemma *Scons-inject2*: *Scons M N* = *Scons M' N'* ==> *N*=*N'*
⟨*proof*⟩

lemma *Scons-inject*:
[| *Scons M N* = *Scons M' N'*; [| *M*=*M'*; *N*=*N'* |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *Scons-Scons-eq* [*iff*]: (*Scons M N* = *Scons M' N'*) = (*M*=*M'* & *N*=*N'*)
⟨*proof*⟩

lemma *Scons-not-Leaf* [*iff*]: *Scons M N* ≠ *Leaf*(*a*)
⟨*proof*⟩

lemmas *Leaf-not-Scons* [*iff*] = *Scons-not-Leaf* [*THEN not-sym*]

lemma *Scons-not-Numb* [*iff*]: *Scons M N* ≠ *Numb*(*k*)

⟨*proof*⟩

lemmas *Numb-not-Scons* [*iff*] = *Scons-not-Numb* [*THEN not-sym*]

lemma *Leaf-not-Numb* [*iff*]: *Leaf*(*a*) ≠ *Numb*(*k*)
⟨*proof*⟩

lemmas *Numb-not-Leaf* [*iff*] = *Leaf-not-Numb* [*THEN not-sym*]

lemma *ndepth-K0*: *ndepth* (*Abs-Node*(%*k*. *Inr* 0, *x*)) = 0
⟨*proof*⟩

lemma *ndepth-Push-Node-aux*:
case-nat (*Inr* (*Suc i*)) *f k* = *Inr* 0 −−> *Suc*(*LEAST x*. *f x* = *Inr* 0) <= *k*
⟨*proof*⟩

lemma *ndepth-Push-Node*:
ndepth (*Push-Node* (*Inr* (*Suc i*)) *n*) = *Suc*(*ndepth*(*n*))
⟨*proof*⟩

lemma *ntrunc-0* [*simp*]: *ntrunc* 0 *M* = {}
⟨*proof*⟩

lemma *ntrunc-Atom* [*simp*]: *ntrunc* (*Suc k*) (*Atom a*) = *Atom*(*a*)
⟨*proof*⟩

lemma *ntrunc-Leaf* [*simp*]: *ntrunc* (*Suc k*) (*Leaf a*) = *Leaf*(*a*)
⟨*proof*⟩

lemma *ntrunc-Numb* [*simp*]: *ntrunc* (*Suc k*) (*Numb i*) = *Numb*(*i*)
⟨*proof*⟩

lemma *ntrunc-Scons* [*simp*]:
ntrunc (*Suc k*) (*Scons M N*) = *Scons* (*ntrunc k M*) (*ntrunc k N*)
⟨*proof*⟩

lemma *ntrunc-one-In0* [*simp*]: *ntrunc* (*Suc 0*) (*In0 M*) = {}
⟨*proof*⟩

lemma *ntrunc-In0* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In0 M*) = *In0* (*ntrunc* (*Suc k*) *M*)
⟨*proof*⟩

lemma *ntrunc-one-In1* [*simp*]: *ntrunc* (*Suc 0*) (*In1 M*) = {}
⟨*proof*⟩

lemma *ntrunc-In1* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In1 M*) = *In1* (*ntrunc* (*Suc k*) *M*)
⟨*proof*⟩

15.3 Set Constructions

lemma *uprodI* [*intro*!]: [| *M*:*A*; *N*:*B* |] ==> *Scons M N* : *uprod A B*
⟨*proof*⟩

lemma *uprodE* [*elim*!]:
[| *c* : *uprod A B*;
!!*x y*. [| *x*:*A*; *y*:*B*; *c* = *Scons x y* |] ==> *P*
|] ==> *P*
⟨*proof*⟩

lemma *uprodE2*: [| *Scons M N* : *uprod A B*; [| *M*:*A*; *N*:*B* |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *usum-In0I* [*intro*]: *M*:*A* ==> *In0*(*M*) : *usum A B*
⟨*proof*⟩

lemma *usum-In1I* [*intro*]: *N*:*B* ==> *In1*(*N*) : *usum A B*
⟨*proof*⟩

lemma *usumE* [*elim*!]:
[| *u* : *usum A B*;
!!*x*. [| *x*:*A*; *u*=*In0*(*x*) |] ==> *P*;
!!*y*. [| *y*:*B*; *u*=*In1*(*y*) |] ==> *P*
|] ==> *P*
⟨*proof*⟩

lemma *In0-not-In1* [*iff*]: $In0(M) \neq In1(N)$
⟨*proof*⟩

lemmas *In1-not-In0* [*iff*] = *In0-not-In1* [*THEN not-sym*]

lemma *In0-inject*: $In0(M) = In0(N) \Longrightarrow M=N$
⟨*proof*⟩

lemma *In1-inject*: $In1(M) = In1(N) \Longrightarrow M=N$
⟨*proof*⟩

lemma *In0-eq* [*iff*]: $(In0\ M = In0\ N) = (M=N)$
⟨*proof*⟩

lemma *In1-eq* [*iff*]: $(In1\ M = In1\ N) = (M=N)$
⟨*proof*⟩

lemma *inj-In0*: *inj In0*
⟨*proof*⟩

lemma *inj-In1*: *inj In1*
⟨*proof*⟩

lemma *Lim-inject*: $Lim\ f = Lim\ g \Longrightarrow f = g$
⟨*proof*⟩

lemma *ntrunc-subsetI*: $ntrunc\ k\ M <= M$
⟨*proof*⟩

lemma *ntrunc-subsetD*: $(!!k.\ ntrunc\ k\ M <= N) \Longrightarrow M<=N$
⟨*proof*⟩

lemma *ntrunc-equality*: $(!!k.\ ntrunc\ k\ M = ntrunc\ k\ N) \Longrightarrow M=N$
⟨*proof*⟩

lemma *ntrunc-o-equality*:
$[|\ !!k.\ (ntrunc(k)\ o\ h1) = (ntrunc(k)\ o\ h2)\ |] \Longrightarrow h1=h2$
⟨*proof*⟩

lemma *uprod-mono*: [| $A <= A'$; $B <= B'$ |] ==> *uprod* A B <= *uprod* A' B'
⟨*proof*⟩

lemma *usum-mono*: [| $A <= A'$; $B <= B'$ |] ==> *usum* A B <= *usum* A' B'
⟨*proof*⟩

lemma *Scons-mono*: [| $M <= M'$; $N <= N'$ |] ==> *Scons* M N <= *Scons* M' N'
⟨*proof*⟩

lemma *In0-mono*: $M <= N$ ==> *In0*(M) <= *In0*(N)
⟨*proof*⟩

lemma *In1-mono*: $M <= N$ ==> *In1*(M) <= *In1*(N)
⟨*proof*⟩

lemma *Split* [*simp*]: *Split* c (*Scons* M N) = c M N
⟨*proof*⟩

lemma *Case-In0* [*simp*]: *Case* c d (*In0* M) = c(M)
⟨*proof*⟩

lemma *Case-In1* [*simp*]: *Case* c d (*In1* N) = d(N)
⟨*proof*⟩

lemma *ntrunc-UN1*: *ntrunc* k (*UN* x. f(x)) = (*UN* x. *ntrunc* k (f x))
⟨*proof*⟩

lemma *Scons-UN1-x*: *Scons* (*UN* x. f x) M = (*UN* x. *Scons* (f x) M)
⟨*proof*⟩

lemma *Scons-UN1-y*: *Scons* M (*UN* x. f x) = (*UN* x. *Scons* M (f x))
⟨*proof*⟩

lemma *In0-UN1*: *In0*(*UN* x. f(x)) = (*UN* x. *In0*(f(x)))
⟨*proof*⟩

lemma *In1-UN1*: *In1*(*UN* x. f(x)) = (*UN* x. *In1*(f(x)))
⟨*proof*⟩

lemma *dprodI* [*intro*!]:
[| (*M*,*M'*):*r*; (*N*,*N'*):*s* |] ==> (*Scons M N*, *Scons M' N'*) : *dprod r s*
⟨*proof*⟩

lemma *dprodE* [*elim*!]:
[| *c* : *dprod r s*;
!!*x y x' y'*. [| (*x*,*x'*) : *r*; (*y*,*y'*) : *s*;
c = (*Scons x y*, *Scons x' y'*) |] ==> *P*
|] ==> *P*
⟨*proof*⟩

lemma *dsum-In0I* [*intro*]: (*M*,*M'*):*r* ==> (*In0*(*M*), *In0*(*M'*)) : *dsum r s*
⟨*proof*⟩

lemma *dsum-In1I* [*intro*]: (*N*,*N'*):*s* ==> (*In1*(*N*), *In1*(*N'*)) : *dsum r s*
⟨*proof*⟩

lemma *dsumE* [*elim*!]:
[| *w* : *dsum r s*;
!!*x x'*. [| (*x*,*x'*) : *r*; *w* = (*In0*(*x*), *In0*(*x'*)) |] ==> *P*;
!!*y y'*. [| (*y*,*y'*) : *s*; *w* = (*In1*(*y*), *In1*(*y'*)) |] ==> *P*
|] ==> *P*
⟨*proof*⟩

lemma *dprod-mono*: [| *r*<=*r'*; *s*<=*s'* |] ==> *dprod r s* <= *dprod r' s'*
⟨*proof*⟩

lemma *dsum-mono*: [| *r*<=*r'*; *s*<=*s'* |] ==> *dsum r s* <= *dsum r' s'*
⟨*proof*⟩

lemma *dprod-Sigma*: (*dprod* (*A* × *B*) (*C* × *D*)) <= (*uprod A C*) × (*uprod B D*)
⟨*proof*⟩

lemmas *dprod-subset-Sigma* = *subset-trans* [*OF dprod-mono dprod-Sigma*]

lemma *dprod-subset-Sigma2*:
(*dprod* (*Sigma A B*) (*Sigma C D*)) <= *Sigma* (*uprod A C*) (*Split* (%*x y*. *uprod*

(*B x*) (*D y*)))
⟨*proof*⟩

lemma *dsum-Sigma*: (*dsum* (*A* × *B*) (*C* × *D*)) <= (*usum A C*) × (*usum B D*)
⟨*proof*⟩

lemmas *dsum-subset-Sigma* = *subset-trans* [*OF dsum-mono dsum-Sigma*]

lemma *Domain-dprod* [*simp*]: *Domain* (*dprod r s*) = *uprod* (*Domain r*) (*Domain s*)
⟨*proof*⟩

lemma *Domain-dsum* [*simp*]: *Domain* (*dsum r s*) = *usum* (*Domain r*) (*Domain s*)
⟨*proof*⟩

hides popular names

hide-type (**open**) *node item*
hide-const (**open**) *Push Node Atom Leaf Numb Lim Split Case*

⟨*ML*⟩

end

16 Bijections between natural numbers and other types

theory *Nat-Bijection*
imports *Main*
begin

16.1 Type *nat* × *nat*

Triangle numbers: 0, 1, 3, 6, 10, 15, ...

definition *triangle* :: *nat* ⇒ *nat*
where *triangle n* = (*n* ∗ *Suc n*) *div 2*

lemma *triangle-0* [*simp*]: *triangle 0* = *0*
⟨*proof*⟩

lemma *triangle-Suc* [*simp*]: *triangle* (*Suc n*) = *triangle n* + *Suc n*
⟨*proof*⟩

definition *prod-encode* :: *nat* × *nat* ⇒ *nat*
where *prod-encode* = (λ(*m*, *n*). *triangle* (*m* + *n*) + *m*)

In this auxiliary function, *triangle k + m* is an invariant.

fun *prod-decode-aux* :: *nat* ⇒ *nat* ⇒ *nat* × *nat*
where *prod-decode-aux k m* =
(*if m* ≤ *k then* (*m*, *k* − *m*) *else prod-decode-aux* (*Suc k*) (*m* − *Suc k*))

declare *prod-decode-aux.simps* [*simp del*]

definition *prod-decode* :: *nat* ⇒ *nat* × *nat*
where *prod-decode* = *prod-decode-aux 0*

lemma *prod-encode-prod-decode-aux*: *prod-encode* (*prod-decode-aux k m*) = *triangle k* + *m*
⟨*proof*⟩

lemma *prod-decode-inverse* [*simp*]: *prod-encode* (*prod-decode n*) = *n*
⟨*proof*⟩

lemma *prod-decode-triangle-add*: *prod-decode* (*triangle k* + *m*) = *prod-decode-aux k m*
⟨*proof*⟩

lemma *prod-encode-inverse* [*simp*]: *prod-decode* (*prod-encode x*) = *x*
⟨*proof*⟩

lemma *inj-prod-encode*: *inj-on prod-encode A*
⟨*proof*⟩

lemma *inj-prod-decode*: *inj-on prod-decode A*
⟨*proof*⟩

lemma *surj-prod-encode*: *surj prod-encode*
⟨*proof*⟩

lemma *surj-prod-decode*: *surj prod-decode*
⟨*proof*⟩

lemma *bij-prod-encode*: *bij prod-encode*
⟨*proof*⟩

lemma *bij-prod-decode*: *bij prod-decode*
⟨*proof*⟩

lemma *prod-encode-eq*: *prod-encode x* = *prod-encode y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *prod-decode-eq*: *prod-decode x* = *prod-decode y* ⟷ *x* = *y*
⟨*proof*⟩

Ordering properties

lemma *le-prod-encode-1*: $a \leq$ *prod-encode* (a, b)
⟨*proof*⟩

lemma *le-prod-encode-2*: $b \leq$ *prod-encode* (a, b)
⟨*proof*⟩

16.2 Type *nat* + *nat*

definition *sum-encode* :: *nat* + *nat* ⇒ *nat*
where *sum-encode x* = (*case x of Inl a* ⇒ *2* ∗ *a* | *Inr b* ⇒ *Suc* (*2* ∗ *b*))

definition *sum-decode* :: *nat* ⇒ *nat* + *nat*
where *sum-decode n* = (*if even n then Inl* (*n div 2*) *else Inr* (*n div 2*))

lemma *sum-encode-inverse* [*simp*]: *sum-decode* (*sum-encode x*) = *x*
⟨*proof*⟩

lemma *sum-decode-inverse* [*simp*]: *sum-encode* (*sum-decode n*) = *n*
⟨*proof*⟩

lemma *inj-sum-encode*: *inj-on sum-encode A*
⟨*proof*⟩

lemma *inj-sum-decode*: *inj-on sum-decode A*
⟨*proof*⟩

lemma *surj-sum-encode*: *surj sum-encode*
⟨*proof*⟩

lemma *surj-sum-decode*: *surj sum-decode*
⟨*proof*⟩

lemma *bij-sum-encode*: *bij sum-encode*
⟨*proof*⟩

lemma *bij-sum-decode*: *bij sum-decode*
⟨*proof*⟩

lemma *sum-encode-eq*: *sum-encode x* = *sum-encode y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *sum-decode-eq*: *sum-decode x* = *sum-decode y* ⟷ *x* = *y*
⟨*proof*⟩

16.3 Type *int*

definition *int-encode* :: *int* ⇒ *nat*
where *int-encode i* = *sum-encode* (*if 0* ≤ *i then Inl* (*nat i*) *else Inr* (*nat* (− *i* − *1*)))

definition *int-decode* :: *nat* ⇒ *int*
where *int-decode n* = (*case sum-decode n of Inl a* ⇒ *int a* | *Inr b* ⇒ − *int b* − *1*)

lemma *int-encode-inverse* [*simp*]: *int-decode* (*int-encode x*) = *x*
⟨*proof*⟩

lemma *int-decode-inverse* [*simp*]: *int-encode* (*int-decode n*) = *n*
⟨*proof*⟩

lemma *inj-int-encode*: *inj-on int-encode A*
⟨*proof*⟩

lemma *inj-int-decode*: *inj-on int-decode A*
⟨*proof*⟩

lemma *surj-int-encode*: *surj int-encode*
⟨*proof*⟩

lemma *surj-int-decode*: *surj int-decode*
⟨*proof*⟩

lemma *bij-int-encode*: *bij int-encode*
⟨*proof*⟩

lemma *bij-int-decode*: *bij int-decode*
⟨*proof*⟩

lemma *int-encode-eq*: *int-encode x* = *int-encode y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *int-decode-eq*: *int-decode x* = *int-decode y* ⟷ *x* = *y*
⟨*proof*⟩

16.4 Type *nat list*

fun *list-encode* :: *nat list* ⇒ *nat*
where
list-encode [] = *0*
| *list-encode* (*x* # *xs*) = *Suc* (*prod-encode* (*x*, *list-encode xs*))

function *list-decode* :: *nat* ⇒ *nat list*
where
list-decode 0 = []
| *list-decode* (*Suc n*) = (*case prod-decode n of* (*x*, *y*) ⇒ *x* # *list-decode y*)
⟨*proof*⟩

termination *list-decode*
⟨*proof*⟩

lemma *list-encode-inverse* [*simp*]: *list-decode* (*list-encode x*) = *x*
⟨*proof*⟩

lemma *list-decode-inverse* [*simp*]: *list-encode* (*list-decode n*) = *n*
⟨*proof*⟩

lemma *inj-list-encode*: *inj-on list-encode A*
⟨*proof*⟩

lemma *inj-list-decode*: *inj-on list-decode A*
⟨*proof*⟩

lemma *surj-list-encode*: *surj list-encode*
⟨*proof*⟩

lemma *surj-list-decode*: *surj list-decode*
⟨*proof*⟩

lemma *bij-list-encode*: *bij list-encode*
⟨*proof*⟩

lemma *bij-list-decode*: *bij list-decode*
⟨*proof*⟩

lemma *list-encode-eq*: *list-encode x* = *list-encode y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *list-decode-eq*: *list-decode x* = *list-decode y* ⟷ *x* = *y*
⟨*proof*⟩

16.5 Finite sets of naturals

16.5.1 Preliminaries

lemma *finite-vimage-Suc-iff*: *finite* (*Suc* −‘ *F*) ⟷ *finite F*
⟨*proof*⟩

lemma *vimage-Suc-insert-0*: *Suc* −‘ *insert 0 A* = *Suc* −‘ *A*
⟨*proof*⟩

lemma *vimage-Suc-insert-Suc*: *Suc* −‘ *insert* (*Suc n*) *A* = *insert n* (*Suc* −‘ *A*)
⟨*proof*⟩

lemma *div2-even-ext-nat*:
fixes *x y* :: *nat*
assumes *x div 2* = *y div 2*
and *even x* ⟷ *even y*
shows *x* = *y*
⟨*proof*⟩

16.5.2 From sets to naturals

definition *set-encode* :: *nat set* ⇒ *nat*
where *set-encode* = *sum* (*op* ^ *2*)

lemma *set-encode-empty* [*simp*]: *set-encode* {} = *0*
⟨*proof*⟩

lemma *set-encode-inf*: ¬ *finite A* ⟹ *set-encode A* = *0*
⟨*proof*⟩

lemma *set-encode-insert* [*simp*]: *finite A* ⟹ *n* ∉ *A* ⟹ *set-encode* (*insert n A*) = *2*^*n* + *set-encode A*
⟨*proof*⟩

lemma *even-set-encode-iff*: *finite A* ⟹ *even* (*set-encode A*) ⟷ *0* ∉ *A*
⟨*proof*⟩

lemma *set-encode-vimage-Suc*: *set-encode* (*Suc* −‘ *A*) = *set-encode A div 2*
⟨*proof*⟩

lemmas *set-encode-div-2* = *set-encode-vimage-Suc* [*symmetric*]

16.5.3 From naturals to sets

definition *set-decode* :: *nat* ⇒ *nat set*
where *set-decode x* = {*n*. *odd* (*x div 2* ^ *n*)}

lemma *set-decode-0* [*simp*]: *0* ∈ *set-decode x* ⟷ *odd x*
⟨*proof*⟩

lemma *set-decode-Suc* [*simp*]: *Suc n* ∈ *set-decode x* ⟷ *n* ∈ *set-decode* (*x div 2*)
⟨*proof*⟩

lemma *set-decode-zero* [*simp*]: *set-decode 0* = {}
⟨*proof*⟩

lemma *set-decode-div-2*: *set-decode* (*x div 2*) = *Suc* −‘ *set-decode x*
⟨*proof*⟩

lemma *set-decode-plus-power-2*:
n ∉ *set-decode z* ⟹ *set-decode* (*2* ^ *n* + *z*) = *insert n* (*set-decode z*)
⟨*proof*⟩

lemma *finite-set-decode* [*simp*]: *finite* (*set-decode n*)
⟨*proof*⟩

16.5.4 Proof of isomorphism

lemma *set-decode-inverse* [*simp*]: *set-encode* (*set-decode n*) = *n*

⟨*proof*⟩

lemma *set-encode-inverse* [*simp*]: *finite A* ⟹ *set-decode* (*set-encode A*) = *A*
⟨*proof*⟩

lemma *inj-on-set-encode*: *inj-on set-encode* (*Collect finite*)
⟨*proof*⟩

lemma *set-encode-eq*: *finite A* ⟹ *finite B* ⟹ *set-encode A* = *set-encode B* ⟷ *A* = *B*
⟨*proof*⟩

lemma *subset-decode-imp-le*:
assumes *set-decode m* ⊆ *set-decode n*
shows *m* ≤ *n*
⟨*proof*⟩

end

17 Encoding (almost) everything into natural numbers

theory *Countable*
imports *Old-Datatype HOL.Rat Nat-Bijection*
begin

17.1 The class of countable types

class *countable* =
assumes *ex-inj*: ∃ *to-nat* :: *'a* ⇒ *nat*. *inj to-nat*

lemma *countable-classI*:
fixes *f* :: *'a* ⇒ *nat*
assumes ⋀*x y*. *f x* = *f y* ⟹ *x* = *y*
shows *OFCLASS*(*'a*, *countable-class*)
⟨*proof*⟩

17.2 Conversion functions

definition *to-nat* :: *'a*::*countable* ⇒ *nat* **where**
to-nat = (*SOME f*. *inj f*)

definition *from-nat* :: *nat* ⇒ *'a*::*countable* **where**
from-nat = *inv* (*to-nat* :: *'a* ⇒ *nat*)

lemma *inj-to-nat* [*simp*]: *inj to-nat*
⟨*proof*⟩

lemma *inj-on-to-nat*[*simp*, *intro*]: *inj-on to-nat S*

⟨*proof*⟩

lemma *surj-from-nat* [*simp*]: *surj from-nat*
⟨*proof*⟩

lemma *to-nat-split* [*simp*]: *to-nat x* = *to-nat y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *from-nat-to-nat* [*simp*]:
from-nat (*to-nat x*) = *x*
⟨*proof*⟩

17.3 Finite types are countable

subclass (**in** *finite*) *countable*
⟨*proof*⟩

17.4 Automatically proving countability of old-style datatypes

context
begin

qualified inductive *finite-item* :: *'a Old-Datatype.item* ⇒ *bool* **where**
undefined: *finite-item undefined*
| *In0*: *finite-item x* ⟹ *finite-item* (*Old-Datatype.In0 x*)
| *In1*: *finite-item x* ⟹ *finite-item* (*Old-Datatype.In1 x*)
| *Leaf*: *finite-item* (*Old-Datatype.Leaf a*)
| *Scons*: ⟦*finite-item x*; *finite-item y*⟧ ⟹ *finite-item* (*Old-Datatype.Scons x y*)

qualified function *nth-item* :: *nat* ⇒ (*'a*::*countable*) *Old-Datatype.item*
where
nth-item 0 = *undefined*
| *nth-item* (*Suc n*) =
(*case sum-decode n of*
Inl i ⇒
(*case sum-decode i of*
Inl j ⇒ *Old-Datatype.In0* (*nth-item j*)
| *Inr j* ⇒ *Old-Datatype.In1* (*nth-item j*))
| *Inr i* ⇒
(*case sum-decode i of*
Inl j ⇒ *Old-Datatype.Leaf* (*from-nat j*)
| *Inr j* ⇒
(*case prod-decode j of*
(*a*, *b*) ⇒ *Old-Datatype.Scons* (*nth-item a*) (*nth-item b*))))
⟨*proof*⟩

lemma *le-sum-encode-Inl*: *x* ≤ *y* ⟹ *x* ≤ *sum-encode* (*Inl y*)
⟨*proof*⟩

lemma *le-sum-encode-Inr*: *x* ≤ *y* ⟹ *x* ≤ *sum-encode* (*Inr y*)

⟨*proof*⟩ **termination**
⟨*proof*⟩

lemma *nth-item-covers*: *finite-item x* ⟹ ∃ *n*. *nth-item n* = *x*
⟨*proof*⟩

theorem *countable-datatype*:
fixes *Rep* :: *'b* ⇒ (*'a*::*countable*) *Old-Datatype.item*
fixes *Abs* :: (*'a*::*countable*) *Old-Datatype.item* ⇒ *'b*
fixes *rep-set* :: (*'a*::*countable*) *Old-Datatype.item* ⇒ *bool*
assumes *type*: *type-definition Rep Abs* (*Collect rep-set*)
assumes *finite-item*: ⋀*x*. *rep-set x* ⟹ *finite-item x*
shows *OFCLASS*(*'b*, *countable-class*)
⟨*proof*⟩

⟨*ML*⟩

end

17.5 Automatically proving countability of datatypes

⟨*ML*⟩

17.6 More Countable types

Naturals

instance *nat* :: *countable*
⟨*proof*⟩

Pairs

instance *prod* :: (*countable*, *countable*) *countable*
⟨*proof*⟩

Sums

instance *sum* :: (*countable*, *countable*) *countable*
⟨*proof*⟩

Integers

instance *int* :: *countable*
⟨*proof*⟩

Options

instance *option* :: (*countable*) *countable*
⟨*proof*⟩

Lists

instance *list* :: (*countable*) *countable*
⟨*proof*⟩

String literals

instance *String.literal* :: *countable*
⟨*proof*⟩

Functions

instance *fun* :: (*finite*, *countable*) *countable*
⟨*proof*⟩

Typereps

instance *typerep* :: *countable*
⟨*proof*⟩

17.7 The rationals are countably infinite

definition *nat-to-rat-surj* :: *nat* ⇒ *rat* **where**
nat-to-rat-surj n = (*let* (*a*, *b*) = *prod-decode n in Fract* (*int-decode a*) (*int-decode b*))

lemma *surj-nat-to-rat-surj*: *surj nat-to-rat-surj*
⟨*proof*⟩

lemma *Rats-eq-range-nat-to-rat-surj*: ℚ = *range nat-to-rat-surj*
⟨*proof*⟩

context *field-char-0*
begin

lemma *Rats-eq-range-of-rat-o-nat-to-rat-surj*:
ℚ = *range* (*of-rat* ∘ *nat-to-rat-surj*)
⟨*proof*⟩

lemma *surj-of-rat-nat-to-rat-surj*:
r ∈ ℚ ⟹ ∃ *n*. *r* = *of-rat* (*nat-to-rat-surj n*)
⟨*proof*⟩

end

instance *rat* :: *countable*
⟨*proof*⟩

end

18 Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *Main*
begin

18.1 The set of natural numbers is infinite

lemma *infinite-nat-iff-unbounded-le*: *infinite* $S \longleftrightarrow (\forall\, m.\ \exists\, n{\geq}m.\ n \in S)$
for S :: *nat set*
⟨*proof*⟩

lemma *infinite-nat-iff-unbounded*: *infinite* $S \longleftrightarrow (\forall\, m.\ \exists\, n{>}m.\ n \in S)$
for S :: *nat set*
⟨*proof*⟩

lemma *finite-nat-iff-bounded*: *finite* $S \longleftrightarrow (\exists\, k.\ S \subseteq \{..{<}k\})$
for S :: *nat set*
⟨*proof*⟩

lemma *finite-nat-iff-bounded-le*: *finite* $S \longleftrightarrow (\exists\, k.\ S \subseteq \{..\ k\})$
for S :: *nat set*
⟨*proof*⟩

lemma *finite-nat-bounded*: *finite* $S \Longrightarrow \exists\, k.\ S \subseteq \{..{<}k\}$
for S :: *nat set*
⟨*proof*⟩

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some k, there is some larger number that is an element of the set.

lemma *unbounded-k-infinite*: $\forall\, m{>}k.\ \exists\, n{>}m.\ n \in S \Longrightarrow$ *infinite* $(S{::}$*nat set*$)$
⟨*proof*⟩

lemma *nat-not-finite*: *finite* $(UNIV{::}$*nat set*$) \Longrightarrow R$
⟨*proof*⟩

lemma *range-inj-infinite*:
fixes f :: *nat* $\Rightarrow$ *'a*
assumes *inj f*
shows *infinite* (*range f*)
⟨*proof*⟩

18.2 The set of integers is also infinite

lemma *infinite-int-iff-infinite-nat-abs*: *infinite* $S \longleftrightarrow$ *infinite* $((nat \circ abs)\ `\ S)$
for S :: *int set*
⟨*proof*⟩

proposition *infinite-int-iff-unbounded-le*: *infinite* $S \longleftrightarrow (\forall\, m.\ \exists\, n.\ |n| \geq m \wedge n \in S)$
for S :: *int set*
⟨*proof*⟩

proposition *infinite-int-iff-unbounded*: *infinite* $S \longleftrightarrow (\forall\, m.\ \exists\, n.\ |n| > m \wedge n \in S)$

for S :: *int set*
⟨*proof*⟩

proposition *finite-int-iff-bounded*: *finite* $S \longleftrightarrow (\exists k.\ abs\ `\ S \subseteq \{..<k\})$
for S :: *int set*
⟨*proof*⟩

proposition *finite-int-iff-bounded-le*: *finite* $S \longleftrightarrow (\exists k.\ abs\ `\ S \subseteq \{..\ k\})$
for S :: *int set*
⟨*proof*⟩

18.3 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

lemma *not-INFM* [*simp*]: $\neg\ (INFM\ x.\ P\ x) \longleftrightarrow (MOST\ x.\ \neg\ P\ x)$
⟨*proof*⟩

lemma *not-MOST* [*simp*]: $\neg\ (MOST\ x.\ P\ x) \longleftrightarrow (INFM\ x.\ \neg\ P\ x)$
⟨*proof*⟩

lemma *INFM-const* [*simp*]: $(INFM\ x{::}'a.\ P) \longleftrightarrow P \wedge infinite\ (UNIV{::}'a\ set)$
⟨*proof*⟩

lemma *MOST-const* [*simp*]: $(MOST\ x{::}'a.\ P) \longleftrightarrow P \vee finite\ (UNIV{::}'a\ set)$
⟨*proof*⟩

lemma *INFM-imp-distrib*: $(INFM\ x.\ P\ x \longrightarrow Q\ x) \longleftrightarrow ((MOST\ x.\ P\ x) \longrightarrow (INFM\ x.\ Q\ x))$
⟨*proof*⟩

lemma *MOST-imp-iff*: $MOST\ x.\ P\ x \Longrightarrow (MOST\ x.\ P\ x \longrightarrow Q\ x) \longleftrightarrow (MOST\ x.\ Q\ x)$
⟨*proof*⟩

lemma *INFM-conjI*: $INFM\ x.\ P\ x \Longrightarrow MOST\ x.\ Q\ x \Longrightarrow INFM\ x.\ P\ x \wedge Q\ x$
⟨*proof*⟩

Properties of quantifiers with injective functions.

lemma *INFM-inj*: $INFM\ x.\ P\ (f\ x) \Longrightarrow inj\ f \Longrightarrow INFM\ x.\ P\ x$
⟨*proof*⟩

lemma *MOST-inj*: $MOST\ x.\ P\ x \Longrightarrow inj\ f \Longrightarrow MOST\ x.\ P\ (f\ x)$
⟨*proof*⟩

Properties of quantifiers with singletons.

lemma *not-INFM-eq* [*simp*]:
$\neg\ (INFM\ x.\ x = a)$

$\neg$ (*INFM* x. $a = x$)
⟨*proof*⟩

lemma *MOST-neq* [*simp*]:
MOST x. $x \neq a$
MOST x. $a \neq x$
⟨*proof*⟩

lemma *INFM-neq* [*simp*]:
(*INFM* x::'a. $x \neq a$) $\longleftrightarrow$ *infinite* (*UNIV*::'a *set*)
(*INFM* x::'a. $a \neq x$) $\longleftrightarrow$ *infinite* (*UNIV*::'a *set*)
⟨*proof*⟩

lemma *MOST-eq* [*simp*]:
(*MOST* x::'a. $x = a$) $\longleftrightarrow$ *finite* (*UNIV*::'a *set*)
(*MOST* x::'a. $a = x$) $\longleftrightarrow$ *finite* (*UNIV*::'a *set*)
⟨*proof*⟩

lemma *MOST-eq-imp*:
MOST x. $x = a \longrightarrow P\ x$
MOST x. $a = x \longrightarrow P\ x$
⟨*proof*⟩

Properties of quantifiers over the naturals.

lemma *MOST-nat*: $(\forall_\infty n.\ P\ n) \longleftrightarrow (\exists\, m.\ \forall\, n{>}m.\ P\ n)$
for P :: *nat* $\Rightarrow$ *bool*
⟨*proof*⟩

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ n) \longleftrightarrow (\exists\, m.\ \forall\, n{\geq}m.\ P\ n)$
for P :: *nat* $\Rightarrow$ *bool*
⟨*proof*⟩

lemma *INFM-nat*: $(\exists_\infty n.\ P\ n) \longleftrightarrow (\forall\, m.\ \exists\, n{>}m.\ P\ n)$
for P :: *nat* $\Rightarrow$ *bool*
⟨*proof*⟩

lemma *INFM-nat-le*: $(\exists_\infty n.\ P\ n) \longleftrightarrow (\forall\, m.\ \exists\, n{\geq}m.\ P\ n)$
for P :: *nat* $\Rightarrow$ *bool*
⟨*proof*⟩

lemma *MOST-INFM*: *infinite* (*UNIV*::'a *set*) $\Longrightarrow$ *MOST* x::'a. $P\ x \Longrightarrow$ *INFM* x::'a. $P\ x$
⟨*proof*⟩

lemma *MOST-Suc-iff*: (*MOST* n. P (*Suc* n)) $\longleftrightarrow$ (*MOST* n. $P\ n$)
⟨*proof*⟩

lemma *MOST-SucI*: *MOST* n. $P\ n \Longrightarrow$ *MOST* n. P (*Suc* n)
and *MOST-SucD*: *MOST* n. P (*Suc* n) $\Longrightarrow$ *MOST* n. $P\ n$

⟨*proof*⟩

lemma *MOST-ge-nat*: *MOST n::nat. m* ≤ *n*
⟨*proof*⟩

lemma *Inf-many-def*: *Inf-many P* ⟷ *infinite* {*x. P x*} ⟨*proof*⟩
lemma *Alm-all-def*: *Alm-all P* ⟷ ¬ (*INFM x.* ¬ *P x*) ⟨*proof*⟩
lemma *INFM-iff-infinite*: (*INFM x. P x*) ⟷ *infinite* {*x. P x*} ⟨*proof*⟩
lemma *MOST-iff-cofinite*: (*MOST x. P x*) ⟷ *finite* {*x.* ¬ *P x*} ⟨*proof*⟩
lemma *INFM-EX*: (∃$_\infty$*x. P x*) ⟹ (∃*x. P x*) ⟨*proof*⟩
lemma *ALL-MOST*: ∀*x. P x* ⟹ ∀$_\infty$*x. P x* ⟨*proof*⟩
lemma *INFM-mono*: ∃$_\infty$*x. P x* ⟹ (⋀*x. P x* ⟹ *Q x*) ⟹ ∃$_\infty$*x. Q x* ⟨*proof*⟩
lemma *MOST-mono*: ∀$_\infty$*x. P x* ⟹ (⋀*x. P x* ⟹ *Q x*) ⟹ ∀$_\infty$*x. Q x* ⟨*proof*⟩
lemma *INFM-disj-distrib*: (∃$_\infty$*x. P x* ∨ *Q x*) ⟷ (∃$_\infty$*x. P x*) ∨ (∃$_\infty$*x. Q x*)
⟨*proof*⟩
lemma *MOST-rev-mp*: ∀$_\infty$*x. P x* ⟹ ∀$_\infty$*x. P x* ⟶ *Q x* ⟹ ∀$_\infty$*x. Q x* ⟨*proof*⟩
lemma *MOST-conj-distrib*: (∀$_\infty$*x. P x* ∧ *Q x*) ⟷ (∀$_\infty$*x. P x*) ∧ (∀$_\infty$*x. Q x*)
⟨*proof*⟩
lemma *MOST-conjI*: *MOST x. P x* ⟹ *MOST x. Q x* ⟹ *MOST x. P x* ∧ *Q x*
⟨*proof*⟩
lemma *INFM-finite-Bex-distrib*: *finite A* ⟹ (*INFM y.* ∃*x*∈*A. P x y*) ⟷ (∃*x*∈*A. INFM y. P x y*) ⟨*proof*⟩
lemma *MOST-finite-Ball-distrib*: *finite A* ⟹ (*MOST y.* ∀*x*∈*A. P x y*) ⟷ (∀*x*∈*A. MOST y. P x y*) ⟨*proof*⟩
lemma *INFM-E*: *INFM x. P x* ⟹ (⋀*x. P x* ⟹ *thesis*) ⟹ *thesis* ⟨*proof*⟩
lemma *MOST-I*: (⋀*x. P x*) ⟹ *MOST x. P x* ⟨*proof*⟩
lemmas *MOST-iff-finiteNeg* = *MOST-iff-cofinite*

18.4 Enumeration of an Infinite Set

The set's element type must be wellordered (e.g. the natural numbers).

Could be generalized to *enumerate′ S n* = (*SOME t. t* ∈ *s* ∧ *finite* {*s* ∈ *S. s* < *t*} ∧ *card* {*s* ∈ *S. s* < *t*} = *n*).

primrec (**in** *wellorder*) *enumerate* :: *'a set* ⇒ *nat* ⇒ *'a*
where
enumerate-0: *enumerate S 0* = (*LEAST n. n* ∈ *S*)
| *enumerate-Suc*: *enumerate S* (*Suc n*) = *enumerate* (*S* − {*LEAST n. n* ∈ *S*}) *n*

lemma *enumerate-Suc′*: *enumerate S* (*Suc n*) = *enumerate* (*S* − {*enumerate S 0*}) *n*
⟨*proof*⟩

lemma *enumerate-in-set*: *infinite S* ⟹ *enumerate S n* ∈ *S*
⟨*proof*⟩

declare *enumerate-0* [*simp del*] *enumerate-Suc* [*simp del*]

lemma *enumerate-step*: *infinite S* $\Longrightarrow$ *enumerate S n* $<$ *enumerate S* (*Suc n*)
⟨*proof*⟩

lemma *enumerate-mono*: $m < n \Longrightarrow$ *infinite S* $\Longrightarrow$ *enumerate S m* $<$ *enumerate S n*
⟨*proof*⟩

lemma *le-enumerate*:
assumes *S*: *infinite S*
shows $n \leq$ *enumerate S n*
⟨*proof*⟩

lemma *enumerate-Suc''*:
fixes *S* :: *'a*::*wellorder set*
assumes *infinite S*
shows *enumerate S* (*Suc n*) = (*LEAST s*. $s \in S \wedge$ *enumerate S n* $< s$)
⟨*proof*⟩

lemma *enumerate-Ex*:
fixes *S* :: *nat set*
assumes *S*: *infinite S*
and *s*: $s \in S$
shows $\exists\, n$. *enumerate S n* $= s$
⟨*proof*⟩

lemma *bij-enumerate*:
fixes *S* :: *nat set*
assumes *S*: *infinite S*
shows *bij-betw* (*enumerate S*) *UNIV S*
⟨*proof*⟩

A pair of weird and wonderful lemmas from HOL Light.

lemma *finite-transitivity-chain*:
assumes *finite A*
and *R*: $\bigwedge x.\ \neg\ R\ x\ x\ \bigwedge x\ y\ z.\ [\![R\ x\ y;\ R\ y\ z]\!] \Longrightarrow R\ x\ z$
and *A*: $\bigwedge x.\ x \in A \Longrightarrow \exists\, y.\ y \in A \wedge R\ x\ y$
shows $A = \{\}$
⟨*proof*⟩

corollary *Union-maximal-sets*:
assumes *finite* $\mathcal{F}$
shows $\bigcup \{T \in \mathcal{F}.\ \forall\, U{\in}\mathcal{F}.\ \neg\ T \subset U\} = \bigcup \mathcal{F}$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

19 Countable sets

theory *Countable-Set*
imports *Countable Infinite-Set*
begin

19.1 Predicate for countable sets

definition *countable* :: *'a set* ⇒ *bool* **where**
countable S ⟷ (∃*f*::*'a* ⇒ *nat*. *inj-on f S*)

lemma *countableE*:
assumes *S*: *countable S* **obtains** *f* :: *'a* ⇒ *nat* **where** *inj-on f S*
⟨*proof*⟩

lemma *countableI*: *inj-on* (*f*::*'a* ⇒ *nat*) *S* ⟹ *countable S*
⟨*proof*⟩

lemma *countableI'*: *inj-on* (*f*::*'a* ⇒ *'b*::*countable*) *S* ⟹ *countable S*
⟨*proof*⟩

lemma *countableE-bij*:
assumes *S*: *countable S* **obtains** *f* :: *nat* ⇒ *'a* **and** *C* :: *nat set* **where** *bij-betw f C S*
⟨*proof*⟩

lemma *countableI-bij*: *bij-betw f* (*C*::*nat set*) *S* ⟹ *countable S*
⟨*proof*⟩

lemma *countable-finite*: *finite S* ⟹ *countable S*
⟨*proof*⟩

lemma *countableI-bij1*: *bij-betw f A B* ⟹ *countable A* ⟹ *countable B*
⟨*proof*⟩

lemma *countableI-bij2*: *bij-betw f B A* ⟹ *countable A* ⟹ *countable B*
⟨*proof*⟩

lemma *countable-iff-bij*[*simp*]: *bij-betw f A B* ⟹ *countable A* ⟷ *countable B*
⟨*proof*⟩

lemma *countable-subset*: *A* ⊆ *B* ⟹ *countable B* ⟹ *countable A*
⟨*proof*⟩

lemma *countableI-type*[*intro*, *simp*]: *countable* (*A*:: *'a* :: *countable set*)
⟨*proof*⟩

19.2 Enumerate a countable set

lemma *countableE-infinite*:

assumes *countable S infinite S*
obtains *e* :: *'a* ⇒ *nat* **where** *bij-betw e S UNIV*
⟨*proof*⟩

lemma *countable-enum-cases*:
assumes *countable S*
obtains (*finite*) *f* :: *'a* ⇒ *nat* **where** *finite S bij-betw f S* {..<*card S*}
| (*infinite*) *f* :: *'a* ⇒ *nat* **where** *infinite S bij-betw f S UNIV*
⟨*proof*⟩

definition *to-nat-on* :: *'a set* ⇒ *'a* ⇒ *nat* **where**
to-nat-on S = (*SOME f*. *if finite S then bij-betw f S* {..< *card S*} *else bij-betw f S UNIV*)

definition *from-nat-into* :: *'a set* ⇒ *nat* ⇒ *'a* **where**
from-nat-into S n = (*if n* ∈ *to-nat-on S* ' *S then inv-into S* (*to-nat-on S*) *n else SOME s*. *s*∈*S*)

lemma *to-nat-on-finite*: *finite S* ⟹ *bij-betw* (*to-nat-on S*) *S* {..< *card S*}
⟨*proof*⟩

lemma *to-nat-on-infinite*: *countable S* ⟹ *infinite S* ⟹ *bij-betw* (*to-nat-on S*) *S UNIV*
⟨*proof*⟩

lemma *bij-betw-from-nat-into-finite*: *finite S* ⟹ *bij-betw* (*from-nat-into S*) {..< *card S*} *S*
⟨*proof*⟩

lemma *bij-betw-from-nat-into*: *countable S* ⟹ *infinite S* ⟹ *bij-betw* (*from-nat-into S*) *UNIV S*
⟨*proof*⟩

lemma *countable-as-injective-image*:
assumes *countable A infinite A*
obtains *f* :: *nat* ⇒ *'a* **where** *A* = *range f inj f*
⟨*proof*⟩

lemma *inj-on-to-nat-on*[*intro*]: *countable A* ⟹ *inj-on* (*to-nat-on A*) *A*
⟨*proof*⟩

lemma *to-nat-on-inj*[*simp*]:
countable A ⟹ *a* ∈ *A* ⟹ *b* ∈ *A* ⟹ *to-nat-on A a* = *to-nat-on A b* ⟷ *a* = *b*
⟨*proof*⟩

lemma *from-nat-into-to-nat-on*[*simp*]: *countable A* ⟹ *a* ∈ *A* ⟹ *from-nat-into A* (*to-nat-on A a*) = *a*
⟨*proof*⟩

lemma *subset-range-from-nat-into*: *countable A* $\Longrightarrow$ *A* $\subseteq$ *range* (*from-nat-into A*)
⟨*proof*⟩

lemma *from-nat-into*: *A* $\neq$ {} $\Longrightarrow$ *from-nat-into A n* $\in$ *A*
⟨*proof*⟩

lemma *range-from-nat-into-subset*: *A* $\neq$ {} $\Longrightarrow$ *range* (*from-nat-into A*) $\subseteq$ *A*
⟨*proof*⟩

lemma *range-from-nat-into*[*simp*]: *A* $\neq$ {} $\Longrightarrow$ *countable A* $\Longrightarrow$ *range* (*from-nat-into A*) = *A*
⟨*proof*⟩

lemma *image-to-nat-on*: *countable A* $\Longrightarrow$ *infinite A* $\Longrightarrow$ *to-nat-on A ' A* = *UNIV*
⟨*proof*⟩

lemma *to-nat-on-surj*: *countable A* $\Longrightarrow$ *infinite A* $\Longrightarrow$ $\exists$ *a*$\in$*A*. *to-nat-on A a* = *n*
⟨*proof*⟩

lemma *to-nat-on-from-nat-into*[*simp*]: *n* $\in$ *to-nat-on A ' A* $\Longrightarrow$ *to-nat-on A* (*from-nat-into A n*) = *n*
⟨*proof*⟩

lemma *to-nat-on-from-nat-into-infinite*[*simp*]:
countable A $\Longrightarrow$ *infinite A* $\Longrightarrow$ *to-nat-on A* (*from-nat-into A n*) = *n*
⟨*proof*⟩

lemma *from-nat-into-inj*:
countable A $\Longrightarrow$ *m* $\in$ *to-nat-on A ' A* $\Longrightarrow$ *n* $\in$ *to-nat-on A ' A* $\Longrightarrow$
from-nat-into A m = *from-nat-into A n* $\longleftrightarrow$ *m* = *n*
⟨*proof*⟩

lemma *from-nat-into-inj-infinite*[*simp*]:
countable A $\Longrightarrow$ *infinite A* $\Longrightarrow$ *from-nat-into A m* = *from-nat-into A n* $\longleftrightarrow$ *m* = *n*
⟨*proof*⟩

lemma *eq-from-nat-into-iff*:
countable A $\Longrightarrow$ *x* $\in$ *A* $\Longrightarrow$ *i* $\in$ *to-nat-on A ' A* $\Longrightarrow$ *x* = *from-nat-into A i* $\longleftrightarrow$ *i* = *to-nat-on A x*
⟨*proof*⟩

lemma *from-nat-into-surj*: *countable A* $\Longrightarrow$ *a* $\in$ *A* $\Longrightarrow$ $\exists$ *n*. *from-nat-into A n* = *a*
⟨*proof*⟩

lemma *from-nat-into-inject*[*simp*]:
A $\neq$ {} $\Longrightarrow$ *countable A* $\Longrightarrow$ *B* $\neq$ {} $\Longrightarrow$ *countable B* $\Longrightarrow$ *from-nat-into A* =

from-nat-into $B \longleftrightarrow A = B$
⟨*proof*⟩

lemma *inj-on-from-nat-into*: *inj-on from-nat-into* $(\{A.\ A \neq \{\} \wedge \mathit{countable}\ A\})$
⟨*proof*⟩

19.3 Closure properties of countability

lemma *countable-SIGMA*[*intro*, *simp*]:
$\mathit{countable}\ I \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow \mathit{countable}\ (A\ i)) \Longrightarrow \mathit{countable}\ (\mathit{SIGMA}\ i : I.\ A\ i)$
⟨*proof*⟩

lemma *countable-image*[*intro*, *simp*]:
assumes *countable A*
shows *countable* (*f*‘*A*)
⟨*proof*⟩

lemma *countable-image-inj-on*: $\mathit{countable}\ (f\ `\ A) \Longrightarrow \mathit{inj\text{-}on}\ f\ A \Longrightarrow \mathit{countable}\ A$
⟨*proof*⟩

lemma *countable-UN*[*intro*, *simp*]:
fixes $I :: 'i\ \mathit{set}$ **and** $A :: 'i => 'a\ \mathit{set}$
assumes *I*: *countable I*
assumes *A*: $\bigwedge i.\ i \in I \Longrightarrow \mathit{countable}\ (A\ i)$
shows *countable* $(\bigcup i{\in}I.\ A\ i)$
⟨*proof*⟩

lemma *countable-Un*[*intro*]: $\mathit{countable}\ A \Longrightarrow \mathit{countable}\ B \Longrightarrow \mathit{countable}\ (A \cup B)$
⟨*proof*⟩

lemma *countable-Un-iff*[*simp*]: $\mathit{countable}\ (A \cup B) \longleftrightarrow \mathit{countable}\ A \wedge \mathit{countable}\ B$
⟨*proof*⟩

lemma *countable-Plus*[*intro*, *simp*]:
$\mathit{countable}\ A \Longrightarrow \mathit{countable}\ B \Longrightarrow \mathit{countable}\ (A <+> B)$
⟨*proof*⟩

lemma *countable-empty*[*intro*, *simp*]: *countable* {}
⟨*proof*⟩

lemma *countable-insert*[*intro*, *simp*]: $\mathit{countable}\ A \Longrightarrow \mathit{countable}\ (\mathit{insert}\ a\ A)$
⟨*proof*⟩

lemma *countable-Int1*[*intro*, *simp*]: $\mathit{countable}\ A \Longrightarrow \mathit{countable}\ (A \cap B)$
⟨*proof*⟩

lemma *countable-Int2*[*intro*, *simp*]: $\mathit{countable}\ B \Longrightarrow \mathit{countable}\ (A \cap B)$

⟨*proof*⟩

lemma *countable-INT*[*intro*, *simp*]: $i \in I \Longrightarrow$ *countable* $(A\ i) \Longrightarrow$ *countable* $(\bigcap i{\in}I.\ A\ i)$
⟨*proof*⟩

lemma *countable-Diff*[*intro*, *simp*]: *countable* $A \Longrightarrow$ *countable* $(A - B)$
⟨*proof*⟩

lemma *countable-insert-eq* [*simp*]: *countable* $(\mathit{insert}\ x\ A) =$ *countable* A
⟨*proof*⟩

lemma *countable-vimage*: $B \subseteq \mathit{range}\ f \Longrightarrow$ *countable* $(f\ -`\ B) \Longrightarrow$ *countable* B
⟨*proof*⟩

lemma *surj-countable-vimage*: *surj* $f \Longrightarrow$ *countable* $(f\ -`\ B) \Longrightarrow$ *countable* B
⟨*proof*⟩

lemma *countable-Collect*[*simp*]: *countable* $A \Longrightarrow$ *countable* $\{a \in A.\ \varphi\ a\}$
⟨*proof*⟩

lemma *countable-Image*:
assumes $\bigwedge y.\ y \in Y \Longrightarrow$ *countable* $(X\ ``\ \{y\})$
assumes *countable* Y
shows *countable* $(X\ ``\ Y)$
⟨*proof*⟩

lemma *countable-relpow*:
fixes $X :: 'a\ \mathit{rel}$
assumes *Image-X*: $\bigwedge Y.$ *countable* $Y \Longrightarrow$ *countable* $(X\ ``\ Y)$
assumes Y: *countable* Y
shows *countable* $((X\ \hat{}\hat{}\ i)\ ``\ Y)$
⟨*proof*⟩

lemma *countable-funpow*:
fixes $f :: 'a\ \mathit{set} \Rightarrow 'a\ \mathit{set}$
assumes $\bigwedge A.$ *countable* $A \Longrightarrow$ *countable* $(f\ A)$
and *countable* A
shows *countable* $((f\ \hat{}\hat{}\ n)\ A)$
⟨*proof*⟩

lemma *countable-rtrancl*:
$(\bigwedge Y.$ *countable* $Y \Longrightarrow$ *countable* $(X\ ``\ Y)) \Longrightarrow$ *countable* $Y \Longrightarrow$ *countable* $(X\hat{}*\ ``\ Y)$
⟨*proof*⟩

lemma *countable-lists*[*intro*, *simp*]:
assumes A: *countable* A **shows** *countable* $(\mathit{lists}\ A)$
⟨*proof*⟩

lemma *Collect-finite-eq-lists*: *Collect finite = set ' lists UNIV*
⟨*proof*⟩

lemma *countable-Collect-finite*: *countable* (*Collect* (*finite*::'*a*::*countable set*⇒*bool*))
⟨*proof*⟩

lemma *countable-int*: *countable* ℤ
⟨*proof*⟩

lemma *countable-rat*: *countable* ℚ
⟨*proof*⟩

lemma *Collect-finite-subset-eq-lists*: {*A*. *finite A* ∧ *A* ⊆ *T*} = *set ' lists T*
⟨*proof*⟩

lemma *countable-Collect-finite-subset*:
countable T ⟹ *countable* {*A*. *finite A* ∧ *A* ⊆ *T*}
⟨*proof*⟩

lemma *countable-set-option* [*simp*]: *countable* (*set-option x*)
⟨*proof*⟩

19.4 Misc lemmas

lemma *countable-subset-image*:
countable B ∧ *B* ⊆ (*f ' A*) ⟷ (∃ *A'*. *countable A'* ∧ *A'* ⊆ *A* ∧ (*B* = *f ' A'*))
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *infinite-countable-subset'*:
assumes *X*: *infinite X* **shows** ∃ *C*⊆*X*. *countable C* ∧ *infinite C*
⟨*proof*⟩

lemma *countable-all*:
assumes *S*: *countable S*
shows (∀ *s*∈*S*. *P s*) ⟷ (∀ *n*::*nat*. *from-nat-into S n* ∈ *S* ⟶ *P* (*from-nat-into S n*))
⟨*proof*⟩

lemma *finite-sequence-to-countable-set*:
assumes *countable X* **obtains** *F* **where** ⋀*i*. *F i* ⊆ *X* ⋀*i*. *F i* ⊆ *F* (*Suc i*) ⋀*i*. *finite* (*F i*) (⋃ *i*. *F i*) = *X*
⟨*proof*⟩

lemma *transfer-countable*[*transfer-rule*]:
bi-unique R ⟹ *rel-fun* (*rel-set R*) *op* = *countable countable*
⟨*proof*⟩

19.5 Uncountable

abbreviation *uncountable* **where**
uncountable A ≡ ¬ *countable A*

lemma *uncountable-def*: *uncountable A* ⟷ *A* ≠ {} ∧ ¬ (∃*f*::(*nat* ⇒ *'a*). *range f* = *A*)
⟨*proof*⟩

lemma *uncountable-bij-betw*: *bij-betw f A B* ⟹ *uncountable B* ⟹ *uncountable A*
⟨*proof*⟩

lemma *uncountable-infinite*: *uncountable A* ⟹ *infinite A*
⟨*proof*⟩

lemma *uncountable-minus-countable*:
uncountable A ⟹ *countable B* ⟹ *uncountable* (*A* − *B*)
⟨*proof*⟩

lemma *countable-Diff-eq* [*simp*]: *countable* (*A* − {*x*}) = *countable A*
⟨*proof*⟩

end

20 Countable Complete Lattices

theory *Countable-Complete-Lattices*
imports *Main Countable-Set*
begin

lemma *UNIV-nat-eq*: *UNIV* = *insert 0* (*range Suc*)
⟨*proof*⟩

class *countable-complete-lattice* = *lattice* + *Inf* + *Sup* + *bot* + *top* +
assumes *ccInf-lower*: *countable A* ⟹ *x* ∈ *A* ⟹ *Inf A* ≤ *x*
assumes *ccInf-greatest*: *countable A* ⟹ (⋀*x*. *x* ∈ *A* ⟹ *z* ≤ *x*) ⟹ *z* ≤ *Inf A*
assumes *ccSup-upper*: *countable A* ⟹ *x* ∈ *A* ⟹ *x* ≤ *Sup A*
assumes *ccSup-least*: *countable A* ⟹ (⋀*x*. *x* ∈ *A* ⟹ *x* ≤ *z*) ⟹ *Sup A* ≤ *z*
assumes *ccInf-empty* [*simp*]: *Inf* {} = *top*
assumes *ccSup-empty* [*simp*]: *Sup* {} = *bot*
begin

subclass *bounded-lattice*
⟨*proof*⟩

lemma *ccINF-lower*: *countable A* ⟹ *i* ∈ *A* ⟹ (*INF i* :*A*. *f i*) ≤ *f i*
⟨*proof*⟩

lemma *ccINF-greatest*: *countable A* $\Longrightarrow$ $(\bigwedge i.\ i \in A \Longrightarrow u \leq f\ i) \Longrightarrow u \leq$ (*INF i :A. f i*)
⟨*proof*⟩

lemma *ccSUP-upper*: *countable A* $\Longrightarrow$ $i \in A \Longrightarrow f\ i \leq$ (*SUP i :A. f i*)
⟨*proof*⟩

lemma *ccSUP-least*: *countable A* $\Longrightarrow$ $(\bigwedge i.\ i \in A \Longrightarrow f\ i \leq u) \Longrightarrow$ (*SUP i :A. f i*) $\leq u$
⟨*proof*⟩

lemma *ccInf-lower2*: *countable A* $\Longrightarrow$ $u \in A \Longrightarrow u \leq v \Longrightarrow$ *Inf A* $\leq v$
⟨*proof*⟩

lemma *ccINF-lower2*: *countable A* $\Longrightarrow$ $i \in A \Longrightarrow f\ i \leq u \Longrightarrow$ (*INF i :A. f i*) $\leq u$
⟨*proof*⟩

lemma *ccSup-upper2*: *countable A* $\Longrightarrow$ $u \in A \Longrightarrow v \leq u \Longrightarrow v \leq$ *Sup A*
⟨*proof*⟩

lemma *ccSUP-upper2*: *countable A* $\Longrightarrow$ $i \in A \Longrightarrow u \leq f\ i \Longrightarrow u \leq$ (*SUP i :A. f i*)
⟨*proof*⟩

lemma *le-ccInf-iff*: *countable A* $\Longrightarrow$ $b \leq$ *Inf A* $\longleftrightarrow$ $(\forall\, a{\in}A.\ b \leq a)$
⟨*proof*⟩

lemma *le-ccINF-iff*: *countable A* $\Longrightarrow$ $u \leq$ (*INF i :A. f i*) $\longleftrightarrow$ $(\forall\, i{\in}A.\ u \leq f\ i)$
⟨*proof*⟩

lemma *ccSup-le-iff*: *countable A* $\Longrightarrow$ *Sup A* $\leq b \longleftrightarrow$ $(\forall\, a{\in}A.\ a \leq b)$
⟨*proof*⟩

lemma *ccSUP-le-iff*: *countable A* $\Longrightarrow$ (*SUP i :A. f i*) $\leq u \longleftrightarrow$ $(\forall\, i{\in}A.\ f\ i \leq u)$
⟨*proof*⟩

lemma *ccInf-insert* [*simp*]: *countable A* $\Longrightarrow$ *Inf* (*insert a A*) = *inf a* (*Inf A*)
⟨*proof*⟩

lemma *ccINF-insert* [*simp*]: *countable A* $\Longrightarrow$ (*INF x*:*insert a A. f x*) = *inf* (*f a*) (*INFIMUM A f*)
⟨*proof*⟩

lemma *ccSup-insert* [*simp*]: *countable A* $\Longrightarrow$ *Sup* (*insert a A*) = *sup a* (*Sup A*)
⟨*proof*⟩

lemma *ccSUP-insert* [*simp*]: *countable A* $\Longrightarrow$ (*SUP x*:*insert a A. f x*) = *sup* (*f a*) (*SUPREMUM A f*)
⟨*proof*⟩

lemma *ccINF-empty* [*simp*]: $(INF\ x{:}\{\}.\ f\ x) = top$
⟨*proof*⟩

lemma *ccSUP-empty* [*simp*]: $(SUP\ x{:}\{\}.\ f\ x) = bot$
⟨*proof*⟩

lemma *ccInf-superset-mono*: *countable* $A \Longrightarrow B \subseteq A \Longrightarrow Inf\ A \leq Inf\ B$
⟨*proof*⟩

lemma *ccSup-subset-mono*: *countable* $B \Longrightarrow A \subseteq B \Longrightarrow Sup\ A \leq Sup\ B$
⟨*proof*⟩

lemma *ccInf-mono*:
assumes [*intro*]: *countable* B *countable* A
assumes $\bigwedge b.\ b \in B \Longrightarrow \exists\, a{\in}A.\ a \leq b$
shows $Inf\ A \leq Inf\ B$
⟨*proof*⟩

lemma *ccINF-mono*:
countable $A \Longrightarrow$ *countable* $B \Longrightarrow (\bigwedge m.\ m \in B \Longrightarrow \exists\, n{\in}A.\ f\ n \leq g\ m) \Longrightarrow (INF\ n{:}A.\ f\ n) \leq (INF\ n{:}B.\ g\ n)$
⟨*proof*⟩

lemma *ccSup-mono*:
assumes [*intro*]: *countable* B *countable* A
assumes $\bigwedge a.\ a \in A \Longrightarrow \exists\, b{\in}B.\ a \leq b$
shows $Sup\ A \leq Sup\ B$
⟨*proof*⟩

lemma *ccSUP-mono*:
countable $A \Longrightarrow$ *countable* $B \Longrightarrow (\bigwedge n.\ n \in A \Longrightarrow \exists\, m{\in}B.\ f\ n \leq g\ m) \Longrightarrow (SUP\ n{:}A.\ f\ n) \leq (SUP\ n{:}B.\ g\ n)$
⟨*proof*⟩

lemma *ccINF-superset-mono*:
countable $A \Longrightarrow B \subseteq A \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow f\ x \leq g\ x) \Longrightarrow (INF\ x{:}A.\ f\ x) \leq (INF\ x{:}B.\ g\ x)$
⟨*proof*⟩

lemma *ccSUP-subset-mono*:
countable $B \Longrightarrow A \subseteq B \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow f\ x \leq g\ x) \Longrightarrow (SUP\ x{:}A.\ f\ x) \leq (SUP\ x{:}B.\ g\ x)$
⟨*proof*⟩

lemma *less-eq-ccInf-inter*: *countable* $A \Longrightarrow$ *countable* $B \Longrightarrow sup\ (Inf\ A)\ (Inf\ B) \leq Inf\ (A \cap B)$
⟨*proof*⟩

lemma *ccSup-inter-less-eq*: *countable A* $\Longrightarrow$ *countable B* $\Longrightarrow$ *Sup* $(A \cap B) \leq$ *inf* (*Sup A*) (*Sup B*)
⟨*proof*⟩

lemma *ccInf-union-distrib*: *countable A* $\Longrightarrow$ *countable B* $\Longrightarrow$ *Inf* $(A \cup B) =$ *inf* (*Inf A*) (*Inf B*)
⟨*proof*⟩

lemma *ccINF-union*:
countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ (*INF* i:$A \cup B$. *M i*) = *inf* (*INF* i:*A*. *M i*) (*INF* i:*B*. *M i*)
⟨*proof*⟩

lemma *ccSup-union-distrib*: *countable A* $\Longrightarrow$ *countable B* $\Longrightarrow$ *Sup* $(A \cup B) =$ *sup* (*Sup A*) (*Sup B*)
⟨*proof*⟩

lemma *ccSUP-union*:
countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ (*SUP* i:$A \cup B$. *M i*) = *sup* (*SUP* i:*A*. *M i*) (*SUP* i:*B*. *M i*)
⟨*proof*⟩

lemma *ccINF-inf-distrib*: *countable A* $\Longrightarrow$ *inf* (*INF* a:*A*. *f a*) (*INF* a:*A*. *g a*) = (*INF* a:*A*. *inf* (*f a*) (*g a*))
⟨*proof*⟩

lemma *ccSUP-sup-distrib*: *countable A* $\Longrightarrow$ *sup* (*SUP* a:*A*. *f a*) (*SUP* a:*A*. *g a*) = (*SUP* a:*A*. *sup* (*f a*) (*g a*))
⟨*proof*⟩

lemma *ccINF-const* [*simp*]: $A \neq \{\}$ $\Longrightarrow$ (*INF i* :*A*. *f*) = *f*
⟨*proof*⟩

lemma *ccSUP-const* [*simp*]: $A \neq \{\}$ $\Longrightarrow$ (*SUP i* :*A*. *f*) = *f*
⟨*proof*⟩

lemma *ccINF-top* [*simp*]: (*INF* x:*A*. *top*) = *top*
⟨*proof*⟩

lemma *ccSUP-bot* [*simp*]: (*SUP* x:*A*. *bot*) = *bot*
⟨*proof*⟩

lemma *ccINF-commute*: *countable A* $\Longrightarrow$ *countable B* $\Longrightarrow$ (*INF* i:*A*. *INF* j:*B*. *f i j*) = (*INF* j:*B*. *INF* i:*A*. *f i j*)
⟨*proof*⟩

lemma *ccSUP-commute*: *countable A* $\Longrightarrow$ *countable B* $\Longrightarrow$ (*SUP* i:*A*. *SUP* j:*B*. *f i j*) = (*SUP* j:*B*. *SUP* i:*A*. *f i j*)

⟨*proof*⟩

end

context
fixes $a :: 'a$::{*countable-complete-lattice*, *linorder*}
begin

lemma *less-ccSup-iff*: *countable* $S \Longrightarrow a < Sup\ S \longleftrightarrow (\exists\, x{\in}S.\ a < x)$
⟨*proof*⟩

lemma *less-ccSUP-iff*: *countable* $A \Longrightarrow a < (SUP\ i{:}A.\ f\ i) \longleftrightarrow (\exists\, x{\in}A.\ a < f\ x)$
⟨*proof*⟩

lemma *ccInf-less-iff*: *countable* $S \Longrightarrow Inf\ S < a \longleftrightarrow (\exists\, x{\in}S.\ x < a)$
⟨*proof*⟩

lemma *ccINF-less-iff*: *countable* $A \Longrightarrow (INF\ i{:}A.\ f\ i) < a \longleftrightarrow (\exists\, x{\in}A.\ f\ x < a)$
⟨*proof*⟩

end

class *countable-complete-distrib-lattice* = *countable-complete-lattice* +
assumes *sup-ccInf*: *countable* $B \Longrightarrow sup\ a\ (Inf\ B) = (INF\ b{:}B.\ sup\ a\ b)$
assumes *inf-ccSup*: *countable* $B \Longrightarrow inf\ a\ (Sup\ B) = (SUP\ b{:}B.\ inf\ a\ b)$
begin

lemma *sup-ccINF*:
countable $B \Longrightarrow sup\ a\ (INF\ b{:}B.\ f\ b) = (INF\ b{:}B.\ sup\ a\ (f\ b))$
⟨*proof*⟩

lemma *inf-ccSUP*:
countable $B \Longrightarrow inf\ a\ (SUP\ b{:}B.\ f\ b) = (SUP\ b{:}B.\ inf\ a\ (f\ b))$
⟨*proof*⟩

subclass *distrib-lattice*
⟨*proof*⟩

lemma *ccInf-sup*:
countable $B \Longrightarrow sup\ (Inf\ B)\ a = (INF\ b{:}B.\ sup\ b\ a)$
⟨*proof*⟩

lemma *ccSup-inf*:
countable $B \Longrightarrow inf\ (Sup\ B)\ a = (SUP\ b{:}B.\ inf\ b\ a)$
⟨*proof*⟩

lemma *ccINF-sup*:
countable $B \Longrightarrow sup\ (INF\ b{:}B.\ f\ b)\ a = (INF\ b{:}B.\ sup\ (f\ b)\ a)$
⟨*proof*⟩

lemma *ccSUP-inf*:
countable B $\Longrightarrow$ *inf* (*SUP b*:*B*. *f b*) *a* = (*SUP b*:*B*. *inf* (*f b*) *a*)
⟨*proof*⟩

lemma *ccINF-sup-distrib2*:
countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ *sup* (*INF a*:*A*. *f a*) (*INF b*:*B*. *g b*) = (*INF a*:*A*. *INF b*:*B*. *sup* (*f a*) (*g b*))
⟨*proof*⟩

lemma *ccSUP-inf-distrib2*:
countable A $\Longrightarrow$ *countable B* $\Longrightarrow$ *inf* (*SUP a*:*A*. *f a*) (*SUP b*:*B*. *g b*) = (*SUP a*:*A*. *SUP b*:*B*. *inf* (*f a*) (*g b*))
⟨*proof*⟩

context
fixes *f* :: *'a* $\Rightarrow$ *'b*::*countable-complete-lattice*
assumes *mono f*
begin

lemma *mono-ccInf*:
countable A $\Longrightarrow$ *f* (*Inf A*) $\leq$ (*INF x*:*A*. *f x*)
⟨*proof*⟩

lemma *mono-ccSup*:
countable A $\Longrightarrow$ (*SUP x*:*A*. *f x*) $\leq$ *f* (*Sup A*)
⟨*proof*⟩

lemma *mono-ccINF*:
countable I $\Longrightarrow$ *f* (*INF i* : *I*. *A i*) $\leq$ (*INF x* : *I*. *f* (*A x*))
⟨*proof*⟩

lemma *mono-ccSUP*:
countable I $\Longrightarrow$ (*SUP x* : *I*. *f* (*A x*)) $\leq$ *f* (*SUP i* : *I*. *A i*)
⟨*proof*⟩

end

end

20.0.1 Instances of countable complete lattices

instance *fun* :: (*type*, *countable-complete-lattice*) *countable-complete-lattice*
⟨*proof*⟩

subclass (**in** *complete-lattice*) *countable-complete-lattice*
⟨*proof*⟩

subclass (**in** *complete-distrib-lattice*) *countable-complete-distrib-lattice*

⟨*proof*⟩

end

21 Cardinal Notations

theory *Cardinal-Notations*
imports *Main*
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** +*c* 65) **and**
BNF-Cardinal-Arithmetic.cprod (**infixr** ∗*c* 80) **and**
BNF-Cardinal-Arithmetic.cexp (**infixr** ^*c* 90)

abbreviation *cinfinite* ≡ *BNF-Cardinal-Arithmetic.cinfinite*
abbreviation *czero* ≡ *BNF-Cardinal-Arithmetic.czero*
abbreviation *cone* ≡ *BNF-Cardinal-Arithmetic.cone*
abbreviation *ctwo* ≡ *BNF-Cardinal-Arithmetic.ctwo*

end

22 Type of (at Most) Countable Sets

theory *Countable-Set-Type*
imports *Countable-Set Cardinal-Notations HOL.Conditionally-Complete-Lattices*
begin

22.1 Cardinal stuff

lemma *countable-card-of-nat*: *countable A* ⟷ |*A*| ≤*o* |*UNIV*::*nat set*|
⟨*proof*⟩

lemma *countable-card-le-natLeq*: *countable A* ⟷ |*A*| ≤*o natLeq*
⟨*proof*⟩

lemma *countable-or-card-of*:
assumes *countable A*
shows (*finite A* ∧ |*A*| <*o* |*UNIV*::*nat set*|) ∨
(*infinite A* ∧ |*A*| =*o* |*UNIV*::*nat set*|)
⟨*proof*⟩

lemma *countable-cases-card-of*[*elim*]:

assumes *countable A*
obtains (*Fin*) *finite A* $|A| <o |UNIV::nat\ set|$
| (*Inf*) *infinite A* $|A| =o |UNIV::nat\ set|$
⟨*proof*⟩

lemma *countable-or*:
countable A $\Longrightarrow (\exists\ f::'a\Rightarrow nat.\ finite\ A \wedge inj\text{-}on\ f\ A) \vee (\exists\ f::'a\Rightarrow nat.\ infinite\ A \wedge bij\text{-}betw\ f\ A\ UNIV)$
⟨*proof*⟩

lemma *countable-cases*[*elim*]:
assumes *countable A*
obtains (*Fin*) $f :: 'a\Rightarrow nat$ **where** *finite A inj-on f A*
| (*Inf*) $f :: 'a\Rightarrow nat$ **where** *infinite A bij-betw f A UNIV*
⟨*proof*⟩

lemma *countable-ordLeq*:
assumes $|A| \leq o\ |B|$ **and** *countable B*
shows *countable A*
⟨*proof*⟩

lemma *countable-ordLess*:
assumes *AB*: $|A| <o |B|$ **and** *B*: *countable B*
shows *countable A*
⟨*proof*⟩

22.2 The type of countable sets

typedef $'a\ cset = \{A :: 'a\ set.\ countable\ A\}$ **morphisms** *rcset acset*
⟨*proof*⟩

setup-lifting *type-definition-cset*

declare
rcset-inverse[*simp*]
acset-inverse[*Transfer.transferred*, *unfolded mem-Collect-eq*, *simp*]
acset-inject[*Transfer.transferred*, *unfolded mem-Collect-eq*, *simp*]
rcset[*Transfer.transferred*, *unfolded mem-Collect-eq*, *simp*]

instantiation *cset* :: (*type*) {*bounded-lattice-bot*, *distrib-lattice*, *minus*}
begin

lift-definition *bot-cset* :: $'a\ cset$ **is** {} **parametric** *empty-transfer* ⟨*proof*⟩

lift-definition *less-eq-cset* :: $'a\ cset \Rightarrow 'a\ cset \Rightarrow bool$
is *subset-eq* **parametric** *subset-transfer* ⟨*proof*⟩

definition *less-cset* :: $'a\ cset \Rightarrow 'a\ cset \Rightarrow bool$
where $xs < ys \equiv xs \leq ys \wedge xs \neq (ys::'a\ cset)$

lemma *less-cset-transfer*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique A*
shows ((*pcr-cset A*) ===> (*pcr-cset A*) ===> *op =*) *op ⊂ op <*
⟨*proof*⟩

lift-definition *sup-cset* :: *'a cset ⇒ 'a cset ⇒ 'a cset*
is *union* **parametric** *union-transfer* ⟨*proof*⟩

lift-definition *inf-cset* :: *'a cset ⇒ 'a cset ⇒ 'a cset*
is *inter* **parametric** *inter-transfer* ⟨*proof*⟩

lift-definition *minus-cset* :: *'a cset ⇒ 'a cset ⇒ 'a cset*
is *minus* **parametric** *Diff-transfer* ⟨*proof*⟩

instance ⟨*proof*⟩

end

abbreviation *cempty* :: *'a cset* **where** *cempty ≡ bot*
abbreviation *csubset-eq* :: *'a cset ⇒ 'a cset ⇒ bool* **where** *csubset-eq xs ys ≡ xs*
≤ ys
abbreviation *csubset* :: *'a cset ⇒ 'a cset ⇒ bool* **where** *csubset xs ys ≡ xs < ys*
abbreviation *cUn* :: *'a cset ⇒ 'a cset ⇒ 'a cset* **where** *cUn xs ys ≡ sup xs ys*
abbreviation *cInt* :: *'a cset ⇒ 'a cset ⇒ 'a cset* **where** *cInt xs ys ≡ inf xs ys*
abbreviation *cDiff* :: *'a cset ⇒ 'a cset ⇒ 'a cset* **where** *cDiff xs ys ≡ minus xs*
ys

lift-definition *cin* :: *'a ⇒ 'a cset ⇒ bool* **is** *op ∈* **parametric** *member-transfer*
⟨*proof*⟩
lift-definition *cinsert* :: *'a ⇒ 'a cset ⇒ 'a cset* **is** *insert* **parametric** *Lifting-Set.insert-transfer*
⟨*proof*⟩
abbreviation *csingle* :: *'a ⇒ 'a cset* **where** *csingle x ≡ cinsert x cempty*
lift-definition *cimage* :: *('a ⇒ 'b) ⇒ 'a cset ⇒ 'b cset* **is** *op ‘* **parametric**
image-transfer
⟨*proof*⟩
lift-definition *cBall* :: *'a cset ⇒ ('a ⇒ bool) ⇒ bool* **is** *Ball* **parametric** *Ball-transfer*
⟨*proof*⟩
lift-definition *cBex* :: *'a cset ⇒ ('a ⇒ bool) ⇒ bool* **is** *Bex* **parametric** *Bex-transfer*
⟨*proof*⟩
lift-definition *cUNION* :: *'a cset ⇒ ('a ⇒ 'b cset) ⇒ 'b cset*
is *UNION* **parametric** *UNION-transfer* ⟨*proof*⟩
definition *cUnion* :: *'a cset cset ⇒ 'a cset* **where** *cUnion A = cUNION A id*

lemma *Union-conv-UNION*: $\bigcup A = UNION\ A\ id$
⟨*proof*⟩

lemma *cUnion-transfer* [*transfer-rule*]:

rel-fun (*pcr-cset* (*pcr-cset A*)) (*pcr-cset A*) *Union cUnion*
⟨*proof*⟩

lemmas *cset-eqI* = *set-eqI*[*Transfer.transferred*]
lemmas *cset-eq-iff*[*no-atp*] = *set-eq-iff*[*Transfer.transferred*]
lemmas *cBallI*[*intro*!] = *ballI*[*Transfer.transferred*]
lemmas *cbspec*[*dest?*] = *bspec*[*Transfer.transferred*]
lemmas *cBallE*[*elim*] = *ballE*[*Transfer.transferred*]
lemmas *cBexI*[*intro*] = *bexI*[*Transfer.transferred*]
lemmas *rev-cBexI*[*intro?*] = *rev-bexI*[*Transfer.transferred*]
lemmas *cBexCI* = *bexCI*[*Transfer.transferred*]
lemmas *cBexE*[*elim*!] = *bexE*[*Transfer.transferred*]
lemmas *cBall-triv*[*simp*] = *ball-triv*[*Transfer.transferred*]
lemmas *cBex-triv*[*simp*] = *bex-triv*[*Transfer.transferred*]
lemmas *cBex-triv-one-point1*[*simp*] = *bex-triv-one-point1*[*Transfer.transferred*]
lemmas *cBex-triv-one-point2*[*simp*] = *bex-triv-one-point2*[*Transfer.transferred*]
lemmas *cBex-one-point1*[*simp*] = *bex-one-point1*[*Transfer.transferred*]
lemmas *cBex-one-point2*[*simp*] = *bex-one-point2*[*Transfer.transferred*]
lemmas *cBall-one-point1*[*simp*] = *ball-one-point1*[*Transfer.transferred*]
lemmas *cBall-one-point2*[*simp*] = *ball-one-point2*[*Transfer.transferred*]
lemmas *cBall-conj-distrib* = *ball-conj-distrib*[*Transfer.transferred*]
lemmas *cBex-disj-distrib* = *bex-disj-distrib*[*Transfer.transferred*]
lemmas *cBall-cong* = *ball-cong*[*Transfer.transferred*]
lemmas *cBex-cong* = *bex-cong*[*Transfer.transferred*]
lemmas *csubsetI*[*intro*!] = *subsetI*[*Transfer.transferred*]
lemmas *csubsetD*[*elim*, *intro?*] = *subsetD*[*Transfer.transferred*]
lemmas *rev-csubsetD*[*no-atp*,*intro?*] = *rev-subsetD*[*Transfer.transferred*]
lemmas *csubsetCE*[*no-atp*,*elim*] = *subsetCE*[*Transfer.transferred*]
lemmas *csubset-eq*[*no-atp*] = *subset-eq*[*Transfer.transferred*]
lemmas *contra-csubsetD*[*no-atp*] = *contra-subsetD*[*Transfer.transferred*]
lemmas *csubset-refl* = *subset-refl*[*Transfer.transferred*]
lemmas *csubset-trans* = *subset-trans*[*Transfer.transferred*]
lemmas *cset-rev-mp* = *set-rev-mp*[*Transfer.transferred*]
lemmas *cset-mp* = *set-mp*[*Transfer.transferred*]
lemmas *csubset-not-fsubset-eq*[*code*] = *subset-not-subset-eq*[*Transfer.transferred*]
lemmas *eq-cmem-trans* = *eq-mem-trans*[*Transfer.transferred*]
lemmas *csubset-antisym*[*intro*!] = *subset-antisym*[*Transfer.transferred*]
lemmas *cequalityD1* = *equalityD1*[*Transfer.transferred*]
lemmas *cequalityD2* = *equalityD2*[*Transfer.transferred*]
lemmas *cequalityE* = *equalityE*[*Transfer.transferred*]
lemmas *cequalityCE*[*elim*] = *equalityCE*[*Transfer.transferred*]
lemmas *eqcset-imp-iff* = *eqset-imp-iff*[*Transfer.transferred*]
lemmas *eqcelem-imp-iff* = *eqelem-imp-iff*[*Transfer.transferred*]
lemmas *cempty-iff*[*simp*] = *empty-iff*[*Transfer.transferred*]
lemmas *cempty-fsubsetI*[*iff*] = *empty-subsetI*[*Transfer.transferred*]
lemmas *equals-cemptyI* = *equals0I*[*Transfer.transferred*]
lemmas *equals-cemptyD* = *equals0D*[*Transfer.transferred*]
lemmas *cBall-cempty*[*simp*] = *ball-empty*[*Transfer.transferred*]
lemmas *cBex-cempty*[*simp*] = *bex-empty*[*Transfer.transferred*]

lemmas *cInt-iff*[*simp*] = *Int-iff*[*Transfer.transferred*]
lemmas *cIntI*[*intro*!] = *IntI*[*Transfer.transferred*]
lemmas *cIntD1* = *IntD1*[*Transfer.transferred*]
lemmas *cIntD2* = *IntD2*[*Transfer.transferred*]
lemmas *cIntE*[*elim*!] = *IntE*[*Transfer.transferred*]
lemmas *cUn-iff*[*simp*] = *Un-iff*[*Transfer.transferred*]
lemmas *cUnI1*[*elim?*] = *UnI1*[*Transfer.transferred*]
lemmas *cUnI2*[*elim?*] = *UnI2*[*Transfer.transferred*]
lemmas *cUnCI*[*intro*!] = *UnCI*[*Transfer.transferred*]
lemmas *cuUnE*[*elim*!] = *UnE*[*Transfer.transferred*]
lemmas *cDiff-iff*[*simp*] = *Diff-iff*[*Transfer.transferred*]
lemmas *cDiffI*[*intro*!] = *DiffI*[*Transfer.transferred*]
lemmas *cDiffD1* = *DiffD1*[*Transfer.transferred*]
lemmas *cDiffD2* = *DiffD2*[*Transfer.transferred*]
lemmas *cDiffE*[*elim*!] = *DiffE*[*Transfer.transferred*]
lemmas *cinsert-iff*[*simp*] = *insert-iff*[*Transfer.transferred*]
lemmas *cinsertI1* = *insertI1*[*Transfer.transferred*]
lemmas *cinsertI2* = *insertI2*[*Transfer.transferred*]
lemmas *cinsertE*[*elim*!] = *insertE*[*Transfer.transferred*]
lemmas *cinsertCI*[*intro*!] = *insertCI*[*Transfer.transferred*]
lemmas *csubset-cinsert-iff* = *subset-insert-iff*[*Transfer.transferred*]
lemmas *cinsert-ident* = *insert-ident*[*Transfer.transferred*]
lemmas *csingletonI*[*intro*!,*no-atp*] = *singletonI*[*Transfer.transferred*]
lemmas *csingletonD*[*dest*!,*no-atp*] = *singletonD*[*Transfer.transferred*]
lemmas *fsingletonE* = *csingletonD* [*elim-format*]
lemmas *csingleton-iff* = *singleton-iff*[*Transfer.transferred*]
lemmas *csingleton-inject*[*dest*!] = *singleton-inject*[*Transfer.transferred*]
lemmas *csingleton-finsert-inj-eq*[*iff*,*no-atp*] = *singleton-insert-inj-eq*[*Transfer.transferred*]
lemmas *csingleton-finsert-inj-eq'*[*iff*,*no-atp*] = *singleton-insert-inj-eq'*[*Transfer.transferred*]
lemmas *csubset-csingletonD* = *subset-singletonD*[*Transfer.transferred*]
lemmas *cDiff-single-cinsert* = *Diff-single-insert*[*Transfer.transferred*]
lemmas *cdoubleton-eq-iff* = *doubleton-eq-iff*[*Transfer.transferred*]
lemmas *cUn-csingleton-iff* = *Un-singleton-iff*[*Transfer.transferred*]
lemmas *csingleton-cUn-iff* = *singleton-Un-iff*[*Transfer.transferred*]
lemmas *cimage-eqI*[*simp*, *intro*] = *image-eqI*[*Transfer.transferred*]
lemmas *cimageI* = *imageI*[*Transfer.transferred*]
lemmas *rev-cimage-eqI* = *rev-image-eqI*[*Transfer.transferred*]
lemmas *cimageE*[*elim*!] = *imageE*[*Transfer.transferred*]
lemmas *Compr-cimage-eq* = *Compr-image-eq*[*Transfer.transferred*]
lemmas *cimage-cUn* = *image-Un*[*Transfer.transferred*]
lemmas *cimage-iff* = *image-iff*[*Transfer.transferred*]
lemmas *cimage-csubset-iff*[*no-atp*] = *image-subset-iff*[*Transfer.transferred*]
lemmas *cimage-csubsetI* = *image-subsetI*[*Transfer.transferred*]
lemmas *cimage-ident*[*simp*] = *image-ident*[*Transfer.transferred*]
lemmas *if-split-cin1* = *if-split-mem1*[*Transfer.transferred*]
lemmas *if-split-cin2* = *if-split-mem2*[*Transfer.transferred*]
lemmas *cpsubsetI*[*intro*!,*no-atp*] = *psubsetI*[*Transfer.transferred*]
lemmas *cpsubsetE*[*elim*!,*no-atp*] = *psubsetE*[*Transfer.transferred*]
lemmas *cpsubset-finsert-iff* = *psubset-insert-iff*[*Transfer.transferred*]

lemmas *cpsubset-eq* = *psubset-eq*[*Transfer.transferred*]
lemmas *cpsubset-imp-fsubset* = *psubset-imp-subset*[*Transfer.transferred*]
lemmas *cpsubset-trans* = *psubset-trans*[*Transfer.transferred*]
lemmas *cpsubsetD* = *psubsetD*[*Transfer.transferred*]
lemmas *cpsubset-csubset-trans* = *psubset-subset-trans*[*Transfer.transferred*]
lemmas *csubset-cpsubset-trans* = *subset-psubset-trans*[*Transfer.transferred*]
lemmas *cpsubset-imp-ex-fmem* = *psubset-imp-ex-mem*[*Transfer.transferred*]
lemmas *csubset-cinsertI* = *subset-insertI*[*Transfer.transferred*]
lemmas *csubset-cinsertI2* = *subset-insertI2*[*Transfer.transferred*]
lemmas *csubset-cinsert* = *subset-insert*[*Transfer.transferred*]
lemmas *cUn-upper1* = *Un-upper1*[*Transfer.transferred*]
lemmas *cUn-upper2* = *Un-upper2*[*Transfer.transferred*]
lemmas *cUn-least* = *Un-least*[*Transfer.transferred*]
lemmas *cInt-lower1* = *Int-lower1*[*Transfer.transferred*]
lemmas *cInt-lower2* = *Int-lower2*[*Transfer.transferred*]
lemmas *cInt-greatest* = *Int-greatest*[*Transfer.transferred*]
lemmas *cDiff-csubset* = *Diff-subset*[*Transfer.transferred*]
lemmas *cDiff-csubset-conv* = *Diff-subset-conv*[*Transfer.transferred*]
lemmas *csubset-cempty*[*simp*] = *subset-empty*[*Transfer.transferred*]
lemmas *not-cpsubset-cempty*[*iff*] = *not-psubset-empty*[*Transfer.transferred*]
lemmas *cinsert-is-cUn* = *insert-is-Un*[*Transfer.transferred*]
lemmas *cinsert-not-cempty*[*simp*] = *insert-not-empty*[*Transfer.transferred*]
lemmas *cempty-not-cinsert* = *empty-not-insert*[*Transfer.transferred*]
lemmas *cinsert-absorb* = *insert-absorb*[*Transfer.transferred*]
lemmas *cinsert-absorb2*[*simp*] = *insert-absorb2*[*Transfer.transferred*]
lemmas *cinsert-commute* = *insert-commute*[*Transfer.transferred*]
lemmas *cinsert-csubset*[*simp*] = *insert-subset*[*Transfer.transferred*]
lemmas *cinsert-cinter-cinsert*[*simp*] = *insert-inter-insert*[*Transfer.transferred*]
lemmas *cinsert-disjoint*[*simp*,*no-atp*] = *insert-disjoint*[*Transfer.transferred*]
lemmas *disjoint-cinsert*[*simp*,*no-atp*] = *disjoint-insert*[*Transfer.transferred*]
lemmas *cimage-cempty*[*simp*] = *image-empty*[*Transfer.transferred*]
lemmas *cimage-cinsert*[*simp*] = *image-insert*[*Transfer.transferred*]
lemmas *cimage-constant* = *image-constant*[*Transfer.transferred*]
lemmas *cimage-constant-conv* = *image-constant-conv*[*Transfer.transferred*]
lemmas *cimage-cimage* = *image-image*[*Transfer.transferred*]
lemmas *cinsert-cimage*[*simp*] = *insert-image*[*Transfer.transferred*]
lemmas *cimage-is-cempty*[*iff*] = *image-is-empty*[*Transfer.transferred*]
lemmas *cempty-is-cimage*[*iff*] = *empty-is-image*[*Transfer.transferred*]
lemmas *cimage-cong* = *image-cong*[*Transfer.transferred*]
lemmas *cimage-cInt-csubset* = *image-Int-subset*[*Transfer.transferred*]
lemmas *cimage-cDiff-csubset* = *image-diff-subset*[*Transfer.transferred*]
lemmas *cInt-absorb* = *Int-absorb*[*Transfer.transferred*]
lemmas *cInt-left-absorb* = *Int-left-absorb*[*Transfer.transferred*]
lemmas *cInt-commute* = *Int-commute*[*Transfer.transferred*]
lemmas *cInt-left-commute* = *Int-left-commute*[*Transfer.transferred*]
lemmas *cInt-assoc* = *Int-assoc*[*Transfer.transferred*]
lemmas *cInt-ac* = *Int-ac*[*Transfer.transferred*]
lemmas *cInt-absorb1* = *Int-absorb1*[*Transfer.transferred*]
lemmas *cInt-absorb2* = *Int-absorb2*[*Transfer.transferred*]

lemmas *cInt-cempty-left* = *Int-empty-left*[*Transfer.transferred*]
lemmas *cInt-cempty-right* = *Int-empty-right*[*Transfer.transferred*]
lemmas *disjoint-iff-cnot-equal* = *disjoint-iff-not-equal*[*Transfer.transferred*]
lemmas *cInt-cUn-distrib* = *Int-Un-distrib*[*Transfer.transferred*]
lemmas *cInt-cUn-distrib2* = *Int-Un-distrib2*[*Transfer.transferred*]
lemmas *cInt-csubset-iff*[*no-atp*, *simp*] = *Int-subset-iff*[*Transfer.transferred*]
lemmas *cUn-absorb* = *Un-absorb*[*Transfer.transferred*]
lemmas *cUn-left-absorb* = *Un-left-absorb*[*Transfer.transferred*]
lemmas *cUn-commute* = *Un-commute*[*Transfer.transferred*]
lemmas *cUn-left-commute* = *Un-left-commute*[*Transfer.transferred*]
lemmas *cUn-assoc* = *Un-assoc*[*Transfer.transferred*]
lemmas *cUn-ac* = *Un-ac*[*Transfer.transferred*]
lemmas *cUn-absorb1* = *Un-absorb1*[*Transfer.transferred*]
lemmas *cUn-absorb2* = *Un-absorb2*[*Transfer.transferred*]
lemmas *cUn-cempty-left* = *Un-empty-left*[*Transfer.transferred*]
lemmas *cUn-cempty-right* = *Un-empty-right*[*Transfer.transferred*]
lemmas *cUn-cinsert-left*[*simp*] = *Un-insert-left*[*Transfer.transferred*]
lemmas *cUn-cinsert-right*[*simp*] = *Un-insert-right*[*Transfer.transferred*]
lemmas *cInt-cinsert-left* = *Int-insert-left*[*Transfer.transferred*]
lemmas *cInt-cinsert-left-if0*[*simp*] = *Int-insert-left-if0*[*Transfer.transferred*]
lemmas *cInt-cinsert-left-if1*[*simp*] = *Int-insert-left-if1*[*Transfer.transferred*]
lemmas *cInt-cinsert-right* = *Int-insert-right*[*Transfer.transferred*]
lemmas *cInt-cinsert-right-if0*[*simp*] = *Int-insert-right-if0*[*Transfer.transferred*]
lemmas *cInt-cinsert-right-if1*[*simp*] = *Int-insert-right-if1*[*Transfer.transferred*]
lemmas *cUn-cInt-distrib* = *Un-Int-distrib*[*Transfer.transferred*]
lemmas *cUn-cInt-distrib2* = *Un-Int-distrib2*[*Transfer.transferred*]
lemmas *cUn-cInt-crazy* = *Un-Int-crazy*[*Transfer.transferred*]
lemmas *csubset-cUn-eq* = *subset-Un-eq*[*Transfer.transferred*]
lemmas *cUn-cempty*[*iff*] = *Un-empty*[*Transfer.transferred*]
lemmas *cUn-csubset-iff*[*no-atp*, *simp*] = *Un-subset-iff*[*Transfer.transferred*]
lemmas *cUn-cDiff-cInt* = *Un-Diff-Int*[*Transfer.transferred*]
lemmas *cDiff-cInt2* = *Diff-Int2*[*Transfer.transferred*]
lemmas *cUn-cInt-assoc-eq* = *Un-Int-assoc-eq*[*Transfer.transferred*]
lemmas *cBall-cUn* = *ball-Un*[*Transfer.transferred*]
lemmas *cBex-cUn* = *bex-Un*[*Transfer.transferred*]
lemmas *cDiff-eq-cempty-iff*[*simp*,*no-atp*] = *Diff-eq-empty-iff*[*Transfer.transferred*]
lemmas *cDiff-cancel*[*simp*] = *Diff-cancel*[*Transfer.transferred*]
lemmas *cDiff-idemp*[*simp*] = *Diff-idemp*[*Transfer.transferred*]
lemmas *cDiff-triv* = *Diff-triv*[*Transfer.transferred*]
lemmas *cempty-cDiff*[*simp*] = *empty-Diff*[*Transfer.transferred*]
lemmas *cDiff-cempty*[*simp*] = *Diff-empty*[*Transfer.transferred*]
lemmas *cDiff-cinsert0*[*simp*,*no-atp*] = *Diff-insert0*[*Transfer.transferred*]
lemmas *cDiff-cinsert* = *Diff-insert*[*Transfer.transferred*]
lemmas *cDiff-cinsert2* = *Diff-insert2*[*Transfer.transferred*]
lemmas *cinsert-cDiff-if* = *insert-Diff-if*[*Transfer.transferred*]
lemmas *cinsert-cDiff1*[*simp*] = *insert-Diff1*[*Transfer.transferred*]
lemmas *cinsert-cDiff-single*[*simp*] = *insert-Diff-single*[*Transfer.transferred*]
lemmas *cinsert-cDiff* = *insert-Diff*[*Transfer.transferred*]
lemmas *cDiff-cinsert-absorb* = *Diff-insert-absorb*[*Transfer.transferred*]

lemmas *cDiff-disjoint*[*simp*] = *Diff-disjoint*[*Transfer.transferred*]
lemmas *cDiff-partition* = *Diff-partition*[*Transfer.transferred*]
lemmas *double-cDiff* = *double-diff*[*Transfer.transferred*]
lemmas *cUn-cDiff-cancel*[*simp*] = *Un-Diff-cancel*[*Transfer.transferred*]
lemmas *cUn-cDiff-cancel2*[*simp*] = *Un-Diff-cancel2*[*Transfer.transferred*]
lemmas *cDiff-cUn* = *Diff-Un*[*Transfer.transferred*]
lemmas *cDiff-cInt* = *Diff-Int*[*Transfer.transferred*]
lemmas *cUn-cDiff* = *Un-Diff*[*Transfer.transferred*]
lemmas *cInt-cDiff* = *Int-Diff*[*Transfer.transferred*]
lemmas *cDiff-cInt-distrib* = *Diff-Int-distrib*[*Transfer.transferred*]
lemmas *cDiff-cInt-distrib2* = *Diff-Int-distrib2*[*Transfer.transferred*]
lemmas *cset-eq-csubset* = *set-eq-subset*[*Transfer.transferred*]
lemmas *csubset-iff*[*no-atp*] = *subset-iff*[*Transfer.transferred*]
lemmas *csubset-iff-pfsubset-eq* = *subset-iff-psubset-eq*[*Transfer.transferred*]
lemmas *all-not-cin-conv*[*simp*] = *all-not-in-conv*[*Transfer.transferred*]
lemmas *ex-cin-conv* = *ex-in-conv*[*Transfer.transferred*]
lemmas *cimage-mono* = *image-mono*[*Transfer.transferred*]
lemmas *cinsert-mono* = *insert-mono*[*Transfer.transferred*]
lemmas *cunion-mono* = *Un-mono*[*Transfer.transferred*]
lemmas *cinter-mono* = *Int-mono*[*Transfer.transferred*]
lemmas *cminus-mono* = *Diff-mono*[*Transfer.transferred*]
lemmas *cin-mono* = *in-mono*[*Transfer.transferred*]
lemmas *cLeast-mono* = *Least-mono*[*Transfer.transferred*]
lemmas *cequalityI* = *equalityI*[*Transfer.transferred*]
lemmas *cUN-iff* [*simp*] = *UN-iff*[*Transfer.transferred*]
lemmas *cUN-I* [*intro*] = *UN-I*[*Transfer.transferred*]
lemmas *cUN-E* [*elim!*] = *UN-E*[*Transfer.transferred*]
lemmas *cUN-upper* = *UN-upper*[*Transfer.transferred*]
lemmas *cUN-least* = *UN-least*[*Transfer.transferred*]
lemmas *cUN-cinsert-distrib* = *UN-insert-distrib*[*Transfer.transferred*]
lemmas *cUN-empty* [*simp*] = *UN-empty*[*Transfer.transferred*]
lemmas *cUN-empty2* [*simp*] = *UN-empty2*[*Transfer.transferred*]
lemmas *cUN-absorb* = *UN-absorb*[*Transfer.transferred*]
lemmas *cUN-cinsert* [*simp*] = *UN-insert*[*Transfer.transferred*]
lemmas *cUN-cUn* [*simp*] = *UN-Un*[*Transfer.transferred*]
lemmas *cUN-cUN-flatten* = *UN-UN-flatten*[*Transfer.transferred*]
lemmas *cUN-csubset-iff* = *UN-subset-iff*[*Transfer.transferred*]
lemmas *cUN-constant* [*simp*] = *UN-constant*[*Transfer.transferred*]
lemmas *cimage-cUnion* = *image-Union*[*Transfer.transferred*]
lemmas *cUNION-cempty-conv* [*simp*] = *UNION-empty-conv*[*Transfer.transferred*]
lemmas *cBall-cUN* = *ball-UN*[*Transfer.transferred*]
lemmas *cBex-cUN* = *bex-UN*[*Transfer.transferred*]
lemmas *cUn-eq-cUN* = *Un-eq-UN*[*Transfer.transferred*]
lemmas *cUN-mono* = *UN-mono*[*Transfer.transferred*]
lemmas *cimage-cUN* = *image-UN*[*Transfer.transferred*]
lemmas *cUN-csingleton* [*simp*] = *UN-singleton*[*Transfer.transferred*]

22.3 Additional lemmas

22.3.1 *cempty*

lemma *cemptyE* [*elim*!]: *cin a cempty* $\Longrightarrow$ *P* ⟨*proof*⟩

22.3.2 *cinsert*

lemma *countable-insert-iff*: *countable* (*insert x A*) $\longleftrightarrow$ *countable A*
⟨*proof*⟩

lemma *set-cinsert*:
assumes *cin x A*
obtains *B* **where** *A* = *cinsert x B* **and** ¬ *cin x B*
⟨*proof*⟩

lemma *mk-disjoint-cinsert*: *cin a A* $\Longrightarrow$ ∃ *B*. *A* = *cinsert a B* ∧ ¬ *cin a B*
⟨*proof*⟩

22.3.3 *cimage*

lemma *subset-cimage-iff*: *csubset-eq B* (*cimage f A*) $\longleftrightarrow$ (∃ *AA*. *csubset-eq AA A* ∧ *B* = *cimage f AA*)
⟨*proof*⟩

22.3.4 bounded quantification

lemma *cBex-simps* [*simp*, *no-atp*]:
⋀*A P Q*. *cBex A* (λ*x*. *P x* ∧ *Q*) = (*cBex A P* ∧ *Q*)
⋀*A P Q*. *cBex A* (λ*x*. *P* ∧ *Q x*) = (*P* ∧ *cBex A Q*)
⋀*P*. *cBex cempty P* = *False*
⋀*a B P*. *cBex* (*cinsert a B*) *P* = (*P a* ∨ *cBex B P*)
⋀*A P f*. *cBex* (*cimage f A*) *P* = *cBex A* (λ*x*. *P* (*f x*))
⋀*A P*. (¬ *cBex A P*) = *cBall A* (λ*x*. ¬ *P x*)
⟨*proof*⟩

lemma *cBall-simps* [*simp*, *no-atp*]:
⋀*A P Q*. *cBall A* (λ*x*. *P x* ∨ *Q*) = (*cBall A P* ∨ *Q*)
⋀*A P Q*. *cBall A* (λ*x*. *P* ∨ *Q x*) = (*P* ∨ *cBall A Q*)
⋀*A P Q*. *cBall A* (λ*x*. *P* ⟶ *Q x*) = (*P* ⟶ *cBall A Q*)
⋀*A P Q*. *cBall A* (λ*x*. *P x* ⟶ *Q*) = (*cBex A P* ⟶ *Q*)
⋀*P*. *cBall cempty P* = *True*
⋀*a B P*. *cBall* (*cinsert a B*) *P* = (*P a* ∧ *cBall B P*)
⋀*A P f*. *cBall* (*cimage f A*) *P* = *cBall A* (λ*x*. *P* (*f x*))
⋀*A P*. (¬ *cBall A P*) = *cBex A* (λ*x*. ¬ *P x*)
⟨*proof*⟩

lemma *atomize-cBall*:
(⋀*x*. *cin x A* ==> *P x*) == *Trueprop* (*cBall A* (λ*x*. *P x*))
⟨*proof*⟩

22.3.5 *cUnion*

lemma *cUNION-cimage*: *cUNION* (*cimage f A*) *g* = *cUNION A* (*g* ∘ *f*)
including *cset.lifting* ⟨*proof*⟩

22.4 Setup for Lifting/Transfer

22.4.1 Relator and predicator properties

lift-definition *rel-cset* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a cset* ⇒ '*b cset* ⇒ *bool*
is *rel-set* **parametric** *rel-set-transfer* ⟨*proof*⟩

lemma *rel-cset-alt-def*:
rel-cset R a b ⟷
(∀ *t* ∈ *rcset a*. ∃ *u* ∈ *rcset b*. *R t u*) ∧
(∀ *t* ∈ *rcset b*. ∃ *u* ∈ *rcset a*. *R u t*)
⟨*proof*⟩

lemma *rel-cset-iff*:
rel-cset R a b ⟷
(∀ *t*. *cin t a* ⟶ (∃ *u*. *cin u b* ∧ *R t u*)) ∧
(∀ *t*. *cin t b* ⟶ (∃ *u*. *cin u a* ∧ *R u t*))
⟨*proof*⟩

lemma *rel-cset-cUNION*:
⟦ *rel-cset Q A B*; *rel-fun Q* (*rel-cset R*) *f g* ⟧
⟹ *rel-cset R* (*cUNION A f*) (*cUNION B g*)
⟨*proof*⟩

lemma *rel-cset-csingle-iff* [*simp*]: *rel-cset R* (*csingle x*) (*csingle y*) ⟷ *R x y*
⟨*proof*⟩

22.4.2 Transfer rules for the Transfer package

Unconditional transfer rules

context includes *lifting-syntax*
begin

lemmas *cempty-parametric* [*transfer-rule*] = *empty-transfer*[*Transfer.transferred*]

lemma *cinsert-parametric* [*transfer-rule*]:
(*A* ===> *rel-cset A* ===> *rel-cset A*) *cinsert cinsert*
⟨*proof*⟩

lemma *cUn-parametric* [*transfer-rule*]:
(*rel-cset A* ===> *rel-cset A* ===> *rel-cset A*) *cUn cUn*
⟨*proof*⟩

lemma *cUnion-parametric* [*transfer-rule*]:
(*rel-cset* (*rel-cset A*) ===> *rel-cset A*) *cUnion cUnion*

⟨*proof*⟩

lemma *cimage-parametric* [*transfer-rule*]:
((A ===> B) ===> *rel-cset* A ===> *rel-cset* B) *cimage cimage*
⟨*proof*⟩

lemma *cBall-parametric* [*transfer-rule*]:
(*rel-cset* A ===> (A ===> *op* =) ===> *op* =) *cBall cBall*
⟨*proof*⟩

lemma *cBex-parametric* [*transfer-rule*]:
(*rel-cset* A ===> (A ===> *op* =) ===> *op* =) *cBex cBex*
⟨*proof*⟩

lemma *rel-cset-parametric* [*transfer-rule*]:
((A ===> B ===> *op* =) ===> *rel-cset* A ===> *rel-cset* B ===> *op* =)
rel-cset rel-cset
⟨*proof*⟩

Rules requiring bi-unique, bi-total or right-total relations

lemma *cin-parametric* [*transfer-rule*]:
bi-unique A ⟹ (A ===> *rel-cset* A ===> *op* =) *cin cin*
⟨*proof*⟩

lemma *cInt-parametric* [*transfer-rule*]:
bi-unique A ⟹ (*rel-cset* A ===> *rel-cset* A ===> *rel-cset* A) *cInt cInt*
⟨*proof*⟩

lemma *cDiff-parametric* [*transfer-rule*]:
bi-unique A ⟹ (*rel-cset* A ===> *rel-cset* A ===> *rel-cset* A) *cDiff cDiff*
⟨*proof*⟩

lemma *csubset-parametric* [*transfer-rule*]:
bi-unique A ⟹ (*rel-cset* A ===> *rel-cset* A ===> *op* =) *csubset-eq csubset-eq*
⟨*proof*⟩

end

lifting-update *cset.lifting*
lifting-forget *cset.lifting*

22.5 Registration as BNF

lemma *card-of-countable-sets-range*:
fixes A :: *'a set*
shows $|\{X.\ X \subseteq A \wedge \mathit{countable}\ X \wedge X \neq \{\}\}| \leq o\ |\{f{::}\mathit{nat} \Rightarrow \mathit{'a}.\ \mathit{range}\ f \subseteq A\}|$
⟨*proof*⟩

lemma *card-of-countable-sets-Func*:
$|\{X.\ X \subseteq A \wedge \mathit{countable}\ X \wedge X \neq \{\}\}| \leq o\ |A|\ \hat{}c\ \mathit{natLeq}$

⟨*proof*⟩

lemma *ordLeq-countable-subsets*:
$|A| \leq o\ |\{X.\ X \subseteq A \wedge countable\ X\}|$
⟨*proof*⟩

lemma *finite-countable-subset*:
$finite\ \{X.\ X \subseteq A \wedge countable\ X\} \longleftrightarrow finite\ A$
⟨*proof*⟩

lemma *rcset-to-rcset*: $countable\ A \Longrightarrow rcset\ (the\text{-}inv\ rcset\ A) = A$
including *cset.lifting*
⟨*proof*⟩

lemma *Collect-Int-Times*: $\{(x,\ y).\ R\ x\ y\} \cap A \times B = \{(x,\ y).\ R\ x\ y \wedge x \in A \wedge y \in B\}$
⟨*proof*⟩

lemma *rel-cset-aux*:
$(\forall t \in rcset\ a.\ \exists u \in rcset\ b.\ R\ t\ u) \wedge (\forall t \in rcset\ b.\ \exists u \in rcset\ a.\ R\ u\ t) \longleftrightarrow$
$((BNF\text{-}Def.Grp\ \{x.\ rcset\ x \subseteq \{(a,\ b).\ R\ a\ b\}\}\ (cimage\ fst))^{-1-1}\ OO$
$BNF\text{-}Def.Grp\ \{x.\ rcset\ x \subseteq \{(a,\ b).\ R\ a\ b\}\}\ (cimage\ snd))\ a\ b$ (**is** *?L* = *?R*)
⟨*proof*⟩ **including** *cset.lifting*
⟨*proof*⟩

bnf *'a cset*
map: *cimage*
sets: *rcset*
bd: *natLeq*
wits: *cempty*
rel: *rel-cset*
⟨*proof*⟩ **including** *cset.lifting* ⟨*proof*⟩ **including** *cset.lifting* ⟨*proof*⟩
including *cset.lifting* ⟨*proof*⟩

end

23 Debugging facilities for code generated towards Isabelle/ML

theory *Debug*
imports *Main*
begin

context
begin

qualified definition *trace* :: *String.literal* ⇒ *unit* **where**

[*simp*]: *trace s* = ()

qualified definition *tracing* :: *String.literal* ⇒ *'a* ⇒ *'a* **where**
[*simp*]: *tracing s* = *id*

lemma [*code*]:
tracing s = (*let u* = *trace s in id*)
⟨*proof*⟩ **definition** *flush* :: *'a* ⇒ *unit* **where**
[*simp*]: *flush x* = ()

qualified definition *flushing* :: *'a* ⇒ *'b* ⇒ *'b* **where**
[*simp*]: *flushing x* = *id*

lemma [*code*, *code-unfold*]:
flushing x = (*let u* = *flush x in id*)
⟨*proof*⟩ **definition** *timing* :: *String.literal* ⇒ (*'a* ⇒ *'b*) ⇒ *'a* ⇒ *'b* **where**
[*simp*]: *timing s f x* = *f x*

end

code-printing
constant *Debug.trace* ⇀ (*Eval*) *Output.tracing*
| **constant** *Debug.flush* ⇀ (*Eval*) *Output.tracing*/ (@{*make'-string*} -) — note indirection via antiquotation
| **constant** *Debug.timing* ⇀ (*Eval*) *Timing.timeap'-msg*

code-reserved *Eval Output Timing*

end

24 Sequence of Properties on Subsequences

theory *Diagonal-Subsequence*
imports *Complex-Main*
begin

locale *subseqs* =
fixes *P*::*nat*⇒(*nat*⇒*nat*)⇒*bool*
assumes *ex-subseq*: ⋀*n s*. *strict-mono* (*s*::*nat*⇒*nat*) ⟹ ∃ *r'*. *strict-mono r'* ∧ *P n* (*s o r'*)
begin

definition *reduce* **where** *reduce s n* = (*SOME r'*::*nat*⇒*nat*. *strict-mono r'* ∧ *P n* (*s o r'*))

lemma *subseq-reduce*[*intro*, *simp*]:
strict-mono s ⟹ *strict-mono* (*reduce s n*)
⟨*proof*⟩

lemma *reduce-holds*:
strict-mono s $\Longrightarrow$ *P n* (*s o reduce s n*)
⟨*proof*⟩

primrec *seqseq* :: *nat* ⇒ *nat* ⇒ *nat* **where**
seqseq 0 = *id*
| *seqseq* (*Suc n*) = *seqseq n o reduce* (*seqseq n*) *n*

lemma *subseq-seqseq*[*intro*, *simp*]: *strict-mono* (*seqseq n*)
⟨*proof*⟩

lemma *seqseq-holds*:
P n (*seqseq* (*Suc n*))
⟨*proof*⟩

definition *diagseq* :: *nat* ⇒ *nat* **where** *diagseq i* = *seqseq i i*

lemma *diagseq-mono*: *diagseq n* < *diagseq* (*Suc n*)
⟨*proof*⟩

lemma *subseq-diagseq*: *strict-mono diagseq*
⟨*proof*⟩

primrec *fold-reduce* **where**
fold-reduce n 0 = *id*
| *fold-reduce n* (*Suc k*) = *fold-reduce n k o reduce* (*seqseq* (*n* + *k*)) (*n* + *k*)

lemma *subseq-fold-reduce*[*intro*, *simp*]: *strict-mono* (*fold-reduce n k*)
⟨*proof*⟩

lemma *ex-subseq-reduce-index*: *seqseq* (*n* + *k*) = *seqseq n o fold-reduce n k*
⟨*proof*⟩

lemma *seqseq-fold-reduce*: *seqseq n* = *fold-reduce 0 n*
⟨*proof*⟩

lemma *diagseq-fold-reduce*: *diagseq n* = *fold-reduce 0 n n*
⟨*proof*⟩

lemma *fold-reduce-add*: *fold-reduce 0* (*m* + *n*) = *fold-reduce 0 m o fold-reduce m n*
⟨*proof*⟩

lemma *diagseq-add*: *diagseq* (*k* + *n*) = (*seqseq k o* (*fold-reduce k n*)) (*k* + *n*)
⟨*proof*⟩

lemma *diagseq-sub*:
assumes *m* ≤ *n* **shows** *diagseq n* = (*seqseq m o* (*fold-reduce m* (*n* − *m*))) *n*
⟨*proof*⟩

lemma *subseq-diagonal-rest*: *strict-mono* ($\lambda x.$ *fold-reduce k x* ($k + x$))
⟨*proof*⟩

lemma *diagseq-seqseq*: *diagseq o* (*op* + *k*) = (*seqseq k o* ($\lambda x.$ *fold-reduce k x* ($k +$ x)))
⟨*proof*⟩

lemma *diagseq-holds*:
assumes *subseq-stable*: $\bigwedge r\ s\ n.$ *strict-mono r* $\Longrightarrow$ *P n s* $\Longrightarrow$ *P n* (*s o r*)
shows *P k* (*diagseq o* (*op* + (*Suc k*)))
⟨*proof*⟩

end

end

25 Partitions and Disjoint Sets

theory *Disjoint-Sets*
imports *Main*
begin

lemma *range-subsetD*: *range f* $\subseteq$ *B* $\Longrightarrow$ *f i* $\in$ *B*
⟨*proof*⟩

lemma *Int-Diff-disjoint*: $A \cap B \cap (A - B) = \{\}$
⟨*proof*⟩

lemma *Int-Diff-Un*: $A \cap B \cup (A - B) = A$
⟨*proof*⟩

lemma *mono-Un*: *mono A* $\Longrightarrow$ $(\bigcup i{\leq}n.\ A\ i) = A\ n$
⟨*proof*⟩

lemma *disjnt-equiv-class*: *equiv A r* $\Longrightarrow$ *disjnt* ($r``\{a\}$) ($r``\{b\}$) $\longleftrightarrow$ $(a, b) \notin r$
⟨*proof*⟩

25.1 Set of Disjoint Sets

abbreviation *disjoint* :: *'a set set* $\Rightarrow$ *bool* **where** *disjoint* $\equiv$ *pairwise disjnt*

lemma *disjoint-def*: *disjoint A* $\longleftrightarrow$ ($\forall\, a{\in}A.\ \forall\, b{\in}A.\ a \neq b \longrightarrow a \cap b = \{\}$)
⟨*proof*⟩

lemma *disjointI*:
($\bigwedge a\ b.\ a \in A \Longrightarrow b \in A \Longrightarrow a \neq b \Longrightarrow a \cap b = \{\}$) $\Longrightarrow$ *disjoint A*
⟨*proof*⟩

lemma *disjointD*:
disjoint A $\Longrightarrow$ *a* $\in$ *A* $\Longrightarrow$ *b* $\in$ *A* $\Longrightarrow$ *a* $\neq$ *b* $\Longrightarrow$ *a* $\cap$ *b* = {}
⟨*proof*⟩

lemma *disjoint-image*: *inj-on f* ($\bigcup$ *A*) $\Longrightarrow$ *disjoint A* $\Longrightarrow$ *disjoint* (*op* ‘ *f* ‘ *A*)
⟨*proof*⟩

lemma assumes *disjoint* (*A* $\cup$ *B*)
shows *disjoint-unionD1*: *disjoint A* **and** *disjoint-unionD2*: *disjoint B*
⟨*proof*⟩

lemma *disjoint-INT*:
assumes $*$: $\bigwedge$*i*. *i* $\in$ *I* $\Longrightarrow$ *disjoint* (*F i*)
shows *disjoint* {$\bigcap$ *i*$\in$*I*. *X i* | *X*. $\forall$ *i*$\in$*I*. *X i* $\in$ *F i*}
⟨*proof*⟩

25.1.1 Family of Disjoint Sets

definition *disjoint-family-on* :: ('*i* $\Rightarrow$ '*a set*) $\Rightarrow$ '*i set* $\Rightarrow$ *bool* **where**
disjoint-family-on A S $\longleftrightarrow$ ($\forall$ *m*$\in$*S*. $\forall$ *n*$\in$*S*. *m* $\neq$ *n* $\longrightarrow$ *A m* $\cap$ *A n* = {})

abbreviation *disjoint-family A* $\equiv$ *disjoint-family-on A UNIV*

lemma *disjoint-family-elem-disjnt*:
assumes *infinite A finite C*
and *df*: *disjoint-family-on B A*
obtains *x* **where** *x* $\in$ *A disjnt C* (*B x*)
⟨*proof*⟩

lemma *disjoint-family-onD*:
disjoint-family-on A I $\Longrightarrow$ *i* $\in$ *I* $\Longrightarrow$ *j* $\in$ *I* $\Longrightarrow$ *i* $\neq$ *j* $\Longrightarrow$ *A i* $\cap$ *A j* = {}
⟨*proof*⟩

lemma *disjoint-family-subset*: *disjoint-family A* $\Longrightarrow$ ($\bigwedge$*x*. *B x* $\subseteq$ *A x*) $\Longrightarrow$ *disjoint-family B*
⟨*proof*⟩

lemma *disjoint-family-on-bisimulation*:
assumes *disjoint-family-on f S*
and $\bigwedge$*n m*. *n* $\in$ *S* $\Longrightarrow$ *m* $\in$ *S* $\Longrightarrow$ *n* $\neq$ *m* $\Longrightarrow$ *f n* $\cap$ *f m* = {} $\Longrightarrow$ *g n* $\cap$ *g m* = {}
shows *disjoint-family-on g S*
⟨*proof*⟩

lemma *disjoint-family-on-mono*:
A $\subseteq$ *B* $\Longrightarrow$ *disjoint-family-on f B* $\Longrightarrow$ *disjoint-family-on f A*
⟨*proof*⟩

lemma *disjoint-family-Suc*:

$(\bigwedge n.\ A\ n \subseteq A\ (Suc\ n)) \Longrightarrow$ *disjoint-family* $(\lambda i.\ A\ (Suc\ i) - A\ i)$
⟨*proof*⟩

lemma *disjoint-family-on-disjoint-image*:
disjoint-family-on A I $\Longrightarrow$ *disjoint* $(A\ `\ I)$
⟨*proof*⟩

lemma *disjoint-family-on-vimageI*: *disjoint-family-on F I* $\Longrightarrow$ *disjoint-family-on* $(\lambda i.\ f\ -`\ F\ i)\ I$
⟨*proof*⟩

lemma *disjoint-image-disjoint-family-on*:
assumes *d*: *disjoint* $(A\ `\ I)$ **and** *i*: *inj-on A I*
shows *disjoint-family-on A I*
⟨*proof*⟩

lemma *disjoint-UN*:
assumes *F*: $\bigwedge i.\ i \in I \Longrightarrow$ *disjoint* $(F\ i)$ **and** $*$: *disjoint-family-on* $(\lambda i.\ \bigcup F\ i)\ I$
shows *disjoint* $(\bigcup i{\in}I.\ F\ i)$
⟨*proof*⟩

lemma *distinct-list-bind*:
assumes *distinct xs* $\bigwedge x.\ x \in set\ xs \Longrightarrow$ *distinct* $(f\ x)$
disjoint-family-on $(set \circ f)$ $(set\ xs)$
shows *distinct* $(List.bind\ xs\ f)$
⟨*proof*⟩

lemma *bij-betw-UNION-disjoint*:
assumes *disj*: *disjoint-family-on* $A'\ I$
assumes *bij*: $\bigwedge i.\ i \in I \Longrightarrow$ *bij-betw f* $(A\ i)$ $(A'\ i)$
shows *bij-betw f* $(\bigcup i{\in}I.\ A\ i)$ $(\bigcup i{\in}I.\ A'\ i)$
⟨*proof*⟩

lemma *disjoint-union*: *disjoint C* $\Longrightarrow$ *disjoint B* $\Longrightarrow \bigcup C \cap \bigcup B = \{\} \Longrightarrow$ *disjoint* $(C \cup B)$
⟨*proof*⟩

The union of an infinite disjoint family of non-empty sets is infinite.

lemma *infinite-disjoint-family-imp-infinite-UNION*:
assumes ¬*finite A* $\bigwedge x.\ x \in A \Longrightarrow f\ x \neq \{\}$ *disjoint-family-on f A*
shows ¬*finite* $(UNION\ A\ f)$
⟨*proof*⟩

25.2 Construct Disjoint Sequences

definition *disjointed* :: $(nat \Rightarrow 'a\ set) \Rightarrow nat \Rightarrow 'a\ set$ **where**
disjointed A n $= A\ n - (\bigcup i{\in}\{0..{<}n\}.\ A\ i)$

lemma *finite-UN-disjointed-eq*: $(\bigcup i{\in}\{0..{<}n\}.\ disjointed\ A\ i) = (\bigcup i{\in}\{0..{<}n\}.\ A\ i)$

⟨*proof*⟩

lemma *UN-disjointed-eq*: $(\bigcup i.\ \mathit{disjointed}\ A\ i) = (\bigcup i.\ A\ i)$
⟨*proof*⟩

lemma *less-disjoint-disjointed*: $m < n \Longrightarrow \mathit{disjointed}\ A\ m \cap \mathit{disjointed}\ A\ n = \{\}$
⟨*proof*⟩

lemma *disjoint-family-disjointed*: *disjoint-family* (*disjointed A*)
⟨*proof*⟩

lemma *disjointed-subset*: $\mathit{disjointed}\ A\ n \subseteq A\ n$
⟨*proof*⟩

lemma *disjointed-0*[*simp*]: $\mathit{disjointed}\ A\ 0 = A\ 0$
⟨*proof*⟩

lemma *disjointed-mono*: $\mathit{mono}\ A \Longrightarrow \mathit{disjointed}\ A\ (\mathit{Suc}\ n) = A\ (\mathit{Suc}\ n) - A\ n$
⟨*proof*⟩

25.3 Partitions

Partitions P of a set A. We explicitly disallow empty sets.

definition *partition-on* :: *'a set* ⇒ *'a set set* ⇒ *bool*
where
$\mathit{partition\text{-}on}\ A\ P \longleftrightarrow \bigcup P = A \wedge \mathit{disjoint}\ P \wedge \{\} \notin P$

lemma *partition-onI*:
$\bigcup P = A \Longrightarrow (\bigwedge p\ q.\ p \in P \Longrightarrow q \in P \Longrightarrow p \neq q \Longrightarrow \mathit{disjnt}\ p\ q) \Longrightarrow \{\} \notin P$
$\Longrightarrow \mathit{partition\text{-}on}\ A\ P$
⟨*proof*⟩

lemma *partition-onD1*: $\mathit{partition\text{-}on}\ A\ P \Longrightarrow A = \bigcup P$
⟨*proof*⟩

lemma *partition-onD2*: $\mathit{partition\text{-}on}\ A\ P \Longrightarrow \mathit{disjoint}\ P$
⟨*proof*⟩

lemma *partition-onD3*: $\mathit{partition\text{-}on}\ A\ P \Longrightarrow \{\} \notin P$
⟨*proof*⟩

25.4 Constructions of partitions

lemma *partition-on-empty*: $\mathit{partition\text{-}on}\ \{\}\ P \longleftrightarrow P = \{\}$
⟨*proof*⟩

lemma *partition-on-space*: $A \neq \{\} \Longrightarrow \mathit{partition\text{-}on}\ A\ \{A\}$
⟨*proof*⟩

lemma *partition-on-singletons*: *partition-on A* ((λ*x*. {*x*}) ‘ *A*)
⟨*proof*⟩

lemma *partition-on-transform*:
assumes *P*: *partition-on A P*
assumes *F-UN*: ⋃(*F* ‘ *P*) = *F* (⋃*P*) **and** *F-disjnt*: ⋀*p q*. *p* ∈ *P* ⟹ *q* ∈ *P* ⟹ *disjnt p q* ⟹ *disjnt* (*F p*) (*F q*)
shows *partition-on* (*F A*) (*F* ‘ *P* − {{}})
⟨*proof*⟩

lemma *partition-on-restrict*: *partition-on A P* ⟹ *partition-on* (*B* ∩ *A*) (*op* ∩ *B* ‘ *P* − {{}})
⟨*proof*⟩

lemma *partition-on-vimage*: *partition-on A P* ⟹ *partition-on* (*f* −‘ *A*) (*op* −‘ *f* ‘ *P* − {{}})
⟨*proof*⟩

lemma *partition-on-inj-image*:
assumes *P*: *partition-on A P* **and** *f*: *inj-on f A*
shows *partition-on* (*f* ‘ *A*) (*op* ‘ *f* ‘ *P* − {{}})
⟨*proof*⟩

25.5 Finiteness of partitions

lemma *finitely-many-partition-on*:
assumes *finite A*
shows *finite* {*P*. *partition-on A P*}
⟨*proof*⟩

lemma *finite-elements*: *finite A* ⟹ *partition-on A P* ⟹ *finite P*
⟨*proof*⟩

25.6 Equivalence of partitions and equivalence classes

lemma *partition-on-quotient*:
assumes *r*: *equiv A r*
shows *partition-on A* (*A* // *r*)
⟨*proof*⟩

lemma *equiv-partition-on*:
assumes *P*: *partition-on A P*
shows *equiv A* {(*x*, *y*). ∃*p* ∈ *P*. *x* ∈ *p* ∧ *y* ∈ *p*}
⟨*proof*⟩

lemma *partition-on-eq-quotient*:
assumes *P*: *partition-on A P*
shows *A* // {(*x*, *y*). ∃*p* ∈ *P*. *x* ∈ *p* ∧ *y* ∈ *p*} = *P*
⟨*proof*⟩

lemma *partition-on-alt*: *partition-on A P* $\longleftrightarrow$ $(\exists\, r.$ *equiv A r* $\wedge$ *P* $=$ *A* $//$ *r*)
⟨*proof*⟩

end

26 Lists with elements distinct as canonical example for datatype invariants

theory *Dlist*
imports *Main*
begin

26.1 The type of distinct lists

typedef *'a dlist* $=$ $\{$*xs*::*'a list*. *distinct xs*$\}$
morphisms *list-of-dlist Abs-dlist*
⟨*proof*⟩

setup-lifting *type-definition-dlist*

lemma *dlist-eq-iff*:
dxs $=$ *dys* $\longleftrightarrow$ *list-of-dlist dxs* $=$ *list-of-dlist dys*
⟨*proof*⟩

lemma *dlist-eqI*:
list-of-dlist dxs $=$ *list-of-dlist dys* $\Longrightarrow$ *dxs* $=$ *dys*
⟨*proof*⟩

Formal, totalized constructor for *'a dlist*:

definition *Dlist* :: *'a list* $\Rightarrow$ *'a dlist* **where**
Dlist xs $=$ *Abs-dlist* (*remdups xs*)

lemma *distinct-list-of-dlist* [*simp*, *intro*]:
distinct (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *list-of-dlist-Dlist* [*simp*]:
list-of-dlist (*Dlist xs*) $=$ *remdups xs*
⟨*proof*⟩

lemma *remdups-list-of-dlist* [*simp*]:
remdups (*list-of-dlist dxs*) $=$ *list-of-dlist dxs*
⟨*proof*⟩

lemma *Dlist-list-of-dlist* [*simp*, *code abstype*]:
Dlist (*list-of-dlist dxs*) $=$ *dxs*
⟨*proof*⟩

Fundamental operations:

context
begin

qualified definition *empty* :: *'a dlist* **where**
empty = *Dlist* []

qualified definition *insert* :: *'a* ⇒ *'a dlist* ⇒ *'a dlist* **where**
insert x dxs = *Dlist* (*List.insert x* (*list-of-dlist dxs*))

qualified definition *remove* :: *'a* ⇒ *'a dlist* ⇒ *'a dlist* **where**
remove x dxs = *Dlist* (*remove1 x* (*list-of-dlist dxs*))

qualified definition *map* :: (*'a* ⇒ *'b*) ⇒ *'a dlist* ⇒ *'b dlist* **where**
map f dxs = *Dlist* (*remdups* (*List.map f* (*list-of-dlist dxs*)))

qualified definition *filter* :: (*'a* ⇒ *bool*) ⇒ *'a dlist* ⇒ *'a dlist* **where**
filter P dxs = *Dlist* (*List.filter P* (*list-of-dlist dxs*))

qualified definition *rotate* :: *nat* ⇒ *'a dlist* ⇒ *'a dlist* **where**
rotate n dxs = *Dlist* (*List.rotate n* (*list-of-dlist dxs*))

end

Derived operations:

context
begin

qualified definition *null* :: *'a dlist* ⇒ *bool* **where**
null dxs = *List.null* (*list-of-dlist dxs*)

qualified definition *member* :: *'a dlist* ⇒ *'a* ⇒ *bool* **where**
member dxs = *List.member* (*list-of-dlist dxs*)

qualified definition *length* :: *'a dlist* ⇒ *nat* **where**
length dxs = *List.length* (*list-of-dlist dxs*)

qualified definition *fold* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'a dlist* ⇒ *'b* ⇒ *'b* **where**
fold f dxs = *List.fold f* (*list-of-dlist dxs*)

qualified definition *foldr* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'a dlist* ⇒ *'b* ⇒ *'b* **where**
foldr f dxs = *List.foldr f* (*list-of-dlist dxs*)

end

26.2 Executable version obeying invariant

lemma *list-of-dlist-empty* [*simp*, *code abstract*]:
list-of-dlist Dlist.empty = []
⟨*proof*⟩

lemma *list-of-dlist-insert* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.insert x dxs*) = *List.insert x* (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *list-of-dlist-remove* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.remove x dxs*) = *remove1 x* (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *list-of-dlist-map* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.map f dxs*) = *remdups* (*List.map f* (*list-of-dlist dxs*))
⟨*proof*⟩

lemma *list-of-dlist-filter* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.filter P dxs*) = *List.filter P* (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *list-of-dlist-rotate* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.rotate n dxs*) = *List.rotate n* (*list-of-dlist dxs*)
⟨*proof*⟩

Explicit executable conversion

definition *dlist-of-list* [*simp*]:
dlist-of-list = *Dlist*

lemma [*code abstract*]:
list-of-dlist (*dlist-of-list xs*) = *remdups xs*
⟨*proof*⟩

Equality

instantiation *dlist* :: (*equal*) *equal*
begin

definition *HOL.equal dxs dys* ⟷ *HOL.equal* (*list-of-dlist dxs*) (*list-of-dlist dys*)

instance
⟨*proof*⟩

end

declare *equal-dlist-def* [*code*]

lemma [*code nbe*]: *HOL.equal* (*dxs* :: 'a::*equal dlist*) *dxs* ⟷ *True*
⟨*proof*⟩

26.3 Induction principle and case distinction

lemma *dlist-induct* [*case-names empty insert*, *induct type*: *dlist*]:
assumes *empty*: *P Dlist.empty*
assumes *insrt*: ⋀*x dxs*. ¬ *Dlist.member dxs x* ⟹ *P dxs* ⟹ *P* (*Dlist.insert x dxs*)

shows *P dxs*
⟨*proof*⟩

lemma *dlist-case* [*cases type*: *dlist*]:
obtains (*empty*) *dxs* = *Dlist.empty*
| (*insert*) *x dys* **where** ¬ *Dlist.member dys x* **and** *dxs* = *Dlist.insert x dys*
⟨*proof*⟩

26.4 Functorial structure

functor *map*: *map*
⟨*proof*⟩

26.5 Quickcheck generators

quickcheck-generator *dlist predicate*: *distinct constructors*: *Dlist.empty*, *Dlist.insert*

26.6 BNF instance

context begin

qualified fun *wpull* :: ('*a* × '*b*) *list* ⇒ ('*b* × '*c*) *list* ⇒ ('*a* × '*c*) *list*
where
wpull [] *ys* = []
| *wpull xs* [] = []
| *wpull* ((*a*, *b*) # *xs*) ((*b*′, *c*) # *ys*) =
(*if b* ∈ *snd* ‘ *set xs then*
(*a*, *the* (*map-of* (*rev* ((*b*′, *c*) # *ys*)) *b*)) # *wpull xs* ((*b*′, *c*) # *ys*)
else if b′ ∈ *fst* ‘ *set ys then*
(*the* (*map-of* (*map prod.swap* (*rev* ((*a*, *b*) # *xs*))) *b*′), *c*) # *wpull* ((*a*, *b*) # *xs*) *ys*
else (*a*, *c*) # *wpull xs ys*)

qualified lemma *wpull-eq-Nil-iff* [*simp*]: *wpull xs ys* = [] ⟷ *xs* = [] ∨ *ys* = []
⟨*proof*⟩ **lemma** *wpull-induct*
[*consumes 1*,
case-names Nil left[*xs eq in-set IH*] *right*[*xs ys eq in-set IH*] *step*[*xs ys eq IH*]]:
assumes *eq*: *remdups* (*map snd xs*) = *remdups* (*map fst ys*)
and *Nil*: *P* [] []
and *left*: ⋀*a b xs b*′ *c ys*.
⟦ *b* ∈ *snd* ‘ *set xs*; *remdups* (*map snd xs*) = *remdups* (*map fst* ((*b*′, *c*) # *ys*));
(*b*, *the* (*map-of* (*rev* ((*b*′, *c*) # *ys*)) *b*)) ∈ *set* ((*b*′, *c*) # *ys*); *P xs* ((*b*′, *c*) # *ys*) ⟧
⟹ *P* ((*a*, *b*) # *xs*) ((*b*′, *c*) # *ys*)
and *right*: ⋀*a b xs b*′ *c ys*.
⟦ *b* ∉ *snd* ‘ *set xs*; *b*′ ∈ *fst* ‘ *set ys*;
remdups (*map snd* ((*a*, *b*) # *xs*)) = *remdups* (*map fst ys*);
(*the* (*map-of* (*map prod.swap* (*rev* ((*a*, *b*) #*xs*))) *b*′), *b*′) ∈ *set* ((*a*, *b*) # *xs*);
P ((*a*, *b*) # *xs*) *ys* ⟧
⟹ *P* ((*a*, *b*) # *xs*) ((*b*′, *c*) # *ys*)

and *step*: $\bigwedge a\ b\ xs\ c\ ys.$
$[\![\ b \notin snd\ `\ set\ xs;\ b \notin fst\ `\ set\ ys;\ remdups\ (map\ snd\ xs) = remdups\ (map\ fst\ ys);$
$P\ xs\ ys\ ]\!]$
$\Longrightarrow P\ ((a,\ b)\ \#\ xs)\ ((b,\ c)\ \#\ ys)$
shows *P xs ys*
⟨*proof*⟩ **lemma** *set-wpull-subset*:
assumes *remdups* (*map snd xs*) = *remdups* (*map fst ys*)
shows *set* (*wpull xs ys*) ⊆ *set xs O set ys*
⟨*proof*⟩ **lemma** *set-fst-wpull*:
assumes *remdups* (*map snd xs*) = *remdups* (*map fst ys*)
shows *fst* ` *set* (*wpull xs ys*) = *fst* ` *set xs*
⟨*proof*⟩ **lemma** *set-snd-wpull*:
assumes *remdups* (*map snd xs*) = *remdups* (*map fst ys*)
shows *snd* ` *set* (*wpull xs ys*) = *snd* ` *set ys*
⟨*proof*⟩ **lemma** *wpull*:
assumes *distinct xs*
and *distinct ys*
and *set xs* ⊆ {(*x*, *y*). *R x y*}
and *set ys* ⊆ {(*x*, *y*). *S x y*}
and *eq*: *remdups* (*map snd xs*) = *remdups* (*map fst ys*)
shows ∃ *zs*. *distinct zs* ∧ *set zs* ⊆ {(*x*, *y*). (*R OO S*) *x y*} ∧
remdups (*map fst zs*) = *remdups* (*map fst xs*) ∧ *remdups* (*map snd zs*) = *remdups* (*map snd ys*)
⟨*proof*⟩ **lift-definition** *set* :: *'a dlist* ⇒ *'a set* **is** *List.set* ⟨*proof*⟩ **lemma** *map-transfer* [*transfer-rule*]:
(*rel-fun op* = (*rel-fun* (*pcr-dlist op* =) (*pcr-dlist op* =))) (λ*f x*. *remdups* (*List.map f x*)) *Dlist.map*
⟨*proof*⟩

bnf *'a dlist*
map: *Dlist.map*
sets: *set*
bd: *natLeq*
wits: *Dlist.empty*
⟨*proof*⟩

lifting-update *dlist.lifting*
lifting-forget *dlist.lifting*

end

end

theory *Simps-Case-Conv*
imports *Main*
keywords
simps-of-case case-of-simps :: *thy-decl*

abbrevs
simps-of-case =
case-of-simps =
begin

⟨*ML*⟩

end

theory *Extended*
imports *Simps-Case-Conv*
begin

datatype *'a extended* = *Fin 'a* | *Pinf* (∞) | *Minf* (−∞)

instantiation *extended* :: (*order*)*order*
begin

fun *less-eq-extended* :: *'a extended* ⇒ *'a extended* ⇒ *bool* **where**
Fin x ≤ *Fin y* = (*x* ≤ *y*) |
- ≤ *Pinf* = *True* |
Minf ≤ - = *True* |
(-::*'a extended*) ≤ - = *False*

case-of-simps *less-eq-extended-case*: *less-eq-extended.simps*

definition *less-extended* :: *'a extended* ⇒ *'a extended* ⇒ *bool* **where**
((*x*::*'a extended*) < *y*) = (*x* ≤ *y* & ¬ *y* ≤ *x*)

instance
⟨*proof*⟩

end

instance *extended* :: (*linorder*)*linorder*
⟨*proof*⟩

lemma *Minf-le*[*simp*]: *Minf* ≤ *y*
⟨*proof*⟩
lemma *le-Pinf*[*simp*]: *x* ≤ *Pinf*
⟨*proof*⟩
lemma *le-Minf*[*simp*]: *x* ≤ *Minf* ⟷ *x* = *Minf*
⟨*proof*⟩
lemma *Pinf-le*[*simp*]: *Pinf* ≤ *x* ⟷ *x* = *Pinf*
⟨*proof*⟩

lemma *less-extended-simps*[*simp*]:

Fin x < Fin y = (x < y)
Fin x < Pinf = True
Fin x < Minf = False
Pinf < h = False
Minf < Fin x = True
Minf < Pinf = True
l < Minf = False
⟨*proof*⟩

lemma *min-extended-simps*[*simp*]:
min (Fin x) (Fin y) = Fin(min x y)
min xx Pinf = xx
min xx Minf = Minf
min Pinf yy = yy
min Minf yy = Minf
⟨*proof*⟩

lemma *max-extended-simps*[*simp*]:
max (Fin x) (Fin y) = Fin(max x y)
max xx Pinf = Pinf
max xx Minf = xx
max Pinf yy = Pinf
max Minf yy = yy
⟨*proof*⟩

instantiation *extended* :: (*zero*)*zero*
begin
definition *0 = Fin(0::'a)*
instance ⟨*proof*⟩
end

declare *zero-extended-def*[*symmetric*, *code-post*]

instantiation *extended* :: (*one*)*one*
begin
definition *1 = Fin(1::'a)*
instance ⟨*proof*⟩
end

declare *one-extended-def*[*symmetric*, *code-post*]

instantiation *extended* :: (*plus*)*plus*
begin

The following definition of of addition is totalized to make it asociative and commutative. Normally the sum of plus and minus infinity is undefined.

fun *plus-extended* **where**
Fin x + Fin y = Fin(x+y) |

Fin x + Pinf = Pinf |
Pinf + Fin x = Pinf |
Pinf + Pinf = Pinf |
Minf + Fin y = Minf |
Fin x + Minf = Minf |
Minf + Minf = Minf |
Minf + Pinf = Pinf |
Pinf + Minf = Pinf

case-of-simps *plus-case*: *plus-extended.simps*

instance ⟨*proof*⟩

end

instance *extended* :: (*ab-semigroup-add*)*ab-semigroup-add*
⟨*proof*⟩

instance *extended* :: (*ordered-ab-semigroup-add*)*ordered-ab-semigroup-add*
⟨*proof*⟩

instance *extended* :: (*comm-monoid-add*)*comm-monoid-add*
⟨*proof*⟩

instantiation *extended* :: (*uminus*)*uminus*
begin

fun *uminus-extended* **where**
− (Fin x) = Fin (− x) |
− Pinf = Minf |
− Minf = Pinf

instance ⟨*proof*⟩

end

instantiation *extended* :: (*ab-group-add*)*minus*
begin
definition *x − y = x + −*(*y*::*'a extended*)
instance ⟨*proof*⟩
end

lemma *minus-extended-simps*[*simp*]:
Fin x − Fin y = Fin(*x − y*)
Fin x − Pinf = Minf
Fin x − Minf = Pinf

Pinf − *Fin y* = *Pinf*
Pinf − *Minf* = *Pinf*
Minf − *Fin y* = *Minf*
Minf − *Pinf* = *Minf*
Minf − *Minf* = *Pinf*
Pinf − *Pinf* = *Pinf*
⟨*proof*⟩

Numerals:

instance *extended* :: ({*ab-semigroup-add*,*one*})*numeral* ⟨*proof*⟩

lemma *Fin-numeral*[*code-post*]: *Fin*(*numeral w*) = *numeral w*
⟨*proof*⟩

lemma *Fin-neg-numeral*[*code-post*]: *Fin* (− *numeral w*) = − *numeral w*
⟨*proof*⟩

instantiation *extended* :: (*lattice*)*bounded-lattice*
begin

definition *bot* = *Minf*
definition *top* = *Pinf*

fun *inf-extended* :: *'a extended* ⇒ *'a extended* ⇒ *'a extended* **where**
inf-extended (*Fin i*) (*Fin j*) = *Fin* (*inf i j*) |
inf-extended a Minf = *Minf* |
inf-extended Minf a = *Minf* |
inf-extended Pinf a = *a* |
inf-extended a Pinf = *a*

fun *sup-extended* :: *'a extended* ⇒ *'a extended* ⇒ *'a extended* **where**
sup-extended (*Fin i*) (*Fin j*) = *Fin* (*sup i j*) |
sup-extended a Pinf = *Pinf* |
sup-extended Pinf a = *Pinf* |
sup-extended Minf a = *a* |
sup-extended a Minf = *a*

case-of-simps *inf-extended-case*: *inf-extended.simps*
case-of-simps *sup-extended-case*: *sup-extended.simps*

instance
⟨*proof*⟩
end

end

27 Continuity and iterations

theory *Order-Continuity*
imports *Complex-Main Countable-Complete-Lattices*
begin

lemma *SUP-nat-binary*:
(*SUP n*::*nat*. *if n = 0 then A else B*) = (*sup A B*::'*a*::*countable-complete-lattice*)
⟨*proof*⟩

lemma *INF-nat-binary*:
(*INF n*::*nat*. *if n = 0 then A else B*) = (*inf A B*::'*a*::*countable-complete-lattice*)
⟨*proof*⟩

The name *continuous* is already taken in *Complex-Main*, so we use *sup-continuous* and *inf-continuous*. These names appear sometimes in literature and have the advantage that these names are duals.

named-theorems *order-continuous-intros*

27.1 Continuity for complete lattices

definition
sup-continuous :: ('*a*::*countable-complete-lattice* ⇒ '*b*::*countable-complete-lattice*) ⇒ *bool*
where
sup-continuous F ⟷ (∀ *M*::*nat* ⇒ '*a*. *mono M* ⟶ *F* (*SUP i*. *M i*) = (*SUP i*. *F* (*M i*)))

lemma *sup-continuousD*: *sup-continuous F* ⟹ *mono M* ⟹ *F* (*SUP i*::*nat*. *M i*) = (*SUP i*. *F* (*M i*))
⟨*proof*⟩

lemma *sup-continuous-mono*:
assumes [*simp*]: *sup-continuous F* **shows** *mono F*
⟨*proof*⟩

lemma [*order-continuous-intros*]:
shows *sup-continuous-const*: *sup-continuous* (λ*x*. *c*)
and *sup-continuous-id*: *sup-continuous* (λ*x*. *x*)
and *sup-continuous-apply*: *sup-continuous* (λ*f*. *f x*)
and *sup-continuous-fun*: (⋀*s*. *sup-continuous* (λ*x*. *P x s*)) ⟹ *sup-continuous P*
and *sup-continuous-If*: *sup-continuous F* ⟹ *sup-continuous G* ⟹ *sup-continuous* (λ*f*. *if C then F f else G f*)
⟨*proof*⟩

lemma *sup-continuous-compose*:

assumes f: *sup-continuous f* **and** g: *sup-continuous g*
shows *sup-continuous* $(\lambda x.\ f\ (g\ x))$
⟨*proof*⟩

lemma *sup-continuous-sup*[*order-continuous-intros*]:
sup-continuous f $\Longrightarrow$ *sup-continuous g* $\Longrightarrow$ *sup-continuous* $(\lambda x.\ sup\ (f\ x)\ (g\ x))$
⟨*proof*⟩

lemma *sup-continuous-inf*[*order-continuous-intros*]:
fixes $P\ Q :: 'a ::$ *countable-complete-lattice* $\Rightarrow 'b ::$ *countable-complete-distrib-lattice*
assumes P: *sup-continuous P* **and** Q: *sup-continuous Q*
shows *sup-continuous* $(\lambda x.\ inf\ (P\ x)\ (Q\ x))$
⟨*proof*⟩

lemma *sup-continuous-and*[*order-continuous-intros*]:
sup-continuous P $\Longrightarrow$ *sup-continuous Q* $\Longrightarrow$ *sup-continuous* $(\lambda x.\ P\ x \wedge Q\ x)$
⟨*proof*⟩

lemma *sup-continuous-or*[*order-continuous-intros*]:
sup-continuous P $\Longrightarrow$ *sup-continuous Q* $\Longrightarrow$ *sup-continuous* $(\lambda x.\ P\ x \vee Q\ x)$
⟨*proof*⟩

lemma *sup-continuous-lfp*:
assumes *sup-continuous F* **shows** *lfp F* $=$ $(SUP\ i.\ (F\ \hat{}\hat{}\ i)\ bot)$ (**is** *lfp F* $=$ *?U*)
⟨*proof*⟩

lemma *lfp-transfer-bounded*:
assumes P: *P bot* $\bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)$ $\bigwedge M.\ (\bigwedge i.\ P\ (M\ i)) \Longrightarrow P\ (SUP\ i{::}nat.\ M\ i)$
assumes α: $\bigwedge M.\ mono\ M \Longrightarrow (\bigwedge i{::}nat.\ P\ (M\ i)) \Longrightarrow \alpha\ (SUP\ i.\ M\ i) = (SUP\ i.\ \alpha\ (M\ i))$
assumes f: *sup-continuous f* **and** g: *sup-continuous g*
assumes [*simp*]: $\bigwedge x.\ P\ x \Longrightarrow x \leq lfp\ f \Longrightarrow \alpha\ (f\ x) = g\ (\alpha\ x)$
assumes *g-bound*: $\bigwedge x.\ \alpha\ bot \leq g\ x$
shows $\alpha\ (lfp\ f) = lfp\ g$
⟨*proof*⟩

lemma *lfp-transfer*:
sup-continuous α $\Longrightarrow$ *sup-continuous f* $\Longrightarrow$ *sup-continuous g* $\Longrightarrow$
$(\bigwedge x.\ \alpha\ bot \leq g\ x) \Longrightarrow (\bigwedge x.\ x \leq lfp\ f \Longrightarrow \alpha\ (f\ x) = g\ (\alpha\ x)) \Longrightarrow \alpha\ (lfp\ f) = lfp\ g$
⟨*proof*⟩

definition
inf-continuous :: $('a{::}$*countable-complete-lattice* $\Rightarrow 'b{::}$*countable-complete-lattice*$)$ $\Rightarrow$ *bool*
where
inf-continuous F $\longleftrightarrow$ $(\forall M{::}nat \Rightarrow 'a.\ antimono\ M \longrightarrow F\ (INF\ i.\ M\ i) = (INF\ i.\ F\ (M\ i)))$

lemma *inf-continuousD*: *inf-continuous F* $\Longrightarrow$ *antimono M* $\Longrightarrow$ *F* (*INF i*::*nat*. *M i*) = (*INF i*. *F* (*M i*))
⟨*proof*⟩

lemma *inf-continuous-mono*:
assumes [*simp*]: *inf-continuous F* **shows** *mono F*
⟨*proof*⟩

lemma [*order-continuous-intros*]:
shows *inf-continuous-const*: *inf-continuous* (λ*x*. *c*)
and *inf-continuous-id*: *inf-continuous* (λ*x*. *x*)
and *inf-continuous-apply*: *inf-continuous* (λ*f*. *f x*)
and *inf-continuous-fun*: (⋀*s*. *inf-continuous* (λ*x*. *P x s*)) $\Longrightarrow$ *inf-continuous P*
and *inf-continuous-If*: *inf-continuous F* $\Longrightarrow$ *inf-continuous G* $\Longrightarrow$ *inf-continuous* (λ*f*. *if C then F f else G f*)
⟨*proof*⟩

lemma *inf-continuous-inf*[*order-continuous-intros*]:
inf-continuous f $\Longrightarrow$ *inf-continuous g* $\Longrightarrow$ *inf-continuous* (λ*x*. *inf* (*f x*) (*g x*))
⟨*proof*⟩

lemma *inf-continuous-sup*[*order-continuous-intros*]:
fixes *P Q* :: *'a* :: *countable-complete-lattice* ⇒ *'b* :: *countable-complete-distrib-lattice*
assumes *P*: *inf-continuous P* **and** *Q*: *inf-continuous Q*
shows *inf-continuous* (λ*x*. *sup* (*P x*) (*Q x*))
⟨*proof*⟩

lemma *inf-continuous-and*[*order-continuous-intros*]:
inf-continuous P $\Longrightarrow$ *inf-continuous Q* $\Longrightarrow$ *inf-continuous* (λ*x*. *P x* ∧ *Q x*)
⟨*proof*⟩

lemma *inf-continuous-or*[*order-continuous-intros*]:
inf-continuous P $\Longrightarrow$ *inf-continuous Q* $\Longrightarrow$ *inf-continuous* (λ*x*. *P x* ∨ *Q x*)
⟨*proof*⟩

lemma *inf-continuous-compose*:
assumes *f*: *inf-continuous f* **and** *g*: *inf-continuous g*
shows *inf-continuous* (λ*x*. *f* (*g x*))
⟨*proof*⟩

lemma *inf-continuous-gfp*:
assumes *inf-continuous F* **shows** *gfp F* = (*INF i*. (*F* ^^ *i*) *top*) (**is** *gfp F* = *?U*)
⟨*proof*⟩

lemma *gfp-transfer*:
assumes *α*: *inf-continuous α* **and** *f*: *inf-continuous f* **and** *g*: *inf-continuous g*
assumes [*simp*]: *α top* = *top* ⋀*x*. *α* (*f x*) = *g* (*α x*)
shows *α* (*gfp f*) = *gfp g*

⟨*proof*⟩

lemma *gfp-transfer-bounded*:
assumes P: $P\ (f\ top)\ \bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)\ \bigwedge M.\ antimono\ M \Longrightarrow (\bigwedge i.\ P\ (M\ i)) \Longrightarrow P\ (INF\ i{::}nat.\ M\ i)$
assumes α: $\bigwedge M.\ antimono\ M \Longrightarrow (\bigwedge i{::}nat.\ P\ (M\ i)) \Longrightarrow \alpha\ (INF\ i.\ M\ i) = (INF\ i.\ \alpha\ (M\ i))$
assumes f: *inf-continuous* f **and** g: *inf-continuous* g
assumes [*simp*]: $\bigwedge x.\ P\ x \Longrightarrow \alpha\ (f\ x) = g\ (\alpha\ x)$
assumes *g-bound*: $\bigwedge x.\ g\ x \leq \alpha\ (f\ top)$
shows $\alpha\ (gfp\ f) = gfp\ g$
⟨*proof*⟩

27.1.1 Least fixed points in countable complete lattices

definition (**in** *countable-complete-lattice*) *cclfp* :: $('a \Rightarrow 'a) \Rightarrow 'a$
where $cclfp\ f = (SUP\ i.\ (f\ \hat{}\hat{}\ i)\ bot)$

lemma *cclfp-unfold*:
assumes *sup-continuous* F **shows** $cclfp\ F = F\ (cclfp\ F)$
⟨*proof*⟩

lemma *cclfp-lowerbound*: **assumes** f: *mono* f **and** A: $f\ A \leq A$ **shows** $cclfp\ f \leq A$
⟨*proof*⟩

lemma *cclfp-transfer*:
assumes *sup-continuous* α *mono* f
assumes $\alpha\ bot = bot\ \bigwedge x.\ \alpha\ (f\ x) = g\ (\alpha\ x)$
shows $\alpha\ (cclfp\ f) = cclfp\ g$
⟨*proof*⟩

end

28 Extended natural numbers (i.e. with infinity)

theory *Extended-Nat*
imports *Main Countable Order-Continuity*
begin

class *infinity* =
fixes *infinity* :: $'a$ (∞)

context
fixes f :: $nat \Rightarrow 'a{::}\{canonically\text{-}ordered\text{-}monoid\text{-}add, linorder\text{-}topology, complete\text{-}linorder\}$
begin

lemma *sums-SUP*[*simp*, *intro*]: $f\ sums\ (SUP\ n.\ \sum i{<}n.\ f\ i)$
⟨*proof*⟩

lemma *suminf-eq-SUP*: $suminf\ f = (SUP\ n.\ \sum i{<}n.\ f\ i)$
⟨*proof*⟩

end

28.1 Type definition

We extend the standard natural numbers by a special value indicating infinity.

typedef *enat* = *UNIV* :: *nat option set* ⟨*proof*⟩

TODO: introduce enat as coinductive datatype, enat is just *of-nat*

definition *enat* :: *nat* ⇒ *enat* **where**
enat n = *Abs-enat* (*Some n*)

instantiation *enat* :: *infinity*
begin

definition ∞ = *Abs-enat None*
instance ⟨*proof*⟩

end

instance *enat* :: *countable*
⟨*proof*⟩

old-rep-datatype *enat* ∞ :: *enat*
⟨*proof*⟩

declare [[*coercion enat*::*nat*⇒*enat*]]

lemmas *enat2-cases* = *enat.exhaust*[*case-product enat.exhaust*]
lemmas *enat3-cases* = *enat.exhaust*[*case-product enat.exhaust enat.exhaust*]

lemma *not-infinity-eq* [*iff*]: $(x \neq \infty) = (\exists\, i.\ x = enat\ i)$
⟨*proof*⟩

lemma *not-enat-eq* [*iff*]: $(\forall\, y.\ x \neq enat\ y) = (x = \infty)$
⟨*proof*⟩

lemma *enat-ex-split*: $(\exists\, c{::}enat.\ P\ c) \longleftrightarrow P\ \infty \vee (\exists\, c{::}nat.\ P\ c)$
⟨*proof*⟩

primrec *the-enat* :: *enat* ⇒ *nat*
where *the-enat* (*enat n*) = *n*

28.2 Constructors and numbers

instantiation *enat* :: *zero-neq-one*
begin

definition
0 = enat 0

definition
1 = enat 1

instance
⟨*proof*⟩

end

definition *eSuc* :: *enat* ⇒ *enat* **where**
eSuc i = (*case i of enat n* ⇒ *enat* (*Suc n*) | ∞ ⇒ ∞)

lemma *enat-0* [*code-post*]: *enat 0 = 0*
⟨*proof*⟩

lemma *enat-1* [*code-post*]: *enat 1 = 1*
⟨*proof*⟩

lemma *enat-0-iff*: *enat x = 0* ⟷ *x = 0 0 = enat x* ⟷ *x = 0*
⟨*proof*⟩

lemma *enat-1-iff*: *enat x = 1* ⟷ *x = 1 1 = enat x* ⟷ *x = 1*
⟨*proof*⟩

lemma *one-eSuc*: *1 = eSuc 0*
⟨*proof*⟩

lemma *infinity-ne-i0* [*simp*]: (∞::*enat*) ≠ *0*
⟨*proof*⟩

lemma *i0-ne-infinity* [*simp*]: *0* ≠ (∞::*enat*)
⟨*proof*⟩

lemma *zero-one-enat-neq*:
¬ *0* = (*1*::*enat*)
¬ *1* = (*0*::*enat*)
⟨*proof*⟩

lemma *infinity-ne-i1* [*simp*]: (∞::*enat*) ≠ *1*
⟨*proof*⟩

lemma *i1-ne-infinity* [*simp*]: *1* ≠ (∞::*enat*)
⟨*proof*⟩

lemma *eSuc-enat*: *eSuc* (*enat n*) = *enat* (*Suc n*)
⟨*proof*⟩

lemma *eSuc-infinity* [*simp*]: *eSuc* ∞ = ∞
⟨*proof*⟩

lemma *eSuc-ne-0* [*simp*]: *eSuc n* ≠ *0*
⟨*proof*⟩

lemma *zero-ne-eSuc* [*simp*]: *0* ≠ *eSuc n*
⟨*proof*⟩

lemma *eSuc-inject* [*simp*]: *eSuc m* = *eSuc n* ⟷ *m* = *n*
⟨*proof*⟩

lemma *eSuc-enat-iff*: *eSuc x* = *enat y* ⟷ (∃ *n*. *y* = *Suc n* ∧ *x* = *enat n*)
⟨*proof*⟩

lemma *enat-eSuc-iff*: *enat y* = *eSuc x* ⟷ (∃ *n*. *y* = *Suc n* ∧ *enat n* = *x*)
⟨*proof*⟩

28.3 Addition

instantiation *enat* :: *comm-monoid-add*
begin

definition [*nitpick-simp*]:
m + *n* = (*case m of* ∞ ⇒ ∞ | *enat m* ⇒ (*case n of* ∞ ⇒ ∞ | *enat n* ⇒ *enat* (*m* + *n*)))

lemma *plus-enat-simps* [*simp*, *code*]:
fixes *q* :: *enat*
shows *enat m* + *enat n* = *enat* (*m* + *n*)
and ∞ + *q* = ∞
and *q* + ∞ = ∞
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *eSuc-plus-1*:
eSuc n = *n* + *1*
⟨*proof*⟩

lemma *plus-1-eSuc*:
1 + *q* = *eSuc q*

$q + 1 = eSuc\ q$
⟨*proof*⟩

lemma *iadd-Suc*: $eSuc\ m + n = eSuc\ (m + n)$
⟨*proof*⟩

lemma *iadd-Suc-right*: $m + eSuc\ n = eSuc\ (m + n)$
⟨*proof*⟩

28.4 Multiplication

instantiation *enat* :: {*comm-semiring-1*, *semiring-no-zero-divisors*}
begin

definition *times-enat-def* [*nitpick-simp*]:
$m * n = (case\ m\ of\ \infty \Rightarrow if\ n = 0\ then\ 0\ else\ \infty \mid enat\ m \Rightarrow$
$(case\ n\ of\ \infty \Rightarrow if\ m = 0\ then\ 0\ else\ \infty \mid enat\ n \Rightarrow enat\ (m * n)))$

lemma *times-enat-simps* [*simp*, *code*]:
$enat\ m * enat\ n = enat\ (m * n)$
$\infty * \infty = (\infty::enat)$
$\infty * enat\ n = (if\ n = 0\ then\ 0\ else\ \infty)$
$enat\ m * \infty = (if\ m = 0\ then\ 0\ else\ \infty)$
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *mult-eSuc*: $eSuc\ m * n = n + m * n$
⟨*proof*⟩

lemma *mult-eSuc-right*: $m * eSuc\ n = m + m * n$
⟨*proof*⟩

lemma *of-nat-eq-enat*: $of\text{-}nat\ n = enat\ n$
⟨*proof*⟩

instance *enat* :: *semiring-char-0*
⟨*proof*⟩

lemma *imult-is-infinity*: $((a::enat) * b = \infty) = (a = \infty \wedge b \neq 0 \vee b = \infty \wedge a \neq 0)$
⟨*proof*⟩

28.5 Numerals

lemma *numeral-eq-enat*:
$numeral\ k = enat\ (numeral\ k)$

⟨*proof*⟩

lemma *enat-numeral* [*code-abbrev*]:
enat (*numeral k*) = *numeral k*
⟨*proof*⟩

lemma *infinity-ne-numeral* [*simp*]: (∞::*enat*) ≠ *numeral k*
⟨*proof*⟩

lemma *numeral-ne-infinity* [*simp*]: *numeral k* ≠ (∞::*enat*)
⟨*proof*⟩

lemma *eSuc-numeral* [*simp*]: *eSuc* (*numeral k*) = *numeral* (*k* + *Num.One*)
⟨*proof*⟩

28.6 Subtraction

instantiation *enat* :: *minus*
begin

definition *diff-enat-def*:
a − *b* = (*case a of* (*enat x*) ⇒ (*case b of* (*enat y*) ⇒ *enat* (*x* − *y*) | ∞ ⇒ *0*)
| ∞ ⇒ ∞)

instance ⟨*proof*⟩

end

lemma *idiff-enat-enat* [*simp*, *code*]: *enat a* − *enat b* = *enat* (*a* − *b*)
⟨*proof*⟩

lemma *idiff-infinity* [*simp*, *code*]: ∞ − *n* = (∞::*enat*)
⟨*proof*⟩

lemma *idiff-infinity-right* [*simp*, *code*]: *enat a* − ∞ = *0*
⟨*proof*⟩

lemma *idiff-0* [*simp*]: (*0*::*enat*) − *n* = *0*
⟨*proof*⟩

lemmas *idiff-enat-0* [*simp*] = *idiff-0* [*unfolded zero-enat-def*]

lemma *idiff-0-right* [*simp*]: (*n*::*enat*) − *0* = *n*
⟨*proof*⟩

lemmas *idiff-enat-0-right* [*simp*] = *idiff-0-right* [*unfolded zero-enat-def*]

lemma *idiff-self* [*simp*]: *n* ≠ ∞ ⟹ (*n*::*enat*) − *n* = *0*
⟨*proof*⟩

lemma *eSuc-minus-eSuc* [*simp*]: $eSuc\ n - eSuc\ m = n - m$
⟨*proof*⟩

lemma *eSuc-minus-1* [*simp*]: $eSuc\ n - 1 = n$
⟨*proof*⟩

28.7 Ordering

instantiation *enat* :: *linordered-ab-semigroup-add*
begin

definition [*nitpick-simp*]:
$m \leq n = (case\ n\ of\ enat\ n1 \Rightarrow (case\ m\ of\ enat\ m1 \Rightarrow m1 \leq n1 \mid \infty \Rightarrow True) \mid \infty \Rightarrow False)$

definition [*nitpick-simp*]:
$m < n = (case\ m\ of\ enat\ m1 \Rightarrow (case\ n\ of\ enat\ n1 \Rightarrow m1 < n1 \mid \infty \Rightarrow False) \mid \infty \Rightarrow True)$

lemma *enat-ord-simps* [*simp*]:
$enat\ m \leq enat\ n \longleftrightarrow m \leq n$
$enat\ m < enat\ n \longleftrightarrow m < n$
$q \leq (\infty::enat)$
$q < (\infty::enat) \longleftrightarrow q \neq \infty$
$(\infty::enat) \leq q \longleftrightarrow q = \infty$
$(\infty::enat) < q \longleftrightarrow False$
⟨*proof*⟩

lemma *numeral-le-enat-iff* [*simp*]:
shows $numeral\ m \leq enat\ n \longleftrightarrow numeral\ m \leq n$
⟨*proof*⟩

lemma *numeral-less-enat-iff* [*simp*]:
shows $numeral\ m < enat\ n \longleftrightarrow numeral\ m < n$
⟨*proof*⟩

lemma *enat-ord-code* [*code*]:
$enat\ m \leq enat\ n \longleftrightarrow m \leq n$
$enat\ m < enat\ n \longleftrightarrow m < n$
$q \leq (\infty::enat) \longleftrightarrow True$
$enat\ m < \infty \longleftrightarrow True$
$\infty \leq enat\ n \longleftrightarrow False$
$(\infty::enat) < q \longleftrightarrow False$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *enat* :: *dioid*
⟨*proof*⟩

instance *enat* :: {*linordered-nonzero-semiring*, *strict-ordered-comm-monoid-add*}
⟨*proof*⟩

lemma *enat-ord-number* [*simp*]:
(*numeral m* :: *enat*) ≤ *numeral n* ⟷ (*numeral m* :: *nat*) ≤ *numeral n*
(*numeral m* :: *enat*) < *numeral n* ⟷ (*numeral m* :: *nat*) < *numeral n*
⟨*proof*⟩

lemma *infinity-ileE* [*elim*!]: ∞ ≤ *enat m* ⟹ *R*
⟨*proof*⟩

lemma *infinity-ilessE* [*elim*!]: ∞ < *enat m* ⟹ *R*
⟨*proof*⟩

lemma *eSuc-ile-mono* [*simp*]: *eSuc n* ≤ *eSuc m* ⟷ *n* ≤ *m*
⟨*proof*⟩

lemma *eSuc-mono* [*simp*]: *eSuc n* < *eSuc m* ⟷ *n* < *m*
⟨*proof*⟩

lemma *ile-eSuc* [*simp*]: *n* ≤ *eSuc n*
⟨*proof*⟩

lemma *not-eSuc-ilei0* [*simp*]: ¬ *eSuc n* ≤ *0*
⟨*proof*⟩

lemma *i0-iless-eSuc* [*simp*]: *0* < *eSuc n*
⟨*proof*⟩

lemma *iless-eSuc0*[*simp*]: (*n* < *eSuc 0*) = (*n* = *0*)
⟨*proof*⟩

lemma *ileI1*: *m* < *n* ⟹ *eSuc m* ≤ *n*
⟨*proof*⟩

lemma *Suc-ile-eq*: *enat* (*Suc m*) ≤ *n* ⟷ *enat m* < *n*
⟨*proof*⟩

lemma *iless-Suc-eq* [*simp*]: *enat m* < *eSuc n* ⟷ *enat m* ≤ *n*
⟨*proof*⟩

lemma *imult-infinity*: (*0*::*enat*) < *n* ⟹ ∞ ∗ *n* = ∞

⟨*proof*⟩

lemma *imult-infinity-right*: $(0::enat) < n \Longrightarrow n * \infty = \infty$
⟨*proof*⟩

lemma *enat-0-less-mult-iff*: $(0 < (m::enat) * n) = (0 < m \wedge 0 < n)$
⟨*proof*⟩

lemma *mono-eSuc*: *mono eSuc*
⟨*proof*⟩

lemma *min-enat-simps* [*simp*]:
$min\ (enat\ m)\ (enat\ n) = enat\ (min\ m\ n)$
$min\ q\ 0 = 0$
$min\ 0\ q = 0$
$min\ q\ (\infty::enat) = q$
$min\ (\infty::enat)\ q = q$
⟨*proof*⟩

lemma *max-enat-simps* [*simp*]:
$max\ (enat\ m)\ (enat\ n) = enat\ (max\ m\ n)$
$max\ q\ 0 = q$
$max\ 0\ q = q$
$max\ q\ \infty = (\infty::enat)$
$max\ \infty\ q = (\infty::enat)$
⟨*proof*⟩

lemma *enat-ile*: $n \leq enat\ m \Longrightarrow \exists\, k.\ n = enat\ k$
⟨*proof*⟩

lemma *enat-iless*: $n < enat\ m \Longrightarrow \exists\, k.\ n = enat\ k$
⟨*proof*⟩

lemma *iadd-le-enat-iff*:
$x + y \leq enat\ n \longleftrightarrow (\exists\, y'\ x'.\ x = enat\ x' \wedge y = enat\ y' \wedge x' + y' \leq n)$
⟨*proof*⟩

lemma *chain-incr*: $\forall\, i.\ \exists\, j.\ Y\ i < Y\ j \Longrightarrow \exists\, j.\ enat\ k < Y\ j$
⟨*proof*⟩

lemma *eSuc-max*: $eSuc\ (max\ x\ y) = max\ (eSuc\ x)\ (eSuc\ y)$
⟨*proof*⟩

lemma *eSuc-Max*:
assumes *finite A* $A \neq \{\}$
shows $eSuc\ (Max\ A) = Max\ (eSuc\ `\ A)$
⟨*proof*⟩

instantiation *enat* :: {*order-bot*, *order-top*}

begin

definition *bot-enat* :: *enat* **where** *bot-enat* = *0*
definition *top-enat* :: *enat* **where** *top-enat* = ∞

instance
⟨*proof*⟩

end

lemma *finite-enat-bounded*:
assumes *le-fin*: $\bigwedge y.\ y \in A \Longrightarrow y \leq enat\ n$
shows *finite A*
⟨*proof*⟩

28.8 Cancellation simprocs

lemma *enat-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty::enat) \vee b = c$
⟨*proof*⟩

lemma *enat-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty::enat) \vee b \leq c$
⟨*proof*⟩

lemma *enat-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty::enat) \wedge b < c$
⟨*proof*⟩

⟨*ML*⟩

TODO: add regression tests for these simprocs

TODO: add simprocs for combining and cancelling numerals

28.9 Well-ordering

lemma *less-enatE*:
[| $n < enat\ m$; !!k. $n = enat\ k$ ==> $k < m$ ==> P |] ==> P
⟨*proof*⟩

lemma *less-infinityE*:
[| $n < \infty$; !!k. $n = enat\ k$ ==> P |] ==> P
⟨*proof*⟩

lemma *enat-less-induct*:
assumes *prem*: !!n. $\forall m::enat.\ m < n$ −−> $P\ m$ ==> $P\ n$ **shows** $P\ n$
⟨*proof*⟩

instance *enat* :: *wellorder*
⟨*proof*⟩

28.10 Complete Lattice

instantiation *enat* :: *complete-lattice*
begin

definition *inf-enat* :: *enat* ⇒ *enat* ⇒ *enat* **where**
inf-enat = *min*

definition *sup-enat* :: *enat* ⇒ *enat* ⇒ *enat* **where**
sup-enat = *max*

definition *Inf-enat* :: *enat set* ⇒ *enat* **where**
Inf-enat A = (*if A* = {} *then* ∞ *else* (*LEAST x*. *x* ∈ *A*))

definition *Sup-enat* :: *enat set* ⇒ *enat* **where**
Sup-enat A = (*if A* = {} *then 0 else if finite A then Max A else* ∞)
instance
⟨*proof*⟩
end

instance *enat* :: *complete-linorder* ⟨*proof*⟩

lemma *eSuc-Sup*: *A* ≠ {} ⟹ *eSuc* (*Sup A*) = *Sup* (*eSuc* ‘ *A*)
⟨*proof*⟩

lemma *sup-continuous-eSuc*: *sup-continuous f* ⟹ *sup-continuous* (λ*x*. *eSuc* (*f x*))
⟨*proof*⟩

28.11 Traditional theorem names

lemmas *enat-defs* = *zero-enat-def one-enat-def eSuc-def*
plus-enat-def less-eq-enat-def less-enat-def

lemma *iadd-is-0*: (*m* + *n* = (*0*::*enat*)) = (*m* = *0* ∧ *n* = *0*)
⟨*proof*⟩

lemma *i0-lb* : (*0*::*enat*) ≤ *n*
⟨*proof*⟩

lemma *ile0-eq*: *n* ≤ (*0*::*enat*) ⟷ *n* = *0*
⟨*proof*⟩

lemma *not-iless0*: ¬ *n* < (*0*::*enat*)
⟨*proof*⟩

lemma *i0-less*[*simp*]: (*0*::*enat*) < *n* ⟷ *n* ≠ *0*
⟨*proof*⟩

lemma *imult-is-0*: ((*m*::*enat*) ∗ *n* = *0*) = (*m* = *0* ∨ *n* = *0*)

⟨*proof*⟩

end

29 Liminf and Limsup on conditionally complete lattices

theory *Liminf-Limsup*
imports *Complex-Main*
begin

lemma (**in** *conditionally-complete-linorder*) *le-cSup-iff*:
assumes $A \neq \{\}$ *bdd-above* A
shows $x \leq Sup\ A \longleftrightarrow (\forall y{<}x.\ \exists a{\in}A.\ y < a)$
⟨*proof*⟩

lemma (**in** *conditionally-complete-linorder*) *le-cSUP-iff*:
$A \neq \{\} \Longrightarrow$ *bdd-above* $(f`A) \Longrightarrow x \leq SUPREMUM\ A\ f \longleftrightarrow (\forall y{<}x.\ \exists i{\in}A.\ y < f\ i)$
⟨*proof*⟩

lemma *le-cSup-iff-less*:
fixes $x :: 'a :: \{conditionally\text{-}complete\text{-}linorder, dense\text{-}linorder\}$
shows $A \neq \{\} \Longrightarrow$ *bdd-above* $(f`A) \Longrightarrow x \leq (SUP\ i{:}A.\ f\ i) \longleftrightarrow (\forall y{<}x.\ \exists i{\in}A.\ y \leq f\ i)$
⟨*proof*⟩

lemma *le-Sup-iff-less*:
fixes $x :: 'a :: \{complete\text{-}linorder, dense\text{-}linorder\}$
shows $x \leq (SUP\ i{:}A.\ f\ i) \longleftrightarrow (\forall y{<}x.\ \exists i{\in}A.\ y \leq f\ i)$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma (**in** *conditionally-complete-linorder*) *cInf-le-iff*:
assumes $A \neq \{\}$ *bdd-below* A
shows $Inf\ A \leq x \longleftrightarrow (\forall y{>}x.\ \exists a{\in}A.\ y > a)$
⟨*proof*⟩

lemma (**in** *conditionally-complete-linorder*) *cINF-le-iff*:
$A \neq \{\} \Longrightarrow$ *bdd-below* $(f`A) \Longrightarrow INFIMUM\ A\ f \leq x \longleftrightarrow (\forall y{>}x.\ \exists i{\in}A.\ y > f\ i)$
⟨*proof*⟩

lemma *cInf-le-iff-less*:
fixes $x :: 'a :: \{conditionally\text{-}complete\text{-}linorder, dense\text{-}linorder\}$
shows $A \neq \{\} \Longrightarrow$ *bdd-below* $(f`A) \Longrightarrow (INF\ i{:}A.\ f\ i) \leq x \longleftrightarrow (\forall y{>}x.\ \exists i{\in}A.\ f\ i \leq y)$
⟨*proof*⟩

lemma *Inf-le-iff-less*:
fixes $x :: 'a :: \{complete\text{-}linorder, dense\text{-}linorder\}$
shows $(INF\ i{:}A.\ f\ i) \leq x \longleftrightarrow (\forall y{>}x.\ \exists i{\in}A.\ f\ i \leq y)$
⟨*proof*⟩

lemma *SUP-pair*:
fixes $f :: - \Rightarrow - \Rightarrow - :: complete\text{-}lattice$
shows $(SUP\ i : A.\ SUP\ j : B.\ f\ i\ j) = (SUP\ p : A \times B.\ f\ (fst\ p)\ (snd\ p))$
⟨*proof*⟩

lemma *INF-pair*:
fixes $f :: - \Rightarrow - \Rightarrow - :: complete\text{-}lattice$
shows $(INF\ i : A.\ INF\ j : B.\ f\ i\ j) = (INF\ p : A \times B.\ f\ (fst\ p)\ (snd\ p))$
⟨*proof*⟩

29.0.1 *Liminf* and *Limsup*

definition $Liminf :: 'a\ filter \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b :: complete\text{-}lattice$ **where**
$Liminf\ F\ f = (SUP\ P{:}\{P.\ eventually\ P\ F\}.\ INF\ x{:}\{x.\ P\ x\}.\ f\ x)$

definition $Limsup :: 'a\ filter \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b :: complete\text{-}lattice$ **where**
$Limsup\ F\ f = (INF\ P{:}\{P.\ eventually\ P\ F\}.\ SUP\ x{:}\{x.\ P\ x\}.\ f\ x)$

abbreviation $liminf \equiv Liminf\ sequentially$

abbreviation $limsup \equiv Limsup\ sequentially$

lemma *Liminf-eqI*:
$(\bigwedge P.\ eventually\ P\ F \Longrightarrow INFIMUM\ (Collect\ P)\ f \leq x) \Longrightarrow$
$(\bigwedge y.\ (\bigwedge P.\ eventually\ P\ F \Longrightarrow INFIMUM\ (Collect\ P)\ f \leq y) \Longrightarrow x \leq y) \Longrightarrow$
$Liminf\ F\ f = x$
⟨*proof*⟩

lemma *Limsup-eqI*:
$(\bigwedge P.\ eventually\ P\ F \Longrightarrow x \leq SUPREMUM\ (Collect\ P)\ f) \Longrightarrow$
$(\bigwedge y.\ (\bigwedge P.\ eventually\ P\ F \Longrightarrow y \leq SUPREMUM\ (Collect\ P)\ f) \Longrightarrow y \leq x)$
$\Longrightarrow Limsup\ F\ f = x$
⟨*proof*⟩

lemma *liminf-SUP-INF*: $liminf\ f = (SUP\ n.\ INF\ m{:}\{n..\}.\ f\ m)$
⟨*proof*⟩

lemma *limsup-INF-SUP*: $limsup\ f = (INF\ n.\ SUP\ m{:}\{n..\}.\ f\ m)$
⟨*proof*⟩

lemma *Limsup-const*:
assumes *ntriv*: $\neg\ trivial\text{-}limit\ F$
shows $Limsup\ F\ (\lambda x.\ c) = c$
⟨*proof*⟩

lemma *Liminf-const*:
assumes *ntriv*: ¬ *trivial-limit F*
shows *Liminf F* (λ*x*. *c*) = *c*
⟨*proof*⟩

lemma *Liminf-mono*:
assumes *ev*: *eventually* (λ*x*. *f x* ≤ *g x*) *F*
shows *Liminf F f* ≤ *Liminf F g*
⟨*proof*⟩

lemma *Liminf-eq*:
assumes *eventually* (λ*x*. *f x* = *g x*) *F*
shows *Liminf F f* = *Liminf F g*
⟨*proof*⟩

lemma *Limsup-mono*:
assumes *ev*: *eventually* (λ*x*. *f x* ≤ *g x*) *F*
shows *Limsup F f* ≤ *Limsup F g*
⟨*proof*⟩

lemma *Limsup-eq*:
assumes *eventually* (λ*x*. *f x* = *g x*) *net*
shows *Limsup net f* = *Limsup net g*
⟨*proof*⟩

lemma *Liminf-bot*[*simp*]: *Liminf bot f* = *top*
⟨*proof*⟩

lemma *Limsup-bot*[*simp*]: *Limsup bot f* = *bot*
⟨*proof*⟩

lemma *Liminf-le-Limsup*:
assumes *ntriv*: ¬ *trivial-limit F*
shows *Liminf F f* ≤ *Limsup F f*
⟨*proof*⟩

lemma *Liminf-bounded*:
assumes *le*: *eventually* (λ*n*. *C* ≤ *X n*) *F*
shows *C* ≤ *Liminf F X*
⟨*proof*⟩

lemma *Limsup-bounded*:
assumes *le*: *eventually* (λ*n*. *X n* ≤ *C*) *F*
shows *Limsup F X* ≤ *C*
⟨*proof*⟩

lemma *le-Limsup*:
assumes *F*: *F* ≠ *bot* **and** *x*: ∀ *F x in F*. *l* ≤ *f x*

shows $l \leq$ *Limsup F f*
⟨*proof*⟩

lemma *Liminf-le*:
assumes *F*: $F \neq$ *bot* **and** *x*: $\forall_F\ x$ *in F*. $f\ x \leq l$
shows *Liminf F f* $\leq l$
⟨*proof*⟩

lemma *le-Liminf-iff*:
fixes *X* :: - ⇒ - :: *complete-linorder*
shows $C \leq$ *Liminf F X* $\longleftrightarrow$ $(\forall\, y{<}C.$ *eventually* $(\lambda x.\ y < X\ x)\ F)$
⟨*proof*⟩

lemma *Limsup-le-iff*:
fixes *X* :: - ⇒ - :: *complete-linorder*
shows $C \geq$ *Limsup F X* $\longleftrightarrow$ $(\forall\, y{>}C.$ *eventually* $(\lambda x.\ y > X\ x)\ F)$
⟨*proof*⟩

lemma *less-LiminfD*:
$y <$ *Liminf F* (*f* :: - ⇒ *'a* :: *complete-linorder*) $\Longrightarrow$ *eventually* $(\lambda x.\ f\ x > y)\ F$
⟨*proof*⟩

lemma *Limsup-lessD*:
$y >$ *Limsup F* (*f* :: - ⇒ *'a* :: *complete-linorder*) $\Longrightarrow$ *eventually* $(\lambda x.\ f\ x < y)\ F$
⟨*proof*⟩

lemma *lim-imp-Liminf*:
fixes *f* :: *'a* ⇒ - :: {*complete-linorder*,*linorder-topology*}
assumes *ntriv*: ¬ *trivial-limit F*
assumes *lim*: $(f \longrightarrow f0)\ F$
shows *Liminf F f* = *f0*
⟨*proof*⟩

lemma *lim-imp-Limsup*:
fixes *f* :: *'a* ⇒ - :: {*complete-linorder*,*linorder-topology*}
assumes *ntriv*: ¬ *trivial-limit F*
assumes *lim*: $(f \longrightarrow f0)\ F$
shows *Limsup F f* = *f0*
⟨*proof*⟩

lemma *Liminf-eq-Limsup*:
fixes *f0* :: *'a* :: {*complete-linorder*,*linorder-topology*}
assumes *ntriv*: ¬ *trivial-limit F*
and *lim*: *Liminf F f* = *f0 Limsup F f* = *f0*
shows $(f \longrightarrow f0)\ F$
⟨*proof*⟩

lemma *tendsto-iff-Liminf-eq-Limsup*:
fixes *f0* :: *'a* :: {*complete-linorder*,*linorder-topology*}

shows ¬ *trivial-limit F* ⟹ (*f* ⟶ *f0*) *F* ⟷ (*Liminf F f* = *f0* ∧ *Limsup F f* = *f0*)
⟨*proof*⟩

lemma *liminf-subseq-mono*:
fixes *X* :: *nat* ⇒ *'a* :: *complete-linorder*
assumes *strict-mono r*
shows *liminf X* ≤ *liminf* (*X* ∘ *r*)
⟨*proof*⟩

lemma *limsup-subseq-mono*:
fixes *X* :: *nat* ⇒ *'a* :: *complete-linorder*
assumes *strict-mono r*
shows *limsup* (*X* ∘ *r*) ≤ *limsup X*
⟨*proof*⟩

lemma *continuous-on-imp-continuous-within*:
continuous-on s f ⟹ *t* ⊆ *s* ⟹ *x* ∈ *s* ⟹ *continuous* (*at x within t*) *f*
⟨*proof*⟩

lemma *Liminf-compose-continuous-mono*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *c*: *continuous-on UNIV f* **and** *am*: *mono f* **and** *F*: *F* ≠ *bot*
shows *Liminf F* (λ*n*. *f* (*g n*)) = *f* (*Liminf F g*)
⟨*proof*⟩

lemma *Limsup-compose-continuous-mono*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *c*: *continuous-on UNIV f* **and** *am*: *mono f* **and** *F*: *F* ≠ *bot*
shows *Limsup F* (λ*n*. *f* (*g n*)) = *f* (*Limsup F g*)
⟨*proof*⟩

lemma *Liminf-compose-continuous-antimono*:
fixes *f* :: *'a*::{*complete-linorder*,*linorder-topology*} ⇒ *'b*::{*complete-linorder*,*linorder-topology*}
assumes *c*: *continuous-on UNIV f*
and *am*: *antimono f*
and *F*: *F* ≠ *bot*
shows *Liminf F* (λ*n*. *f* (*g n*)) = *f* (*Limsup F g*)
⟨*proof*⟩

lemma *Limsup-compose-continuous-antimono*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *c*: *continuous-on UNIV f* **and** *am*: *antimono f* **and** *F*: *F* ≠ *bot*
shows *Limsup F* (λ*n*. *f* (*g n*)) = *f* (*Liminf F g*)
⟨*proof*⟩

lemma *Liminf-filtermap-le*: *Liminf* (*filtermap f F*) *g* ≤ *Liminf F* (λ*x*. *g* (*f x*))
⟨*proof*⟩

lemma *Limsup-filtermap-ge*: *Limsup* (*filtermap f F*) *g* ≥ *Limsup F* (λ*x*. *g* (*f x*))
⟨*proof*⟩

lemma *Liminf-least*: (⋀*P*. *eventually P F* ⟹ (*INF x*:*Collect P*. *f x*) ≤ *x*) ⟹ *Liminf F f* ≤ *x*
⟨*proof*⟩

lemma *Limsup-greatest*: (⋀*P*. *eventually P F* ⟹ *x* ≤ (*SUP x*:*Collect P*. *f x*)) ⟹ *Limsup F f* ≥ *x*
⟨*proof*⟩

lemma *Liminf-filtermap-ge*: *inj f* ⟹ *Liminf* (*filtermap f F*) *g* ≥ *Liminf F* (λ*x*. *g* (*f x*))
⟨*proof*⟩

lemma *Limsup-filtermap-le*: *inj f* ⟹ *Limsup* (*filtermap f F*) *g* ≤ *Limsup F* (λ*x*. *g* (*f x*))
⟨*proof*⟩

lemma *Liminf-filtermap-eq*: *inj f* ⟹ *Liminf* (*filtermap f F*) *g* = *Liminf F* (λ*x*. *g* (*f x*))
⟨*proof*⟩

lemma *Limsup-filtermap-eq*: *inj f* ⟹ *Limsup* (*filtermap f F*) *g* = *Limsup F* (λ*x*. *g* (*f x*))
⟨*proof*⟩

29.1 More Limits

lemma *convergent-limsup-cl*:
fixes *X* :: *nat* ⇒ *'a*::{*complete-linorder*,*linorder-topology*}
shows *convergent X* ⟹ *limsup X* = *lim X*
⟨*proof*⟩

lemma *convergent-liminf-cl*:
fixes *X* :: *nat* ⇒ *'a*::{*complete-linorder*,*linorder-topology*}
shows *convergent X* ⟹ *liminf X* = *lim X*
⟨*proof*⟩

lemma *lim-increasing-cl*:
assumes ⋀*n m*. *n* ≥ *m* ⟹ *f n* ≥ *f m*
obtains *l* **where** *f* ⟶ (*l*::*'a*::{*complete-linorder*,*linorder-topology*})
⟨*proof*⟩

lemma *lim-decreasing-cl*:
assumes ⋀*n m*. *n* ≥ *m* ⟹ *f n* ≤ *f m*

obtains l **where** $f \longrightarrow (l::'a::\{complete\text{-}linorder,linorder\text{-}topology\})$
⟨*proof*⟩

lemma *compact-complete-linorder*:
fixes $X :: nat \Rightarrow 'a::\{complete\text{-}linorder,linorder\text{-}topology\}$
shows $\exists l\ r.\ strict\text{-}mono\ r \wedge (X \circ r) \longrightarrow l$
⟨*proof*⟩

lemma *tendsto-Limsup*:
fixes $f :: - \Rightarrow 'a :: \{complete\text{-}linorder,linorder\text{-}topology\}$
shows $F \neq bot \Longrightarrow Limsup\ F\ f = Liminf\ F\ f \Longrightarrow (f \longrightarrow Limsup\ F\ f)\ F$
⟨*proof*⟩

lemma *tendsto-Liminf*:
fixes $f :: - \Rightarrow 'a :: \{complete\text{-}linorder,linorder\text{-}topology\}$
shows $F \neq bot \Longrightarrow Limsup\ F\ f = Liminf\ F\ f \Longrightarrow (f \longrightarrow Liminf\ F\ f)\ F$
⟨*proof*⟩

end

30 Extended real number line

theory *Extended-Real*
imports *Complex-Main Extended-Nat Liminf-Limsup*
begin

This should be part of *Extended-Nat* or *Order-Continuity*, but then the AFP-entry *Jinja-Thread* fails, as it does overload certain named from *Complex-Main*.

lemma *incseq-sumI2*:
fixes $f :: 'i \Rightarrow nat \Rightarrow 'a::ordered\text{-}comm\text{-}monoid\text{-}add$
shows $(\bigwedge n.\ n \in A \Longrightarrow mono\ (f\ n)) \Longrightarrow mono\ (\lambda i.\ \sum n{\in}A.\ f\ n\ i)$
⟨*proof*⟩

lemma *incseq-sumI*:
fixes $f :: nat \Rightarrow 'a::ordered\text{-}comm\text{-}monoid\text{-}add$
assumes $\bigwedge i.\ 0 \leq f\ i$
shows $incseq\ (\lambda i.\ sum\ f\ \{..< i\})$
⟨*proof*⟩

lemma *continuous-at-left-imp-sup-continuous*:
fixes $f :: 'a::\{complete\text{-}linorder,\ linorder\text{-}topology\} \Rightarrow 'b::\{complete\text{-}linorder, linorder\text{-}topology\}$
assumes $mono\ f\ \bigwedge x.\ continuous\ (at\text{-}left\ x)\ f$
shows $sup\text{-}continuous\ f$
⟨*proof*⟩

lemma *sup-continuous-at-left*:
fixes $f :: 'a::\{complete\text{-}linorder,\ linorder\text{-}topology,\ first\text{-}countable\text{-}topology\} \Rightarrow$

'b::{*complete-linorder*, *linorder-topology*}
assumes *f*: *sup-continuous f*
shows *continuous* (*at-left x*) *f*
⟨*proof*⟩

lemma *sup-continuous-iff-at-left*:
fixes *f* :: 'a::{*complete-linorder*, *linorder-topology*, *first-countable-topology*} ⇒
'b::{*complete-linorder*, *linorder-topology*}
shows *sup-continuous f* ⟷ (∀ *x*. *continuous* (*at-left x*) *f*) ∧ *mono f*
⟨*proof*⟩

lemma *continuous-at-right-imp-inf-continuous*:
fixes *f* :: 'a::{*complete-linorder*, *linorder-topology*} ⇒ 'b::{*complete-linorder*, *linorder-topology*}
assumes *mono f* ⋀*x*. *continuous* (*at-right x*) *f*
shows *inf-continuous f*
⟨*proof*⟩

lemma *inf-continuous-at-right*:
fixes *f* :: 'a::{*complete-linorder*, *linorder-topology*, *first-countable-topology*} ⇒
'b::{*complete-linorder*, *linorder-topology*}
assumes *f*: *inf-continuous f*
shows *continuous* (*at-right x*) *f*
⟨*proof*⟩

lemma *inf-continuous-iff-at-right*:
fixes *f* :: 'a::{*complete-linorder*, *linorder-topology*, *first-countable-topology*} ⇒
'b::{*complete-linorder*, *linorder-topology*}
shows *inf-continuous f* ⟷ (∀ *x*. *continuous* (*at-right x*) *f*) ∧ *mono f*
⟨*proof*⟩

instantiation *enat* :: *linorder-topology*
begin

definition *open-enat* :: *enat set* ⇒ *bool* **where**
open-enat = *generate-topology* (*range lessThan* ∪ *range greaterThan*)

instance
⟨*proof*⟩

end

lemma *open-enat*: *open* {*enat n*}
⟨*proof*⟩

lemma *open-enat-iff*:
fixes *A* :: *enat set*
shows *open A* ⟷ (∞ ∈ *A* ⟶ (∃ *n*::*nat*. {*n* <..} ⊆ *A*))
⟨*proof*⟩

lemma *nhds-enat*: *nhds x* = (*if x* = ∞ *then INF i. principal* {*enat i..*} *else principal* {*x*})
⟨*proof*⟩

instance *enat* :: *topological-comm-monoid-add*
⟨*proof*⟩

For more lemmas about the extended real numbers see `~~/src/HOL/Analysis/Extended_Real_Limits.thy`.

30.1 Definition and basic properties

datatype *ereal* = *ereal real* | *PInfty* | *MInfty*

lemma *ereal-cong*: *x* = *y* ⟹ *ereal x* = *ereal y* ⟨*proof*⟩

instantiation *ereal* :: *uminus*
begin

fun *uminus-ereal* **where**
  − (*ereal r*) = *ereal* (− *r*)
| − *PInfty* = *MInfty*
| − *MInfty* = *PInfty*

instance ⟨*proof*⟩

end

instantiation *ereal* :: *infinity*
begin

definition (∞::*ereal*) = *PInfty*
instance ⟨*proof*⟩

end

declare [[*coercion ereal* :: *real* ⇒ *ereal*]]

lemma *ereal-uminus-uminus*[*simp*]:
  fixes *a* :: *ereal*
  shows − (− *a*) = *a*
  ⟨*proof*⟩

lemma
  shows *PInfty-eq-infinity*[*simp*]: *PInfty* = ∞
    and *MInfty-eq-minfinity*[*simp*]: *MInfty* = − ∞
    and *MInfty-neq-PInfty*[*simp*]: ∞ ≠ − (∞::*ereal*) − ∞ ≠ (∞::*ereal*)
    and *MInfty-neq-ereal*[*simp*]: *ereal r* ≠ − ∞ − ∞ ≠ *ereal r*

and *PInfty-neq-ereal*[*simp*]: *ereal r* $\neq \infty$ $\infty \neq$ *ereal r*
and *PInfty-cases*[*simp*]: (*case* ∞ *of ereal r* $\Rightarrow$ *f r* | *PInfty* $\Rightarrow$ *y* | *MInfty* $\Rightarrow$ *z*) = *y*
and *MInfty-cases*[*simp*]: (*case* $-\infty$ *of ereal r* $\Rightarrow$ *f r* | *PInfty* $\Rightarrow$ *y* | *MInfty* $\Rightarrow$ *z*) = *z*
⟨*proof*⟩

declare
PInfty-eq-infinity[*code-post*]
MInfty-eq-minfinity[*code-post*]

lemma [*code-unfold*]:
∞ = *PInfty*
$-$ *PInfty* = *MInfty*
⟨*proof*⟩

lemma *inj-ereal*[*simp*]: *inj-on ereal A*
⟨*proof*⟩

lemma *ereal-cases*[*cases type*: *ereal*]:
obtains (*real*) *r* **where** *x* = *ereal r*
| (*PInf*) *x* = ∞
| (*MInf*) *x* = $-\infty$
⟨*proof*⟩

lemmas *ereal2-cases* = *ereal-cases*[*case-product ereal-cases*]
lemmas *ereal3-cases* = *ereal2-cases*[*case-product ereal-cases*]

lemma *ereal-all-split*: $\bigwedge$*P*. ($\forall$ *x*::*ereal*. *P x*) $\longleftrightarrow$ *P* ∞ $\wedge$ ($\forall$ *x*. *P* (*ereal x*)) $\wedge$ *P* ($-\infty$)
⟨*proof*⟩

lemma *ereal-ex-split*: $\bigwedge$*P*. ($\exists$ *x*::*ereal*. *P x*) $\longleftrightarrow$ *P* ∞ $\vee$ ($\exists$ *x*. *P* (*ereal x*)) $\vee$ *P* ($-\infty$)
⟨*proof*⟩

lemma *ereal-uminus-eq-iff*[*simp*]:
fixes *a b* :: *ereal*
shows $-a = -b \longleftrightarrow a = b$
⟨*proof*⟩

function *real-of-ereal* :: *ereal* $\Rightarrow$ *real* **where**
real-of-ereal (*ereal r*) = *r*
| *real-of-ereal* ∞ = *0*
| *real-of-ereal* ($-\infty$) = *0*
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *real-of-ereal*[*simp*]:

real-of-ereal $(-\ x :: ereal) = -\ (real\text{-}of\text{-}ereal\ x)$
⟨*proof*⟩

lemma *range-ereal*[*simp*]: $range\ ereal = UNIV - \{\infty, -\infty\}$
⟨*proof*⟩

lemma *ereal-range-uminus*[*simp*]: $range\ uminus = (UNIV :: ereal\ set)$
⟨*proof*⟩

instantiation *ereal* :: *abs*
begin

function *abs-ereal* **where**
$|ereal\ r| = ereal\ |r|$
$| |-\infty| = (\infty :: ereal)$
$| |\infty| = (\infty :: ereal)$
⟨*proof*⟩
termination ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *abs-eq-infinity-cases*[*elim*!]:
fixes $x :: ereal$
assumes $|x| = \infty$
obtains $x = \infty \mid x = -\infty$
⟨*proof*⟩

lemma *abs-neq-infinity-cases*[*elim*!]:
fixes $x :: ereal$
assumes $|x| \neq \infty$
obtains r **where** $x = ereal\ r$
⟨*proof*⟩

lemma *abs-ereal-uminus*[*simp*]:
fixes $x :: ereal$
shows $|-\ x| = |x|$
⟨*proof*⟩

lemma *ereal-infinity-cases*:
fixes $a :: ereal$
shows $a \neq \infty \Longrightarrow a \neq -\infty \Longrightarrow |a| \neq \infty$
⟨*proof*⟩

30.1.1 Addition

instantiation *ereal* :: {*one*,*comm-monoid-add*,*zero-neq-one*}
begin

definition $0 = ereal\ 0$
definition $1 = ereal\ 1$

function *plus-ereal* **where**
  $ereal\ r + ereal\ p = ereal\ (r + p)$
| $\infty + a = (\infty :: ereal)$
| $a + \infty = (\infty :: ereal)$
| $ereal\ r + -\infty = -\infty$
| $-\infty + ereal\ p = -(\infty :: ereal)$
| $-\infty + -\infty = -(\infty :: ereal)$
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *Infty-neq-0*[*simp*]:
  $(\infty :: ereal) \neq 0\ 0 \neq (\infty :: ereal)$
  $-(\infty :: ereal) \neq 0\ 0 \neq -(\infty :: ereal)$
  ⟨*proof*⟩

lemma *ereal-eq-0*[*simp*]:
  $ereal\ r = 0 \longleftrightarrow r = 0$
  $0 = ereal\ r \longleftrightarrow r = 0$
  ⟨*proof*⟩

lemma *ereal-eq-1*[*simp*]:
  $ereal\ r = 1 \longleftrightarrow r = 1$
  $1 = ereal\ r \longleftrightarrow r = 1$
  ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *ereal-0-plus* [*simp*]: $ereal\ 0 + x = x$
  and *plus-ereal-0* [*simp*]: $x + ereal\ 0 = x$
⟨*proof*⟩

instance *ereal* :: *numeral* ⟨*proof*⟩

lemma *real-of-ereal-0*[*simp*]: $real\text{-}of\text{-}ereal\ (0 :: ereal) = 0$
  ⟨*proof*⟩

lemma *abs-ereal-zero*[*simp*]: $|0| = (0 :: ereal)$
  ⟨*proof*⟩

lemma *ereal-uminus-zero*[*simp*]: $-\ 0 = (0 :: ereal)$
  ⟨*proof*⟩

lemma *ereal-uminus-zero-iff* [*simp*]:
fixes a :: *ereal*
shows $-a = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *ereal-plus-eq-PInfty*[*simp*]:
fixes a b :: *ereal*
shows $a + b = \infty \longleftrightarrow a = \infty \vee b = \infty$
⟨*proof*⟩

lemma *ereal-plus-eq-MInfty*[*simp*]:
fixes a b :: *ereal*
shows $a + b = -\infty \longleftrightarrow (a = -\infty \vee b = -\infty) \wedge a \neq \infty \wedge b \neq \infty$
⟨*proof*⟩

lemma *ereal-add-cancel-left*:
fixes a b :: *ereal*
assumes $a \neq -\infty$
shows $a + b = a + c \longleftrightarrow a = \infty \vee b = c$
⟨*proof*⟩

lemma *ereal-add-cancel-right*:
fixes a b :: *ereal*
assumes $a \neq -\infty$
shows $b + a = c + a \longleftrightarrow a = \infty \vee b = c$
⟨*proof*⟩

lemma *ereal-real*: *ereal* (*real-of-ereal* x) = (*if* $|x| = \infty$ *then* 0 *else* x)
⟨*proof*⟩

lemma *real-of-ereal-add*:
fixes a b :: *ereal*
shows *real-of-ereal* $(a + b)$ =
(*if* $(|a| = \infty) \wedge (|b| = \infty) \vee (|a| \neq \infty) \wedge (|b| \neq \infty)$ *then real-of-ereal* a + *real-of-ereal* b *else* 0)
⟨*proof*⟩

30.1.2 Linear order on *ereal*

instantiation *ereal* :: *linorder*
begin

function *less-ereal*
where
ereal $x <$ *ereal* $y \longleftrightarrow x < y$
| $(\infty$::*ereal*$) < a \longleftrightarrow$ *False*
| $a < -(\infty$::*ereal*$) \longleftrightarrow$ *False*
| *ereal* $x < \infty \longleftrightarrow$ *True*
| $-\infty <$ *ereal* $r \longleftrightarrow$ *True*

| $-\infty < (\infty::ereal) \longleftrightarrow True$
⟨*proof*⟩
termination ⟨*proof*⟩

definition $x \leq (y::ereal) \longleftrightarrow x < y \vee x = y$

lemma *ereal-infty-less*[*simp*]:
fixes $x :: ereal$
shows $x < \infty \longleftrightarrow (x \neq \infty)$
$-\infty < x \longleftrightarrow (x \neq -\infty)$
⟨*proof*⟩

lemma *ereal-infty-less-eq*[*simp*]:
fixes $x :: ereal$
shows $\infty \leq x \longleftrightarrow x = \infty$
and $x \leq -\infty \longleftrightarrow x = -\infty$
⟨*proof*⟩

lemma *ereal-less*[*simp*]:
$ereal\ r < 0 \longleftrightarrow (r < 0)$
$0 < ereal\ r \longleftrightarrow (0 < r)$
$ereal\ r < 1 \longleftrightarrow (r < 1)$
$1 < ereal\ r \longleftrightarrow (1 < r)$
$0 < (\infty::ereal)$
$-(\infty::ereal) < 0$
⟨*proof*⟩

lemma *ereal-less-eq*[*simp*]:
$x \leq (\infty::ereal)$
$-(\infty::ereal) \leq x$
$ereal\ r \leq ereal\ p \longleftrightarrow r \leq p$
$ereal\ r \leq 0 \longleftrightarrow r \leq 0$
$0 \leq ereal\ r \longleftrightarrow 0 \leq r$
$ereal\ r \leq 1 \longleftrightarrow r \leq 1$
$1 \leq ereal\ r \longleftrightarrow 1 \leq r$
⟨*proof*⟩

lemma *ereal-infty-less-eq2*:
$a \leq b \Longrightarrow a = \infty \Longrightarrow b = (\infty::ereal)$
$a \leq b \Longrightarrow b = -\infty \Longrightarrow a = -(\infty::ereal)$
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *ereal-dense2*: $x < y \Longrightarrow \exists z.\ x < ereal\ z \wedge ereal\ z < y$
⟨*proof*⟩

instance *ereal* :: *dense-linorder*
⟨*proof*⟩

instance *ereal* :: *ordered-comm-monoid-add*
⟨*proof*⟩

lemma *ereal-one-not-less-zero-ereal*[*simp*]: ¬ *1* < (*0*::*ereal*)
⟨*proof*⟩

lemma *real-of-ereal-positive-mono*:
fixes *x y* :: *ereal*
shows *0* ≤ *x* ⟹ *x* ≤ *y* ⟹ *y* ≠ ∞ ⟹ *real-of-ereal x* ≤ *real-of-ereal y*
⟨*proof*⟩

lemma *ereal-MInfty-lessI*[*intro*, *simp*]:
fixes *a* :: *ereal*
shows *a* ≠ −∞ ⟹ −∞ < *a*
⟨*proof*⟩

lemma *ereal-less-PInfty*[*intro*, *simp*]:
fixes *a* :: *ereal*
shows *a* ≠ ∞ ⟹ *a* < ∞
⟨*proof*⟩

lemma *ereal-less-ereal-Ex*:
fixes *a b* :: *ereal*
shows *x* < *ereal r* ⟷ *x* = −∞ ∨ (∃ *p*. *p* < *r* ∧ *x* = *ereal p*)
⟨*proof*⟩

lemma *less-PInf-Ex-of-nat*: *x* ≠ ∞ ⟷ (∃ *n*::*nat*. *x* < *ereal* (*real n*))
⟨*proof*⟩

lemma *ereal-add-mono*:
fixes *a b c d* :: *ereal*
assumes *a* ≤ *b*
and *c* ≤ *d*
shows *a* + *c* ≤ *b* + *d*
⟨*proof*⟩

lemma *ereal-minus-le-minus*[*simp*]:
fixes *a b* :: *ereal*
shows − *a* ≤ − *b* ⟷ *b* ≤ *a*
⟨*proof*⟩

lemma *ereal-minus-less-minus*[*simp*]:
fixes *a b* :: *ereal*
shows − *a* < − *b* ⟷ *b* < *a*
⟨*proof*⟩

lemma *ereal-le-real-iff*:
$x \leq$ *real-of-ereal* $y \longleftrightarrow (|y| \neq \infty \longrightarrow$ *ereal* $x \leq y) \wedge (|y| = \infty \longrightarrow x \leq 0)$
⟨*proof*⟩

lemma *real-le-ereal-iff*:
real-of-ereal $y \leq x \longleftrightarrow (|y| \neq \infty \longrightarrow y \leq$ *ereal* $x) \wedge (|y| = \infty \longrightarrow 0 \leq x)$
⟨*proof*⟩

lemma *ereal-less-real-iff*:
$x <$ *real-of-ereal* $y \longleftrightarrow (|y| \neq \infty \longrightarrow$ *ereal* $x < y) \wedge (|y| = \infty \longrightarrow x < 0)$
⟨*proof*⟩

lemma *real-less-ereal-iff*:
real-of-ereal $y < x \longleftrightarrow (|y| \neq \infty \longrightarrow y <$ *ereal* $x) \wedge (|y| = \infty \longrightarrow 0 < x)$
⟨*proof*⟩

lemma *real-of-ereal-pos*:
fixes $x ::$ *ereal*
shows $0 \leq x \Longrightarrow 0 \leq$ *real-of-ereal* x ⟨*proof*⟩

lemmas *real-of-ereal-ord-simps* =
ereal-le-real-iff real-le-ereal-iff ereal-less-real-iff real-less-ereal-iff

lemma *abs-ereal-ge0*[*simp*]: $0 \leq x \Longrightarrow |x ::$ *ereal*$| = x$
⟨*proof*⟩

lemma *abs-ereal-less0*[*simp*]: $x < 0 \Longrightarrow |x ::$ *ereal*$| = -x$
⟨*proof*⟩

lemma *abs-ereal-pos*[*simp*]: $0 \leq |x ::$ *ereal*$|$
⟨*proof*⟩

lemma *ereal-abs-leI*:
fixes $x\ y ::$ *ereal*
shows $[\![x \leq y; -x \leq y]\!] \Longrightarrow |x| \leq y$
⟨*proof*⟩

lemma *real-of-ereal-le-0*[*simp*]: *real-of-ereal* $(x ::$ *ereal*$) \leq 0 \longleftrightarrow x \leq 0 \vee x = \infty$
⟨*proof*⟩

lemma *abs-real-of-ereal*[*simp*]: $|$*real-of-ereal* $(x ::$ *ereal*$)| =$ *real-of-ereal* $|x|$
⟨*proof*⟩

lemma *zero-less-real-of-ereal*:
fixes $x ::$ *ereal*
shows $0 <$ *real-of-ereal* $x \longleftrightarrow 0 < x \wedge x \neq \infty$
⟨*proof*⟩

lemma *ereal-0-le-uminus-iff* [*simp*]:
fixes a :: *ereal*
shows $0 \leq - a \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *ereal-uminus-le-0-iff* [*simp*]:
fixes a :: *ereal*
shows $- a \leq 0 \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *ereal-add-strict-mono*:
fixes $a\ b\ c\ d$ :: *ereal*
assumes $a \leq b$
and $0 \leq a$
and $a \neq \infty$
and $c < d$
shows $a + c < b + d$
⟨*proof*⟩

lemma *ereal-less-add*:
fixes $a\ b\ c$ :: *ereal*
shows $|a| \neq \infty \Longrightarrow c < b \Longrightarrow a + c < a + b$
⟨*proof*⟩

lemma *ereal-add-nonneg-eq-0-iff*:
fixes $a\ b$ :: *ereal*
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a + b = 0 \longleftrightarrow a = 0 \wedge b = 0$
⟨*proof*⟩

lemma *ereal-uminus-eq-reorder*: $- a = b \longleftrightarrow a = (-b::ereal)$
⟨*proof*⟩

lemma *ereal-uminus-less-reorder*: $- a < b \longleftrightarrow -b < (a::ereal)$
⟨*proof*⟩

lemma *ereal-less-uminus-reorder*: $a < - b \longleftrightarrow b < - (a::ereal)$
⟨*proof*⟩

lemma *ereal-uminus-le-reorder*: $- a \leq b \longleftrightarrow -b \leq (a::ereal)$
⟨*proof*⟩

lemmas *ereal-uminus-reorder* =
ereal-uminus-eq-reorder ereal-uminus-less-reorder ereal-uminus-le-reorder

lemma *ereal-bot*:
fixes x :: *ereal*
assumes $\bigwedge B.\ x \leq ereal\ B$
shows $x = - \infty$
⟨*proof*⟩

lemma *ereal-top*:
fixes x :: *ereal*
assumes $\bigwedge B.\ x \geq ereal\ B$
shows $x = \infty$
⟨*proof*⟩

lemma
shows *ereal-max*[*simp*]: $ereal\ (max\ x\ y) = max\ (ereal\ x)\ (ereal\ y)$
and *ereal-min*[*simp*]: $ereal\ (min\ x\ y) = min\ (ereal\ x)\ (ereal\ y)$
⟨*proof*⟩

lemma *ereal-max-0*: $max\ 0\ (ereal\ r) = ereal\ (max\ 0\ r)$
⟨*proof*⟩

lemma
fixes f :: $nat \Rightarrow ereal$
shows *ereal-incseq-uminus*[*simp*]: $incseq\ (\lambda x.\ -f\ x) \longleftrightarrow decseq\ f$
and *ereal-decseq-uminus*[*simp*]: $decseq\ (\lambda x.\ -f\ x) \longleftrightarrow incseq\ f$
⟨*proof*⟩

lemma *incseq-ereal*: $incseq\ f \Longrightarrow incseq\ (\lambda x.\ ereal\ (f\ x))$
⟨*proof*⟩

lemma *ereal-add-nonneg-nonneg*[*simp*]:
fixes $a\ b$ :: *ereal*
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq a + b$
⟨*proof*⟩

lemma *sum-ereal*[*simp*]: $(\sum x{\in}A.\ ereal\ (f\ x)) = ereal\ (\sum x{\in}A.\ f\ x)$
⟨*proof*⟩

lemma *sum-list-ereal* [*simp*]: $sum\text{-}list\ (map\ (\lambda x.\ ereal\ (f\ x))\ xs) = ereal\ (sum\text{-}list\ (map\ f\ xs))$
⟨*proof*⟩

lemma *sum-Pinfty*:
fixes f :: $'a \Rightarrow ereal$
shows $(\sum x{\in}P.\ f\ x) = \infty \longleftrightarrow finite\ P \wedge (\exists\, i{\in}P.\ f\ i = \infty)$
⟨*proof*⟩

lemma *sum-Inf*:
fixes f :: $'a \Rightarrow ereal$
shows $|sum\ f\ A| = \infty \longleftrightarrow finite\ A \wedge (\exists\, i{\in}A.\ |f\ i| = \infty)$
⟨*proof*⟩

lemma *sum-real-of-ereal*:
fixes f :: $'i \Rightarrow ereal$
assumes $\bigwedge x.\ x \in S \Longrightarrow |f\ x| \neq \infty$

shows $(\sum x{\in}S.\ \textit{real-of-ereal}\ (f\ x)) = \textit{real-of-ereal}\ (\textit{sum}\ f\ S)$
⟨*proof*⟩

lemma *sum-ereal-0*:
fixes $f :: 'a \Rightarrow \textit{ereal}$
assumes *finite A*
and $\bigwedge i.\ i \in A \Longrightarrow 0 \le f\ i$
shows $(\sum x{\in}A.\ f\ x) = 0 \longleftrightarrow (\forall\, i{\in}A.\ f\ i = 0)$
⟨*proof*⟩

30.1.3 Multiplication

instantiation *ereal* :: {*comm-monoid-mult*,*sgn*}
begin

function *sgn-ereal* :: *ereal* ⇒ *ereal* **where**
$\textit{sgn}\ (\textit{ereal}\ r) = \textit{ereal}\ (\textit{sgn}\ r)$
| $\textit{sgn}\ (\infty{::}\textit{ereal}) = 1$
| $\textit{sgn}\ (-\infty{::}\textit{ereal}) = -1$
⟨*proof*⟩
termination ⟨*proof*⟩

function *times-ereal* **where**
$\textit{ereal}\ r * \textit{ereal}\ p = \textit{ereal}\ (r * p)$
| $\textit{ereal}\ r * \infty = (\textit{if}\ r = 0\ \textit{then}\ 0\ \textit{else if}\ r > 0\ \textit{then}\ \infty\ \textit{else}\ -\infty)$
| $\infty * \textit{ereal}\ r = (\textit{if}\ r = 0\ \textit{then}\ 0\ \textit{else if}\ r > 0\ \textit{then}\ \infty\ \textit{else}\ -\infty)$
| $\textit{ereal}\ r * -\infty = (\textit{if}\ r = 0\ \textit{then}\ 0\ \textit{else if}\ r > 0\ \textit{then}\ -\infty\ \textit{else}\ \infty)$
| $-\infty * \textit{ereal}\ r = (\textit{if}\ r = 0\ \textit{then}\ 0\ \textit{else if}\ r > 0\ \textit{then}\ -\infty\ \textit{else}\ \infty)$
| $(\infty{::}\textit{ereal}) * \infty = \infty$
| $-(\infty{::}\textit{ereal}) * \infty = -\infty$
| $(\infty{::}\textit{ereal}) * -\infty = -\infty$
| $-(\infty{::}\textit{ereal}) * -\infty = \infty$
⟨*proof*⟩
termination ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma [*simp*]:
shows *ereal-1-times*: $\textit{ereal}\ 1 * x = x$
and *times-ereal-1*: $x * \textit{ereal}\ 1 = x$
⟨*proof*⟩

lemma *one-not-le-zero-ereal*[*simp*]: $\neg\ (1 \le (0{::}\textit{ereal}))$
⟨*proof*⟩

lemma *real-ereal-1*[*simp*]: $\textit{real-of-ereal}\ (1{::}\textit{ereal}) = 1$

⟨*proof*⟩

lemma *real-of-ereal-le-1*:
fixes a :: *ereal*
shows $a \leq 1 \Longrightarrow$ *real-of-ereal* $a \leq 1$
⟨*proof*⟩

lemma *abs-ereal-one*[*simp*]: $|1| = (1::ereal)$
⟨*proof*⟩

lemma *ereal-mult-zero*[*simp*]:
fixes a :: *ereal*
shows $a * 0 = 0$
⟨*proof*⟩

lemma *ereal-zero-mult*[*simp*]:
fixes a :: *ereal*
shows $0 * a = 0$
⟨*proof*⟩

lemma *ereal-m1-less-0*[*simp*]: $-(1::ereal) < 0$
⟨*proof*⟩

lemma *ereal-times*[*simp*]:
$1 \neq (\infty::ereal)$ $(\infty::ereal) \neq 1$
$1 \neq -(\infty::ereal)$ $-(\infty::ereal) \neq 1$
⟨*proof*⟩

lemma *ereal-plus-1*[*simp*]:
$1 + ereal\ r = ereal\ (r + 1)$
$ereal\ r + 1 = ereal\ (r + 1)$
$1 + -(\infty::ereal) = -\infty$
$-(\infty::ereal) + 1 = -\infty$
⟨*proof*⟩

lemma *ereal-zero-times*[*simp*]:
fixes $a\ b$ :: *ereal*
shows $a * b = 0 \longleftrightarrow a = 0 \vee b = 0$
⟨*proof*⟩

lemma *ereal-mult-eq-PInfty*[*simp*]:
$a * b = (\infty::ereal) \longleftrightarrow$
$(a = \infty \wedge b > 0) \vee (a > 0 \wedge b = \infty) \vee (a = -\infty \wedge b < 0) \vee (a < 0 \wedge b = -\infty)$
⟨*proof*⟩

lemma *ereal-mult-eq-MInfty*[*simp*]:
$a * b = -(\infty::ereal) \longleftrightarrow$
$(a = \infty \wedge b < 0) \vee (a < 0 \wedge b = \infty) \vee (a = -\infty \wedge b > 0) \vee (a > 0 \wedge b =$

$-\infty$)
⟨*proof*⟩

lemma *ereal-abs-mult*: $|x * y :: ereal| = |x| * |y|$
⟨*proof*⟩

lemma *ereal-0-less-1*[*simp*]: $0 < (1::ereal)$
⟨*proof*⟩

lemma *ereal-mult-minus-left*[*simp*]:
fixes $a\ b :: ereal$
shows $-a * b = -(a * b)$
⟨*proof*⟩

lemma *ereal-mult-minus-right*[*simp*]:
fixes $a\ b :: ereal$
shows $a * -b = -(a * b)$
⟨*proof*⟩

lemma *ereal-mult-infty*[*simp*]:
$a * (\infty::ereal) = (if\ a = 0\ then\ 0\ else\ if\ 0 < a\ then\ \infty\ else\ -\infty)$
⟨*proof*⟩

lemma *ereal-infty-mult*[*simp*]:
$(\infty::ereal) * a = (if\ a = 0\ then\ 0\ else\ if\ 0 < a\ then\ \infty\ else\ -\infty)$
⟨*proof*⟩

lemma *ereal-mult-strict-right-mono*:
assumes $a < b$
and $0 < c$
and $c < (\infty::ereal)$
shows $a * c < b * c$
⟨*proof*⟩

lemma *ereal-mult-strict-left-mono*:
$a < b \Longrightarrow 0 < c \Longrightarrow c < (\infty::ereal) \Longrightarrow c * a < c * b$
⟨*proof*⟩

lemma *ereal-mult-right-mono*:
fixes $a\ b\ c :: ereal$
shows $a \leq b \Longrightarrow 0 \leq c \Longrightarrow a * c \leq b * c$
⟨*proof*⟩

lemma *ereal-mult-left-mono*:
fixes $a\ b\ c :: ereal$
shows $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$
⟨*proof*⟩

lemma *zero-less-one-ereal*[*simp*]: $0 \leq (1::ereal)$

⟨*proof*⟩

lemma *ereal-0-le-mult*[*simp*]: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq a * (b :: ereal)$
⟨*proof*⟩

lemma *ereal-right-distrib*:
fixes $r\ a\ b :: ereal$
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow r * (a + b) = r * a + r * b$
⟨*proof*⟩

lemma *ereal-left-distrib*:
fixes $r\ a\ b :: ereal$
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow (a + b) * r = a * r + b * r$
⟨*proof*⟩

lemma *ereal-mult-le-0-iff*:
fixes $a\ b :: ereal$
shows $a * b \leq 0 \longleftrightarrow (0 \leq a \wedge b \leq 0) \vee (a \leq 0 \wedge 0 \leq b)$
⟨*proof*⟩

lemma *ereal-zero-le-0-iff*:
fixes $a\ b :: ereal$
shows $0 \leq a * b \longleftrightarrow (0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0)$
⟨*proof*⟩

lemma *ereal-mult-less-0-iff*:
fixes $a\ b :: ereal$
shows $a * b < 0 \longleftrightarrow (0 < a \wedge b < 0) \vee (a < 0 \wedge 0 < b)$
⟨*proof*⟩

lemma *ereal-zero-less-0-iff*:
fixes $a\ b :: ereal$
shows $0 < a * b \longleftrightarrow (0 < a \wedge 0 < b) \vee (a < 0 \wedge b < 0)$
⟨*proof*⟩

lemma *ereal-left-mult-cong*:
fixes $a\ b\ c :: ereal$
shows $c = d \Longrightarrow (d \neq 0 \Longrightarrow a = b) \Longrightarrow a * c = b * d$
⟨*proof*⟩

lemma *ereal-right-mult-cong*:
fixes $a\ b\ c :: ereal$
shows $c = d \Longrightarrow (d \neq 0 \Longrightarrow a = b) \Longrightarrow c * a = d * b$
⟨*proof*⟩

lemma *ereal-distrib*:
fixes $a\ b\ c :: ereal$
assumes $a \neq \infty \vee b \neq -\infty$
and $a \neq -\infty \vee b \neq \infty$

and $|c| \neq \infty$
shows $(a + b) * c = a * c + b * c$
⟨*proof*⟩

lemma *numeral-eq-ereal* [*simp*]: $numeral\ w = ereal\ (numeral\ w)$
⟨*proof*⟩

lemma *distrib-left-ereal-nn*:
$c \geq 0 \Longrightarrow (x + y) * ereal\ c = x * ereal\ c + y * ereal\ c$
⟨*proof*⟩

lemma *sum-ereal-right-distrib*:
fixes $f :: 'a \Rightarrow ereal$
shows $(\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i) \Longrightarrow r * sum\ f\ A = (\sum n{\in}A.\ r * f\ n)$
⟨*proof*⟩

lemma *sum-ereal-left-distrib*:
$(\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i) \Longrightarrow sum\ f\ A * r = (\sum n{\in}A.\ f\ n * r :: ereal)$
⟨*proof*⟩

lemma *sum-distrib-right-ereal*:
$c \geq 0 \Longrightarrow sum\ f\ A * ereal\ c = (\sum x{\in}A.\ f\ x * c :: ereal)$
⟨*proof*⟩

lemma *ereal-le-epsilon*:
fixes $x\ y :: ereal$
assumes $\forall e.\ 0 < e \longrightarrow x \leq y + e$
shows $x \leq y$
⟨*proof*⟩

lemma *ereal-le-epsilon2*:
fixes $x\ y :: ereal$
assumes $\forall e.\ 0 < e \longrightarrow x \leq y + ereal\ e$
shows $x \leq y$
⟨*proof*⟩

lemma *ereal-le-real*:
fixes $x\ y :: ereal$
assumes $\forall z.\ x \leq ereal\ z \longrightarrow y \leq ereal\ z$
shows $y \leq x$
⟨*proof*⟩

lemma *prod-ereal-0*:
fixes $f :: 'a \Rightarrow ereal$
shows $(\prod i{\in}A.\ f\ i) = 0 \longleftrightarrow finite\ A \wedge (\exists i{\in}A.\ f\ i = 0)$
⟨*proof*⟩

lemma *prod-ereal-pos*:
fixes $f :: 'a \Rightarrow ereal$

assumes *pos*: $\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i$
shows $0 \leq (\prod i{\in}I.\ f\ i)$
⟨*proof*⟩

lemma *prod-PInf*:
fixes $f :: 'a \Rightarrow ereal$
assumes $\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i$
shows $(\prod i{\in}I.\ f\ i) = \infty \longleftrightarrow finite\ I \wedge (\exists\, i{\in}I.\ f\ i = \infty) \wedge (\forall\, i{\in}I.\ f\ i \neq 0)$
⟨*proof*⟩

lemma *prod-ereal*: $(\prod i{\in}A.\ ereal\ (f\ i)) = ereal\ (prod\ f\ A)$
⟨*proof*⟩

30.1.4 Power

lemma *ereal-power*[*simp*]: $(ereal\ x)\ \hat{}\ n = ereal\ (x\hat{}n)$
⟨*proof*⟩

lemma *ereal-power-PInf*[*simp*]: $(\infty{::}ereal)\ \hat{}\ n = (if\ n = 0\ then\ 1\ else\ \infty)$
⟨*proof*⟩

lemma *ereal-power-uminus*[*simp*]:
fixes $x :: ereal$
shows $(-\ x)\ \hat{}\ n = (if\ even\ n\ then\ x\ \hat{}\ n\ else\ -\ (x\hat{}n))$
⟨*proof*⟩

lemma *ereal-power-numeral*[*simp*]:
$(numeral\ num :: ereal)\ \hat{}\ n = ereal\ (numeral\ num\ \hat{}\ n)$
⟨*proof*⟩

lemma *zero-le-power-ereal*[*simp*]:
fixes $a :: ereal$
assumes $0 \leq a$
shows $0 \leq a\ \hat{}\ n$
⟨*proof*⟩

30.1.5 Subtraction

lemma *ereal-minus-minus-image*[*simp*]:
fixes $S :: ereal\ set$
shows $uminus\ `\ uminus\ `\ S = S$
⟨*proof*⟩

lemma *ereal-uminus-lessThan*[*simp*]:
fixes $a :: ereal$
shows $uminus\ `\ \{..<a\} = \{-a<..\}$
⟨*proof*⟩

lemma *ereal-uminus-greaterThan*[*simp*]: $uminus\ `\ \{(a{::}ereal)<..\} = \{..<-a\}$
⟨*proof*⟩

instantiation *ereal* :: *minus*
begin

definition $x - y = x + -(y::ereal)$
instance ⟨*proof*⟩

end

lemma *ereal-minus*[*simp*]:
$ereal\ r - ereal\ p = ereal\ (r - p)$
$-\infty - ereal\ r = -\infty$
$ereal\ r - \infty = -\infty$
$(\infty::ereal) - x = \infty$
$-(\infty::ereal) - \infty = -\infty$
$x - -y = x + y$
$x - 0 = x$
$0 - x = -x$
⟨*proof*⟩

lemma *ereal-x-minus-x*[*simp*]: $x - x = (if\ |x| = \infty\ then\ \infty\ else\ 0::ereal)$
⟨*proof*⟩

lemma *ereal-eq-minus-iff*:
fixes $x\ y\ z$ :: *ereal*
shows $x = z - y \longleftrightarrow$
$(|y| \neq \infty \longrightarrow x + y = z) \wedge$
$(y = -\infty \longrightarrow x = \infty) \wedge$
$(y = \infty \longrightarrow z = \infty \longrightarrow x = \infty) \wedge$
$(y = \infty \longrightarrow z \neq \infty \longrightarrow x = -\infty)$
⟨*proof*⟩

lemma *ereal-eq-minus*:
fixes $x\ y\ z$ :: *ereal*
shows $|y| \neq \infty \Longrightarrow x = z - y \longleftrightarrow x + y = z$
⟨*proof*⟩

lemma *ereal-less-minus-iff*:
fixes $x\ y\ z$ :: *ereal*
shows $x < z - y \longleftrightarrow$
$(y = \infty \longrightarrow z = \infty \wedge x \neq \infty) \wedge$
$(y = -\infty \longrightarrow x \neq \infty) \wedge$
$(|y| \neq \infty \longrightarrow x + y < z)$
⟨*proof*⟩

lemma *ereal-less-minus*:
fixes $x\ y\ z$ :: *ereal*
shows $|y| \neq \infty \Longrightarrow x < z - y \longleftrightarrow x + y < z$
⟨*proof*⟩

lemma *ereal-le-minus-iff*:
fixes $x\ y\ z :: ereal$
shows $x \leq z - y \longleftrightarrow (y = \infty \longrightarrow z \neq \infty \longrightarrow x = -\infty) \wedge (|y| \neq \infty \longrightarrow x + y \leq z)$
⟨*proof*⟩

lemma *ereal-le-minus*:
fixes $x\ y\ z :: ereal$
shows $|y| \neq \infty \Longrightarrow x \leq z - y \longleftrightarrow x + y \leq z$
⟨*proof*⟩

lemma *ereal-minus-less-iff*:
fixes $x\ y\ z :: ereal$
shows $x - y < z \longleftrightarrow y \neq -\infty \wedge (y = \infty \longrightarrow x \neq \infty \wedge z \neq -\infty) \wedge (y \neq \infty \longrightarrow x < z + y)$
⟨*proof*⟩

lemma *ereal-minus-less*:
fixes $x\ y\ z :: ereal$
shows $|y| \neq \infty \Longrightarrow x - y < z \longleftrightarrow x < z + y$
⟨*proof*⟩

lemma *ereal-minus-le-iff*:
fixes $x\ y\ z :: ereal$
shows $x - y \leq z \longleftrightarrow$
$(y = -\infty \longrightarrow z = \infty) \wedge$
$(y = \infty \longrightarrow x = \infty \longrightarrow z = \infty) \wedge$
$(|y| \neq \infty \longrightarrow x \leq z + y)$
⟨*proof*⟩

lemma *ereal-minus-le*:
fixes $x\ y\ z :: ereal$
shows $|y| \neq \infty \Longrightarrow x - y \leq z \longleftrightarrow x \leq z + y$
⟨*proof*⟩

lemma *ereal-minus-eq-minus-iff*:
fixes $a\ b\ c :: ereal$
shows $a - b = a - c \longleftrightarrow$
$b = c \vee a = \infty \vee (a = -\infty \wedge b \neq -\infty \wedge c \neq -\infty)$
⟨*proof*⟩

lemma *ereal-add-le-add-iff*:
fixes $a\ b\ c :: ereal$
shows $c + a \leq c + b \longleftrightarrow$
$a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$
⟨*proof*⟩

lemma *ereal-add-le-add-iff2*:

fixes $a\ b\ c :: ereal$
shows $a + c \leq b + c \longleftrightarrow a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$
⟨*proof*⟩

lemma *ereal-mult-le-mult-iff*:
fixes $a\ b\ c :: ereal$
shows $|c| \neq \infty \Longrightarrow c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
⟨*proof*⟩

lemma *ereal-minus-mono*:
fixes $A\ B\ C\ D :: ereal$ **assumes** $A \leq B\ D \leq C$
shows $A - C \leq B - D$
⟨*proof*⟩

lemma *ereal-mono-minus-cancel*:
fixes $a\ b\ c :: ereal$
shows $c - a \leq c - b \Longrightarrow 0 \leq c \Longrightarrow c < \infty \Longrightarrow b \leq a$
⟨*proof*⟩

lemma *real-of-ereal-minus*:
fixes $a\ b :: ereal$
shows *real-of-ereal* $(a - b) =$ (*if* $|a| = \infty \vee |b| = \infty$ *then 0 else real-of-ereal a* $-$ *real-of-ereal b*)
⟨*proof*⟩

lemma *real-of-ereal-minus'*: $|x| = \infty \longleftrightarrow |y| = \infty \Longrightarrow$ *real-of-ereal x* $-$ *real-of-ereal* $y =$ *real-of-ereal* $(x - y :: ereal)$
⟨*proof*⟩

lemma *ereal-diff-positive*:
fixes $a\ b :: ereal$ **shows** $a \leq b \Longrightarrow 0 \leq b - a$
⟨*proof*⟩

lemma *ereal-between*:
fixes $x\ e :: ereal$
assumes $|x| \neq \infty$
and $0 < e$
shows $x - e < x$
and $x < x + e$
⟨*proof*⟩

lemma *ereal-minus-eq-PInfty-iff*:
fixes $x\ y :: ereal$
shows $x - y = \infty \longleftrightarrow y = -\infty \vee x = \infty$
⟨*proof*⟩

lemma *ereal-diff-add-eq-diff-diff-swap*:
fixes $x\ y\ z :: ereal$
shows $|y| \neq \infty \Longrightarrow x - (y + z) = x - y - z$

⟨*proof*⟩

lemma *ereal-diff-add-assoc2*:
fixes $x\ y\ z$ :: *ereal*
shows $x + y - z = x - z + y$
⟨*proof*⟩

lemma *ereal-add-uminus-conv-diff*: **fixes** $x\ y\ z$ :: *ereal* **shows** $-x + y = y - x$
⟨*proof*⟩

lemma *ereal-minus-diff-eq*:
fixes $x\ y$ :: *ereal*
shows $[\![x = \infty \longrightarrow y \neq \infty; x = -\infty \longrightarrow y \neq -\infty]\!] \Longrightarrow -(x - y) = y - x$
⟨*proof*⟩

lemma *ediff-le-self* [*simp*]: $x - y \leq (x :: enat)$
⟨*proof*⟩

30.1.6 Division

instantiation *ereal* :: *inverse*
begin

function *inverse-ereal* **where**
inverse (*ereal r*) = (*if r* = *0 then* ∞ *else ereal* (*inverse r*))
| *inverse* (∞::*ereal*) = *0*
| *inverse* (−∞::*ereal*) = *0*
⟨*proof*⟩
termination ⟨*proof*⟩

definition *x div y* = *x* ∗ *inverse* (*y* :: *ereal*)

instance ⟨*proof*⟩

end

lemma *real-of-ereal-inverse*[*simp*]:
fixes *a* :: *ereal*
shows *real-of-ereal* (*inverse a*) = *1* / *real-of-ereal a*
⟨*proof*⟩

lemma *ereal-inverse*[*simp*]:
inverse (*0*::*ereal*) = ∞
inverse (*1*::*ereal*) = *1*
⟨*proof*⟩

lemma *ereal-divide*[*simp*]:
ereal r / *ereal p* = (*if p* = *0 then ereal r* ∗ ∞ *else ereal* (*r* / *p*))
⟨*proof*⟩

lemma *ereal-divide-same*[*simp*]:
fixes x :: *ereal*
shows $x \,/\, x = (\mathit{if}\ |x| = \infty \vee x = 0\ \mathit{then}\ 0\ \mathit{else}\ 1)$
⟨*proof*⟩

lemma *ereal-inv-inv*[*simp*]:
fixes x :: *ereal*
shows $\mathit{inverse}\ (\mathit{inverse}\ x) = (\mathit{if}\ x \neq -\infty\ \mathit{then}\ x\ \mathit{else}\ \infty)$
⟨*proof*⟩

lemma *ereal-inverse-minus*[*simp*]:
fixes x :: *ereal*
shows $\mathit{inverse}\ (-\ x) = (\mathit{if}\ x = 0\ \mathit{then}\ \infty\ \mathit{else}\ -\mathit{inverse}\ x)$
⟨*proof*⟩

lemma *ereal-uminus-divide*[*simp*]:
fixes $x\ y$ :: *ereal*
shows $-\ x \,/\, y = -\ (x \,/\, y)$
⟨*proof*⟩

lemma *ereal-divide-Infty*[*simp*]:
fixes x :: *ereal*
shows $x \,/\, \infty = 0\ x \,/\, -\infty = 0$
⟨*proof*⟩

lemma *ereal-divide-one*[*simp*]: $x \,/\, 1 = (x{::}\mathit{ereal})$
⟨*proof*⟩

lemma *ereal-divide-ereal*[*simp*]: $\infty \,/\, \mathit{ereal}\ r = (\mathit{if}\ 0 \leq r\ \mathit{then}\ \infty\ \mathit{else}\ -\infty)$
⟨*proof*⟩

lemma *ereal-inverse-nonneg-iff*: $0 \leq \mathit{inverse}\ (x :: \mathit{ereal}) \longleftrightarrow 0 \leq x \vee x = -\infty$
⟨*proof*⟩

lemma *inverse-ereal-ge0I*: $0 \leq (x :: \mathit{ereal}) \Longrightarrow 0 \leq \mathit{inverse}\ x$
⟨*proof*⟩

lemma *zero-le-divide-ereal*[*simp*]:
fixes a :: *ereal*
assumes $0 \leq a$
and $0 \leq b$
shows $0 \leq a \,/\, b$
⟨*proof*⟩

lemma *ereal-le-divide-pos*:
fixes $x\ y\ z$ :: *ereal*
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow y \leq z \,/\, x \longleftrightarrow x * y \leq z$
⟨*proof*⟩

lemma *ereal-divide-le-pos*:
fixes $x\ y\ z :: ereal$
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow z\ /\ x \leq y \longleftrightarrow z \leq x * y$
⟨*proof*⟩

lemma *ereal-le-divide-neg*:
fixes $x\ y\ z :: ereal$
shows $x < 0 \Longrightarrow x \neq -\infty \Longrightarrow y \leq z\ /\ x \longleftrightarrow z \leq x * y$
⟨*proof*⟩

lemma *ereal-divide-le-neg*:
fixes $x\ y\ z :: ereal$
shows $x < 0 \Longrightarrow x \neq -\infty \Longrightarrow z\ /\ x \leq y \longleftrightarrow x * y \leq z$
⟨*proof*⟩

lemma *ereal-inverse-antimono-strict*:
fixes $x\ y :: ereal$
shows $0 \leq x \Longrightarrow x < y \Longrightarrow inverse\ y < inverse\ x$
⟨*proof*⟩

lemma *ereal-inverse-antimono*:
fixes $x\ y :: ereal$
shows $0 \leq x \Longrightarrow x \leq y \Longrightarrow inverse\ y \leq inverse\ x$
⟨*proof*⟩

lemma *inverse-inverse-Pinfty-iff* [*simp*]:
fixes $x :: ereal$
shows $inverse\ x = \infty \longleftrightarrow x = 0$
⟨*proof*⟩

lemma *ereal-inverse-eq-0*:
fixes $x :: ereal$
shows $inverse\ x = 0 \longleftrightarrow x = \infty \vee x = -\infty$
⟨*proof*⟩

lemma *ereal-0-gt-inverse*:
fixes $x :: ereal$
shows $0 < inverse\ x \longleftrightarrow x \neq \infty \wedge 0 \leq x$
⟨*proof*⟩

lemma *ereal-inverse-le-0-iff*:
fixes $x :: ereal$
shows $inverse\ x \leq 0 \longleftrightarrow x < 0 \vee x = \infty$
⟨*proof*⟩

lemma *ereal-divide-eq-0-iff*: $x\ /\ y = 0 \longleftrightarrow x = 0 \vee |y :: ereal| = \infty$
⟨*proof*⟩

lemma *ereal-mult-less-right*:
fixes a b c :: *ereal*
assumes $b * a < c * a$
and $0 < a$
and $a < \infty$
shows $b < c$
⟨*proof*⟩

lemma *ereal-mult-divide*: **fixes** a b :: *ereal* **shows** $0 < b \Longrightarrow b < \infty \Longrightarrow b * (a / b) = a$
⟨*proof*⟩

lemma *ereal-power-divide*:
fixes x y :: *ereal*
shows $y \neq 0 \Longrightarrow (x / y) \hat{} n = x \hat{} n / y \hat{} n$
⟨*proof*⟩

lemma *ereal-le-mult-one-interval*:
fixes x y :: *ereal*
assumes *y*: $y \neq -\infty$
assumes *z*: $\bigwedge z.\ 0 < z \Longrightarrow z < 1 \Longrightarrow z * x \leq y$
shows $x \leq y$
⟨*proof*⟩

lemma *ereal-divide-right-mono*[*simp*]:
fixes x y z :: *ereal*
assumes $x \leq y$
and $0 < z$
shows $x / z \leq y / z$
⟨*proof*⟩

lemma *ereal-divide-left-mono*[*simp*]:
fixes x y z :: *ereal*
assumes $y \leq x$
and $0 < z$
and $0 < x * y$
shows $z / x \leq z / y$
⟨*proof*⟩

lemma *ereal-divide-zero-left*[*simp*]:
fixes a :: *ereal*
shows $0 / a = 0$
⟨*proof*⟩

lemma *ereal-times-divide-eq-left*[*simp*]:
fixes a b c :: *ereal*
shows $b / c * a = b * a / c$
⟨*proof*⟩

lemma *ereal-times-divide-eq*: $a * (b \,/\, c :: ereal) = a * b \,/\, c$
⟨*proof*⟩

lemma *ereal-inverse-real*: $|z| \neq \infty \Longrightarrow z \neq 0 \Longrightarrow ereal\ (inverse\ (real\text{-}of\text{-}ereal\ z)) = inverse\ z$
⟨*proof*⟩

lemma *ereal-inverse-mult*:
$a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow inverse\ (a * (b{::}ereal)) = inverse\ a * inverse\ b$
⟨*proof*⟩

30.2 Complete lattice

instantiation *ereal* :: *lattice*
begin

definition [*simp*]: $sup\ x\ y = (max\ x\ y :: ereal)$
definition [*simp*]: $inf\ x\ y = (min\ x\ y :: ereal)$
instance ⟨*proof*⟩

end

instantiation *ereal* :: *complete-lattice*
begin

definition $bot = (-\infty{::}ereal)$
definition $top = (\infty{::}ereal)$

definition $Sup\ S = (SOME\ x :: ereal.\ (\forall\, y{\in}S.\ y \leq x) \wedge (\forall\, z.\ (\forall\, y{\in}S.\ y \leq z) \longrightarrow x \leq z))$
definition $Inf\ S = (SOME\ x :: ereal.\ (\forall\, y{\in}S.\ x \leq y) \wedge (\forall\, z.\ (\forall\, y{\in}S.\ z \leq y) \longrightarrow z \leq x))$

lemma *ereal-complete-Sup*:
fixes S :: *ereal set*
shows $\exists\, x.\ (\forall\, y{\in}S.\ y \leq x) \wedge (\forall\, z.\ (\forall\, y{\in}S.\ y \leq z) \longrightarrow x \leq z)$
⟨*proof*⟩

lemma *ereal-complete-uminus-eq*:
fixes S :: *ereal set*
shows $(\forall\, y{\in}uminus`S.\ y \leq x) \wedge (\forall\, z.\ (\forall\, y{\in}uminus`S.\ y \leq z) \longrightarrow x \leq z)$
$\longleftrightarrow (\forall\, y{\in}S.\ -x \leq y) \wedge (\forall\, z.\ (\forall\, y{\in}S.\ z \leq y) \longrightarrow z \leq -x)$
⟨*proof*⟩

lemma *ereal-complete-Inf*:
$\exists\, x.\ (\forall\, y{\in}S{::}ereal\ set.\ x \leq y) \wedge (\forall\, z.\ (\forall\, y{\in}S.\ z \leq y) \longrightarrow z \leq x)$
⟨*proof*⟩

instance

⟨*proof*⟩

end

instance *ereal* :: *complete-linorder* ⟨*proof*⟩

instance *ereal* :: *linear-continuum*
⟨*proof*⟩

30.2.1 Topological space

instantiation *ereal* :: *linear-continuum-topology*
begin

definition *open-ereal* :: *ereal set* ⇒ *bool* **where**
open-ereal-generated: *open-ereal* = *generate-topology* (*range lessThan* ∪ *range greaterThan*)

instance
⟨*proof*⟩

end

lemma *continuous-on-ereal*[*continuous-intros*]:
assumes *f*: *continuous-on s f* **shows** *continuous-on s* (λ*x*. *ereal* (*f x*))
⟨*proof*⟩

lemma *tendsto-ereal*[*tendsto-intros*, *simp*, *intro*]: (*f* ⟶ *x*) *F* ⟹ ((λ*x*. *ereal* (*f x*)) ⟶ *ereal x*) *F*
⟨*proof*⟩

lemma *tendsto-uminus-ereal*[*tendsto-intros*, *simp*, *intro*]: (*f* ⟶ *x*) *F* ⟹ ((λ*x*. − *f x*::*ereal*) ⟶ − *x*) *F*
⟨*proof*⟩

lemma *at-infty-ereal-eq-at-top*: *at* ∞ = *filtermap ereal at-top*
⟨*proof*⟩

lemma *ereal-Lim-uminus*: (*f* ⟶ *f0*) *net* ⟷ ((λ*x*. − *f x*::*ereal*) ⟶ − *f0*) *net*
⟨*proof*⟩

lemma *ereal-divide-less-iff*: *0* < (*c*::*ereal*) ⟹ *c* < ∞ ⟹ *a* / *c* < *b* ⟷ *a* < *b* ∗ *c*
⟨*proof*⟩

lemma *ereal-less-divide-iff*: *0* < (*c*::*ereal*) ⟹ *c* < ∞ ⟹ *a* < *b* / *c* ⟷ *a* ∗ *c* < *b*
⟨*proof*⟩

lemma *tendsto-cmult-ereal*[*tendsto-intros*, *simp*, *intro*]:
assumes *c*: $|c| \neq \infty$ **and** *f*: $(f \longrightarrow x)\ F$ **shows** $((\lambda x.\ c * f\ x{::}ereal) \longrightarrow c * x)\ F$
⟨*proof*⟩

lemma *tendsto-cmult-ereal-not-0*[*tendsto-intros*, *simp*, *intro*]:
assumes $x \neq 0$ **and** *f*: $(f \longrightarrow x)\ F$ **shows** $((\lambda x.\ c * f\ x{::}ereal) \longrightarrow c * x)\ F$
⟨*proof*⟩

lemma *tendsto-cadd-ereal*[*tendsto-intros*, *simp*, *intro*]:
assumes *c*: $y \neq -\infty\ x \neq -\infty$ **and** *f*: $(f \longrightarrow x)\ F$ **shows** $((\lambda x.\ f\ x + y{::}ereal) \longrightarrow x + y)\ F$
⟨*proof*⟩

lemma *tendsto-add-left-ereal*[*tendsto-intros*, *simp*, *intro*]:
assumes *c*: $|y| \neq \infty$ **and** *f*: $(f \longrightarrow x)\ F$ **shows** $((\lambda x.\ f\ x + y{::}ereal) \longrightarrow x + y)\ F$
⟨*proof*⟩

lemma *continuous-at-ereal*[*continuous-intros*]: *continuous F f* $\Longrightarrow$ *continuous F* $(\lambda x.\ ereal\ (f\ x))$
⟨*proof*⟩

lemma *ereal-Sup*:
assumes $*$: $|SUP\ a{:}A.\ ereal\ a| \neq \infty$
shows $ereal\ (Sup\ A) = (SUP\ a{:}A.\ ereal\ a)$
⟨*proof*⟩

lemma *ereal-SUP*: $|SUP\ a{:}A.\ ereal\ (f\ a)| \neq \infty \Longrightarrow ereal\ (SUP\ a{:}A.\ f\ a) = (SUP\ a{:}A.\ ereal\ (f\ a))$
⟨*proof*⟩

lemma *ereal-Inf*:
assumes $*$: $|INF\ a{:}A.\ ereal\ a| \neq \infty$
shows $ereal\ (Inf\ A) = (INF\ a{:}A.\ ereal\ a)$
⟨*proof*⟩

lemma *ereal-Inf'*:
assumes $*$: *bdd-below A A* $\neq \{\}$
shows $ereal\ (Inf\ A) = (INF\ a{:}A.\ ereal\ a)$
⟨*proof*⟩

lemma *ereal-INF*: $|INF\ a{:}A.\ ereal\ (f\ a)| \neq \infty \Longrightarrow ereal\ (INF\ a{:}A.\ f\ a) = (INF\ a{:}A.\ ereal\ (f\ a))$
⟨*proof*⟩

lemma *ereal-Sup-uminus-image-eq*: $Sup\ (uminus\ `\ S{::}ereal\ set) = -\ Inf\ S$

⟨*proof*⟩

lemma *ereal-SUP-uminus-eq*:
fixes $f :: 'a \Rightarrow ereal$
shows $(SUP\ x{:}S.\ uminus\ (f\ x)) = -\ (INF\ x{:}S.\ f\ x)$
⟨*proof*⟩

lemma *ereal-inj-on-uminus*[*intro*, *simp*]: *inj-on uminus* $(A :: ereal\ set)$
⟨*proof*⟩

lemma *ereal-Inf-uminus-image-eq*: $Inf\ (uminus\ `\ S{::}ereal\ set) = -\ Sup\ S$
⟨*proof*⟩

lemma *ereal-INF-uminus-eq*:
fixes $f :: 'a \Rightarrow ereal$
shows $(INF\ x{:}S.\ -\ f\ x) = -\ (SUP\ x{:}S.\ f\ x)$
⟨*proof*⟩

lemma *ereal-SUP-uminus*:
fixes $f :: 'a \Rightarrow ereal$
shows $(SUP\ i : R.\ -\ f\ i) = -\ (INF\ i : R.\ f\ i)$
⟨*proof*⟩

lemma *ereal-SUP-not-infty*:
fixes $f :: - \Rightarrow ereal$
shows $A \neq \{\} \Longrightarrow l \neq -\infty \Longrightarrow u \neq \infty \Longrightarrow \forall\, a{\in}A.\ l \leq f\ a \wedge f\ a \leq u \Longrightarrow |SUPREMUM\ A\ f| \neq \infty$
⟨*proof*⟩

lemma *ereal-INF-not-infty*:
fixes $f :: - \Rightarrow ereal$
shows $A \neq \{\} \Longrightarrow l \neq -\infty \Longrightarrow u \neq \infty \Longrightarrow \forall\, a{\in}A.\ l \leq f\ a \wedge f\ a \leq u \Longrightarrow |INFIMUM\ A\ f| \neq \infty$
⟨*proof*⟩

lemma *ereal-image-uminus-shift*:
fixes $X\ Y :: ereal\ set$
shows $uminus\ `\ X = Y \longleftrightarrow X = uminus\ `\ Y$
⟨*proof*⟩

lemma *Sup-eq-MInfty*:
fixes $S :: ereal\ set$
shows $Sup\ S = -\infty \longleftrightarrow S = \{\} \vee S = \{-\infty\}$
⟨*proof*⟩

lemma *Inf-eq-PInfty*:
fixes $S :: ereal\ set$
shows $Inf\ S = \infty \longleftrightarrow S = \{\} \vee S = \{\infty\}$
⟨*proof*⟩

lemma *Inf-eq-MInfty*:
fixes S :: *ereal set*
shows $-\infty \in S \Longrightarrow Inf\ S = -\infty$
⟨*proof*⟩

lemma *Sup-eq-PInfty*:
fixes S :: *ereal set*
shows $\infty \in S \Longrightarrow Sup\ S = \infty$
⟨*proof*⟩

lemma *not-MInfty-nonneg*[*simp*]: $0 \leq (x::ereal) \Longrightarrow x \neq -\infty$
⟨*proof*⟩

lemma *Sup-ereal-close*:
fixes e :: *ereal*
assumes $0 < e$
and S: $|Sup\ S| \neq \infty\ S \neq \{\}$
shows $\exists x{\in}S.\ Sup\ S - e < x$
⟨*proof*⟩

lemma *Inf-ereal-close*:
fixes e :: *ereal*
assumes $|Inf\ X| \neq \infty$
and $0 < e$
shows $\exists x{\in}X.\ x < Inf\ X + e$
⟨*proof*⟩

lemma *SUP-PInfty*:
$(\bigwedge n::nat.\ \exists i{\in}A.\ ereal\ (real\ n) \leq f\ i) \Longrightarrow (SUP\ i{:}A.\ f\ i :: ereal) = \infty$
⟨*proof*⟩

lemma *SUP-nat-Infty*: $(SUP\ i::nat.\ ereal\ (real\ i)) = \infty$
⟨*proof*⟩

lemma *SUP-ereal-add-left*:
assumes $I \neq \{\}\ c \neq -\infty$
shows $(SUP\ i{:}I.\ f\ i + c :: ereal) = (SUP\ i{:}I.\ f\ i) + c$
⟨*proof*⟩

lemma *SUP-ereal-add-right*:
fixes c :: *ereal*
shows $I \neq \{\} \Longrightarrow c \neq -\infty \Longrightarrow (SUP\ i{:}I.\ c + f\ i) = c + (SUP\ i{:}I.\ f\ i)$
⟨*proof*⟩

lemma *SUP-ereal-minus-right*:
assumes $I \neq \{\}\ c \neq -\infty$
shows $(SUP\ i{:}I.\ c - f\ i :: ereal) = c - (INF\ i{:}I.\ f\ i)$
⟨*proof*⟩

lemma *SUP-ereal-minus-left*:
assumes $I \neq \{\}$ $c \neq \infty$
shows $(SUP\ i{:}I.\ f\ i - c{::}\ ereal) = (SUP\ i{:}I.\ f\ i) - c$
⟨*proof*⟩

lemma *INF-ereal-minus-right*:
assumes $I \neq \{\}$ **and** $|c| \neq \infty$
shows $(INF\ i{:}I.\ c - f\ i) = c - (SUP\ i{:}I.\ f\ i{::}ereal)$
⟨*proof*⟩

lemma *SUP-ereal-le-addI*:
fixes $f :: 'i \Rightarrow ereal$
assumes $\bigwedge i.\ f\ i + y \leq z$ **and** $y \neq -\infty$
shows $SUPREMUM\ UNIV\ f + y \leq z$
⟨*proof*⟩

lemma *SUP-combine*:
fixes $f :: 'a{::}semilattice\text{-}sup \Rightarrow 'a{::}semilattice\text{-}sup \Rightarrow 'b{::}complete\text{-}lattice$
assumes *mono*: $\bigwedge a\ b\ c\ d.\ a \leq b \Longrightarrow c \leq d \Longrightarrow f\ a\ c \leq f\ b\ d$
shows $(SUP\ i{:}UNIV.\ SUP\ j{:}UNIV.\ f\ i\ j) = (SUP\ i.\ f\ i\ i)$
⟨*proof*⟩

lemma *SUP-ereal-add*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes *inc*: *incseq f incseq g*
and *pos*: $\bigwedge i.\ f\ i \neq -\infty\ \bigwedge i.\ g\ i \neq -\infty$
shows $(SUP\ i.\ f\ i + g\ i) = SUPREMUM\ UNIV\ f + SUPREMUM\ UNIV\ g$
⟨*proof*⟩

lemma *INF-eq-minf*: $(INF\ i{:}I.\ f\ i{::}ereal) \neq -\infty \longleftrightarrow (\exists\, b{>}-\infty.\ \forall\, i{\in}I.\ b \leq f\ i)$
⟨*proof*⟩

lemma *INF-ereal-add-left*:
assumes $I \neq \{\}$ $c \neq -\infty$ $\bigwedge x.\ x \in I \Longrightarrow 0 \leq f\ x$
shows $(INF\ i{:}I.\ f\ i + c :: ereal) = (INF\ i{:}I.\ f\ i) + c$
⟨*proof*⟩

lemma *INF-ereal-add-right*:
assumes $I \neq \{\}$ $c \neq -\infty$ $\bigwedge x.\ x \in I \Longrightarrow 0 \leq f\ x$
shows $(INF\ i{:}I.\ c + f\ i :: ereal) = c + (INF\ i{:}I.\ f\ i)$
⟨*proof*⟩

lemma *INF-ereal-add-directed*:
fixes $f\ g :: 'a \Rightarrow ereal$
assumes *nonneg*: $\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i\ \bigwedge i.\ i \in I \Longrightarrow 0 \leq g\ i$
assumes *directed*: $\bigwedge i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow \exists\, k{\in}I.\ f\ i + g\ j \geq f\ k + g\ k$
shows $(INF\ i{:}I.\ f\ i + g\ i) = (INF\ i{:}I.\ f\ i) + (INF\ i{:}I.\ g\ i)$
⟨*proof*⟩

lemma *INF-ereal-add*:
fixes $f :: nat \Rightarrow ereal$
assumes *decseq f decseq g*
and *fin*: $\bigwedge i.\ f\ i \neq \infty\ \bigwedge i.\ g\ i \neq \infty$
shows $(INF\ i.\ f\ i + g\ i) = INFIMUM\ UNIV\ f + INFIMUM\ UNIV\ g$
⟨*proof*⟩

lemma *SUP-ereal-add-pos*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes *inc*: *incseq f incseq g*
and *pos*: $\bigwedge i.\ 0 \leq f\ i\ \bigwedge i.\ 0 \leq g\ i$
shows $(SUP\ i.\ f\ i + g\ i) = SUPREMUM\ UNIV\ f + SUPREMUM\ UNIV\ g$
⟨*proof*⟩

lemma *SUP-ereal-sum*:
fixes $f\ g :: 'a \Rightarrow nat \Rightarrow ereal$
assumes $\bigwedge n.\ n \in A \Longrightarrow incseq\ (f\ n)$
and *pos*: $\bigwedge n\ i.\ n \in A \Longrightarrow 0 \leq f\ n\ i$
shows $(SUP\ i.\ \sum n{\in}A.\ f\ n\ i) = (\sum n{\in}A.\ SUPREMUM\ UNIV\ (f\ n))$
⟨*proof*⟩

lemma *SUP-ereal-mult-left*:
fixes $f :: 'a \Rightarrow ereal$
assumes $I \neq \{\}$
assumes f: $\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i$ **and** c: $0 \leq c$
shows $(SUP\ i{:}I.\ c * f\ i) = c * (SUP\ i{:}I.\ f\ i)$
⟨*proof*⟩

lemma *countable-approach*:
fixes $x :: ereal$
assumes $x \neq -\infty$
shows $\exists f.\ incseq\ f \wedge (\forall i{::}nat.\ f\ i < x) \wedge (f \longrightarrow x)$
⟨*proof*⟩

lemma *Sup-countable-SUP*:
assumes $A \neq \{\}$
shows $\exists f{::}nat \Rightarrow ereal.\ incseq\ f \wedge range\ f \subseteq A \wedge Sup\ A = (SUP\ i.\ f\ i)$
⟨*proof*⟩

lemma *Inf-countable-INF*:
assumes $A \neq \{\}$ **shows** $\exists f{::}nat \Rightarrow ereal.\ decseq\ f \wedge range\ f \subseteq A \wedge Inf\ A = (INF\ i.\ f\ i)$
⟨*proof*⟩

lemma *SUP-countable-SUP*:
$A \neq \{\} \Longrightarrow \exists f{::}nat \Rightarrow ereal.\ range\ f \subseteq g`A \wedge SUPREMUM\ A\ g = SUPREMUM\ UNIV\ f$
⟨*proof*⟩

30.3 Relation to *enat*

definition *ereal-of-enat n* = (*case n of enat n* ⇒ *ereal* (*real n*) | ∞ ⇒ ∞)

declare [[*coercion ereal-of-enat* :: *enat* ⇒ *ereal*]]
declare [[*coercion* (λ*n*. *ereal* (*real n*)) :: *nat* ⇒ *ereal*]]

lemma *ereal-of-enat-simps*[*simp*]:
ereal-of-enat (*enat n*) = *ereal n*
ereal-of-enat ∞ = ∞
⟨*proof*⟩

lemma *ereal-of-enat-le-iff*[*simp*]: *ereal-of-enat m* ≤ *ereal-of-enat n* ⟷ *m* ≤ *n*
⟨*proof*⟩

lemma *ereal-of-enat-less-iff*[*simp*]: *ereal-of-enat m* < *ereal-of-enat n* ⟷ *m* < *n*
⟨*proof*⟩

lemma *numeral-le-ereal-of-enat-iff*[*simp*]: *numeral m* ≤ *ereal-of-enat n* ⟷ *numeral m* ≤ *n*
⟨*proof*⟩

lemma *numeral-less-ereal-of-enat-iff*[*simp*]: *numeral m* < *ereal-of-enat n* ⟷ *numeral m* < *n*
⟨*proof*⟩

lemma *ereal-of-enat-ge-zero-cancel-iff*[*simp*]: *0* ≤ *ereal-of-enat n* ⟷ *0* ≤ *n*
⟨*proof*⟩

lemma *ereal-of-enat-gt-zero-cancel-iff*[*simp*]: *0* < *ereal-of-enat n* ⟷ *0* < *n*
⟨*proof*⟩

lemma *ereal-of-enat-zero*[*simp*]: *ereal-of-enat 0* = *0*
⟨*proof*⟩

lemma *ereal-of-enat-inf*[*simp*]: *ereal-of-enat n* = ∞ ⟷ *n* = ∞
⟨*proof*⟩

lemma *ereal-of-enat-add*: *ereal-of-enat* (*m* + *n*) = *ereal-of-enat m* + *ereal-of-enat n*
⟨*proof*⟩

lemma *ereal-of-enat-sub*:
assumes *n* ≤ *m*
shows *ereal-of-enat* (*m* − *n*) = *ereal-of-enat m* − *ereal-of-enat n*
⟨*proof*⟩

lemma *ereal-of-enat-mult*:
ereal-of-enat (*m* ∗ *n*) = *ereal-of-enat m* ∗ *ereal-of-enat n*
⟨*proof*⟩

lemmas *ereal-of-enat-pushin* = *ereal-of-enat-add ereal-of-enat-sub ereal-of-enat-mult*
lemmas *ereal-of-enat-pushout* = *ereal-of-enat-pushin*[*symmetric*]

lemma *ereal-of-enat-nonneg*: *ereal-of-enat n* ≥ *0*
⟨*proof*⟩

lemma *ereal-of-enat-Sup*:
assumes $A \neq \{\}$ **shows** *ereal-of-enat* (*Sup A*) = (*SUP a* : *A*. *ereal-of-enat a*)
⟨*proof*⟩

lemma *ereal-of-enat-SUP*:
$A \neq \{\} \Longrightarrow$ *ereal-of-enat* (*SUP a*:*A*. *f a*) = (*SUP a* : *A*. *ereal-of-enat* (*f a*))
⟨*proof*⟩

30.4 Limits on *ereal*

lemma *open-PInfty*: *open A* ⟹ ∞ ∈ *A* ⟹ (∃ *x*. {*ereal x*<..} ⊆ *A*)
⟨*proof*⟩

lemma *open-MInfty*: *open A* ⟹ −∞ ∈ *A* ⟹ (∃ *x*. {..<*ereal x*} ⊆ *A*)
⟨*proof*⟩

lemma *open-ereal-vimage*: *open S* ⟹ *open* (*ereal* −‘ *S*)
⟨*proof*⟩

lemma *open-ereal*: *open S* ⟹ *open* (*ereal* ‘ *S*)
⟨*proof*⟩

lemma *eventually-finite*:
fixes *x* :: *ereal*
assumes |*x*| ≠ ∞ (*f* ⟶ *x*) *F*
shows *eventually* (λ*x*. |*f x*| ≠ ∞) *F*
⟨*proof*⟩

lemma *open-ereal-def*:
open A ⟷ *open* (*ereal* −‘ *A*) ∧ (∞ ∈ *A* ⟶ (∃ *x*. {*ereal x* <..} ⊆ *A*)) ∧ (−∞ ∈ *A* ⟶ (∃ *x*. {..<*ereal x*} ⊆ *A*))
(**is** *open A* ⟷ *?rhs*)
⟨*proof*⟩

lemma *open-PInfty2*:
assumes *open A*
and ∞ ∈ *A*
obtains *x* **where** {*ereal x*<..} ⊆ *A*
⟨*proof*⟩

lemma *open-MInfty2*:
assumes *open A*
and $-\infty \in A$
obtains *x* **where** $\{..<ereal\ x\} \subseteq A$
⟨*proof*⟩

lemma *ereal-openE*:
assumes *open A*
obtains *x y* **where** *open* (*ereal* −‘ *A*)
and $\infty \in A \Longrightarrow \{ereal\ x<..\} \subseteq A$
and $-\infty \in A \Longrightarrow \{..<ereal\ y\} \subseteq A$
⟨*proof*⟩

lemmas *open-ereal-lessThan* = *open-lessThan*[**where** *'a*=*ereal*]
lemmas *open-ereal-greaterThan* = *open-greaterThan*[**where** *'a*=*ereal*]
lemmas *ereal-open-greaterThanLessThan* = *open-greaterThanLessThan*[**where** *'a*=*ereal*]
lemmas *closed-ereal-atLeast* = *closed-atLeast*[**where** *'a*=*ereal*]
lemmas *closed-ereal-atMost* = *closed-atMost*[**where** *'a*=*ereal*]
lemmas *closed-ereal-atLeastAtMost* = *closed-atLeastAtMost*[**where** *'a*=*ereal*]
lemmas *closed-ereal-singleton* = *closed-singleton*[**where** *'a*=*ereal*]

lemma *ereal-open-cont-interval*:
fixes *S* :: *ereal set*
assumes *open S*
and $x \in S$
and $|x| \neq \infty$
obtains *e* **where** $e > 0$ **and** $\{x-e <..< x+e\} \subseteq S$
⟨*proof*⟩

lemma *ereal-open-cont-interval2*:
fixes *S* :: *ereal set*
assumes *open S*
and $x \in S$
and *x*: $|x| \neq \infty$
obtains *a b* **where** $a < x$ **and** $x < b$ **and** $\{a <..< b\} \subseteq S$
⟨*proof*⟩

30.4.1 Convergent sequences

lemma *lim-real-of-ereal*[*simp*]:
assumes *lim*: $(f \longrightarrow ereal\ x)\ net$
shows $((\lambda x.\ real\text{-}of\text{-}ereal\ (f\ x)) \longrightarrow x)\ net$
⟨*proof*⟩

lemma *lim-ereal*[*simp*]: $((\lambda n.\ ereal\ (f\ n)) \longrightarrow ereal\ x)\ net \longleftrightarrow (f \longrightarrow x)\ net$
⟨*proof*⟩

lemma *convergent-real-imp-convergent-ereal*:
assumes *convergent a*

shows *convergent* $(\lambda n.\ ereal\ (a\ n))$ **and** $lim\ (\lambda n.\ ereal\ (a\ n)) = ereal\ (lim\ a)$
⟨*proof*⟩

lemma *tendsto-PInfty*: $(f \longrightarrow \infty)\ F \longleftrightarrow (\forall r.\ eventually\ (\lambda x.\ ereal\ r < f\ x)\ F)$
⟨*proof*⟩

lemma *tendsto-PInfty'*: $(f \longrightarrow \infty)\ F = (\forall r{>}c.\ eventually\ (\lambda x.\ ereal\ r < f\ x)\ F)$
⟨*proof*⟩

lemma *tendsto-PInfty-eq-at-top*:
$((\lambda z.\ ereal\ (f\ z)) \longrightarrow \infty)\ F \longleftrightarrow (LIM\ z\ F.\ f\ z :> at\text{-}top)$
⟨*proof*⟩

lemma *tendsto-MInfty*: $(f \longrightarrow -\infty)\ F \longleftrightarrow (\forall r.\ eventually\ (\lambda x.\ f\ x < ereal\ r)\ F)$
⟨*proof*⟩

lemma *tendsto-MInfty'*: $(f \longrightarrow -\infty)\ F = (\forall r{<}c.\ eventually\ (\lambda x.\ ereal\ r > f\ x)\ F)$
⟨*proof*⟩

lemma *Lim-PInfty*: $f \longrightarrow \infty \longleftrightarrow (\forall B.\ \exists N.\ \forall n{\geq}N.\ f\ n \geq ereal\ B)$
⟨*proof*⟩

lemma *Lim-MInfty*: $f \longrightarrow -\infty \longleftrightarrow (\forall B.\ \exists N.\ \forall n{\geq}N.\ ereal\ B \geq f\ n)$
⟨*proof*⟩

lemma *Lim-bounded-PInfty*: $f \longrightarrow l \Longrightarrow (\bigwedge n.\ f\ n \leq ereal\ B) \Longrightarrow l \neq \infty$
⟨*proof*⟩

lemma *Lim-bounded-MInfty*: $f \longrightarrow l \Longrightarrow (\bigwedge n.\ ereal\ B \leq f\ n) \Longrightarrow l \neq -\infty$
⟨*proof*⟩

lemma *tendsto-zero-erealI*:
assumes $\bigwedge e.\ e > 0 \Longrightarrow eventually\ (\lambda x.\ |f\ x| < ereal\ e)\ F$
shows $(f \longrightarrow 0)\ F$
⟨*proof*⟩

lemma *tendsto-explicit*:
$f \longrightarrow f0 \longleftrightarrow (\forall S.\ open\ S \longrightarrow f0 \in S \longrightarrow (\exists N.\ \forall n{\geq}N.\ f\ n \in S))$
⟨*proof*⟩

lemma *Lim-bounded-PInfty2*: $f \longrightarrow l \Longrightarrow \forall n{\geq}N.\ f\ n \leq ereal\ B \Longrightarrow l \neq \infty$
⟨*proof*⟩

lemma *Lim-bounded-ereal*: $f \longrightarrow (l :: 'a{::}linorder\text{-}topology) \Longrightarrow \forall n{\geq}M.\ f\ n \leq C \Longrightarrow l \leq C$
⟨*proof*⟩

lemma *Lim-bounded2-ereal*:
assumes *lim*:$f \longrightarrow (l :: 'a::linorder\text{-}topology)$
and *ge*: $\forall n \geq N.\ f\ n \geq C$
shows $l \geq C$
⟨*proof*⟩

lemma *real-of-ereal-mult*[*simp*]:
fixes $a\ b :: ereal$
shows *real-of-ereal* $(a * b) =$ *real-of-ereal* $a *$ *real-of-ereal* b
⟨*proof*⟩

lemma *real-of-ereal-eq-0*:
fixes $x :: ereal$
shows *real-of-ereal* $x = 0 \longleftrightarrow x = \infty \vee x = -\infty \vee x = 0$
⟨*proof*⟩

lemma *tendsto-ereal-realD*:
fixes $f :: 'a \Rightarrow ereal$
assumes $x \neq 0$
and *tendsto*: $((\lambda x.\ ereal\ (real\text{-}of\text{-}ereal\ (f\ x))) \longrightarrow x)\ net$
shows $(f \longrightarrow x)\ net$
⟨*proof*⟩

lemma *tendsto-ereal-realI*:
fixes $f :: 'a \Rightarrow ereal$
assumes x: $|x| \neq \infty$ **and** *tendsto*: $(f \longrightarrow x)\ net$
shows $((\lambda x.\ ereal\ (real\text{-}of\text{-}ereal\ (f\ x))) \longrightarrow x)\ net$
⟨*proof*⟩

lemma *ereal-mult-cancel-left*:
fixes $a\ b\ c :: ereal$
shows $a * b = a * c \longleftrightarrow (|a| = \infty \wedge 0 < b * c) \vee a = 0 \vee b = c$
⟨*proof*⟩

lemma *tendsto-add-ereal*:
fixes $x\ y :: ereal$
assumes x: $|x| \neq \infty$ **and** y: $|y| \neq \infty$
assumes f: $(f \longrightarrow x)\ F$ **and** g: $(g \longrightarrow y)\ F$
shows $((\lambda x.\ f\ x + g\ x) \longrightarrow x + y)\ F$
⟨*proof*⟩

lemma *tendsto-add-ereal-nonneg*:
fixes $x\ y :: ereal$
assumes $x \neq -\infty\ y \neq -\infty\ (f \longrightarrow x)\ F\ (g \longrightarrow y)\ F$
shows $((\lambda x.\ f\ x + g\ x) \longrightarrow x + y)\ F$
⟨*proof*⟩

lemma *ereal-inj-affinity*:

fixes $m\ t :: ereal$
assumes $|m| \neq \infty$
and $m \neq 0$
and $|t| \neq \infty$
shows *inj-on* $(\lambda x.\ m * x + t)\ A$
⟨*proof*⟩

lemma *ereal-PInfty-eq-plus*[*simp*]:
fixes $a\ b :: ereal$
shows $\infty = a + b \longleftrightarrow a = \infty \vee b = \infty$
⟨*proof*⟩

lemma *ereal-MInfty-eq-plus*[*simp*]:
fixes $a\ b :: ereal$
shows $-\infty = a + b \longleftrightarrow (a = -\infty \wedge b \neq \infty) \vee (b = -\infty \wedge a \neq \infty)$
⟨*proof*⟩

lemma *ereal-less-divide-pos*:
fixes $x\ y :: ereal$
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow y < z\ /\ x \longleftrightarrow x * y < z$
⟨*proof*⟩

lemma *ereal-divide-less-pos*:
fixes $x\ y\ z :: ereal$
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow y\ /\ x < z \longleftrightarrow y < x * z$
⟨*proof*⟩

lemma *ereal-divide-eq*:
fixes $a\ b\ c :: ereal$
shows $b \neq 0 \Longrightarrow |b| \neq \infty \Longrightarrow a\ /\ b = c \longleftrightarrow a = b * c$
⟨*proof*⟩

lemma *ereal-inverse-not-MInfty*[*simp*]: $inverse\ (a{::}ereal) \neq -\infty$
⟨*proof*⟩

lemma *ereal-mult-m1*[*simp*]: $x * ereal\ (-1) = -x$
⟨*proof*⟩

lemma *ereal-real'*:
assumes $|x| \neq \infty$
shows $ereal\ (real\text{-}of\text{-}ereal\ x) = x$
⟨*proof*⟩

lemma *real-ereal-id*: $real\text{-}of\text{-}ereal \circ ereal = id$
⟨*proof*⟩

lemma *open-image-ereal*: $open(UNIV - \{\ \infty\ ,\ (-\infty :: ereal)\})$
⟨*proof*⟩

lemma *ereal-le-distrib*:
fixes $a\ b\ c$ :: *ereal*
shows $c * (a + b) \leq c * a + c * b$
⟨*proof*⟩

lemma *ereal-pos-distrib*:
fixes $a\ b\ c$ :: *ereal*
assumes $0 \leq c$
and $c \neq \infty$
shows $c * (a + b) = c * a + c * b$
⟨*proof*⟩

lemma *ereal-max-mono*: $(a{::}ereal) \leq b \Longrightarrow c \leq d \Longrightarrow max\ a\ c \leq max\ b\ d$
⟨*proof*⟩

lemma *ereal-max-least*: $(a{::}ereal) \leq x \Longrightarrow c \leq x \Longrightarrow max\ a\ c \leq x$
⟨*proof*⟩

lemma *ereal-LimI-finite*:
fixes x :: *ereal*
assumes $|x| \neq \infty$
and $\bigwedge r.\ 0 < r \Longrightarrow \exists N.\ \forall n{\geq}N.\ u\ n < x + r \wedge x < u\ n + r$
shows $u \longrightarrow x$
⟨*proof*⟩

lemma *tendsto-obtains-N*:
assumes $f \longrightarrow f0$
assumes *open* S
and $f0 \in S$
obtains N **where** $\forall n{\geq}N.\ f\ n \in S$
⟨*proof*⟩

lemma *ereal-LimI-finite-iff*:
fixes x :: *ereal*
assumes $|x| \neq \infty$
shows $u \longrightarrow x \longleftrightarrow (\forall r.\ 0 < r \longrightarrow (\exists N.\ \forall n{\geq}N.\ u\ n < x + r \wedge x < u\ n + r))$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *ereal-Limsup-uminus*:
fixes f :: $'a \Rightarrow ereal$
shows *Limsup net* $(\lambda x.\ -\ (f\ x)) = -\ Liminf\ net\ f$
⟨*proof*⟩

lemma *liminf-bounded-iff*:
fixes x :: $nat \Rightarrow ereal$
shows $C \leq liminf\ x \longleftrightarrow (\forall B{<}C.\ \exists N.\ \forall n{\geq}N.\ B < x\ n)$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)

⟨*proof*⟩

lemma *Liminf-add-le*:
fixes *f g* :: - ⇒ *ereal*
assumes *F*: *F* ≠ *bot*
assumes *ev*: *eventually* (λ*x*. 0 ≤ *f x*) *F eventually* (λ*x*. 0 ≤ *g x*) *F*
shows *Liminf F f* + *Liminf F g* ≤ *Liminf F* (λ*x*. *f x* + *g x*)
⟨*proof*⟩

lemma *Sup-ereal-mult-right'*:
assumes *nonempty*: *Y* ≠ {}
and *x*: *x* ≥ 0
shows (*SUP i*:*Y*. *f i*) ∗ *ereal x* = (*SUP i*:*Y*. *f i* ∗ *ereal x*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *Sup-ereal-mult-left'*:
⟦ *Y* ≠ {}; *x* ≥ 0 ⟧ ⟹ *ereal x* ∗ (*SUP i*:*Y*. *f i*) = (*SUP i*:*Y*. *ereal x* ∗ *f i*)
⟨*proof*⟩

lemma *sup-continuous-add*[*order-continuous-intros*]:
fixes *f g* :: 'a::*complete-lattice* ⇒ *ereal*
assumes *nn*: ⋀*x*. 0 ≤ *f x* ⋀*x*. 0 ≤ *g x* **and** *cont*: *sup-continuous f sup-continuous g*
shows *sup-continuous* (λ*x*. *f x* + *g x*)
⟨*proof*⟩

lemma *sup-continuous-mult-right*[*order-continuous-intros*]:
0 ≤ *c* ⟹ *c* < ∞ ⟹ *sup-continuous f* ⟹ *sup-continuous* (λ*x*. *f x* ∗ *c* :: *ereal*)
⟨*proof*⟩

lemma *sup-continuous-mult-left*[*order-continuous-intros*]:
0 ≤ *c* ⟹ *c* < ∞ ⟹ *sup-continuous f* ⟹ *sup-continuous* (λ*x*. *c* ∗ *f x* :: *ereal*)
⟨*proof*⟩

lemma *sup-continuous-ereal-of-enat*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* (λ*x*. *ereal-of-enat* (*f x*))
⟨*proof*⟩

30.4.2 Sums

lemma *sums-ereal-positive*:
fixes *f* :: *nat* ⇒ *ereal*
assumes ⋀*i*. 0 ≤ *f i*
shows *f sums* (*SUP n*. ∑ *i*<*n*. *f i*)
⟨*proof*⟩

lemma *summable-ereal-pos*:
fixes *f* :: *nat* ⇒ *ereal*
assumes ⋀*i*. 0 ≤ *f i*

shows *summable f*
⟨*proof*⟩

lemma *sums-ereal*: $(\lambda x.\ ereal\ (f\ x))\ sums\ ereal\ x \longleftrightarrow f\ sums\ x$
⟨*proof*⟩

lemma *suminf-ereal-eq-SUP*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \le f\ i$
shows $(\sum x.\ f\ x) = (SUP\ n.\ \sum i{<}n.\ f\ i)$
⟨*proof*⟩

lemma *suminf-bound*:
fixes $f :: nat \Rightarrow ereal$
assumes $\forall N.\ (\sum n{<}N.\ f\ n) \le x$
and *pos*: $\bigwedge n.\ 0 \le f\ n$
shows $suminf\ f \le x$
⟨*proof*⟩

lemma *suminf-bound-add*:
fixes $f :: nat \Rightarrow ereal$
assumes $\forall N.\ (\sum n{<}N.\ f\ n) + y \le x$
and *pos*: $\bigwedge n.\ 0 \le f\ n$
and $y \neq -\infty$
shows $suminf\ f + y \le x$
⟨*proof*⟩

lemma *suminf-upper*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge n.\ 0 \le f\ n$
shows $(\sum n{<}N.\ f\ n) \le (\sum n.\ f\ n)$
⟨*proof*⟩

lemma *suminf-0-le*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge n.\ 0 \le f\ n$
shows $0 \le (\sum n.\ f\ n)$
⟨*proof*⟩

lemma *suminf-le-pos*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge N.\ f\ N \le g\ N$
and $\bigwedge N.\ 0 \le f\ N$
shows $suminf\ f \le suminf\ g$
⟨*proof*⟩

lemma *suminf-half-series-ereal*: $(\sum n.\ (1/2 :: ereal)\ \hat{}\ Suc\ n) = 1$
⟨*proof*⟩

lemma *suminf-add-ereal*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \leq f\ i$
and $\bigwedge i.\ 0 \leq g\ i$
shows $(\sum i.\ f\ i + g\ i) = suminf\ f + suminf\ g$
⟨*proof*⟩

lemma *suminf-cmult-ereal*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \leq f\ i$
and $0 \leq a$
shows $(\sum i.\ a * f\ i) = a * suminf\ f$
⟨*proof*⟩

lemma *suminf-PInfty*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \leq f\ i$
and $suminf\ f \neq \infty$
shows $f\ i \neq \infty$
⟨*proof*⟩

lemma *suminf-PInfty-fun*:
assumes $\bigwedge i.\ 0 \leq f\ i$
and $suminf\ f \neq \infty$
shows $\exists f'.\ f = (\lambda x.\ ereal\ (f'\ x))$
⟨*proof*⟩

lemma *summable-ereal*:
assumes $\bigwedge i.\ 0 \leq f\ i$
and $(\sum i.\ ereal\ (f\ i)) \neq \infty$
shows *summable f*
⟨*proof*⟩

lemma *suminf-ereal*:
assumes $\bigwedge i.\ 0 \leq f\ i$
and $(\sum i.\ ereal\ (f\ i)) \neq \infty$
shows $(\sum i.\ ereal\ (f\ i)) = ereal\ (suminf\ f)$
⟨*proof*⟩

lemma *suminf-ereal-minus*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes *ord*: $\bigwedge i.\ g\ i \leq f\ i\ \bigwedge i.\ 0 \leq g\ i$
and *fin*: $suminf\ f \neq \infty\ suminf\ g \neq \infty$
shows $(\sum i.\ f\ i - g\ i) = suminf\ f - suminf\ g$
⟨*proof*⟩

lemma *suminf-ereal-PInf* [*simp*]: $(\sum x.\ \infty :: ereal) = \infty$
⟨*proof*⟩

lemma *summable-real-of-ereal*:
fixes $f :: nat \Rightarrow ereal$
assumes f: $\bigwedge i.\ 0 \leq f\ i$
and *fin*: $(\sum i.\ f\ i) \neq \infty$
shows *summable* $(\lambda i.\ real\text{-}of\text{-}ereal\ (f\ i))$
⟨*proof*⟩

lemma *suminf-SUP-eq*:
fixes $f :: nat \Rightarrow nat \Rightarrow ereal$
assumes $\bigwedge i.$ *incseq* $(\lambda n.\ f\ n\ i)$
and $\bigwedge n\ i.\ 0 \leq f\ n\ i$
shows $(\sum i.\ SUP\ n.\ f\ n\ i) = (SUP\ n.\ \sum i.\ f\ n\ i)$
⟨*proof*⟩

lemma *suminf-sum-ereal*:
fixes $f :: - \Rightarrow - \Rightarrow ereal$
assumes *nonneg*: $\bigwedge i\ a.\ a \in A \Longrightarrow 0 \leq f\ i\ a$
shows $(\sum i.\ \sum a{\in}A.\ f\ i\ a) = (\sum a{\in}A.\ \sum i.\ f\ i\ a)$
⟨*proof*⟩

lemma *suminf-ereal-eq-0*:
fixes $f :: nat \Rightarrow ereal$
assumes *nneg*: $\bigwedge i.\ 0 \leq f\ i$
shows $(\sum i.\ f\ i) = 0 \longleftrightarrow (\forall\, i.\ f\ i = 0)$
⟨*proof*⟩

lemma *suminf-ereal-offset-le*:
fixes $f :: nat \Rightarrow ereal$
assumes f: $\bigwedge i.\ 0 \leq f\ i$
shows $(\sum i.\ f\ (i + k)) \leq suminf\ f$
⟨*proof*⟩

lemma *sums-suminf-ereal*: f *sums* $x \Longrightarrow (\sum i.\ ereal\ (f\ i)) = ereal\ x$
⟨*proof*⟩

lemma *suminf-ereal'*: *summable* $f \Longrightarrow (\sum i.\ ereal\ (f\ i)) = ereal\ (\sum i.\ f\ i)$
⟨*proof*⟩

lemma *suminf-ereal-finite*: *summable* $f \Longrightarrow (\sum i.\ ereal\ (f\ i)) \neq \infty$
⟨*proof*⟩

lemma *suminf-ereal-finite-neg*:
assumes *summable* f
shows $(\sum x.\ ereal\ (f\ x)) \neq -\infty$
⟨*proof*⟩

lemma *SUP-ereal-add-directed*:
fixes $f\ g :: 'a \Rightarrow ereal$
assumes *nonneg*: $\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i\ \bigwedge i.\ i \in I \Longrightarrow 0 \leq g\ i$

assumes *directed*: $\bigwedge i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow \exists k{\in}I.\ f\ i + g\ j \leq f\ k + g\ k$
shows $(SUP\ i{:}I.\ f\ i + g\ i) = (SUP\ i{:}I.\ f\ i) + (SUP\ i{:}I.\ g\ i)$
⟨*proof*⟩

lemma *SUP-ereal-sum-directed*:
fixes $f\ g :: 'a \Rightarrow 'b \Rightarrow ereal$
assumes $I \neq \{\}$
assumes *directed*: $\bigwedge N\ i\ j.\ N \subseteq A \Longrightarrow i \in I \Longrightarrow j \in I \Longrightarrow \exists k{\in}I.\ \forall n{\in}N.\ f\ n\ i \leq f\ n\ k \wedge f\ n\ j \leq f\ n\ k$
assumes *nonneg*: $\bigwedge n\ i.\ i \in I \Longrightarrow n \in A \Longrightarrow 0 \leq f\ n\ i$
shows $(SUP\ i{:}I.\ \sum n{\in}A.\ f\ n\ i) = (\sum n{\in}A.\ SUP\ i{:}I.\ f\ n\ i)$
⟨*proof*⟩

lemma *suminf-SUP-eq-directed*:
fixes $f :: - \Rightarrow nat \Rightarrow ereal$
assumes $I \neq \{\}$
assumes *directed*: $\bigwedge N\ i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow finite\ N \Longrightarrow \exists k{\in}I.\ \forall n{\in}N.\ f\ i\ n \leq f\ k\ n \wedge f\ j\ n \leq f\ k\ n$
assumes *nonneg*: $\bigwedge n\ i.\ 0 \leq f\ n\ i$
shows $(\sum i.\ SUP\ n{:}I.\ f\ n\ i) = (SUP\ n{:}I.\ \sum i.\ f\ n\ i)$
⟨*proof*⟩

lemma *ereal-dense3*:
fixes $x\ y :: ereal$
shows $x < y \Longrightarrow \exists r{::}rat.\ x < real\text{-}of\text{-}rat\ r \wedge real\text{-}of\text{-}rat\ r < y$
⟨*proof*⟩

lemma *continuous-within-ereal*[*intro*, *simp*]: $x \in A \Longrightarrow continuous\ (at\ x\ within\ A)\ ereal$
⟨*proof*⟩

lemma *ereal-open-uminus*:
fixes $S :: ereal\ set$
assumes *open S*
shows *open* $(uminus\ `\ S)$
⟨*proof*⟩

lemma *ereal-uminus-complement*:
fixes $S :: ereal\ set$
shows $uminus\ `\ (-\ S) = -\ uminus\ `\ S$
⟨*proof*⟩

lemma *ereal-closed-uminus*:
fixes $S :: ereal\ set$
assumes *closed S*
shows *closed* $(uminus\ `\ S)$
⟨*proof*⟩

lemma *ereal-open-affinity-pos*:

fixes S :: *ereal set*
assumes *open* S
and m: $m \neq \infty$ $0 < m$
and t: $|t| \neq \infty$
shows *open* $((\lambda x.\ m * x + t)\ `\ S)$
⟨*proof*⟩

lemma *ereal-open-affinity*:
fixes S :: *ereal set*
assumes *open* S
and m: $|m| \neq \infty$ $m \neq 0$
and t: $|t| \neq \infty$
shows *open* $((\lambda x.\ m * x + t)\ `\ S)$
⟨*proof*⟩

lemma *open-uminus-iff*:
fixes S :: *ereal set*
shows *open* $(uminus\ `\ S) \longleftrightarrow$ *open* S
⟨*proof*⟩

lemma *ereal-Liminf-uminus*:
fixes f :: $'a \Rightarrow$ *ereal*
shows *Liminf net* $(\lambda x.\ -\ (f\ x)) = -$ *Limsup net f*
⟨*proof*⟩

lemma *Liminf-PInfty*:
fixes f :: $'a \Rightarrow$ *ereal*
assumes ¬ *trivial-limit net*
shows $(f \longrightarrow \infty)$ *net* $\longleftrightarrow$ *Liminf net f* $= \infty$
⟨*proof*⟩

lemma *Limsup-MInfty*:
fixes f :: $'a \Rightarrow$ *ereal*
assumes ¬ *trivial-limit net*
shows $(f \longrightarrow -\infty)$ *net* $\longleftrightarrow$ *Limsup net f* $= -\infty$
⟨*proof*⟩

lemma *convergent-ereal*: — RENAME
fixes X :: *nat* $\Rightarrow$ $'a$ :: {*complete-linorder*,*linorder-topology*}
shows *convergent* $X \longleftrightarrow$ *limsup* $X =$ *liminf* X
⟨*proof*⟩

lemma *limsup-le-liminf-real*:
fixes X :: *nat* $\Rightarrow$ *real* **and** L :: *real*
assumes *1*: *limsup* $X \leq L$ **and** *2*: $L \leq$ *liminf* X
shows $X \longrightarrow L$
⟨*proof*⟩

lemma *liminf-PInfty*:

fixes $X :: nat \Rightarrow ereal$
shows $X \longrightarrow \infty \longleftrightarrow liminf\ X = \infty$
⟨*proof*⟩

lemma *limsup-MInfty*:
fixes $X :: nat \Rightarrow ereal$
shows $X \longrightarrow -\infty \longleftrightarrow limsup\ X = -\infty$
⟨*proof*⟩

lemma *ereal-lim-mono*:
fixes $X\ Y :: nat \Rightarrow 'a$::*linorder-topology*
assumes $\bigwedge n.\ N \leq n \Longrightarrow X\ n \leq Y\ n$
and $X \longrightarrow x$
and $Y \longrightarrow y$
shows $x \leq y$
⟨*proof*⟩

lemma *incseq-le-ereal*:
fixes $X :: nat \Rightarrow 'a$::*linorder-topology*
assumes *inc*: *incseq* X
and *lim*: $X \longrightarrow L$
shows $X\ N \leq L$
⟨*proof*⟩

lemma *decseq-ge-ereal*:
assumes *dec*: *decseq* X
and *lim*: $X \longrightarrow (L::'a$::*linorder-topology*$)$
shows $X\ N \geq L$
⟨*proof*⟩

lemma *bounded-abs*:
fixes $a :: real$
assumes $a \leq x$
and $x \leq b$
shows $|x| \leq max\ |a|\ |b|$
⟨*proof*⟩

lemma *ereal-Sup-lim*:
fixes $a :: 'a$::{*complete-linorder*,*linorder-topology*}
assumes $\bigwedge n.\ b\ n \in s$
and $b \longrightarrow a$
shows $a \leq Sup\ s$
⟨*proof*⟩

lemma *ereal-Inf-lim*:
fixes $a :: 'a$::{*complete-linorder*,*linorder-topology*}
assumes $\bigwedge n.\ b\ n \in s$
and $b \longrightarrow a$
shows $Inf\ s \leq a$

⟨*proof*⟩

lemma *SUP-Lim-ereal*:
fixes X :: *nat* ⇒ *'a*::{*complete-linorder*,*linorder-topology*}
assumes *inc*: *incseq X*
and *l*: $X \longrightarrow l$
shows (*SUP n. X n*) = *l*
⟨*proof*⟩

lemma *INF-Lim-ereal*:
fixes X :: *nat* ⇒ *'a*::{*complete-linorder*,*linorder-topology*}
assumes *dec*: *decseq X*
and *l*: $X \longrightarrow l$
shows (*INF n. X n*) = *l*
⟨*proof*⟩

lemma *SUP-eq-LIMSEQ*:
assumes *mono f*
shows (*SUP n. ereal* (*f n*)) = *ereal x* ⟷ *f* ⟶ *x*
⟨*proof*⟩

lemma *liminf-ereal-cminus*:
fixes *f* :: *nat* ⇒ *ereal*
assumes $c \neq -\infty$
shows *liminf* (λ*x*. *c* − *f x*) = *c* − *limsup f*
⟨*proof*⟩

30.4.3 Continuity

lemma *continuous-at-of-ereal*:
|*x0* :: *ereal*| ≠ ∞ ⟹ *continuous* (*at x0*) *real-of-ereal*
⟨*proof*⟩

lemma *nhds-ereal*: *nhds* (*ereal r*) = *filtermap ereal* (*nhds r*)
⟨*proof*⟩

lemma *at-ereal*: *at* (*ereal r*) = *filtermap ereal* (*at r*)
⟨*proof*⟩

lemma *at-left-ereal*: *at-left* (*ereal r*) = *filtermap ereal* (*at-left r*)
⟨*proof*⟩

lemma *at-right-ereal*: *at-right* (*ereal r*) = *filtermap ereal* (*at-right r*)
⟨*proof*⟩

lemma
shows *at-left-PInf*: *at-left* ∞ = *filtermap ereal at-top*
and *at-right-MInf*: *at-right* (−∞) = *filtermap ereal at-bot*
⟨*proof*⟩

lemma *ereal-tendsto-simps1*:
$((f \circ \textit{real-of-ereal}) \longrightarrow y)\ (\textit{at-left}\ (\textit{ereal}\ x)) \longleftrightarrow (f \longrightarrow y)\ (\textit{at-left}\ x)$
$((f \circ \textit{real-of-ereal}) \longrightarrow y)\ (\textit{at-right}\ (\textit{ereal}\ x)) \longleftrightarrow (f \longrightarrow y)\ (\textit{at-right}\ x)$
$((f \circ \textit{real-of-ereal}) \longrightarrow y)\ (\textit{at-left}\ (\infty::\textit{ereal})) \longleftrightarrow (f \longrightarrow y)\ \textit{at-top}$
$((f \circ \textit{real-of-ereal}) \longrightarrow y)\ (\textit{at-right}\ (-\infty::\textit{ereal})) \longleftrightarrow (f \longrightarrow y)\ \textit{at-bot}$
⟨*proof*⟩

lemma *ereal-tendsto-simps2*:
$((\textit{ereal} \circ f) \longrightarrow \textit{ereal}\ a)\ F \longleftrightarrow (f \longrightarrow a)\ F$
$((\textit{ereal} \circ f) \longrightarrow \infty)\ F \longleftrightarrow (\textit{LIM}\ x\ F.\ f\ x :> \textit{at-top})$
$((\textit{ereal} \circ f) \longrightarrow -\infty)\ F \longleftrightarrow (\textit{LIM}\ x\ F.\ f\ x :> \textit{at-bot})$
⟨*proof*⟩

lemma *inverse-infty-ereal-tendsto-0*: $\textit{inverse} \longrightarrow (0::\textit{ereal})$
⟨*proof*⟩

lemma *inverse-ereal-tendsto-pos*:
fixes $x :: \textit{ereal}$ **assumes** $0 < x$
shows $\textit{inverse} \longrightarrow \textit{inverse}\ x$
⟨*proof*⟩

lemma *inverse-ereal-tendsto-at-right-0*: $(\textit{inverse} \longrightarrow \infty)\ (\textit{at-right}\ (0::\textit{ereal}))$
⟨*proof*⟩

lemmas *ereal-tendsto-simps = ereal-tendsto-simps1 ereal-tendsto-simps2*

lemma *continuous-at-iff-ereal*:
fixes $f :: 'a::\textit{t2-space} \Rightarrow \textit{real}$
shows $\textit{continuous}\ (\textit{at}\ x0\ \textit{within}\ s)\ f \longleftrightarrow \textit{continuous}\ (\textit{at}\ x0\ \textit{within}\ s)\ (\textit{ereal} \circ f)$
⟨*proof*⟩

lemma *continuous-on-iff-ereal*:
fixes $f :: 'a::\textit{t2-space} => \textit{real}$
assumes *open A*
shows $\textit{continuous-on}\ A\ f \longleftrightarrow \textit{continuous-on}\ A\ (\textit{ereal} \circ f)$
⟨*proof*⟩

lemma *continuous-on-real*: $\textit{continuous-on}\ (\textit{UNIV} - \{\infty, -\infty::\textit{ereal}\})\ \textit{real-of-ereal}$
⟨*proof*⟩

lemma *continuous-on-iff-real*:
fixes $f :: 'a::\textit{t2-space} \Rightarrow \textit{ereal}$
assumes $*$: $\bigwedge x.\ x \in A \Longrightarrow |f\ x| \neq \infty$
shows $\textit{continuous-on}\ A\ f \longleftrightarrow \textit{continuous-on}\ A\ (\textit{real-of-ereal} \circ f)$
⟨*proof*⟩

lemma *continuous-uminus-ereal* [*continuous-intros*]: $\textit{continuous-on}\ (A :: \textit{ereal set})$ *uminus*

⟨*proof*⟩

lemma *ereal-uminus-atMost* [*simp*]: *uminus* ' {..(*a*::*ereal*)} = {−*a*..}
⟨*proof*⟩

lemma *continuous-on-inverse-ereal* [*continuous-intros*]:
continuous-on {*0*::*ereal* ..} *inverse*
⟨*proof*⟩

lemma *continuous-inverse-ereal-nonpos*: *continuous-on* ({..<*0*} :: *ereal set*) *inverse*
⟨*proof*⟩

lemma *tendsto-inverse-ereal*:
assumes (*f* ⟶ (*c* :: *ereal*)) *F*
assumes *eventually* (λ*x*. *f x* ≥ *0*) *F*
shows ((λ*x*. *inverse* (*f x*)) ⟶ *inverse c*) *F*
⟨*proof*⟩

30.4.4 liminf and limsup

lemma *Limsup-ereal-mult-right*:
assumes *F* ≠ *bot* (*c*::*real*) ≥ *0*
shows *Limsup F* (λ*n*. *f n* ∗ *ereal c*) = *Limsup F f* ∗ *ereal c*
⟨*proof*⟩

lemma *Liminf-ereal-mult-right*:
assumes *F* ≠ *bot* (*c*::*real*) ≥ *0*
shows *Liminf F* (λ*n*. *f n* ∗ *ereal c*) = *Liminf F f* ∗ *ereal c*
⟨*proof*⟩

lemma *Limsup-ereal-mult-left*:
assumes *F* ≠ *bot* (*c*::*real*) ≥ *0*
shows *Limsup F* (λ*n*. *ereal c* ∗ *f n*) = *ereal c* ∗ *Limsup F f*
⟨*proof*⟩

lemma *limsup-ereal-mult-right*:
(*c*::*real*) ≥ *0* ⟹ *limsup* (λ*n*. *f n* ∗ *ereal c*) = *limsup f* ∗ *ereal c*
⟨*proof*⟩

lemma *limsup-ereal-mult-left*:
(*c*::*real*) ≥ *0* ⟹ *limsup* (λ*n*. *ereal c* ∗ *f n*) = *ereal c* ∗ *limsup f*
⟨*proof*⟩

lemma *Limsup-add-ereal-right*:
F ≠ *bot* ⟹ *abs c* ≠ ∞ ⟹ *Limsup F* (λ*n*. *g n* + (*c* :: *ereal*)) = *Limsup F g* + *c*
⟨*proof*⟩

lemma *Limsup-add-ereal-left*:

$F \neq bot \Longrightarrow abs\ c \neq \infty \Longrightarrow Limsup\ F\ (\lambda n.\ (c :: ereal) + g\ n) = c + Limsup\ F\ g$
⟨*proof*⟩

lemma *Liminf-add-ereal-right*:
$F \neq bot \Longrightarrow abs\ c \neq \infty \Longrightarrow Liminf\ F\ (\lambda n.\ g\ n + (c :: ereal)) = Liminf\ F\ g + c$
⟨*proof*⟩

lemma *Liminf-add-ereal-left*:
$F \neq bot \Longrightarrow abs\ c \neq \infty \Longrightarrow Liminf\ F\ (\lambda n.\ (c :: ereal) + g\ n) = c + Liminf\ F\ g$
⟨*proof*⟩

lemma
assumes $F \neq bot$
assumes *nonneg*: $eventually\ (\lambda x.\ f\ x \geq (0::ereal))\ F$
shows *Liminf-inverse-ereal*: $Liminf\ F\ (\lambda x.\ inverse\ (f\ x)) = inverse\ (Limsup\ F\ f)$
and *Limsup-inverse-ereal*: $Limsup\ F\ (\lambda x.\ inverse\ (f\ x)) = inverse\ (Liminf\ F\ f)$
⟨*proof*⟩

lemma *ereal-diff-le-mono-left*: $[\![\ x \leq z;\ 0 \leq y\ ]\!] \Longrightarrow x - y \leq (z :: ereal)$
⟨*proof*⟩

lemma *neg-0-less-iff-less-erea* [*simp*]: $0 < - a \longleftrightarrow (a :: ereal) < 0$
⟨*proof*⟩

lemma *not-infty-ereal*: $|x| \neq \infty \longleftrightarrow (\exists x'.\ x = ereal\ x')$
⟨*proof*⟩

lemma *neq-PInf-trans*: **fixes** $x\ y :: ereal$ **shows** $[\![\ y \neq \infty;\ x \leq y\ ]\!] \Longrightarrow x \neq \infty$
⟨*proof*⟩

lemma *mult-2-ereal*: $ereal\ 2 * x = x + x$
⟨*proof*⟩

lemma *ereal-diff-le-self*: $0 \leq y \Longrightarrow x - y \leq (x :: ereal)$
⟨*proof*⟩

lemma *ereal-le-add-self*: $0 \leq y \Longrightarrow x \leq x + (y :: ereal)$
⟨*proof*⟩

lemma *ereal-le-add-self2*: $0 \leq y \Longrightarrow x \leq y + (x :: ereal)$
⟨*proof*⟩

lemma *ereal-le-add-mono1*: $[\![\ x \leq y;\ 0 \leq (z :: ereal)\ ]\!] \Longrightarrow x \leq y + z$
⟨*proof*⟩

lemma *ereal-le-add-mono2*: $[\![\ x \leq z;\ 0 \leq (y :: ereal)\ ]\!] \Longrightarrow x \leq y + z$

⟨*proof*⟩

lemma *ereal-diff-nonpos*:
fixes *a b* :: *ereal* **shows** ⟦ $a \leq b$; $a = \infty \Longrightarrow b \neq \infty$; $a = -\infty \Longrightarrow b \neq -\infty$ ⟧
$\Longrightarrow a - b \leq 0$
⟨*proof*⟩

lemma *minus-ereal-0* [*simp*]: $x - ereal\ 0 = x$
⟨*proof*⟩

lemma *ereal-diff-eq-0-iff*: **fixes** *a b* :: *ereal*
shows $(|a| = \infty \Longrightarrow |b| \neq \infty) \Longrightarrow a - b = 0 \longleftrightarrow a = b$
⟨*proof*⟩

lemma *SUP-ereal-eq-0-iff-nonneg*:
fixes *f* :: - ⇒ *ereal* **and** *A*
assumes *nonneg*: $\forall x{\in}A.\ f\ x \geq 0$
and *A*:$A \neq \{\}$
shows $(SUP\ x{:}A.\ f\ x) = 0 \longleftrightarrow (\forall x{\in}A.\ f\ x = 0)$ (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *ereal-divide-le-posI*:
fixes *x y z* :: *ereal*
shows $x > 0 \Longrightarrow z \neq -\infty \Longrightarrow z \leq x * y \Longrightarrow z\ /\ x \leq y$
⟨*proof*⟩

lemma *add-diff-eq-ereal*: **fixes** *x y z* :: *ereal*
shows $x + (y - z) = x + y - z$
⟨*proof*⟩

lemma *ereal-diff-gr0*:
fixes *a b* :: *ereal* **shows** $a < b \Longrightarrow 0 < b - a$
⟨*proof*⟩

lemma *ereal-minus-minus*: **fixes** *x y z* :: *ereal* **shows**
$(|y| = \infty \Longrightarrow |z| \neq \infty) \Longrightarrow x - (y - z) = x + z - y$
⟨*proof*⟩

lemma *diff-add-eq-ereal*: **fixes** *a b c* :: *ereal* **shows** $a - b + c = a + c - b$
⟨*proof*⟩

lemma *diff-diff-commute-ereal*: **fixes** *x y z* :: *ereal* **shows** $x - y - z = x - z - y$
⟨*proof*⟩

lemma *ereal-diff-eq-MInfty-iff*: **fixes** *x y* :: *ereal* **shows** $x - y = -\infty \longleftrightarrow x = -\infty \wedge y \neq -\infty \vee y = \infty \wedge |x| \neq \infty$
⟨*proof*⟩

lemma *ereal-diff-add-inverse*: **fixes** $x\ y$ :: *ereal* **shows** $|x| \neq \infty \Longrightarrow x + y - x = y$
⟨*proof*⟩

lemma *tendsto-diff-ereal*:
fixes $x\ y$ :: *ereal*
assumes x: $|x| \neq \infty$ **and** y: $|y| \neq \infty$
assumes f: $(f \longrightarrow x)\ F$ **and** g: $(g \longrightarrow y)\ F$
shows $((\lambda x.\ f\ x - g\ x) \longrightarrow x - y)\ F$
⟨*proof*⟩

30.4.5 Tests for code generator

value $-\infty$ :: *ereal*
value $|-\infty|$ :: *ereal*
value *4 + 5 / 4 − ereal 2* :: *ereal*
value *ereal 3* $< \infty$
value *real-of-ereal* $(\infty$::*ereal*$) = 0$

end

31 Indicator Function

theory *Indicator-Function*
imports *Complex-Main Disjoint-Sets*
begin

definition *indicator S x* = (*if x* ∈ *S then 1 else 0*)

lemma *indicator-simps*[*simp*]:
$x \in S \Longrightarrow$ *indicator S x* = *1*
$x \notin S \Longrightarrow$ *indicator S x* = *0*
⟨*proof*⟩

lemma *indicator-pos-le*[*intro*, *simp*]: (*0*::*'a*::*linordered-semidom*) ≤ *indicator S x*
and *indicator-le-1*[*intro*, *simp*]: *indicator S x* ≤ (*1*::*'a*::*linordered-semidom*)
⟨*proof*⟩

lemma *indicator-abs-le-1*: |*indicator S x*| ≤ (*1*::*'a*::*linordered-idom*)
⟨*proof*⟩

lemma *indicator-eq-0-iff*: *indicator A x* = (*0*::*'a*::*zero-neq-one*) ⟷ *x* ∉ *A*
⟨*proof*⟩

lemma *indicator-eq-1-iff*: *indicator A x* = (*1*::*'a*::*zero-neq-one*) ⟷ *x* ∈ *A*
⟨*proof*⟩

lemma *indicator-UNIV* [*simp*]: *indicator UNIV* = (λ*x*. *1*)
⟨*proof*⟩

lemma *indicator-leI*:
$(x \in A \Longrightarrow y \in B) \Longrightarrow (\mathit{indicator}\ A\ x :: 'a{::}\mathit{linordered\text{-}nonzero\text{-}semiring}) \leq \mathit{indicator}\ B\ y$
⟨*proof*⟩

lemma *split-indicator*: $P\ (\mathit{indicator}\ S\ x) \longleftrightarrow ((x \in S \longrightarrow P\ 1) \wedge (x \notin S \longrightarrow P\ 0))$
⟨*proof*⟩

lemma *split-indicator-asm*: $P\ (\mathit{indicator}\ S\ x) \longleftrightarrow (\neg\ (x \in S \wedge \neg\ P\ 1 \vee x \notin S \wedge \neg\ P\ 0))$
⟨*proof*⟩

lemma *indicator-inter-arith*: $\mathit{indicator}\ (A \cap B)\ x = \mathit{indicator}\ A\ x * (\mathit{indicator}\ B\ x{::}'a{::}\mathit{semiring\text{-}1})$
⟨*proof*⟩

lemma *indicator-union-arith*:
$\mathit{indicator}\ (A \cup B)\ x = \mathit{indicator}\ A\ x + \mathit{indicator}\ B\ x - \mathit{indicator}\ A\ x * (\mathit{indicator}\ B\ x :: 'a{::}\mathit{ring\text{-}1})$
⟨*proof*⟩

lemma *indicator-inter-min*: $\mathit{indicator}\ (A \cap B)\ x = \mathit{min}\ (\mathit{indicator}\ A\ x)\ (\mathit{indicator}\ B\ x{::}'a{::}\mathit{linordered\text{-}semidom})$
and *indicator-union-max*: $\mathit{indicator}\ (A \cup B)\ x = \mathit{max}\ (\mathit{indicator}\ A\ x)\ (\mathit{indicator}\ B\ x{::}'a{::}\mathit{linordered\text{-}semidom})$
⟨*proof*⟩

lemma *indicator-disj-union*:
$A \cap B = \{\} \Longrightarrow \mathit{indicator}\ (A \cup B)\ x = (\mathit{indicator}\ A\ x + \mathit{indicator}\ B\ x :: 'a{::}\mathit{linordered\text{-}semidom})$
⟨*proof*⟩

lemma *indicator-compl*: $\mathit{indicator}\ (-\ A)\ x = 1 - (\mathit{indicator}\ A\ x :: 'a{::}\mathit{ring\text{-}1})$
and *indicator-diff*: $\mathit{indicator}\ (A - B)\ x = \mathit{indicator}\ A\ x * (1 - \mathit{indicator}\ B\ x ::'a{::}\mathit{ring\text{-}1})$
⟨*proof*⟩

lemma *indicator-times*:
$\mathit{indicator}\ (A \times B)\ x = \mathit{indicator}\ A\ (\mathit{fst}\ x) * (\mathit{indicator}\ B\ (\mathit{snd}\ x) :: 'a{::}\mathit{semiring\text{-}1})$
⟨*proof*⟩

lemma *indicator-sum*:
$\mathit{indicator}\ (A <+> B)\ x = (\mathit{case}\ x\ \mathit{of}\ \mathit{Inl}\ x \Rightarrow \mathit{indicator}\ A\ x \mid \mathit{Inr}\ x \Rightarrow \mathit{indicator}\ B\ x)$
⟨*proof*⟩

lemma *indicator-image*: $\mathit{inj}\ f \Longrightarrow \mathit{indicator}\ (f\ `\ X)\ (f\ x) = (\mathit{indicator}\ X\ x{::}\text{-}{::}\mathit{zero\text{-}neq\text{-}one})$

⟨*proof*⟩

lemma *indicator-vimage*: *indicator* (*f* −‘ *A*) *x* = *indicator A* (*f x*)
⟨*proof*⟩

lemma
fixes *f* :: *'a* ⇒ *'b*::*semiring-1*
assumes *finite A*
shows *sum-mult-indicator*[*simp*]: (∑ *x* ∈ *A*. *f x* ∗ *indicator B x*) = (∑ *x* ∈ *A* ∩ *B*. *f x*)
and *sum-indicator-mult*[*simp*]: (∑ *x* ∈ *A*. *indicator B x* ∗ *f x*) = (∑ *x* ∈ *A* ∩ *B*. *f x*)
⟨*proof*⟩

lemma *sum-indicator-eq-card*:
assumes *finite A*
shows (∑ *x* ∈ *A*. *indicator B x*) = *card* (*A Int B*)
⟨*proof*⟩

lemma *sum-indicator-scaleR*[*simp*]:
finite A ⟹
(∑ *x* ∈ *A*. *indicator* (*B x*) (*g x*) ∗$_R$ *f x*) = (∑ *x* ∈ {*x*∈*A*. *g x* ∈ *B x*}. *f x* :: *'a*::*real-vector*)
⟨*proof*⟩

lemma *LIMSEQ-indicator-incseq*:
assumes *incseq A*
shows (λ*i*. *indicator* (*A i*) *x* :: *'a*::{*topological-space*,*one*,*zero*}) ⟶ *indicator* (⋃ *i*. *A i*) *x*
⟨*proof*⟩

lemma *LIMSEQ-indicator-UN*:
(λ*k*. *indicator* (⋃ *i*<*k*. *A i*) *x* :: *'a*::{*topological-space*,*one*,*zero*}) ⟶ *indicator* (⋃ *i*. *A i*) *x*
⟨*proof*⟩

lemma *LIMSEQ-indicator-decseq*:
assumes *decseq A*
shows (λ*i*. *indicator* (*A i*) *x* :: *'a*::{*topological-space*,*one*,*zero*}) ⟶ *indicator* (⋂ *i*. *A i*) *x*
⟨*proof*⟩

lemma *LIMSEQ-indicator-INT*:
(λ*k*. *indicator* (⋂ *i*<*k*. *A i*) *x* :: *'a*::{*topological-space*,*one*,*zero*}) ⟶ *indicator* (⋂ *i*. *A i*) *x*
⟨*proof*⟩

lemma *indicator-add*:
A ∩ *B* = {} ⟹ (*indicator A x*::-::*monoid-add*) + *indicator B x* = *indicator* (*A*

$\cup$ B) x
⟨*proof*⟩

lemma *of-real-indicator*: *of-real* (*indicator* A x) = *indicator* A x
⟨*proof*⟩

lemma *real-of-nat-indicator*: *real* (*indicator* A x :: *nat*) = *indicator* A x
⟨*proof*⟩

lemma *abs-indicator*: |*indicator* A x :: 'a::*linordered-idom*| = *indicator* A x
⟨*proof*⟩

lemma *mult-indicator-subset*:
$A \subseteq B \Longrightarrow$ *indicator* A x * *indicator* B x = (*indicator* A x :: 'a::*comm-semiring-1*)
⟨*proof*⟩

lemma *indicator-sums*:
assumes $\bigwedge i\ j.\ i \neq j \Longrightarrow A\ i \cap A\ j = \{\}$
shows ($\lambda i.$ *indicator* (A i) x::*real*) *sums* *indicator* ($\bigcup i.\ A\ i$) x
⟨*proof*⟩

The indicator function of the union of a disjoint family of sets is the sum over all the individual indicators.

lemma *indicator-UN-disjoint*:
finite $A \Longrightarrow$ *disjoint-family-on* f $A \Longrightarrow$ *indicator* (*UNION* A f) x = ($\sum y \in A.$ *indicator* (f y) x)
⟨*proof*⟩

end

31.1 The type of non-negative extended real numbers

theory *Extended-Nonnegative-Real*
imports *Extended-Real Indicator-Function*
begin

lemma *ereal-ineq-diff-add*:
assumes $b \neq (-\infty$::*ereal*) $a \geq b$
shows $a = b + (a - b)$
⟨*proof*⟩

lemma *Limsup-const-add*:
fixes c :: 'a::{*complete-linorder*, *linorder-topology*, *topological-monoid-add*, *ordered-ab-semigroup-add*}
shows $F \neq bot \Longrightarrow$ *Limsup* F ($\lambda x.\ c + f\ x$) = c + *Limsup* F f
⟨*proof*⟩

lemma *Liminf-const-add*:
fixes c :: 'a::{*complete-linorder*, *linorder-topology*, *topological-monoid-add*, *ordered-ab-semigroup-add*}
shows $F \neq bot \Longrightarrow$ *Liminf* F ($\lambda x.\ c + f\ x$) = c + *Liminf* F f

⟨*proof*⟩

lemma *Liminf-add-const*:
fixes $c :: 'a::\{complete\text{-}linorder, linorder\text{-}topology, topological\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\}$
shows $F \neq bot \Longrightarrow Liminf\ F\ (\lambda x.\ f\ x + c) = Liminf\ F\ f + c$
⟨*proof*⟩

lemma *sums-offset*:
fixes $f\ g :: nat \Rightarrow 'a :: \{t2\text{-}space, topological\text{-}comm\text{-}monoid\text{-}add\}$
assumes $(\lambda n.\ f\ (n + i))\ sums\ l$ **shows** $f\ sums\ (l + (\sum j{<}i.\ f\ j))$
⟨*proof*⟩

lemma *suminf-offset*:
fixes $f\ g :: nat \Rightarrow 'a :: \{t2\text{-}space, topological\text{-}comm\text{-}monoid\text{-}add\}$
shows $summable\ (\lambda j.\ f\ (j + i)) \Longrightarrow suminf\ f = (\sum j.\ f\ (j + i)) + (\sum j{<}i.\ f\ j)$
⟨*proof*⟩

lemma *eventually-at-left-1*: $(\bigwedge z{::}real.\ 0 < z \Longrightarrow z < 1 \Longrightarrow P\ z) \Longrightarrow eventually\ P\ (at\text{-}left\ 1)$
⟨*proof*⟩

lemma *mult-eq-1*:
fixes $a\ b :: 'a :: \{ordered\text{-}semiring, comm\text{-}monoid\text{-}mult\}$
shows $0 \leq a \Longrightarrow a \leq 1 \Longrightarrow b \leq 1 \Longrightarrow a * b = 1 \longleftrightarrow (a = 1 \wedge b = 1)$
⟨*proof*⟩

lemma *ereal-add-diff-cancel*:
fixes $a\ b :: ereal$
shows $|b| \neq \infty \Longrightarrow (a + b) - b = a$
⟨*proof*⟩

lemma *add-top*:
fixes $x :: 'a::\{order\text{-}top, ordered\text{-}comm\text{-}monoid\text{-}add\}$
shows $0 \leq x \Longrightarrow x + top = top$
⟨*proof*⟩

lemma *top-add*:
fixes $x :: 'a::\{order\text{-}top, ordered\text{-}comm\text{-}monoid\text{-}add\}$
shows $0 \leq x \Longrightarrow top + x = top$
⟨*proof*⟩

lemma *le-lfp*: $mono\ f \Longrightarrow x \leq lfp\ f \Longrightarrow f\ x \leq lfp\ f$
⟨*proof*⟩

lemma *lfp-transfer*:
assumes α: *sup-continuous* α **and** f: *sup-continuous* f **and** *mg*: *mono* g
assumes *bot*: $\alpha\ bot \leq lfp\ g$ **and** *eq*: $\bigwedge x.\ x \leq lfp\ f \Longrightarrow \alpha\ (f\ x) = g\ (\alpha\ x)$
shows $\alpha\ (lfp\ f) = lfp\ g$
⟨*proof*⟩

lemma *sup-continuous-applyD*: *sup-continuous f* $\Longrightarrow$ *sup-continuous* ($\lambda x.\ f\ x\ h$)
⟨*proof*⟩

lemma *sup-continuous-SUP*[*order-continuous-intros*]:
fixes *M* :: - $\Rightarrow$ - $\Rightarrow$ 'a::*complete-lattice*
assumes *M*: $\bigwedge i.\ i \in I \Longrightarrow$ *sup-continuous* (*M i*)
shows *sup-continuous* (*SUP i*:*I*. *M i*)
⟨*proof*⟩

lemma *sup-continuous-apply-SUP*[*order-continuous-intros*]:
fixes *M* :: - $\Rightarrow$ - $\Rightarrow$ 'a::*complete-lattice*
shows ($\bigwedge i.\ i \in I \Longrightarrow$ *sup-continuous* (*M i*)) $\Longrightarrow$ *sup-continuous* ($\lambda x.$ *SUP i*:*I*. *M i x*)
⟨*proof*⟩

lemma *sup-continuous-lfp'*[*order-continuous-intros*]:
assumes *1*: *sup-continuous f*
assumes *2*: $\bigwedge g.$ *sup-continuous g* $\Longrightarrow$ *sup-continuous* (*f g*)
shows *sup-continuous* (*lfp f*)
⟨*proof*⟩

lemma *sup-continuous-lfp''*[*order-continuous-intros*]:
assumes *1*: $\bigwedge s.$ *sup-continuous* (*f s*)
assumes *2*: $\bigwedge g.$ *sup-continuous g* $\Longrightarrow$ *sup-continuous* ($\lambda s.$ *f s* (*g s*))
shows *sup-continuous* ($\lambda x.$ *lfp* (*f x*))
⟨*proof*⟩

lemma *mono-INF-fun*:
($\bigwedge x\ y.$ *mono* (*F x y*)) $\Longrightarrow$ *mono* ($\lambda z\ x.$ *INF y* : *X x*. *F x y z* :: 'a :: *complete-lattice*)
⟨*proof*⟩

lemma *continuous-on-max*:
fixes *f g* :: 'a::*topological-space* $\Rightarrow$ 'b::*linorder-topology*
shows *continuous-on A f* $\Longrightarrow$ *continuous-on A g* $\Longrightarrow$ *continuous-on A* ($\lambda x.$ *max* (*f x*) (*g x*))
⟨*proof*⟩

lemma *continuous-on-cmult-ereal*:
|*c*::*ereal*| $\neq \infty \Longrightarrow$ *continuous-on A f* $\Longrightarrow$ *continuous-on A* ($\lambda x.$ *c* $*$ *f x*)
⟨*proof*⟩

context *linordered-nonzero-semiring*
begin

lemma *of-nat-nonneg* [*simp*]: *0* $\leq$ *of-nat n*
⟨*proof*⟩

lemma *of-nat-mono*[*simp*]: $i \leq j \Longrightarrow$ *of-nat* $i \leq$ *of-nat* j
⟨*proof*⟩

end

lemma *real-of-nat-Sup*:
assumes $A \neq \{\}$ *bdd-above* A
shows *of-nat* (*Sup* A) = (*SUP* a:A. *of-nat* a :: *real*)
⟨*proof*⟩

lemma *of-nat-less*[*simp*]:
$m < n \Longrightarrow$ *of-nat* $m <$ (*of-nat* n::$'a$::{*linordered-nonzero-semiring*, *semiring-char-0*})
⟨*proof*⟩

lemma *of-nat-le-iff*[*simp*]:
of-nat $m \leq$ (*of-nat* n::$'a$::{*linordered-nonzero-semiring*, *semiring-char-0*}) ⟷ $m \leq n$
⟨*proof*⟩

lemma (**in** *complete-lattice*) *SUP-sup-const1*:
$I \neq \{\} \Longrightarrow$ (*SUP* i:I. *sup* c (f i)) = *sup* c (*SUP* i:I. f i)
⟨*proof*⟩

lemma (**in** *complete-lattice*) *SUP-sup-const2*:
$I \neq \{\} \Longrightarrow$ (*SUP* i:I. *sup* (f i) c) = *sup* (*SUP* i:I. f i) c
⟨*proof*⟩

lemma *one-less-of-natD*:
(1::$'a$::*linordered-semidom*) < *of-nat* $n \Longrightarrow$ 1 < n
⟨*proof*⟩

lemma *sum-le-suminf*:
fixes f :: *nat* ⇒ $'a$::{*ordered-comm-monoid-add*, *linorder-topology*}
shows *summable* $f \Longrightarrow$ *finite* $I \Longrightarrow \forall\, m{\in}{-}\ I.\ 0 \leq f\ m \Longrightarrow$ *sum* f $I \leq$ *suminf* f
⟨*proof*⟩

lemma *suminf-eq-SUP-real*:
assumes X: *summable* X $\bigwedge i.\ 0 \leq X\ i$ **shows** *suminf* X = (*SUP* i. $\sum n{<}i.\ X$ n::*real*)
⟨*proof*⟩

31.2 Defining the extended non-negative reals

Basic definitions and type class setup

typedef *ennreal* = $\{x$ :: *ereal*. $0 \leq x\}$
morphisms *enn2ereal e2ennreal'*
⟨*proof*⟩

definition *e2ennreal x = e2ennreal′ (max 0 x)*

lemma *enn2ereal-range*: *e2ennreal ‘ {0..} = UNIV*
⟨*proof*⟩

lemma *type-definition-ennreal′*: *type-definition enn2ereal e2ennreal {x. 0 ≤ x}*
⟨*proof*⟩

setup-lifting *type-definition-ennreal′*

declare [[*coercion e2ennreal*]]

instantiation *ennreal* :: *complete-linorder*
begin

lift-definition *top-ennreal* :: *ennreal* **is** *top* ⟨*proof*⟩
lift-definition *bot-ennreal* :: *ennreal* **is** *0* ⟨*proof*⟩
lift-definition *sup-ennreal* :: *ennreal ⇒ ennreal ⇒ ennreal* **is** *sup* ⟨*proof*⟩
lift-definition *inf-ennreal* :: *ennreal ⇒ ennreal ⇒ ennreal* **is** *inf* ⟨*proof*⟩

lift-definition *Inf-ennreal* :: *ennreal set ⇒ ennreal* **is** *Inf*
⟨*proof*⟩

lift-definition *Sup-ennreal* :: *ennreal set ⇒ ennreal* **is** *sup 0 ∘ Sup*
⟨*proof*⟩

lift-definition *less-eq-ennreal* :: *ennreal ⇒ ennreal ⇒ bool* **is** *op ≤* ⟨*proof*⟩
lift-definition *less-ennreal* :: *ennreal ⇒ ennreal ⇒ bool* **is** *op <* ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *pcr-ennreal-enn2ereal*[*simp*]: *pcr-ennreal (enn2ereal x) x*
⟨*proof*⟩

lemma *rel-fun-eq-pcr-ennreal*: *rel-fun op = pcr-ennreal f g ⟷ f = enn2ereal ∘ g*
⟨*proof*⟩

instantiation *ennreal* :: *infinity*
begin

definition *infinity-ennreal* :: *ennreal*
where
[*simp*]: *∞ = (top*::*ennreal)*

instance ⟨*proof*⟩

end

instantiation *ennreal* :: {*semiring-1-no-zero-divisors*, *comm-semiring-1*}
begin

lift-definition *one-ennreal* :: *ennreal* **is** *1* ⟨*proof*⟩
lift-definition *zero-ennreal* :: *ennreal* **is** *0* ⟨*proof*⟩
lift-definition *plus-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** *op +* ⟨*proof*⟩
lift-definition *times-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** *op ∗* ⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *ennreal* :: *minus*
begin

lift-definition *minus-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** *λa b. max 0* (*a − b*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *ennreal* :: *numeral* ⟨*proof*⟩

instantiation *ennreal* :: *inverse*
begin

lift-definition *inverse-ennreal* :: *ennreal* ⇒ *ennreal* **is** *inverse*
⟨*proof*⟩

definition *divide-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal*
where *x div y = x ∗ inverse* (*y* :: *ennreal*)

instance ⟨*proof*⟩

end

lemma *ennreal-zero-less-one*: *0 <* (*1*::*ennreal*) — TODO: remove
⟨*proof*⟩

instance *ennreal* :: *dioid*
⟨*proof*⟩

instance *ennreal* :: *ordered-comm-semiring*
⟨*proof*⟩

instance *ennreal* :: *linordered-nonzero-semiring*
⟨*proof*⟩

instance *ennreal* :: *strict-ordered-ab-semigroup-add*
⟨*proof*⟩

declare [[*coercion of-nat* :: *nat* ⇒ *ennreal*]]

lemma *e2ennreal-neg*: $x \leq 0 \Longrightarrow e2ennreal\ x = 0$
⟨*proof*⟩

lemma *e2ennreal-mono*: $x \leq y \Longrightarrow e2ennreal\ x \leq e2ennreal\ y$
⟨*proof*⟩

lemma *enn2ereal-nonneg*[*simp*]: $0 \leq enn2ereal\ x$
⟨*proof*⟩

lemma *ereal-ennreal-cases*:
obtains b **where** $0 \leq a\ a = enn2ereal\ b \mid a < 0$
⟨*proof*⟩

lemma *rel-fun-liminf*[*transfer-rule*]: *rel-fun* (*rel-fun op* = *pcr-ennreal*) *pcr-ennreal liminf liminf*
⟨*proof*⟩

lemma *rel-fun-limsup*[*transfer-rule*]: *rel-fun* (*rel-fun op* = *pcr-ennreal*) *pcr-ennreal limsup limsup*
⟨*proof*⟩

lemma *sum-enn2ereal*[*simp*]: $(\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i) \Longrightarrow (\sum i{\in}I.\ enn2ereal\ (f\ i)) = enn2ereal\ (sum\ f\ I)$
⟨*proof*⟩

lemma *transfer-e2ennreal-sum* [*transfer-rule*]:
rel-fun (*rel-fun op* = *pcr-ennreal*) (*rel-fun op* = *pcr-ennreal*) *sum sum*
⟨*proof*⟩

lemma *enn2ereal-of-nat*[*simp*]: *enn2ereal* (*of-nat n*) = *ereal n*
⟨*proof*⟩

lemma *enn2ereal-numeral*[*simp*]: *enn2ereal* (*numeral a*) = *numeral a*
⟨*proof*⟩

lemma *transfer-numeral*[*transfer-rule*]: *pcr-ennreal* (*numeral a*) (*numeral a*)
⟨*proof*⟩

31.3 Cancellation simprocs

lemma *ennreal-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty::ennreal) \vee b = c$
⟨*proof*⟩

lemma *ennreal-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty::ennreal) \vee b \leq c$
⟨*proof*⟩

lemma *ereal-add-left-cancel-less*:
fixes $a\ b\ c :: ereal$
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a + b < a + c \longleftrightarrow a \neq \infty \wedge b < c$
⟨*proof*⟩

lemma *ennreal-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty::ennreal) \wedge b < c$
⟨*proof*⟩

⟨*ML*⟩

31.4 Order with top

lemma *ennreal-zero-less-top*[*simp*]: $0 < (top::ennreal)$
⟨*proof*⟩

lemma *ennreal-one-less-top*[*simp*]: $1 < (top::ennreal)$
⟨*proof*⟩

lemma *ennreal-zero-neq-top*[*simp*]: $0 \neq (top::ennreal)$
⟨*proof*⟩

lemma *ennreal-top-neq-zero*[*simp*]: $(top::ennreal) \neq 0$
⟨*proof*⟩

lemma *ennreal-top-neq-one*[*simp*]: $top \neq (1::ennreal)$
⟨*proof*⟩

lemma *ennreal-one-neq-top*[*simp*]: $1 \neq (top::ennreal)$
⟨*proof*⟩

lemma *ennreal-add-less-top*[*simp*]:
fixes $a\ b :: ennreal$
shows $a + b < top \longleftrightarrow a < top \wedge b < top$
⟨*proof*⟩

lemma *ennreal-add-eq-top*[*simp*]:
fixes $a\ b :: ennreal$
shows $a + b = top \longleftrightarrow a = top \vee b = top$
⟨*proof*⟩

lemma *ennreal-sum-less-top*[*simp*]:

fixes $f :: 'a \Rightarrow ennreal$
shows $finite\ I \Longrightarrow (\sum i{\in}I.\ f\ i) < top \longleftrightarrow (\forall\, i{\in}I.\ f\ i < top)$
⟨*proof*⟩

lemma *ennreal-sum-eq-top*[*simp*]:
fixes $f :: 'a \Rightarrow ennreal$
shows $finite\ I \Longrightarrow (\sum i{\in}I.\ f\ i) = top \longleftrightarrow (\exists\, i{\in}I.\ f\ i = top)$
⟨*proof*⟩

lemma *ennreal-mult-eq-top-iff*:
fixes $a\ b :: ennreal$
shows $a * b = top \longleftrightarrow (a = top \wedge b \neq 0) \vee (b = top \wedge a \neq 0)$
⟨*proof*⟩

lemma *ennreal-top-eq-mult-iff*:
fixes $a\ b :: ennreal$
shows $top = a * b \longleftrightarrow (a = top \wedge b \neq 0) \vee (b = top \wedge a \neq 0)$
⟨*proof*⟩

lemma *ennreal-mult-less-top*:
fixes $a\ b :: ennreal$
shows $a * b < top \longleftrightarrow (a = 0 \vee b = 0 \vee (a < top \wedge b < top))$
⟨*proof*⟩

lemma *top-power-ennreal*: $top\ \hat{}\ n = (if\ n = 0\ then\ 1\ else\ top :: ennreal)$
⟨*proof*⟩

lemma *ennreal-prod-eq-0*[*simp*]:
fixes $f :: 'a \Rightarrow ennreal$
shows $(prod\ f\ A = 0) = (finite\ A \wedge (\exists\, i{\in}A.\ f\ i = 0))$
⟨*proof*⟩

lemma *ennreal-prod-eq-top*:
fixes $f :: 'a \Rightarrow ennreal$
shows $(\prod i{\in}I.\ f\ i) = top \longleftrightarrow (finite\ I \wedge ((\forall\, i{\in}I.\ f\ i \neq 0) \wedge (\exists\, i{\in}I.\ f\ i = top)))$
⟨*proof*⟩

lemma *ennreal-top-mult*: $top * a = (if\ a = 0\ then\ 0\ else\ top :: ennreal)$
⟨*proof*⟩

lemma *ennreal-mult-top*: $a * top = (if\ a = 0\ then\ 0\ else\ top :: ennreal)$
⟨*proof*⟩

lemma *enn2ereal-eq-top-iff*[*simp*]: $enn2ereal\ x = \infty \longleftrightarrow x = top$
⟨*proof*⟩

lemma *enn2ereal-top*: $enn2ereal\ top = \infty$
⟨*proof*⟩

lemma *e2ennreal-infty*: $e2ennreal\ \infty = top$
⟨*proof*⟩

lemma *ennreal-top-minus*[*simp*]: $top - x = (top::ennreal)$
⟨*proof*⟩

lemma *minus-top-ennreal*: $x - top = (if\ x = top\ then\ top\ else\ 0::\ ennreal)$
⟨*proof*⟩

lemma *bot-ennreal*: $bot = (0::ennreal)$
⟨*proof*⟩

lemma *ennreal-of-nat-neq-top*[*simp*]: $of\text{-}nat\ i \neq (top::ennreal)$
⟨*proof*⟩

lemma *numeral-eq-of-nat*: $(numeral\ a::ennreal) = of\text{-}nat\ (numeral\ a)$
⟨*proof*⟩

lemma *of-nat-less-top*: $of\text{-}nat\ i < (top::ennreal)$
⟨*proof*⟩

lemma *top-neq-numeral*[*simp*]: $top \neq (numeral\ i::ennreal)$
⟨*proof*⟩

lemma *ennreal-numeral-less-top*[*simp*]: $numeral\ i < (top::ennreal)$
⟨*proof*⟩

lemma *ennreal-add-bot*[*simp*]: $bot + x = (x::ennreal)$
⟨*proof*⟩

instance *ennreal* :: *semiring-char-0*
⟨*proof*⟩

31.5 Arithmetic

lemma *ennreal-minus-zero*[*simp*]: $a - (0::ennreal) = a$
⟨*proof*⟩

lemma *ennreal-add-diff-cancel-right*[*simp*]:
fixes $x\ y\ z$ :: *ennreal* **shows** $y \neq top \Longrightarrow (x + y) - y = x$
⟨*proof*⟩

lemma *ennreal-add-diff-cancel-left*[*simp*]:
fixes $x\ y\ z$ :: *ennreal* **shows** $y \neq top \Longrightarrow (y + x) - y = x$
⟨*proof*⟩

lemma
fixes $a\ b$ :: *ennreal*
shows $a - b = 0 \Longrightarrow a \leq b$

⟨*proof*⟩

lemma *ennreal-minus-cancel*:
fixes $a\ b\ c$ :: *ennreal*
shows $c \neq top \Longrightarrow a \leq c \Longrightarrow b \leq c \Longrightarrow c - a = c - b \Longrightarrow a = b$
⟨*proof*⟩

lemma *sup-const-add-ennreal*:
fixes $a\ b\ c$ :: *ennreal*
shows $sup\ (c + a)\ (c + b) = c + sup\ a\ b$
⟨*proof*⟩

lemma *ennreal-diff-add-assoc*:
fixes $a\ b\ c$ :: *ennreal*
shows $a \leq b \Longrightarrow c + b - a = c + (b - a)$
⟨*proof*⟩

lemma *mult-divide-eq-ennreal*:
fixes $a\ b$ :: *ennreal*
shows $b \neq 0 \Longrightarrow b \neq top \Longrightarrow (a * b)\ /\ b = a$
⟨*proof*⟩

lemma *divide-mult-eq*: $a \neq 0 \Longrightarrow a \neq \infty \Longrightarrow x * a\ /\ (b * a) = x\ /\ (b{::}ennreal)$
⟨*proof*⟩

lemma *ennreal-mult-divide-eq*:
fixes $a\ b$ :: *ennreal*
shows $b \neq 0 \Longrightarrow b \neq top \Longrightarrow (a * b)\ /\ b = a$
⟨*proof*⟩

lemma *ennreal-add-diff-cancel*:
fixes $a\ b$ :: *ennreal*
shows $b \neq \infty \Longrightarrow (a + b) - b = a$
⟨*proof*⟩

lemma *ennreal-minus-eq-0*:
$a - b = 0 \Longrightarrow a \leq (b{::}ennreal)$
⟨*proof*⟩

lemma *ennreal-mono-minus-cancel*:
fixes $a\ b\ c$ :: *ennreal*
shows $a - b \leq a - c \Longrightarrow a < top \Longrightarrow b \leq a \Longrightarrow c \leq a \Longrightarrow c \leq b$
⟨*proof*⟩

lemma *ennreal-mono-minus*:
fixes $a\ b\ c$ :: *ennreal*
shows $c \leq b \Longrightarrow a - b \leq a - c$
⟨*proof*⟩

lemma *ennreal-minus-pos-iff*:
fixes *a b* :: *ennreal*
shows $a < top \vee b < top \Longrightarrow 0 < a - b \Longrightarrow b < a$
⟨*proof*⟩

lemma *ennreal-inverse-top*[*simp*]: *inverse top* = (*0*::*ennreal*)
⟨*proof*⟩

lemma *ennreal-inverse-zero*[*simp*]: *inverse 0* = (*top*::*ennreal*)
⟨*proof*⟩

lemma *ennreal-top-divide*: *top* / (*x*::*ennreal*) = (*if x* = *top then 0 else top*)
⟨*proof*⟩

lemma *ennreal-zero-divide*[*simp*]: *0* / (*x*::*ennreal*) = *0*
⟨*proof*⟩

lemma *ennreal-divide-zero*[*simp*]: *x* / (*0*::*ennreal*) = (*if x* = *0 then 0 else top*)
⟨*proof*⟩

lemma *ennreal-divide-top*[*simp*]: *x* / (*top*::*ennreal*) = *0*
⟨*proof*⟩

lemma *ennreal-times-divide*: *a* ∗ (*b* / *c*) = *a* ∗ *b* / (*c*::*ennreal*)
⟨*proof*⟩

lemma *ennreal-zero-less-divide*: *0* < *a* / *b* ⟷ (*0* < *a* ∧ *b* < (*top*::*ennreal*))
⟨*proof*⟩

lemma *divide-right-mono-ennreal*:
fixes *a b c* :: *ennreal*
shows $a \leq b \Longrightarrow a / c \leq b / c$
⟨*proof*⟩

lemma *ennreal-mult-strict-right-mono*: (*a*::*ennreal*) < *c* ⟹ *0* < *b* ⟹ *b* < *top*
⟹ *a* ∗ *b* < *c* ∗ *b*
⟨*proof*⟩

lemma *ennreal-indicator-less*[*simp*]:
indicator A x ≤ (*indicator B x*::*ennreal*) ⟷ (*x* ∈ *A* ⟶ *x* ∈ *B*)
⟨*proof*⟩

lemma *ennreal-inverse-positive*: *0* < *inverse x* ⟷ (*x*::*ennreal*) ≠ *top*
⟨*proof*⟩

lemma *ennreal-inverse-mult'*: ((*0* < *b* ∨ *a* < *top*) ∧ (*0* < *a* ∨ *b* < *top*)) ⟹
inverse (*a* ∗ *b*::*ennreal*) = *inverse a* ∗ *inverse b*
⟨*proof*⟩

lemma *ennreal-inverse-mult*: $a < top \Longrightarrow b < top \Longrightarrow inverse\ (a * b::ennreal) = inverse\ a * inverse\ b$
⟨*proof*⟩

lemma *ennreal-inverse-1*[*simp*]: $inverse\ (1::ennreal) = 1$
⟨*proof*⟩

lemma *ennreal-inverse-eq-0-iff*[*simp*]: $inverse\ (a::ennreal) = 0 \longleftrightarrow a = top$
⟨*proof*⟩

lemma *ennreal-inverse-eq-top-iff*[*simp*]: $inverse\ (a::ennreal) = top \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *ennreal-divide-eq-0-iff*[*simp*]: $(a::ennreal)\ /\ b = 0 \longleftrightarrow (a = 0 \vee b = top)$
⟨*proof*⟩

lemma *ennreal-divide-eq-top-iff*: $(a::ennreal)\ /\ b = top \longleftrightarrow ((a \neq 0 \wedge b = 0) \vee (a = top \wedge b \neq top))$
⟨*proof*⟩

lemma *one-divide-one-divide-ennreal*[*simp*]: $1\ /\ (1\ /\ c) = (c::ennreal)$
including *ennreal.lifting*
⟨*proof*⟩

lemma *ennreal-mult-left-cong*:
$((a::ennreal) \neq 0 \Longrightarrow b = c) \Longrightarrow a * b = a * c$
⟨*proof*⟩

lemma *ennreal-mult-right-cong*:
$((a::ennreal) \neq 0 \Longrightarrow b = c) \Longrightarrow b * a = c * a$
⟨*proof*⟩

lemma *ennreal-zero-less-mult-iff*: $0 < a * b \longleftrightarrow 0 < a \wedge 0 < (b::ennreal)$
⟨*proof*⟩

lemma *less-diff-eq-ennreal*:
fixes $a\ b\ c :: ennreal$
shows $b < top \vee c < top \Longrightarrow a < b - c \longleftrightarrow a + c < b$
⟨*proof*⟩

lemma *diff-add-cancel-ennreal*:
fixes $a\ b :: ennreal$ **shows** $a \leq b \Longrightarrow b - a + a = b$
⟨*proof*⟩

lemma *ennreal-diff-self*[*simp*]: $a \neq top \Longrightarrow a - a = (0::ennreal)$
⟨*proof*⟩

lemma *ennreal-minus-mono*:
fixes $a\ b\ c :: ennreal$

shows $a \leq c \Longrightarrow d \leq b \Longrightarrow a - b \leq c - d$
⟨*proof*⟩

lemma *ennreal-minus-eq-top*[*simp*]: $a - (b::ennreal) = top \longleftrightarrow a = top$
⟨*proof*⟩

lemma *ennreal-divide-self*[*simp*]: $a \neq 0 \Longrightarrow a < top \Longrightarrow a \mathbin{/} a = (1::ennreal)$
⟨*proof*⟩

31.6 Coercion from *real* to *ennreal*

lift-definition *ennreal* :: *real* ⇒ *ennreal* **is** *sup 0* ∘ *ereal*
⟨*proof*⟩

declare [[*coercion ennreal*]]

lemma *ennreal-cong*: $x = y \Longrightarrow ennreal\ x = ennreal\ y$ ⟨*proof*⟩

lemma *ennreal-cases*[*cases type*: *ennreal*]:
fixes x :: *ennreal*
obtains (*real*) r :: *real* **where** $0 \leq r\ x = ennreal\ r \mid (top)\ x = top$
⟨*proof*⟩

lemmas *ennreal2-cases* = *ennreal-cases*[*case-product ennreal-cases*]
lemmas *ennreal3-cases* = *ennreal-cases*[*case-product ennreal2-cases*]

lemma *ennreal-neq-top*[*simp*]: $ennreal\ r \neq top$
⟨*proof*⟩

lemma *top-neq-ennreal*[*simp*]: $top \neq ennreal\ r$
⟨*proof*⟩

lemma *ennreal-less-top*[*simp*]: $ennreal\ x < top$
⟨*proof*⟩

lemma *ennreal-neg*: $x \leq 0 \Longrightarrow ennreal\ x = 0$
⟨*proof*⟩

lemma *ennreal-inj*[*simp*]:
$0 \leq a \Longrightarrow 0 \leq b \Longrightarrow ennreal\ a = ennreal\ b \longleftrightarrow a = b$
⟨*proof*⟩

lemma *ennreal-le-iff*[*simp*]: $0 \leq y \Longrightarrow ennreal\ x \leq ennreal\ y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *le-ennreal-iff*: $0 \leq r \Longrightarrow x \leq ennreal\ r \longleftrightarrow (\exists\, q{\geq}0.\ x = ennreal\ q \wedge q \leq r)$
⟨*proof*⟩

lemma *ennreal-less-iff*: $0 \leq r \Longrightarrow ennreal\ r < ennreal\ q \longleftrightarrow r < q$
⟨*proof*⟩

lemma *ennreal-eq-zero-iff*[*simp*]: $0 \leq x \Longrightarrow ennreal\ x = 0 \longleftrightarrow x = 0$
⟨*proof*⟩

lemma *ennreal-less-zero-iff*[*simp*]: $0 < ennreal\ x \longleftrightarrow 0 < x$
⟨*proof*⟩

lemma *ennreal-lessI*: $0 < q \Longrightarrow r < q \Longrightarrow ennreal\ r < ennreal\ q$
⟨*proof*⟩

lemma *ennreal-leI*: $x \leq y \Longrightarrow ennreal\ x \leq ennreal\ y$
⟨*proof*⟩

lemma *enn2ereal-ennreal*[*simp*]: $0 \leq x \Longrightarrow enn2ereal\ (ennreal\ x) = x$
⟨*proof*⟩

lemma *e2ennreal-enn2ereal*[*simp*]: $e2ennreal\ (enn2ereal\ x) = x$
⟨*proof*⟩

lemma *ennreal-0*[*simp*]: $ennreal\ 0 = 0$
⟨*proof*⟩

lemma *ennreal-1*[*simp*]: $ennreal\ 1 = 1$
⟨*proof*⟩

lemma *ennreal-eq-0-iff*: $ennreal\ x = 0 \longleftrightarrow x \leq 0$
⟨*proof*⟩

lemma *ennreal-le-iff2*: $ennreal\ x \leq ennreal\ y \longleftrightarrow ((0 \leq y \wedge x \leq y) \vee (x \leq 0 \wedge y \leq 0))$
⟨*proof*⟩

lemma *ennreal-eq-1*[*simp*]: $ennreal\ x = 1 \longleftrightarrow x = 1$
⟨*proof*⟩

lemma *ennreal-le-1*[*simp*]: $ennreal\ x \leq 1 \longleftrightarrow x \leq 1$
⟨*proof*⟩

lemma *ennreal-ge-1*[*simp*]: $ennreal\ x \geq 1 \longleftrightarrow x \geq 1$
⟨*proof*⟩

lemma *one-less-ennreal*[*simp*]: $1 < ennreal\ x \longleftrightarrow 1 < x$
⟨*proof*⟩

lemma *ennreal-plus*[*simp*]:
$0 \leq a \Longrightarrow 0 \leq b \Longrightarrow ennreal\ (a + b) = ennreal\ a + ennreal\ b$
⟨*proof*⟩

lemma *sum-ennreal*[*simp*]: $(\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i) \Longrightarrow (\sum i{\in}I.\ ennreal\ (f\ i)) = ennreal\ (sum\ f\ I)$
⟨*proof*⟩

lemma *sum-list-ennreal*[*simp*]:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x \geq 0$
shows $sum\text{-}list\ (map\ (\lambda x.\ ennreal\ (f\ x))\ xs) = ennreal\ (sum\text{-}list\ (map\ f\ xs))$
⟨*proof*⟩

lemma *ennreal-of-nat-eq-real-of-nat*: $of\text{-}nat\ i = ennreal\ (of\text{-}nat\ i)$
⟨*proof*⟩

lemma *of-nat-le-ennreal-iff*[*simp*]: $0 \leq r \Longrightarrow of\text{-}nat\ i \leq ennreal\ r \longleftrightarrow of\text{-}nat\ i \leq r$
⟨*proof*⟩

lemma *ennreal-le-of-nat-iff*[*simp*]: $ennreal\ r \leq of\text{-}nat\ i \longleftrightarrow r \leq of\text{-}nat\ i$
⟨*proof*⟩

lemma *ennreal-indicator*: $ennreal\ (indicator\ A\ x) = indicator\ A\ x$
⟨*proof*⟩

lemma *ennreal-numeral*[*simp*]: $ennreal\ (numeral\ n) = numeral\ n$
⟨*proof*⟩

lemma *min-ennreal*: $0 \leq x \Longrightarrow 0 \leq y \Longrightarrow min\ (ennreal\ x)\ (ennreal\ y) = ennreal\ (min\ x\ y)$
⟨*proof*⟩

lemma *ennreal-half*[*simp*]: $ennreal\ (1/2) = inverse\ 2$
⟨*proof*⟩

lemma *ennreal-minus*: $0 \leq q \Longrightarrow ennreal\ r - ennreal\ q = ennreal\ (r - q)$
⟨*proof*⟩

lemma *ennreal-minus-top*[*simp*]: $ennreal\ a - top = 0$
⟨*proof*⟩

lemma *ennreal-mult*: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow ennreal\ (a * b) = ennreal\ a * ennreal\ b$
⟨*proof*⟩

lemma *ennreal-mult'*: $0 \leq a \Longrightarrow ennreal\ (a * b) = ennreal\ a * ennreal\ b$
⟨*proof*⟩

lemma *indicator-mult-ennreal*: $indicator\ A\ x * ennreal\ r = ennreal\ (indicator\ A\ x * r)$
⟨*proof*⟩

lemma *ennreal-mult''*: $0 \leq b \Longrightarrow ennreal\ (a * b) = ennreal\ a * ennreal\ b$
⟨*proof*⟩

lemma *numeral-mult-ennreal*: $0 \leq x \Longrightarrow numeral\ b * ennreal\ x = ennreal\ (numeral\ b * x)$
⟨*proof*⟩

lemma *ennreal-power*: $0 \leq r \Longrightarrow ennreal\ r\ \hat{}\ n = ennreal\ (r\ \hat{}\ n)$
⟨*proof*⟩

lemma *power-eq-top-ennreal*: $x\ \hat{}\ n = top \longleftrightarrow (n \neq 0 \wedge (x{::}ennreal) = top)$
⟨*proof*⟩

lemma *inverse-ennreal*: $0 < r \Longrightarrow inverse\ (ennreal\ r) = ennreal\ (inverse\ r)$
⟨*proof*⟩

lemma *divide-ennreal*: $0 \leq r \Longrightarrow 0 < q \Longrightarrow ennreal\ r\ /\ ennreal\ q = ennreal\ (r\ /\ q)$
⟨*proof*⟩

lemma *ennreal-inverse-power*: $inverse\ (x\ \hat{}\ n :: ennreal) = inverse\ x\ \hat{}\ n$
⟨*proof*⟩

lemma *ennreal-divide-numeral*: $0 \leq x \Longrightarrow ennreal\ x\ /\ numeral\ b = ennreal\ (x\ /\ numeral\ b)$
⟨*proof*⟩

lemma *prod-ennreal*: $(\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i) \Longrightarrow (\prod i{\in}A.\ ennreal\ (f\ i)) = ennreal\ (prod\ f\ A)$
⟨*proof*⟩

lemma *mult-right-ennreal-cancel*: $a * ennreal\ c = b * ennreal\ c \longleftrightarrow (a = b \vee c \leq 0)$
⟨*proof*⟩

lemma *ennreal-le-epsilon*:
$(\bigwedge e{::}real.\ y < top \Longrightarrow 0 < e \Longrightarrow x \leq y + ennreal\ e) \Longrightarrow x \leq y$
⟨*proof*⟩

lemma *ennreal-rat-dense*:
fixes $x\ y :: ennreal$
shows $x < y \Longrightarrow \exists r{::}rat.\ x < real\text{-}of\text{-}rat\ r \wedge real\text{-}of\text{-}rat\ r < y$
⟨*proof*⟩

lemma *ennreal-Ex-less-of-nat*: $(x{::}ennreal) < top \Longrightarrow \exists n.\ x < of\text{-}nat\ n$
⟨*proof*⟩

31.7 Coercion from *ennreal* to *real*

definition *enn2real x = real-of-ereal (enn2ereal x)*

lemma *enn2real-nonneg*[*simp*]: *0 ≤ enn2real x*
⟨*proof*⟩

lemma *enn2real-mono*: *a ≤ b ⟹ b < top ⟹ enn2real a ≤ enn2real b*
⟨*proof*⟩

lemma *enn2real-of-nat*[*simp*]: *enn2real (of-nat n) = n*
⟨*proof*⟩

lemma *enn2real-ennreal*[*simp*]: *0 ≤ r ⟹ enn2real (ennreal r) = r*
⟨*proof*⟩

lemma *ennreal-enn2real*[*simp*]: *r < top ⟹ ennreal (enn2real r) = r*
⟨*proof*⟩

lemma *real-of-ereal-enn2ereal*[*simp*]: *real-of-ereal (enn2ereal x) = enn2real x*
⟨*proof*⟩

lemma *enn2real-top*[*simp*]: *enn2real top = 0*
⟨*proof*⟩

lemma *enn2real-0*[*simp*]: *enn2real 0 = 0*
⟨*proof*⟩

lemma *enn2real-1*[*simp*]: *enn2real 1 = 1*
⟨*proof*⟩

lemma *enn2real-numeral*[*simp*]: *enn2real (numeral n) = (numeral n)*
⟨*proof*⟩

lemma *enn2real-mult*: *enn2real (a * b) = enn2real a * enn2real b*
⟨*proof*⟩

lemma *enn2real-leI*: *0 ≤ B ⟹ x ≤ ennreal B ⟹ enn2real x ≤ B*
⟨*proof*⟩

lemma *enn2real-positive-iff*: *0 < enn2real x ⟷ (0 < x ∧ x < top)*
⟨*proof*⟩

lemma *enn2real-eq-1-iff*[*simp*]: *enn2real x = 1 ⟷ x = 1*
⟨*proof*⟩

31.8 Coercion from *enat* to *ennreal*

definition *ennreal-of-enat* :: *enat ⇒ ennreal*
where

ennreal-of-enat n = (*case n of* ∞ ⇒ *top* | *enat n* ⇒ *of-nat n*)

declare [[*coercion ennreal-of-enat*]]
declare [[*coercion of-nat* :: *nat* ⇒ *ennreal*]]

lemma *ennreal-of-enat-infty*[*simp*]: *ennreal-of-enat* ∞ = ∞
⟨*proof*⟩

lemma *ennreal-of-enat-enat*[*simp*]: *ennreal-of-enat* (*enat n*) = *of-nat n*
⟨*proof*⟩

lemma *ennreal-of-enat-0*[*simp*]: *ennreal-of-enat 0* = *0*
⟨*proof*⟩

lemma *ennreal-of-enat-1*[*simp*]: *ennreal-of-enat 1* = *1*
⟨*proof*⟩

lemma *ennreal-top-neq-of-nat*[*simp*]: (*top*::*ennreal*) ≠ *of-nat i*
⟨*proof*⟩

lemma *ennreal-of-enat-inj*[*simp*]: *ennreal-of-enat i* = *ennreal-of-enat j* ⟷ *i* = *j*
⟨*proof*⟩

lemma *ennreal-of-enat-le-iff*[*simp*]: *ennreal-of-enat m* ≤ *ennreal-of-enat n* ⟷ *m* ≤ *n*
⟨*proof*⟩

lemma *of-nat-less-ennreal-of-nat*[*simp*]: *of-nat n* ≤ *ennreal-of-enat x* ⟷ *of-nat n* ≤ *x*
⟨*proof*⟩

lemma *ennreal-of-enat-Sup*: *ennreal-of-enat* (*Sup X*) = (*SUP x*:*X*. *ennreal-of-enat x*)
⟨*proof*⟩

lemma *ennreal-of-enat-eSuc*[*simp*]: *ennreal-of-enat* (*eSuc x*) = *1* + *ennreal-of-enat x*
⟨*proof*⟩

31.9 Topology on *ennreal*

lemma *enn2ereal-Iio*: *enn2ereal* −‘ {..<*a*} = (*if 0* ≤ *a then* {..< *e2ennreal a*} *else* {})
⟨*proof*⟩

lemma *enn2ereal-Ioi*: *enn2ereal* −‘ {*a* <..} = (*if 0* ≤ *a then* {*e2ennreal a* <..} *else UNIV*)
⟨*proof*⟩

instantiation *ennreal* :: *linear-continuum-topology*
begin

definition *open-ennreal* :: *ennreal set* ⇒ *bool*
where (*open* :: *ennreal set* ⇒ *bool*) = *generate-topology* (*range lessThan* ∪ *range greaterThan*)

instance
⟨*proof*⟩

end

lemma *continuous-on-e2ennreal*: *continuous-on A e2ennreal*
⟨*proof*⟩

lemma *continuous-at-e2ennreal*: *continuous* (*at x within A*) *e2ennreal*
⟨*proof*⟩

lemma *continuous-on-enn2ereal*: *continuous-on UNIV enn2ereal*
⟨*proof*⟩

lemma *continuous-at-enn2ereal*: *continuous* (*at x within A*) *enn2ereal*
⟨*proof*⟩

lemma *sup-continuous-e2ennreal*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* (λ*x*. *e2ennreal* (*f x*))
⟨*proof*⟩

lemma *sup-continuous-enn2ereal*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* (λ*x*. *enn2ereal* (*f x*))
⟨*proof*⟩

lemma *sup-continuous-mult-left-ennreal'*:
fixes *c* :: *ennreal*
shows *sup-continuous* (λ*x*. *c* ∗ *x*)
⟨*proof*⟩

lemma *sup-continuous-mult-left-ennreal*[*order-continuous-intros*]:
sup-continuous f ⟹ *sup-continuous* (λ*x*. *c* ∗ *f x* :: *ennreal*)
⟨*proof*⟩

lemma *sup-continuous-mult-right-ennreal*[*order-continuous-intros*]:
sup-continuous f ⟹ *sup-continuous* (λ*x*. *f x* ∗ *c* :: *ennreal*)
⟨*proof*⟩

lemma *sup-continuous-divide-ennreal*[*order-continuous-intros*]:
fixes *f g* :: *'a*::*complete-lattice* ⇒ *ennreal*
shows *sup-continuous f* ⟹ *sup-continuous* (λ*x*. *f x* / *c*)
⟨*proof*⟩

lemma *transfer-enn2ereal-continuous-on* [*transfer-rule*]:
rel-fun (*op* =) (*rel-fun* (*rel-fun op* = *pcr-ennreal*) *op* =) *continuous-on continuous-on*
⟨*proof*⟩

lemma *transfer-sup-continuous*[*transfer-rule*]:
(*rel-fun* (*rel-fun* (*op* =) *pcr-ennreal*) *op* =) *sup-continuous sup-continuous*
⟨*proof*⟩

lemma *continuous-on-ennreal*[*tendsto-intros*]:
continuous-on A f ⟹ *continuous-on A* (λ*x*. *ennreal* (*f x*))
⟨*proof*⟩

lemma *tendsto-ennrealD*:
assumes *lim*: ((λ*x*. *ennreal* (*f x*)) ⟶ *ennreal x*) *F*
assumes ∗: ∀ *F x in F*. *0* ≤ *f x* **and** *x*: *0* ≤ *x*
shows (*f* ⟶ *x*) *F*
⟨*proof*⟩

lemma *tendsto-ennreal-iff*[*simp*]:
∀ *F x in F*. *0* ≤ *f x* ⟹ *0* ≤ *x* ⟹ ((λ*x*. *ennreal* (*f x*)) ⟶ *ennreal x*) *F* ⟷
(*f* ⟶ *x*) *F*
⟨*proof*⟩

lemma *tendsto-enn2ereal-iff*[*simp*]: ((λ*i*. *enn2ereal* (*f i*)) ⟶ *enn2ereal x*) *F*
⟷ (*f* ⟶ *x*) *F*
⟨*proof*⟩

lemma *continuous-on-add-ennreal*:
fixes *f g* :: *'a*::*topological-space* ⇒ *ennreal*
shows *continuous-on A f* ⟹ *continuous-on A g* ⟹ *continuous-on A* (λ*x*. *f x*
+ *g x*)
⟨*proof*⟩

lemma *continuous-on-inverse-ennreal*[*continuous-intros*]:
fixes *f* :: *'a*::*topological-space* ⇒ *ennreal*
shows *continuous-on A f* ⟹ *continuous-on A* (λ*x*. *inverse* (*f x*))
⟨*proof*⟩

instance *ennreal* :: *topological-comm-monoid-add*
⟨*proof*⟩

lemma *sup-continuous-add-ennreal*[*order-continuous-intros*]:
fixes *f g* :: *'a*::*complete-lattice* ⇒ *ennreal*
shows *sup-continuous f* ⟹ *sup-continuous g* ⟹ *sup-continuous* (λ*x*. *f x* + *g*
x)
⟨*proof*⟩

lemma *ennreal-suminf-lessD*: (∑ *i*. *f i* :: *ennreal*) < *x* ⟹ *f i* < *x*

⟨*proof*⟩

lemma *sums-ennreal*[*simp*]: $(\bigwedge i.\ 0 \le f\ i) \Longrightarrow 0 \le x \Longrightarrow (\lambda i.\ ennreal\ (f\ i))\ sums\ ennreal\ x \longleftrightarrow f\ sums\ x$
⟨*proof*⟩

lemma *summable-suminf-not-top*: $(\bigwedge i.\ 0 \le f\ i) \Longrightarrow (\sum i.\ ennreal\ (f\ i)) \neq top \Longrightarrow summable\ f$
⟨*proof*⟩

lemma *suminf-ennreal*[*simp*]:
$(\bigwedge i.\ 0 \le f\ i) \Longrightarrow (\sum i.\ ennreal\ (f\ i)) \neq top \Longrightarrow (\sum i.\ ennreal\ (f\ i)) = ennreal\ (\sum i.\ f\ i)$
⟨*proof*⟩

lemma *sums-enn2ereal*[*simp*]: $(\lambda i.\ enn2ereal\ (f\ i))\ sums\ enn2ereal\ x \longleftrightarrow f\ sums\ x$
⟨*proof*⟩

lemma *suminf-enn2ereal*[*simp*]: $(\sum i.\ enn2ereal\ (f\ i)) = enn2ereal\ (suminf\ f)$
⟨*proof*⟩

lemma *transfer-e2ennreal-suminf* [*transfer-rule*]: *rel-fun* (*rel-fun op* = *pcr-ennreal*) *pcr-ennreal suminf suminf*
⟨*proof*⟩

lemma *ennreal-suminf-cmult*[*simp*]: $(\sum i.\ r * f\ i) = r * (\sum i.\ f\ i :: ennreal)$
⟨*proof*⟩

lemma *ennreal-suminf-multc*[*simp*]: $(\sum i.\ f\ i * r) = (\sum i.\ f\ i :: ennreal) * r$
⟨*proof*⟩

lemma *ennreal-suminf-divide*[*simp*]: $(\sum i.\ f\ i\ /\ r) = (\sum i.\ f\ i :: ennreal)\ /\ r$
⟨*proof*⟩

lemma *ennreal-suminf-neq-top*: $summable\ f \Longrightarrow (\bigwedge i.\ 0 \le f\ i) \Longrightarrow (\sum i.\ ennreal\ (f\ i)) \neq top$
⟨*proof*⟩

lemma *suminf-ennreal-eq*:
$(\bigwedge i.\ 0 \le f\ i) \Longrightarrow f\ sums\ x \Longrightarrow (\sum i.\ ennreal\ (f\ i)) = ennreal\ x$
⟨*proof*⟩

lemma *ennreal-suminf-bound-add*:
fixes $f :: nat \Rightarrow ennreal$
shows $(\bigwedge N.\ (\sum n{<}N.\ f\ n) + y \le x) \Longrightarrow suminf\ f + y \le x$
⟨*proof*⟩

lemma *ennreal-suminf-SUP-eq-directed*:

fixes $f :: 'a \Rightarrow nat \Rightarrow ennreal$
assumes $*$: $\bigwedge N\ i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow finite\ N \Longrightarrow \exists k{\in}I.\ \forall n{\in}N.\ f\ i\ n \leq f\ k\ n \wedge f\ j\ n \leq f\ k\ n$
shows $(\sum n.\ SUP\ i{:}I.\ f\ i\ n) = (SUP\ i{:}I.\ \sum n.\ f\ i\ n)$
⟨*proof*⟩

lemma *INF-ennreal-add-const*:
fixes $f\ g :: nat \Rightarrow ennreal$
shows $(INF\ i.\ f\ i + c) = (INF\ i.\ f\ i) + c$
⟨*proof*⟩

lemma *INF-ennreal-const-add*:
fixes $f\ g :: nat \Rightarrow ennreal$
shows $(INF\ i.\ c + f\ i) = c + (INF\ i.\ f\ i)$
⟨*proof*⟩

lemma *SUP-mult-left-ennreal*: $c * (SUP\ i{:}I.\ f\ i) = (SUP\ i{:}I.\ c * f\ i :: ennreal)$
⟨*proof*⟩

lemma *SUP-mult-right-ennreal*: $(SUP\ i{:}I.\ f\ i) * c = (SUP\ i{:}I.\ f\ i * c :: ennreal)$
⟨*proof*⟩

lemma *SUP-divide-ennreal*: $(SUP\ i{:}I.\ f\ i) / c = (SUP\ i{:}I.\ f\ i / c :: ennreal)$
⟨*proof*⟩

lemma *ennreal-SUP-of-nat-eq-top*: $(SUP\ x.\ of\text{-}nat\ x :: ennreal) = top$
⟨*proof*⟩

lemma *ennreal-SUP-eq-top*:
fixes $f :: 'a \Rightarrow ennreal$
assumes $\bigwedge n.\ \exists i{\in}I.\ of\text{-}nat\ n \leq f\ i$
shows $(SUP\ i : I.\ f\ i) = top$
⟨*proof*⟩

lemma *ennreal-INF-const-minus*:
fixes $f :: 'a \Rightarrow ennreal$
shows $I \neq \{\} \Longrightarrow (SUP\ x{:}I.\ c - f\ x) = c - (INF\ x{:}I.\ f\ x)$
⟨*proof*⟩

lemma *of-nat-Sup-ennreal*:
assumes $A \neq \{\}$ *bdd-above* A
shows $of\text{-}nat\ (Sup\ A) = (SUP\ a{:}A.\ of\text{-}nat\ a :: ennreal)$
⟨*proof*⟩

lemma *ennreal-tendsto-const-minus*:
fixes $g :: 'a \Rightarrow ennreal$
assumes ae: $\forall_F\ x\ in\ F.\ g\ x \leq c$
assumes g: $((\lambda x.\ c - g\ x) \longrightarrow 0)\ F$
shows $(g \longrightarrow c)\ F$

⟨*proof*⟩

lemma *ennreal-SUP-add*:
fixes *f g* :: *nat* ⇒ *ennreal*
shows *incseq f* ⟹ *incseq g* ⟹ (*SUP i*. *f i* + *g i*) = *SUPREMUM UNIV f* + *SUPREMUM UNIV g*
⟨*proof*⟩

lemma *ennreal-SUP-sum*:
fixes *f* :: *'a* ⇒ *nat* ⇒ *ennreal*
shows (⋀*i*. *i* ∈ *I* ⟹ *incseq* (*f i*)) ⟹ (*SUP n*. ∑ *i*∈*I*. *f i n*) = (∑ *i*∈*I*. *SUP n*. *f i n*)
⟨*proof*⟩

lemma *ennreal-liminf-minus*:
fixes *f* :: *nat* ⇒ *ennreal*
shows (⋀*n*. *f n* ≤ *c*) ⟹ *liminf* (λ*n*. *c* − *f n*) = *c* − *limsup f*
⟨*proof*⟩

lemma *ennreal-continuous-on-cmult*:
(*c*::*ennreal*) < *top* ⟹ *continuous-on A f* ⟹ *continuous-on A* (λ*x*. *c* ∗ *f x*)
⟨*proof*⟩

lemma *ennreal-tendsto-cmult*:
(*c*::*ennreal*) < *top* ⟹ (*f* ⟶ *x*) *F* ⟹ ((λ*x*. *c* ∗ *f x*) ⟶ *c* ∗ *x*) *F*
⟨*proof*⟩

lemma *tendsto-ennrealI*[*intro*, *simp*]:
(*f* ⟶ *x*) *F* ⟹ ((λ*x*. *ennreal* (*f x*)) ⟶ *ennreal x*) *F*
⟨*proof*⟩

lemma *ennreal-suminf-minus*:
fixes *f g* :: *nat* ⇒ *ennreal*
shows (⋀*i*. *g i* ≤ *f i*) ⟹ *suminf f* ≠ *top* ⟹ *suminf g* ≠ *top* ⟹ (∑ *i*. *f i* − *g i*) = *suminf f* − *suminf g*
⟨*proof*⟩

lemma *ennreal-Sup-countable-SUP*:
A ≠ {} ⟹ ∃*f*::*nat* ⇒ *ennreal*. *incseq f* ∧ *range f* ⊆ *A* ∧ *Sup A* = (*SUP i*. *f i*)
⟨*proof*⟩

lemma *ennreal-Inf-countable-INF*:
A ≠ {} ⟹ ∃*f*::*nat* ⇒ *ennreal*. *decseq f* ∧ *range f* ⊆ *A* ∧ *Inf A* = (*INF i*. *f i*)
including *ennreal.lifting*
⟨*proof*⟩

lemma *ennreal-SUP-countable-SUP*:
A ≠ {} ⟹ ∃*f*::*nat* ⇒ *ennreal*. *range f* ⊆ *g*‘*A* ∧ *SUPREMUM A g* = *SUPREMUM UNIV f*

⟨*proof*⟩

lemma *of-nat-tendsto-top-ennreal*: $(\lambda n{::}nat.\ of\text{-}nat\ n :: ennreal) \longrightarrow top$
⟨*proof*⟩

lemma *SUP-sup-continuous-ennreal*:
fixes $f :: ennreal \Rightarrow 'a{::}complete\text{-}lattice$
assumes f: *sup-continuous* f **and** $I \neq \{\}$
shows $(SUP\ i{:}I.\ f\ (g\ i)) = f\ (SUP\ i{:}I.\ g\ i)$
⟨*proof*⟩

lemma *ennreal-suminf-SUP-eq*:
fixes $f :: nat \Rightarrow nat \Rightarrow ennreal$
shows $(\bigwedge i.\ incseq\ (\lambda n.\ f\ n\ i)) \Longrightarrow (\sum i.\ SUP\ n.\ f\ n\ i) = (SUP\ n.\ \sum i.\ f\ n\ i)$
⟨*proof*⟩

lemma *ennreal-SUP-add-left*:
fixes $c :: ennreal$
shows $I \neq \{\} \Longrightarrow (SUP\ i{:}I.\ f\ i + c) = (SUP\ i{:}I.\ f\ i) + c$
⟨*proof*⟩

lemma *ennreal-SUP-const-minus*:
fixes $f :: 'a \Rightarrow ennreal$
shows $I \neq \{\} \Longrightarrow c < top \Longrightarrow (INF\ x{:}I.\ c - f\ x) = c - (SUP\ x{:}I.\ f\ x)$
⟨*proof*⟩

31.10 Approximation lemmas

lemma *INF-approx-ennreal*:
fixes $x{::}ennreal$ **and** $e{::}real$
assumes $e > 0$
assumes *INF*: $x = (INF\ i : A.\ f\ i)$
assumes $x \neq \infty$
shows $\exists i \in A.\ f\ i < x + e$
⟨*proof*⟩

lemma *SUP-approx-ennreal*:
fixes $x{::}ennreal$ **and** $e{::}real$
assumes $e > 0\ A \neq \{\}$
assumes *SUP*: $x = (SUP\ i : A.\ f\ i)$
assumes $x \neq \infty$
shows $\exists i \in A.\ x < f\ i + e$
⟨*proof*⟩

lemma *ennreal-approx-SUP*:
fixes $x{::}ennreal$
assumes *f-bound*: $\bigwedge i.\ i \in A \Longrightarrow f\ i \leq x$
assumes *approx*: $\bigwedge e.\ (e{::}real) > 0 \Longrightarrow \exists i \in A.\ x \leq f\ i + e$
shows $x = (SUP\ i : A.\ f\ i)$

⟨*proof*⟩

lemma *ennreal-approx-INF*:
fixes *x*::*ennreal*
assumes *f-bound*: $\bigwedge i.\ i \in A \Longrightarrow x \leq f\ i$
assumes *approx*: $\bigwedge e.\ (e{::}real) > 0 \Longrightarrow \exists\, i \in A.\ f\ i \leq x + e$
shows $x = (INF\ i : A.\ f\ i)$
⟨*proof*⟩

lemma *ennreal-approx-unit*:
$(\bigwedge a{::}ennreal.\ 0 < a \Longrightarrow a < 1 \Longrightarrow a * z \leq y) \Longrightarrow z \leq y$
⟨*proof*⟩

lemma *suminf-ennreal2*:
$(\bigwedge i.\ 0 \leq f\ i) \Longrightarrow summable\ f \Longrightarrow (\sum i.\ ennreal\ (f\ i)) = ennreal\ (\sum i.\ f\ i)$
⟨*proof*⟩

lemma *less-top-ennreal*: $x < top \longleftrightarrow (\exists\, r{\geq}0.\ x = ennreal\ r)$
⟨*proof*⟩

lemma *tendsto-top-iff-ennreal*:
fixes $f :: 'a \Rightarrow ennreal$
shows $(f \longrightarrow top)\ F \longleftrightarrow (\forall\, l{\geq}0.\ eventually\ (\lambda x.\ ennreal\ l < f\ x)\ F)$
⟨*proof*⟩

lemma *ennreal-tendsto-top-eq-at-top*:
$((\lambda z.\ ennreal\ (f\ z)) \longrightarrow top)\ F \longleftrightarrow (LIM\ z\ F.\ f\ z :> at\text{-}top)$
⟨*proof*⟩

lemma *tendsto-0-if-Limsup-eq-0-ennreal*:
fixes $f :: - \Rightarrow ennreal$
shows $Limsup\ F\ f = 0 \Longrightarrow (f \longrightarrow 0)\ F$
⟨*proof*⟩

lemma *diff-le-self-ennreal*[*simp*]: $a - b \leq (a{::}ennreal)$
⟨*proof*⟩

lemma *ennreal-ineq-diff-add*: $b \leq a \Longrightarrow a = b + (a - b{::}ennreal)$
⟨*proof*⟩

lemma *ennreal-mult-strict-left-mono*: $(a{::}ennreal) < c \Longrightarrow 0 < b \Longrightarrow b < top \Longrightarrow b * a < b * c$
⟨*proof*⟩

lemma *ennreal-between*: $0 < e \Longrightarrow 0 < x \Longrightarrow x < top \Longrightarrow x - e < (x{::}ennreal)$
⟨*proof*⟩

lemma *minus-less-iff-ennreal*: $b < top \Longrightarrow b \leq a \Longrightarrow a - b < c \longleftrightarrow a < c + (b{::}ennreal)$

⟨*proof*⟩

lemma *tendsto-zero-ennreal*:
assumes *ev*: $\bigwedge r.\ 0 < r \Longrightarrow \forall_F\ x\ in\ F.\ f\ x < ennreal\ r$
shows $(f \longrightarrow 0)\ F$
⟨*proof*⟩

lifting-update *ennreal.lifting*
lifting-forget *ennreal.lifting*

31.11 *ennreal* theorems

lemma *neq-top-trans*: **fixes** $x\ y :: ennreal$ **shows** $[\![\ y \neq top;\ x \leq y\ ]\!] \Longrightarrow x \neq top$
⟨*proof*⟩

lemma *diff-diff-ennreal*: **fixes** $a\ b :: ennreal$ **shows** $a \leq b \Longrightarrow b \neq \infty \Longrightarrow b - (b - a) = a$
⟨*proof*⟩

lemma *ennreal-less-one-iff* [*simp*]: $ennreal\ x < 1 \longleftrightarrow x < 1$
⟨*proof*⟩

lemma *SUP-const-minus-ennreal*:
fixes $f :: 'a \Rightarrow ennreal$ **shows** $I \neq \{\} \Longrightarrow (SUP\ x{:}I.\ c - f\ x) = c - (INF\ x{:}I.\ f\ x)$
including *ennreal.lifting*
⟨*proof*⟩

lemma *zero-minus-ennreal*[*simp*]: $0 - (a{::}ennreal) = 0$
including *ennreal.lifting*
⟨*proof*⟩

lemma *diff-diff-commute-ennreal*:
fixes $a\ b\ c :: ennreal$ **shows** $a - b - c = a - c - b$
⟨*proof*⟩

lemma *diff-gr0-ennreal*: $b < (a{::}ennreal) \Longrightarrow 0 < a - b$
including *ennreal.lifting* ⟨*proof*⟩

lemma *divide-le-posI-ennreal*:
fixes $x\ y\ z :: ennreal$
shows $x > 0 \Longrightarrow z \leq x * y \Longrightarrow z\ /\ x \leq y$
⟨*proof*⟩

lemma *add-diff-eq-ennreal*:
fixes $x\ y\ z :: ennreal$
shows $z \leq y \Longrightarrow x + (y - z) = x + y - z$
including *ennreal.lifting*
⟨*proof*⟩

lemma *add-diff-inverse-ennreal*:
fixes $x\ y :: ennreal$ **shows** $x \leq y \Longrightarrow x + (y - x) = y$
⟨*proof*⟩

lemma *add-diff-eq-iff-ennreal*[*simp*]:
fixes $x\ y :: ennreal$ **shows** $x + (y - x) = y \longleftrightarrow x \leq y$
⟨*proof*⟩

lemma *add-diff-le-ennreal*: $a + b - c \leq a + (b - c{::}ennreal)$
⟨*proof*⟩

lemma *diff-eq-0-ennreal*: $a < top \Longrightarrow a \leq b \Longrightarrow a - b = (0{::}ennreal)$
⟨*proof*⟩

lemma *diff-diff-ennreal'*: **fixes** $x\ y\ z :: ennreal$ **shows** $z \leq y \Longrightarrow y - z \leq x \Longrightarrow x - (y - z) = x + z - y$
⟨*proof*⟩

lemma *diff-diff-ennreal''*: **fixes** $x\ y\ z :: ennreal$
shows $z \leq y \Longrightarrow x - (y - z) = (if\ y - z \leq x\ then\ x + z - y\ else\ 0)$
⟨*proof*⟩

lemma *power-less-top-ennreal*: **fixes** $x :: ennreal$ **shows** $x\ \hat{}\ n < top \longleftrightarrow x < top \vee n = 0$
⟨*proof*⟩

lemma *ennreal-divide-times*: $(a\ /\ b) * c = a * (c\ /\ b :: ennreal)$
⟨*proof*⟩

lemma *diff-less-top-ennreal*: $a - b < top \longleftrightarrow\ a < (top :: ennreal)$
⟨*proof*⟩

lemma *divide-less-ennreal*: $b \neq 0 \Longrightarrow b < top \Longrightarrow a\ /\ b < c \longleftrightarrow a < (c * b :: ennreal)$
⟨*proof*⟩

lemma *one-less-numeral*[*simp*]: $1 < (numeral\ n{::}ennreal) \longleftrightarrow (num.One < n)$
⟨*proof*⟩

lemma *divide-eq-1-ennreal*: $a\ /\ b = (1{::}ennreal) \longleftrightarrow (b \neq top \wedge b \neq 0 \wedge b = a)$
⟨*proof*⟩

lemma *ennreal-mult-cancel-left*: $(a * b = a * c) = (a = top \wedge b \neq 0 \wedge c \neq 0 \vee a = 0 \vee b = (c{::}ennreal))$
⟨*proof*⟩

lemma *ennreal-minus-if*: $ennreal\ a - ennreal\ b = ennreal\ (if\ 0 \leq b\ then\ (if\ b \leq a\ then\ a - b\ else\ 0)\ else\ a)$

⟨*proof*⟩

lemma *ennreal-plus-if*: *ennreal a + ennreal b = ennreal (if 0 ≤ a then (if 0 ≤ b then a + b else a) else b)*
⟨*proof*⟩

lemma *power-le-one-iff*: *0 ≤ (a::real) ⟹ a ^ n ≤ 1 ⟷ (n = 0 ∨ a ≤ 1)*
⟨*proof*⟩

lemma *ennreal-diff-le-mono-left*: *a ≤ b ⟹ a − c ≤ (b::ennreal)*
⟨*proof*⟩

lemma *ennreal-minus-le-iff*: *a − b ≤ c ⟷ (a ≤ b + (c::ennreal) ∧ (a = top ∧ b = top ⟶ c = top))*
⟨*proof*⟩

lemma *ennreal-le-minus-iff*: *a ≤ b − c ⟷ (a + c ≤ (b::ennreal) ∨ (a = 0 ∧ b ≤ c))*
⟨*proof*⟩

lemma *diff-add-eq-diff-diff-swap-ennreal*: *x − (y + z :: ennreal) = x − y − z*
⟨*proof*⟩

lemma *diff-add-assoc2-ennreal*: *b ≤ a ⟹ (a − b + c::ennreal) = a + c − b*
⟨*proof*⟩

lemma *diff-gt-0-iff-gt-ennreal*: *0 < a − b ⟷ (a = top ∧ b = top ∨ b < (a::ennreal))*
⟨*proof*⟩

lemma *diff-eq-0-iff-ennreal*: *(a − b::ennreal) = 0 ⟷ (a < top ∧ a ≤ b)*
⟨*proof*⟩

lemma *add-diff-self-ennreal*: *a + (b − a::ennreal) = (if a ≤ b then b else a)*
⟨*proof*⟩

lemma *diff-add-self-ennreal*: *(b − a + a::ennreal) = (if a ≤ b then b else a)*
⟨*proof*⟩

lemma *ennreal-minus-cancel-iff*:
fixes *a b c :: ennreal*
shows *a − b = a − c ⟷ (b = c ∨ (a ≤ b ∧ a ≤ c) ∨ a = top)*
⟨*proof*⟩

lemma *SUP-diff-ennreal*:
c < top ⟹ (SUP i:I. f i − c :: ennreal) = (SUP i:I. f i) − c
⟨*proof*⟩

lemma *ennreal-SUP-add-right*:

fixes c :: *ennreal* **shows** $I \neq \{\} \Longrightarrow c + (SUP\ i{:}I.\ f\ i) = (SUP\ i{:}I.\ c + f\ i)$
⟨*proof*⟩

lemma *SUP-add-directed-ennreal*:
fixes $f\ g$ :: - ⇒ *ennreal*
assumes *directed*: $\bigwedge i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow \exists k{\in}I.\ f\ i + g\ j \leq f\ k + g\ k$
shows $(SUP\ i{:}I.\ f\ i + g\ i) = (SUP\ i{:}I.\ f\ i) + (SUP\ i{:}I.\ g\ i)$
⟨*proof*⟩

lemma *enn2real-eq-0-iff*: *enn2real* $x = 0 \longleftrightarrow x = 0 \vee x = top$
⟨*proof*⟩

lemma (**in** −) *continuous-on-diff-ereal*:
continuous-on $A\ f \Longrightarrow$ *continuous-on* $A\ g \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow |f\ x| \neq \infty) \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow |g\ x| \neq \infty) \Longrightarrow$ *continuous-on* $A\ (\lambda z.\ f\ z - g\ z{::}ereal)$
⟨*proof*⟩

lemma (**in** −) *continuous-on-diff-ennreal*:
continuous-on $A\ f \Longrightarrow$ *continuous-on* $A\ g \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow f\ x \neq top) \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow g\ x \neq top) \Longrightarrow$ *continuous-on* $A\ (\lambda z.\ f\ z - g\ z{::}ennreal)$
including *ennreal.lifting*
⟨*proof*⟩

lemma (**in** −) *tendsto-diff-ennreal*:
$(f \longrightarrow x)\ F \Longrightarrow (g \longrightarrow y)\ F \Longrightarrow x \neq top \Longrightarrow y \neq top \Longrightarrow ((\lambda z.\ f\ z - g\ z{::}ennreal) \longrightarrow x - y)\ F$
⟨*proof*⟩

end

32 Type of finite sets defined as a subtype of sets

theory *FSet*
imports *Main Countable*
begin

32.1 Definition of the type

typedef *'a fset* = $\{A$ :: *'a set*. *finite* $A\}$ **morphisms** *fset Abs-fset*
⟨*proof*⟩

setup-lifting *type-definition-fset*

32.2 Basic operations and type class instantiations

instantiation *fset* :: (*finite*) *finite*
begin
instance ⟨*proof*⟩
end

instantiation *fset* :: (*type*) {*bounded-lattice-bot*, *distrib-lattice*, *minus*}
begin

lift-definition *bot-fset* :: *'a fset* **is** {} **parametric** *empty-transfer* ⟨*proof*⟩

lift-definition *less-eq-fset* :: *'a fset* ⇒ *'a fset* ⇒ *bool* **is** *subset-eq* **parametric** *subset-transfer*
⟨*proof*⟩

definition *less-fset* :: *'a fset* ⇒ *'a fset* ⇒ *bool* **where** *xs* < *ys* ≡ *xs* ≤ *ys* ∧ *xs* ≠ (*ys*::*'a fset*)

lemma *less-fset-transfer*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique A*
shows ((*pcr-fset A*) ===> (*pcr-fset A*) ===> *op* =) *op* ⊂ *op* <
⟨*proof*⟩

lift-definition *sup-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *union* **parametric** *union-transfer*
⟨*proof*⟩

lift-definition *inf-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *inter* **parametric** *inter-transfer*
⟨*proof*⟩

lift-definition *minus-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *minus* **parametric** *Diff-transfer*
⟨*proof*⟩

instance
⟨*proof*⟩

end

abbreviation *fempty* :: *'a fset* ({||}) **where** {||} ≡ *bot*
abbreviation *fsubset-eq* :: *'a fset* ⇒ *'a fset* ⇒ *bool* (**infix** |⊆| *50*) **where** *xs* |⊆| *ys* ≡ *xs* ≤ *ys*
abbreviation *fsubset* :: *'a fset* ⇒ *'a fset* ⇒ *bool* (**infix** |⊂| *50*) **where** *xs* |⊂| *ys* ≡ *xs* < *ys*
abbreviation *funion* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* (**infixl** |∪| *65*) **where** *xs* |∪| *ys* ≡ *sup xs ys*
abbreviation *finter* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* (**infixl** |∩| *65*) **where** *xs* |∩| *ys* ≡ *inf xs ys*
abbreviation *fminus* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* (**infixl** |−| *65*) **where** *xs* |−| *ys* ≡ *minus xs ys*

instantiation *fset* :: (*equal*) *equal*
begin

definition *HOL.equal A B* $\longleftrightarrow$ *A* $|\subseteq|$ *B* $\wedge$ *B* $|\subseteq|$ *A*
instance ⟨*proof*⟩
end

instantiation *fset* :: (*type*) *conditionally-complete-lattice*
begin

context includes *lifting-syntax*
begin

lemma *right-total-Inf-fset-transfer*:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *right-total A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*)
(λ*S*. *if finite* ($\bigcap$ *S* ∩ *Collect* (*Domainp A*)) *then* $\bigcap$ *S* ∩ *Collect* (*Domainp A*) *else* {})
(λ*S*. *if finite* (*Inf S*) *then Inf S else* {})
⟨*proof*⟩

lemma *Inf-fset-transfer*:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *bi-total A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*) (λ*A*. *if finite* (*Inf A*) *then Inf A else* {})
(λ*A*. *if finite* (*Inf A*) *then Inf A else* {})
⟨*proof*⟩

lift-definition *Inf-fset* :: *'a fset set* ⇒ *'a fset* **is** λ*A*. *if finite* (*Inf A*) *then Inf A else* {}
parametric *right-total-Inf-fset-transfer Inf-fset-transfer* ⟨*proof*⟩

lemma *Sup-fset-transfer*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*) (λ*A*. *if finite* (*Sup A*) *then Sup A else* {})
(λ*A*. *if finite* (*Sup A*) *then Sup A else* {}) ⟨*proof*⟩

lift-definition *Sup-fset* :: *'a fset set* ⇒ *'a fset* **is** λ*A*. *if finite* (*Sup A*) *then Sup A else* {}
parametric *Sup-fset-transfer* ⟨*proof*⟩

lemma *finite-Sup*: ∃ *z*. *finite z* ∧ (∀ *a*. *a* ∈ *X* ⟶ *a* ≤ *z*) ⟹ *finite* (*Sup X*)
⟨*proof*⟩

lemma *transfer-bdd-below*[*transfer-rule*]: (*rel-set* (*pcr-fset op* =) ===> *op* =) *bdd-below bdd-below*
⟨*proof*⟩

end

instance

⟨*proof*⟩
end

instantiation *fset* :: (*finite*) *complete-lattice*
begin

lift-definition *top-fset* :: *'a fset* **is** *UNIV* **parametric** *right-total-UNIV-transfer UNIV-transfer*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *fset* :: (*finite*) *complete-boolean-algebra*
begin

lift-definition *uminus-fset* :: *'a fset* ⇒ *'a fset* **is** *uminus*
parametric *right-total-Compl-transfer Compl-transfer* ⟨*proof*⟩

instance
⟨*proof*⟩

end

abbreviation *fUNIV* :: *'a*::*finite fset* **where** *fUNIV* ≡ *top*
abbreviation *fuminus* :: *'a*::*finite fset* ⇒ *'a fset* (|−| - [*81*] *80*) **where** |−| *x* ≡ *uminus x*

declare *top-fset.rep-eq*[*simp*]

32.3 Other operations

lift-definition *finsert* :: *'a* ⇒ *'a fset* ⇒ *'a fset* **is** *insert* **parametric** *Lifting-Set.insert-transfer*
⟨*proof*⟩

syntax
-insert-fset :: *args* => *'a fset* ({|(-)|})

translations
{|*x*, *xs*|} == *CONST finsert x* {|*xs*|}
{|*x*|} == *CONST finsert x* {||}

lift-definition *fmember* :: *'a* ⇒ *'a fset* ⇒ *bool* (**infix** |∈| *50*) **is** *Set.member*
parametric *member-transfer* ⟨*proof*⟩

abbreviation *notin-fset* :: *'a* ⇒ *'a fset* ⇒ *bool* (**infix** |∉| *50*) **where** *x* |∉| *S* ≡ ¬ (*x* |∈| *S*)

context includes *lifting-syntax*
begin

lift-definition *ffilter* :: (*'a* ⇒ *bool*) ⇒ *'a fset* ⇒ *'a fset* **is** *Set.filter*
parametric *Lifting-Set.filter-transfer* ⟨*proof*⟩

lift-definition *fPow* :: *'a fset* ⇒ *'a fset fset* **is** *Pow* **parametric** *Pow-transfer*
⟨*proof*⟩

lift-definition *fcard* :: *'a fset* ⇒ *nat* **is** *card* **parametric** *card-transfer* ⟨*proof*⟩

lift-definition *fimage* :: (*'a* ⇒ *'b*) ⇒ *'a fset* ⇒ *'b fset* (**infixr** |‘| *90*) **is** *image*
parametric *image-transfer* ⟨*proof*⟩

lift-definition *fthe-elem* :: *'a fset* ⇒ *'a* **is** *the-elem* ⟨*proof*⟩

lift-definition *fbind* :: *'a fset* ⇒ (*'a* ⇒ *'b fset*) ⇒ *'b fset* **is** *Set.bind* **parametric**
bind-transfer
⟨*proof*⟩

lift-definition *ffUnion* :: *'a fset fset* ⇒ *'a fset* **is** *Union* **parametric** *Union-transfer*
⟨*proof*⟩

lift-definition *fBall* :: *'a fset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **is** *Ball* **parametric** *Ball-transfer*
⟨*proof*⟩
lift-definition *fBex* :: *'a fset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **is** *Bex* **parametric** *Bex-transfer*
⟨*proof*⟩

lift-definition *ffold* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'b* ⇒ *'a fset* ⇒ *'b* **is** *Finite-Set.fold*
⟨*proof*⟩

lift-definition *fset-of-list* :: *'a list* ⇒ *'a fset* **is** *set* ⟨*proof*⟩

32.4 Transferred lemmas from Set.thy

lemmas *fset-eqI* = *set-eqI*[*Transfer.transferred*]
lemmas *fset-eq-iff*[*no-atp*] = *set-eq-iff*[*Transfer.transferred*]
lemmas *fBallI*[*intro*!] = *ballI*[*Transfer.transferred*]
lemmas *fbspec*[*dest?*] = *bspec*[*Transfer.transferred*]
lemmas *fBallE*[*elim*] = *ballE*[*Transfer.transferred*]
lemmas *fBexI*[*intro*] = *bexI*[*Transfer.transferred*]
lemmas *rev-fBexI*[*intro?*] = *rev-bexI*[*Transfer.transferred*]
lemmas *fBexCI* = *bexCI*[*Transfer.transferred*]
lemmas *fBexE*[*elim*!] = *bexE*[*Transfer.transferred*]
lemmas *fBall-triv*[*simp*] = *ball-triv*[*Transfer.transferred*]
lemmas *fBex-triv*[*simp*] = *bex-triv*[*Transfer.transferred*]
lemmas *fBex-triv-one-point1*[*simp*] = *bex-triv-one-point1*[*Transfer.transferred*]
lemmas *fBex-triv-one-point2*[*simp*] = *bex-triv-one-point2*[*Transfer.transferred*]

lemmas *fBex-one-point1*[*simp*] = *bex-one-point1*[*Transfer.transferred*]
lemmas *fBex-one-point2*[*simp*] = *bex-one-point2*[*Transfer.transferred*]
lemmas *fBall-one-point1*[*simp*] = *ball-one-point1*[*Transfer.transferred*]
lemmas *fBall-one-point2*[*simp*] = *ball-one-point2*[*Transfer.transferred*]
lemmas *fBall-conj-distrib* = *ball-conj-distrib*[*Transfer.transferred*]
lemmas *fBex-disj-distrib* = *bex-disj-distrib*[*Transfer.transferred*]
lemmas *fBall-cong*[*fundef-cong*] = *ball-cong*[*Transfer.transferred*]
lemmas *fBex-cong*[*fundef-cong*] = *bex-cong*[*Transfer.transferred*]
lemmas *fsubsetI*[*intro*!] = *subsetI*[*Transfer.transferred*]
lemmas *fsubsetD*[*elim*, *intro?*] = *subsetD*[*Transfer.transferred*]
lemmas *rev-fsubsetD*[*no-atp*,*intro?*] = *rev-subsetD*[*Transfer.transferred*]
lemmas *fsubsetCE*[*no-atp*,*elim*] = *subsetCE*[*Transfer.transferred*]
lemmas *fsubset-eq*[*no-atp*] = *subset-eq*[*Transfer.transferred*]
lemmas *contra-fsubsetD*[*no-atp*] = *contra-subsetD*[*Transfer.transferred*]
lemmas *fsubset-refl* = *subset-refl*[*Transfer.transferred*]
lemmas *fsubset-trans* = *subset-trans*[*Transfer.transferred*]
lemmas *fset-rev-mp* = *set-rev-mp*[*Transfer.transferred*]
lemmas *fset-mp* = *set-mp*[*Transfer.transferred*]
lemmas *fsubset-not-fsubset-eq*[*code*] = *subset-not-subset-eq*[*Transfer.transferred*]
lemmas *eq-fmem-trans* = *eq-mem-trans*[*Transfer.transferred*]
lemmas *fsubset-antisym*[*intro*!] = *subset-antisym*[*Transfer.transferred*]
lemmas *fequalityD1* = *equalityD1*[*Transfer.transferred*]
lemmas *fequalityD2* = *equalityD2*[*Transfer.transferred*]
lemmas *fequalityE* = *equalityE*[*Transfer.transferred*]
lemmas *fequalityCE*[*elim*] = *equalityCE*[*Transfer.transferred*]
lemmas *eqfset-imp-iff* = *eqset-imp-iff*[*Transfer.transferred*]
lemmas *eqfelem-imp-iff* = *eqelem-imp-iff*[*Transfer.transferred*]
lemmas *fempty-iff*[*simp*] = *empty-iff*[*Transfer.transferred*]
lemmas *fempty-fsubsetI*[*iff*] = *empty-subsetI*[*Transfer.transferred*]
lemmas *equalsffemptyI* = *equals0I*[*Transfer.transferred*]
lemmas *equalsffemptyD* = *equals0D*[*Transfer.transferred*]
lemmas *fBall-fempty*[*simp*] = *ball-empty*[*Transfer.transferred*]
lemmas *fBex-fempty*[*simp*] = *bex-empty*[*Transfer.transferred*]
lemmas *fPow-iff*[*iff*] = *Pow-iff*[*Transfer.transferred*]
lemmas *fPowI* = *PowI*[*Transfer.transferred*]
lemmas *fPowD* = *PowD*[*Transfer.transferred*]
lemmas *fPow-bottom* = *Pow-bottom*[*Transfer.transferred*]
lemmas *fPow-top* = *Pow-top*[*Transfer.transferred*]
lemmas *fPow-not-fempty* = *Pow-not-empty*[*Transfer.transferred*]
lemmas *finter-iff*[*simp*] = *Int-iff*[*Transfer.transferred*]
lemmas *finterI*[*intro*!] = *IntI*[*Transfer.transferred*]
lemmas *finterD1* = *IntD1*[*Transfer.transferred*]
lemmas *finterD2* = *IntD2*[*Transfer.transferred*]
lemmas *finterE*[*elim*!] = *IntE*[*Transfer.transferred*]
lemmas *funion-iff*[*simp*] = *Un-iff*[*Transfer.transferred*]
lemmas *funionI1*[*elim?*] = *UnI1*[*Transfer.transferred*]
lemmas *funionI2*[*elim?*] = *UnI2*[*Transfer.transferred*]
lemmas *funionCI*[*intro*!] = *UnCI*[*Transfer.transferred*]
lemmas *funionE*[*elim*!] = *UnE*[*Transfer.transferred*]

lemmas *fminus-iff*[*simp*] = *Diff-iff*[*Transfer.transferred*]
lemmas *fminusI*[*intro*!] = *DiffI*[*Transfer.transferred*]
lemmas *fminusD1* = *DiffD1*[*Transfer.transferred*]
lemmas *fminusD2* = *DiffD2*[*Transfer.transferred*]
lemmas *fminusE*[*elim*!] = *DiffE*[*Transfer.transferred*]
lemmas *finsert-iff*[*simp*] = *insert-iff*[*Transfer.transferred*]
lemmas *finsertI1* = *insertI1*[*Transfer.transferred*]
lemmas *finsertI2* = *insertI2*[*Transfer.transferred*]
lemmas *finsertE*[*elim*!] = *insertE*[*Transfer.transferred*]
lemmas *finsertCI*[*intro*!] = *insertCI*[*Transfer.transferred*]
lemmas *fsubset-finsert-iff* = *subset-insert-iff*[*Transfer.transferred*]
lemmas *finsert-ident* = *insert-ident*[*Transfer.transferred*]
lemmas *fsingletonI*[*intro*!,*no-atp*] = *singletonI*[*Transfer.transferred*]
lemmas *fsingletonD*[*dest*!,*no-atp*] = *singletonD*[*Transfer.transferred*]
lemmas *fsingleton-iff* = *singleton-iff*[*Transfer.transferred*]
lemmas *fsingleton-inject*[*dest*!] = *singleton-inject*[*Transfer.transferred*]
lemmas *fsingleton-finsert-inj-eq*[*iff*,*no-atp*] = *singleton-insert-inj-eq*[*Transfer.transferred*]
lemmas *fsingleton-finsert-inj-eq'*[*iff*,*no-atp*] = *singleton-insert-inj-eq'*[*Transfer.transferred*]
lemmas *fsubset-fsingletonD* = *subset-singletonD*[*Transfer.transferred*]
lemmas *fminus-single-finsert* = *Diff-single-insert*[*Transfer.transferred*]
lemmas *fdoubleton-eq-iff* = *doubleton-eq-iff*[*Transfer.transferred*]
lemmas *funion-fsingleton-iff* = *Un-singleton-iff*[*Transfer.transferred*]
lemmas *fsingleton-funion-iff* = *singleton-Un-iff*[*Transfer.transferred*]
lemmas *fimage-eqI*[*simp*, *intro*] = *image-eqI*[*Transfer.transferred*]
lemmas *fimageI* = *imageI*[*Transfer.transferred*]
lemmas *rev-fimage-eqI* = *rev-image-eqI*[*Transfer.transferred*]
lemmas *fimageE*[*elim*!] = *imageE*[*Transfer.transferred*]
lemmas *Compr-fimage-eq* = *Compr-image-eq*[*Transfer.transferred*]
lemmas *fimage-funion* = *image-Un*[*Transfer.transferred*]
lemmas *fimage-iff* = *image-iff*[*Transfer.transferred*]
lemmas *fimage-fsubset-iff*[*no-atp*] = *image-subset-iff*[*Transfer.transferred*]
lemmas *fimage-fsubsetI* = *image-subsetI*[*Transfer.transferred*]
lemmas *fimage-ident*[*simp*] = *image-ident*[*Transfer.transferred*]
lemmas *if-split-fmem1* = *if-split-mem1*[*Transfer.transferred*]
lemmas *if-split-fmem2* = *if-split-mem2*[*Transfer.transferred*]
lemmas *pfsubsetI*[*intro*!,*no-atp*] = *psubsetI*[*Transfer.transferred*]
lemmas *pfsubsetE*[*elim*!,*no-atp*] = *psubsetE*[*Transfer.transferred*]
lemmas *pfsubset-finsert-iff* = *psubset-insert-iff*[*Transfer.transferred*]
lemmas *pfsubset-eq* = *psubset-eq*[*Transfer.transferred*]
lemmas *pfsubset-imp-fsubset* = *psubset-imp-subset*[*Transfer.transferred*]
lemmas *pfsubset-trans* = *psubset-trans*[*Transfer.transferred*]
lemmas *pfsubsetD* = *psubsetD*[*Transfer.transferred*]
lemmas *pfsubset-fsubset-trans* = *psubset-subset-trans*[*Transfer.transferred*]
lemmas *fsubset-pfsubset-trans* = *subset-psubset-trans*[*Transfer.transferred*]
lemmas *pfsubset-imp-ex-fmem* = *psubset-imp-ex-mem*[*Transfer.transferred*]
lemmas *fimage-fPow-mono* = *image-Pow-mono*[*Transfer.transferred*]
lemmas *fimage-fPow-surj* = *image-Pow-surj*[*Transfer.transferred*]
lemmas *fsubset-finsertI* = *subset-insertI*[*Transfer.transferred*]
lemmas *fsubset-finsertI2* = *subset-insertI2*[*Transfer.transferred*]

lemmas *fsubset-finsert* = *subset-insert*[*Transfer.transferred*]
lemmas *funion-upper1* = *Un-upper1*[*Transfer.transferred*]
lemmas *funion-upper2* = *Un-upper2*[*Transfer.transferred*]
lemmas *funion-least* = *Un-least*[*Transfer.transferred*]
lemmas *finter-lower1* = *Int-lower1*[*Transfer.transferred*]
lemmas *finter-lower2* = *Int-lower2*[*Transfer.transferred*]
lemmas *finter-greatest* = *Int-greatest*[*Transfer.transferred*]
lemmas *fminus-fsubset* = *Diff-subset*[*Transfer.transferred*]
lemmas *fminus-fsubset-conv* = *Diff-subset-conv*[*Transfer.transferred*]
lemmas *fsubset-fempty*[*simp*] = *subset-empty*[*Transfer.transferred*]
lemmas *not-pfsubset-fempty*[*iff*] = *not-psubset-empty*[*Transfer.transferred*]
lemmas *finsert-is-funion* = *insert-is-Un*[*Transfer.transferred*]
lemmas *finsert-not-fempty*[*simp*] = *insert-not-empty*[*Transfer.transferred*]
lemmas *fempty-not-finsert* = *empty-not-insert*[*Transfer.transferred*]
lemmas *finsert-absorb* = *insert-absorb*[*Transfer.transferred*]
lemmas *finsert-absorb2*[*simp*] = *insert-absorb2*[*Transfer.transferred*]
lemmas *finsert-commute* = *insert-commute*[*Transfer.transferred*]
lemmas *finsert-fsubset*[*simp*] = *insert-subset*[*Transfer.transferred*]
lemmas *finsert-inter-finsert*[*simp*] = *insert-inter-insert*[*Transfer.transferred*]
lemmas *finsert-disjoint*[*simp*,*no-atp*] = *insert-disjoint*[*Transfer.transferred*]
lemmas *disjoint-finsert*[*simp*,*no-atp*] = *disjoint-insert*[*Transfer.transferred*]
lemmas *fimage-fempty*[*simp*] = *image-empty*[*Transfer.transferred*]
lemmas *fimage-finsert*[*simp*] = *image-insert*[*Transfer.transferred*]
lemmas *fimage-constant* = *image-constant*[*Transfer.transferred*]
lemmas *fimage-constant-conv* = *image-constant-conv*[*Transfer.transferred*]
lemmas *fimage-fimage* = *image-image*[*Transfer.transferred*]
lemmas *finsert-fimage*[*simp*] = *insert-image*[*Transfer.transferred*]
lemmas *fimage-is-fempty*[*iff*] = *image-is-empty*[*Transfer.transferred*]
lemmas *fempty-is-fimage*[*iff*] = *empty-is-image*[*Transfer.transferred*]
lemmas *fimage-cong* = *image-cong*[*Transfer.transferred*]
lemmas *fimage-finter-fsubset* = *image-Int-subset*[*Transfer.transferred*]
lemmas *fimage-fminus-fsubset* = *image-diff-subset*[*Transfer.transferred*]
lemmas *finter-absorb* = *Int-absorb*[*Transfer.transferred*]
lemmas *finter-left-absorb* = *Int-left-absorb*[*Transfer.transferred*]
lemmas *finter-commute* = *Int-commute*[*Transfer.transferred*]
lemmas *finter-left-commute* = *Int-left-commute*[*Transfer.transferred*]
lemmas *finter-assoc* = *Int-assoc*[*Transfer.transferred*]
lemmas *finter-ac* = *Int-ac*[*Transfer.transferred*]
lemmas *finter-absorb1* = *Int-absorb1*[*Transfer.transferred*]
lemmas *finter-absorb2* = *Int-absorb2*[*Transfer.transferred*]
lemmas *finter-fempty-left* = *Int-empty-left*[*Transfer.transferred*]
lemmas *finter-fempty-right* = *Int-empty-right*[*Transfer.transferred*]
lemmas *disjoint-iff-fnot-equal* = *disjoint-iff-not-equal*[*Transfer.transferred*]
lemmas *finter-funion-distrib* = *Int-Un-distrib*[*Transfer.transferred*]
lemmas *finter-funion-distrib2* = *Int-Un-distrib2*[*Transfer.transferred*]
lemmas *finter-fsubset-iff*[*no-atp*, *simp*] = *Int-subset-iff*[*Transfer.transferred*]
lemmas *funion-absorb* = *Un-absorb*[*Transfer.transferred*]
lemmas *funion-left-absorb* = *Un-left-absorb*[*Transfer.transferred*]
lemmas *funion-commute* = *Un-commute*[*Transfer.transferred*]

lemmas *funion-left-commute* = *Un-left-commute*[*Transfer.transferred*]
lemmas *funion-assoc* = *Un-assoc*[*Transfer.transferred*]
lemmas *funion-ac* = *Un-ac*[*Transfer.transferred*]
lemmas *funion-absorb1* = *Un-absorb1*[*Transfer.transferred*]
lemmas *funion-absorb2* = *Un-absorb2*[*Transfer.transferred*]
lemmas *funion-fempty-left* = *Un-empty-left*[*Transfer.transferred*]
lemmas *funion-fempty-right* = *Un-empty-right*[*Transfer.transferred*]
lemmas *funion-finsert-left*[*simp*] = *Un-insert-left*[*Transfer.transferred*]
lemmas *funion-finsert-right*[*simp*] = *Un-insert-right*[*Transfer.transferred*]
lemmas *finter-finsert-left* = *Int-insert-left*[*Transfer.transferred*]
lemmas *finter-finsert-left-ifffempty*[*simp*] = *Int-insert-left-if0*[*Transfer.transferred*]
lemmas *finter-finsert-left-if1*[*simp*] = *Int-insert-left-if1*[*Transfer.transferred*]
lemmas *finter-finsert-right* = *Int-insert-right*[*Transfer.transferred*]
lemmas *finter-finsert-right-ifffempty*[*simp*] = *Int-insert-right-if0*[*Transfer.transferred*]
lemmas *finter-finsert-right-if1*[*simp*] = *Int-insert-right-if1*[*Transfer.transferred*]
lemmas *funion-finter-distrib* = *Un-Int-distrib*[*Transfer.transferred*]
lemmas *funion-finter-distrib2* = *Un-Int-distrib2*[*Transfer.transferred*]
lemmas *funion-finter-crazy* = *Un-Int-crazy*[*Transfer.transferred*]
lemmas *fsubset-funion-eq* = *subset-Un-eq*[*Transfer.transferred*]
lemmas *funion-fempty*[*iff*] = *Un-empty*[*Transfer.transferred*]
lemmas *funion-fsubset-iff*[*no-atp*, *simp*] = *Un-subset-iff*[*Transfer.transferred*]
lemmas *funion-fminus-finter* = *Un-Diff-Int*[*Transfer.transferred*]
lemmas *fminus-finter2* = *Diff-Int2*[*Transfer.transferred*]
lemmas *funion-finter-assoc-eq* = *Un-Int-assoc-eq*[*Transfer.transferred*]
lemmas *fBall-funion* = *ball-Un*[*Transfer.transferred*]
lemmas *fBex-funion* = *bex-Un*[*Transfer.transferred*]
lemmas *fminus-eq-fempty-iff*[*simp*,*no-atp*] = *Diff-eq-empty-iff*[*Transfer.transferred*]
lemmas *fminus-cancel*[*simp*] = *Diff-cancel*[*Transfer.transferred*]
lemmas *fminus-idemp*[*simp*] = *Diff-idemp*[*Transfer.transferred*]
lemmas *fminus-triv* = *Diff-triv*[*Transfer.transferred*]
lemmas *fempty-fminus*[*simp*] = *empty-Diff*[*Transfer.transferred*]
lemmas *fminus-fempty*[*simp*] = *Diff-empty*[*Transfer.transferred*]
lemmas *fminus-finsertffempty*[*simp*,*no-atp*] = *Diff-insert0*[*Transfer.transferred*]
lemmas *fminus-finsert* = *Diff-insert*[*Transfer.transferred*]
lemmas *fminus-finsert2* = *Diff-insert2*[*Transfer.transferred*]
lemmas *finsert-fminus-if* = *insert-Diff-if*[*Transfer.transferred*]
lemmas *finsert-fminus1*[*simp*] = *insert-Diff1*[*Transfer.transferred*]
lemmas *finsert-fminus-single*[*simp*] = *insert-Diff-single*[*Transfer.transferred*]
lemmas *finsert-fminus* = *insert-Diff*[*Transfer.transferred*]
lemmas *fminus-finsert-absorb* = *Diff-insert-absorb*[*Transfer.transferred*]
lemmas *fminus-disjoint*[*simp*] = *Diff-disjoint*[*Transfer.transferred*]
lemmas *fminus-partition* = *Diff-partition*[*Transfer.transferred*]
lemmas *double-fminus* = *double-diff*[*Transfer.transferred*]
lemmas *funion-fminus-cancel*[*simp*] = *Un-Diff-cancel*[*Transfer.transferred*]
lemmas *funion-fminus-cancel2*[*simp*] = *Un-Diff-cancel2*[*Transfer.transferred*]
lemmas *fminus-funion* = *Diff-Un*[*Transfer.transferred*]
lemmas *fminus-finter* = *Diff-Int*[*Transfer.transferred*]
lemmas *funion-fminus* = *Un-Diff*[*Transfer.transferred*]
lemmas *finter-fminus* = *Int-Diff*[*Transfer.transferred*]

lemmas *fminus-finter-distrib* = *Diff-Int-distrib*[*Transfer.transferred*]
lemmas *fminus-finter-distrib2* = *Diff-Int-distrib2*[*Transfer.transferred*]
lemmas *fUNIV-bool*[*no-atp*] = *UNIV-bool*[*Transfer.transferred*]
lemmas *fPow-fempty*[*simp*] = *Pow-empty*[*Transfer.transferred*]
lemmas *fPow-finsert* = *Pow-insert*[*Transfer.transferred*]
lemmas *funion-fPow-fsubset* = *Un-Pow-subset*[*Transfer.transferred*]
lemmas *fPow-finter-eq*[*simp*] = *Pow-Int-eq*[*Transfer.transferred*]
lemmas *fset-eq-fsubset* = *set-eq-subset*[*Transfer.transferred*]
lemmas *fsubset-iff*[*no-atp*] = *subset-iff*[*Transfer.transferred*]
lemmas *fsubset-iff-pfsubset-eq* = *subset-iff-psubset-eq*[*Transfer.transferred*]
lemmas *all-not-fin-conv*[*simp*] = *all-not-in-conv*[*Transfer.transferred*]
lemmas *ex-fin-conv* = *ex-in-conv*[*Transfer.transferred*]
lemmas *fimage-mono* = *image-mono*[*Transfer.transferred*]
lemmas *fPow-mono* = *Pow-mono*[*Transfer.transferred*]
lemmas *finsert-mono* = *insert-mono*[*Transfer.transferred*]
lemmas *funion-mono* = *Un-mono*[*Transfer.transferred*]
lemmas *finter-mono* = *Int-mono*[*Transfer.transferred*]
lemmas *fminus-mono* = *Diff-mono*[*Transfer.transferred*]
lemmas *fin-mono* = *in-mono*[*Transfer.transferred*]
lemmas *fthe-felem-eq*[*simp*] = *the-elem-eq*[*Transfer.transferred*]
lemmas *fLeast-mono* = *Least-mono*[*Transfer.transferred*]
lemmas *fbind-fbind* = *bind-bind*[*Transfer.transferred*]
lemmas *fempty-fbind*[*simp*] = *empty-bind*[*Transfer.transferred*]
lemmas *nonfempty-fbind-const* = *nonempty-bind-const*[*Transfer.transferred*]
lemmas *fbind-const* = *bind-const*[*Transfer.transferred*]
lemmas *ffmember-filter*[*simp*] = *member-filter*[*Transfer.transferred*]
lemmas *fequalityI* = *equalityI*[*Transfer.transferred*]
lemmas *fset-of-list-simps*[*simp*] = *set-simps*[*Transfer.transferred*]
lemmas *fset-of-list-append*[*simp*] = *set-append*[*Transfer.transferred*]
lemmas *fset-of-list-rev*[*simp*] = *set-rev*[*Transfer.transferred*]
lemmas *fset-of-list-map*[*simp*] = *set-map*[*Transfer.transferred*]

32.5 Additional lemmas

32.5.1 *ffUnion*

lemmas *ffUnion-funion-distrib*[*simp*] = *Union-Un-distrib*[*Transfer.transferred*]

32.5.2 *fbind*

lemma *fbind-cong*[*fundef-cong*]: $A = B \Longrightarrow (\bigwedge x.\ x\ |\in|\ B \Longrightarrow f\ x = g\ x) \Longrightarrow$ *fbind* $A\ f =$ *fbind* $B\ g$
⟨*proof*⟩

32.5.3 *fsingleton*

lemmas *fsingletonE* = *fsingletonD* [*elim-format*]

32.5.4 *femepty*

lemma *fempty-ffilter*[*simp*]: *ffilter* (λ-. *False*) *A* = {||}
⟨*proof*⟩

lemma *femptyE* [*elim*!]: *a* |∈| {||} ⟹ *P*
⟨*proof*⟩

32.5.5 *fset*

lemmas *fset-simps*[*simp*] = *bot-fset.rep-eq finsert.rep-eq*

lemma *finite-fset* [*simp*]:
shows *finite* (*fset S*)
⟨*proof*⟩

lemmas *fset-cong* = *fset-inject*

lemma *filter-fset* [*simp*]:
shows *fset* (*ffilter P xs*) = *Collect P* ∩ *fset xs*
⟨*proof*⟩

lemma *notin-fset*: *x* |∉| *S* ⟷ *x* ∉ *fset S* ⟨*proof*⟩

lemmas *inter-fset*[*simp*] = *inf-fset.rep-eq*

lemmas *union-fset*[*simp*] = *sup-fset.rep-eq*

lemmas *minus-fset*[*simp*] = *minus-fset.rep-eq*

32.5.6 *ffilter*

lemma *subset-ffilter*:
ffilter P A |⊆| *ffilter Q A* = (∀ *x*. *x* |∈| *A* ⟶ *P x* ⟶ *Q x*)
⟨*proof*⟩

lemma *eq-ffilter*:
(*ffilter P A* = *ffilter Q A*) = (∀*x*. *x* |∈| *A* ⟶ *P x* = *Q x*)
⟨*proof*⟩

lemma *pfsubset-ffilter*:
(⋀*x*. *x* |∈| *A* ⟹ *P x* ⟹ *Q x*) ⟹ (*x* |∈| *A* & ¬ *P x* & *Q x*) ⟹
ffilter P A |⊂| *ffilter Q A*
⟨*proof*⟩

32.5.7 *fset-of-list*

lemma *fset-of-list-filter*[*simp*]:
fset-of-list (*filter P xs*) = *ffilter P* (*fset-of-list xs*)

⟨*proof*⟩

lemma *fset-of-list-subset*[*intro*]:
set xs ⊆ *set ys* ⟹ *fset-of-list xs* |⊆| *fset-of-list ys*
⟨*proof*⟩

lemma *fset-of-list-elem*: (*x* |∈| *fset-of-list xs*) ⟷ (*x* ∈ *set xs*)
⟨*proof*⟩

32.5.8 *finsert*

lemma *set-finsert*:
assumes *x* |∈| *A*
obtains *B* **where** *A* = *finsert x B* **and** *x* |∉| *B*
⟨*proof*⟩

lemma *mk-disjoint-finsert*: *a* |∈| *A* ⟹ ∃ *B*. *A* = *finsert a B* ∧ *a* |∉| *B*
⟨*proof*⟩

lemma *finsert-eq-iff*:
assumes *a* |∉| *A* **and** *b* |∉| *B*
shows (*finsert a A* = *finsert b B*) =
(*if a* = *b then A* = *B else* ∃ *C*. *A* = *finsert b C* ∧ *b* |∉| *C* ∧ *B* = *finsert a C* ∧ *a* |∉| *C*)
⟨*proof*⟩

32.5.9 *fimage*

lemma *subset-fimage-iff*: (*B* |⊆| *f*|‘|*A*) = (∃ *AA*. *AA* |⊆| *A* ∧ *B* = *f*|‘|*AA*)
⟨*proof*⟩

32.5.10 bounded quantification

lemma *bex-simps* [*simp*, *no-atp*]:
⋀*A P Q*. *fBex A* (λ*x*. *P x* ∧ *Q*) = (*fBex A P* ∧ *Q*)
⋀*A P Q*. *fBex A* (λ*x*. *P* ∧ *Q x*) = (*P* ∧ *fBex A Q*)
⋀*P*. *fBex* {||} *P* = *False*
⋀*a B P*. *fBex* (*finsert a B*) *P* = (*P a* ∨ *fBex B P*)
⋀*A P f*. *fBex* (*f* |‘| *A*) *P* = *fBex A* (λ*x*. *P* (*f x*))
⋀*A P*. (¬ *fBex A P*) = *fBall A* (λ*x*. ¬ *P x*)
⟨*proof*⟩

lemma *ball-simps* [*simp*, *no-atp*]:
⋀*A P Q*. *fBall A* (λ*x*. *P x* ∨ *Q*) = (*fBall A P* ∨ *Q*)
⋀*A P Q*. *fBall A* (λ*x*. *P* ∨ *Q x*) = (*P* ∨ *fBall A Q*)
⋀*A P Q*. *fBall A* (λ*x*. *P* ⟶ *Q x*) = (*P* ⟶ *fBall A Q*)
⋀*A P Q*. *fBall A* (λ*x*. *P x* ⟶ *Q*) = (*fBex A P* ⟶ *Q*)
⋀*P*. *fBall* {||} *P* = *True*
⋀*a B P*. *fBall* (*finsert a B*) *P* = (*P a* ∧ *fBall B P*)
⋀*A P f*. *fBall* (*f* |‘| *A*) *P* = *fBall A* (λ*x*. *P* (*f x*))

$\bigwedge A\ P.\ (\neg\ fBall\ A\ P) = fBex\ A\ (\lambda x.\ \neg\ P\ x)$
⟨*proof*⟩

lemma *atomize-fBall*:
$(\bigwedge x.\ x\ |\in|\ A ==> P\ x) == Trueprop\ (fBall\ A\ (\lambda x.\ P\ x))$
⟨*proof*⟩

lemma *fBall-mono*[*mono*]: $P \leq Q \Longrightarrow fBall\ S\ P \leq fBall\ S\ Q$
⟨*proof*⟩

end

32.5.11 *fcard*

lemma *fcard-fempty*:
$fcard\ \{||\} = 0$
⟨*proof*⟩

lemma *fcard-finsert-disjoint*:
$x\ |\notin|\ A \Longrightarrow fcard\ (finsert\ x\ A) = Suc\ (fcard\ A)$
⟨*proof*⟩

lemma *fcard-finsert-if*:
$fcard\ (finsert\ x\ A) = (if\ x\ |\in|\ A\ then\ fcard\ A\ else\ Suc\ (fcard\ A))$
⟨*proof*⟩

lemma *fcard-0-eq* [*simp*, *no-atp*]:
$fcard\ A = 0 \longleftrightarrow A = \{||\}$
⟨*proof*⟩

lemma *fcard-Suc-fminus1*:
$x\ |\in|\ A \Longrightarrow Suc\ (fcard\ (A\ |-|\ \{|x|\})) = fcard\ A$
⟨*proof*⟩

lemma *fcard-fminus-fsingleton*:
$x\ |\in|\ A \Longrightarrow fcard\ (A\ |-|\ \{|x|\}) = fcard\ A - 1$
⟨*proof*⟩

lemma *fcard-fminus-fsingleton-if*:
$fcard\ (A\ |-|\ \{|x|\}) = (if\ x\ |\in|\ A\ then\ fcard\ A - 1\ else\ fcard\ A)$
⟨*proof*⟩

lemma *fcard-fminus-finsert*[*simp*]:
assumes $a\ |\in|\ A$ **and** $a\ |\notin|\ B$
shows $fcard\ (A\ |-|\ finsert\ a\ B) = fcard\ (A\ |-|\ B) - 1$
⟨*proof*⟩

lemma *fcard-finsert*: $fcard\ (finsert\ x\ A) = Suc\ (fcard\ (A\ |-|\ \{|x|\}))$

⟨*proof*⟩

lemma *fcard-finsert-le*: *fcard A* ≤ *fcard* (*finsert x A*)
⟨*proof*⟩

lemma *fcard-mono*:
A |⊆| *B* ⟹ *fcard A* ≤ *fcard B*
⟨*proof*⟩

lemma *fcard-seteq*: *A* |⊆| *B* ⟹ *fcard B* ≤ *fcard A* ⟹ *A* = *B*
⟨*proof*⟩

lemma *pfsubset-fcard-mono*: *A* |⊂| *B* ⟹ *fcard A* < *fcard B*
⟨*proof*⟩

lemma *fcard-funion-finter*:
fcard A + *fcard B* = *fcard* (*A* |∪| *B*) + *fcard* (*A* |∩| *B*)
⟨*proof*⟩

lemma *fcard-funion-disjoint*:
A |∩| *B* = {||} ⟹ *fcard* (*A* |∪| *B*) = *fcard A* + *fcard B*
⟨*proof*⟩

lemma *fcard-funion-fsubset*:
B |⊆| *A* ⟹ *fcard* (*A* |−| *B*) = *fcard A* − *fcard B*
⟨*proof*⟩

lemma *diff-fcard-le-fcard-fminus*:
fcard A − *fcard B* ≤ *fcard*(*A* |−| *B*)
⟨*proof*⟩

lemma *fcard-fminus1-less*: *x* |∈| *A* ⟹ *fcard* (*A* |−| {|*x*|}) < *fcard A*
⟨*proof*⟩

lemma *fcard-fminus2-less*:
x |∈| *A* ⟹ *y* |∈| *A* ⟹ *fcard* (*A* |−| {|*x*|} |−| {|*y*|}) < *fcard A*
⟨*proof*⟩

lemma *fcard-fminus1-le*: *fcard* (*A* |−| {|*x*|}) ≤ *fcard A*
⟨*proof*⟩

lemma *fcard-pfsubset*: *A* |⊆| *B* ⟹ *fcard A* < *fcard B* ⟹ *A* < *B*
⟨*proof*⟩

32.5.12 *ffold*

context *comp-fun-commute*
begin
lemmas *ffold-empty*[*simp*] = *fold-empty*[*Transfer.transferred*]

lemma *ffold-finsert* [*simp*]:
assumes $x \mid\notin\mid A$
shows $\mathit{ffold}\ f\ z\ (\mathit{finsert}\ x\ A) = f\ x\ (\mathit{ffold}\ f\ z\ A)$
⟨*proof*⟩

lemma *ffold-fun-left-comm*:
$f\ x\ (\mathit{ffold}\ f\ z\ A) = \mathit{ffold}\ f\ (f\ x\ z)\ A$
⟨*proof*⟩

lemma *ffold-finsert2*:
$x \mid\notin\mid A \Longrightarrow \mathit{ffold}\ f\ z\ (\mathit{finsert}\ x\ A) = \mathit{ffold}\ f\ (f\ x\ z)\ A$
⟨*proof*⟩

lemma *ffold-rec*:
assumes $x \mid\in\mid A$
shows $\mathit{ffold}\ f\ z\ A = f\ x\ (\mathit{ffold}\ f\ z\ (A \mid-\mid \{\mid x \mid\}))$
⟨*proof*⟩

lemma *ffold-finsert-fremove*:
$\mathit{ffold}\ f\ z\ (\mathit{finsert}\ x\ A) = f\ x\ (\mathit{ffold}\ f\ z\ (A \mid-\mid \{\mid x \mid\}))$
⟨*proof*⟩
end

lemma *ffold-fimage*:
assumes *inj-on g* (*fset A*)
shows $\mathit{ffold}\ f\ z\ (g \mid`\mid A) = \mathit{ffold}\ (f \circ g)\ z\ A$
⟨*proof*⟩

lemma *ffold-cong*:
assumes *comp-fun-commute f comp-fun-commute g*
$\bigwedge x.\ x \mid\in\mid A \Longrightarrow f\ x = g\ x$
and $s = t$ **and** $A = B$
shows $\mathit{ffold}\ f\ s\ A = \mathit{ffold}\ g\ t\ B$
⟨*proof*⟩

context *comp-fun-idem*
begin

lemma *ffold-finsert-idem*:
$\mathit{ffold}\ f\ z\ (\mathit{finsert}\ x\ A) = f\ x\ (\mathit{ffold}\ f\ z\ A)$
⟨*proof*⟩

declare *ffold-finsert* [*simp del*] *ffold-finsert-idem* [*simp*]

lemma *ffold-finsert-idem2*:
$\mathit{ffold}\ f\ z\ (\mathit{finsert}\ x\ A) = \mathit{ffold}\ f\ (f\ x\ z)\ A$
⟨*proof*⟩

end

32.5.13 Group operations

locale *comm-monoid-fset* = *comm-monoid*
begin

sublocale *set*: *comm-monoid-set* ⟨*proof*⟩

lift-definition *F* :: ('*b* ⇒ '*a*) ⇒ '*b fset* ⇒ '*a* **is** *set.F* ⟨*proof*⟩

lemmas *cong*[*fundef-cong*] = *set.cong*[*Transfer.transferred*]

lemma *strong-cong*[*cong*]:
assumes *A* = *B* ⋀*x*. *x* |∈| *B* =*simp*=> *g x* = *h x*
shows *F g A* = *F h B*
⟨*proof*⟩

end

context *comm-monoid-add* **begin**

sublocale *fsum*: *comm-monoid-fset plus 0*
defines *fsum* = *fsum.F*
rewrites *comm-monoid-set.F plus 0* = *sum*
⟨*proof*⟩

end

32.5.14 Semilattice operations

locale *semilattice-fset* = *semilattice*
begin

sublocale *set*: *semilattice-set* ⟨*proof*⟩

lift-definition *F* :: '*a fset* ⇒ '*a* **is** *set.F* ⟨*proof*⟩

lemma *eq-fold*: *F* (*finsert x A*) = *ffold f x A*
⟨*proof*⟩

lemma *singleton* [*simp*]: *F* {|*x*|} = *x*
⟨*proof*⟩

lemma *insert-not-elem*: *x* |∉| *A* ⟹ *A* ≠ {||} ⟹ *F* (*finsert x A*) = *x* ∗ *F A*
⟨*proof*⟩

lemma *in-idem*: *x* |∈| *A* ⟹ *x* ∗ *F A* = *F A*
⟨*proof*⟩

lemma *insert* [*simp*]: $A \neq \{||\} \Longrightarrow F\ (\mathit{finsert}\ x\ A) = x * F\ A$
⟨*proof*⟩

end

locale *semilattice-order-fset* = *binary?*: *semilattice-order* + *semilattice-fset*
begin

end

context *linorder* **begin**

sublocale *fMin*: *semilattice-order-fset min less-eq less*
defines *fMin* = *fMin.F*
rewrites *semilattice-set.F min* = *Min*
⟨*proof*⟩

sublocale *fMax*: *semilattice-order-fset max greater-eq greater*
defines *fMax* = *fMax.F*
rewrites *semilattice-set.F max* = *Max*
⟨*proof*⟩

end

lemma *mono-fMax-commute*: *mono f* $\Longrightarrow A \neq \{||\} \Longrightarrow f\ (\mathit{fMax}\ A) = \mathit{fMax}\ (f\ |`|\ A)$
⟨*proof*⟩

lemma *mono-fMin-commute*: *mono f* $\Longrightarrow A \neq \{||\} \Longrightarrow f\ (\mathit{fMin}\ A) = \mathit{fMin}\ (f\ |`|\ A)$
⟨*proof*⟩

lemma *fMax-in*[*simp*]: $A \neq \{||\} \Longrightarrow \mathit{fMax}\ A\ |\in|\ A$
⟨*proof*⟩

lemma *fMin-in*[*simp*]: $A \neq \{||\} \Longrightarrow \mathit{fMin}\ A\ |\in|\ A$
⟨*proof*⟩

lemma *fMax-ge*[*simp*]: $x\ |\in|\ A \Longrightarrow x \leq \mathit{fMax}\ A$
⟨*proof*⟩

lemma *fMin-le*[*simp*]: $x\ |\in|\ A \Longrightarrow \mathit{fMin}\ A \leq x$
⟨*proof*⟩

lemma *fMax-eqI*: $(\bigwedge y.\ y\ |\in|\ A \Longrightarrow y \leq x) \Longrightarrow x\ |\in|\ A \Longrightarrow \mathit{fMax}\ A = x$
⟨*proof*⟩

lemma *fMin-eqI*: $(\bigwedge y.\ y\ |\in|\ A \Longrightarrow x \leq y) \Longrightarrow x\ |\in|\ A \Longrightarrow \mathit{fMin}\ A = x$

⟨*proof*⟩

lemma *fMax-finsert*[*simp*]: *fMax* (*finsert x A*) = (*if A* = {||} *then x else max x* (*fMax A*))
⟨*proof*⟩

lemma *fMin-finsert*[*simp*]: *fMin* (*finsert x A*) = (*if A* = {||} *then x else min x* (*fMin A*))
⟨*proof*⟩

context *linorder* **begin**

lemma *fset-linorder-max-induct*[*case-names fempty finsert*]:
assumes *P* {||}
and ⋀*x S*. ⟦∀ *y*. *y* |∈| *S* ⟶ *y* < *x*; *P S*⟧ ⟹ *P* (*finsert x S*)
shows *P S*
⟨*proof*⟩

lemma *fset-linorder-min-induct*[*case-names fempty finsert*]:
assumes *P* {||}
and ⋀*x S*. ⟦∀ *y*. *y* |∈| *S* ⟶ *y* > *x*; *P S*⟧ ⟹ *P* (*finsert x S*)
shows *P S*
⟨*proof*⟩

end

32.6 Choice in fsets

lemma *fset-choice*:
assumes ∀ *x*. *x* |∈| *A* ⟶ (∃ *y*. *P x y*)
shows ∃ *f*. ∀ *x*. *x* |∈| *A* ⟶ *P x* (*f x*)
⟨*proof*⟩

32.7 Induction and Cases rules for fsets

lemma *fset-exhaust* [*case-names empty insert*, *cases type*: *fset*]:
assumes *fempty-case*: *S* = {||} ⟹ *P*
and *finsert-case*: ⋀*x S'*. *S* = *finsert x S'* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *fset-induct* [*case-names empty insert*]:
assumes *fempty-case*: *P* {||}
and *finsert-case*: ⋀*x S*. *P S* ⟹ *P* (*finsert x S*)
shows *P S*
⟨*proof*⟩

lemma *fset-induct-stronger* [*case-names empty insert*, *induct type*: *fset*]:
assumes *empty-fset-case*: *P* {||}
and *insert-fset-case*: ⋀*x S*. ⟦*x* |∉| *S*; *P S*⟧ ⟹ *P* (*finsert x S*)

shows $P\ S$
⟨*proof*⟩

lemma *fset-card-induct*:
assumes *empty-fset-case*: $P\ \{||\}$
and *card-fset-Suc-case*: $\bigwedge S\ T.\ Suc\ (fcard\ S) = (fcard\ T) \Longrightarrow P\ S \Longrightarrow P\ T$
shows $P\ S$
⟨*proof*⟩

lemma *fset-strong-cases*:
obtains $xs = \{||\}$
| $ys\ x$ **where** $x\ |\notin|\ ys$ **and** $xs = finsert\ x\ ys$
⟨*proof*⟩

lemma *fset-induct2*:
$P\ \{||\}\ \{||\} \Longrightarrow$
$(\bigwedge x\ xs.\ x\ |\notin|\ xs \Longrightarrow P\ (finsert\ x\ xs)\ \{||\}) \Longrightarrow$
$(\bigwedge y\ ys.\ y\ |\notin|\ ys \Longrightarrow P\ \{||\}\ (finsert\ y\ ys)) \Longrightarrow$
$(\bigwedge x\ xs\ y\ ys.\ [\![P\ xs\ ys;\ x\ |\notin|\ xs;\ y\ |\notin|\ ys]\!] \Longrightarrow P\ (finsert\ x\ xs)\ (finsert\ y\ ys)) \Longrightarrow$
$P\ xsa\ ysa$
⟨*proof*⟩

32.8 Setup for Lifting/Transfer

32.8.1 Relator and predicator properties

lift-definition *rel-fset* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ fset \Rightarrow 'b\ fset \Rightarrow bool$ **is** *rel-set*
parametric *rel-set-transfer* ⟨*proof*⟩

lemma *rel-fset-alt-def*: $rel\text{-}fset\ R = (\lambda A\ B.\ (\forall x.\exists y.\ x|\in|A \longrightarrow y|\in|B \wedge R\ x\ y)$
$\wedge\ (\forall y.\ \exists x.\ y|\in|B \longrightarrow x|\in|A \wedge R\ x\ y))$
⟨*proof*⟩

lemma *finite-rel-set*:
assumes *fin*: *finite X finite Z*
assumes *R-S*: *rel-set* $(R\ OO\ S)\ X\ Z$
shows $\exists Y.\ finite\ Y \wedge rel\text{-}set\ R\ X\ Y \wedge rel\text{-}set\ S\ Y\ Z$
⟨*proof*⟩

32.8.2 Transfer rules for the Transfer package

Unconditional transfer rules

context includes *lifting-syntax*
begin

lemmas *fempty-transfer* [*transfer-rule*] = *empty-transfer*[*Transfer.transferred*]

lemma *finsert-transfer* [*transfer-rule*]:
$(A$ ===> *rel-fset A* ===> *rel-fset A*$)$ *finsert finsert*

⟨*proof*⟩

lemma *funion-transfer* [*transfer-rule*]:
(*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) *funion funion*
⟨*proof*⟩

lemma *ffUnion-transfer* [*transfer-rule*]:
(*rel-fset* (*rel-fset A*) ===> *rel-fset A*) *ffUnion ffUnion*
⟨*proof*⟩

lemma *fimage-transfer* [*transfer-rule*]:
((*A* ===> *B*) ===> *rel-fset A* ===> *rel-fset B*) *fimage fimage*
⟨*proof*⟩

lemma *fBall-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> *op* =) ===> *op* =) *fBall fBall*
⟨*proof*⟩

lemma *fBex-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> *op* =) ===> *op* =) *fBex fBex*
⟨*proof*⟩

lemma *fPow-transfer* [*transfer-rule*]:
(*rel-fset A* ===> *rel-fset* (*rel-fset A*)) *fPow fPow*
⟨*proof*⟩

lemma *rel-fset-transfer* [*transfer-rule*]:
((*A* ===> *B* ===> *op* =) ===> *rel-fset A* ===> *rel-fset B* ===> *op* =)
rel-fset rel-fset
⟨*proof*⟩

lemma *bind-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> *rel-fset B*) ===> *rel-fset B*) *fbind fbind*
⟨*proof*⟩

Rules requiring bi-unique, bi-total or right-total relations

lemma *fmember-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*A* ===> *rel-fset A* ===> *op* =) (*op* |∈|) (*op* |∈|)
⟨*proof*⟩

lemma *finter-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) *finter finter*
⟨*proof*⟩

lemma *fminus-transfer* [*transfer-rule*]:
assumes *bi-unique A*

shows (*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) (*op* |−|) (*op* |−|)
⟨*proof*⟩

lemma *fsubset-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> *op* =) (*op* |⊆|) (*op* |⊆|)
⟨*proof*⟩

lemma *fSup-transfer* [*transfer-rule*]:
bi-unique A ⟹ (*rel-set* (*rel-fset A*) ===> *rel-fset A*) *Sup Sup*
⟨*proof*⟩

lemma *fInf-transfer* [*transfer-rule*]:
assumes *bi-unique A* **and** *bi-total A*
shows (*rel-set* (*rel-fset A*) ===> *rel-fset A*) *Inf Inf*
⟨*proof*⟩

lemma *ffilter-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows ((*A* ===> *op*=) ===> *rel-fset A* ===> *rel-fset A*) *ffilter ffilter*
⟨*proof*⟩

lemma *card-transfer* [*transfer-rule*]:
bi-unique A ⟹ (*rel-fset A* ===> *op* =) *fcard fcard*
⟨*proof*⟩

end

lifting-update *fset.lifting*
lifting-forget *fset.lifting*

32.9 BNF setup

context
includes *fset.lifting*
begin

lemma *rel-fset-alt*:
rel-fset R a b ⟷ (∀ *t* ∈ *fset a*. ∃ *u* ∈ *fset b*. *R t u*) ∧ (∀ *t* ∈ *fset b*. ∃ *u* ∈ *fset a*. *R u t*)
⟨*proof*⟩

lemma *fset-to-fset*: *finite A* ⟹ *fset* (*the-inv fset A*) = *A*
⟨*proof*⟩

lemma *rel-fset-aux*:
(∀ *t* ∈ *fset a*. ∃ *u* ∈ *fset b*. *R t u*) ∧ (∀ *u* ∈ *fset b*. ∃ *t* ∈ *fset a*. *R t u*) ⟷

((*BNF-Def.Grp* {*a*. *fset a* ⊆ {(*a*, *b*). *R a b*}} (*fimage fst*))$^{-1-1}$ *OO*
BNF-Def.Grp {*a*. *fset a* ⊆ {(*a*, *b*). *R a b*}} (*fimage snd*)) *a b* (**is** *?L* = *?R*)
⟨*proof*⟩

bnf *'a fset*
map: *fimage*
sets: *fset*
bd: *natLeq*
wits: {||}
rel: *rel-fset*
⟨*proof*⟩

lemma *rel-fset-fset*: *rel-set* χ (*fset A1*) (*fset A2*) = *rel-fset* χ *A1 A2*
⟨*proof*⟩

end

lemmas [*simp*] = *fset.map-comp fset.map-id fset.set-map*

32.10 Size setup

context includes *fset.lifting* **begin**
lift-definition *size-fset* :: ('*a* ⇒ *nat*) ⇒ *'a fset* ⇒ *nat* **is** λ*f*. *sum* (*Suc* ∘ *f*) ⟨*proof*⟩
end

instantiation *fset* :: (*type*) *size* **begin**
definition *size-fset* **where**
size-fset-overloaded-def: *size-fset* = *FSet.size-fset* (λ-. *0*)
instance ⟨*proof*⟩
end

lemmas *size-fset-simps*[*simp*] =
size-fset-def[*THEN meta-eq-to-obj-eq*, *THEN fun-cong*, *THEN fun-cong*,
unfolded map-fun-def comp-def id-apply]

lemmas *size-fset-overloaded-simps*[*simp*] =
size-fset-simps[*of* λ-. *0*, *unfolded add-0-left add-0-right*,
folded size-fset-overloaded-def]

lemma *fset-size-o-map*: *inj f* ⟹ *size-fset g* ∘ *fimage f* = *size-fset* (*g* ∘ *f*)
⟨*proof*⟩
including *fset.lifting* ⟨*proof*⟩

⟨*ML*⟩

lifting-update *fset.lifting*
lifting-forget *fset.lifting*

32.11 Advanced relator customization

lemma *rel-set-rel-sum*[*simp*]:
rel-set (*rel-sum* χ φ) *A1 A2* $\longleftrightarrow$
rel-set χ (*Inl* −‘ *A1*) (*Inl* −‘ *A2*) ∧ *rel-set* φ (*Inr* −‘ *A1*) (*Inr* −‘ *A2*)
(**is** *?L* $\longleftrightarrow$ *?Rl* ∧ *?Rr*)
⟨*proof*⟩

32.11.1 Countability

lemma *exists-fset-of-list*: ∃ *xs*. *fset-of-list xs* = *S*
including *fset.lifting*
⟨*proof*⟩

lemma *fset-of-list-surj*[*simp*, *intro*]: *surj fset-of-list*
⟨*proof*⟩

instance *fset* :: (*countable*) *countable*
⟨*proof*⟩

32.12 Quickcheck setup

Setup adapted from sets.

notation *Quickcheck-Exhaustive.orelse* (**infixr** *orelse* 55)

definition (**in** *term-syntax*) [*code-unfold*]:
valterm-femptyset = *Code-Evaluation.valtermify* ({||} :: (′*a* :: *typerep*) *fset*)

definition (**in** *term-syntax*) [*code-unfold*]:
valtermify-finsert x s = *Code-Evaluation.valtermify finsert* {·} (*x* :: (′*a* :: *typerep* ∗ -)) {·} *s*

instantiation *fset* :: (*exhaustive*) *exhaustive*
begin

fun *exhaustive-fset* **where**
exhaustive-fset f i = (*if i* = *0 then None else* (*f* {||} *orelse exhaustive-fset* (λ*A*. *f A orelse Quickcheck-Exhaustive.exhaustive* (λ*x*. *if x* |∈| *A then None else f* (*finsert x A*)) (*i* − *1*)) (*i* − *1*)))

instance ⟨*proof*⟩

end

instantiation *fset* :: (*full-exhaustive*) *full-exhaustive*
begin

fun *full-exhaustive-fset* **where**
full-exhaustive-fset f i = (*if i* = *0 then None else* (*f valterm-femptyset orelse full-exhaustive-fset* (λ*A*. *f A orelse Quickcheck-Exhaustive.full-exhaustive* (λ*x*. *if*

fst x |∈| fst A then None else f (valtermify-finsert x A)) (i − 1)) (i − 1)))

instance ⟨*proof*⟩

end

no-notation *Quickcheck-Exhaustive.orelse* (**infixr** *orelse 55*)

notation *scomp* (**infixl** ∘→ *60*)

instantiation *fset* :: (*random*) *random*
begin

fun *random-aux-fset* :: *natural* ⇒ *natural* ⇒ *natural* × *natural* ⇒ ('*a fset* × (*unit* ⇒ *term*)) × *natural* × *natural* **where**
random-aux-fset 0 j = Quickcheck-Random.collapse (Random.select-weight [(1, Pair valterm-femptyset)]) |
random-aux-fset (Code-Numeral.Suc i) j =
Quickcheck-Random.collapse (Random.select-weight
[(1, Pair valterm-femptyset),
(Code-Numeral.Suc i,
Quickcheck-Random.random j ∘→ (λx. random-aux-fset i j ∘→ (λs. Pair (valtermify-finsert x s))))])

lemma [*code*]:
random-aux-fset i j =
Quickcheck-Random.collapse (Random.select-weight [(1, Pair valterm-femptyset),
(i, Quickcheck-Random.random j ∘→ (λx. random-aux-fset (i − 1) j ∘→ (λs. Pair (valtermify-finsert x s))))])
⟨*proof*⟩

definition *random-fset i = random-aux-fset i i*

instance ⟨*proof*⟩

end

no-notation *scomp* (**infixl** ∘→ *60*)

end

33 Type of finite maps defined as a subtype of maps

theory *Finite-Map*
imports *FSet AList*
begin

33.1 Auxiliary constants and lemmas over *map*

context includes *lifting-syntax* **begin**

abbreviation *rel-map* :: $('b \Rightarrow 'c \Rightarrow bool) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'c) \Rightarrow bool$ **where**
rel-map f $\equiv$ *op* = ===> *rel-option f*

lemma *map-empty-transfer*[*transfer-rule*]: *rel-map A Map.empty Map.empty*
⟨*proof*⟩

lemma *ran-transfer*[*transfer-rule*]: (*rel-map A* ===> *rel-set A*) *ran ran*
⟨*proof*⟩

lemma *ran-alt-def*: *ran m* = (*the* ∘ *m*) ‘ *dom m*
⟨*proof*⟩

lemma *dom-transfer*[*transfer-rule*]: (*rel-map A* ===> *op* =) *dom dom*
⟨*proof*⟩

definition *map-upd* :: $'a \Rightarrow 'b \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ **where**
map-upd k v m = *m*($k \mapsto v$)

lemma *map-upd-transfer*[*transfer-rule*]:
(*op* = ===> *A* ===> *rel-map A* ===> *rel-map A*) *map-upd map-upd*
⟨*proof*⟩

definition *map-filter* :: $('a \Rightarrow bool) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ **where**
map-filter P m = (λx. *if P x then m x else None*)

lemma *map-filter-map-of*[*simp*]: *map-filter P* (*map-of m*) = *map-of* [(*k*, -) ← *m*. *P k*]
⟨*proof*⟩

lemma *map-filter-transfer*[*transfer-rule*]:
(*op* = ===> *rel-map A* ===> *rel-map A*) *map-filter map-filter*
⟨*proof*⟩

lemma *map-filter-finite*[*intro*]:
assumes *finite* (*dom m*)
shows *finite* (*dom* (*map-filter P m*))
⟨*proof*⟩

definition *map-drop* :: $'a \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ **where**
map-drop a = *map-filter* ($\lambda a'$. $a' \neq a$)

lemma *map-drop-transfer*[*transfer-rule*]:
(*op* = ===> *rel-map A* ===> *rel-map A*) *map-drop map-drop*
⟨*proof*⟩

definition *map-drop-set* :: *'a set* $\Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ **where**

map-drop-set A = map-filter (λ*a*. *a* ∉ *A*)

lemma *map-drop-set-transfer*[*transfer-rule*]:
(*op* = ===> *rel-map A* ===> *rel-map A*) *map-drop-set map-drop-set*
⟨*proof*⟩

definition *map-restrict-set* :: '*a set* ⇒ ('*a* ⇀ '*b*) ⇒ ('*a* ⇀ '*b*) **where**
map-restrict-set A = map-filter (λ*a*. *a* ∈ *A*)

lemma *map-restrict-set-transfer*[*transfer-rule*]:
(*op* = ===> *rel-map A* ===> *rel-map A*) *map-restrict-set map-restrict-set*
⟨*proof*⟩

lemma *map-add-transfer*[*transfer-rule*]:
(*rel-map A* ===> *rel-map A* ===> *rel-map A*) *op* ++ *op* ++
⟨*proof*⟩

definition *map-pred* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*a* ⇀ '*b*) ⇒ *bool* **where**
map-pred P m ⟷ (∀ *x*. *case m x of None* ⇒ *True* | *Some y* ⇒ *P x y*)

lemma *map-pred-transfer*[*transfer-rule*]:
((*op* = ===> *A* ===> *op* =) ===> *rel-map A* ===> *op* =) *map-pred map-pred*
⟨*proof*⟩

definition *rel-map-on-set* :: '*a set* ⇒ ('*b* ⇒ '*c* ⇒ *bool*) ⇒ ('*a* ⇀ '*b*) ⇒ ('*a* ⇀ '*c*) ⇒ *bool* **where**
rel-map-on-set S P = eq-onp (λ*x*. *x* ∈ *S*) ===> *rel-option P*

lemma *map-of-transfer*[*transfer-rule*]:
includes *lifting-syntax*
shows (*list-all2* (*rel-prod op* = *A*) ===> *rel-map A*) *map-of map-of*
⟨*proof*⟩

definition *set-of-map* :: ('*a* ⇀ '*b*) ⇒ ('*a* × '*b*) *set* **where**
set-of-map m = {(*k*, *v*)|*k v*. *m k* = *Some v*}

lemma *set-of-map-alt-def*: *set-of-map m* = (λ*k*. (*k*, *the* (*m k*))) ‘ *dom m*
⟨*proof*⟩

lemma *set-of-map-finite*: *finite* (*dom m*) ⟹ *finite* (*set-of-map m*)
⟨*proof*⟩

lemma *set-of-map-inj*: *inj set-of-map*
⟨*proof*⟩

end

33.2 Abstract characterisation

typedef $('a, 'b)$ *fmap* $= \{m.$ *finite* $(dom\ m)\} :: ('a \rightharpoonup 'b)$ *set*
  morphisms *fmlookup Abs-fmap*
⟨*proof*⟩

setup-lifting *type-definition-fmap*

lemma *fmlookup-finite*[*intro*, *simp*]: *finite* $(dom\ (fmlookup\ m))$
⟨*proof*⟩

lemma *fmap-ext*:
  assumes $\bigwedge x.\ fmlookup\ m\ x = fmlookup\ n\ x$
  shows $m = n$
⟨*proof*⟩

33.3 Operations

context
  includes *fset.lifting*
begin

lift-definition *fmran* :: $('a, 'b)$ *fmap* $\Rightarrow 'b$ *fset*
  is *ran*
  parametric *ran-transfer*
⟨*proof*⟩

lemma *fmlookup-ran-iff*: $y \mid\in\mid fmran\ m \longleftrightarrow (\exists x.\ fmlookup\ m\ x = Some\ y)$
⟨*proof*⟩

lemma *fmranI*: $fmlookup\ m\ x = Some\ y \Longrightarrow y \mid\in\mid fmran\ m$ ⟨*proof*⟩

lemma *fmranE*[*elim*]:
  assumes $y \mid\in\mid fmran\ m$
  obtains x **where** $fmlookup\ m\ x = Some\ y$
⟨*proof*⟩

lift-definition *fmdom* :: $('a, 'b)$ *fmap* $\Rightarrow 'a$ *fset*
  is *dom*
  parametric *dom-transfer*
⟨*proof*⟩

lemma *fmlookup-dom-iff*: $x \mid\in\mid fmdom\ m \longleftrightarrow (\exists a.\ fmlookup\ m\ x = Some\ a)$
⟨*proof*⟩

lemma *fmdom-notI*: $fmlookup\ m\ x = None \Longrightarrow x \mid\notin\mid fmdom\ m$ ⟨*proof*⟩
lemma *fmdomI*: $fmlookup\ m\ x = Some\ y \Longrightarrow x \mid\in\mid fmdom\ m$ ⟨*proof*⟩
lemma *fmdom-notD*[*dest*]: $x \mid\notin\mid fmdom\ m \Longrightarrow fmlookup\ m\ x = None$ ⟨*proof*⟩

lemma *fmdomE*[*elim*]:

assumes x $|\in|$ *fmdom* m
obtains y **where** *fmlookup* m x = *Some* y
⟨*proof*⟩

lift-definition *fmdom'* :: ('*a*, '*b*) *fmap* ⇒ '*a set*
is *dom*
parametric *dom-transfer*
⟨*proof*⟩

lemma *fmlookup-dom'-iff*: $x \in$ *fmdom'* $m \longleftrightarrow (\exists\, a.$ *fmlookup* m x = *Some* $a)$
⟨*proof*⟩

lemma *fmdom'-notI*: *fmlookup* m x = *None* ⟹ $x \notin$ *fmdom'* m ⟨*proof*⟩
lemma *fmdom'I*: *fmlookup* m x = *Some* y ⟹ $x \in$ *fmdom'* m ⟨*proof*⟩
lemma *fmdom'-notD*[*dest*]: $x \notin$ *fmdom'* m ⟹ *fmlookup* m x = *None* ⟨*proof*⟩

lemma *fmdom'E*[*elim*]:
assumes $x \in$ *fmdom'* m
obtains x y **where** *fmlookup* m x = *Some* y
⟨*proof*⟩

lemma *fmdom'-alt-def*: *fmdom'* m = *fset* (*fmdom* m)
⟨*proof*⟩

lift-definition *fmempty* :: ('*a*, '*b*) *fmap*
is *Map.empty*
parametric *map-empty-transfer*
⟨*proof*⟩

lemma *fmempty-lookup*[*simp*]: *fmlookup fmempty* x = *None*
⟨*proof*⟩

lemma *fmdom-empty*[*simp*]: *fmdom fmempty* = {||} ⟨*proof*⟩
lemma *fmdom'-empty*[*simp*]: *fmdom'* *fmempty* = {} ⟨*proof*⟩
lemma *fmran-empty*[*simp*]: *fmran fmempty* = *fempty* ⟨*proof*⟩

lift-definition *fmupd* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *fmap* ⇒ ('*a*, '*b*) *fmap*
is *map-upd*
parametric *map-upd-transfer*
⟨*proof*⟩

lemma *fmupd-lookup*[*simp*]: *fmlookup* (*fmupd* a b m) a' = (*if* a = a' *then Some* b *else fmlookup* m a')
⟨*proof*⟩

lemma *fmdom-fmupd*[*simp*]: *fmdom* (*fmupd* a b m) = *finsert* a (*fmdom* m) ⟨*proof*⟩
lemma *fmdom'-fmupd*[*simp*]: *fmdom'* (*fmupd* a b m) = *insert* a (*fmdom'* m)
⟨*proof*⟩

lift-definition *fmfilter* :: $('a \Rightarrow bool) \Rightarrow ('a, 'b)\ fmap \Rightarrow ('a, 'b)\ fmap$
  is *map-filter*
  parametric *map-filter-transfer*
⟨*proof*⟩

lemma *fmdom-filter*[*simp*]: *fmdom* (*fmfilter P m*) = *ffilter P* (*fmdom m*)
⟨*proof*⟩

lemma *fmdom'-filter*[*simp*]: *fmdom'* (*fmfilter P m*) = *Set.filter P* (*fmdom' m*)
⟨*proof*⟩

lemma *fmlookup-filter*[*simp*]: *fmlookup* (*fmfilter P m*) *x* = (*if P x then fmlookup m x else None*)
⟨*proof*⟩

lemma *fmfilter-empty*[*simp*]: *fmfilter P fmempty* = *fmempty*
⟨*proof*⟩

lemma *fmfilter-true*[*simp*]:
  assumes $\bigwedge x\ y.$ *fmlookup m x* = *Some y* $\Longrightarrow$ *P x*
  shows *fmfilter P m* = *m*
⟨*proof*⟩

lemma *fmfilter-false*[*simp*]:
  assumes $\bigwedge x\ y.$ *fmlookup m x* = *Some y* $\Longrightarrow \neg$ *P x*
  shows *fmfilter P m* = *fmempty*
⟨*proof*⟩

lemma *fmfilter-comp*[*simp*]: *fmfilter P* (*fmfilter Q m*) = *fmfilter* ($\lambda x.$ *P x* $\wedge$ *Q x*) *m*
⟨*proof*⟩

lemma *fmfilter-comm*: *fmfilter P* (*fmfilter Q m*) = *fmfilter Q* (*fmfilter P m*)
⟨*proof*⟩

lemma *fmfilter-cong*[*cong*]:
  assumes $\bigwedge x\ y.$ *fmlookup m x* = *Some y* $\Longrightarrow$ *P x* = *Q x*
  shows *fmfilter P m* = *fmfilter Q m*
⟨*proof*⟩

lemma *fmfilter-cong'*[*fundef-cong*]:
  assumes $\bigwedge x.$ *x* $\in$ *fmdom' m* $\Longrightarrow$ *P x* = *Q x*
  shows *fmfilter P m* = *fmfilter Q m*
⟨*proof*⟩

lemma *fmfilter-upd*[*simp*]:
  fmfilter P (*fmupd x y m*) = (*if P x then fmupd x y* (*fmfilter P m*) *else fmfilter P m*)
⟨*proof*⟩

lift-definition *fmdrop* :: *'a* ⇒ (*'a*, *'b*) *fmap* ⇒ (*'a*, *'b*) *fmap*
  is *map-drop*
  parametric *map-drop-transfer*
⟨*proof*⟩

lemma *fmdrop-lookup*[*simp*]: *fmlookup* (*fmdrop a m*) *a* = *None*
⟨*proof*⟩

lift-definition *fmdrop-set* :: *'a set* ⇒ (*'a*, *'b*) *fmap* ⇒ (*'a*, *'b*) *fmap*
  is *map-drop-set*
  parametric *map-drop-set-transfer*
⟨*proof*⟩

lift-definition *fmdrop-fset* :: *'a fset* ⇒ (*'a*, *'b*) *fmap* ⇒ (*'a*, *'b*) *fmap*
  is *map-drop-set*
  parametric *map-drop-set-transfer*
⟨*proof*⟩

lift-definition *fmrestrict-set* :: *'a set* ⇒ (*'a*, *'b*) *fmap* ⇒ (*'a*, *'b*) *fmap*
  is *map-restrict-set*
  parametric *map-restrict-set-transfer*
⟨*proof*⟩

lift-definition *fmrestrict-fset* :: *'a fset* ⇒ (*'a*, *'b*) *fmap* ⇒ (*'a*, *'b*) *fmap*
  is *map-restrict-set*
  parametric *map-restrict-set-transfer*
⟨*proof*⟩

lemma *fmfilter-alt-defs*:
  fmdrop a = *fmfilter* (λ*a'*. *a'* ≠ *a*)
  fmdrop-set A = *fmfilter* (λ*a*. *a* ∉ *A*)
  fmdrop-fset B = *fmfilter* (λ*a*. *a* |∉| *B*)
  fmrestrict-set A = *fmfilter* (λ*a*. *a* ∈ *A*)
  fmrestrict-fset B = *fmfilter* (λ*a*. *a* |∈| *B*)
⟨*proof*⟩

lemma *fmdom-drop*[*simp*]: *fmdom* (*fmdrop a m*) = *fmdom m* − {|*a*|} ⟨*proof*⟩
lemma *fmdom'-drop*[*simp*]: *fmdom'* (*fmdrop a m*) = *fmdom' m* − {*a*} ⟨*proof*⟩
lemma *fmdom'-drop-set*[*simp*]: *fmdom'* (*fmdrop-set A m*) = *fmdom' m* − *A* ⟨*proof*⟩
lemma *fmdom-drop-fset*[*simp*]: *fmdom* (*fmdrop-fset A m*) = *fmdom m* − *A* ⟨*proof*⟩
lemma *fmdom'-restrict-set*: *fmdom'* (*fmrestrict-set A m*) ⊆ *A* ⟨*proof*⟩
lemma *fmdom-restrict-fset*: *fmdom* (*fmrestrict-fset A m*) |⊆| *A* ⟨*proof*⟩

lemma *fmdom'-drop-fset*[*simp*]: *fmdom'* (*fmdrop-fset A m*) = *fmdom' m* − *fset A*
⟨*proof*⟩

lemma *fmdom'-restrict-fset*: *fmdom'* (*fmrestrict-fset A m*) ⊆ *fset A*
⟨*proof*⟩

lemma *fmlookup-drop*[*simp*]:
fmlookup (*fmdrop a m*) *x* = (*if x* ≠ *a then fmlookup m x else None*)
⟨*proof*⟩

lemma *fmlookup-drop-set*[*simp*]:
fmlookup (*fmdrop-set A m*) *x* = (*if x* ∉ *A then fmlookup m x else None*)
⟨*proof*⟩

lemma *fmlookup-drop-fset*[*simp*]:
fmlookup (*fmdrop-fset A m*) *x* = (*if x* |∉| *A then fmlookup m x else None*)
⟨*proof*⟩

lemma *fmlookup-restrict-set*[*simp*]:
fmlookup (*fmrestrict-set A m*) *x* = (*if x* ∈ *A then fmlookup m x else None*)
⟨*proof*⟩

lemma *fmlookup-restrict-fset*[*simp*]:
fmlookup (*fmrestrict-fset A m*) *x* = (*if x* |∈| *A then fmlookup m x else None*)
⟨*proof*⟩

lemma *fmrestrict-set-dom*[*simp*]: *fmrestrict-set* (*fmdom′ m*) *m* = *m*
⟨*proof*⟩

lemma *fmrestrict-fset-dom*[*simp*]: *fmrestrict-fset* (*fmdom m*) *m* = *m*
⟨*proof*⟩

lemma *fmdrop-empty*[*simp*]: *fmdrop a fmempty* = *fmempty*
⟨*proof*⟩

lemma *fmdrop-set-empty*[*simp*]: *fmdrop-set A fmempty* = *fmempty*
⟨*proof*⟩

lemma *fmdrop-fset-empty*[*simp*]: *fmdrop-fset A fmempty* = *fmempty*
⟨*proof*⟩

lemma *fmrestrict-set-empty*[*simp*]: *fmrestrict-set A fmempty* = *fmempty*
⟨*proof*⟩

lemma *fmrestrict-fset-empty*[*simp*]: *fmrestrict-fset A fmempty* = *fmempty*
⟨*proof*⟩

lemma *fmdrop-set-null*[*simp*]: *fmdrop-set* {} *m* = *m*
⟨*proof*⟩

lemma *fmdrop-fset-null*[*simp*]: *fmdrop-fset* {|||} *m* = *m*
⟨*proof*⟩

lemma *fmdrop-set-single*[*simp*]: *fmdrop-set* {*a*} *m* = *fmdrop a m*

⟨*proof*⟩

lemma *fmdrop-fset-single*[*simp*]: *fmdrop-fset* {|*a*|} *m* = *fmdrop a m*
⟨*proof*⟩

lemma *fmrestrict-set-null*[*simp*]: *fmrestrict-set* {} *m* = *fmempty*
⟨*proof*⟩

lemma *fmrestrict-fset-null*[*simp*]: *fmrestrict-fset* {||} *m* = *fmempty*
⟨*proof*⟩

lemma *fmdrop-comm*: *fmdrop a* (*fmdrop b m*) = *fmdrop b* (*fmdrop a m*)
⟨*proof*⟩

lemma *fmdrop-set-insert*[*simp*]: *fmdrop-set* (*insert x S*) *m* = *fmdrop x* (*fmdrop-set S m*)
⟨*proof*⟩

lemma *fmdrop-fset-insert*[*simp*]: *fmdrop-fset* (*finsert x S*) *m* = *fmdrop x* (*fmdrop-fset S m*)
⟨*proof*⟩

lift-definition *fmadd* :: ('*a*, '*b*) *fmap* ⇒ ('*a*, '*b*) *fmap* ⇒ ('*a*, '*b*) *fmap* (**infixl** ++$_f$ *100*)
is *map-add*
parametric *map-add-transfer*
⟨*proof*⟩

lemma *fmlookup-add*[*simp*]:
fmlookup (*m* ++$_f$ *n*) *x* = (*if x* |∈| *fmdom n then fmlookup n x else fmlookup m x*)
⟨*proof*⟩

lemma *fmdom-add*[*simp*]: *fmdom* (*m* ++$_f$ *n*) = *fmdom m* |∪| *fmdom n* ⟨*proof*⟩
lemma *fmdom'-add*[*simp*]: *fmdom'* (*m* ++$_f$ *n*) = *fmdom' m* ∪ *fmdom' n* ⟨*proof*⟩

lemma *fmadd-drop-left-dom*: *fmdrop-fset* (*fmdom n*) *m* ++$_f$ *n* = *m* ++$_f$ *n*
⟨*proof*⟩

lemma *fmadd-restrict-right-dom*: *fmrestrict-fset* (*fmdom n*) (*m* ++$_f$ *n*) = *n*
⟨*proof*⟩

lemma *fmfilter-add-distrib*[*simp*]: *fmfilter P* (*m* ++$_f$ *n*) = *fmfilter P m* ++$_f$ *fmfilter P n*
⟨*proof*⟩

lemma *fmdrop-add-distrib*[*simp*]: *fmdrop a* (*m* ++$_f$ *n*) = *fmdrop a m* ++$_f$ *fmdrop a n*
⟨*proof*⟩

lemma *fmdrop-set-add-distrib*[*simp*]: *fmdrop-set A* (*m* ++$_f$ *n*) = *fmdrop-set A m* ++$_f$ *fmdrop-set A n*
⟨*proof*⟩

lemma *fmdrop-fset-add-distrib*[*simp*]: *fmdrop-fset A* (*m* ++$_f$ *n*) = *fmdrop-fset A m* ++$_f$ *fmdrop-fset A n*
⟨*proof*⟩

lemma *fmrestrict-set-add-distrib*[*simp*]:
fmrestrict-set A (*m* ++$_f$ *n*) = *fmrestrict-set A m* ++$_f$ *fmrestrict-set A n*
⟨*proof*⟩

lemma *fmrestrict-fset-add-distrib*[*simp*]:
fmrestrict-fset A (*m* ++$_f$ *n*) = *fmrestrict-fset A m* ++$_f$ *fmrestrict-fset A n*
⟨*proof*⟩

lemma *fmadd-empty*[*simp*]: *fmempty* ++$_f$ *m* = *m m* ++$_f$ *fmempty* = *m*
⟨*proof*⟩

lemma *fmadd-idempotent*[*simp*]: *m* ++$_f$ *m* = *m*
⟨*proof*⟩

lemma *fmadd-assoc*[*simp*]: *m* ++$_f$ (*n* ++$_f$ *p*) = *m* ++$_f$ *n* ++$_f$ *p*
⟨*proof*⟩

lemma *fmadd-fmupd*[*simp*]: *m* ++$_f$ *fmupd a b n* = *fmupd a b* (*m* ++$_f$ *n*)
⟨*proof*⟩

lift-definition *fmpred* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*a*, '*b*) *fmap* ⇒ *bool*
is *map-pred*
parametric *map-pred-transfer*
⟨*proof*⟩

lemma *fmpredI*[*intro*]:
assumes ⋀*x y*. *fmlookup m x* = *Some y* ⟹ *P x y*
shows *fmpred P m*
⟨*proof*⟩

lemma *fmpredD*[*dest*]: *fmpred P m* ⟹ *fmlookup m x* = *Some y* ⟹ *P x y*
⟨*proof*⟩

lemma *fmpred-iff*: *fmpred P m* ⟷ (∀ *x y*. *fmlookup m x* = *Some y* ⟶ *P x y*)
⟨*proof*⟩

lemma *fmpred-alt-def*: *fmpred P m* ⟷ *fBall* (*fmdom m*) (λ*x*. *P x* (*the* (*fmlookup m x*)))
⟨*proof*⟩

lemma *fmpred-empty*[*intro*!, *simp*]: *fmpred P fmempty*
⟨*proof*⟩

lemma *fmpred-upd*[*intro*]: *fmpred P m* ⟹ *P x y* ⟹ *fmpred P* (*fmupd x y m*)
⟨*proof*⟩

lemma *fmpred-updD*[*dest*]: *fmpred P* (*fmupd x y m*) ⟹ *P x y*
⟨*proof*⟩

lemma *fmpred-add*[*intro*]: *fmpred P m* ⟹ *fmpred P n* ⟹ *fmpred P* (*m* ++$_f$ *n*)
⟨*proof*⟩

lemma *fmpred-filter*[*intro*]: *fmpred P m* ⟹ *fmpred P* (*fmfilter Q m*)
⟨*proof*⟩

lemma *fmpred-drop*[*intro*]: *fmpred P m* ⟹ *fmpred P* (*fmdrop a m*)
⟨*proof*⟩

lemma *fmpred-drop-set*[*intro*]: *fmpred P m* ⟹ *fmpred P* (*fmdrop-set A m*)
⟨*proof*⟩

lemma *fmpred-drop-fset*[*intro*]: *fmpred P m* ⟹ *fmpred P* (*fmdrop-fset A m*)
⟨*proof*⟩

lemma *fmpred-restrict-set*[*intro*]: *fmpred P m* ⟹ *fmpred P* (*fmrestrict-set A m*)
⟨*proof*⟩

lemma *fmpred-restrict-fset*[*intro*]: *fmpred P m* ⟹ *fmpred P* (*fmrestrict-fset A m*)
⟨*proof*⟩

lemma *fmpred-cases*[*consumes 1*]:
assumes *fmpred P m*
obtains (*none*) *fmlookup m x* = *None* | (*some*) *y* **where** *fmlookup m x* = *Some y P x y*
⟨*proof*⟩

lift-definition *fmsubset* :: ('*a*, '*b*) *fmap* ⇒ ('*a*, '*b*) *fmap* ⇒ *bool* (**infix** ⊆$_f$ *50*)
is *map-le*
⟨*proof*⟩

lemma *fmsubset-alt-def*: *m* ⊆$_f$ *n* ⟷ *fmpred* (λ*k v*. *fmlookup n k* = *Some v*) *m*
⟨*proof*⟩

lemma *fmsubset-pred*: *fmpred P m* ⟹ *n* ⊆$_f$ *m* ⟹ *fmpred P n*
⟨*proof*⟩

lemma *fmsubset-filter-mono*: *m* ⊆$_f$ *n* ⟹ *fmfilter P m* ⊆$_f$ *fmfilter P n*
⟨*proof*⟩

lemma *fmsubset-drop-mono*: $m \subseteq_f n \Longrightarrow$ *fmdrop a m* $\subseteq_f$ *fmdrop a n*
⟨*proof*⟩

lemma *fmsubset-drop-set-mono*: $m \subseteq_f n \Longrightarrow$ *fmdrop-set A m* $\subseteq_f$ *fmdrop-set A n*
⟨*proof*⟩

lemma *fmsubset-drop-fset-mono*: $m \subseteq_f n \Longrightarrow$ *fmdrop-fset A m* $\subseteq_f$ *fmdrop-fset A n*
⟨*proof*⟩

lemma *fmsubset-restrict-set-mono*: $m \subseteq_f n \Longrightarrow$ *fmrestrict-set A m* $\subseteq_f$ *fmrestrict-set A n*
⟨*proof*⟩

lemma *fmsubset-restrict-fset-mono*: $m \subseteq_f n \Longrightarrow$ *fmrestrict-fset A m* $\subseteq_f$ *fmrestrict-fset A n*
⟨*proof*⟩

lift-definition *fset-of-fmap* :: $('a, 'b)$ *fmap* $\Rightarrow$ $('a \times 'b)$ *fset* **is** *set-of-map*
⟨*proof*⟩

lemma *fset-of-fmap-inj*[*intro*, *simp*]: *inj fset-of-fmap*
⟨*proof*⟩

lemma *fset-of-fmap-iff*[*simp*]: $(a, b) \mid\in\mid$ *fset-of-fmap m* $\longleftrightarrow$ *fmlookup m a* = *Some b*
⟨*proof*⟩

lemma *fset-of-fmap-iff'*[*simp*]: $(a, b) \in$ *fset* (*fset-of-fmap m*) $\longleftrightarrow$ *fmlookup m a* = *Some b*
⟨*proof*⟩

lift-definition *fmap-of-list* :: $('a \times 'b)$ *list* $\Rightarrow$ $('a, 'b)$ *fmap*
  is *map-of*
  parametric *map-of-transfer*
⟨*proof*⟩

lemma *fmap-of-list-simps*[*simp*]:
  fmap-of-list [] = *fmempty*
  fmap-of-list $((k, v) \# kvs)$ = *fmupd k v* (*fmap-of-list kvs*)
⟨*proof*⟩

lemma *fmap-of-list-app*[*simp*]: *fmap-of-list* (*xs* @ *ys*) = *fmap-of-list ys* $++_f$ *fmap-of-list xs*
⟨*proof*⟩

lemma *fmupd-alt-def*: *fmupd k v m* = *m* $++_f$ *fmap-of-list* $[(k, v)]$
⟨*proof*⟩

lemma *fmpred-of-list*[*intro*]:
  assumes $\bigwedge k\ v.\ (k,\ v) \in set\ xs \Longrightarrow P\ k\ v$
  shows *fmpred P* (*fmap-of-list xs*)
⟨*proof*⟩

lemma *fmap-of-list-SomeD*: *fmlookup* (*fmap-of-list xs*) *k* = *Some v* ⟹ (*k*, *v*) ∈ *set xs*
⟨*proof*⟩

lemma *fmdom-fmap-of-list*[*simp*]: *fmdom* (*fmap-of-list xs*) = *fset-of-list* (*map fst xs*)
⟨*proof*⟩

lift-definition *fmrel-on-fset* :: *'a fset* ⇒ (*'b* ⇒ *'c* ⇒ *bool*) ⇒ (*'a*, *'b*) *fmap* ⇒ (*'a*, *'c*) *fmap* ⇒ *bool*
  is *rel-map-on-set*
⟨*proof*⟩

lemma *fmrel-on-fset-alt-def*: *fmrel-on-fset S P m n* ⟷ *fBall S* (λ*x*. *rel-option P* (*fmlookup m x*) (*fmlookup n x*))
⟨*proof*⟩

lemma *fmrel-on-fsetI*[*intro*]:
  assumes $\bigwedge x.\ x\ |\in|\ S \Longrightarrow rel\text{-}option\ P\ (fmlookup\ m\ x)\ (fmlookup\ n\ x)$
  shows *fmrel-on-fset S P m n*
⟨*proof*⟩

lemma *fmrel-on-fset-mono*[*mono*]: *R* ≤ *Q* ⟹ *fmrel-on-fset S R* ≤ *fmrel-on-fset S Q*
⟨*proof*⟩

lemma *fmrel-on-fsetD*: *x* |∈| *S* ⟹ *fmrel-on-fset S P m n* ⟹ *rel-option P* (*fmlookup m x*) (*fmlookup n x*)
⟨*proof*⟩

lemma *fmrel-on-fsubset*: *fmrel-on-fset S R m n* ⟹ *T* |⊆| *S* ⟹ *fmrel-on-fset T R m n*
⟨*proof*⟩

lemma *fmrel-on-fset-unionI*:
  fmrel-on-fset A R m n ⟹ *fmrel-on-fset B R m n* ⟹ *fmrel-on-fset* (*A* |∪| *B*) *R m n*
⟨*proof*⟩

lemma *fmrel-on-fset-updateI*:
  assumes *fmrel-on-fset S P m n P* v_1 v_2
  shows *fmrel-on-fset* (*finsert k S*) *P* (*fmupd k* v_1 *m*) (*fmupd k* v_2 *n*)
⟨*proof*⟩

end

33.4 BNF setup

lift-bnf ('*a*, *fmran'*: '*b*) *fmap* [*wits*: *Map.empty*]
  for *map*: *fmmap*
    rel: *fmrel*
  ⟨*proof*⟩

declare *fmap.pred-mono*[*mono*]

context includes *lifting-syntax* **begin**

lemma *fmmap-transfer*[*transfer-rule*]:
  (*op* = ===> *pcr-fmap op* = *op* = ===> *pcr-fmap op* = *op* =) (λ*f*. *op* ∘ (*map-option f*)) *fmmap*
  ⟨*proof*⟩

lemma *fmran'-transfer*[*transfer-rule*]:
  (*pcr-fmap op* = *op* = ===> *op* =) (λ*x*. *UNION* (*range x*) *set-option*) *fmran'*
  ⟨*proof*⟩

lemma *fmrel-transfer*[*transfer-rule*]:
  (*op* = ===> *pcr-fmap op* = *op* = ===> *pcr-fmap op* = *op* = ===> *op* =) *rel-map fmrel*
  ⟨*proof*⟩

end

lemma *fmran'-alt-def*: *fmran' m* = *fset* (*fmran m*)
including *fset.lifting*
⟨*proof*⟩

lemma *fmlookup-ran'-iff*: *y* ∈ *fmran' m* ⟷ (∃ *x*. *fmlookup m x* = *Some y*)
⟨*proof*⟩

lemma *fmran'I*: *fmlookup m x* = *Some y* ⟹ *y* ∈ *fmran' m* ⟨*proof*⟩

lemma *fmran'E*[*elim*]:
  assumes *y* ∈ *fmran' m*
  obtains *x* **where** *fmlookup m x* = *Some y*
⟨*proof*⟩

lemma *fmrel-iff*: *fmrel R m n* ⟷ (∀ *x*. *rel-option R* (*fmlookup m x*) (*fmlookup n x*))
⟨*proof*⟩

lemma *fmrelI*[*intro*]:
assumes $\bigwedge x.$ *rel-option R* (*fmlookup m x*) (*fmlookup n x*)
shows *fmrel R m n*
⟨*proof*⟩

lemma *fmrel-upd*[*intro*]: *fmrel P m n* $\Longrightarrow$ *P x y* $\Longrightarrow$ *fmrel P* (*fmupd k x m*) (*fmupd k y n*)
⟨*proof*⟩

lemma *fmrelD*[*dest*]: *fmrel P m n* $\Longrightarrow$ *rel-option P* (*fmlookup m x*) (*fmlookup n x*)
⟨*proof*⟩

lemma *fmrel-addI*[*intro*]:
assumes *fmrel P m n fmrel P a b*
shows *fmrel P* (*m* $++_f$ *a*) (*n* $++_f$ *b*)
⟨*proof*⟩

lemma *fmrel-cases*[*consumes 1*]:
assumes *fmrel P m n*
obtains (*none*) *fmlookup m x* = *None fmlookup n x* = *None*
| (*some*) *a b* **where** *fmlookup m x* = *Some a fmlookup n x* = *Some b P a b*
⟨*proof*⟩

lemma *fmrel-filter*[*intro*]: *fmrel P m n* $\Longrightarrow$ *fmrel P* (*fmfilter Q m*) (*fmfilter Q n*)
⟨*proof*⟩

lemma *fmrel-drop*[*intro*]: *fmrel P m n* $\Longrightarrow$ *fmrel P* (*fmdrop a m*) (*fmdrop a n*)
⟨*proof*⟩

lemma *fmrel-drop-set*[*intro*]: *fmrel P m n* $\Longrightarrow$ *fmrel P* (*fmdrop-set A m*) (*fmdrop-set A n*)
⟨*proof*⟩

lemma *fmrel-drop-fset*[*intro*]: *fmrel P m n* $\Longrightarrow$ *fmrel P* (*fmdrop-fset A m*) (*fmdrop-fset A n*)
⟨*proof*⟩

lemma *fmrel-restrict-set*[*intro*]: *fmrel P m n* $\Longrightarrow$ *fmrel P* (*fmrestrict-set A m*) (*fmrestrict-set A n*)
⟨*proof*⟩

lemma *fmrel-restrict-fset*[*intro*]: *fmrel P m n* $\Longrightarrow$ *fmrel P* (*fmrestrict-fset A m*) (*fmrestrict-fset A n*)
⟨*proof*⟩

lemma *fmrel-on-fset-fmrel-restrict*:
fmrel-on-fset S P m n $\longleftrightarrow$ *fmrel P* (*fmrestrict-fset S m*) (*fmrestrict-fset S n*)
⟨*proof*⟩

lemma *fmrel-on-fset-refl-strong*:
assumes $\bigwedge x\ y.\ x\ |\in|\ S \Longrightarrow fmlookup\ m\ x = Some\ y \Longrightarrow P\ y\ y$
shows *fmrel-on-fset S P m m*
⟨*proof*⟩

lemma *fmrel-on-fset-addI*:
assumes *fmrel-on-fset S P m n fmrel-on-fset S P a b*
shows *fmrel-on-fset S P* $(m +\!\!+_f a)$ $(n +\!\!+_f b)$
⟨*proof*⟩

lemma *fmrel-fmdom-eq*:
assumes *fmrel P x y*
shows *fmdom x = fmdom y*
⟨*proof*⟩

lemma *fmrel-fmdom′-eq*: *fmrel P x y* $\Longrightarrow$ *fmdom′ x = fmdom′ y*
⟨*proof*⟩

lemma *fmrel-rel-fmran*:
assumes *fmrel P x y*
shows *rel-fset P* (*fmran x*) (*fmran y*)
⟨*proof*⟩

lemma *fmrel-rel-fmran′*: *fmrel P x y* $\Longrightarrow$ *rel-set P* (*fmran′ x*) (*fmran′ y*)
⟨*proof*⟩

lemma *pred-fmap-fmpred*[*simp*]: *pred-fmap P* = *fmpred* (λ-. *P*)
⟨*proof*⟩
including *fset.lifting*
⟨*proof*⟩

lemma *pred-fmap-id*[*simp*]: *pred-fmap id* (*fmmap f m*) $\longleftrightarrow$ *pred-fmap f m*
⟨*proof*⟩

lemma *pred-fmapD*: *pred-fmap P m* $\Longrightarrow$ *x* $|\in|$ *fmran m* $\Longrightarrow$ *P x*
⟨*proof*⟩

lemma *fmlookup-map*[*simp*]: *fmlookup* (*fmmap f m*) *x* = *map-option f* (*fmlookup m x*)
⟨*proof*⟩

lemma *fmpred-map*[*simp*]: *fmpred P* (*fmmap f m*) $\longleftrightarrow$ *fmpred* (λ*k v*. *P k* (*f v*)) *m*
⟨*proof*⟩

lemma *fmpred-id*[*simp*]: *fmpred* (λ-. *id*) (*fmmap f m*) $\longleftrightarrow$ *fmpred* (λ-. *f*) *m*
⟨*proof*⟩

lemma *fmmap-add*[*simp*]: *fmmap f* (*m* ++$_f$ *n*) = *fmmap f m* ++$_f$ *fmmap f n*
⟨*proof*⟩

lemma *fmmap-empty*[*simp*]: *fmmap f fmempty* = *fmempty*
⟨*proof*⟩

lemma *fmdom-map*[*simp*]: *fmdom* (*fmmap f m*) = *fmdom m*
including *fset.lifting*
⟨*proof*⟩

lemma *fmdom'-map*[*simp*]: *fmdom'* (*fmmap f m*) = *fmdom' m*
⟨*proof*⟩

lemma *fmran-fmmap*[*simp*]: *fmran* (*fmmap f m*) = *f* |‘| *fmran m*
including *fset.lifting*
⟨*proof*⟩

lemma *fmran'-fmmap*[*simp*]: *fmran'* (*fmmap f m*) = *f* ‘ *fmran' m*
⟨*proof*⟩

lemma *fmfilter-fmmap*[*simp*]: *fmfilter P* (*fmmap f m*) = *fmmap f* (*fmfilter P m*)
⟨*proof*⟩

lemma *fmdrop-fmmap*[*simp*]: *fmdrop a* (*fmmap f m*) = *fmmap f* (*fmdrop a m*)
⟨*proof*⟩
lemma *fmdrop-set-fmmap*[*simp*]: *fmdrop-set A* (*fmmap f m*) = *fmmap f* (*fmdrop-set A m*) ⟨*proof*⟩
lemma *fmdrop-fset-fmmap*[*simp*]: *fmdrop-fset A* (*fmmap f m*) = *fmmap f* (*fmdrop-fset A m*) ⟨*proof*⟩
lemma *fmrestrict-set-fmmap*[*simp*]: *fmrestrict-set A* (*fmmap f m*) = *fmmap f* (*fmrestrict-set A m*) ⟨*proof*⟩
lemma *fmrestrict-fset-fmmap*[*simp*]: *fmrestrict-fset A* (*fmmap f m*) = *fmmap f* (*fmrestrict-fset A m*) ⟨*proof*⟩

lemma *fmmap-subset*[*intro*]: *m* ⊆$_f$ *n* ⟹ *fmmap f m* ⊆$_f$ *fmmap f n*
⟨*proof*⟩

lemma *fmmap-fset-of-fmap*: *fset-of-fmap* (*fmmap f m*) = (λ(*k*, *v*). (*k*, *f v*)) |‘| *fset-of-fmap m*
including *fset.lifting*
⟨*proof*⟩

33.5 *size* setup

definition *size-fmap* :: ('*a* ⇒ *nat*) ⇒ ('*b* ⇒ *nat*) ⇒ ('*a*, '*b*) *fmap* ⇒ *nat* **where**
[*simp*]: *size-fmap f g m* = *size-fset* (λ(*a*, *b*). *f a* + *g b*) (*fset-of-fmap m*)

instantiation *fmap* :: (*type*, *type*) *size* **begin**

definition *size-fmap* **where**
size-fmap-overloaded-def: *size-fmap* = *Finite-Map.size-fmap* (λ-. *0*) (λ-. *0*)

instance ⟨*proof*⟩

end

lemma *size-fmap-overloaded-simps*[*simp*]: *size x* = *size* (*fset-of-fmap x*)
⟨*proof*⟩

lemma *fmap-size-o-map*: *inj h* ⟹ *size-fmap f g* ∘ *fmmap h* = *size-fmap f* (*g* ∘ *h*)
⟨*proof*⟩

⟨*ML*⟩

33.6 Additional operations

lift-definition *fmmap-keys* :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ ('*a*, '*b*) *fmap* ⇒ ('*a*, '*c*) *fmap* **is**
λ*f m a*. *map-option* (*f a*) (*m a*)
⟨*proof*⟩

lemma *fmpred-fmmap-keys*[*simp*]: *fmpred P* (*fmmap-keys f m*) = *fmpred* (λ*a b*. *P a* (*f a b*)) *m*
⟨*proof*⟩

lemma *fmdom-fmmap-keys*[*simp*]: *fmdom* (*fmmap-keys f m*) = *fmdom m*
including *fset.lifting*
⟨*proof*⟩

lemma *fmlookup-fmmap-keys*[*simp*]: *fmlookup* (*fmmap-keys f m*) *x* = *map-option* (*f x*) (*fmlookup m x*)
⟨*proof*⟩

lemma *fmfilter-fmmap-keys*[*simp*]: *fmfilter P* (*fmmap-keys f m*) = *fmmap-keys f* (*fmfilter P m*)
⟨*proof*⟩

lemma *fmdrop-fmmap-keys*[*simp*]: *fmdrop a* (*fmmap-keys f m*) = *fmmap-keys f* (*fmdrop a m*)
⟨*proof*⟩

lemma *fmdrop-set-fmmap-keys*[*simp*]: *fmdrop-set A* (*fmmap-keys f m*) = *fmmap-keys f* (*fmdrop-set A m*)
⟨*proof*⟩

lemma *fmdrop-fset-fmmap-keys*[*simp*]: *fmdrop-fset A* (*fmmap-keys f m*) = *fmmap-keys f* (*fmdrop-fset A m*)
⟨*proof*⟩

lemma *fmrestrict-set-fmmap-keys*[*simp*]: *fmrestrict-set A* (*fmmap-keys f m*) = *fmmap-keys f* (*fmrestrict-set A m*)
⟨*proof*⟩

lemma *fmrestrict-fset-fmmap-keys*[*simp*]: *fmrestrict-fset A* (*fmmap-keys f m*) = *fmmap-keys f* (*fmrestrict-fset A m*)
⟨*proof*⟩

lemma *fmmap-keys-subset*[*intro*]: *m* ⊆$_f$ *n* ⟹ *fmmap-keys f m* ⊆$_f$ *fmmap-keys f n*
⟨*proof*⟩

33.7 Lifting/transfer setup

context includes *lifting-syntax* **begin**

lemma *fmempty-transfer*[*simp*, *intro*, *transfer-rule*]: *fmrel P fmempty fmempty*
⟨*proof*⟩

lemma *fmadd-transfer*[*transfer-rule*]:
(*fmrel P* ===> *fmrel P* ===> *fmrel P*) *fmadd fmadd*
⟨*proof*⟩

lemma *fmupd-transfer*[*transfer-rule*]:
(*op* = ===> *P* ===> *fmrel P* ===> *fmrel P*) *fmupd fmupd*
⟨*proof*⟩

end

33.8 View as datatype

lemma *fmap-distinct*[*simp*]:
fmempty ≠ *fmupd k v m*
fmupd k v m ≠ *fmempty*
⟨*proof*⟩

lifting-update *fmap.lifting*

lemma *fmap-exhaust*[*case-names fmempty fmupd*, *cases type*: *fmap*]:
assumes *fmempty*: *m* = *fmempty* ⟹ *P*
assumes *fmupd*: ⋀*x y m'*. *m* = *fmupd x y m'* ⟹ *x* |∉| *fmdom m'* ⟹ *P*
shows *P*
⟨*proof*⟩ **including** *fmap.lifting fset.lifting*
⟨*proof*⟩

lemma *fmap-induct*[*case-names fmempty fmupd*, *induct type*: *fmap*]:
assumes *P fmempty*
assumes (⋀*x y m*. *P m* ⟹ *fmlookup m x* = *None* ⟹ *P* (*fmupd x y m*))
shows *P m*

⟨*proof*⟩

33.9 Code setup

instantiation *fmap* :: (*type*, *equal*) *equal* **begin**

definition *equal-fmap* ≡ *fmrel HOL.equal*

instance ⟨*proof*⟩

end

lemma *fBall-alt-def*: *fBall S P* ⟷ (∀ *x*. *x* |∈| *S* ⟶ *P x*)
⟨*proof*⟩

lemma *fmrel-code*:
fmrel R m n ⟷
fBall (*fmdom m*) (λ*x*. *rel-option R* (*fmlookup m x*) (*fmlookup n x*)) ∧
fBall (*fmdom n*) (λ*x*. *rel-option R* (*fmlookup m x*) (*fmlookup n x*))
⟨*proof*⟩

lemmas [*code*] =
fmrel-code
fmran'-alt-def
fmdom'-alt-def
fmfilter-alt-defs
pred-fmap-fmpred
fmsubset-alt-def
fmupd-alt-def
fmrel-on-fset-alt-def
fmpred-alt-def

code-datatype *fmap-of-list*
quickcheck-generator *fmap constructors*: *fmap-of-list*

context includes *fset.lifting* **begin**

lemma *fmlookup-of-list*[*code*]: *fmlookup* (*fmap-of-list m*) = *map-of m*
⟨*proof*⟩

lemma *fmempty-of-list*[*code*]: *fmempty* = *fmap-of-list* []
⟨*proof*⟩

lemma *fmran-of-list*[*code*]: *fmran* (*fmap-of-list m*) = *snd* |‘| *fset-of-list* (*AList.clearjunk m*)
⟨*proof*⟩

lemma *fmdom-of-list*[*code*]: *fmdom* (*fmap-of-list m*) = *fst* |‘| *fset-of-list m*

⟨*proof*⟩

lemma *fmfilter-of-list*[*code*]: *fmfilter P* (*fmap-of-list m*) = *fmap-of-list* (*filter* (λ(*k*, -). *P k*) *m*)
⟨*proof*⟩

lemma *fmadd-of-list*[*code*]: *fmap-of-list m* ++$_f$ *fmap-of-list n* = *fmap-of-list* (*AList.merge m n*)
⟨*proof*⟩

lemma *fmmap-of-list*[*code*]: *fmmap f* (*fmap-of-list m*) = *fmap-of-list* (*map* (*apsnd f*) *m*)
⟨*proof*⟩

lemma *fmmap-keys-of-list*[*code*]: *fmmap-keys f* (*fmap-of-list m*) = *fmap-of-list* (*map* (λ(*a*, *b*). (*a*, *f a b*)) *m*)
⟨*proof*⟩

end

33.10 Instances

lemma *exists-map-of*:
assumes *finite* (*dom m*) **shows** ∃ *xs*. *map-of xs* = *m*
⟨*proof*⟩

lemma *exists-fmap-of-list*: ∃ *xs*. *fmap-of-list xs* = *m*
⟨*proof*⟩

lemma *fmap-of-list-surj*[*simp*, *intro*]: *surj fmap-of-list*
⟨*proof*⟩

instance *fmap* :: (*countable*, *countable*) *countable*
⟨*proof*⟩

instance *fmap* :: (*finite*, *finite*) *finite*
⟨*proof*⟩

lifting-update *fmap.lifting*
lifting-forget *fmap.lifting*

end

34 Logarithm of Natural Numbers

theory *Log-Nat*
imports *Complex-Main*
begin

definition *floorlog* :: *nat* ⇒ *nat* ⇒ *nat* **where**
floorlog b a = (*if a* > *0* ∧ *b* > *1 then nat* ⌊*log b a*⌋ + *1 else 0*)

lemma *floorlog-mono*: *x* ≤ *y* ⟹ *floorlog b x* ≤ *floorlog b y*
⟨*proof*⟩

lemma *floorlog-bounds*:
assumes *x* > *0 b* > *1*
shows *b* ^ (*floorlog b x* − *1*) ≤ *x* ∧ *x* < *b* ^ (*floorlog b x*)
⟨*proof*⟩

lemma *floorlog-power*[*simp*]:
assumes *a* > *0 b* > *1*
shows *floorlog b* (*a* ∗ *b* ^ *c*) = *floorlog b a* + *c*
⟨*proof*⟩

lemma *floor-log-add-eqI*:
fixes *a*::*nat* **and** *b*::*nat* **and** *r*::*real*
assumes *b* > *1 a* ≥ *1 0* ≤ *r r* < *1*
shows ⌊*log b* (*a* + *r*)⌋ = ⌊*log b a*⌋
⟨*proof*⟩

lemma *divide-nat-diff-div-nat-less-one*:
fixes *x b*::*nat* **shows** *x* / *b* − *x div b* < *1*
⟨*proof*⟩

lemma *floor-log-div*:
fixes *b x* :: *nat* **assumes** *b* > *1 x* > *0 x div b* > *0*
shows ⌊*log b x*⌋ = ⌊*log b* (*x div b*)⌋ + *1*
⟨*proof*⟩

lemma *compute-floorlog*[*code*]:
floorlog b x = (*if x* > *0* ∧ *b* > *1 then floorlog b* (*x div b*) + *1 else 0*)
⟨*proof*⟩

lemma *floor-log-eq-if*:
fixes *b x y* :: *nat*
assumes *x div b* = *y div b b* > *1 x* > *0 x div b* ≥ *1*
shows *floor*(*log b x*) = *floor*(*log b y*)
⟨*proof*⟩

lemma *floorlog-eq-if*:
fixes *b x y* :: *nat*
assumes *x div b* = *y div b b* > *1 x* > *0 x div b* ≥ *1*
shows *floorlog b x* = *floorlog b y*
⟨*proof*⟩

definition *bitlen* :: *int* ⇒ *int* **where** *bitlen a* = *floorlog 2* (*nat a*)

lemma *bitlen-alt-def*: *bitlen a = (if a > 0 then ⌊log 2 a⌋ + 1 else 0)*
⟨*proof*⟩

lemma *bitlen-nonneg*: *0 ≤ bitlen x*
⟨*proof*⟩

lemma *bitlen-bounds*:
assumes *x > 0*
shows *2 ^ nat (bitlen x − 1) ≤ x ∧ x < 2 ^ nat (bitlen x)*
⟨*proof*⟩

lemma *bitlen-pow2*[*simp*]:
assumes *b > 0*
shows *bitlen (b * 2 ^ c) = bitlen b + c*
⟨*proof*⟩

lemma *compute-bitlen*[*code*]:
bitlen x = (if x > 0 then bitlen (x div 2) + 1 else 0)
⟨*proof*⟩

lemma *bitlen-eq-zero-iff*: *bitlen x = 0 ⟷ x ≤ 0*
⟨*proof*⟩

lemma *bitlen-div*:
assumes *0 < m*
shows *1 ≤ real-of-int m / 2^nat (bitlen m − 1)*
and *real-of-int m / 2^nat (bitlen m − 1) < 2*
⟨*proof*⟩

end

35 Various algebraic structures combined with a lattice

theory *Lattice-Algebras*
imports *Complex-Main*
begin

class *semilattice-inf-ab-group-add = ordered-ab-group-add + semilattice-inf*
begin

lemma *add-inf-distrib-left*: *a + inf b c = inf (a + b) (a + c)*
⟨*proof*⟩

lemma *add-inf-distrib-right*: *inf a b + c = inf (a + c) (b + c)*
⟨*proof*⟩

end

class *semilattice-sup-ab-group-add* = *ordered-ab-group-add* + *semilattice-sup*
begin

lemma *add-sup-distrib-left*: $a + sup\ b\ c = sup\ (a + b)\ (a + c)$
⟨*proof*⟩

lemma *add-sup-distrib-right*: $sup\ a\ b + c = sup\ (a + c)\ (b + c)$
⟨*proof*⟩

end

class *lattice-ab-group-add* = *ordered-ab-group-add* + *lattice*
begin

subclass *semilattice-inf-ab-group-add* ⟨*proof*⟩
subclass *semilattice-sup-ab-group-add* ⟨*proof*⟩

lemmas *add-sup-inf-distribs* =
add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left

lemma *inf-eq-neg-sup*: $inf\ a\ b = -\ sup\ (-\ a)\ (-\ b)$
⟨*proof*⟩

lemma *sup-eq-neg-inf*: $sup\ a\ b = -\ inf\ (-\ a)\ (-\ b)$
⟨*proof*⟩

lemma *neg-inf-eq-sup*: $-\ inf\ a\ b = sup\ (-\ a)\ (-\ b)$
⟨*proof*⟩

lemma *diff-inf-eq-sup*: $a - inf\ b\ c = a + sup\ (-\ b)\ (-\ c)$
⟨*proof*⟩

lemma *neg-sup-eq-inf*: $-\ sup\ a\ b = inf\ (-\ a)\ (-\ b)$
⟨*proof*⟩

lemma *diff-sup-eq-inf*: $a - sup\ b\ c = a + inf\ (-\ b)\ (-\ c)$
⟨*proof*⟩

lemma *add-eq-inf-sup*: $a + b = sup\ a\ b + inf\ a\ b$
⟨*proof*⟩

35.1 Positive Part, Negative Part, Absolute Value

definition *nprt* :: $'a \Rightarrow 'a$
where $nprt\ x = inf\ x\ 0$

definition *pprt* :: $'a \Rightarrow 'a$

where $pprt\ x = sup\ x\ 0$

lemma *pprt-neg*: $pprt\ (-\ x) = -\ nprt\ x$
⟨*proof*⟩

lemma *nprt-neg*: $nprt\ (-\ x) = -\ pprt\ x$
⟨*proof*⟩

lemma *prts*: $a = pprt\ a + nprt\ a$
⟨*proof*⟩

lemma *zero-le-pprt*[*simp*]: $0 \leq pprt\ a$
⟨*proof*⟩

lemma *nprt-le-zero*[*simp*]: $nprt\ a \leq 0$
⟨*proof*⟩

lemma *le-eq-neg*: $a \leq -\ b \longleftrightarrow a + b \leq 0$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *pprt-0*[*simp*]: $pprt\ 0 = 0$ ⟨*proof*⟩
lemma *nprt-0*[*simp*]: $nprt\ 0 = 0$ ⟨*proof*⟩

lemma *pprt-eq-id* [*simp*, *no-atp*]: $0 \leq x \Longrightarrow pprt\ x = x$
⟨*proof*⟩

lemma *nprt-eq-id* [*simp*, *no-atp*]: $x \leq 0 \Longrightarrow nprt\ x = x$
⟨*proof*⟩

lemma *pprt-eq-0* [*simp*, *no-atp*]: $x \leq 0 \Longrightarrow pprt\ x = 0$
⟨*proof*⟩

lemma *nprt-eq-0* [*simp*, *no-atp*]: $0 \leq x \Longrightarrow nprt\ x = 0$
⟨*proof*⟩

lemma *sup-0-imp-0*:
assumes $sup\ a\ (-\ a) = 0$
shows $a = 0$
⟨*proof*⟩

lemma *inf-0-imp-0*: $inf\ a\ (-\ a) = 0 \Longrightarrow a = 0$
⟨*proof*⟩

lemma *inf-0-eq-0* [*simp*, *no-atp*]: $inf\ a\ (-\ a) = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *sup-0-eq-0* [*simp*, *no-atp*]: $sup\ a\ (-\ a) = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *zero-le-double-add-iff-zero-le-single-add* [*simp*]: $0 \leq a + a \longleftrightarrow 0 \leq a$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *double-zero* [*simp*]: $a + a = 0 \longleftrightarrow a = 0$
⟨*proof*⟩

lemma *zero-less-double-add-iff-zero-less-single-add* [*simp*]: $0 < a + a \longleftrightarrow 0 < a$
⟨*proof*⟩

lemma *double-add-le-zero-iff-single-add-le-zero* [*simp*]: $a + a \leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *double-add-less-zero-iff-single-less-zero* [*simp*]: $a + a < 0 \longleftrightarrow a < 0$
⟨*proof*⟩

declare *neg-inf-eq-sup* [*simp*]
and *neg-sup-eq-inf* [*simp*]
and *diff-inf-eq-sup* [*simp*]
and *diff-sup-eq-inf* [*simp*]

lemma *le-minus-self-iff*: $a \leq - a \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *minus-le-self-iff*: $- a \leq a \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *zero-le-iff-zero-nprt*: $0 \leq a \longleftrightarrow nprt\ a = 0$
⟨*proof*⟩

lemma *le-zero-iff-zero-pprt*: $a \leq 0 \longleftrightarrow pprt\ a = 0$
⟨*proof*⟩

lemma *le-zero-iff-pprt-id*: $0 \leq a \longleftrightarrow pprt\ a = a$
⟨*proof*⟩

lemma *zero-le-iff-nprt-id*: $a \leq 0 \longleftrightarrow nprt\ a = a$
⟨*proof*⟩

lemma *pprt-mono* [*simp*, *no-atp*]: $a \leq b \Longrightarrow pprt\ a \leq pprt\ b$
⟨*proof*⟩

lemma *nprt-mono* [*simp*, *no-atp*]: $a \leq b \Longrightarrow nprt\ a \leq nprt\ b$
⟨*proof*⟩

end

lemmas *add-sup-inf-distribs* =

add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left

class *lattice-ab-group-add-abs* = *lattice-ab-group-add* + *abs* +
  assumes *abs-lattice*: $|a| = sup\ a\ (-\ a)$
begin

lemma *abs-prts*: $|a| = pprt\ a\ -\ nprt\ a$
⟨*proof*⟩

subclass *ordered-ab-group-add-abs*
⟨*proof*⟩

end

lemma *sup-eq-if*:
  fixes *a* :: 'a::{*lattice-ab-group-add*,*linorder*}
  shows $sup\ a\ (-\ a) = (if\ a < 0\ then\ -\ a\ else\ a)$
  ⟨*proof*⟩

lemma *abs-if-lattice*:
  fixes *a* :: 'a::{*lattice-ab-group-add-abs*,*linorder*}
  shows $|a| = (if\ a < 0\ then\ -\ a\ else\ a)$
  ⟨*proof*⟩

lemma *estimate-by-abs*:
  fixes *a b c* :: 'a::*lattice-ab-group-add-abs*
  assumes $a + b \leq c$
  shows $a \leq c + |b|$
⟨*proof*⟩

class *lattice-ring* = *ordered-ring* + *lattice-ab-group-add-abs*
begin

subclass *semilattice-inf-ab-group-add* ⟨*proof*⟩
subclass *semilattice-sup-ab-group-add* ⟨*proof*⟩

end

lemma *abs-le-mult*:
  fixes *a b* :: 'a::*lattice-ring*
  shows $|a * b| \leq |a| * |b|$
⟨*proof*⟩

instance *lattice-ring* ⊆ *ordered-ring-abs*
⟨*proof*⟩

lemma *mult-le-prts*:
  fixes *a b* :: 'a::*lattice-ring*

assumes *a1* ≤ *a*
and *a* ≤ *a2*
and *b1* ≤ *b*
and *b* ≤ *b2*
shows *a* ∗ *b* ≤
pprt a2 ∗ *pprt b2* + *pprt a1* ∗ *nprt b2* + *nprt a2* ∗ *pprt b1* + *nprt a1* ∗ *nprt b1*
⟨*proof*⟩

lemma *mult-ge-prts*:
fixes *a b* :: *'a*::*lattice-ring*
assumes *a1* ≤ *a*
and *a* ≤ *a2*
and *b1* ≤ *b*
and *b* ≤ *b2*
shows *a* ∗ *b* ≥
nprt a1 ∗ *pprt b2* + *nprt a2* ∗ *nprt b2* + *pprt a1* ∗ *pprt b1* + *pprt a2* ∗ *nprt b1*
⟨*proof*⟩

instance *int* :: *lattice-ring*
⟨*proof*⟩

instance *real* :: *lattice-ring*
⟨*proof*⟩

end

36 Floating-Point Numbers

theory *Float*
imports *Log-Nat Lattice-Algebras*
begin

definition *float* = {*m* ∗ *2 powr e* | (*m* :: *int*) (*e* :: *int*). *True*}

typedef *float* = *float*
morphisms *real-of-float float-of*
⟨*proof*⟩

setup-lifting *type-definition-float*

declare *real-of-float* [*code-unfold*]

lemmas *float-of-inject*[*simp*]

declare [[*coercion real-of-float* :: *float* ⇒ *real*]]

lemma *real-of-float-eq*: *f1* = *f2* ⟷ *real-of-float f1* = *real-of-float f2* **for** *f1 f2* ::

float
⟨*proof*⟩

declare *real-of-float-inverse*[*simp*] *float-of-inverse* [*simp*]
declare *real-of-float* [*simp*]

36.1 Real operations preserving the representation as floating point number

lemma *floatI*: $m * 2\ powr\ e = x \Longrightarrow x \in float$ **for** $m\ e :: int$
⟨*proof*⟩

lemma *zero-float*[*simp*]: $0 \in float$
⟨*proof*⟩

lemma *one-float*[*simp*]: $1 \in float$
⟨*proof*⟩

lemma *numeral-float*[*simp*]: $numeral\ i \in float$
⟨*proof*⟩

lemma *neg-numeral-float*[*simp*]: $-\ numeral\ i \in float$
⟨*proof*⟩

lemma *real-of-int-float*[*simp*]: $real\text{-}of\text{-}int\ x \in float$ **for** $x :: int$
⟨*proof*⟩

lemma *real-of-nat-float*[*simp*]: $real\ x \in float$ **for** $x :: nat$
⟨*proof*⟩

lemma *two-powr-int-float*[*simp*]: $2\ powr\ (real\text{-}of\text{-}int\ i) \in float$ **for** $i :: int$
⟨*proof*⟩

lemma *two-powr-nat-float*[*simp*]: $2\ powr\ (real\ i) \in float$ **for** $i :: nat$
⟨*proof*⟩

lemma *two-powr-minus-int-float*[*simp*]: $2\ powr\ -\ (real\text{-}of\text{-}int\ i) \in float$ **for** $i :: int$
⟨*proof*⟩

lemma *two-powr-minus-nat-float*[*simp*]: $2\ powr\ -\ (real\ i) \in float$ **for** $i :: nat$
⟨*proof*⟩

lemma *two-powr-numeral-float*[*simp*]: $2\ powr\ numeral\ i \in float$
⟨*proof*⟩

lemma *two-powr-neg-numeral-float*[*simp*]: $2\ powr\ -\ numeral\ i \in float$
⟨*proof*⟩

lemma *two-pow-float*[*simp*]: $2\ \hat{}\ n \in float$

⟨*proof*⟩

lemma *plus-float*[*simp*]: $r \in \mathit{float} \Longrightarrow p \in \mathit{float} \Longrightarrow r + p \in \mathit{float}$
⟨*proof*⟩

lemma *uminus-float*[*simp*]: $x \in \mathit{float} \Longrightarrow -x \in \mathit{float}$
⟨*proof*⟩

lemma *times-float*[*simp*]: $x \in \mathit{float} \Longrightarrow y \in \mathit{float} \Longrightarrow x * y \in \mathit{float}$
⟨*proof*⟩

lemma *minus-float*[*simp*]: $x \in \mathit{float} \Longrightarrow y \in \mathit{float} \Longrightarrow x - y \in \mathit{float}$
⟨*proof*⟩

lemma *abs-float*[*simp*]: $x \in \mathit{float} \Longrightarrow |x| \in \mathit{float}$
⟨*proof*⟩

lemma *sgn-of-float*[*simp*]: $x \in \mathit{float} \Longrightarrow \mathit{sgn}\ x \in \mathit{float}$
⟨*proof*⟩

lemma *div-power-2-float*[*simp*]: $x \in \mathit{float} \Longrightarrow x\ /\ 2\hat{}d \in \mathit{float}$
⟨*proof*⟩

lemma *div-power-2-int-float*[*simp*]: $x \in \mathit{float} \Longrightarrow x\ /\ (2{::}\mathit{int})\hat{}d \in \mathit{float}$
⟨*proof*⟩

lemma *div-numeral-Bit0-float*[*simp*]:
assumes $x\ /\ \mathit{numeral}\ n \in \mathit{float}$
shows $x\ /\ (\mathit{numeral}\ (\mathit{Num.Bit0}\ n)) \in \mathit{float}$
⟨*proof*⟩

lemma *div-neg-numeral-Bit0-float*[*simp*]:
assumes $x\ /\ \mathit{numeral}\ n \in \mathit{float}$
shows $x\ /\ (-\ \mathit{numeral}\ (\mathit{Num.Bit0}\ n)) \in \mathit{float}$
⟨*proof*⟩

lemma *power-float*[*simp*]:
assumes $a \in \mathit{float}$
shows $a\ \hat{}\ b \in \mathit{float}$
⟨*proof*⟩

lift-definition *Float* :: $\mathit{int} \Rightarrow \mathit{int} \Rightarrow \mathit{float}$ **is** $\lambda(m{::}\mathit{int})\ (e{::}\mathit{int}).\ m * 2\ \mathit{powr}\ e$
⟨*proof*⟩
declare *Float.rep-eq*[*simp*]

code-datatype *Float*

lemma *compute-real-of-float*[*code*]:

real-of-float (*Float m e*) = (*if e* ≥ *0 then m* ∗ *2* ^ *nat e else m* / *2* ^ (*nat* (−*e*)))
⟨*proof*⟩

36.2 Arithmetic operations on floating point numbers

instantiation *float* :: {*ring-1*,*linorder*,*linordered-ring*,*linordered-idom*,*numeral*,*equal*}
begin

lift-definition *zero-float* :: *float* **is** *0* ⟨*proof*⟩
declare *zero-float.rep-eq*[*simp*]

lift-definition *one-float* :: *float* **is** *1* ⟨*proof*⟩
declare *one-float.rep-eq*[*simp*]

lift-definition *plus-float* :: *float* ⇒ *float* ⇒ *float* **is** *op* + ⟨*proof*⟩
declare *plus-float.rep-eq*[*simp*]

lift-definition *times-float* :: *float* ⇒ *float* ⇒ *float* **is** *op* ∗ ⟨*proof*⟩
declare *times-float.rep-eq*[*simp*]

lift-definition *minus-float* :: *float* ⇒ *float* ⇒ *float* **is** *op* − ⟨*proof*⟩
declare *minus-float.rep-eq*[*simp*]

lift-definition *uminus-float* :: *float* ⇒ *float* **is** *uminus* ⟨*proof*⟩
declare *uminus-float.rep-eq*[*simp*]

lift-definition *abs-float* :: *float* ⇒ *float* **is** *abs* ⟨*proof*⟩
declare *abs-float.rep-eq*[*simp*]

lift-definition *sgn-float* :: *float* ⇒ *float* **is** *sgn* ⟨*proof*⟩
declare *sgn-float.rep-eq*[*simp*]

lift-definition *equal-float* :: *float* ⇒ *float* ⇒ *bool* **is** *op* = :: *real* ⇒ *real* ⇒ *bool*
⟨*proof*⟩

lift-definition *less-eq-float* :: *float* ⇒ *float* ⇒ *bool* **is** *op* ≤ ⟨*proof*⟩
declare *less-eq-float.rep-eq*[*simp*]

lift-definition *less-float* :: *float* ⇒ *float* ⇒ *bool* **is** *op* < ⟨*proof*⟩
declare *less-float.rep-eq*[*simp*]

instance
⟨*proof*⟩

end

lemma *real-of-float* [*simp*]: *real-of-float* (*of-nat n*) = *of-nat n*
⟨*proof*⟩

lemma *real-of-float-of-int-eq* [*simp*]: *real-of-float* (*of-int z*) = *of-int z*
⟨*proof*⟩

lemma *Float-0-eq-0*[*simp*]: *Float 0 e* = *0*
⟨*proof*⟩

lemma *real-of-float-power*[*simp*]: *real-of-float* (*f^n*) = *real-of-float f^n* **for** *f* :: *float*
⟨*proof*⟩

lemma *real-of-float-min*: *real-of-float* (*min x y*) = *min* (*real-of-float x*) (*real-of-float y*)
and *real-of-float-max*: *real-of-float* (*max x y*) = *max* (*real-of-float x*) (*real-of-float y*)
for *x y* :: *float*
⟨*proof*⟩

instance *float* :: *unbounded-dense-linorder*
⟨*proof*⟩

instantiation *float* :: *lattice-ab-group-add*
begin

definition *inf-float* :: *float* ⇒ *float* ⇒ *float*
where *inf-float a b* = *min a b*

definition *sup-float* :: *float* ⇒ *float* ⇒ *float*
where *sup-float a b* = *max a b*

instance
⟨*proof*⟩

end

lemma *float-numeral*[*simp*]: *real-of-float* (*numeral x* :: *float*) = *numeral x*
⟨*proof*⟩

lemma *transfer-numeral* [*transfer-rule*]:
rel-fun (*op* =) *pcr-float* (*numeral* :: - ⇒ *real*) (*numeral* :: - ⇒ *float*)
⟨*proof*⟩

lemma *float-neg-numeral*[*simp*]: *real-of-float* (− *numeral x* :: *float*) = − *numeral x*
⟨*proof*⟩

lemma *transfer-neg-numeral* [*transfer-rule*]:
rel-fun (*op* =) *pcr-float* (− *numeral* :: - ⇒ *real*) (− *numeral* :: - ⇒ *float*)
⟨*proof*⟩

lemma *float-of-numeral*[*simp*]: *numeral k* = *float-of* (*numeral k*)

and *float-of-neg-numeral*[*simp*]: $-$ *numeral k* = *float-of* ($-$ *numeral k*)
⟨*proof*⟩

36.3 Quickcheck

instantiation *float* :: *exhaustive*
begin

definition *exhaustive-float* **where**
exhaustive-float f d =
Quickcheck-Exhaustive.exhaustive (λ*x*. *Quickcheck-Exhaustive.exhaustive* (λ*y*. *f* (*Float x y*)) *d*) *d*

instance ⟨*proof*⟩

end

definition (**in** *term-syntax*) [*code-unfold*]:
valtermify-float x y = *Code-Evaluation.valtermify Float* {·} *x* {·} *y*

instantiation *float* :: *full-exhaustive*
begin

definition
full-exhaustive-float f d =
Quickcheck-Exhaustive.full-exhaustive
(λ*x*. *Quickcheck-Exhaustive.full-exhaustive* (λ*y*. *f* (*valtermify-float x y*)) *d*) *d*

instance ⟨*proof*⟩

end

instantiation *float* :: *random*
begin

definition *Quickcheck-Random.random i* =
scomp (*Quickcheck-Random.random* (*2* ^ *nat-of-natural i*))
(λ*man*. *scomp* (*Quickcheck-Random.random i*) (λ*exp*. *Pair* (*valtermify-float man exp*)))

instance ⟨*proof*⟩

end

36.4 Represent floats as unique mantissa and exponent

lemma *int-induct-abs*[*case-names less*]:
fixes *j* :: *int*
assumes *H*: ⋀*n*. (⋀*i*. |*i*| < |*n*| ⟹ *P i*) ⟹ *P n*
shows *P j*

⟨*proof*⟩

lemma *int-cancel-factors*:
fixes *n* :: *int*
assumes *1 < r*
shows *n = 0 ∨ (∃ k i. n = k * r ^ i ∧ ¬ r dvd k)*
⟨*proof*⟩

lemma *mult-powr-eq-mult-powr-iff-asym*:
fixes *m1 m2 e1 e2* :: *int*
assumes *m1*: *¬ 2 dvd m1*
and *e1 ≤ e2*
shows *m1 * 2 powr e1 = m2 * 2 powr e2 ⟷ m1 = m2 ∧ e1 = e2*
(**is** *?lhs ⟷ ?rhs*)
⟨*proof*⟩

lemma *mult-powr-eq-mult-powr-iff*:
*¬ 2 dvd m1 ⟹ ¬ 2 dvd m2 ⟹ m1 * 2 powr e1 = m2 * 2 powr e2 ⟷ m1 = m2 ∧ e1 = e2*
for *m1 m2 e1 e2* :: *int*
⟨*proof*⟩

lemma *floatE-normed*:
assumes *x*: *x ∈ float*
obtains (*zero*) *x = 0*
| (*powr*) *m e* :: *int* **where** *x = m * 2 powr e ¬ 2 dvd m x ≠ 0*
⟨*proof*⟩

lemma *float-normed-cases*:
fixes *f* :: *float*
obtains (*zero*) *f = 0*
| (*powr*) *m e* :: *int* **where** *real-of-float f = m * 2 powr e ¬ 2 dvd m f ≠ 0*
⟨*proof*⟩

definition *mantissa* :: *float ⇒ int*
where *mantissa f =*
fst (SOME p::int × int. (f = 0 ∧ fst p = 0 ∧ snd p = 0) ∨
*(f ≠ 0 ∧ real-of-float f = real-of-int (fst p) * 2 powr real-of-int (snd p) ∧ ¬ 2 dvd fst p))*

definition *exponent* :: *float ⇒ int*
where *exponent f =*
snd (SOME p::int × int. (f = 0 ∧ fst p = 0 ∧ snd p = 0) ∨
*(f ≠ 0 ∧ real-of-float f = real-of-int (fst p) * 2 powr real-of-int (snd p) ∧ ¬ 2 dvd fst p))*

lemma *exponent-0*[*simp*]: *exponent (float-of 0) = 0* (**is** *?E*)
and *mantissa-0*[*simp*]: *mantissa (float-of 0) = 0* (**is** *?M*)
⟨*proof*⟩

lemma *mantissa-exponent*: *real-of-float f = mantissa f * 2 powr exponent f* (**is** *?E*)
and *mantissa-not-dvd*: *f ≠ (float-of 0) ⟹ ¬ 2 dvd mantissa f* (**is** *- ⟹ ?D*)
⟨*proof*⟩

lemma *mantissa-noteq-0*: *f ≠ float-of 0 ⟹ mantissa f ≠ 0*
⟨*proof*⟩

lemma
fixes *m e* :: *int*
defines *f ≡ float-of (m * 2 powr e)*
assumes *dvd*: *¬ 2 dvd m*
shows *mantissa-float*: *mantissa f = m* (**is** *?M*)
and *exponent-float*: *m ≠ 0 ⟹ exponent f = e* (**is** *- ⟹ ?E*)
⟨*proof*⟩

36.5 Compute arithmetic operations

lemma *Float-mantissa-exponent*: *Float (mantissa f) (exponent f) = f*
⟨*proof*⟩

lemma *Float-cases* [*cases type*: *float*]:
fixes *f* :: *float*
obtains (*Float*) *m e* :: *int* **where** *f = Float m e*
⟨*proof*⟩

lemma *denormalize-shift*:
assumes *f-def*: *f ≡ Float m e*
and *not-0*: *f ≠ float-of 0*
obtains *i* **where** *m = mantissa f * 2 ^ i e = exponent f - i*
⟨*proof*⟩

context
begin

qualified lemma *compute-float-zero*[*code-unfold*, *code*]: *0 = Float 0 0*
⟨*proof*⟩ **lemma** *compute-float-one*[*code-unfold*, *code*]: *1 = Float 1 0*
⟨*proof*⟩

lift-definition *normfloat* :: *float ⇒ float* **is** *λx. x* ⟨*proof*⟩
lemma *normloat-id*[*simp*]: *normfloat x = x* ⟨*proof*⟩ **lemma** *compute-normfloat*[*code*]:
normfloat (Float m e) =
(if m mod 2 = 0 ∧ m ≠ 0 then normfloat (Float (m div 2) (e + 1))
else if m = 0 then 0 else Float m e)
⟨*proof*⟩ **lemma** *compute-float-numeral*[*code-abbrev*]: *Float (numeral k) 0 = numeral k*
⟨*proof*⟩ **lemma** *compute-float-neg-numeral*[*code-abbrev*]: *Float (- numeral k) 0 = - numeral k*

⟨*proof*⟩ **lemma** *compute-float-uminus*[*code*]: − *Float m1 e1* = *Float* (− *m1*) *e1*
⟨*proof*⟩ **lemma** *compute-float-times*[*code*]: *Float m1 e1* ∗ *Float m2 e2* = *Float* (*m1* ∗ *m2*) (*e1* + *e2*)
⟨*proof*⟩ **lemma** *compute-float-plus*[*code*]:
Float m1 e1 + *Float m2 e2* =
(*if m1* = *0 then Float m2 e2*
else if m2 = *0 then Float m1 e1*
else if e1 ≤ *e2 then Float* (*m1* + *m2* ∗ *2^nat* (*e2* − *e1*)) *e1*
else Float (*m2* + *m1* ∗ *2^nat* (*e1* − *e2*)) *e2*)
⟨*proof*⟩ **lemma** *compute-float-minus*[*code*]: *f* − *g* = *f* + (−*g*) **for** *f g* :: *float*
⟨*proof*⟩ **lemma** *compute-float-sgn*[*code*]:
sgn (*Float m1 e1*) = (*if* 0 < *m1 then* 1 *else if m1* < 0 *then* −1 *else* 0)
⟨*proof*⟩

lift-definition *is-float-pos* :: *float* ⇒ *bool* **is** *op* < 0 :: *real* ⇒ *bool* ⟨*proof*⟩ **lemma** *compute-is-float-pos*[*code*]: *is-float-pos* (*Float m e*) ⟷ 0 < *m*
⟨*proof*⟩

lift-definition *is-float-nonneg* :: *float* ⇒ *bool* **is** *op* ≤ 0 :: *real* ⇒ *bool* ⟨*proof*⟩ **lemma** *compute-is-float-nonneg*[*code*]: *is-float-nonneg* (*Float m e*) ⟷ 0 ≤ *m*
⟨*proof*⟩

lift-definition *is-float-zero* :: *float* ⇒ *bool* **is** *op* = 0 :: *real* ⇒ *bool* ⟨*proof*⟩ **lemma** *compute-is-float-zero*[*code*]: *is-float-zero* (*Float m e*) ⟷ 0 = *m*
⟨*proof*⟩ **lemma** *compute-float-abs*[*code*]: |*Float m e*| = *Float* |*m*| *e*
⟨*proof*⟩ **lemma** *compute-float-eq*[*code*]: *equal-class.equal f g* = *is-float-zero* (*f* − *g*)
⟨*proof*⟩

end

36.6 Lemmas for types *real*, *nat*, *int*

lemmas *real-of-ints* =
of-int-add
of-int-minus
of-int-diff
of-int-mult
of-int-power
of-int-numeral of-int-neg-numeral

lemmas *int-of-reals* = *real-of-ints*[*symmetric*]

36.7 Rounding Real Numbers

definition *round-down* :: *int* ⇒ *real* ⇒ *real*
where *round-down prec x* = ⌊*x* ∗ *2 powr prec*⌋ ∗ *2 powr* −*prec*

definition *round-up* :: *int* ⇒ *real* ⇒ *real*
where *round-up prec x* = ⌈*x* ∗ *2 powr prec*⌉ ∗ *2 powr* −*prec*

lemma *round-down-float*[*simp*]: *round-down prec x* ∈ *float*
⟨*proof*⟩

lemma *round-up-float*[*simp*]: *round-up prec x* ∈ *float*
⟨*proof*⟩

lemma *round-up*: *x* ≤ *round-up prec x*
⟨*proof*⟩

lemma *round-down*: *round-down prec x* ≤ *x*
⟨*proof*⟩

lemma *round-up-0*[*simp*]: *round-up p 0* = *0*
⟨*proof*⟩

lemma *round-down-0*[*simp*]: *round-down p 0* = *0*
⟨*proof*⟩

lemma *round-up-diff-round-down*: *round-up prec x* − *round-down prec x* ≤ *2 powr* −*prec*
⟨*proof*⟩

lemma *round-down-shift*: *round-down p* (*x* ∗ *2 powr k*) = *2 powr k* ∗ *round-down* (*p* + *k*) *x*
⟨*proof*⟩

lemma *round-up-shift*: *round-up p* (*x* ∗ *2 powr k*) = *2 powr k* ∗ *round-up* (*p* + *k*) *x*
⟨*proof*⟩

lemma *round-up-uminus-eq*: *round-up p* (−*x*) = − *round-down p x*
and *round-down-uminus-eq*: *round-down p* (−*x*) = − *round-up p x*
⟨*proof*⟩

lemma *round-up-mono*: *x* ≤ *y* ⟹ *round-up p x* ≤ *round-up p y*
⟨*proof*⟩

lemma *round-up-le1*:
assumes *x* ≤ *1 prec* ≥ *0*
shows *round-up prec x* ≤ *1*
⟨*proof*⟩

lemma *round-up-less1*:
assumes *x* < *1* / *2 p* > *0*
shows *round-up p x* < *1*
⟨*proof*⟩

lemma *round-down-ge1*:

assumes x: $x \geq 1$
assumes *prec*: $p \geq -\ log\ 2\ x$
shows $1 \leq$ *round-down* $p\ x$
⟨*proof*⟩

lemma *round-up-le0*: $x \leq 0 \Longrightarrow$ *round-up* $p\ x \leq 0$
⟨*proof*⟩

36.8 Rounding Floats

definition *div-twopow* :: *int* ⇒ *nat* ⇒ *int*
where [*simp*]: *div-twopow* $x\ n = x\ div\ (2\ \hat{}\ n)$

definition *mod-twopow* :: *int* ⇒ *nat* ⇒ *int*
where [*simp*]: *mod-twopow* $x\ n = x\ mod\ (2\ \hat{}\ n)$

lemma *compute-div-twopow*[*code*]:
div-twopow $x\ n = (if\ x = 0 \vee x = -1 \vee n = 0\ then\ x\ else\ div\text{-}twopow\ (x\ div\ 2)\ (n - 1))$
⟨*proof*⟩

lemma *compute-mod-twopow*[*code*]:
mod-twopow $x\ n = (if\ n = 0\ then\ 0\ else\ x\ mod\ 2 + 2 * mod\text{-}twopow\ (x\ div\ 2)\ (n - 1))$
⟨*proof*⟩

lift-definition *float-up* :: *int* ⇒ *float* ⇒ *float* **is** *round-up* ⟨*proof*⟩
declare *float-up.rep-eq*[*simp*]

lemma *round-up-correct*: *round-up* $e\ f - f \in \{0..2\ powr\ -e\}$
⟨*proof*⟩

lemma *float-up-correct*: *real-of-float* (*float-up* $e\ f$) − *real-of-float* $f \in \{0..2\ powr\ -e\}$
⟨*proof*⟩

lift-definition *float-down* :: *int* ⇒ *float* ⇒ *float* **is** *round-down* ⟨*proof*⟩
declare *float-down.rep-eq*[*simp*]

lemma *round-down-correct*: $f -$ (*round-down* $e\ f$) $\in \{0..2\ powr\ -e\}$
⟨*proof*⟩

lemma *float-down-correct*: *real-of-float* $f -$ *real-of-float* (*float-down* $e\ f$) $\in \{0..2\ powr\ -e\}$
⟨*proof*⟩

context
begin

qualified lemma *compute-float-down*[*code*]:
float-down p (*Float m e*) =
(*if p* + *e* < *0 then Float* (*div-twopow m* (*nat* (−(*p* + *e*)))) (−*p*) *else Float m e*)
⟨*proof*⟩

lemma *abs-round-down-le*: |*f* − (*round-down e f*)| ≤ *2 powr* −*e*
⟨*proof*⟩

lemma *abs-round-up-le*: |*f* − (*round-up e f*)| ≤ *2 powr* −*e*
⟨*proof*⟩

lemma *round-down-nonneg*: *0* ≤ *s* ⟹ *0* ≤ *round-down p s*
⟨*proof*⟩

lemma *ceil-divide-floor-conv*:
assumes *b* ≠ *0*
shows ⌈*real-of-int a* / *real-of-int b*⌉ =
(*if b dvd a then a div b else* ⌊*real-of-int a* / *real-of-int b*⌋ + *1*)
⟨*proof*⟩ **lemma** *compute-float-up*[*code*]: *float-up p x* = − *float-down p* (−*x*)
⟨*proof*⟩

end

lemma *bitlen-Float*:
fixes *m e*
defines *f* ≡ *Float m e*
shows *bitlen* |*mantissa f*| + *exponent f* = (*if m* = *0 then 0 else bitlen* |*m*| + *e*)
⟨*proof*⟩

lemma *float-gt1-scale*:
assumes *1* ≤ *Float m e*
shows *0* ≤ *e* + (*bitlen m* − *1*)
⟨*proof*⟩

36.9 Truncating Real Numbers

definition *truncate-down*::*nat* ⇒ *real* ⇒ *real*
where *truncate-down prec x* = *round-down* (*prec* − ⌊*log 2* |*x*|⌋) *x*

lemma *truncate-down*: *truncate-down prec x* ≤ *x*
⟨*proof*⟩

lemma *truncate-down-le*: *x* ≤ *y* ⟹ *truncate-down prec x* ≤ *y*
⟨*proof*⟩

lemma *truncate-down-zero*[*simp*]: *truncate-down prec 0* = *0*
⟨*proof*⟩

lemma *truncate-down-float*[*simp*]: *truncate-down p x* ∈ *float*
⟨*proof*⟩

definition *truncate-up*::*nat* ⇒ *real* ⇒ *real*
where *truncate-up prec x* = *round-up* (*prec* − ⌊*log 2* |*x*|⌋) *x*

lemma *truncate-up*: *x* ≤ *truncate-up prec x*
⟨*proof*⟩

lemma *truncate-up-le*: *x* ≤ *y* ⟹ *x* ≤ *truncate-up prec y*
⟨*proof*⟩

lemma *truncate-up-zero*[*simp*]: *truncate-up prec 0* = *0*
⟨*proof*⟩

lemma *truncate-up-uminus-eq*: *truncate-up prec* (−*x*) = − *truncate-down prec x*
and *truncate-down-uminus-eq*: *truncate-down prec* (−*x*) = − *truncate-up prec x*
⟨*proof*⟩

lemma *truncate-up-float*[*simp*]: *truncate-up p x* ∈ *float*
⟨*proof*⟩

lemma *mult-powr-eq*: *0* < *b* ⟹ *b* ≠ *1* ⟹ *0* < *x* ⟹ *x* ∗ *b powr y* = *b powr* (*y* + *log b x*)
⟨*proof*⟩

lemma *truncate-down-pos*:
assumes *x* > *0*
shows *truncate-down p x* > *0*
⟨*proof*⟩

lemma *truncate-down-nonneg*: *0* ≤ *y* ⟹ *0* ≤ *truncate-down prec y*
⟨*proof*⟩

lemma *truncate-down-ge1*: *1* ≤ *x* ⟹ *1* ≤ *truncate-down p x*
⟨*proof*⟩

lemma *truncate-up-nonpos*: *x* ≤ *0* ⟹ *truncate-up prec x* ≤ *0*
⟨*proof*⟩

lemma *truncate-up-le1*:
assumes *x* ≤ *1*
shows *truncate-up p x* ≤ *1*
⟨*proof*⟩

lemma *truncate-down-shift-int*:
truncate-down p (*x* ∗ *2 powr real-of-int k*) = *truncate-down p x* ∗ *2 powr k*
⟨*proof*⟩

lemma *truncate-down-shift-nat*: *truncate-down p* (*x* ∗ *2 powr real k*) = *truncate-down p x* ∗ *2 powr k*
⟨*proof*⟩

lemma *truncate-up-shift-int*: *truncate-up p* (*x* ∗ *2 powr real-of-int k*) = *truncate-up p x* ∗ *2 powr k*
⟨*proof*⟩

lemma *truncate-up-shift-nat*: *truncate-up p* (*x* ∗ *2 powr real k*) = *truncate-up p x* ∗ *2 powr k*
⟨*proof*⟩

36.10 Truncating Floats

lift-definition *float-round-up* :: *nat* ⇒ *float* ⇒ *float* **is** *truncate-up*
⟨*proof*⟩

lemma *float-round-up*: *real-of-float x* ≤ *real-of-float* (*float-round-up prec x*)
⟨*proof*⟩

lemma *float-round-up-zero*[*simp*]: *float-round-up prec 0* = *0*
⟨*proof*⟩

lift-definition *float-round-down* :: *nat* ⇒ *float* ⇒ *float* **is** *truncate-down*
⟨*proof*⟩

lemma *float-round-down*: *real-of-float* (*float-round-down prec x*) ≤ *real-of-float x*
⟨*proof*⟩

lemma *float-round-down-zero*[*simp*]: *float-round-down prec 0* = *0*
⟨*proof*⟩

lemmas *float-round-up-le* = *order-trans*[*OF - float-round-up*]
and *float-round-down-le* = *order-trans*[*OF float-round-down*]

lemma *minus-float-round-up-eq*: − *float-round-up prec x* = *float-round-down prec* (− *x*)
and *minus-float-round-down-eq*: − *float-round-down prec x* = *float-round-up prec* (− *x*)
⟨*proof*⟩

context
begin

qualified lemma *compute-float-round-down*[*code*]:
float-round-down prec (*Float m e*) =
(*let d* = *bitlen* |*m*| − *int prec* − *1 in*
if 0 < *d then Float* (*div-twopow m* (*nat d*)) (*e* + *d*)

else Float m e)
⟨*proof*⟩ **lemma** *compute-float-round-up*[*code*]:
float-round-up prec x = − *float-round-down prec* (−*x*)
⟨*proof*⟩

end

36.11 Approximation of positive rationals

lemma *div-mult-twopow-eq*: *a div* ((*2*::*nat*) ^ *n*) *div b* = *a div* (*b* ∗ *2* ^ *n*) **for** *a b* :: *nat*
⟨*proof*⟩

lemma *real-div-nat-eq-floor-of-divide*: *a div b* = *real-of-int* ⌊*a* / *b*⌋ **for** *a b* :: *nat*
⟨*proof*⟩

definition *rat-precision prec x y* =
(*let d* = *bitlen x* − *bitlen y*
in int prec − *d* + (*if Float* (*abs x*) *0* < *Float* (*abs y*) *d then 1 else 0*))

lemma *floor-log-divide-eq*:
assumes *i* > *0 j* > *0 p* > *1*
shows ⌊*log p* (*i* / *j*)⌋ = *floor* (*log p i*) − *floor* (*log p j*) −
(*if i* ≥ *j* ∗ *p powr* (*floor* (*log p i*) − *floor* (*log p j*)) *then 0 else 1*)
⟨*proof*⟩

lemma *truncate-down-rat-precision*:
truncate-down prec (*real x* / *real y*) = *round-down* (*rat-precision prec x y*) (*real x* / *real y*)
and *truncate-up-rat-precision*:
truncate-up prec (*real x* / *real y*) = *round-up* (*rat-precision prec x y*) (*real x* / *real y*)
⟨*proof*⟩

lift-definition *lapprox-posrat* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *float*
is λ*prec* (*x*::*nat*) (*y*::*nat*). *truncate-down prec* (*x* / *y*)
⟨*proof*⟩

context
begin

qualified lemma *compute-lapprox-posrat*[*code*]:
lapprox-posrat prec x y =
(*let*
l = *rat-precision prec x y*;
d = *if* 0 ≤ *l then x* ∗ *2*^*nat l div y else x div 2*^*nat* (− *l*) *div y*
in normfloat (*Float d* (− *l*)))
⟨*proof*⟩

end

lift-definition *rapprox-posrat* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *float*
is λ*prec* (*x*::*nat*) (*y*::*nat*). *truncate-up prec* (*x* / *y*)
⟨*proof*⟩

context
begin

qualified lemma *compute-rapprox-posrat*[*code*]:
fixes *prec x y*
defines *l* ≡ *rat-precision prec x y*
shows *rapprox-posrat prec x y* =
(*let*
l = *l*;
(*r*, *s*) = *if* 0 ≤ *l then* (*x* ∗ 2^*nat l*, *y*) *else* (*x*, *y* ∗ 2^*nat*(−*l*));
d = *r div s*;
m = *r mod s*
in normfloat (*Float* (*d* + (*if m* = 0 ∨ *y* = 0 *then* 0 *else* 1)) (− *l*)))
⟨*proof*⟩

end

lemma *rat-precision-pos*:
assumes 0 ≤ *x*
and 0 < *y*
and 2 ∗ *x* < *y*
shows *rat-precision n* (*int x*) (*int y*) > 0
⟨*proof*⟩

lemma *rapprox-posrat-less1*:
0 ≤ *x* ⟹ 0 < *y* ⟹ 2 ∗ *x* < *y* ⟹ *real-of-float* (*rapprox-posrat n x y*) < 1
⟨*proof*⟩

lift-definition *lapprox-rat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float* **is**
λ*prec* (*x*::*int*) (*y*::*int*). *truncate-down prec* (*x* / *y*)
⟨*proof*⟩

context
begin

qualified lemma *compute-lapprox-rat*[*code*]:
lapprox-rat prec x y =
(*if y* = 0 *then* 0
else if 0 ≤ *x then*
(*if* 0 < *y then lapprox-posrat prec* (*nat x*) (*nat y*)
else − (*rapprox-posrat prec* (*nat x*) (*nat* (−*y*))))
else
(*if* 0 < *y*

then − (*rapprox-posrat prec* (*nat* (−*x*)) (*nat y*))
else lapprox-posrat prec (*nat* (−*x*)) (*nat* (−*y*))))
⟨*proof*⟩

lift-definition *rapprox-rat* :: *nat* ⇒ *int* ⇒ *int* ⇒ *float* **is**
λ*prec* (*x*::*int*) (*y*::*int*). *truncate-up prec* (*x* / *y*)
⟨*proof*⟩

lemma *rapprox-rat* = *rapprox-posrat*
⟨*proof*⟩

lemma *lapprox-rat* = *lapprox-posrat*
⟨*proof*⟩ **lemma** *compute-rapprox-rat*[*code*]:
rapprox-rat prec x y = − *lapprox-rat prec* (−*x*) *y*
⟨*proof*⟩ **lemma** *compute-truncate-down*[*code*]:
truncate-down p (*Ratreal r*) = (*let* (*a*, *b*) = *quotient-of r in lapprox-rat p a b*)
⟨*proof*⟩ **lemma** *compute-truncate-up*[*code*]:
truncate-up p (*Ratreal r*) = (*let* (*a*, *b*) = *quotient-of r in rapprox-rat p a b*)
⟨*proof*⟩

end

36.12 Division

definition *real-divl prec a b* = *truncate-down prec* (*a* / *b*)

definition *real-divr prec a b* = *truncate-up prec* (*a* / *b*)

lift-definition *float-divl* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *real-divl*
⟨*proof*⟩

context
begin

qualified lemma *compute-float-divl*[*code*]:
float-divl prec (*Float m1 s1*) (*Float m2 s2*) = *lapprox-rat prec m1 m2* ∗ *Float 1* (*s1* − *s2*)
⟨*proof*⟩

lift-definition *float-divr* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *real-divr*
⟨*proof*⟩ **lemma** *compute-float-divr*[*code*]:
float-divr prec x y = − *float-divl prec* (−*x*) *y*
⟨*proof*⟩

end

36.13 Approximate Power

lemma *div2-less-self*[*termination-simp*]: *odd n* ⟹ *n div 2* < *n* **for** *n* :: *nat*
⟨*proof*⟩

fun *power-down* :: *nat* ⇒ *real* ⇒ *nat* ⇒ *real*
where
power-down p x 0 = 1
| *power-down p x (Suc n) =*
(if odd n then truncate-down (Suc p) ((power-down p x (Suc n div 2))²)
*else truncate-down (Suc p) (x * power-down p x n))*

fun *power-up* :: *nat* ⇒ *real* ⇒ *nat* ⇒ *real*
where
power-up p x 0 = 1
| *power-up p x (Suc n) =*
(if odd n then truncate-up p ((power-up p x (Suc n div 2))²)
*else truncate-up p (x * power-up p x n))*

lift-definition *power-up-fl* :: *nat* ⇒ *float* ⇒ *nat* ⇒ *float* **is** *power-up*
⟨*proof*⟩

lift-definition *power-down-fl* :: *nat* ⇒ *float* ⇒ *nat* ⇒ *float* **is** *power-down*
⟨*proof*⟩

lemma *power-float-transfer*[*transfer-rule*]:
(rel-fun pcr-float (rel-fun op = pcr-float)) op ^ op ^
⟨*proof*⟩

lemma *compute-power-up-fl*[*code*]:
power-up-fl p x 0 = 1
power-up-fl p x (Suc n) =
(if odd n then float-round-up p ((power-up-fl p x (Suc n div 2))²)
*else float-round-up p (x * power-up-fl p x n))*
and *compute-power-down-fl*[*code*]:
power-down-fl p x 0 = 1
power-down-fl p x (Suc n) =
(if odd n then float-round-down (Suc p) ((power-down-fl p x (Suc n div 2))²)
*else float-round-down (Suc p) (x * power-down-fl p x n))*
⟨*proof*⟩

lemma *power-down-pos*: *0 < x ⟹ 0 < power-down p x n*
⟨*proof*⟩

lemma *power-down-nonneg*: *0 ≤ x ⟹ 0 ≤ power-down p x n*
⟨*proof*⟩

lemma *power-down*: *0 ≤ x ⟹ power-down p x n ≤ x ^ n*
⟨*proof*⟩

lemma *power-up*: *0 ≤ x ⟹ x ^ n ≤ power-up p x n*
⟨*proof*⟩

lemmas *power-up-le* = *order-trans*[*OF* - *power-up*]
and *power-up-less* = *less-le-trans*[*OF* - *power-up*]
and *power-down-le* = *order-trans*[*OF power-down*]

lemma *power-down-fl*: $0 \leq x \Longrightarrow$ *power-down-fl p x n* $\leq x \hat{} n$
⟨*proof*⟩

lemma *power-up-fl*: $0 \leq x \Longrightarrow x \hat{} n \leq$ *power-up-fl p x n*
⟨*proof*⟩

lemma *real-power-up-fl*: *real-of-float* (*power-up-fl p x n*) = *power-up p x n*
⟨*proof*⟩

lemma *real-power-down-fl*: *real-of-float* (*power-down-fl p x n*) = *power-down p x n*
⟨*proof*⟩

36.14 Approximate Addition

definition *plus-down prec x y* = *truncate-down prec* (*x* + *y*)

definition *plus-up prec x y* = *truncate-up prec* (*x* + *y*)

lemma *float-plus-down-float*[*intro*, *simp*]: $x \in$ *float* $\Longrightarrow y \in$ *float* $\Longrightarrow$ *plus-down p x y* $\in$ *float*
⟨*proof*⟩

lemma *float-plus-up-float*[*intro*, *simp*]: $x \in$ *float* $\Longrightarrow y \in$ *float* $\Longrightarrow$ *plus-up p x y* $\in$ *float*
⟨*proof*⟩

lift-definition *float-plus-down* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *plus-down* ⟨*proof*⟩

lift-definition *float-plus-up* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *plus-up* ⟨*proof*⟩

lemma *plus-down*: *plus-down prec x y* $\leq x + y$
and *plus-up*: $x + y \leq$ *plus-up prec x y*
⟨*proof*⟩

lemma *float-plus-down*: *real-of-float* (*float-plus-down prec x y*) $\leq x + y$
and *float-plus-up*: $x + y \leq$ *real-of-float* (*float-plus-up prec x y*)
⟨*proof*⟩

lemmas *plus-down-le* = *order-trans*[*OF plus-down*]
and *plus-up-le* = *order-trans*[*OF* - *plus-up*]
and *float-plus-down-le* = *order-trans*[*OF float-plus-down*]
and *float-plus-up-le* = *order-trans*[*OF* - *float-plus-up*]

lemma *compute-plus-up*[*code*]: *plus-up p x y* = − *plus-down p* (−*x*) (−*y*)

⟨*proof*⟩

lemma *truncate-down-log2-eqI*:
assumes $\lfloor log\ 2\ |x| \rfloor = \lfloor log\ 2\ |y| \rfloor$
assumes $\lfloor x * 2\ powr\ (p - \lfloor log\ 2\ |x| \rfloor) \rfloor = \lfloor y * 2\ powr\ (p - \lfloor log\ 2\ |x| \rfloor) \rfloor$
shows *truncate-down p x* = *truncate-down p y*
⟨*proof*⟩

lemma *sum-neq-zeroI*:
$|a| \geq k \Longrightarrow |b| < k \Longrightarrow a + b \neq 0$
$|a| > k \Longrightarrow |b| \leq k \Longrightarrow a + b \neq 0$
for *a k* :: *real*
⟨*proof*⟩

lemma *abs-real-le-2-powr-bitlen*[*simp*]: $|real\text{-}of\text{-}int\ m2| < 2\ powr\ real\text{-}of\text{-}int\ (bitlen\ |m2|)$
⟨*proof*⟩

lemma *floor-sum-times-2-powr-sgn-eq*:
fixes *ai p q* :: *int*
and *a b* :: *real*
assumes $a * 2\ powr\ p = ai$
and *b-le-1*: $|b * 2\ powr\ (p + 1)| \leq 1$
and *leqp*: $q \leq p$
shows $\lfloor (a + b) * 2\ powr\ q \rfloor = \lfloor (2 * ai + sgn\ b) * 2\ powr\ (q - p - 1) \rfloor$
⟨*proof*⟩

lemma *log2-abs-int-add-less-half-sgn-eq*:
fixes *ai* :: *int*
and *b* :: *real*
assumes $|b| \leq 1/2$
and $ai \neq 0$
shows $\lfloor log\ 2\ |real\text{-}of\text{-}int\ ai + b| \rfloor = \lfloor log\ 2\ |ai + sgn\ b\ /\ 2| \rfloor$
⟨*proof*⟩

context
begin

qualified lemma *compute-far-float-plus-down*:
fixes *m1 e1 m2 e2* :: *int*
and *p* :: *nat*
defines $k1 \equiv Suc\ p - nat\ (bitlen\ |m1|)$
assumes *H*: $bitlen\ |m2| \leq e1 - e2 - k1 - 2\ m1 \neq 0\ m2 \neq 0\ e1 \geq e2$
shows *float-plus-down p* (*Float m1 e1*) (*Float m2 e2*) =
float-round-down p (*Float* $(m1 * 2\ \hat{}\ (Suc\ (Suc\ k1)) + sgn\ m2)\ (e1 - int\ k1 - 2)$)
⟨*proof*⟩

lemma *compute-float-plus-down-naive*[*code*]: *float-plus-down p x y* = *float-round-down*

$p\ (x + y)$
⟨*proof*⟩ **lemma** *compute-float-plus-down*[*code*]:
fixes *p*::*nat* **and** *m1 e1 m2 e2*::*int*
shows *float-plus-down p* (*Float m1 e1*) (*Float m2 e2*) =
(*if m1 = 0 then float-round-down p* (*Float m2 e2*)
else if m2 = 0 then float-round-down p (*Float m1 e1*)
else
(*if e1 ≥ e2 then*
(*let k1 = Suc p − nat* (*bitlen* |*m1*|) *in*
if bitlen |*m2*| > *e1 − e2 − k1 − 2*
then float-round-down p ((*Float m1 e1*) + (*Float m2 e2*))
else float-round-down p (*Float* (*m1 ∗ 2 ^* (*Suc* (*Suc k1*)) + *sgn m2*) (*e1 − int k1 − 2*)))
else float-plus-down p (*Float m2 e2*) (*Float m1 e1*)))
⟨*proof*⟩ **lemma** *compute-float-plus-up*[*code*]: *float-plus-up p x y* = − *float-plus-down p* (−*x*) (−*y*)
⟨*proof*⟩

lemma *mantissa-zero*[*simp*]: *mantissa 0* = *0*
⟨*proof*⟩ **lemma** *compute-float-less*[*code*]: *a* < *b* ⟷ *is-float-pos* (*float-plus-down 0 b* (− *a*))
⟨*proof*⟩ **lemma** *compute-float-le*[*code*]: *a* ≤ *b* ⟷ *is-float-nonneg* (*float-plus-down 0 b* (− *a*))
⟨*proof*⟩

end

36.15 Lemmas needed by Approximate

lemma *Float-num*[*simp*]:
real-of-float (*Float 1 0*) = *1*
real-of-float (*Float 1 1*) = *2*
real-of-float (*Float 1 2*) = *4*
real-of-float (*Float 1* (− *1*)) = *1/2*
real-of-float (*Float 1* (− *2*)) = *1/4*
real-of-float (*Float 1* (− *3*)) = *1/8*
real-of-float (*Float* (− *1*) *0*) = −*1*
real-of-float (*Float* (*numeral n*) *0*) = *numeral n*
real-of-float (*Float* (− *numeral n*) *0*) = − *numeral n*
⟨*proof*⟩

lemma *real-of-Float-int*[*simp*]: *real-of-float* (*Float n 0*) = *real n*
⟨*proof*⟩

lemma *float-zero*[*simp*]: *real-of-float* (*Float 0 e*) = *0*
⟨*proof*⟩

lemma *abs-div-2-less*: *a* ≠ *0* ⟹ *a* ≠ −*1* ⟹ |(*a*::*int*) *div 2*| < |*a*|
⟨*proof*⟩

lemma *lapprox-rat*: *real-of-float* (*lapprox-rat prec x y*) ≤ *real-of-int x* / *real-of-int y*
⟨*proof*⟩

lemma *mult-div-le*:
fixes *a b* :: *int*
assumes *b* > *0*
shows *a* ≥ *b* ∗ (*a div b*)
⟨*proof*⟩

lemma *lapprox-rat-nonneg*:
assumes *0* ≤ *x* **and** *0* ≤ *y*
shows *0* ≤ *real-of-float* (*lapprox-rat n x y*)
⟨*proof*⟩

lemma *rapprox-rat*: *real-of-int x* / *real-of-int y* ≤ *real-of-float* (*rapprox-rat prec x y*)
⟨*proof*⟩

lemma *rapprox-rat-le1*:
assumes *0* ≤ *x 0* < *y x* ≤ *y*
shows *real-of-float* (*rapprox-rat n x y*) ≤ *1*
⟨*proof*⟩

lemma *rapprox-rat-nonneg-nonpos*: *0* ≤ *x* ⟹ *y* ≤ *0* ⟹ *real-of-float* (*rapprox-rat n x y*) ≤ *0*
⟨*proof*⟩

lemma *rapprox-rat-nonpos-nonneg*: *x* ≤ *0* ⟹ *0* ≤ *y* ⟹ *real-of-float* (*rapprox-rat n x y*) ≤ *0*
⟨*proof*⟩

lemma *real-divl*: *real-divl prec x y* ≤ *x* / *y*
⟨*proof*⟩

lemma *real-divr*: *x* / *y* ≤ *real-divr prec x y*
⟨*proof*⟩

lemma *float-divl*: *real-of-float* (*float-divl prec x y*) ≤ *x* / *y*
⟨*proof*⟩

lemma *real-divl-lower-bound*: *0* ≤ *x* ⟹ *0* ≤ *y* ⟹ *0* ≤ *real-divl prec x y*
⟨*proof*⟩

lemma *float-divl-lower-bound*: *0* ≤ *x* ⟹ *0* ≤ *y* ⟹ *0* ≤ *real-of-float* (*float-divl prec x y*)
⟨*proof*⟩

lemma *exponent-1*: *exponent 1 = 0*
⟨*proof*⟩

lemma *mantissa-1*: *mantissa 1 = 1*
⟨*proof*⟩

lemma *bitlen-1*: *bitlen 1 = 1*
⟨*proof*⟩

lemma *mantissa-eq-zero-iff*: *mantissa x = 0 ⟷ x = 0*
(**is** *?lhs ⟷ ?rhs*)
⟨*proof*⟩

lemma *float-upper-bound*: *x ≤ 2 powr (bitlen |mantissa x| + exponent x)*
⟨*proof*⟩

lemma *real-divl-pos-less1-bound*:
assumes *0 < x x ≤ 1*
shows *1 ≤ real-divl prec 1 x*
⟨*proof*⟩

lemma *float-divl-pos-less1-bound*:
0 < real-of-float x ⟹ real-of-float x ≤ 1 ⟹ prec ≥ 1 ⟹
1 ≤ real-of-float (float-divl prec 1 x)
⟨*proof*⟩

lemma *float-divr*: *real-of-float x / real-of-float y ≤ real-of-float (float-divr prec x y)*
⟨*proof*⟩

lemma *real-divr-pos-less1-lower-bound*:
assumes *0 < x*
and *x ≤ 1*
shows *1 ≤ real-divr prec 1 x*
⟨*proof*⟩

lemma *float-divr-pos-less1-lower-bound*: *0 < x ⟹ x ≤ 1 ⟹ 1 ≤ float-divr prec 1 x*
⟨*proof*⟩

lemma *real-divr-nonpos-pos-upper-bound*: *x ≤ 0 ⟹ 0 ≤ y ⟹ real-divr prec x y ≤ 0*
⟨*proof*⟩

lemma *float-divr-nonpos-pos-upper-bound*:
real-of-float x ≤ 0 ⟹ 0 ≤ real-of-float y ⟹ real-of-float (float-divr prec x y) ≤ 0
⟨*proof*⟩

lemma *real-divr-nonneg-neg-upper-bound*: $0 \leq x \Longrightarrow y \leq 0 \Longrightarrow$ *real-divr prec x y* ≤ 0
⟨*proof*⟩

lemma *float-divr-nonneg-neg-upper-bound*:
$0 \leq$ *real-of-float x* $\Longrightarrow$ *real-of-float y* $\leq 0 \Longrightarrow$ *real-of-float* (*float-divr prec x y*) ≤ 0
⟨*proof*⟩

lemma *truncate-up-nonneg-mono*:
assumes $0 \leq x$ $x \leq y$
shows *truncate-up prec x* $\leq$ *truncate-up prec y*
⟨*proof*⟩

lemma *truncate-up-switch-sign-mono*:
assumes $x \leq 0$ $0 \leq y$
shows *truncate-up prec x* $\leq$ *truncate-up prec y*
⟨*proof*⟩

lemma *truncate-down-switch-sign-mono*:
assumes $x \leq 0$
and $0 \leq y$
and $x \leq y$
shows *truncate-down prec x* $\leq$ *truncate-down prec y*
⟨*proof*⟩

lemma *truncate-down-nonneg-mono*:
assumes $0 \leq x$ $x \leq y$
shows *truncate-down prec x* $\leq$ *truncate-down prec y*
⟨*proof*⟩

lemma *truncate-down-eq-truncate-up*: *truncate-down p x* $= -$ *truncate-up p* $(-x)$
and *truncate-up-eq-truncate-down*: *truncate-up p x* $= -$ *truncate-down p* $(-x)$
⟨*proof*⟩

lemma *truncate-down-mono*: $x \leq y \Longrightarrow$ *truncate-down p x* $\leq$ *truncate-down p y*
⟨*proof*⟩

lemma *truncate-up-mono*: $x \leq y \Longrightarrow$ *truncate-up p x* $\leq$ *truncate-up p y*
⟨*proof*⟩

lemma *Float-le-zero-iff*: *Float a b* $\leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *real-of-float-pprt*[*simp*]:
fixes *a* :: *float*
shows *real-of-float* (*pprt a*) $=$ *pprt* (*real-of-float a*)
⟨*proof*⟩

lemma *real-of-float-nprt*[*simp*]:
fixes *a* :: *float*
shows *real-of-float* (*nprt a*) = *nprt* (*real-of-float a*)
⟨*proof*⟩

context
begin

lift-definition *int-floor-fl* :: *float* ⇒ *int* **is** *floor* ⟨*proof*⟩ **lemma** *compute-int-floor-fl*[*code*]:
int-floor-fl (*Float m e*) = (*if* 0 ≤ *e then m* ∗ 2 ^ *nat e else m div* (2 ^ (*nat* (−*e*))))
⟨*proof*⟩

lift-definition *floor-fl* :: *float* ⇒ *float* **is** λ*x*. *real-of-int* ⌊*x*⌋
⟨*proof*⟩ **lemma** *compute-floor-fl*[*code*]:
floor-fl (*Float m e*) = (*if* 0 ≤ *e then Float m e else Float* (*m div* (2 ^ (*nat* (−*e*)))) 0)
⟨*proof*⟩

end

lemma *floor-fl*: *real-of-float* (*floor-fl x*) ≤ *real-of-float x*
⟨*proof*⟩

lemma *int-floor-fl*: *real-of-int* (*int-floor-fl x*) ≤ *real-of-float x*
⟨*proof*⟩

lemma *floor-pos-exp*: *exponent* (*floor-fl x*) ≥ 0
⟨*proof*⟩

lemma *compute-mantissa*[*code*]:
mantissa (*Float m e*) =
(*if m* = 0 *then* 0 *else if* 2 *dvd m then mantissa* (*normfloat* (*Float m e*)) *else m*)
⟨*proof*⟩

lemma *compute-exponent*[*code*]:
exponent (*Float m e*) =
(*if m* = 0 *then* 0 *else if* 2 *dvd m then exponent* (*normfloat* (*Float m e*)) *else e*)
⟨*proof*⟩

end

37 Pi and Function Sets

theory *FuncSet*
imports *HOL.Hilbert-Choice Main*
abbrevs *PiE* = Pi_E
PIE = Π_E
begin

definition $Pi :: 'a\ set \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow ('a \Rightarrow 'b)\ set$
where $Pi\ A\ B = \{f.\ \forall x.\ x \in A \longrightarrow f\ x \in B\ x\}$

definition $extensional :: 'a\ set \Rightarrow ('a \Rightarrow 'b)\ set$
where $extensional\ A = \{f.\ \forall x.\ x \notin A \longrightarrow f\ x = undefined\}$

definition $restrict :: ('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow 'b$
where $restrict\ f\ A = (\lambda x.\ if\ x \in A\ then\ f\ x\ else\ undefined)$

abbreviation $funcset :: 'a\ set \Rightarrow 'b\ set \Rightarrow ('a \Rightarrow 'b)\ set$ (**infixr** $\rightarrow$ *60*)
where $A \rightarrow B \equiv Pi\ A\ (\lambda\text{-}.\ B)$

syntax
$\text{-}Pi :: pttrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow ('a \Rightarrow 'b)\ set$ (*(3*Π -$\in$-./ -) *10*)
$\text{-}lam :: pttrn \Rightarrow 'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$ (*(3*λ-$\in$-./ -) *[0,0,3] 3*)
translations
$\Pi\ x{\in}A.\ B \rightleftharpoons CONST\ Pi\ A\ (\lambda x.\ B)$
$\lambda x{\in}A.\ f \rightleftharpoons CONST\ restrict\ (\lambda x.\ f)\ A$

definition $compose :: 'a\ set \Rightarrow ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'c)$
where $compose\ A\ g\ f = (\lambda x{\in}A.\ g\ (f\ x))$

37.1 Basic Properties of *Pi*

lemma *Pi-I*[*intro*!]: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow f \in Pi\ A\ B$
⟨*proof*⟩

lemma *Pi-I'*[*simp*]: $(\bigwedge x.\ x \in A \longrightarrow f\ x \in B\ x) \Longrightarrow f \in Pi\ A\ B$
⟨*proof*⟩

lemma *funcsetI*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow f \in A \rightarrow B$
⟨*proof*⟩

lemma *Pi-mem*: $f \in Pi\ A\ B \Longrightarrow x \in A \Longrightarrow f\ x \in B\ x$
⟨*proof*⟩

lemma *Pi-iff*: $f \in Pi\ I\ X \longleftrightarrow (\forall i{\in}I.\ f\ i \in X\ i)$
⟨*proof*⟩

lemma *PiE* [*elim*]: $f \in Pi\ A\ B \Longrightarrow (f\ x \in B\ x \Longrightarrow Q) \Longrightarrow (x \notin A \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *Pi-cong*: $(\bigwedge w.\ w \in A \Longrightarrow f\ w = g\ w) \Longrightarrow f \in Pi\ A\ B \longleftrightarrow g \in Pi\ A\ B$
⟨*proof*⟩

lemma *funcset-id* [*simp*]: $(\lambda x.\ x) \in A \rightarrow A$
⟨*proof*⟩

lemma *funcset-mem*: $f \in A \to B \Longrightarrow x \in A \Longrightarrow f\ x \in B$
⟨*proof*⟩

lemma *funcset-image*: $f \in A \to B \Longrightarrow f\ `\ A \subseteq B$
⟨*proof*⟩

lemma *image-subset-iff-funcset*: $F\ `\ A \subseteq B \longleftrightarrow F \in A \to B$
⟨*proof*⟩

lemma *Pi-eq-empty*[*simp*]: $(\Pi\ x \in A.\ B\ x) = \{\} \longleftrightarrow (\exists\, x{\in}A.\ B\ x = \{\})$
⟨*proof*⟩

lemma *Pi-empty* [*simp*]: $Pi\ \{\}\ B = UNIV$
⟨*proof*⟩

lemma *Pi-Int*: $Pi\ I\ E \cap Pi\ I\ F = (\Pi\ i{\in}I.\ E\ i \cap F\ i)$
⟨*proof*⟩

lemma *Pi-UN*:
fixes $A :: nat \Rightarrow 'i \Rightarrow 'a\ set$
assumes *finite I*
and *mono*: $\bigwedge i\ n\ m.\ i \in I \Longrightarrow n \le m \Longrightarrow A\ n\ i \subseteq A\ m\ i$
shows $(\bigcup n.\ Pi\ I\ (A\ n)) = (\Pi\ i{\in}I.\ \bigcup n.\ A\ n\ i)$
⟨*proof*⟩

lemma *Pi-UNIV* [*simp*]: $A \to UNIV = UNIV$
⟨*proof*⟩

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x \subseteq C\ x) \Longrightarrow Pi\ A\ B \subseteq Pi\ A\ C$
⟨*proof*⟩

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: $A' \subseteq A \Longrightarrow Pi\ A\ B \subseteq Pi\ A'\ B$
⟨*proof*⟩

lemma *prod-final*:
assumes *1*: $fst \circ f \in Pi\ A\ B$
and *2*: $snd \circ f \in Pi\ A\ C$
shows $f \in (\Pi\ z \in A.\ B\ z \times C\ z)$
⟨*proof*⟩

lemma *Pi-split-domain*[*simp*]: $x \in Pi\ (I \cup J)\ X \longleftrightarrow x \in Pi\ I\ X \wedge x \in Pi\ J\ X$
⟨*proof*⟩

lemma *Pi-split-insert-domain*[*simp*]: $x \in Pi\ (insert\ i\ I)\ X \longleftrightarrow x \in Pi\ I\ X \wedge x\ i \in X\ i$
⟨*proof*⟩

lemma *Pi-cancel-fupd-range*[*simp*]: $i \notin I \Longrightarrow x \in Pi\ I\ (B(i := b)) \longleftrightarrow x \in Pi\ I\ B$
⟨*proof*⟩

lemma *Pi-cancel-fupd*[*simp*]: $i \notin I \Longrightarrow x(i := a) \in Pi\ I\ B \longleftrightarrow x \in Pi\ I\ B$
⟨*proof*⟩

lemma *Pi-fupd-iff*: $i \in I \Longrightarrow f \in Pi\ I\ (B(i := A)) \longleftrightarrow f \in Pi\ (I - \{i\})\ B \wedge f\ i \in A$
⟨*proof*⟩

37.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*: $f \in A \to B \Longrightarrow g \in B \to C \Longrightarrow compose\ A\ g\ f \in A \to C$
⟨*proof*⟩

lemma *compose-assoc*:
assumes $f \in A \to B$
and $g \in B \to C$
and $h \in C \to D$
shows $compose\ A\ h\ (compose\ A\ g\ f) = compose\ A\ (compose\ B\ h\ g)\ f$
⟨*proof*⟩

lemma *compose-eq*: $x \in A \Longrightarrow compose\ A\ g\ f\ x = g\ (f\ x)$
⟨*proof*⟩

lemma *surj-compose*: $f\ `\ A = B \Longrightarrow g\ `\ B = C \Longrightarrow compose\ A\ g\ f\ `\ A = C$
⟨*proof*⟩

37.3 Bounded Abstraction: *restrict*

lemma *restrict-cong*: $I = J \Longrightarrow (\bigwedge i.\ i \in J =simp=> f\ i = g\ i) \Longrightarrow restrict\ f\ I = restrict\ g\ J$
⟨*proof*⟩

lemma *restrict-in-funcset*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow (\lambda x{\in}A.\ f\ x) \in A \to B$
⟨*proof*⟩

lemma *restrictI*[*intro!*]: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow (\lambda x{\in}A.\ f\ x) \in Pi\ A\ B$
⟨*proof*⟩

lemma *restrict-apply*[*simp*]: $(\lambda y{\in}A.\ f\ y)\ x = (if\ x \in A\ then\ f\ x\ else\ undefined)$
⟨*proof*⟩

lemma *restrict-apply'*: $x \in A \Longrightarrow (\lambda y{\in}A.\ f\ y)\ x = f\ x$
⟨*proof*⟩

lemma *restrict-ext*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x) \Longrightarrow (\lambda x{\in}A.\ f\ x) = (\lambda x{\in}A.\ g\ x)$
⟨*proof*⟩

lemma *restrict-UNIV*: *restrict f UNIV* = *f*
⟨*proof*⟩

lemma *inj-on-restrict-eq* [*simp*]: *inj-on* (*restrict f A*) *A* = *inj-on f A*
⟨*proof*⟩

lemma *Id-compose*: $f \in A \to B \Longrightarrow f \in$ *extensional A* $\Longrightarrow$ *compose A* ($\lambda y{\in}B.\ y$) *f* = *f*
⟨*proof*⟩

lemma *compose-Id*: $g \in A \to B \Longrightarrow g \in$ *extensional A* $\Longrightarrow$ *compose A g* ($\lambda x{\in}A.$ *x*) = *g*
⟨*proof*⟩

lemma *image-restrict-eq* [*simp*]: (*restrict f A*) ‘ *A* = *f* ‘ *A*
⟨*proof*⟩

lemma *restrict-restrict*[*simp*]: *restrict* (*restrict f A*) *B* = *restrict f* (*A* ∩ *B*)
⟨*proof*⟩

lemma *restrict-fupd*[*simp*]: *i* ∉ *I* $\Longrightarrow$ *restrict* (*f* (*i* := *x*)) *I* = *restrict f I*
⟨*proof*⟩

lemma *restrict-upd*[*simp*]: *i* ∉ *I* $\Longrightarrow$ (*restrict f I*)(*i* := *y*) = *restrict* (*f*(*i* := *y*)) (*insert i I*)
⟨*proof*⟩

lemma *restrict-Pi-cancel*: *restrict x I* ∈ *Pi I A* ⟷ *x* ∈ *Pi I A*
⟨*proof*⟩

37.4 Bijections Between Sets

The definition of *bij-betw* is in *Fun.thy*, but most of the theorems belong here, or need at least *Hilbert-Choice*.

lemma *bij-betwI*:
assumes *f* ∈ *A* → *B*
and *g* ∈ *B* → *A*
and *g-f*: $\bigwedge x.\ x{\in}A \Longrightarrow g\ (f\ x) = x$
and *f-g*: $\bigwedge y.\ y{\in}B \Longrightarrow f\ (g\ y) = y$
shows *bij-betw f A B*
⟨*proof*⟩

lemma *bij-betw-imp-funcset*: *bij-betw f A B* $\Longrightarrow$ *f* ∈ *A* → *B*
⟨*proof*⟩

lemma *inj-on-compose*: *bij-betw f A B* $\Longrightarrow$ *inj-on g B* $\Longrightarrow$ *inj-on* (*compose A g f*) *A*
⟨*proof*⟩

lemma *bij-betw-compose*: *bij-betw f A B* $\Longrightarrow$ *bij-betw g B C* $\Longrightarrow$ *bij-betw* (*compose A g f*) *A C*
⟨*proof*⟩

lemma *bij-betw-restrict-eq* [*simp*]: *bij-betw* (*restrict f A*) *A B* = *bij-betw f A B*
⟨*proof*⟩

37.5 Extensionality

lemma *extensional-empty*[*simp*]: *extensional* {} = {$\lambda x.$ *undefined*}
⟨*proof*⟩

lemma *extensional-arb*: *f* $\in$ *extensional A* $\Longrightarrow$ *x* $\notin$ *A* $\Longrightarrow$ *f x* = *undefined*
⟨*proof*⟩

lemma *restrict-extensional* [*simp*]: *restrict f A* $\in$ *extensional A*
⟨*proof*⟩

lemma *compose-extensional* [*simp*]: *compose A f g* $\in$ *extensional A*
⟨*proof*⟩

lemma *extensionalityI*:
assumes *f* $\in$ *extensional A*
and *g* $\in$ *extensional A*
and $\bigwedge x.$ *x* $\in$ *A* $\Longrightarrow$ *f x* = *g x*
shows *f* = *g*
⟨*proof*⟩

lemma *extensional-restrict*: *f* $\in$ *extensional A* $\Longrightarrow$ *restrict f A* = *f*
⟨*proof*⟩

lemma *extensional-subset*: *f* $\in$ *extensional A* $\Longrightarrow$ *A* $\subseteq$ *B* $\Longrightarrow$ *f* $\in$ *extensional B*
⟨*proof*⟩

lemma *inv-into-funcset*: *f ' A* = *B* $\Longrightarrow$ ($\lambda x \in B.$ *inv-into A f x*) $\in$ *B* $\rightarrow$ *A*
⟨*proof*⟩

lemma *compose-inv-into-id*: *bij-betw f A B* $\Longrightarrow$ *compose A* ($\lambda y \in B.$ *inv-into A f y*) *f* = ($\lambda x \in A.$ *x*)
⟨*proof*⟩

lemma *compose-id-inv-into*: *f ' A* = *B* $\Longrightarrow$ *compose B f* ($\lambda y \in B.$ *inv-into A f y*) = ($\lambda x \in B.$ *x*)
⟨*proof*⟩

lemma *extensional-insert*[*intro*, *simp*]:
assumes *a* $\in$ *extensional* (*insert i I*)
shows *a*(*i* := *b*) $\in$ *extensional* (*insert i I*)

⟨*proof*⟩

lemma *extensional-Int*[*simp*]: *extensional I* ∩ *extensional I′* = *extensional* (*I* ∩ *I′*)
⟨*proof*⟩

lemma *extensional-UNIV*[*simp*]: *extensional UNIV* = *UNIV*
⟨*proof*⟩

lemma *restrict-extensional-sub*[*intro*]: *A* ⊆ *B* ⟹ *restrict f A* ∈ *extensional B*
⟨*proof*⟩

lemma *extensional-insert-undefined*[*intro*, *simp*]:
a ∈ *extensional* (*insert i I*) ⟹ *a*(*i* := *undefined*) ∈ *extensional I*
⟨*proof*⟩

lemma *extensional-insert-cancel*[*intro*, *simp*]:
a ∈ *extensional I* ⟹ *a* ∈ *extensional* (*insert i I*)
⟨*proof*⟩

37.6 Cardinality

lemma *card-inj*: *f* ∈ *A* → *B* ⟹ *inj-on f A* ⟹ *finite B* ⟹ *card A* ≤ *card B*
⟨*proof*⟩

lemma *card-bij*:
assumes *f* ∈ *A* → *B inj-on f A*
and *g* ∈ *B* → *A inj-on g B*
and *finite A finite B*
shows *card A* = *card B*
⟨*proof*⟩

37.7 Extensional Function Spaces

definition *PiE* :: *'a set* ⇒ (*'a* ⇒ *'b set*) ⇒ (*'a* ⇒ *'b*) *set*
where *PiE S T* = *Pi S T* ∩ *extensional S*

abbreviation Pi_E *A B* ≡ *PiE A B*

syntax
-PiE :: *pttrn* ⇒ *'a set* ⇒ *'b set* ⇒ (*'a* ⇒ *'b*) *set* ((*3*Π_E *-*∈*-./ -*) *10*)
translations
Π_E *x*∈*A*. *B* ⇌ *CONST* Pi_E *A* (λ*x*. *B*)

abbreviation *extensional-funcset* :: *'a set* ⇒ *'b set* ⇒ (*'a* ⇒ *'b*) *set* (**infixr** $\rightarrow_E$ *60*)
where *A* $\rightarrow_E$ *B* ≡ (Π_E *i*∈*A*. *B*)

lemma *extensional-funcset-def*: *extensional-funcset S T* = (*S* → *T*) ∩ *extensional S*

⟨*proof*⟩

lemma *PiE-empty-domain*[*simp*]: $Pi_E\ \{\}\ T = \{\lambda x.\ undefined\}$
⟨*proof*⟩

lemma *PiE-UNIV-domain*: $Pi_E\ UNIV\ T = Pi\ UNIV\ T$
⟨*proof*⟩

lemma *PiE-empty-range*[*simp*]: $i \in I \Longrightarrow F\ i = \{\} \Longrightarrow (\Pi_E\ i{\in}I.\ F\ i) = \{\}$
⟨*proof*⟩

lemma *PiE-eq-empty-iff*: $Pi_E\ I\ F = \{\} \longleftrightarrow (\exists\, i{\in}I.\ F\ i = \{\})$
⟨*proof*⟩

lemma *PiE-arb*: $f \in Pi_E\ S\ T \Longrightarrow x \notin S \Longrightarrow f\ x = undefined$
⟨*proof*⟩

lemma *PiE-mem*: $f \in Pi_E\ S\ T \Longrightarrow x \in S \Longrightarrow f\ x \in T\ x$
⟨*proof*⟩

lemma *PiE-fun-upd*: $y \in T\ x \Longrightarrow f \in Pi_E\ S\ T \Longrightarrow f(x := y) \in Pi_E\ (insert\ x\ S)\ T$
⟨*proof*⟩

lemma *fun-upd-in-PiE*: $x \notin S \Longrightarrow f \in Pi_E\ (insert\ x\ S)\ T \Longrightarrow f(x := undefined) \in Pi_E\ S\ T$
⟨*proof*⟩

lemma *PiE-insert-eq*: $Pi_E\ (insert\ x\ S)\ T = (\lambda(y,\ g).\ g(x := y))\ `\ (T\ x \times Pi_E\ S\ T)$
⟨*proof*⟩

lemma *PiE-Int*: $Pi_E\ I\ A \cap Pi_E\ I\ B = Pi_E\ I\ (\lambda x.\ A\ x \cap B\ x)$
⟨*proof*⟩

lemma *PiE-cong*: $(\bigwedge i.\ i{\in}I \Longrightarrow A\ i = B\ i) \Longrightarrow Pi_E\ I\ A = Pi_E\ I\ B$
⟨*proof*⟩

lemma *PiE-E* [*elim*]:
assumes $f \in Pi_E\ A\ B$
obtains $x \in A$ **and** $f\ x \in B\ x$
| $x \notin A$ **and** $f\ x = undefined$
⟨*proof*⟩

lemma *PiE-I*[*intro!*]:
$(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow (\bigwedge x.\ x \notin A \Longrightarrow f\ x = undefined) \Longrightarrow f \in Pi_E\ A\ B$
⟨*proof*⟩

lemma *PiE-mono*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x \subseteq C\ x) \Longrightarrow Pi_E\ A\ B \subseteq Pi_E\ A\ C$
⟨*proof*⟩

lemma *PiE-iff*: $f \in Pi_E\ I\ X \longleftrightarrow (\forall i{\in}I.\ f\ i \in X\ i) \wedge f \in extensional\ I$
⟨*proof*⟩

lemma *PiE-restrict*[*simp*]: $f \in Pi_E\ A\ B \Longrightarrow restrict\ f\ A = f$
⟨*proof*⟩

lemma *restrict-PiE*[*simp*]: $restrict\ f\ I \in Pi_E\ I\ S \longleftrightarrow f \in Pi\ I\ S$
⟨*proof*⟩

lemma *PiE-eq-subset*:
assumes *ne*: $\bigwedge i.\ i \in I \Longrightarrow F\ i \neq \{\}$ $\bigwedge i.\ i \in I \Longrightarrow F'\ i \neq \{\}$
and *eq*: $Pi_E\ I\ F = Pi_E\ I\ F'$
and $i \in I$
shows $F\ i \subseteq F'\ i$
⟨*proof*⟩

lemma *PiE-eq-iff-not-empty*:
assumes *ne*: $\bigwedge i.\ i \in I \Longrightarrow F\ i \neq \{\}$ $\bigwedge i.\ i \in I \Longrightarrow F'\ i \neq \{\}$
shows $Pi_E\ I\ F = Pi_E\ I\ F' \longleftrightarrow (\forall i{\in}I.\ F\ i = F'\ i)$
⟨*proof*⟩

lemma *PiE-eq-iff*:
$Pi_E\ I\ F = Pi_E\ I\ F' \longleftrightarrow (\forall i{\in}I.\ F\ i = F'\ i) \vee ((\exists i{\in}I.\ F\ i = \{\}) \wedge (\exists i{\in}I.\ F'\ i = \{\}))$
⟨*proof*⟩

lemma *extensional-funcset-fun-upd-restricts-rangeI*:
$\forall y \in S.\ f\ x \neq f\ y \Longrightarrow f \in (insert\ x\ S) \rightarrow_E T \Longrightarrow f(x := undefined) \in S \rightarrow_E (T - \{f\ x\})$
⟨*proof*⟩

lemma *extensional-funcset-fun-upd-extends-rangeI*:
assumes $a \in T$ $f \in S \rightarrow_E (T - \{a\})$
shows $f(x := a) \in insert\ x\ S \rightarrow_E T$
⟨*proof*⟩

37.7.1 Injective Extensional Function Spaces

lemma *extensional-funcset-fun-upd-inj-onI*:
assumes $f \in S \rightarrow_E (T - \{a\})$
and *inj-on f S*
shows *inj-on* $(f(x := a))\ S$
⟨*proof*⟩

lemma *extensional-funcset-extend-domain-inj-on-eq*:
assumes $x \notin S$

shows $\{f.\ f \in (insert\ x\ S) \to_E T \wedge inj\text{-}on\ f\ (insert\ x\ S)\} =$
$(\lambda(y,\ g).\ g(x{:=}y))\ `\ \{(y,\ g).\ y \in T \wedge g \in S \to_E (T - \{y\}) \wedge inj\text{-}on\ g\ S\}$
⟨*proof*⟩

lemma *extensional-funcset-extend-domain-inj-onI*:
assumes $x \notin S$
shows *inj-on* $(\lambda(y,\ g).\ g(x := y))\ \{(y,\ g).\ y \in T \wedge g \in S \to_E (T - \{y\}) \wedge inj\text{-}on\ g\ S\}$
⟨*proof*⟩

37.7.2 Cardinality

lemma *finite-PiE*: *finite* $S \Longrightarrow (\bigwedge i.\ i \in S \Longrightarrow finite\ (T\ i)) \Longrightarrow finite\ (\Pi_E\ i \in S.\ T\ i)$
⟨*proof*⟩

lemma *inj-combinator*: $x \notin S \Longrightarrow inj\text{-}on\ (\lambda(y,\ g).\ g(x := y))\ (T\ x \times Pi_E\ S\ T)$
⟨*proof*⟩

lemma *card-PiE*: *finite* $S \Longrightarrow card\ (\Pi_E\ i \in S.\ T\ i) = (\prod\ i{\in}S.\ card\ (T\ i))$
⟨*proof*⟩

end

38 Pointwise instantiation of functions to division

theory *Function-Division*
imports *Function-Algebras*
begin

38.1 Syntactic with division

instantiation *fun* :: (*type*, *inverse*) *inverse*
begin

definition $inverse\ f = inverse \circ f$

definition $f\ div\ g = (\lambda x.\ f\ x\ /\ g\ x)$

instance ⟨*proof*⟩

end

lemma *inverse-fun-apply* [*simp*]:
$inverse\ f\ x = inverse\ (f\ x)$
⟨*proof*⟩

lemma *divide-fun-apply* [*simp*]:
$(f\ /\ g)\ x = f\ x\ /\ g\ x$

⟨*proof*⟩

Unfortunately, we cannot lift this operations to algebraic type classes for division: being different from the constant zero function $f \neq (0::'a)$ is too weak as precondition. So we must introduce our own set of lemmas.

abbreviation *zero-free* :: $('b \Rightarrow 'a::field) \Rightarrow bool$ **where**
zero-free f $\equiv \neg\, (\exists\, x.\ f\ x = 0)$

lemma *fun-left-inverse*:
fixes $f :: 'b \Rightarrow 'a::field$
shows *zero-free f* $\Longrightarrow$ *inverse f* $*$ *f* $= 1$
⟨*proof*⟩

lemma *fun-right-inverse*:
fixes $f :: 'b \Rightarrow 'a::field$
shows *zero-free f* $\Longrightarrow$ *f* $*$ *inverse f* $= 1$
⟨*proof*⟩

lemma *fun-divide-inverse*:
fixes *f g* $:: 'b \Rightarrow 'a::field$
shows *f* / *g* $=$ *f* $*$ *inverse g*
⟨*proof*⟩

Feel free to extend this.

Another possibility would be a reformulation of the division type classes to user a *zero-free* predicate rather than a direct $a \neq (0::'a)$ condition.

end

39 Preorders with explicit equivalence relation

theory *Preorder*
imports *HOL.Orderings*
begin

class *preorder-equiv* = *preorder*
begin

definition *equiv* :: $'a \Rightarrow 'a \Rightarrow bool$
where *equiv x y* $\longleftrightarrow$ $x \leq y \wedge y \leq x$

notation
equiv (*op* $\approx$) **and**
equiv ((-/ $\approx$ -) [*51*, *51*] *50*)

lemma *refl* [*iff*]: $x \approx x$
⟨*proof*⟩

lemma *trans*: $x \approx y \Longrightarrow y \approx z \Longrightarrow x \approx z$

⟨*proof*⟩

lemma *antisym*: $x \leq y \Longrightarrow y \leq x \Longrightarrow x \approx y$
⟨*proof*⟩

lemma *less-le*: $x < y \longleftrightarrow x \leq y \wedge \neg\, x \approx y$
⟨*proof*⟩

lemma *le-less*: $x \leq y \longleftrightarrow x < y \vee x \approx y$
⟨*proof*⟩

lemma *le-imp-less-or-eq*: $x \leq y \Longrightarrow x < y \vee x \approx y$
⟨*proof*⟩

lemma *less-imp-not-eq*: $x < y \Longrightarrow x \approx y \longleftrightarrow \mathit{False}$
⟨*proof*⟩

lemma *less-imp-not-eq2*: $x < y \Longrightarrow y \approx x \longleftrightarrow \mathit{False}$
⟨*proof*⟩

lemma *neq-le-trans*: $\neg\, a \approx b \Longrightarrow a \leq b \Longrightarrow a < b$
⟨*proof*⟩

lemma *le-neq-trans*: $a \leq b \Longrightarrow \neg\, a \approx b \Longrightarrow a < b$
⟨*proof*⟩

lemma *antisym-conv*: $y \leq x \Longrightarrow x \leq y \longleftrightarrow x \approx y$
⟨*proof*⟩

end

end

40 Common discrete functions

theory *Discrete*
imports *Complex-Main*
begin

40.1 Discrete logarithm

context
begin

qualified fun *log* :: *nat* ⇒ *nat*
where [*simp del*]: *log n* = (*if n* < *2 then 0 else Suc* (*log* (*n div 2*)))

lemma *log-induct* [*consumes 1*, *case-names one double*]:
fixes *n* :: *nat*

assumes $n > 0$
assumes *one*: $P\ 1$
assumes *double*: $\bigwedge n.\ n \geq 2 \Longrightarrow P\ (n\ div\ 2) \Longrightarrow P\ n$
shows $P\ n$
⟨*proof*⟩

lemma *log-zero* [*simp*]: $log\ 0 = 0$
⟨*proof*⟩

lemma *log-one* [*simp*]: $log\ 1 = 0$
⟨*proof*⟩

lemma *log-Suc-zero* [*simp*]: $log\ (Suc\ 0) = 0$
⟨*proof*⟩

lemma *log-rec*: $n \geq 2 \Longrightarrow log\ n = Suc\ (log\ (n\ div\ 2))$
⟨*proof*⟩

lemma *log-twice* [*simp*]: $n \neq 0 \Longrightarrow log\ (2 * n) = Suc\ (log\ n)$
⟨*proof*⟩

lemma *log-half* [*simp*]: $log\ (n\ div\ 2) = log\ n - 1$
⟨*proof*⟩

lemma *log-exp* [*simp*]: $log\ (2\ \hat{}\ n) = n$
⟨*proof*⟩

lemma *log-mono*: *mono log*
⟨*proof*⟩

lemma *log-exp2-le*:
assumes $n > 0$
shows $2\ \hat{}\ log\ n \leq n$
⟨*proof*⟩

lemma *log-exp2-gt*: $2 * 2\ \hat{}\ log\ n > n$
⟨*proof*⟩

lemma *log-exp2-ge*: $2 * 2\ \hat{}\ log\ n \geq n$
⟨*proof*⟩

lemma *log-le-iff*: $m \leq n \Longrightarrow log\ m \leq log\ n$
⟨*proof*⟩

lemma *log-eqI*:
assumes $n > 0\ 2\hat{}k \leq n\ n < 2 * 2\hat{}k$
shows $log\ n = k$
⟨*proof*⟩

lemma *log-altdef*: *log n* = (*if n* = *0 then 0 else nat* ⌊*Transcendental.log 2* (*real-of-nat n*)⌋)
⟨*proof*⟩

40.2 Discrete square root

qualified definition *sqrt* :: *nat* ⇒ *nat*
where *sqrt n* = *Max* {*m*. $m^2 \leq n$}

lemma *sqrt-aux*:
fixes *n* :: *nat*
shows *finite* {*m*. $m^2 \leq n$} **and** {*m*. $m^2 \leq n$} ≠ {}
⟨*proof*⟩

lemma *sqrt-unique*:
assumes *m ^2* ≤ *n n* < (*Suc m*) *^2*
shows *Discrete.sqrt n* = *m*
⟨*proof*⟩

lemma *sqrt-code*[*code*]: *sqrt n* = *Max* (*Set.filter* (λ*m*. $m^2 \leq n$) {*0..n*})
⟨*proof*⟩

lemma *sqrt-inverse-power2* [*simp*]: *sqrt* (n^2) = *n*
⟨*proof*⟩

lemma *sqrt-zero* [*simp*]: *sqrt 0* = *0*
⟨*proof*⟩

lemma *sqrt-one* [*simp*]: *sqrt 1* = *1*
⟨*proof*⟩

lemma *mono-sqrt*: *mono sqrt*
⟨*proof*⟩

lemma *mono-sqrt'*: *m* ≤ *n* ⟹ *Discrete.sqrt m* ≤ *Discrete.sqrt n*
⟨*proof*⟩

lemma *sqrt-greater-zero-iff* [*simp*]: *sqrt n* > *0* ⟷ *n* > *0*
⟨*proof*⟩

lemma *sqrt-power2-le* [*simp*]: (*sqrt n*)^2 ≤ *n*
⟨*proof*⟩

lemma *sqrt-le*: *sqrt n* ≤ *n*
⟨*proof*⟩

Additional facts about the discrete square root, thanks to Julian Biendarra, Manuel Eberl

lemma *Suc-sqrt-power2-gt*: *n* < (*Suc* (*Discrete.sqrt n*)) *^2*

⟨*proof*⟩

lemma *le-sqrt-iff*: $x \leq$ *Discrete.sqrt* $y \longleftrightarrow x\hat{}2 \leq y$
⟨*proof*⟩

lemma *le-sqrtI*: $x\hat{}2 \leq y \Longrightarrow x \leq$ *Discrete.sqrt* y
⟨*proof*⟩

lemma *sqrt-le-iff*: *Discrete.sqrt* $y \leq x \longleftrightarrow (\forall z.\ z\hat{}2 \leq y \longrightarrow z \leq x)$
⟨*proof*⟩

lemma *sqrt-leI*:
$(\bigwedge z.\ z\hat{}2 \leq y \Longrightarrow z \leq x) \Longrightarrow$ *Discrete.sqrt* $y \leq x$
⟨*proof*⟩

lemma *sqrt-Suc*:
Discrete.sqrt (*Suc* n) = (*if* $\exists m.$ *Suc* $n = m\hat{}2$ *then Suc* (*Discrete.sqrt* n) *else Discrete.sqrt* n)
⟨*proof*⟩

end

end

41 Comparing growth of functions on natural numbers by a preorder relation

theory *Function-Growth*
imports *Main Preorder Discrete*
begin

context *linorder*
begin

lemma *mono-invE*:
fixes $f :: 'a \Rightarrow 'b$::*order*
assumes *mono* f
assumes $f\,x < f\,y$
obtains $x < y$
⟨*proof*⟩

end

lemma (**in** *semidom-divide*) *power-diff*:
fixes $a :: 'a$
assumes $a \neq 0$

assumes $m \geq n$
shows $a \mathbin{\hat{}} (m - n) = (a \mathbin{\hat{}} m) \mathbin{div} (a \mathbin{\hat{}} n)$
⟨*proof*⟩

41.1 Motivation

When comparing growth of functions in computer science, it is common to adhere on Landau Symbols ("O-Notation"). However these come at the cost of notational oddities, particularly writing $f = O(g)$ for $f \in O(g)$ etc.

Here we suggest a different way, following Hardy (G. H. Hardy and J. E. Littlewood, Some problems of Diophantine approximation, Acta Mathematica 37 (1914), p. 225). We establish a quasi order relation $\lesssim$ on functions such that $f \lesssim g \longleftrightarrow f \in O(g)$. From a didactic point of view, this does not only avoid the notational oddities mentioned above but also emphasizes the key insight of a growth hierarchy of functions: $(\lambda n.\ 0) \lesssim (\lambda n.\ k) \lesssim \mathit{Discrete.log} \lesssim \mathit{Discrete.sqrt} \lesssim \mathit{id} \lesssim \ldots$.

41.2 Model

Our growth functions are of type $\mathbb{N} \Rightarrow \mathbb{N}$. This is different to the usual conventions for Landau symbols for which $\mathbb{R} \Rightarrow \mathbb{R}$ would be appropriate, but we argue that $\mathbb{R} \Rightarrow \mathbb{R}$ is more appropriate for analysis, whereas our setting is discrete.

Note that we also restrict the additional coefficients to $\mathbb{N}$, something we discuss at the particular definitions.

41.3 The $\lesssim$ relation

definition *less-eq-fun* :: $(\mathit{nat} \Rightarrow \mathit{nat}) \Rightarrow (\mathit{nat} \Rightarrow \mathit{nat}) \Rightarrow \mathit{bool}$ (**infix** $\lesssim$ *50*)
where
$f \lesssim g \longleftrightarrow (\exists\, c{>}0.\ \exists\, n.\ \forall\, m{>}n.\ f\ m \leq c * g\ m)$

This yields $f \lesssim g \longleftrightarrow f \in O(g)$. Note that c is restricted to $\mathbb{N}$. This does not pose any problems since if $f \in O(g)$ holds for a $c \in \mathbb{R}$, it also holds for $\lceil c \rceil \in \mathbb{N}$ by transitivity.

lemma *less-eq-funI* [*intro?*]:
assumes $\exists\, c{>}0.\ \exists\, n.\ \forall\, m{>}n.\ f\ m \leq c * g\ m$
shows $f \lesssim g$
⟨*proof*⟩

lemma *not-less-eq-funI*:
assumes $\bigwedge c\ n.\ c > 0 \Longrightarrow \exists\, m{>}n.\ c * g\ m < f\ m$
shows $\neg\ f \lesssim g$
⟨*proof*⟩

lemma *less-eq-funE* [*elim?*]:

assumes $f \lesssim g$
obtains $n\ c$ **where** $c > 0$ **and** $\bigwedge m.\ m > n \Longrightarrow f\ m \leq c * g\ m$
⟨*proof*⟩

lemma *not-less-eq-funE*:
assumes $\neg\ f \lesssim g$ **and** $c > 0$
obtains m **where** $m > n$ **and** $c * g\ m < f\ m$
⟨*proof*⟩

41.4 The $\approx$ relation, the equivalence relation induced by $\lesssim$

definition *equiv-fun* :: $(nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat) \Rightarrow bool$ (**infix** $\cong$ *50*)
where
$f \cong g \longleftrightarrow$
$(\exists\, c_1{>}0.\ \exists\, c_2{>}0.\ \exists\, n.\ \forall\, m{>}n.\ f\ m \leq c_1 * g\ m \wedge g\ m \leq c_2 * f\ m)$

This yields $f \cong g \longleftrightarrow f \in \Theta(g)$. Concerning c_1 and c_2 restricted to *nat*, see note above on $(\lesssim)$.

lemma *equiv-funI*:
assumes $\exists\, c_1{>}0.\ \exists\, c_2{>}0.\ \exists\, n.\ \forall\, m{>}n.\ f\ m \leq c_1 * g\ m \wedge g\ m \leq c_2 * f\ m$
shows $f \cong g$
⟨*proof*⟩

lemma *not-equiv-funI*:
assumes $\bigwedge c_1\ c_2\ n.\ c_1 > 0 \Longrightarrow c_2 > 0 \Longrightarrow$
$\exists\, m{>}n.\ c_1 * f\ m < g\ m \vee c_2 * g\ m < f\ m$
shows $\neg\ f \cong g$
⟨*proof*⟩

lemma *equiv-funE*:
assumes $f \cong g$
obtains $n\ c_1\ c_2$ **where** $c_1 > 0$ **and** $c_2 > 0$
and $\bigwedge m.\ m > n \Longrightarrow f\ m \leq c_1 * g\ m \wedge g\ m \leq c_2 * f\ m$
⟨*proof*⟩

lemma *not-equiv-funE*:
fixes $n\ c_1\ c_2$
assumes $\neg\ f \cong g$ **and** $c_1 > 0$ **and** $c_2 > 0$
obtains m **where** $m > n$
and $c_1 * f\ m < g\ m \vee c_2 * g\ m < f\ m$
⟨*proof*⟩

41.5 The $\prec$ relation, the strict part of $\lesssim$

definition *less-fun* :: $(nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat) \Rightarrow bool$ (**infix** $\prec$ *50*)
where
$f \prec g \longleftrightarrow f \lesssim g \wedge \neg\ g \lesssim f$

lemma *less-funI*:
assumes $\exists\, c{>}0.\ \exists\, n.\ \forall\, m{>}n.\ f\ m \leq c * g\ m$

and $\bigwedge c\ n.\ c > 0 \Longrightarrow \exists\, m{>}n.\ c * f\ m < g\ m$
shows $f \prec g$
⟨*proof*⟩

lemma *not-less-funI*:
assumes $\bigwedge c\ n.\ c > 0 \Longrightarrow \exists\, m{>}n.\ c * g\ m < f\ m$
and $\exists\, c{>}0.\ \exists\, n.\ \forall\, m{>}n.\ g\ m \leq c * f\ m$
shows $\neg\ f \prec g$
⟨*proof*⟩

lemma *less-funE* [*elim?*]:
assumes $f \prec g$
obtains $n\ c$ **where** $c > 0$ **and** $\bigwedge m.\ m > n \Longrightarrow f\ m \leq c * g\ m$
and $\bigwedge c\ n.\ c > 0 \Longrightarrow \exists\, m{>}n.\ c * f\ m < g\ m$
⟨*proof*⟩

lemma *not-less-funE*:
assumes $\neg\ f \prec g$ **and** $c > 0$
obtains m **where** $m > n$ **and** $c * g\ m < f\ m$
| $d\ q$ **where** $\bigwedge m.\ d > 0 \Longrightarrow m > q \Longrightarrow g\ q \leq d * f\ q$
⟨*proof*⟩

I did not find a proof for $f \prec g \longleftrightarrow f \in o(g)$. Maybe this only holds if f and/or g are of a certain class of functions. However $f \in o(g) \longrightarrow f \prec g$ is provable, and this yields a handy introduction rule.

Note that D. Knuth ignores o altogether. So what ...

Something still has to be said about the coefficient c in the definition of $(\prec)$. In the typical definition of o, it occurs on the *right* hand side of the $(>)$. The reason is that the situation is dual to the definition of O: the definition works since c may become arbitrary small. Since this is not possible within $\mathbb{N}$, we push the coefficient to the left hand side instead such that it may become arbitrary big instead.

lemma *less-fun-strongI*:
assumes $\bigwedge c.\ c > 0 \Longrightarrow \exists\, n.\ \forall\, m{>}n.\ c * f\ m < g\ m$
shows $f \prec g$
⟨*proof*⟩

41.6 $\lesssim$ is a preorder

This yields all lemmas relating $\lesssim$, $\prec$ and $\cong$.

interpretation *fun-order*: *preorder-equiv less-eq-fun less-fun*
rewrites *fun-order.equiv* = *equiv-fun*
⟨*proof*⟩

declare *fun-order.antisym* [*intro?*]

41.7 Simple examples

Most of these are left as constructive exercises for the reader. Note that additional preconditions to the functions may be necessary. The list here is by no means to be intended as complete construction set for typical functions, here surely something has to be added yet.

$(\lambda n.\ f\ n\ +\ k) \cong f$

lemma *equiv-fun-mono-const*:
assumes *mono f* **and** $\exists\, n.\ f\ n > 0$
shows $(\lambda n.\ f\ n\ +\ k) \cong f$
⟨*proof*⟩

lemma
assumes *strict-mono f*
shows $(\lambda n.\ f\ n\ +\ k) \cong f$
⟨*proof*⟩

lemma
$(\lambda n.\ Suc\ k\ *\ f\ n) \cong f$
⟨*proof*⟩

lemma
$f \lesssim (\lambda n.\ f\ n\ +\ g\ n)$
⟨*proof*⟩

lemma
$(\lambda\text{-}.\ 0) \prec (\lambda n.\ Suc\ k)$
⟨*proof*⟩

lemma
$(\lambda\text{-}.\ k) \prec Discrete.log$
⟨*proof*⟩

$Discrete.log \prec Discrete.sqrt$

lemma
$Discrete.sqrt \prec id$
⟨*proof*⟩

lemma
$id \prec (\lambda n.\ n^2)$
⟨*proof*⟩

lemma
$(\lambda n.\ n\ \hat{}\ k) \prec (\lambda n.\ n\ \hat{}\ Suc\ k)$
⟨*proof*⟩

$(\lambda n.\ n^k) \prec op\ \hat{}\ 2$

end

42 Lexical order on functions

theory *Fun-Lexorder*
imports *Main*
begin

definition *less-fun* :: ('*a*::*linorder* ⇒ '*b*::*linorder*) ⇒ ('*a* ⇒ '*b*) ⇒ *bool*
where
  less-fun f g ⟷ (∃ *k*. *f k* < *g k* ∧ (∀ *k'* < *k*. *f k'* = *g k'*))

lemma *less-funI*:
  assumes ∃ *k*. *f k* < *g k* ∧ (∀ *k'* < *k*. *f k'* = *g k'*)
  shows *less-fun f g*
  ⟨*proof*⟩

lemma *less-funE*:
  assumes *less-fun f g*
  obtains *k* **where** *f k* < *g k* **and** ⋀*k'*. *k'* < *k* ⟹ *f k'* = *g k'*
  ⟨*proof*⟩

lemma *less-fun-asym*:
  assumes *less-fun f g*
  shows ¬ *less-fun g f*
⟨*proof*⟩

lemma *less-fun-irrefl*:
  ¬ *less-fun f f*
⟨*proof*⟩

lemma *less-fun-trans*:
  assumes *less-fun f g* **and** *less-fun g h*
  shows *less-fun f h*
⟨*proof*⟩

lemma *order-less-fun*:
  class.order (λ*f g*. *less-fun f g* ∨ *f* = *g*) *less-fun*
  ⟨*proof*⟩

lemma *less-fun-trichotomy*:
  assumes *finite* {*k*. *f k* ≠ *g k*}
  shows *less-fun f g* ∨ *f* = *g* ∨ *less-fun g f*
⟨*proof*⟩

end

43 The 'going_to' filter

theory *Going-To-Filter*
  imports *Complex-Main*

begin

definition *going-to-within* :: (*'a* ⇒ *'b*) ⇒ *'b filter* ⇒ *'a set* ⇒ *'a filter*
((-)/ *going'-to* (-)/ *within* (-) [*1000*,*60*,*60*] *60*) **where**
f going-to F within A = *inf* (*filtercomap f F*) (*principal A*)

abbreviation *going-to* :: (*'a* ⇒ *'b*) ⇒ *'b filter* ⇒ *'a filter*
(**infix** *going'-to 60*)
where *f going-to F* ≡ *f going-to F within UNIV*

The 'going_to' filter is, in a sense, the opposite of *filtermap*. It corresponds to the intuition of, given a function $f : A \to B$ and a filter F on the range of B, looking at such values of x that $f(x)$ approaches F. This can be written as *f going-to F*.

A classic example is the *at-infinity* filter, which describes the neigbourhood of infinity (i. e. all values sufficiently far away from the zero). This can also be written as *norm going-to at-top*.

Additionally, the 'going_to' filter can be restricted with an optional 'within' parameter. For instance, if one would would want to consider the filter of complex numbers near infinity that do not lie on the negative real line, one could write *cmod going-to at-top within − complex-of-real ' {..0}*.

A third, less mathematical example lies in the complexity analysis of algorithms. Suppose we wanted to say that an algorithm on lists takes $O(n^2)$ time where n is the length of the input list. We can write this using the Landau symbols from the AFP, where the underlying filter is *length going-to sequentially*. If, on the other hand, we want to look the complexity of the algorithm on sorted lists, we could use the filter *length going-to sequentially within {xs. sorted xs}*.

lemma *going-to-def*: *f going-to F* = *filtercomap f F*
⟨*proof*⟩

lemma *eventually-going-toI* [*intro*]:
assumes *eventually P F*
shows *eventually* (λ*x*. *P* (*f x*)) (*f going-to F*)
⟨*proof*⟩

lemma *filterlim-going-toI-weak* [*intro*]: *filterlim f F* (*f going-to F within A*)
⟨*proof*⟩

lemma *going-to-mono*: *F* ≤ *G* ⟹ *A* ⊆ *B* ⟹ *f going-to F within A* ≤ *f going-to G within B*
⟨*proof*⟩

lemma *going-to-inf*:
f going-to (*inf F G*) *within A* = *inf* (*f going-to F within A*) (*f going-to G within A*)
⟨*proof*⟩

lemma *going-to-sup*:
f going-to (*sup F G*) *within A* ≥ *sup* (*f going-to F within A*) (*f going-to G within A*)
⟨*proof*⟩

lemma *going-to-top* [*simp*]: *f going-to top within A* = *principal A*
⟨*proof*⟩

lemma *going-to-bot* [*simp*]: *f going-to bot within A* = *bot*
⟨*proof*⟩

lemma *going-to-principal*:
f going-to principal A within B = *principal* (*f* −‘ *A* ∩ *B*)
⟨*proof*⟩

lemma *going-to-within-empty* [*simp*]: *f going-to F within* {} = *bot*
⟨*proof*⟩

lemma *going-to-within-union* [*simp*]:
f going-to F within (*A* ∪ *B*) = *sup* (*f going-to F within A*) (*f going-to F within B*)
⟨*proof*⟩

lemma *eventually-going-to-at-top-linorder*:
fixes *f* :: *'a* ⇒ *'b* :: *linorder*
shows *eventually P* (*f going-to at-top within A*) ⟷ (∃ *C*. ∀ *x*∈*A*. *f x* ≥ *C* ⟶ *P x*)
⟨*proof*⟩

lemma *eventually-going-to-at-bot-linorder*:
fixes *f* :: *'a* ⇒ *'b* :: *linorder*
shows *eventually P* (*f going-to at-bot within A*) ⟷ (∃ *C*. ∀ *x*∈*A*. *f x* ≤ *C* ⟶ *P x*)
⟨*proof*⟩

lemma *eventually-going-to-at-top-dense*:
fixes *f* :: *'a* ⇒ *'b* :: {*linorder*,*no-top*}
shows *eventually P* (*f going-to at-top within A*) ⟷ (∃ *C*. ∀ *x*∈*A*. *f x* > *C* ⟶ *P x*)
⟨*proof*⟩

lemma *eventually-going-to-at-bot-dense*:
fixes *f* :: *'a* ⇒ *'b* :: {*linorder*,*no-bot*}
shows *eventually P* (*f going-to at-bot within A*) ⟷ (∃ *C*. ∀ *x*∈*A*. *f x* < *C* ⟶ *P x*)
⟨*proof*⟩

lemma *eventually-going-to-nhds*:

eventually P (f going-to nhds a within A) $\longleftrightarrow$
$(\exists S.\ open\ S \wedge a \in S \wedge (\forall x{\in}A.\ f\ x \in S \longrightarrow P\ x))$
⟨*proof*⟩

lemma *eventually-going-to-at*:
eventually P (f going-to (at a within B) within A) $\longleftrightarrow$
$(\exists S.\ open\ S \wedge a \in S \wedge (\forall x{\in}A.\ f\ x \in B \cap S - \{a\} \longrightarrow P\ x))$
⟨*proof*⟩

lemma *norm-going-to-at-top-eq*: *norm going-to at-top* = *at-infinity*
⟨*proof*⟩

lemmas *at-infinity-altdef* = *norm-going-to-at-top-eq* [*symmetric*]

end

44 Big sum and product over function bodies

theory *Groups-Big-Fun*
imports
Main
begin

44.1 Abstract product

locale *comm-monoid-fun* = *comm-monoid*
begin

definition $G :: ('b \Rightarrow 'a) \Rightarrow 'a$
where
expand-set: $G\ g$ = *comm-monoid-set.F f* **1** $g\ \{a.\ g\ a \neq \mathbf{1}\}$

interpretation *F*: *comm-monoid-set f* **1**
⟨*proof*⟩

lemma *expand-superset*:
assumes *finite A* **and** $\{a.\ g\ a \neq \mathbf{1}\} \subseteq A$
shows $G\ g = F.F\ g\ A$
⟨*proof*⟩

lemma *conditionalize*:
assumes *finite A*
shows $F.F\ g\ A = G\ (\lambda a.$ *if* $a \in A$ *then* $g\ a$ *else* $\mathbf{1})$
⟨*proof*⟩

lemma *neutral* [*simp*]:
$G\ (\lambda a.\ \mathbf{1}) = \mathbf{1}$
⟨*proof*⟩

lemma *update* [*simp*]:
assumes *finite* {*a*. *g a* ≠ **1**}
assumes *g a* = **1**
shows *G* (*g*(*a* := *b*)) = *b* ∗ *G g*
⟨*proof*⟩

lemma *infinite* [*simp*]:
¬ *finite* {*a*. *g a* ≠ **1**} ⟹ *G g* = **1**
⟨*proof*⟩

lemma *cong*:
assumes ⋀*a*. *g a* = *h a*
shows *G g* = *G h*
⟨*proof*⟩

lemma *strong-cong* [*cong*]:
assumes ⋀*a*. *g a* = *h a*
shows *G* (λ*a*. *g a*) = *G* (λ*a*. *h a*)
⟨*proof*⟩

lemma *not-neutral-obtains-not-neutral*:
assumes *G g* ≠ **1**
obtains *a* **where** *g a* ≠ **1**
⟨*proof*⟩

lemma *reindex-cong*:
assumes *bij l*
assumes *g* ∘ *l* = *h*
shows *G g* = *G h*
⟨*proof*⟩

lemma *distrib*:
assumes *finite* {*a*. *g a* ≠ **1**} **and** *finite* {*a*. *h a* ≠ **1**}
shows *G* (λ*a*. *g a* ∗ *h a*) = *G g* ∗ *G h*
⟨*proof*⟩

lemma *commute*:
assumes *finite C*
assumes *subset*: {*a*. ∃ *b*. *g a b* ≠ **1**} × {*b*. ∃ *a*. *g a b* ≠ **1**} ⊆ *C* (**is** *?A* × *?B* ⊆ *C*)
shows *G* (λ*a*. *G* (*g a*)) = *G* (λ*b*. *G* (λ*a*. *g a b*))
⟨*proof*⟩

lemma *cartesian-product*:
assumes *finite C*
assumes *subset*: {*a*. ∃ *b*. *g a b* ≠ **1**} × {*b*. ∃ *a*. *g a b* ≠ **1**} ⊆ *C* (**is** *?A* × *?B* ⊆ *C*)
shows *G* (λ*a*. *G* (*g a*)) = *G* (λ(*a*, *b*). *g a b*)
⟨*proof*⟩

lemma *cartesian-product2*:
assumes *fin*: *finite D*
assumes *subset*: $\{(a,\ b).\ \exists\, c.\ g\ a\ b\ c \neq \mathbf{1}\} \times \{c.\ \exists\, a\ b.\ g\ a\ b\ c \neq \mathbf{1}\} \subseteq D$ (**is** *?AB* $\times$ *?C* $\subseteq$ *D*)
shows $G\ (\lambda(a,\ b).\ G\ (g\ a\ b)) = G\ (\lambda(a,\ b,\ c).\ g\ a\ b\ c)$
⟨*proof*⟩

lemma *delta* [*simp*]:
$G\ (\lambda b.\ \textit{if}\ b = a\ \textit{then}\ g\ b\ \textit{else}\ \mathbf{1}) = g\ a$
⟨*proof*⟩

lemma *delta'* [*simp*]:
$G\ (\lambda b.\ \textit{if}\ a = b\ \textit{then}\ g\ b\ \textit{else}\ \mathbf{1}) = g\ a$
⟨*proof*⟩

end

44.2 Concrete sum

context *comm-monoid-add*
begin

sublocale *Sum-any*: *comm-monoid-fun plus 0*
defines *Sum-any* = *Sum-any.G*
rewrites *comm-monoid-set.F plus 0* = *sum*
⟨*proof*⟩

end

syntax (*ASCII*)
-Sum-any :: *pttrn* ⇒ *'a* ⇒ *'a*::*comm-monoid-add* (*(3SUM -. -) [0, 10] 10*)
syntax
-Sum-any :: *pttrn* ⇒ *'a* ⇒ *'a*::*comm-monoid-add* (*(3*$\sum$ *-. -) [0, 10] 10*)
translations
$\sum a.\ b \rightleftharpoons$ *CONST Sum-any* $(\lambda a.\ b)$

lemma *Sum-any-left-distrib*:
fixes *r* :: *'a* :: *semiring-0*
assumes *finite* $\{a.\ g\ a \neq 0\}$
shows *Sum-any g* $*$ *r* $= (\sum n.\ g\ n * r)$
⟨*proof*⟩

lemma *Sum-any-right-distrib*:
fixes *r* :: *'a* :: *semiring-0*
assumes *finite* $\{a.\ g\ a \neq 0\}$
shows *r* $*$ *Sum-any g* $= (\sum n.\ r * g\ n)$
⟨*proof*⟩

lemma *Sum-any-product*:
fixes $f\ g :: 'b \Rightarrow 'a$::*semiring-0*
assumes *finite* $\{a.\ f\ a \neq 0\}$ **and** *finite* $\{b.\ g\ b \neq 0\}$
shows *Sum-any* $f * $ *Sum-any* $g = (\sum a.\ \sum b.\ f\ a * g\ b)$
⟨*proof*⟩

lemma *Sum-any-eq-zero-iff* [*simp*]:
fixes $f :: 'a \Rightarrow nat$
assumes *finite* $\{a.\ f\ a \neq 0\}$
shows *Sum-any* $f = 0 \longleftrightarrow f = (\lambda\text{-}.\ 0)$
⟨*proof*⟩

44.3 Concrete product

context *comm-monoid-mult*
begin

sublocale *Prod-any*: *comm-monoid-fun times 1*
defines *Prod-any* = *Prod-any.G*
rewrites *comm-monoid-set.F times 1* = *prod*
⟨*proof*⟩

end

syntax (*ASCII*)
-Prod-any :: *pttrn* ⇒ $'a$ ⇒ $'a$::*comm-monoid-mult* ((*3PROD* -. -) [*0*, *10*] *10*)
syntax
-Prod-any :: *pttrn* ⇒ $'a$ ⇒ $'a$::*comm-monoid-mult* ((*3*$\prod$ -. -) [*0*, *10*] *10*)
translations
$\prod a.\ b$ == *CONST Prod-any* $(\lambda a.\ b)$

lemma *Prod-any-zero*:
fixes $f :: 'b \Rightarrow 'a$ :: *comm-semiring-1*
assumes *finite* $\{a.\ f\ a \neq 1\}$
assumes $f\ a = 0$
shows $(\prod a.\ f\ a) = 0$
⟨*proof*⟩

lemma *Prod-any-not-zero*:
fixes $f :: 'b \Rightarrow 'a$ :: *comm-semiring-1*
assumes *finite* $\{a.\ f\ a \neq 1\}$
assumes $(\prod a.\ f\ a) \neq 0$
shows $f\ a \neq 0$
⟨*proof*⟩

lemma *power-Sum-any*:
assumes *finite* $\{a.\ f\ a \neq 0\}$
shows $c\ \hat{}\ (\sum a.\ f\ a) = (\prod a.\ c\ \hat{}\ f\ a)$
⟨*proof*⟩

end

45 Immutable Arrays with Code Generation

theory *IArray*
imports *Main*
begin

Immutable arrays are lists wrapped up in an additional constructor. There are no update operations. Hence code generation can safely implement this type by efficient target language arrays. Currently only SML is provided. Should be extended to other target languages and more operations.

Note that arrays cannot be printed directly but only by turning them into lists first. Arrays could be converted back into lists for printing if they were wrapped up in an additional constructor.

context
begin

datatype *'a iarray* = *IArray 'a list*

qualified primrec *list-of* :: *'a iarray* ⇒ *'a list* **where**
list-of (*IArray xs*) = *xs*

qualified definition *of-fun* :: (*nat* ⇒ *'a*) ⇒ *nat* ⇒ *'a iarray* **where**
[*simp*]: *of-fun f n* = *IArray* (*map f* [*0*..<*n*])

qualified definition *sub* :: *'a iarray* ⇒ *nat* ⇒ *'a* (**infixl** !! *100*) **where**
[*simp*]: *as* !! *n* = *IArray.list-of as* ! *n*

qualified definition *length* :: *'a iarray* ⇒ *nat* **where**
[*simp*]: *length as* = *List.length* (*IArray.list-of as*)

qualified fun *all* :: (*'a* ⇒ *bool*) ⇒ *'a iarray* ⇒ *bool* **where**
all p (*IArray as*) = (*ALL a* : *set as*. *p a*)

qualified fun *exists* :: (*'a* ⇒ *bool*) ⇒ *'a iarray* ⇒ *bool* **where**
exists p (*IArray as*) = (*EX a* : *set as*. *p a*)

lemma *list-of-code* [*code*]:
IArray.list-of as = *map* (λ*n*. *as* !! *n*) [*0* ..< *IArray.length as*]
⟨*proof*⟩

end

45.1 Code Generation

code-reserved *SML Vector*

code-printing
  type-constructor *iarray* ⇀ (*SML*) - *Vector.vector*
| **constant** *IArray* ⇀ (*SML*) *Vector.fromList*
| **constant** *IArray.all* ⇀ (*SML*) *Vector.all*
| **constant** *IArray.exists* ⇀ (*SML*) *Vector.exists*

lemma [*code*]:
  size (*as* :: *'a iarray*) = *Suc* (*length* (*IArray.list-of as*))
  ⟨*proof*⟩

lemma [*code*]:
  size-iarray f as = *Suc* (*size-list f* (*IArray.list-of as*))
  ⟨*proof*⟩

lemma [*code*]:
  rec-iarray f as = *f* (*IArray.list-of as*)
  ⟨*proof*⟩

lemma [*code*]:
  case-iarray f as = *f* (*IArray.list-of as*)
  ⟨*proof*⟩

lemma [*code*]:
  set-iarray as = *set* (*IArray.list-of as*)
  ⟨*proof*⟩

lemma [*code*]:
  map-iarray f as = *IArray* (*map f* (*IArray.list-of as*))
  ⟨*proof*⟩

lemma [*code*]:
  rel-iarray r as bs = *list-all2 r* (*IArray.list-of as*) (*IArray.list-of bs*)
  ⟨*proof*⟩

lemma [*code*]:
  HOL.equal as bs ⟷ *HOL.equal* (*IArray.list-of as*) (*IArray.list-of bs*)
  ⟨*proof*⟩

context
begin

qualified primrec *tabulate* :: *integer* × (*integer* ⇒ *'a*) ⇒ *'a iarray* **where**
  tabulate (*n*, *f*) = *IArray* (*map* (*f* ∘ *integer-of-nat*) [*0*..<*nat-of-integer n*])

end

lemma [*code*]:
IArray.of-fun f n = *IArray.tabulate* (*integer-of-nat n*, *f* ∘ *nat-of-integer*)
⟨*proof*⟩

code-printing
constant *IArray.tabulate* ⇀ (*SML*) *Vector.tabulate*

context
begin

qualified primrec *sub'* :: *'a iarray* × *integer* ⇒ *'a* **where**
[*code del*]: *sub'* (*as*, *n*) = *IArray.list-of as* ! *nat-of-integer n*

end

lemma [*code*]:
IArray.sub' (*IArray as*, *n*) = *as* ! *nat-of-integer n*
⟨*proof*⟩

lemma [*code*]:
as !! *n* = *IArray.sub'* (*as*, *integer-of-nat n*)
⟨*proof*⟩

code-printing
constant *IArray.sub'* ⇀ (*SML*) *Vector.sub*

context
begin

qualified definition *length'* :: *'a iarray* ⇒ *integer* **where**
[*code del*, *simp*]: *length' as* = *integer-of-nat* (*List.length* (*IArray.list-of as*))

end

lemma [*code*]:
IArray.length' (*IArray as*) = *integer-of-nat* (*List.length as*)
⟨*proof*⟩

lemma [*code*]:
IArray.length as = *nat-of-integer* (*IArray.length' as*)
⟨*proof*⟩

context *term-syntax*
begin

lemma [*code*]:
Code-Evaluation.term-of (*as* :: *'a*::*typerep iarray*) =
Code-Evaluation.Const (*STR ''IArray.iarray.IArray''*) (*TYPEREP*(*'a list* ⇒ *'a iarray*)) <·> (*Code-Evaluation.term-of* (*IArray.list-of as*))

⟨*proof*⟩

end

code-printing
constant *IArray.length'* ⇀ (*SML*) *Vector.length*

end

theory *Lattice-Constructions*
imports *Main*
begin

45.2 Values extended by a bottom element

datatype *'a bot* = *Value 'a* | *Bot*

instantiation *bot* :: (*preorder*) *preorder*
begin

definition *less-eq-bot* **where**
x ≤ *y* ⟷ (*case x of Bot* ⇒ *True* | *Value x* ⇒ (*case y of Bot* ⇒ *False* | *Value y* ⇒ *x* ≤ *y*))

definition *less-bot* **where**
x < *y* ⟷ (*case y of Bot* ⇒ *False* | *Value y* ⇒ (*case x of Bot* ⇒ *True* | *Value x* ⇒ *x* < *y*))

lemma *less-eq-bot-Bot* [*simp*]: *Bot* ≤ *x*
⟨*proof*⟩

lemma *less-eq-bot-Bot-code* [*code*]: *Bot* ≤ *x* ⟷ *True*
⟨*proof*⟩

lemma *less-eq-bot-Bot-is-Bot*: *x* ≤ *Bot* ⟹ *x* = *Bot*
⟨*proof*⟩

lemma *less-eq-bot-Value-Bot* [*simp*, *code*]: *Value x* ≤ *Bot* ⟷ *False*
⟨*proof*⟩

lemma *less-eq-bot-Value* [*simp*, *code*]: *Value x* ≤ *Value y* ⟷ *x* ≤ *y*
⟨*proof*⟩

lemma *less-bot-Bot* [*simp*, *code*]: *x* < *Bot* ⟷ *False*
⟨*proof*⟩

lemma *less-bot-Bot-is-Value*: *Bot* < *x* ⟹ ∃ *z*. *x* = *Value z*
⟨*proof*⟩

lemma *less-bot-Bot-Value* [*simp*]: *Bot* < *Value x*
⟨*proof*⟩

lemma *less-bot-Bot-Value-code* [*code*]: *Bot* < *Value x* ⟷ *True*
⟨*proof*⟩

lemma *less-bot-Value* [*simp*, *code*]: *Value x* < *Value y* ⟷ *x* < *y*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *bot* :: (*order*) *order*
⟨*proof*⟩

instance *bot* :: (*linorder*) *linorder*
⟨*proof*⟩

instantiation *bot* :: (*order*) *bot*
begin
definition *bot* = *Bot*
instance ⟨*proof*⟩
end

instantiation *bot* :: (*top*) *top*
begin
definition *top* = *Value top*
instance ⟨*proof*⟩
end

instantiation *bot* :: (*semilattice-inf*) *semilattice-inf*
begin

definition *inf-bot*
where
inf x y =
(*case x of*
Bot ⇒ *Bot*
| *Value v* ⇒
(*case y of*
Bot ⇒ *Bot*
| *Value v'* ⇒ *Value* (*inf v v'*)))

instance
⟨*proof*⟩

end

instantiation *bot* :: (*semilattice-sup*) *semilattice-sup*
begin

definition *sup-bot*
where
sup x y =
(*case x of*
Bot ⇒ *y*
| *Value v* ⇒
(*case y of*
Bot ⇒ *x*
| *Value v'* ⇒ *Value* (*sup v v'*)))

instance
⟨*proof*⟩

end

instance *bot* :: (*lattice*) *bounded-lattice-bot*
⟨*proof*⟩

45.3 Values extended by a top element

datatype *'a top* = *Value 'a* | *Top*

instantiation *top* :: (*preorder*) *preorder*
begin

definition *less-eq-top* **where**
$x \leq y \longleftrightarrow$ (*case y of Top* ⇒ *True* | *Value y* ⇒ (*case x of Top* ⇒ *False* | *Value x* ⇒ $x \leq y$))

definition *less-top* **where**
$x < y \longleftrightarrow$ (*case x of Top* ⇒ *False* | *Value x* ⇒ (*case y of Top* ⇒ *True* | *Value y* ⇒ $x < y$))

lemma *less-eq-top-Top* [*simp*]: $x \leq$ *Top*
⟨*proof*⟩

lemma *less-eq-top-Top-code* [*code*]: $x \leq$ *Top* $\longleftrightarrow$ *True*
⟨*proof*⟩

lemma *less-eq-top-is-Top*: *Top* $\leq x \Longrightarrow x =$ *Top*
⟨*proof*⟩

lemma *less-eq-top-Top-Value* [*simp*, *code*]: *Top* $\leq$ *Value x* $\longleftrightarrow$ *False*
⟨*proof*⟩

lemma *less-eq-top-Value-Value* [*simp*, *code*]: *Value x* ≤ *Value y* ⟷ *x* ≤ *y*
⟨*proof*⟩

lemma *less-top-Top* [*simp*, *code*]: *Top* < *x* ⟷ *False*
⟨*proof*⟩

lemma *less-top-Top-is-Value*: *x* < *Top* ⟹ ∃ *z*. *x* = *Value z*
⟨*proof*⟩

lemma *less-top-Value-Top* [*simp*]: *Value x* < *Top*
⟨*proof*⟩

lemma *less-top-Value-Top-code* [*code*]: *Value x* < *Top* ⟷ *True*
⟨*proof*⟩

lemma *less-top-Value* [*simp*, *code*]: *Value x* < *Value y* ⟷ *x* < *y*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *top* :: (*order*) *order*
⟨*proof*⟩

instance *top* :: (*linorder*) *linorder*
⟨*proof*⟩

instantiation *top* :: (*order*) *top*
begin
definition *top* = *Top*
instance ⟨*proof*⟩
end

instantiation *top* :: (*bot*) *bot*
begin
definition *bot* = *Value bot*
instance ⟨*proof*⟩
end

instantiation *top* :: (*semilattice-inf*) *semilattice-inf*
begin

definition *inf-top*
where
inf x y =
(*case x of*

```
      Top ⇒ y
    | Value v ⇒
        (case y of
          Top ⇒ x
        | Value v' ⇒ Value (inf v v')))

instance
  ⟨proof⟩

end

instantiation top :: (semilattice-sup) semilattice-sup
begin

definition sup-top
where
  sup x y =
    (case x of
      Top ⇒ Top
    | Value v ⇒
        (case y of
          Top ⇒ Top
        | Value v' ⇒ Value (sup v v')))

instance
  ⟨proof⟩

end

instance top :: (lattice) bounded-lattice-top
  ⟨proof⟩
```

45.4 Values extended by a top and a bottom element

```
datatype 'a flat-complete-lattice = Value 'a | Bot | Top

instantiation flat-complete-lattice :: (type) order
begin

definition less-eq-flat-complete-lattice
where
  x ≤ y ≡
    (case x of
      Bot ⇒ True
    | Value v1 ⇒
        (case y of
          Bot ⇒ False
        | Value v2 ⇒ v1 = v2
        | Top ⇒ True)
```

```
    | Top ⇒ y = Top)

definition less-flat-complete-lattice
where
  x < y =
    (case x of
      Bot ⇒ y ≠ Bot
    | Value v1 ⇒ y = Top
    | Top ⇒ False)

lemma [simp]: Bot ≤ y
  ⟨proof⟩

lemma [simp]: y ≤ Top
  ⟨proof⟩

lemma greater-than-two-values:
  assumes a ≠ b Value a ≤ z Value b ≤ z
  shows z = Top
  ⟨proof⟩

lemma lesser-than-two-values:
  assumes a ≠ b z ≤ Value a z ≤ Value b
  shows z = Bot
  ⟨proof⟩

instance
  ⟨proof⟩

end

instantiation flat-complete-lattice :: (type) bot
begin
  definition bot = Bot
  instance ⟨proof⟩
end

instantiation flat-complete-lattice :: (type) top
begin
  definition top = Top
  instance ⟨proof⟩
end

instantiation flat-complete-lattice :: (type) lattice
begin

definition inf-flat-complete-lattice
where
  inf x y =
```

```
    (case x of
      Bot ⇒ Bot
    | Value v1 ⇒
        (case y of
          Bot ⇒ Bot
        | Value v2 ⇒ if v1 = v2 then x else Bot
        | Top ⇒ x)
    | Top ⇒ y)

definition sup-flat-complete-lattice
where
  sup x y =
    (case x of
      Bot ⇒ y
    | Value v1 ⇒
        (case y of
          Bot ⇒ x
        | Value v2 ⇒ if v1 = v2 then x else Top
        | Top ⇒ Top)
    | Top ⇒ Top)

instance
  ⟨proof⟩

end

instantiation flat-complete-lattice :: (type) complete-lattice
begin

definition Sup-flat-complete-lattice
where
  Sup A =
    (if A = {} ∨ A = {Bot} then Bot
     else if ∃ v. A − {Bot} = {Value v} then Value (THE v. A − {Bot} = {Value
v})
     else Top)

definition Inf-flat-complete-lattice
where
  Inf A =
    (if A = {} ∨ A = {Top} then Top
     else if ∃ v. A − {Top} = {Value v} then Value (THE v. A − {Top} = {Value
v})
     else Bot)

instance
⟨proof⟩

end
```

end

46 Infinite Streams

theory *Stream*
imports *Nat-Bijection*
begin

codatatype (*sset*: $'a$) *stream* =
SCons (*shd*: $'a$) (*stl*: $'a$ *stream*) (**infixr** ## 65)
for
map: *smap*
rel: *stream-all2*

context
begin

qualified definition *smember* :: $'a \Rightarrow 'a$ *stream* $\Rightarrow$ *bool* **where**
[*code-abbrev*]: *smember x s* $\longleftrightarrow$ *x* $\in$ *sset s*

lemma *smember-code*[*code*, *simp*]: *smember x* (*y* ## *s*) = (*if x* = *y then True else smember x s*)
⟨*proof*⟩

end

lemmas *smap-simps*[*simp*] = *stream.map-sel*
lemmas *shd-sset* = *stream.set-sel*(*1*)
lemmas *stl-sset* = *stream.set-sel*(*2*)

theorem *sset-induct*[*consumes 1*, *case-names shd stl*, *induct set*: *sset*]:
assumes *y* $\in$ *sset s* **and** $\bigwedge s$. *P* (*shd s*) *s* **and** $\bigwedge s\ y$. ⟦*y* $\in$ *sset* (*stl s*); *P y* (*stl s*)⟧ $\Longrightarrow$ *P y s*
shows *P y s*
⟨*proof*⟩

lemma *smap-ctr*: *smap f s* = *x* ## *s'* $\longleftrightarrow$ *f* (*shd s*) = *x* $\wedge$ *smap f* (*stl s*) = *s'*
⟨*proof*⟩

46.1 prepend list to stream

primrec *shift* :: $'a$ *list* $\Rightarrow$ $'a$ *stream* $\Rightarrow$ $'a$ *stream* (**infixr** @− 65) **where**
shift [] *s* = *s*
| *shift* (*x* # *xs*) *s* = *x* ## *shift xs s*

lemma *smap-shift*[*simp*]: *smap f* (*xs* @− *s*) = *map f xs* @− *smap f s*
⟨*proof*⟩

lemma *shift-append*[*simp*]: $(xs \mathbin{@} ys) \mathbin{@-} s = xs \mathbin{@-} ys \mathbin{@-} s$
⟨*proof*⟩

lemma *shift-simps*[*simp*]:
$shd\ (xs \mathbin{@-} s) = (if\ xs = []\ then\ shd\ s\ else\ hd\ xs)$
$stl\ (xs \mathbin{@-} s) = (if\ xs = []\ then\ stl\ s\ else\ tl\ xs \mathbin{@-} s)$
⟨*proof*⟩

lemma *sset-shift*[*simp*]: $sset\ (xs \mathbin{@-} s) = set\ xs \cup sset\ s$
⟨*proof*⟩

lemma *shift-left-inj*[*simp*]: $xs \mathbin{@-} s1 = xs \mathbin{@-} s2 \longleftrightarrow s1 = s2$
⟨*proof*⟩

46.2 set of streams with elements in some fixed set

context
notes [[*inductive-internals*]]
begin

coinductive-set
streams :: $'a\ set \Rightarrow 'a\ stream\ set$
for $A :: 'a\ set$
where
Stream[*intro*!, *simp*, *no-atp*]: $[\![a \in A;\ s \in streams\ A]\!] \Longrightarrow a \mathbin{\#\#} s \in streams\ A$

end

lemma *in-streams*: $stl\ s \in streams\ S \Longrightarrow shd\ s \in S \Longrightarrow s \in streams\ S$
⟨*proof*⟩

lemma *streamsE*: $s \in streams\ A \Longrightarrow (shd\ s \in A \Longrightarrow stl\ s \in streams\ A \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *Stream-image*: $x \mathbin{\#\#} y \in (op \mathbin{\#\#}\ x')\ `\ Y \longleftrightarrow x = x' \wedge y \in Y$
⟨*proof*⟩

lemma *shift-streams*: $[\![w \in lists\ A;\ s \in streams\ A]\!] \Longrightarrow w \mathbin{@-} s \in streams\ A$
⟨*proof*⟩

lemma *streams-Stream*: $x \mathbin{\#\#} s \in streams\ A \longleftrightarrow x \in A \wedge s \in streams\ A$
⟨*proof*⟩

lemma *streams-stl*: $s \in streams\ A \Longrightarrow stl\ s \in streams\ A$
⟨*proof*⟩

lemma *streams-shd*: $s \in streams\ A \Longrightarrow shd\ s \in A$

⟨*proof*⟩

lemma *sset-streams*:
assumes *sset s* ⊆ *A*
shows *s* ∈ *streams A*
⟨*proof*⟩

lemma *streams-sset*:
assumes *s* ∈ *streams A*
shows *sset s* ⊆ *A*
⟨*proof*⟩

lemma *streams-iff-sset*: *s* ∈ *streams A* ⟷ *sset s* ⊆ *A*
⟨*proof*⟩

lemma *streams-mono*: *s* ∈ *streams A* ⟹ *A* ⊆ *B* ⟹ *s* ∈ *streams B*
⟨*proof*⟩

lemma *streams-mono2*: *S* ⊆ *T* ⟹ *streams S* ⊆ *streams T*
⟨*proof*⟩

lemma *smap-streams*: *s* ∈ *streams A* ⟹ (⋀*x*. *x* ∈ *A* ⟹ *f x* ∈ *B*) ⟹ *smap f s* ∈ *streams B*
⟨*proof*⟩

lemma *streams-empty*: *streams* {} = {}
⟨*proof*⟩

lemma *streams-UNIV*[*simp*]: *streams UNIV* = *UNIV*
⟨*proof*⟩

46.3 nth, take, drop for streams

primrec *snth* :: *'a stream* ⇒ *nat* ⇒ *'a* (**infixl** !! *100*) **where**
s !! *0* = *shd s*
| *s* !! *Suc n* = *stl s* !! *n*

lemma *snth-Stream*: (*x* ## *s*) !! *Suc i* = *s* !! *i*
⟨*proof*⟩

lemma *snth-smap*[*simp*]: *smap f s* !! *n* = *f* (*s* !! *n*)
⟨*proof*⟩

lemma *shift-snth-less*[*simp*]: *p* < *length xs* ⟹ (*xs* @− *s*) !! *p* = *xs* ! *p*
⟨*proof*⟩

lemma *shift-snth-ge*[*simp*]: *p* ≥ *length xs* ⟹ (*xs* @− *s*) !! *p* = *s* !! (*p* − *length xs*)
⟨*proof*⟩

lemma *shift-snth*: $(xs\ @{-}\ s)\ !!\ n = (\mathit{if}\ n < \mathit{length}\ xs\ \mathit{then}\ xs\ !\ n\ \mathit{else}\ s\ !!\ (n - \mathit{length}\ xs))$
⟨*proof*⟩

lemma *snth-sset*[*simp*]: $s\ !!\ n \in \mathit{sset}\ s$
⟨*proof*⟩

lemma *sset-range*: $\mathit{sset}\ s = \mathit{range}\ (\mathit{snth}\ s)$
⟨*proof*⟩

lemma *streams-iff-snth*: $s \in \mathit{streams}\ X \longleftrightarrow (\forall\, n.\ s\ !!\ n \in X)$
⟨*proof*⟩

lemma *snth-in*: $s \in \mathit{streams}\ X \Longrightarrow s\ !!\ n \in X$
⟨*proof*⟩

primrec *stake* :: $\mathit{nat} \Rightarrow 'a\ \mathit{stream} \Rightarrow 'a\ \mathit{list}$ **where**
$\mathit{stake}\ 0\ s = []$
| $\mathit{stake}\ (\mathit{Suc}\ n)\ s = \mathit{shd}\ s\ \#\ \mathit{stake}\ n\ (\mathit{stl}\ s)$

lemma *length-stake*[*simp*]: $\mathit{length}\ (\mathit{stake}\ n\ s) = n$
⟨*proof*⟩

lemma *stake-smap*[*simp*]: $\mathit{stake}\ n\ (\mathit{smap}\ f\ s) = \mathit{map}\ f\ (\mathit{stake}\ n\ s)$
⟨*proof*⟩

lemma *take-stake*: $\mathit{take}\ n\ (\mathit{stake}\ m\ s) = \mathit{stake}\ (\mathit{min}\ n\ m)\ s$
⟨*proof*⟩

primrec *sdrop* :: $\mathit{nat} \Rightarrow 'a\ \mathit{stream} \Rightarrow 'a\ \mathit{stream}$ **where**
$\mathit{sdrop}\ 0\ s = s$
| $\mathit{sdrop}\ (\mathit{Suc}\ n)\ s = \mathit{sdrop}\ n\ (\mathit{stl}\ s)$

lemma *sdrop-simps*[*simp*]:
$\mathit{shd}\ (\mathit{sdrop}\ n\ s) = s\ !!\ n\ \mathit{stl}\ (\mathit{sdrop}\ n\ s) = \mathit{sdrop}\ (\mathit{Suc}\ n)\ s$
⟨*proof*⟩

lemma *sdrop-smap*[*simp*]: $\mathit{sdrop}\ n\ (\mathit{smap}\ f\ s) = \mathit{smap}\ f\ (\mathit{sdrop}\ n\ s)$
⟨*proof*⟩

lemma *sdrop-stl*: $\mathit{sdrop}\ n\ (\mathit{stl}\ s) = \mathit{stl}\ (\mathit{sdrop}\ n\ s)$
⟨*proof*⟩

lemma *drop-stake*: $\mathit{drop}\ n\ (\mathit{stake}\ m\ s) = \mathit{stake}\ (m - n)\ (\mathit{sdrop}\ n\ s)$
⟨*proof*⟩

lemma *stake-sdrop*: $\mathit{stake}\ n\ s\ @{-}\ \mathit{sdrop}\ n\ s = s$
⟨*proof*⟩

lemma *id-stake-snth-sdrop*:
s = stake i s @− s !! i ## sdrop (Suc i) s
⟨*proof*⟩

lemma *smap-alt*: *smap f s = s' ⟷ (∀ n. f (s !! n) = s' !! n)* (**is** *?L = ?R*)
⟨*proof*⟩

lemma *stake-invert-Nil*[*iff*]: *stake n s = [] ⟷ n = 0*
⟨*proof*⟩

lemma *sdrop-shift*: *sdrop i (w @− s) = drop i w @− sdrop (i − length w) s*
⟨*proof*⟩

lemma *stake-shift*: *stake i (w @− s) = take i w @ stake (i − length w) s*
⟨*proof*⟩

lemma *stake-add*[*simp*]: *stake m s @ stake n (sdrop m s) = stake (m + n) s*
⟨*proof*⟩

lemma *sdrop-add*[*simp*]: *sdrop n (sdrop m s) = sdrop (m + n) s*
⟨*proof*⟩

lemma *sdrop-snth*: *sdrop n s !! m = s !! (n + m)*
⟨*proof*⟩

partial-function (*tailrec*) *sdrop-while* :: *('a ⇒ bool) ⇒ 'a stream ⇒ 'a stream*
where
sdrop-while P s = (if P (shd s) then sdrop-while P (stl s) else s)

lemma *sdrop-while-SCons*[*code*]:
sdrop-while P (a ## s) = (if P a then sdrop-while P s else a ## s)
⟨*proof*⟩

lemma *sdrop-while-sdrop-LEAST*:
assumes *∃ n. P (s !! n)*
shows *sdrop-while (Not o P) s = sdrop (LEAST n. P (s !! n)) s*
⟨*proof*⟩

primcorec *sfilter* **where**
shd (sfilter P s) = shd (sdrop-while (Not o P) s)
| *stl (sfilter P s) = sfilter P (stl (sdrop-while (Not o P) s))*

lemma *sfilter-Stream*: *sfilter P (x ## s) = (if P x then x ## sfilter P s else sfilter P s)*
⟨*proof*⟩

46.4 unary predicates lifted to streams

definition *stream-all P s* = (∀ *p*. *P* (*s* !! *p*))

lemma *stream-all-iff*[*iff*]: *stream-all P s* ⟷ *Ball* (*sset s*) *P*
⟨*proof*⟩

lemma *stream-all-shift*[*simp*]: *stream-all P* (*xs* @− *s*) = (*list-all P xs* ∧ *stream-all P s*)
⟨*proof*⟩

lemma *stream-all-Stream*: *stream-all P* (*x* ## *X*) ⟷ *P x* ∧ *stream-all P X*
⟨*proof*⟩

46.5 recurring stream out of a list

primcorec *cycle* :: *'a list* ⇒ *'a stream* **where**
shd (*cycle xs*) = *hd xs*
| *stl* (*cycle xs*) = *cycle* (*tl xs* @ [*hd xs*])

lemma *cycle-decomp*: *u* ≠ [] ⟹ *cycle u* = *u* @− *cycle u*
⟨*proof*⟩

lemma *cycle-Cons*[*code*]: *cycle* (*x* # *xs*) = *x* ## *cycle* (*xs* @ [*x*])
⟨*proof*⟩

lemma *cycle-rotated*: ⟦*v* ≠ []; *cycle u* = *v* @− *s*⟧ ⟹ *cycle* (*tl u* @ [*hd u*]) = *tl v* @− *s*
⟨*proof*⟩

lemma *stake-append*: *stake n* (*u* @− *s*) = *take* (*min* (*length u*) *n*) *u* @ *stake* (*n* − *length u*) *s*
⟨*proof*⟩

lemma *stake-cycle-le*[*simp*]:
assumes *u* ≠ [] *n* < *length u*
shows *stake n* (*cycle u*) = *take n u*
⟨*proof*⟩

lemma *stake-cycle-eq*[*simp*]: *u* ≠ [] ⟹ *stake* (*length u*) (*cycle u*) = *u*
⟨*proof*⟩

lemma *sdrop-cycle-eq*[*simp*]: *u* ≠ [] ⟹ *sdrop* (*length u*) (*cycle u*) = *cycle u*
⟨*proof*⟩

lemma *stake-cycle-eq-mod-0*[*simp*]: ⟦*u* ≠ []; *n mod length u* = *0*⟧ ⟹
stake n (*cycle u*) = *concat* (*replicate* (*n div length u*) *u*)
⟨*proof*⟩

lemma *sdrop-cycle-eq-mod-0*[*simp*]: ⟦*u* ≠ []; *n mod length u* = *0*⟧ ⟹

sdrop n (cycle u) = cycle u
⟨*proof*⟩

lemma *stake-cycle*: *u ≠ [] ⟹*
stake n (cycle u) = concat (replicate (n div length u) u) @ take (n mod length u) u
⟨*proof*⟩

lemma *sdrop-cycle*: *u ≠ [] ⟹ sdrop n (cycle u) = cycle (rotate (n mod length u) u)*
⟨*proof*⟩

lemma *sset-cycle*[*simp*]:
assumes *xs ≠ []*
shows *sset (cycle xs) = set xs*
⟨*proof*⟩

46.6 iterated application of a function

primcorec *siterate* **where**
shd (siterate f x) = x
| *stl (siterate f x) = siterate f (f x)*

lemma *stake-Suc*: *stake (Suc n) s = stake n s @ [s !! n]*
⟨*proof*⟩

lemma *snth-siterate*[*simp*]: *siterate f x !! n = (f^^n) x*
⟨*proof*⟩

lemma *sdrop-siterate*[*simp*]: *sdrop n (siterate f x) = siterate f ((f^^n) x)*
⟨*proof*⟩

lemma *stake-siterate*[*simp*]: *stake n (siterate f x) = map (λn. (f^^n) x) [0 ..< n]*
⟨*proof*⟩

lemma *sset-siterate*: *sset (siterate f x) = {(f^^n) x | n. True}*
⟨*proof*⟩

lemma *smap-siterate*: *smap f (siterate f x) = siterate f (f x)*
⟨*proof*⟩

46.7 stream repeating a single element

abbreviation *sconst ≡ siterate id*

lemma *shift-replicate-sconst*[*simp*]: *replicate n x @− sconst x = sconst x*
⟨*proof*⟩

lemma *sset-sconst*[*simp*]: *sset (sconst x) = {x}*
⟨*proof*⟩

lemma *sconst-alt*: $s = sconst\ x \longleftrightarrow sset\ s = \{x\}$
⟨*proof*⟩

lemma *sconst-cycle*: $sconst\ x = cycle\ [x]$
⟨*proof*⟩

lemma *smap-sconst*: $smap\ f\ (sconst\ x) = sconst\ (f\ x)$
⟨*proof*⟩

lemma *sconst-streams*: $x \in A \Longrightarrow sconst\ x \in streams\ A$
⟨*proof*⟩

lemma *streams-empty-iff*: $streams\ S = \{\} \longleftrightarrow S = \{\}$
⟨*proof*⟩

46.8 stream of natural numbers

abbreviation $fromN \equiv siterate\ Suc$

abbreviation $nats \equiv fromN\ 0$

lemma *sset-fromN*[*simp*]: $sset\ (fromN\ n) = \{n\ ..\}$
⟨*proof*⟩

lemma *stream-smap-fromN*: $s = smap\ (\lambda j.\ let\ i = j - n\ in\ s\ !!\ i)\ (fromN\ n)$
⟨*proof*⟩

lemma *stream-smap-nats*: $s = smap\ (snth\ s)\ nats$
⟨*proof*⟩

46.9 flatten a stream of lists

primcorec *flat* **where**
$shd\ (flat\ ws) = hd\ (shd\ ws)$
| $stl\ (flat\ ws) = flat\ (if\ tl\ (shd\ ws) = []\ then\ stl\ ws\ else\ tl\ (shd\ ws)\ \#\#\ stl\ ws)$

lemma *flat-Cons*[*simp*, *code*]: $flat\ ((x\ \#\ xs)\ \#\#\ ws) = x\ \#\#\ flat\ (if\ xs = []\ then\ ws\ else\ xs\ \#\#\ ws)$
⟨*proof*⟩

lemma *flat-Stream*[*simp*]: $xs \neq [] \Longrightarrow flat\ (xs\ \#\#\ ws) = xs\ @-\ flat\ ws$
⟨*proof*⟩

lemma *flat-unfold*: $shd\ ws \neq [] \Longrightarrow flat\ ws = shd\ ws\ @-\ flat\ (stl\ ws)$
⟨*proof*⟩

lemma *flat-snth*: $\forall xs \in sset\ s.\ xs \neq [] \Longrightarrow flat\ s\ !!\ n = (if\ n < length\ (shd\ s)\ then\ shd\ s\ !\ n\ else\ flat\ (stl\ s)\ !!\ (n - length\ (shd\ s)))$
⟨*proof*⟩

lemma *sset-flat*[*simp*]: $\forall xs \in sset\ s.\ xs \neq [] \Longrightarrow$
sset (*flat s*) = $(\bigcup xs \in sset\ s.\ set\ xs)$ (**is** *?P* $\Longrightarrow$ *?L* = *?R*)
⟨*proof*⟩

46.10 merge a stream of streams

definition *smerge* :: *'a stream stream* ⇒ *'a stream* **where**
smerge ss = *flat* (*smap* (λ*n*. *map* (λ*s*. *s* !! *n*) (*stake* (*Suc n*) *ss*) @ *stake n* (*ss* !! *n*)) *nats*)

lemma *stake-nth*[*simp*]: *m* < *n* ⟹ *stake n s* ! *m* = *s* !! *m*
⟨*proof*⟩

lemma *snth-sset-smerge*: *ss* !! *n* !! *m* ∈ *sset* (*smerge ss*)
⟨*proof*⟩

lemma *sset-smerge*: *sset* (*smerge ss*) = *UNION* (*sset ss*) *sset*
⟨*proof*⟩

46.11 product of two streams

definition *sproduct* :: *'a stream* ⇒ *'b stream* ⇒ (*'a* × *'b*) *stream* **where**
sproduct s1 s2 = *smerge* (*smap* (λ*x*. *smap* (*Pair x*) *s2*) *s1*)

lemma *sset-sproduct*: *sset* (*sproduct s1 s2*) = *sset s1* × *sset s2*
⟨*proof*⟩

46.12 interleave two streams

primcorec *sinterleave* **where**
shd (*sinterleave s1 s2*) = *shd s1*
| *stl* (*sinterleave s1 s2*) = *sinterleave s2* (*stl s1*)

lemma *sinterleave-code*[*code*]:
sinterleave (*x* ## *s1*) *s2* = *x* ## *sinterleave s2 s1*
⟨*proof*⟩

lemma *sinterleave-snth*[*simp*]:
even n ⟹ *sinterleave s1 s2* !! *n* = *s1* !! (*n div 2*)
odd n ⟹ *sinterleave s1 s2* !! *n* = *s2* !! (*n div 2*)
⟨*proof*⟩

lemma *sset-sinterleave*: *sset* (*sinterleave s1 s2*) = *sset s1* ∪ *sset s2*
⟨*proof*⟩

46.13 zip

primcorec *szip* **where**
shd (*szip s1 s2*) = (*shd s1*, *shd s2*)

| *stl* (*szip s1 s2*) = *szip* (*stl s1*) (*stl s2*)

lemma *szip-unfold*[*code*]: *szip* (*a ## s1*) (*b ## s2*) = (*a*, *b*) *##* (*szip s1 s2*)
⟨*proof*⟩

lemma *snth-szip*[*simp*]: *szip s1 s2* !! *n* = (*s1* !! *n*, *s2* !! *n*)
⟨*proof*⟩

lemma *stake-szip*[*simp*]:
stake n (*szip s1 s2*) = *zip* (*stake n s1*) (*stake n s2*)
⟨*proof*⟩

lemma *sdrop-szip*[*simp*]: *sdrop n* (*szip s1 s2*) = *szip* (*sdrop n s1*) (*sdrop n s2*)
⟨*proof*⟩

lemma *smap-szip-fst*:
smap (λ*x*. *f* (*fst x*)) (*szip s1 s2*) = *smap f s1*
⟨*proof*⟩

lemma *smap-szip-snd*:
smap (λ*x*. *g* (*snd x*)) (*szip s1 s2*) = *smap g s2*
⟨*proof*⟩

46.14 zip via function

primcorec *smap2* **where**
shd (*smap2 f s1 s2*) = *f* (*shd s1*) (*shd s2*)
| *stl* (*smap2 f s1 s2*) = *smap2 f* (*stl s1*) (*stl s2*)

lemma *smap2-unfold*[*code*]:
smap2 f (*a ## s1*) (*b ## s2*) = *f a b ##* (*smap2 f s1 s2*)
⟨*proof*⟩

lemma *smap2-szip*:
smap2 f s1 s2 = *smap* (*case-prod f*) (*szip s1 s2*)
⟨*proof*⟩

lemma *smap-smap2*[*simp*]:
smap f (*smap2 g s1 s2*) = *smap2* (λ*x y*. *f* (*g x y*)) *s1 s2*
⟨*proof*⟩

lemma *smap2-alt*:
(*smap2 f s1 s2* = *s*) = (∀ *n*. *f* (*s1* !! *n*) (*s2* !! *n*) = *s* !! *n*)
⟨*proof*⟩

lemma *snth-smap2*[*simp*]:
smap2 f s1 s2 !! *n* = *f* (*s1* !! *n*) (*s2* !! *n*)
⟨*proof*⟩

lemma *stake-smap2*[*simp*]:
stake n (*smap2 f s1 s2*) = *map* (*case-prod f*) (*zip* (*stake n s1*) (*stake n s2*))
⟨*proof*⟩

lemma *sdrop-smap2*[*simp*]:
sdrop n (*smap2 f s1 s2*) = *smap2 f* (*sdrop n s1*) (*sdrop n s2*)
⟨*proof*⟩

end

47 List prefixes, suffixes, and homeomorphic embedding

theory *Sublist*
imports *Main*
begin

47.1 Prefix order on lists

definition *prefix* :: *'a list* ⇒ *'a list* ⇒ *bool*
where *prefix xs ys* ⟷ (∃*zs*. *ys* = *xs* @ *zs*)

definition *strict-prefix* :: *'a list* ⇒ *'a list* ⇒ *bool*
where *strict-prefix xs ys* ⟷ *prefix xs ys* ∧ *xs* ≠ *ys*

interpretation *prefix-order*: *order prefix strict-prefix*
⟨*proof*⟩

interpretation *prefix-bot*: *order-bot Nil prefix strict-prefix*
⟨*proof*⟩

lemma *prefixI* [*intro?*]: *ys* = *xs* @ *zs* ⟹ *prefix xs ys*
⟨*proof*⟩

lemma *prefixE* [*elim?*]:
assumes *prefix xs ys*
obtains *zs* **where** *ys* = *xs* @ *zs*
⟨*proof*⟩

lemma *strict-prefixI'* [*intro?*]: *ys* = *xs* @ *z* # *zs* ⟹ *strict-prefix xs ys*
⟨*proof*⟩

lemma *strict-prefixE'* [*elim?*]:
assumes *strict-prefix xs ys*
obtains *z zs* **where** *ys* = *xs* @ *z* # *zs*
⟨*proof*⟩

lemma *strict-prefixI* [*intro?*]: *prefix xs ys* $\Longrightarrow$ *xs* $\neq$ *ys* $\Longrightarrow$ *strict-prefix xs ys*
⟨*proof*⟩

lemma *strict-prefixE* [*elim?*]:
fixes *xs ys* :: *'a list*
assumes *strict-prefix xs ys*
obtains *prefix xs ys* **and** *xs* $\neq$ *ys*
⟨*proof*⟩

47.2 Basic properties of prefixes

theorem *Nil-prefix* [*simp*]: *prefix* [] *xs*
⟨*proof*⟩

theorem *prefix-Nil* [*simp*]: (*prefix xs* []) = (*xs* = [])
⟨*proof*⟩

lemma *prefix-snoc* [*simp*]: *prefix xs* (*ys* @ [*y*]) $\longleftrightarrow$ *xs* = *ys* @ [*y*] $\vee$ *prefix xs ys*
⟨*proof*⟩

lemma *Cons-prefix-Cons* [*simp*]: *prefix* (*x* # *xs*) (*y* # *ys*) = (*x* = *y* $\wedge$ *prefix xs ys*)
⟨*proof*⟩

lemma *prefix-code* [*code*]:
prefix [] *xs* $\longleftrightarrow$ *True*
prefix (*x* # *xs*) [] $\longleftrightarrow$ *False*
prefix (*x* # *xs*) (*y* # *ys*) $\longleftrightarrow$ *x* = *y* $\wedge$ *prefix xs ys*
⟨*proof*⟩

lemma *same-prefix-prefix* [*simp*]: *prefix* (*xs* @ *ys*) (*xs* @ *zs*) = *prefix ys zs*
⟨*proof*⟩

lemma *same-prefix-nil* [*simp*]: *prefix* (*xs* @ *ys*) *xs* = (*ys* = [])
⟨*proof*⟩

lemma *prefix-prefix* [*simp*]: *prefix xs ys* $\Longrightarrow$ *prefix xs* (*ys* @ *zs*)
⟨*proof*⟩

lemma *append-prefixD*: *prefix* (*xs* @ *ys*) *zs* $\Longrightarrow$ *prefix xs zs*
⟨*proof*⟩

theorem *prefix-Cons*: *prefix xs* (*y* # *ys*) = (*xs* = [] $\vee$ ($\exists$ *zs*. *xs* = *y* # *zs* $\wedge$ *prefix zs ys*))
⟨*proof*⟩

theorem *prefix-append*:
prefix xs (*ys* @ *zs*) = (*prefix xs ys* $\vee$ ($\exists$ *us*. *xs* = *ys* @ *us* $\wedge$ *prefix us zs*))

⟨*proof*⟩

lemma *append-one-prefix*:
prefix xs ys ⟹ *length xs* < *length ys* ⟹ *prefix* (*xs* @ [*ys* ! *length xs*]) *ys*
⟨*proof*⟩

theorem *prefix-length-le*: *prefix xs ys* ⟹ *length xs* ≤ *length ys*
⟨*proof*⟩

lemma *prefix-same-cases*:
prefix (*xs*$_1$::'*a list*) *ys* ⟹ *prefix xs*$_2$ *ys* ⟹ *prefix xs*$_1$ *xs*$_2$ ∨ *prefix xs*$_2$ *xs*$_1$
⟨*proof*⟩

lemma *prefix-length-prefix*:
prefix ps xs ⟹ *prefix qs xs* ⟹ *length ps* ≤ *length qs* ⟹ *prefix ps qs*
⟨*proof*⟩

lemma *set-mono-prefix*: *prefix xs ys* ⟹ *set xs* ⊆ *set ys*
⟨*proof*⟩

lemma *take-is-prefix*: *prefix* (*take n xs*) *xs*
⟨*proof*⟩

lemma *prefixeq-butlast*: *prefix* (*butlast xs*) *xs*
⟨*proof*⟩

lemma *map-prefixI*: *prefix xs ys* ⟹ *prefix* (*map f xs*) (*map f ys*)
⟨*proof*⟩

lemma *prefix-length-less*: *strict-prefix xs ys* ⟹ *length xs* < *length ys*
⟨*proof*⟩

lemma *prefix-snocD*: *prefix* (*xs*@[*x*]) *ys* ⟹ *strict-prefix xs ys*
⟨*proof*⟩

lemma *strict-prefix-simps* [*simp*, *code*]:
strict-prefix xs [] ⟷ *False*
strict-prefix [] (*x* # *xs*) ⟷ *True*
strict-prefix (*x* # *xs*) (*y* # *ys*) ⟷ *x* = *y* ∧ *strict-prefix xs ys*
⟨*proof*⟩

lemma *take-strict-prefix*: *strict-prefix xs ys* ⟹ *strict-prefix* (*take n xs*) *ys*
⟨*proof*⟩

lemma *not-prefix-cases*:
assumes *pfx*: ¬ *prefix ps ls*
obtains
(*c1*) *ps* ≠ [] **and** *ls* = []
| (*c2*) *a as x xs* **where** *ps* = *a*#*as* **and** *ls* = *x*#*xs* **and** *x* = *a* **and** ¬ *prefix as*

xs
| (*c3*) *a as x xs* **where** *ps* = *a*#*as* **and** *ls* = *x*#*xs* **and** *x* ≠ *a*
⟨*proof*⟩

lemma *not-prefix-induct* [*consumes 1*, *case-names Nil Neq Eq*]:
assumes *np*: ¬ *prefix ps ls*
and *base*: ⋀*x xs*. *P* (*x*#*xs*) []
and *r1*: ⋀*x xs y ys*. *x* ≠ *y* ⟹ *P* (*x*#*xs*) (*y*#*ys*)
and *r2*: ⋀*x xs y ys*. ⟦ *x* = *y*; ¬ *prefix xs ys*; *P xs ys* ⟧ ⟹ *P* (*x*#*xs*) (*y*#*ys*)
shows *P ps ls* ⟨*proof*⟩

47.3 Prefixes

primrec *prefixes* **where**
prefixes [] = [[]] |
prefixes (*x*#*xs*) = [] # *map* (*op* # *x*) (*prefixes xs*)

lemma *in-set-prefixes*[*simp*]: *xs* ∈ *set* (*prefixes ys*) ⟷ *prefix xs ys*
⟨*proof*⟩

lemma *length-prefixes*[*simp*]: *length* (*prefixes xs*) = *length xs*+*1*
⟨*proof*⟩

lemma *distinct-prefixes* [*intro*]: *distinct* (*prefixes xs*)
⟨*proof*⟩

lemma *prefixes-snoc* [*simp*]: *prefixes* (*xs*@[*x*]) = *prefixes xs* @ [*xs*@[*x*]]
⟨*proof*⟩

lemma *prefixes-not-Nil* [*simp*]: *prefixes xs* ≠ []
⟨*proof*⟩

lemma *hd-prefixes* [*simp*]: *hd* (*prefixes xs*) = []
⟨*proof*⟩

lemma *last-prefixes* [*simp*]: *last* (*prefixes xs*) = *xs*
⟨*proof*⟩

lemma *prefixes-append*:
prefixes (*xs* @ *ys*) = *prefixes xs* @ *map* (λ*ys'*. *xs* @ *ys'*) (*tl* (*prefixes ys*))
⟨*proof*⟩

lemma *prefixes-eq-snoc*:
prefixes ys = *xs* @ [*x*] ⟷
(*ys* = [] ∧ *xs* = [] ∨ (∃ *z zs*. *ys* = *zs*@[*z*] ∧ *xs* = *prefixes zs*)) ∧ *x* = *ys*
⟨*proof*⟩

lemma *prefixes-tailrec* [*code*]:
prefixes xs = *rev* (*snd* (*foldl* (λ(*acc1*, *acc2*) *x*. (*x*#*acc1*, *rev* (*x*#*acc1*)#*acc2*))

([],[[]]) *xs*))
⟨*proof*⟩

lemma *set-prefixes-eq*: *set* (*prefixes xs*) = {*ys*. *prefix ys xs*}
⟨*proof*⟩

lemma *card-set-prefixes* [*simp*]: *card* (*set* (*prefixes xs*)) = *Suc* (*length xs*)
⟨*proof*⟩

lemma *set-prefixes-append*:
set (*prefixes* (*xs* @ *ys*)) = *set* (*prefixes xs*) ∪ {*xs* @ *ys'* |*ys'*. *ys'* ∈ *set* (*prefixes ys*)}
⟨*proof*⟩

47.4 Longest Common Prefix

definition *Longest-common-prefix* :: *'a list set* ⇒ *'a list* **where**
Longest-common-prefix L = (*ARG-MAX length ps*. ∀ *xs* ∈ *L*. *prefix ps xs*)

lemma *Longest-common-prefix-ex*: *L* ≠ {} ⟹
∃ *ps*. (∀ *xs* ∈ *L*. *prefix ps xs*) ∧ (∀ *qs*. (∀ *xs* ∈ *L*. *prefix qs xs*) ⟶ *size qs* ≤ *size ps*)
(**is** - ⟹ ∃ *ps*. *?P L ps*)
⟨*proof*⟩

lemma *Longest-common-prefix-unique*: *L* ≠ {} ⟹
∃! *ps*. (∀ *xs* ∈ *L*. *prefix ps xs*) ∧ (∀ *qs*. (∀ *xs* ∈ *L*. *prefix qs xs*) ⟶ *size qs* ≤ *size ps*)
⟨*proof*⟩

lemma *Longest-common-prefix-eq*:
⟦ *L* ≠ {}; ∀ *xs* ∈ *L*. *prefix ps xs*;
∀ *qs*. (∀ *xs* ∈ *L*. *prefix qs xs*) ⟶ *size qs* ≤ *size ps* ⟧
⟹ *Longest-common-prefix L* = *ps*
⟨*proof*⟩

lemma *Longest-common-prefix-prefix*:
xs ∈ *L* ⟹ *prefix* (*Longest-common-prefix L*) *xs*
⟨*proof*⟩

lemma *Longest-common-prefix-longest*:
L ≠ {} ⟹ ∀ *xs*∈*L*. *prefix ps xs* ⟹ *length ps* ≤ *length*(*Longest-common-prefix L*)
⟨*proof*⟩

lemma *Longest-common-prefix-max-prefix*:
L ≠ {} ⟹ ∀ *xs*∈*L*. *prefix ps xs* ⟹ *prefix ps* (*Longest-common-prefix L*)
⟨*proof*⟩

lemma *Longest-common-prefix-Nil*: $[] \in L \Longrightarrow$ *Longest-common-prefix* $L = []$
⟨*proof*⟩

lemma *Longest-common-prefix-image-Cons*: $L \neq \{\} \Longrightarrow$
Longest-common-prefix $(op\ \#\ x\ `\ L) = x\ \#$ *Longest-common-prefix* L
⟨*proof*⟩

lemma *Longest-common-prefix-eq-Cons*: **assumes** $L \neq \{\}$ $[] \notin L$ $\forall xs{\in}L.\ hd\ xs = x$
shows *Longest-common-prefix* $L = x\ \#$ *Longest-common-prefix* $\{ys.\ x\#ys \in L\}$
⟨*proof*⟩

lemma *Longest-common-prefix-eq-Nil*:
$[\![x\#ys \in L;\ y\#zs \in L;\ x \neq y\ ]\!] \Longrightarrow$ *Longest-common-prefix* $L = []$
⟨*proof*⟩

fun *longest-common-prefix* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
longest-common-prefix $(x\#xs)\ (y\#ys) =$
(*if* $x{=}y$ *then* $x\ \#$ *longest-common-prefix xs ys else* $[]$) |
longest-common-prefix - - = []

lemma *longest-common-prefix-prefix1*:
prefix (*longest-common-prefix xs ys*) *xs*
⟨*proof*⟩

lemma *longest-common-prefix-prefix2*:
prefix (*longest-common-prefix xs ys*) *ys*
⟨*proof*⟩

lemma *longest-common-prefix-max-prefix*:
$[\![$ *prefix ps xs*; *prefix ps ys* $]\!]$
$\Longrightarrow$ *prefix ps* (*longest-common-prefix xs ys*)
⟨*proof*⟩

47.5 Parallel lists

definition *parallel* :: *'a list* ⇒ *'a list* ⇒ *bool* (**infixl** ∥ *50*)
where $(xs \parallel ys) = (\neg$ *prefix xs ys* $\wedge \neg$ *prefix ys xs*$)$

lemma *parallelI* [*intro*]: ¬ *prefix xs ys* $\Longrightarrow \neg$ *prefix ys xs* $\Longrightarrow xs \parallel ys$
⟨*proof*⟩

lemma *parallelE* [*elim*]:
assumes $xs \parallel ys$
obtains ¬ *prefix xs ys* $\wedge \neg$ *prefix ys xs*
⟨*proof*⟩

theorem *prefix-cases*:

obtains *prefix xs ys* | *strict-prefix ys xs* | *xs* ∥ *ys*
⟨*proof*⟩

theorem *parallel-decomp*:
xs ∥ *ys* ⟹ ∃ *as b bs c cs*. *b* ≠ *c* ∧ *xs* = *as* @ *b* # *bs* ∧ *ys* = *as* @ *c* # *cs*
⟨*proof*⟩

lemma *parallel-append*: *a* ∥ *b* ⟹ *a* @ *c* ∥ *b* @ *d*
⟨*proof*⟩

lemma *parallel-appendI*: *xs* ∥ *ys* ⟹ *x* = *xs* @ *xs'* ⟹ *y* = *ys* @ *ys'* ⟹ *x* ∥ *y*
⟨*proof*⟩

lemma *parallel-commute*: *a* ∥ *b* ⟷ *b* ∥ *a*
⟨*proof*⟩

47.6 Suffix order on lists

definition *suffix* :: *'a list* ⇒ *'a list* ⇒ *bool*
where *suffix xs ys* = (∃ *zs*. *ys* = *zs* @ *xs*)

definition *strict-suffix* :: *'a list* ⇒ *'a list* ⇒ *bool*
where *strict-suffix xs ys* ⟷ *suffix xs ys* ∧ *xs* ≠ *ys*

interpretation *suffix-order*: *order suffix strict-suffix*
⟨*proof*⟩

interpretation *suffix-bot*: *order-bot Nil suffix strict-suffix*
⟨*proof*⟩

lemma *suffixI* [*intro?*]: *ys* = *zs* @ *xs* ⟹ *suffix xs ys*
⟨*proof*⟩

lemma *suffixE* [*elim?*]:
assumes *suffix xs ys*
obtains *zs* **where** *ys* = *zs* @ *xs*
⟨*proof*⟩

lemma *suffix-tl* [*simp*]: *suffix* (*tl xs*) *xs*
⟨*proof*⟩

lemma *strict-suffix-tl* [*simp*]: *xs* ≠ [] ⟹ *strict-suffix* (*tl xs*) *xs*
⟨*proof*⟩

lemma *Nil-suffix* [*simp*]: *suffix* [] *xs*
⟨*proof*⟩

lemma *suffix-Nil* [*simp*]: (*suffix xs* []) = (*xs* = [])
⟨*proof*⟩

lemma *suffix-ConsI*: *suffix xs ys* $\Longrightarrow$ *suffix xs* (*y* # *ys*)
⟨*proof*⟩

lemma *suffix-ConsD*: *suffix* (*x* # *xs*) *ys* $\Longrightarrow$ *suffix xs ys*
⟨*proof*⟩

lemma *suffix-appendI*: *suffix xs ys* $\Longrightarrow$ *suffix xs* (*zs* @ *ys*)
⟨*proof*⟩

lemma *suffix-appendD*: *suffix* (*zs* @ *xs*) *ys* $\Longrightarrow$ *suffix xs ys*
⟨*proof*⟩

lemma *strict-suffix-set-subset*: *strict-suffix xs ys* $\Longrightarrow$ *set xs* $\subseteq$ *set ys*
⟨*proof*⟩

lemma *suffix-set-subset*: *suffix xs ys* $\Longrightarrow$ *set xs* $\subseteq$ *set ys*
⟨*proof*⟩

lemma *suffix-ConsD2*: *suffix* (*x* # *xs*) (*y* # *ys*) $\Longrightarrow$ *suffix xs ys*
⟨*proof*⟩

lemma *suffix-to-prefix* [*code*]: *suffix xs ys* $\longleftrightarrow$ *prefix* (*rev xs*) (*rev ys*)
⟨*proof*⟩

lemma *strict-suffix-to-prefix* [*code*]: *strict-suffix xs ys* $\longleftrightarrow$ *strict-prefix* (*rev xs*) (*rev ys*)
⟨*proof*⟩

lemma *distinct-suffix*: *distinct ys* $\Longrightarrow$ *suffix xs ys* $\Longrightarrow$ *distinct xs*
⟨*proof*⟩

lemma *suffix-map*: *suffix xs ys* $\Longrightarrow$ *suffix* (*map f xs*) (*map f ys*)
⟨*proof*⟩

lemma *suffix-drop*: *suffix* (*drop n as*) *as*
⟨*proof*⟩

lemma *suffix-take*: *suffix xs ys* $\Longrightarrow$ *ys* = *take* (*length ys* − *length xs*) *ys* @ *xs*
⟨*proof*⟩

lemma *strict-suffix-reflclp-conv*: *strict-suffix*$^{==}$ = *suffix*
⟨*proof*⟩

lemma *suffix-lists*: *suffix xs ys* $\Longrightarrow$ *ys* $\in$ *lists A* $\Longrightarrow$ *xs* $\in$ *lists A*
⟨*proof*⟩

lemma *suffix-snoc* [*simp*]: *suffix xs* (*ys* @ [*y*]) $\longleftrightarrow$ *xs* = [] $\vee$ ($\exists$ *zs*. *xs* = *zs* @ [*y*] $\wedge$ *suffix zs ys*)

⟨*proof*⟩

lemma *snoc-suffix-snoc* [*simp*]: *suffix* (*xs* @ [*x*]) (*ys* @ [*y*]) = (*x* = *y* ∧ *suffix xs ys*)
⟨*proof*⟩

lemma *same-suffix-suffix* [*simp*]: *suffix* (*ys* @ *xs*) (*zs* @ *xs*) = *suffix ys zs*
⟨*proof*⟩

lemma *same-suffix-nil* [*simp*]: *suffix* (*ys* @ *xs*) *xs* = (*ys* = [])
⟨*proof*⟩

theorem *suffix-Cons*: *suffix xs* (*y* # *ys*) ⟷ *xs* = *y* # *ys* ∨ *suffix xs ys*
⟨*proof*⟩

theorem *suffix-append*:
suffix xs (*ys* @ *zs*) ⟷ *suffix xs zs* ∨ (∃ *xs'*. *xs* = *xs'* @ *zs* ∧ *suffix xs' ys*)
⟨*proof*⟩

theorem *suffix-length-le*: *suffix xs ys* ⟹ *length xs* ≤ *length ys*
⟨*proof*⟩

lemma *suffix-same-cases*:
suffix (*xs*$_1$::'*a list*) *ys* ⟹ *suffix xs*$_2$ *ys* ⟹ *suffix xs*$_1$ *xs*$_2$ ∨ *suffix xs*$_2$ *xs*$_1$
⟨*proof*⟩

lemma *suffix-length-suffix*:
suffix ps xs ⟹ *suffix qs xs* ⟹ *length ps* ≤ *length qs* ⟹ *suffix ps qs*
⟨*proof*⟩

lemma *suffix-length-less*: *strict-suffix xs ys* ⟹ *length xs* < *length ys*
⟨*proof*⟩

lemma *suffix-ConsD'*: *suffix* (*x*#*xs*) *ys* ⟹ *strict-suffix xs ys*
⟨*proof*⟩

lemma *drop-strict-suffix*: *strict-suffix xs ys* ⟹ *strict-suffix* (*drop n xs*) *ys*
⟨*proof*⟩

lemma *not-suffix-cases*:
assumes *pfx*: ¬ *suffix ps ls*
obtains
(*c1*) *ps* ≠ [] **and** *ls* = []
| (*c2*) *a as x xs* **where** *ps* = *as*@[*a*] **and** *ls* = *xs*@[*x*] **and** *x* = *a* **and** ¬ *suffix as xs*
| (*c3*) *a as x xs* **where** *ps* = *as*@[*a*] **and** *ls* = *xs*@[*x*] **and** *x* ≠ *a*
⟨*proof*⟩

lemma *not-suffix-induct* [*consumes 1*, *case-names Nil Neq Eq*]:

assumes *np*: ¬ *suffix ps ls*
and *base*: ⋀*x xs*. *P* (*xs*@[*x*]) []
and *r1*: ⋀*x xs y ys*. *x* ≠ *y* ⟹ *P* (*xs*@[*x*]) (*ys*@[*y*])
and *r2*: ⋀*x xs y ys*. ⟦ *x* = *y*; ¬ *suffix xs ys*; *P xs ys* ⟧ ⟹ *P* (*xs*@[*x*]) (*ys*@[*y*])
shows *P ps ls* ⟨*proof*⟩

lemma *parallelD1*: *x* ∥ *y* ⟹ ¬ *prefix x y*
⟨*proof*⟩

lemma *parallelD2*: *x* ∥ *y* ⟹ ¬ *prefix y x*
⟨*proof*⟩

lemma *parallel-Nil1* [*simp*]: ¬ *x* ∥ []
⟨*proof*⟩

lemma *parallel-Nil2* [*simp*]: ¬ [] ∥ *x*
⟨*proof*⟩

lemma *Cons-parallelI1*: *a* ≠ *b* ⟹ *a* # *as* ∥ *b* # *bs*
⟨*proof*⟩

lemma *Cons-parallelI2*: ⟦ *a* = *b*; *as* ∥ *bs* ⟧ ⟹ *a* # *as* ∥ *b* # *bs*
⟨*proof*⟩

lemma *not-equal-is-parallel*:
assumes *neq*: *xs* ≠ *ys*
and *len*: *length xs* = *length ys*
shows *xs* ∥ *ys*
⟨*proof*⟩

47.7 Suffixes

primrec *suffixes* **where**
suffixes [] = [[]]
| *suffixes* (*x*#*xs*) = *suffixes xs* @ [*x* # *xs*]

lemma *in-set-suffixes* [*simp*]: *xs* ∈ *set* (*suffixes ys*) ⟷ *suffix xs ys*
⟨*proof*⟩

lemma *distinct-suffixes* [*intro*]: *distinct* (*suffixes xs*)
⟨*proof*⟩

lemma *length-suffixes* [*simp*]: *length* (*suffixes xs*) = *Suc* (*length xs*)
⟨*proof*⟩

lemma *suffixes-snoc* [*simp*]: *suffixes* (*xs* @ [*x*]) = [] # *map* (λ*ys*. *ys* @ [*x*]) (*suffixes xs*)
⟨*proof*⟩

lemma *suffixes-not-Nil* [*simp*]: *suffixes xs* ≠ []
⟨*proof*⟩

lemma *hd-suffixes* [*simp*]: *hd* (*suffixes xs*) = []
⟨*proof*⟩

lemma *last-suffixes* [*simp*]: *last* (*suffixes xs*) = *xs*
⟨*proof*⟩

lemma *suffixes-append*:
suffixes (*xs* @ *ys*) = *suffixes ys* @ *map* (λ*xs'*. *xs'* @ *ys*) (*tl* (*suffixes xs*))
⟨*proof*⟩

lemma *suffixes-eq-snoc*:
suffixes ys = *xs* @ [*x*] ⟷
(*ys* = [] ∧ *xs* = [] ∨ (∃ *z zs*. *ys* = *z*#*zs* ∧ *xs* = *suffixes zs*)) ∧ *x* = *ys*
⟨*proof*⟩

lemma *suffixes-tailrec* [*code*]:
suffixes xs = *rev* (*snd* (*foldl* (λ(*acc1*, *acc2*) *x*. (*x*#*acc1*, (*x*#*acc1*)#*acc2*)) ([],[[]]) (*rev xs*)))
⟨*proof*⟩

lemma *set-suffixes-eq*: *set* (*suffixes xs*) = {*ys*. *suffix ys xs*}
⟨*proof*⟩

lemma *card-set-suffixes* [*simp*]: *card* (*set* (*suffixes xs*)) = *Suc* (*length xs*)
⟨*proof*⟩

lemma *set-suffixes-append*:
set (*suffixes* (*xs* @ *ys*)) = *set* (*suffixes ys*) ∪ {*xs'* @ *ys* |*xs'*. *xs'* ∈ *set* (*suffixes xs*)}
⟨*proof*⟩

lemma *suffixes-conv-prefixes*: *suffixes xs* = *map rev* (*prefixes* (*rev xs*))
⟨*proof*⟩

lemma *prefixes-conv-suffixes*: *prefixes xs* = *map rev* (*suffixes* (*rev xs*))
⟨*proof*⟩

lemma *prefixes-rev*: *prefixes* (*rev xs*) = *map rev* (*suffixes xs*)
⟨*proof*⟩

lemma *suffixes-rev*: *suffixes* (*rev xs*) = *map rev* (*prefixes xs*)
⟨*proof*⟩

47.8 Homeomorphic embedding on lists

inductive *list-emb* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool$
  for $P :: ('a \Rightarrow 'a \Rightarrow bool)$
where
  list-emb-Nil [*intro*, *simp*]: *list-emb P* [] *ys*
| *list-emb-Cons* [*intro*] : *list-emb P xs ys* $\Longrightarrow$ *list-emb P xs* (*y*#*ys*)
| *list-emb-Cons2* [*intro*]: *P x y* $\Longrightarrow$ *list-emb P xs ys* $\Longrightarrow$ *list-emb P* (*x*#*xs*) (*y*#*ys*)

lemma *list-emb-mono*:
  assumes $\bigwedge x\ y.\ P\ x\ y \longrightarrow Q\ x\ y$
  shows *list-emb P xs ys* $\longrightarrow$ *list-emb Q xs ys*
⟨*proof*⟩

lemma *list-emb-Nil2* [*simp*]:
  assumes *list-emb P xs* [] **shows** *xs* = []
  ⟨*proof*⟩

lemma *list-emb-refl*:
  assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow P\ x\ x$
  shows *list-emb P xs xs*
  ⟨*proof*⟩

lemma *list-emb-Cons-Nil* [*simp*]: *list-emb P* (*x*#*xs*) [] = *False*
⟨*proof*⟩

lemma *list-emb-append2* [*intro*]: *list-emb P xs ys* $\Longrightarrow$ *list-emb P xs* (*zs* @ *ys*)
  ⟨*proof*⟩

lemma *list-emb-prefix* [*intro*]:
  assumes *list-emb P xs ys* **shows** *list-emb P xs* (*ys* @ *zs*)
  ⟨*proof*⟩

lemma *list-emb-ConsD*:
  assumes *list-emb P* (*x*#*xs*) *ys*
  shows $\exists$ *us v vs. ys* = *us* @ *v* # *vs* $\wedge$ *P x v* $\wedge$ *list-emb P xs vs*
⟨*proof*⟩

lemma *list-emb-appendD*:
  assumes *list-emb P* (*xs* @ *ys*) *zs*
  shows $\exists$ *us vs. zs* = *us* @ *vs* $\wedge$ *list-emb P xs us* $\wedge$ *list-emb P ys vs*
⟨*proof*⟩

lemma *list-emb-strict-suffix*:
  assumes *list-emb P xs ys* **and** *strict-suffix ys zs*
  shows *list-emb P xs zs*
  ⟨*proof*⟩

lemma *list-emb-suffix*:
  assumes *list-emb P xs ys* **and** *suffix ys zs*

shows *list-emb P xs zs*
⟨*proof*⟩

lemma *list-emb-length*: *list-emb P xs ys* ⟹ *length xs* ≤ *length ys*
⟨*proof*⟩

lemma *list-emb-trans*:
assumes ⋀*x y z*. ⟦*x* ∈ *set xs*; *y* ∈ *set ys*; *z* ∈ *set zs*; *P x y*; *P y z*⟧ ⟹ *P x z*
shows ⟦*list-emb P xs ys*; *list-emb P ys zs*⟧ ⟹ *list-emb P xs zs*
⟨*proof*⟩

lemma *list-emb-set*:
assumes *list-emb P xs ys* **and** *x* ∈ *set xs*
obtains *y* **where** *y* ∈ *set ys* **and** *P x y*
⟨*proof*⟩

lemma *list-emb-Cons-iff1* [*simp*]:
assumes *P x y*
shows *list-emb P* (*x*#*xs*) (*y*#*ys*) ⟷ *list-emb P xs ys*
⟨*proof*⟩

lemma *list-emb-Cons-iff2* [*simp*]:
assumes ¬*P x y*
shows *list-emb P* (*x*#*xs*) (*y*#*ys*) ⟷ *list-emb P* (*x*#*xs*) *ys*
⟨*proof*⟩

lemma *list-emb-code* [*code*]:
list-emb P [] *ys* ⟷ *True*
list-emb P (*x*#*xs*) [] ⟷ *False*
list-emb P (*x*#*xs*) (*y*#*ys*) ⟷ (*if P x y then list-emb P xs ys else list-emb P* (*x*#*xs*) *ys*)
⟨*proof*⟩

47.9 Subsequences (special case of homeomorphic embedding)

abbreviation *subseq* :: *'a list* ⇒ *'a list* ⇒ *bool*
where *subseq xs ys* ≡ *list-emb* (*op* =) *xs ys*

definition *strict-subseq* **where** *strict-subseq xs ys* ⟷ *xs* ≠ *ys* ∧ *subseq xs ys*

lemma *subseq-Cons2*: *subseq xs ys* ⟹ *subseq* (*x*#*xs*) (*x*#*ys*) ⟨*proof*⟩

lemma *subseq-same-length*:
assumes *subseq xs ys* **and** *length xs* = *length ys* **shows** *xs* = *ys*
⟨*proof*⟩

lemma *not-subseq-length* [*simp*]: *length ys* < *length xs* ⟹ ¬ *subseq xs ys*
⟨*proof*⟩

lemma *subseq-Cons'*: *subseq* (*x*#*xs*) *ys* ⟹ *subseq xs ys*
⟨*proof*⟩

lemma *subseq-Cons2'*:
assumes *subseq* (*x*#*xs*) (*x*#*ys*) **shows** *subseq xs ys*
⟨*proof*⟩

lemma *subseq-Cons2-neq*:
assumes *subseq* (*x*#*xs*) (*y*#*ys*)
shows *x* ≠ *y* ⟹ *subseq* (*x*#*xs*) *ys*
⟨*proof*⟩

lemma *subseq-Cons2-iff* [*simp*]:
subseq (*x*#*xs*) (*y*#*ys*) = (*if x* = *y then subseq xs ys else subseq* (*x*#*xs*) *ys*)
⟨*proof*⟩

lemma *subseq-append'*: *subseq* (*zs* @ *xs*) (*zs* @ *ys*) ⟷ *subseq xs ys*
⟨*proof*⟩

interpretation *subseq-order*: *order subseq strict-subseq*
⟨*proof*⟩

lemma *in-set-subseqs* [*simp*]: *xs* ∈ *set* (*subseqs ys*) ⟷ *subseq xs ys*
⟨*proof*⟩

lemma *set-subseqs-eq*: *set* (*subseqs ys*) = {*xs*. *subseq xs ys*}
⟨*proof*⟩

lemma *subseq-append-le-same-iff*: *subseq* (*xs* @ *ys*) *ys* ⟷ *xs* = []
⟨*proof*⟩

lemma *subseq-singleton-left*: *subseq* [*x*] *ys* ⟷ *x* ∈ *set ys*
⟨*proof*⟩

lemma *list-emb-append-mono*:
⟦ *list-emb P xs xs'*; *list-emb P ys ys'* ⟧ ⟹ *list-emb P* (*xs*@*ys*) (*xs'*@*ys'*)
⟨*proof*⟩

lemma *prefix-imp-subseq* [*intro*]: *prefix xs ys* ⟹ *subseq xs ys*
⟨*proof*⟩

lemma *suffix-imp-subseq* [*intro*]: *suffix xs ys* ⟹ *subseq xs ys*
⟨*proof*⟩

47.10 Appending elements

lemma *subseq-append* [*simp*]:
subseq (*xs* @ *zs*) (*ys* @ *zs*) ⟷ *subseq xs ys* (**is** *?l* = *?r*)

⟨*proof*⟩

lemma *subseq-append-iff*:
subseq xs (*ys* @ *zs*) ⟷ (∃ *xs1 xs2*. *xs* = *xs1* @ *xs2* ∧ *subseq xs1 ys* ∧ *subseq xs2 zs*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *subseq-appendE* [*case-names append*]:
assumes *subseq xs* (*ys* @ *zs*)
obtains *xs1 xs2* **where** *xs* = *xs1* @ *xs2 subseq xs1 ys subseq xs2 zs*
⟨*proof*⟩

lemma *subseq-drop-many*: *subseq xs ys* ⟹ *subseq xs* (*zs* @ *ys*)
⟨*proof*⟩

lemma *subseq-rev-drop-many*: *subseq xs ys* ⟹ *subseq xs* (*ys* @ *zs*)
⟨*proof*⟩

47.11 Relation to standard list operations

lemma *subseq-map*:
assumes *subseq xs ys* **shows** *subseq* (*map f xs*) (*map f ys*)
⟨*proof*⟩

lemma *subseq-filter-left* [*simp*]: *subseq* (*filter P xs*) *xs*
⟨*proof*⟩

lemma *subseq-filter* [*simp*]:
assumes *subseq xs ys* **shows** *subseq* (*filter P xs*) (*filter P ys*)
⟨*proof*⟩

lemma *subseq-conv-nths*:
subseq xs ys ⟷ (∃ *N*. *xs* = *nths ys N*) (**is** *?L* = *?R*)
⟨*proof*⟩

47.12 Contiguous sublists

definition *sublist* :: *'a list* ⇒ *'a list* ⇒ *bool* **where**
sublist xs ys = (∃ *ps ss*. *ys* = *ps* @ *xs* @ *ss*)

definition *strict-sublist* :: *'a list* ⇒ *'a list* ⇒ *bool* **where**
strict-sublist xs ys ⟷ *sublist xs ys* ∧ *xs* ≠ *ys*

interpretation *sublist-order*: *order sublist strict-sublist*
⟨*proof*⟩

lemma *sublist-Nil-left* [*simp*, *intro*]: *sublist* [] *ys*
⟨*proof*⟩

lemma *sublist-Cons-Nil* [*simp*]: ¬*sublist* (*x*#*xs*) []
⟨*proof*⟩

lemma *sublist-Nil-right* [*simp*]: *sublist xs* [] ⟷ *xs* = []
⟨*proof*⟩

lemma *sublist-appendI* [*simp*, *intro*]: *sublist xs* (*ps* @ *xs* @ *ss*)
⟨*proof*⟩

lemma *sublist-append-leftI* [*simp*, *intro*]: *sublist xs* (*ps* @ *xs*)
⟨*proof*⟩

lemma *sublist-append-rightI* [*simp*, *intro*]: *sublist xs* (*xs* @ *ss*)
⟨*proof*⟩

lemma *sublist-altdef*: *sublist xs ys* ⟷ (∃ *ys'*. *prefix ys' ys* ∧ *suffix xs ys'*)
⟨*proof*⟩

lemma *sublist-altdef'*: *sublist xs ys* ⟷ (∃ *ys'*. *suffix ys' ys* ∧ *prefix xs ys'*)
⟨*proof*⟩

lemma *sublist-Cons-right*: *sublist xs* (*y* # *ys*) ⟷ *prefix xs* (*y* # *ys*) ∨ *sublist xs ys*
⟨*proof*⟩

lemma *sublist-code* [*code*]:
sublist [] *ys* ⟷ *True*
sublist (*x* # *xs*) [] ⟷ *False*
sublist (*x* # *xs*) (*y* # *ys*) ⟷ *prefix* (*x* # *xs*) (*y* # *ys*) ∨ *sublist* (*x* # *xs*) *ys*
⟨*proof*⟩

lemma *sublist-append*:
sublist xs (*ys* @ *zs*) ⟷
sublist xs ys ∨ *sublist xs zs* ∨ (∃ *xs1 xs2*. *xs* = *xs1* @ *xs2* ∧ *suffix xs1 ys* ∧ *prefix xs2 zs*)
⟨*proof*⟩

primrec *sublists* :: *'a list* ⇒ *'a list list* **where**
sublists [] = [[]]
| *sublists* (*x* # *xs*) = *sublists xs* @ *map* (*op* # *x*) (*prefixes xs*)

lemma *in-set-sublists* [*simp*]: *xs* ∈ *set* (*sublists ys*) ⟷ *sublist xs ys*
⟨*proof*⟩

lemma *set-sublists-eq*: *set* (*sublists xs*) = {*ys*. *sublist ys xs*}
⟨*proof*⟩

lemma *length-sublists* [*simp*]: *length* (*sublists xs*) = *Suc* (*length xs* ∗ *Suc* (*length*

xs) *div 2*)
⟨*proof*⟩

lemma *sublist-length-le*: *sublist xs ys* ⟹ *length xs* ≤ *length ys*
⟨*proof*⟩

lemma *set-mono-sublist*: *sublist xs ys* ⟹ *set xs* ⊆ *set ys*
⟨*proof*⟩

lemma *prefix-imp-sublist* [*simp*, *intro*]: *prefix xs ys* ⟹ *sublist xs ys*
⟨*proof*⟩

lemma *suffix-imp-sublist* [*simp*, *intro*]: *suffix xs ys* ⟹ *sublist xs ys*
⟨*proof*⟩

lemma *sublist-take* [*simp*, *intro*]: *sublist* (*take n xs*) *xs*
⟨*proof*⟩

lemma *sublist-drop* [*simp*, *intro*]: *sublist* (*drop n xs*) *xs*
⟨*proof*⟩

lemma *sublist-tl* [*simp*, *intro*]: *sublist* (*tl xs*) *xs*
⟨*proof*⟩

lemma *sublist-butlast* [*simp*, *intro*]: *sublist* (*butlast xs*) *xs*
⟨*proof*⟩

lemma *sublist-rev* [*simp*]: *sublist* (*rev xs*) (*rev ys*) = *sublist xs ys*
⟨*proof*⟩

lemma *sublist-rev-left*: *sublist* (*rev xs*) *ys* = *sublist xs* (*rev ys*)
⟨*proof*⟩

lemma *sublist-rev-right*: *sublist xs* (*rev ys*) = *sublist* (*rev xs*) *ys*
⟨*proof*⟩

lemma *snoc-sublist-snoc*:
sublist (*xs* @ [*x*]) (*ys* @ [*y*]) ⟷
(*x* = *y* ∧ *suffix xs ys* ∨ *sublist* (*xs* @ [*x*]) *ys*)
⟨*proof*⟩

lemma *sublist-snoc*:
sublist xs (*ys* @ [*y*]) ⟷ *suffix xs* (*ys* @ [*y*]) ∨ *sublist xs ys*
⟨*proof*⟩

lemma *sublist-imp-subseq* [*intro*]: *sublist xs ys* ⟹ *subseq xs ys*
⟨*proof*⟩

47.13 Parametricity

context includes *lifting-syntax*
begin

private lemma *prefix-primrec*:
prefix = *rec-list* (λ*xs*. *True*) (λ*x xs xsa ys*.
case *ys of* [] ⇒ *False* | *y* # *ys* ⇒ *x* = *y* ∧ *xsa ys*)
⟨*proof*⟩ **lemma** *sublist-primrec*:
sublist = (λ*xs ys*. *rec-list* (λ*xs*. *xs* = []) (λ*y ys ysa xs*. *prefix xs* (*y* # *ys*) ∨ *ysa xs*) *ys xs*)
⟨*proof*⟩ **lemma** *list-emb-primrec*:
list-emb = (λ*uu uua uuaa*. *rec-list* (λ*P xs*. *List.null xs*) (λ*y ys ysa P xs*. *case xs of* [] ⇒ *True*
| *x* # *xs* ⇒ *if P x y then ysa P xs else ysa P* (*x* # *xs*)) *uuaa uu uua*)
⟨*proof*⟩

lemma *prefix-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> *op* =) *prefix prefix*
⟨*proof*⟩

lemma *suffix-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> *op* =) *suffix suffix*
⟨*proof*⟩

lemma *sublist-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> *op* =) *sublist sublist*
⟨*proof*⟩

lemma *parallel-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> *op* =) *parallel parallel*
⟨*proof*⟩

lemma *list-emb-transfer* [*transfer-rule*]:
((*A* ===> *A* ===> *op* =) ===> *list-all2 A* ===> *list-all2 A* ===> *op* =) *list-emb list-emb*
⟨*proof*⟩

lemma *strict-prefix-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> *op* =) *strict-prefix strict-prefix*
⟨*proof*⟩

lemma *strict-suffix-transfer* [*transfer-rule*]:

assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> *op* =) *strict-suffix strict-suffix*
⟨*proof*⟩

lemma *strict-subseq-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> *op* =) *strict-subseq strict-subseq*
⟨*proof*⟩

lemma *strict-sublist-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> *op* =) *strict-sublist strict-sublist*
⟨*proof*⟩

lemma *prefixes-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2* (*list-all2 A*)) *prefixes prefixes*
⟨*proof*⟩

lemma *suffixes-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2* (*list-all2 A*)) *suffixes suffixes*
⟨*proof*⟩

lemma *sublists-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2* (*list-all2 A*)) *sublists sublists*
⟨*proof*⟩

end

end

48 Linear Temporal Logic on Streams

theory *Linear-Temporal-Logic-on-Streams*
imports *Stream Sublist Extended-Nat Infinite-Set*
begin

49 Preliminaries

lemma *shift-prefix*:
assumes *xl* @− *xs* = *yl* @− *ys* **and** *length xl* ≤ *length yl*
shows *prefix xl yl*
⟨*proof*⟩

lemma *shift-prefix-cases*:
assumes *xl* @− *xs* = *yl* @− *ys*

shows *prefix xl yl* ∨ *prefix yl xl*
⟨*proof*⟩

50 Linear temporal logic

abbreviation (*input*) *IMPL* (**infix** *impl 60*)
where φ *impl* ψ ≡ λ *xs*. φ *xs* ⟶ ψ *xs*

abbreviation (*input*) *OR* (**infix** *or 60*)
where φ *or* ψ ≡ λ *xs*. φ *xs* ∨ ψ *xs*

abbreviation (*input*) *AND* (**infix** *aand 60*)
where φ *aand* ψ ≡ λ *xs*. φ *xs* ∧ ψ *xs*

abbreviation (*input*) *not* φ ≡ λ *xs*. ¬ φ *xs*

abbreviation (*input*) *true* ≡ λ *xs*. *True*

abbreviation (*input*) *false* ≡ λ *xs*. *False*

lemma *impl-not-or*: φ *impl* ψ = (*not* φ) *or* ψ
⟨*proof*⟩

lemma *not-or*: *not* (φ *or* ψ) = (*not* φ) *aand* (*not* ψ)
⟨*proof*⟩

lemma *not-aand*: *not* (φ *aand* ψ) = (*not* φ) *or* (*not* ψ)
⟨*proof*⟩

lemma *non-not*[*simp*]: *not* (*not* φ) = φ ⟨*proof*⟩

fun *holds* **where** *holds P xs* ⟷ *P* (*shd xs*)
fun *nxt* **where** *nxt* φ *xs* = φ (*stl xs*)

definition *HLD s* = *holds* (λ*x*. *x* ∈ *s*)

abbreviation *HLD-nxt* (**infixr** · *65*) **where**
s · *P* ≡ *HLD s aand nxt P*

context
notes [[*inductive-internals*]]
begin

inductive *ev* **for** φ **where**
base: φ *xs* ⟹ *ev* φ *xs*
|
step: *ev* φ (*stl xs*) ⟹ *ev* φ *xs*

coinductive *alw* **for** φ **where**
alw: $[\![\varphi\ xs;\ alw\ \varphi\ (stl\ xs)]\!] \Longrightarrow alw\ \varphi\ xs$

coinductive *UNTIL* (**infix** *until 60*) **for** $\varphi\ \psi$ **where**
base: $\psi\ xs \Longrightarrow (\varphi\ until\ \psi)\ xs$
|
step: $[\![\varphi\ xs;\ (\varphi\ until\ \psi)\ (stl\ xs)]\!] \Longrightarrow (\varphi\ until\ \psi)\ xs$

end

lemma *holds-mono*:
assumes *holds*: *holds P xs* **and** *0*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows *holds Q xs*
⟨*proof*⟩

lemma *holds-aand*:
(*holds P aand holds Q*) *steps* $\longleftrightarrow$ *holds* (λ *step*. *P step* $\wedge$ *Q step*) *steps* ⟨*proof*⟩

lemma *HLD-iff*: *HLD s* $\omega \longleftrightarrow$ *shd* $\omega \in s$
⟨*proof*⟩

lemma *HLD-Stream*[*simp*]: *HLD X* (*x* ## ω) $\longleftrightarrow x \in X$
⟨*proof*⟩

lemma *nxt-mono*:
assumes *nxt*: *nxt* φ *xs* **and** *0*: $\bigwedge xs.\ \varphi\ xs \Longrightarrow \psi\ xs$
shows *nxt* ψ *xs*
⟨*proof*⟩

declare *ev.intros*[*intro*]
declare *alw.cases*[*elim*]

lemma *ev-induct-strong*[*consumes 1*, *case-names base step*]:
ev $\varphi\ x \Longrightarrow (\bigwedge xs.\ \varphi\ xs \Longrightarrow P\ xs) \Longrightarrow (\bigwedge xs.\ ev\ \varphi\ (stl\ xs) \Longrightarrow \neg\ \varphi\ xs \Longrightarrow P\ (stl\ xs) \Longrightarrow P\ xs) \Longrightarrow P\ x$
⟨*proof*⟩

lemma *alw-coinduct*[*consumes 1*, *case-names alw stl*]:
X x $\Longrightarrow (\bigwedge x.\ X\ x \Longrightarrow \varphi\ x) \Longrightarrow (\bigwedge x.\ X\ x \Longrightarrow \neg\ alw\ \varphi\ (stl\ x) \Longrightarrow X\ (stl\ x)) \Longrightarrow alw\ \varphi\ x$
⟨*proof*⟩

lemma *ev-mono*:
assumes *ev*: *ev* φ *xs* **and** *0*: $\bigwedge xs.\ \varphi\ xs \Longrightarrow \psi\ xs$
shows *ev* ψ *xs*
⟨*proof*⟩

lemma *alw-mono*:

assumes *alw*: *alw* φ *xs* **and** *0*: $\bigwedge$ *xs*. φ *xs* $\Longrightarrow$ ψ *xs*
shows *alw* ψ *xs*
⟨*proof*⟩

lemma *until-monoL*:
assumes *until*: (*φ1 until ψ*) *xs* **and** *0*: $\bigwedge$ *xs*. *φ1 xs* $\Longrightarrow$ *φ2 xs*
shows (*φ2 until ψ*) *xs*
⟨*proof*⟩

lemma *until-monoR*:
assumes *until*: (*φ until ψ1*) *xs* **and** *0*: $\bigwedge$ *xs*. *ψ1 xs* $\Longrightarrow$ *ψ2 xs*
shows (*φ until ψ2*) *xs*
⟨*proof*⟩

lemma *until-mono*:
assumes *until*: (*φ1 until ψ1*) *xs* **and**
0: $\bigwedge$ *xs*. *φ1 xs* $\Longrightarrow$ *φ2 xs* $\bigwedge$ *xs*. *ψ1 xs* $\Longrightarrow$ *ψ2 xs*
shows (*φ2 until ψ2*) *xs*
⟨*proof*⟩

lemma *until-false*: *φ until false* = *alw φ*
⟨*proof*⟩

lemma *ev-nxt*: *ev φ* = (*φ or nxt* (*ev φ*))
⟨*proof*⟩

lemma *alw-nxt*: *alw φ* = (*φ aand nxt* (*alw φ*))
⟨*proof*⟩

lemma *ev-ev*[*simp*]: *ev* (*ev φ*) = *ev φ*
⟨*proof*⟩

lemma *alw-alw*[*simp*]: *alw* (*alw φ*) = *alw φ*
⟨*proof*⟩

lemma *ev-shift*:
assumes *ev φ xs*
shows *ev φ* (*xl* @− *xs*)
⟨*proof*⟩

lemma *ev-imp-shift*:
assumes *ev φ xs* **shows** ∃ *xl xs2*. *xs* = *xl* @− *xs2* ∧ *φ xs2*
⟨*proof*⟩

lemma *alw-ev-shift*: *alw φ xs1* $\Longrightarrow$ *ev* (*alw φ*) (*xl* @− *xs1*)
⟨*proof*⟩

lemma *alw-shift*:
assumes *alw φ* (*xl* @− *xs*)

shows *alw φ xs*
⟨*proof*⟩

lemma *ev-ex-nxt*:
assumes *ev φ xs*
shows ∃ *n*. (*nxt* ^^ *n*) *φ xs*
⟨*proof*⟩

lemma *alw-sdrop*:
assumes *alw φ xs* **shows** *alw φ* (*sdrop n xs*)
⟨*proof*⟩

lemma *nxt-sdrop*: (*nxt* ^^ *n*) *φ xs* ⟷ *φ* (*sdrop n xs*)
⟨*proof*⟩

definition *wait φ xs* ≡ *LEAST n*. (*nxt* ^^ *n*) *φ xs*

lemma *nxt-wait*:
assumes *ev φ xs* **shows** (*nxt* ^^ (*wait φ xs*)) *φ xs*
⟨*proof*⟩

lemma *nxt-wait-least*:
assumes *ev*: *ev φ xs* **and** *nxt*: (*nxt* ^^ *n*) *φ xs* **shows** *wait φ xs* ≤ *n*
⟨*proof*⟩

lemma *sdrop-wait*:
assumes *ev φ xs* **shows** *φ* (*sdrop* (*wait φ xs*) *xs*)
⟨*proof*⟩

lemma *sdrop-wait-least*:
assumes *ev*: *ev φ xs* **and** *nxt*: *φ* (*sdrop n xs*) **shows** *wait φ xs* ≤ *n*
⟨*proof*⟩

lemma *nxt-ev*: (*nxt* ^^ *n*) *φ xs* ⟹ *ev φ xs*
⟨*proof*⟩

lemma *not-ev*: *not* (*ev φ*) = *alw* (*not φ*)
⟨*proof*⟩

lemma *not-alw*: *not* (*alw φ*) = *ev* (*not φ*)
⟨*proof*⟩

lemma *not-ev-not*[*simp*]: *not* (*ev* (*not φ*)) = *alw φ*
⟨*proof*⟩

lemma *not-alw-not*[*simp*]: *not* (*alw* (*not φ*)) = *ev φ*
⟨*proof*⟩

lemma *alw-ev-sdrop*:

assumes *alw* (*ev* φ) (*sdrop m xs*)
shows *alw* (*ev* φ) *xs*
⟨*proof*⟩

lemma *ev-alw-imp-alw-ev*:
assumes *ev* (*alw* φ) *xs* **shows** *alw* (*ev* φ) *xs*
⟨*proof*⟩

lemma *alw-aand*: *alw* (φ *aand* ψ) = *alw* φ *aand alw* ψ
⟨*proof*⟩

lemma *ev-or*: *ev* (φ *or* ψ) = *ev* φ *or ev* ψ
⟨*proof*⟩

lemma *ev-alw-aand*:
assumes φ: *ev* (*alw* φ) *xs* **and** ψ: *ev* (*alw* ψ) *xs*
shows *ev* (*alw* (φ *aand* ψ)) *xs*
⟨*proof*⟩

lemma *ev-alw-alw-impl*:
assumes *ev* (*alw* φ) *xs* **and** *alw* (*alw* φ *impl ev* ψ) *xs*
shows *ev* ψ *xs*
⟨*proof*⟩

lemma *ev-alw-stl*[*simp*]: *ev* (*alw* φ) (*stl x*) ⟷ *ev* (*alw* φ) *x*
⟨*proof*⟩

lemma *alw-alw-impl-ev*:
alw (*alw* φ *impl ev* ψ) = (*ev* (*alw* φ) *impl alw* (*ev* ψ)) (**is** *?A* = *?B*)
⟨*proof*⟩

lemma *ev-alw-impl*:
assumes *ev* φ *xs* **and** *alw* (φ *impl* ψ) *xs* **shows** *ev* ψ *xs*
⟨*proof*⟩

lemma *ev-alw-impl-ev*:
assumes *ev* φ *xs* **and** *alw* (φ *impl ev* ψ) *xs* **shows** *ev* ψ *xs*
⟨*proof*⟩

lemma *alw-mp*:
assumes *alw* φ *xs* **and** *alw* (φ *impl* ψ) *xs*
shows *alw* ψ *xs*
⟨*proof*⟩

lemma *all-imp-alw*:
assumes ⋀ *xs*. φ *xs* **shows** *alw* φ *xs*
⟨*proof*⟩

lemma *alw-impl-ev-alw*:

assumes *alw* (*φ impl ev ψ*) *xs*
shows *alw* (*ev φ impl ev ψ*) *xs*
⟨*proof*⟩

lemma *ev-holds-sset*:
ev (*holds P*) *xs* ⟷ (∃ *x* ∈ *sset xs*. *P x*) (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *alw-invar*:
assumes *φ xs* **and** *alw* (*φ impl nxt φ*) *xs*
shows *alw φ xs*
⟨*proof*⟩

lemma *variance*:
assumes *1*: *φ xs* **and** *2*: *alw* (*φ impl* (*ψ or nxt φ*)) *xs*
shows (*alw φ or ev ψ*) *xs*
⟨*proof*⟩

lemma *ev-alw-imp-nxt*:
assumes *e*: *ev φ xs* **and** *a*: *alw* (*φ impl* (*nxt φ*)) *xs*
shows *ev* (*alw φ*) *xs*
⟨*proof*⟩

inductive *ev-at* :: ('*a stream* ⇒ *bool*) ⇒ *nat* ⇒ '*a stream* ⇒ *bool* **for** *P* :: '*a stream* ⇒ *bool* **where**
  base: *P ω* ⟹ *ev-at P 0 ω*
| *step*:¬ *P ω* ⟹ *ev-at P n* (*stl ω*) ⟹ *ev-at P* (*Suc n*) *ω*

inductive-simps *ev-at-0*[*simp*]: *ev-at P 0 ω*
inductive-simps *ev-at-Suc*[*simp*]: *ev-at P* (*Suc n*) *ω*

lemma *ev-at-imp-snth*: *ev-at P n ω* ⟹ *P* (*sdrop n ω*)
  ⟨*proof*⟩

lemma *ev-at-HLD-imp-snth*: *ev-at* (*HLD X*) *n ω* ⟹ *ω* !! *n* ∈ *X*
  ⟨*proof*⟩

lemma *ev-at-HLD-single-imp-snth*: *ev-at* (*HLD* {*x*}) *n ω* ⟹ *ω* !! *n* = *x*
  ⟨*proof*⟩

lemma *ev-at-unique*: *ev-at P n ω* ⟹ *ev-at P m ω* ⟹ *n* = *m*
⟨*proof*⟩

lemma *ev-iff-ev-at*: *ev P ω* ⟷ (∃ *n*. *ev-at P n ω*)
⟨*proof*⟩

lemma *ev-at-shift*: *ev-at* (*HLD X*) *i* (*stake* (*Suc i*) *ω* @− *ω*′ :: '*s stream*) ⟷

ev-at (*HLD X*) *i* ω
⟨*proof*⟩

lemma *ev-iff-ev-at-unqiue*: *ev P* $\omega \longleftrightarrow (\exists!n.$ *ev-at P n* $\omega)$
⟨*proof*⟩

lemma *alw-HLD-iff-streams*: *alw* (*HLD X*) $\omega \longleftrightarrow \omega \in$ *streams X*
⟨*proof*⟩

lemma *not-HLD*: *not* (*HLD X*) = *HLD* (− *X*)
⟨*proof*⟩

lemma *not-alw-iff*: ¬ (*alw P* ω) $\longleftrightarrow$ *ev* (*not P*) ω
⟨*proof*⟩

lemma *not-ev-iff*: ¬ (*ev P* ω) $\longleftrightarrow$ *alw* (*not P*) ω
⟨*proof*⟩

lemma *ev-Stream*: *ev P* (*x* ## *s*) $\longleftrightarrow$ *P* (*x* ## *s*) ∨ *ev P s*
⟨*proof*⟩

lemma *alw-ev-imp-ev-alw*:
assumes *alw* (*ev P*) ω **shows** *ev* (*P aand alw* (*ev P*)) ω
⟨*proof*⟩

lemma *ev-False*: *ev* ($\lambda x.$ *False*) $\omega \longleftrightarrow$ *False*
⟨*proof*⟩

lemma *alw-False*: *alw* ($\lambda x.$ *False*) $\omega \longleftrightarrow$ *False*
⟨*proof*⟩

lemma *ev-iff-sdrop*: *ev P* $\omega \longleftrightarrow (\exists\, m.$ *P* (*sdrop m* ω))
⟨*proof*⟩

lemma *alw-iff-sdrop*: *alw P* $\omega \longleftrightarrow (\forall\, m.$ *P* (*sdrop m* ω))
⟨*proof*⟩

lemma *infinite-iff-alw-ev*: *infinite* {*m.* *P* (*sdrop m* ω)} $\longleftrightarrow$ *alw* (*ev P*) ω
⟨*proof*⟩

lemma *alw-inv*:
assumes *stl*: $\bigwedge s.$ *f* (*stl s*) = *stl* (*f s*)
shows *alw P* (*f s*) $\longleftrightarrow$ *alw* ($\lambda x.$ *P* (*f x*)) *s*
⟨*proof*⟩

lemma *ev-inv*:
assumes *stl*: $\bigwedge s.$ *f* (*stl s*) = *stl* (*f s*)
shows *ev P* (*f s*) $\longleftrightarrow$ *ev* ($\lambda x.$ *P* (*f x*)) *s*
⟨*proof*⟩

lemma *alw-smap*: *alw P* (*smap f s*) ⟷ *alw* (λ*x*. *P* (*smap f x*)) *s*
⟨*proof*⟩

lemma *ev-smap*: *ev P* (*smap f s*) ⟷ *ev* (λ*x*. *P* (*smap f x*)) *s*
⟨*proof*⟩

lemma *alw-cong*:
assumes *P*: *alw P ω* **and** *eq*: ⋀*ω*. *P ω* ⟹ *Q1 ω* ⟷ *Q2 ω*
shows *alw Q1 ω* ⟷ *alw Q2 ω*
⟨*proof*⟩

lemma *ev-cong*:
assumes *P*: *alw P ω* **and** *eq*: ⋀*ω*. *P ω* ⟹ *Q1 ω* ⟷ *Q2 ω*
shows *ev Q1 ω* ⟷ *ev Q2 ω*
⟨*proof*⟩

lemma *alwD*: *alw P x* ⟹ *P x*
⟨*proof*⟩

lemma *alw-alwD*: *alw P ω* ⟹ *alw* (*alw P*) *ω*
⟨*proof*⟩

lemma *alw-ev-stl*: *alw* (*ev P*) (*stl ω*) ⟷ *alw* (*ev P*) *ω*
⟨*proof*⟩

lemma *holds-Stream*: *holds P* (*x ## s*) ⟷ *P x*
⟨*proof*⟩

lemma *holds-eq1*[*simp*]: *holds* (*op* = *x*) = *HLD* {*x*}
⟨*proof*⟩

lemma *holds-eq2*[*simp*]: *holds* (λ*y*. *y* = *x*) = *HLD* {*x*}
⟨*proof*⟩

lemma *not-holds-eq*[*simp*]: *holds* (− *op* = *x*) = *not* (*HLD* {*x*})
⟨*proof*⟩

Strong until

context
notes [[*inductive-internals*]]
begin

inductive *suntil* (**infix** *suntil 60*) **for** *φ ψ* **where**
base: *ψ ω* ⟹ (*φ suntil ψ*) *ω*
| *step*: *φ ω* ⟹ (*φ suntil ψ*) (*stl ω*) ⟹ (*φ suntil ψ*) *ω*

inductive-simps *suntil-Stream*: (*φ suntil ψ*) (*x ## s*)

end

lemma *suntil-induct-strong*[*consumes 1*, *case-names base step*]:
(*φ suntil ψ*) *x* ⟹
(⋀*ω*. *ψ ω* ⟹ *P ω*) ⟹
(⋀*ω*. *φ ω* ⟹ ¬ *ψ ω* ⟹ (*φ suntil ψ*) (*stl ω*) ⟹ *P* (*stl ω*) ⟹ *P ω*) ⟹ *P x*
⟨*proof*⟩

lemma *ev-suntil*: (*φ suntil ψ*) *ω* ⟹ *ev ψ ω*
⟨*proof*⟩

lemma *suntil-inv*:
assumes *stl*: ⋀*s*. *f* (*stl s*) = *stl* (*f s*)
shows (*P suntil Q*) (*f s*) ⟷ ((λ*x*. *P* (*f x*)) *suntil* (λ*x*. *Q* (*f x*))) *s*
⟨*proof*⟩

lemma *suntil-smap*: (*P suntil Q*) (*smap f s*) ⟷ ((λ*x*. *P* (*smap f x*)) *suntil* (λ*x*. *Q* (*smap f x*))) *s*
⟨*proof*⟩

lemma *hld-smap*: *HLD x* (*smap f s*) = *holds* (λ*y*. *f y* ∈ *x*) *s*
⟨*proof*⟩

lemma *suntil-mono*:
assumes *eq*: ⋀*ω*. *P ω* ⟹ *Q1 ω* ⟹ *Q2 ω* ⋀*ω*. *P ω* ⟹ *R1 ω* ⟹ *R2 ω*
assumes ∗: (*Q1 suntil R1*) *ω alw P ω* **shows** (*Q2 suntil R2*) *ω*
⟨*proof*⟩

lemma *suntil-cong*:
alw P ω ⟹ (⋀*ω*. *P ω* ⟹ *Q1 ω* ⟷ *Q2 ω*) ⟹ (⋀*ω*. *P ω* ⟹ *R1 ω* ⟷ *R2 ω*) ⟹
(*Q1 suntil R1*) *ω* ⟷ (*Q2 suntil R2*) *ω*
⟨*proof*⟩

lemma *ev-suntil-iff*: *ev* (*P suntil Q*) *ω* ⟷ *ev Q ω*
⟨*proof*⟩

lemma *true-suntil*: ((λ-. *True*) *suntil P*) = *ev P*
⟨*proof*⟩

lemma *suntil-lfp*: (*φ suntil ψ*) = *lfp* (λ*P s*. *ψ s* ∨ (*φ s* ∧ *P* (*stl s*)))
⟨*proof*⟩

lemma *sfilter-P*[*simp*]: *P* (*shd s*) ⟹ *sfilter P s* = *shd s* ## *sfilter P* (*stl s*)
⟨*proof*⟩

lemma *sfilter-not-P*[*simp*]: ¬ *P* (*shd s*) ⟹ *sfilter P s* = *sfilter P* (*stl s*)
⟨*proof*⟩

lemma *sfilter-eq*:
assumes *ev* (*holds P*) *s*
shows *sfilter P s* = *x* ## *s'* ⟷
P x ∧ (*not* (*holds P*) *suntil* (*HLD* {*x*} *aand nxt* (λ*s*. *sfilter P s* = *s'*))) *s*
⟨*proof*⟩

lemma *sfilter-streams*:
alw (*ev* (*holds P*)) ω ⟹ ω ∈ *streams A* ⟹ *sfilter P* ω ∈ *streams* {*x*∈*A*. *P x*}
⟨*proof*⟩

lemma *alw-sfilter*:
assumes ∗: *alw* (*ev* (*holds P*)) *s*
shows *alw Q* (*sfilter P s*) ⟷ *alw* (λ*x*. *Q* (*sfilter P x*)) *s*
⟨*proof*⟩

lemma *ev-sfilter*:
assumes ∗: *alw* (*ev* (*holds P*)) *s*
shows *ev Q* (*sfilter P s*) ⟷ *ev* (λ*x*. *Q* (*sfilter P x*)) *s*
⟨*proof*⟩

lemma *holds-sfilter*:
assumes *ev* (*holds Q*) *s* **shows** *holds P* (*sfilter Q s*) ⟷ (*not* (*holds Q*) *suntil* (*holds* (*Q aand P*))) *s*
⟨*proof*⟩

lemma *suntil-aand-nxt*:
(φ *suntil* (φ *aand nxt* ψ)) ω ⟷ (φ *aand nxt* (φ *suntil* ψ)) ω
⟨*proof*⟩

lemma *alw-sconst*: *alw P* (*sconst x*) ⟷ *P* (*sconst x*)
⟨*proof*⟩

lemma *ev-sconst*: *ev P* (*sconst x*) ⟷ *P* (*sconst x*)
⟨*proof*⟩

lemma *suntil-sconst*: (φ *suntil* ψ) (*sconst x*) ⟷ ψ (*sconst x*)
⟨*proof*⟩

lemma *hld-smap'*: *HLD x* (*smap f s*) = *HLD* (*f* −‘ *x*) *s*
⟨*proof*⟩

lemma *pigeonhole-stream*:
assumes *alw* (*HLD s*) ω
assumes *finite s*
shows ∃ *x*∈*s*. *alw* (*ev* (*HLD* {*x*})) ω
⟨*proof*⟩

lemma *ev-eq-suntil*: *ev P* ω ⟷ (*not P suntil P*) ω
⟨*proof*⟩

end

51 Lists as vectors

theory *ListVector*
imports *HOL.List Main*
begin

A vector-space like structure of lists and arithmetic operations on them. Is only a vector space if restricted to lists of the same length.

Multiplication with a scalar:

abbreviation *scale* :: ('*a*::*times*) ⇒ '*a list* ⇒ '*a list* (**infix** $*_s$ *70*)
where $x *_s xs \equiv map\ (op * x)\ xs$

lemma *scale1*[*simp*]: (*1*::'*a*::*monoid-mult*) $*_s$ *xs* = *xs*
⟨*proof*⟩

51.1 + and −

fun *zipwith0* :: ('*a*::*zero* ⇒ '*b*::*zero* ⇒ '*c*) ⇒ '*a list* ⇒ '*b list* ⇒ '*c list*
where
zipwith0 f [] [] = [] |
zipwith0 f (*x*#*xs*) (*y*#*ys*) = *f x y* # *zipwith0 f xs ys* |
zipwith0 f (*x*#*xs*) [] = *f x 0* # *zipwith0 f xs* [] |
zipwith0 f [] (*y*#*ys*) = *f 0 y* # *zipwith0 f* [] *ys*

instantiation *list* :: ({*zero*, *plus*}) *plus*
begin

definition
list-add-def: *op* + = *zipwith0* (*op* +)

instance ⟨*proof*⟩

end

instantiation *list* :: ({*zero*, *uminus*}) *uminus*
begin

definition
list-uminus-def: *uminus* = *map uminus*

instance ⟨*proof*⟩

end

instantiation *list* :: ({*zero*,*minus*}) *minus*

begin

definition
list-diff-def: $op - = zipwith0\ (op -)$

instance ⟨*proof*⟩

end

lemma *zipwith0-Nil*[*simp*]: $zipwith0\ f\ []\ ys = map\ (f\ 0)\ ys$
⟨*proof*⟩

lemma *list-add-Nil*[*simp*]: $[] + xs = (xs::'a::monoid\text{-}add\ list)$
⟨*proof*⟩

lemma *list-add-Nil2*[*simp*]: $xs + [] = (xs::'a::monoid\text{-}add\ list)$
⟨*proof*⟩

lemma *list-add-Cons*[*simp*]: $(x\#xs) + (y\#ys) = (x+y)\#(xs+ys)$
⟨*proof*⟩

lemma *list-diff-Nil*[*simp*]: $[] - xs = -(xs::'a::group\text{-}add\ list)$
⟨*proof*⟩

lemma *list-diff-Nil2*[*simp*]: $xs - [] = (xs::'a::group\text{-}add\ list)$
⟨*proof*⟩

lemma *list-diff-Cons-Cons*[*simp*]: $(x\#xs) - (y\#ys) = (x-y)\#(xs-ys)$
⟨*proof*⟩

lemma *list-uminus-Cons*[*simp*]: $-(x\#xs) = (-x)\#(-xs)$
⟨*proof*⟩

lemma *self-list-diff*:
$xs - xs = replicate\ (length(xs::'a::group\text{-}add\ list))\ 0$
⟨*proof*⟩

lemma *list-add-assoc*: **fixes** $xs :: 'a::monoid\text{-}add\ list$
shows $(xs+ys)+zs = xs+(ys+zs)$
⟨*proof*⟩

51.2 Inner product

definition $iprod :: 'a::ring\ list \Rightarrow 'a\ list \Rightarrow 'a\ (\langle\text{-},\text{-}\rangle)$ **where**
$\langle xs,ys\rangle = (\sum (x,y) \leftarrow zip\ xs\ ys.\ x*y)$

lemma *iprod-Nil*[*simp*]: $\langle[],ys\rangle = 0$
⟨*proof*⟩

lemma *iprod-Nil2*[*simp*]: $\langle xs,[]\rangle = 0$
⟨*proof*⟩

lemma *iprod-Cons*[*simp*]: $\langle x\#xs,y\#ys\rangle = x*y + \langle xs,ys\rangle$
⟨*proof*⟩

lemma *iprod0-if-coeffs0*: $\forall\, c{\in}set\ cs.\ c = 0 \Longrightarrow \langle cs,xs\rangle = 0$
⟨*proof*⟩

lemma *iprod-uminus*[*simp*]: $\langle -xs,ys\rangle = -\langle xs,ys\rangle$
⟨*proof*⟩

lemma *iprod-left-add-distrib*: $\langle xs + ys,zs\rangle = \langle xs,zs\rangle + \langle ys,zs\rangle$
⟨*proof*⟩

lemma *iprod-left-diff-distrib*: $\langle xs - ys,\ zs\rangle = \langle xs,zs\rangle - \langle ys,zs\rangle$
⟨*proof*⟩

lemma *iprod-assoc*: $\langle x *_s xs,\ ys\rangle = x * \langle xs,ys\rangle$
⟨*proof*⟩

end

52 Definitions of Least Upper Bounds and Greatest Lower Bounds

theory *Lub-Glb*
imports *Complex-Main*
begin

Thanks to suggestions by James Margetson

definition *setle* :: *'a set* ⇒ *'a*::*ord* ⇒ *bool* (**infixl** *<= 70)
where $S *<= x = (ALL\ y{:}\ S.\ y \le x)$

definition *setge* :: *'a*::*ord* ⇒ *'a set* ⇒ *bool* (**infixl** <=* 70)
where $x <=* S = (ALL\ y{:}\ S.\ x \le y)$

52.1 Rules for the Relations *<= and <=*

lemma *setleI*: $ALL\ y{:}\ S.\ y \le x \Longrightarrow S *<= x$
⟨*proof*⟩

lemma *setleD*: $S *<= x \Longrightarrow y{:}\ S \Longrightarrow y \le x$
⟨*proof*⟩

lemma *setgeI*: $ALL\ y{:}\ S.\ x \le y \Longrightarrow x <=* S$
⟨*proof*⟩

lemma *setgeD*: $x <=* S \Longrightarrow y$: $S \Longrightarrow x \leq y$
⟨*proof*⟩

definition *leastP* :: $('a \Rightarrow bool) \Rightarrow 'a::ord \Rightarrow bool$
where $leastP\ P\ x = (P\ x \wedge x <=* Collect\ P)$

definition *isUb* :: $'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$
where $isUb\ R\ S\ x = (S\ *<=\ x \wedge x: R)$

definition *isLub* :: $'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$
where $isLub\ R\ S\ x = leastP\ (isUb\ R\ S)\ x$

definition *ubs* :: $'a\ set \Rightarrow 'a::ord\ set \Rightarrow 'a\ set$
where $ubs\ R\ S = Collect\ (isUb\ R\ S)$

52.2 Rules about the Operators *leastP*, *ub* and *lub*

lemma *leastPD1*: $leastP\ P\ x \Longrightarrow P\ x$
⟨*proof*⟩

lemma *leastPD2*: $leastP\ P\ x \Longrightarrow x <=* Collect\ P$
⟨*proof*⟩

lemma *leastPD3*: $leastP\ P\ x \Longrightarrow y$: $Collect\ P \Longrightarrow x \leq y$
⟨*proof*⟩

lemma *isLubD1*: $isLub\ R\ S\ x \Longrightarrow S\ *<=\ x$
⟨*proof*⟩

lemma *isLubD1a*: $isLub\ R\ S\ x \Longrightarrow x$: R
⟨*proof*⟩

lemma *isLub-isUb*: $isLub\ R\ S\ x \Longrightarrow isUb\ R\ S\ x$
⟨*proof*⟩

lemma *isLubD2*: $isLub\ R\ S\ x \Longrightarrow y : S \Longrightarrow y \leq x$
⟨*proof*⟩

lemma *isLubD3*: $isLub\ R\ S\ x \Longrightarrow leastP\ (isUb\ R\ S)\ x$
⟨*proof*⟩

lemma *isLubI1*: $leastP(isUb\ R\ S)\ x \Longrightarrow isLub\ R\ S\ x$
⟨*proof*⟩

lemma *isLubI2*: $isUb\ R\ S\ x \Longrightarrow x <=* Collect\ (isUb\ R\ S) \Longrightarrow isLub\ R\ S\ x$
⟨*proof*⟩

lemma *isUbD*: $isUb\ R\ S\ x \Longrightarrow y : S \Longrightarrow y \leq x$

⟨*proof*⟩

lemma *isUbD2*: *isUb R S x* ⟹ *S* ∗<= *x*
⟨*proof*⟩

lemma *isUbD2a*: *isUb R S x* ⟹ *x*: *R*
⟨*proof*⟩

lemma *isUbI*: *S* ∗<= *x* ⟹ *x*: *R* ⟹ *isUb R S x*
⟨*proof*⟩

lemma *isLub-le-isUb*: *isLub R S x* ⟹ *isUb R S y* ⟹ *x* ≤ *y*
⟨*proof*⟩

lemma *isLub-ubs*: *isLub R S x* ⟹ *x* <=∗ *ubs R S*
⟨*proof*⟩

lemma *isLub-unique*: [| *isLub R S x*; *isLub R S y* |] ==> *x* = (*y*::′*a*::*linorder*)
⟨*proof*⟩

lemma *isUb-UNIV-I*: (⋀*y*. *y* ∈ *S* ⟹ *y* ≤ *u*) ⟹ *isUb UNIV S u*
⟨*proof*⟩

definition *greatestP* :: (′*a* ⇒ *bool*) ⇒ ′*a*::*ord* ⇒ *bool*
where *greatestP P x* = (*P x* ∧ *Collect P* ∗<= *x*)

definition *isLb* :: ′*a set* ⇒ ′*a set* ⇒ ′*a*::*ord* ⇒ *bool*
where *isLb R S x* = (*x* <=∗ *S* ∧ *x*: *R*)

definition *isGlb* :: ′*a set* ⇒ ′*a set* ⇒ ′*a*::*ord* ⇒ *bool*
where *isGlb R S x* = *greatestP* (*isLb R S*) *x*

definition *lbs* :: ′*a set* ⇒ ′*a*::*ord set* ⇒ ′*a set*
where *lbs R S* = *Collect* (*isLb R S*)

52.3 Rules about the Operators *greatestP*, *isLb* and *isGlb*

lemma *greatestPD1*: *greatestP P x* ⟹ *P x*
⟨*proof*⟩

lemma *greatestPD2*: *greatestP P x* ⟹ *Collect P* ∗<= *x*
⟨*proof*⟩

lemma *greatestPD3*: *greatestP P x* ⟹ *y*: *Collect P* ⟹ *x* ≥ *y*
⟨*proof*⟩

lemma *isGlbD1*: *isGlb R S x* ⟹ *x* <=∗ *S*
⟨*proof*⟩

lemma *isGlbD1a*: $isGlb\ R\ S\ x \Longrightarrow x{:}\ R$
⟨*proof*⟩

lemma *isGlb-isLb*: $isGlb\ R\ S\ x \Longrightarrow isLb\ R\ S\ x$
⟨*proof*⟩

lemma *isGlbD2*: $isGlb\ R\ S\ x \Longrightarrow y : S \Longrightarrow y \geq x$
⟨*proof*⟩

lemma *isGlbD3*: $isGlb\ R\ S\ x \Longrightarrow greatestP\ (isLb\ R\ S)\ x$
⟨*proof*⟩

lemma *isGlbI1*: $greatestP\ (isLb\ R\ S)\ x \Longrightarrow isGlb\ R\ S\ x$
⟨*proof*⟩

lemma *isGlbI2*: $isLb\ R\ S\ x \Longrightarrow Collect\ (isLb\ R\ S)\ *<=\ x \Longrightarrow isGlb\ R\ S\ x$
⟨*proof*⟩

lemma *isLbD*: $isLb\ R\ S\ x \Longrightarrow y : S \Longrightarrow y \geq x$
⟨*proof*⟩

lemma *isLbD2*: $isLb\ R\ S\ x \Longrightarrow x <=*\ S$
⟨*proof*⟩

lemma *isLbD2a*: $isLb\ R\ S\ x \Longrightarrow x{:}\ R$
⟨*proof*⟩

lemma *isLbI*: $x <=*\ S \Longrightarrow x{:}\ R \Longrightarrow isLb\ R\ S\ x$
⟨*proof*⟩

lemma *isGlb-le-isLb*: $isGlb\ R\ S\ x \Longrightarrow isLb\ R\ S\ y \Longrightarrow x \geq y$
⟨*proof*⟩

lemma *isGlb-ubs*: $isGlb\ R\ S\ x \Longrightarrow lbs\ R\ S\ *<=\ x$
⟨*proof*⟩

lemma *isGlb-unique*: $[|\ isGlb\ R\ S\ x;\ isGlb\ R\ S\ y\ |] ==> x = (y{::}'a{::}linorder)$
⟨*proof*⟩

lemma *bdd-above-setle*: $bdd\text{-}above\ A \longleftrightarrow (\exists\, a.\ A\ *<=\ a)$
⟨*proof*⟩

lemma *bdd-below-setge*: $bdd\text{-}below\ A \longleftrightarrow (\exists\, a.\ a <=*\ A)$
⟨*proof*⟩

lemma *isLub-cSup*:
$(S{::}'a :: conditionally\text{-}complete\text{-}lattice\ set) \neq \{\} \Longrightarrow (\exists\, b.\ S\ *<=\ b) \Longrightarrow isLub\ UNIV\ S\ (Sup\ S)$

⟨*proof*⟩

lemma *isGlb-cInf*:
(*S*::'*a* :: *conditionally-complete-lattice set*) ≠ {} ⟹ (∃ *b*. *b* <=∗ *S*) ⟹ *isGlb UNIV S* (*Inf S*)
⟨*proof*⟩

lemma *cSup-le*: (*S*::'*a*::*conditionally-complete-lattice set*) ≠ {} ⟹ *S* ∗<= *b* ⟹ *Sup S* ≤ *b*
⟨*proof*⟩

lemma *cInf-ge*: (*S*::'*a* :: *conditionally-complete-lattice set*) ≠ {} ⟹ *b* <=∗ *S* ⟹ *Inf S* ≥ *b*
⟨*proof*⟩

lemma *cSup-bounds*:
fixes *S* :: '*a* :: *conditionally-complete-lattice set*
shows *S* ≠ {} ⟹ *a* <=∗ *S* ⟹ *S* ∗<= *b* ⟹ *a* ≤ *Sup S* ∧ *Sup S* ≤ *b*
⟨*proof*⟩

lemma *cSup-unique*: (*S*::'*a* :: {*conditionally-complete-linorder*, *no-bot*} *set*) ∗<= *b* ⟹ (∀ *b*'<*b*. ∃ *x*∈*S*. *b*' < *x*) ⟹ *Sup S* = *b*
⟨*proof*⟩

lemma *cInf-unique*: *b* <=∗ (*S*::'*a* :: {*conditionally-complete-linorder*, *no-top*} *set*) ⟹ (∀ *b*'>*b*. ∃ *x*∈*S*. *b*' > *x*) ⟹ *Inf S* = *b*
⟨*proof*⟩

Use completeness of reals (supremum property) to show that any bounded sequence has a least upper bound

lemma *reals-complete*: ∃ *X*. *X* ∈ *S* ⟹ ∃ *Y*. *isUb* (*UNIV*::*real set*) *S Y* ⟹ ∃ *t*. *isLub* (*UNIV* :: *real set*) *S t*
⟨*proof*⟩

lemma *Bseq-isUb*: ⋀*X* :: *nat* ⇒ *real*. *Bseq X* ⟹ ∃ *U*. *isUb* (*UNIV*::*real set*) {*x*. ∃ *n*. *X n* = *x*} *U*
⟨*proof*⟩

lemma *Bseq-isLub*: ⋀*X* :: *nat* ⇒ *real*. *Bseq X* ⟹ ∃ *U*. *isLub* (*UNIV*::*real set*) {*x*. ∃ *n*. *X n* = *x*} *U*
⟨*proof*⟩

lemma *isLub-mono-imp-LIMSEQ*:
fixes *X* :: *nat* ⇒ *real*
assumes *u*: *isLub UNIV* {*x*. ∃ *n*. *X n* = *x*} *u*
assumes *X*: ∀ *m n*. *m* ≤ *n* ⟶ *X m* ≤ *X n*
shows *X* ⟶ *u*
⟨*proof*⟩

lemmas *real-isGlb-unique* = *isGlb-unique*[**where** *'a=real*]

lemma *real-le-inf-subset*: $t \neq \{\} \Longrightarrow t \subseteq s \Longrightarrow \exists\, b.\ b <=* s \Longrightarrow Inf\ s \leq Inf\ (t::real\ set)$
⟨*proof*⟩

lemma *real-ge-sup-subset*: $t \neq \{\} \Longrightarrow t \subseteq s \Longrightarrow \exists\, b.\ s *<= b \Longrightarrow Sup\ s \geq Sup\ (t::real\ set)$
⟨*proof*⟩

end

53 An abstract view on maps for code generation.

theory *Mapping*
imports *Main*
begin

53.1 Parametricity transfer rules

lemma *map-of-foldr*: *map-of xs* = *foldr* $(\lambda(k,\ v)\ m.\ m(k \mapsto v))$ *xs Map.empty*
⟨*proof*⟩

context includes *lifting-syntax*
begin

lemma *empty-parametric*: (*A* ===> *rel-option B*) *Map.empty Map.empty*
⟨*proof*⟩

lemma *lookup-parametric*: ((*A* ===> *B*) ===> *A* ===> *B*) $(\lambda m\ k.\ m\ k)$ $(\lambda m\ k.\ m\ k)$
⟨*proof*⟩

lemma *update-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> *B* ===> (*A* ===> *rel-option B*) ===> *A* ===> *rel-option B*)
$(\lambda k\ v\ m.\ m(k \mapsto v))$ $(\lambda k\ v\ m.\ m(k \mapsto v))$
⟨*proof*⟩

lemma *delete-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> (*A* ===> *rel-option B*) ===> *A* ===> *rel-option B*)
$(\lambda k\ m.\ m(k := None))$ $(\lambda k\ m.\ m(k := None))$
⟨*proof*⟩

lemma *is-none-parametric* [*transfer-rule*]:
(*rel-option A* ===> *HOL.eq*) *Option.is-none Option.is-none*
⟨*proof*⟩

lemma *dom-parametric*:
assumes [*transfer-rule*]: *bi-total A*
shows ((*A* ===> *rel-option B*) ===> *rel-set A*) *dom dom*
⟨*proof*⟩

lemma *map-of-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique R1*
shows (*list-all2* (*rel-prod R1 R2*) ===> *R1* ===> *rel-option R2*) *map-of map-of*
⟨*proof*⟩

lemma *map-entry-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> (*B* ===> *B*) ===> (*A* ===> *rel-option B*) ===> *A* ===> *rel-option B*)
(λ*k f m*. (*case m k of None* ⇒ *m*
| *Some v* ⇒ *m* (*k* ↦ (*f v*)))) (λ*k f m*. (*case m k of None* ⇒ *m*
| *Some v* ⇒ *m* (*k* ↦ (*f v*))))
⟨*proof*⟩

lemma *tabulate-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> (*A* ===> *B*) ===> *A* ===> *rel-option B*)
(λ*ks f*. (*map-of* (*map* (λ*k*. (*k*, *f k*)) *ks*))) (λ*ks f*. (*map-of* (*map* (λ*k*. (*k*, *f k*)) *ks*)))
⟨*proof*⟩

lemma *bulkload-parametric*:
(*list-all2 A* ===> *HOL.eq* ===> *rel-option A*)
(λ*xs k*. *if k* < *length xs then Some* (*xs* ! *k*) *else None*)
(λ*xs k*. *if k* < *length xs then Some* (*xs* ! *k*) *else None*)
⟨*proof*⟩

lemma *map-parametric*:
((*A* ===> *B*) ===> (*C* ===> *D*) ===> (*B* ===> *rel-option C*) ===> *A* ===> *rel-option D*)
(λ*f g m*. (*map-option g* ∘ *m* ∘ *f*)) (λ*f g m*. (*map-option g* ∘ *m* ∘ *f*))
⟨*proof*⟩

lemma *combine-with-key-parametric*:
((*A* ===> *B* ===> *B* ===> *B*) ===> (*A* ===> *rel-option B*) ===> (*A* ===> *rel-option B*) ===>
(*A* ===> *rel-option B*)) (λ*f m1 m2 x*. *combine-options* (*f x*) (*m1 x*) (*m2 x*))
(λ*f m1 m2 x*. *combine-options* (*f x*) (*m1 x*) (*m2 x*))
⟨*proof*⟩

lemma *combine-parametric*:
((*B* ===> *B* ===> *B*) ===> (*A* ===> *rel-option B*) ===> (*A* ===>

rel-option B) ===>
(*A* ===> *rel-option B*)) (λ*f m1 m2 x*. *combine-options f* (*m1 x*) (*m2 x*))
(λ*f m1 m2 x*. *combine-options f* (*m1 x*) (*m2 x*))
⟨*proof*⟩

end

53.2 Type definition and primitive operations

typedef ('*a*, '*b*) *mapping* = *UNIV* :: ('*a* ⇀ '*b*) *set*
morphisms *rep Mapping* ⟨*proof*⟩

setup-lifting *type-definition-mapping*

lift-definition *empty* :: ('*a*, '*b*) *mapping*
is *Map.empty* **parametric** *empty-parametric* ⟨*proof*⟩

lift-definition *lookup* :: ('*a*, '*b*) *mapping* ⇒ '*a* ⇒ '*b option*
is λ*m k*. *m k* **parametric** *lookup-parametric* ⟨*proof*⟩

definition *lookup-default d m k* = (*case Mapping.lookup m k of None* ⇒ *d* | *Some v* ⇒ *v*)

lemma [*code abstract*]:
lookup (*Mapping f*) = *f*
⟨*proof*⟩

lift-definition *update* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
is λ*k v m*. *m*(*k* ↦ *v*) **parametric** *update-parametric* ⟨*proof*⟩

lift-definition *delete* :: '*a* ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
is λ*k m*. *m*(*k* := *None*) **parametric** *delete-parametric* ⟨*proof*⟩

lift-definition *filter* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
is λ*P m k*. *case m k of None* ⇒ *None* | *Some v* ⇒ *if P k v then Some v else None* ⟨*proof*⟩

lift-definition *keys* :: ('*a*, '*b*) *mapping* ⇒ '*a set*
is *dom* **parametric** *dom-parametric* ⟨*proof*⟩

lift-definition *tabulate* :: '*a list* ⇒ ('*a* ⇒ '*b*) ⇒ ('*a*, '*b*) *mapping*
is λ*ks f*. (*map-of* (*List.map* (λ*k*. (*k*, *f k*)) *ks*)) **parametric** *tabulate-parametric* ⟨*proof*⟩

lift-definition *bulkload* :: '*a list* ⇒ (*nat*, '*a*) *mapping*
is λ*xs k*. *if k* < *length xs then Some* (*xs* ! *k*) *else None* **parametric** *bulkload-parametric* ⟨*proof*⟩

lift-definition *map* :: ('*c* ⇒ '*a*) ⇒ ('*b* ⇒ '*d*) ⇒ ('*a*, '*b*) *mapping* ⇒ ('*c*, '*d*)

mapping
is $\lambda f\ g\ m.\ (\mathit{map\text{-}option}\ g \circ m \circ f)$ **parametric** *map-parametric* ⟨*proof*⟩

lift-definition *map-values* :: $('c \Rightarrow 'a \Rightarrow 'b) \Rightarrow ('c, 'a)\ \mathit{mapping} \Rightarrow ('c, 'b)\ \mathit{mapping}$
is $\lambda f\ m\ x.\ \mathit{map\text{-}option}\ (f\ x)\ (m\ x)$ ⟨*proof*⟩

lift-definition *combine-with-key* ::
$('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a,'b)\ \mathit{mapping} \Rightarrow ('a,'b)\ \mathit{mapping} \Rightarrow ('a,'b)\ \mathit{mapping}$
is $\lambda f\ m1\ m2\ x.\ \mathit{combine\text{-}options}\ (f\ x)\ (m1\ x)\ (m2\ x)$ **parametric** *combine-with-key-parametric* ⟨*proof*⟩

lift-definition *combine* ::
$('b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a,'b)\ \mathit{mapping} \Rightarrow ('a,'b)\ \mathit{mapping} \Rightarrow ('a,'b)\ \mathit{mapping}$
is $\lambda f\ m1\ m2\ x.\ \mathit{combine\text{-}options}\ f\ (m1\ x)\ (m2\ x)$ **parametric** *combine-parametric* ⟨*proof*⟩

definition *All-mapping* $m\ P \longleftrightarrow$
$(\forall x.\ \mathit{case}\ \mathit{Mapping.lookup}\ m\ x\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{True} \mid \mathit{Some}\ y \Rightarrow P\ x\ y)$

declare [[*code drop*: *map*]]

53.3 Functorial structure

functor *map*: *map*
⟨*proof*⟩

53.4 Derived operations

definition *ordered-keys* :: $('a{::}\mathit{linorder}, 'b)\ \mathit{mapping} \Rightarrow 'a\ \mathit{list}$
where $\mathit{ordered\text{-}keys}\ m = (\mathit{if}\ \mathit{finite}\ (\mathit{keys}\ m)\ \mathit{then}\ \mathit{sorted\text{-}list\text{-}of\text{-}set}\ (\mathit{keys}\ m)\ \mathit{else}\ [])$

definition *is-empty* :: $('a, 'b)\ \mathit{mapping} \Rightarrow \mathit{bool}$
where $\mathit{is\text{-}empty}\ m \longleftrightarrow \mathit{keys}\ m = \{\}$

definition *size* :: $('a, 'b)\ \mathit{mapping} \Rightarrow \mathit{nat}$
where $\mathit{size}\ m = (\mathit{if}\ \mathit{finite}\ (\mathit{keys}\ m)\ \mathit{then}\ \mathit{card}\ (\mathit{keys}\ m)\ \mathit{else}\ 0)$

definition *replace* :: $'a \Rightarrow 'b \Rightarrow ('a, 'b)\ \mathit{mapping} \Rightarrow ('a, 'b)\ \mathit{mapping}$
where $\mathit{replace}\ k\ v\ m = (\mathit{if}\ k \in \mathit{keys}\ m\ \mathit{then}\ \mathit{update}\ k\ v\ m\ \mathit{else}\ m)$

definition *default* :: $'a \Rightarrow 'b \Rightarrow ('a, 'b)\ \mathit{mapping} \Rightarrow ('a, 'b)\ \mathit{mapping}$
where $\mathit{default}\ k\ v\ m = (\mathit{if}\ k \in \mathit{keys}\ m\ \mathit{then}\ m\ \mathit{else}\ \mathit{update}\ k\ v\ m)$

Manual derivation of transfer rule is non-trivial

lift-definition *map-entry* :: $'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b)\ \mathit{mapping} \Rightarrow ('a, 'b)\ \mathit{mapping}$
is
$\lambda k\ f\ m.$
$(\mathit{case}\ m\ k\ \mathit{of}$
$\mathit{None} \Rightarrow m$

| *Some v* ⇒ *m* (*k* ↦ (*f v*))) **parametric** *map-entry-parametric* ⟨*proof*⟩

lemma *map-entry-code* [*code*]:
map-entry k f m =
(*case lookup m k of*
None ⇒ *m*
| *Some v* ⇒ *update k* (*f v*) *m*)
⟨*proof*⟩

definition *map-default* :: *'a* ⇒ *'b* ⇒ (*'b* ⇒ *'b*) ⇒ (*'a*, *'b*) *mapping* ⇒ (*'a*, *'b*) *mapping*
where *map-default k v f m* = *map-entry k f* (*default k v m*)

definition *of-alist* :: (*'k* × *'v*) *list* ⇒ (*'k*, *'v*) *mapping*
where *of-alist xs* = *foldr* (λ(*k*, *v*) *m*. *update k v m*) *xs empty*

instantiation *mapping* :: (*type*, *type*) *equal*
begin

definition *HOL.equal m1 m2* ⟷ (∀ *k*. *lookup m1 k* = *lookup m2 k*)

instance
⟨*proof*⟩

end

context includes *lifting-syntax*
begin

lemma [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A*
and [*transfer-rule*]: *bi-unique B*
shows (*pcr-mapping A B* ===> *pcr-mapping A B* ===> *op*=) *HOL.eq HOL.equal*
⟨*proof*⟩

lemma *of-alist-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique R1*
shows (*list-all2* (*rel-prod R1 R2*) ===> *pcr-mapping R1 R2*) *map-of of-alist*
⟨*proof*⟩

end

53.5 Properties

lemma *mapping-eqI*: (⋀*x*. *lookup m x* = *lookup m' x*) ⟹ *m* = *m'*
⟨*proof*⟩

lemma *mapping-eqI'*:
assumes ⋀*x*. *x* ∈ *Mapping.keys m* ⟹ *Mapping.lookup-default d m x* = *Map-*

ping.lookup-default d m' x
and *Mapping.keys m = Mapping.keys m'*
shows *m = m'*
⟨*proof*⟩

lemma *lookup-update*: *lookup (update k v m) k = Some v*
⟨*proof*⟩

lemma *lookup-update-neq*: *k ≠ k' ⟹ lookup (update k v m) k' = lookup m k'*
⟨*proof*⟩

lemma *lookup-update'*: *Mapping.lookup (update k v m) k' = (if k = k' then Some v else lookup m k')*
⟨*proof*⟩

lemma *lookup-empty*: *lookup empty k = None*
⟨*proof*⟩

lemma *lookup-filter*:
lookup (filter P m) k =
(case lookup m k of
None ⇒ None
| Some v ⇒ if P k v then Some v else None)
⟨*proof*⟩

lemma *lookup-map-values*: *lookup (map-values f m) k = map-option (f k) (lookup m k)*
⟨*proof*⟩

lemma *lookup-default-empty*: *lookup-default d empty k = d*
⟨*proof*⟩

lemma *lookup-default-update*: *lookup-default d (update k v m) k = v*
⟨*proof*⟩

lemma *lookup-default-update-neq*:
k ≠ k' ⟹ lookup-default d (update k v m) k' = lookup-default d m k'
⟨*proof*⟩

lemma *lookup-default-update'*:
lookup-default d (update k v m) k' = (if k = k' then v else lookup-default d m k')
⟨*proof*⟩

lemma *lookup-default-filter*:
lookup-default d (filter P m) k =
(if P k (lookup-default d m k) then lookup-default d m k else d)
⟨*proof*⟩

lemma *lookup-default-map-values*:

lookup-default (*f k d*) (*map-values f m*) *k* = *f k* (*lookup-default d m k*)
⟨*proof*⟩

lemma *lookup-combine-with-key*:
Mapping.lookup (*combine-with-key f m1 m2*) *x* =
combine-options (*f x*) (*Mapping.lookup m1 x*) (*Mapping.lookup m2 x*)
⟨*proof*⟩

lemma *combine-altdef*: *combine f m1 m2* = *combine-with-key* (λ-. *f*) *m1 m2*
⟨*proof*⟩

lemma *lookup-combine*:
Mapping.lookup (*combine f m1 m2*) *x* =
combine-options f (*Mapping.lookup m1 x*) (*Mapping.lookup m2 x*)
⟨*proof*⟩

lemma *lookup-default-neutral-combine-with-key*:
assumes ⋀*x*. *f k d x* = *x* ⋀*x*. *f k x d* = *x*
shows *Mapping.lookup-default d* (*combine-with-key f m1 m2*) *k* =
f k (*Mapping.lookup-default d m1 k*) (*Mapping.lookup-default d m2 k*)
⟨*proof*⟩

lemma *lookup-default-neutral-combine*:
assumes ⋀*x*. *f d x* = *x* ⋀*x*. *f x d* = *x*
shows *Mapping.lookup-default d* (*combine f m1 m2*) *x* =
f (*Mapping.lookup-default d m1 x*) (*Mapping.lookup-default d m2 x*)
⟨*proof*⟩

lemma *lookup-map-entry*: *lookup* (*map-entry x f m*) *x* = *map-option f* (*lookup m x*)
⟨*proof*⟩

lemma *lookup-map-entry-neq*: *x* ≠ *y* ⟹ *lookup* (*map-entry x f m*) *y* = *lookup m y*
⟨*proof*⟩

lemma *lookup-map-entry′*:
lookup (*map-entry x f m*) *y* =
(*if x* = *y then map-option f* (*lookup m y*) *else lookup m y*)
⟨*proof*⟩

lemma *lookup-default*: *lookup* (*default x d m*) *x* = *Some* (*lookup-default d m x*)
⟨*proof*⟩

lemma *lookup-default-neq*: *x* ≠ *y* ⟹ *lookup* (*default x d m*) *y* = *lookup m y*
⟨*proof*⟩

lemma *lookup-default′*:
lookup (*default x d m*) *y* =

(*if* $x = y$ *then Some* (*lookup-default* $d\ m\ x$) *else lookup* $m\ y$)
⟨*proof*⟩

lemma *lookup-map-default*: *lookup* (*map-default* $x\ d\ f\ m$) x = *Some* (f (*lookup-default* $d\ m\ x$))
⟨*proof*⟩

lemma *lookup-map-default-neq*: $x \neq y \Longrightarrow$ *lookup* (*map-default* $x\ d\ f\ m$) y = *lookup* $m\ y$
⟨*proof*⟩

lemma *lookup-map-default'*:
lookup (*map-default* $x\ d\ f\ m$) y =
(*if* $x = y$ *then Some* (f (*lookup-default* $d\ m\ x$)) *else lookup* $m\ y$)
⟨*proof*⟩

lemma *lookup-tabulate*:
assumes *distinct xs*
shows *Mapping.lookup* (*Mapping.tabulate xs f*) x = (*if* $x \in$ *set xs then Some* (f x) *else None*)
⟨*proof*⟩

lemma *lookup-of-alist*: *Mapping.lookup* (*Mapping.of-alist xs*) k = *map-of xs k*
⟨*proof*⟩

lemma *keys-is-none-rep* [*code-unfold*]: $k \in$ *keys* $m \longleftrightarrow \neg$ (*Option.is-none* (*lookup* $m\ k$))
⟨*proof*⟩

lemma *update-update*:
update $k\ v$ (*update* $k\ w\ m$) = *update* $k\ v\ m$
$k \neq l \Longrightarrow$ *update* $k\ v$ (*update* $l\ w\ m$) = *update* $l\ w$ (*update* $k\ v\ m$)
⟨*proof*⟩

lemma *update-delete* [*simp*]: *update* $k\ v$ (*delete* $k\ m$) = *update* $k\ v\ m$
⟨*proof*⟩

lemma *delete-update*:
delete k (*update* $k\ v\ m$) = *delete* $k\ m$
$k \neq l \Longrightarrow$ *delete* k (*update* $l\ v\ m$) = *update* $l\ v$ (*delete* $k\ m$)
⟨*proof*⟩

lemma *delete-empty* [*simp*]: *delete* k *empty* = *empty*
⟨*proof*⟩

lemma *replace-update*:
$k \notin$ *keys* $m \Longrightarrow$ *replace* $k\ v\ m = m$
$k \in$ *keys* $m \Longrightarrow$ *replace* $k\ v\ m$ = *update* $k\ v\ m$
⟨*proof*⟩

lemma *map-values-update*: *map-values f* (*update k v m*) = *update k* (*f k v*) (*map-values f m*)
⟨*proof*⟩

lemma *size-mono*: *finite* (*keys m'*) ⟹ *keys m* ⊆ *keys m'* ⟹ *size m* ≤ *size m'*
⟨*proof*⟩

lemma *size-empty* [*simp*]: *size empty* = *0*
⟨*proof*⟩

lemma *size-update*:
finite (*keys m*) ⟹ *size* (*update k v m*) =
(*if k* ∈ *keys m then size m else Suc* (*size m*))
⟨*proof*⟩

lemma *size-delete*: *size* (*delete k m*) = (*if k* ∈ *keys m then size m* − *1 else size m*)
⟨*proof*⟩

lemma *size-tabulate* [*simp*]: *size* (*tabulate ks f*) = *length* (*remdups ks*)
⟨*proof*⟩

lemma *keys-filter*: *keys* (*filter P m*) ⊆ *keys m*
⟨*proof*⟩

lemma *size-filter*: *finite* (*keys m*) ⟹ *size* (*filter P m*) ≤ *size m*
⟨*proof*⟩

lemma *bulkload-tabulate*: *bulkload xs* = *tabulate* [*0..<length xs*] (*nth xs*)
⟨*proof*⟩

lemma *is-empty-empty* [*simp*]: *is-empty empty*
⟨*proof*⟩

lemma *is-empty-update* [*simp*]: ¬ *is-empty* (*update k v m*)
⟨*proof*⟩

lemma *is-empty-delete*: *is-empty* (*delete k m*) ⟷ *is-empty m* ∨ *keys m* = {*k*}
⟨*proof*⟩

lemma *is-empty-replace* [*simp*]: *is-empty* (*replace k v m*) ⟷ *is-empty m*
⟨*proof*⟩

lemma *is-empty-default* [*simp*]: ¬ *is-empty* (*default k v m*)
⟨*proof*⟩

lemma *is-empty-map-entry* [*simp*]: *is-empty* (*map-entry k f m*) ⟷ *is-empty m*
⟨*proof*⟩

lemma *is-empty-map-values* [*simp*]: *is-empty* (*map-values f m*) ⟷ *is-empty m*
⟨*proof*⟩

lemma *is-empty-map-default* [*simp*]: ¬ *is-empty* (*map-default k v f m*)
⟨*proof*⟩

lemma *keys-dom-lookup*: *keys m* = *dom* (*Mapping.lookup m*)
⟨*proof*⟩

lemma *keys-empty* [*simp*]: *keys empty* = {}
⟨*proof*⟩

lemma *keys-update* [*simp*]: *keys* (*update k v m*) = *insert k* (*keys m*)
⟨*proof*⟩

lemma *keys-delete* [*simp*]: *keys* (*delete k m*) = *keys m* − {*k*}
⟨*proof*⟩

lemma *keys-replace* [*simp*]: *keys* (*replace k v m*) = *keys m*
⟨*proof*⟩

lemma *keys-default* [*simp*]: *keys* (*default k v m*) = *insert k* (*keys m*)
⟨*proof*⟩

lemma *keys-map-entry* [*simp*]: *keys* (*map-entry k f m*) = *keys m*
⟨*proof*⟩

lemma *keys-map-default* [*simp*]: *keys* (*map-default k v f m*) = *insert k* (*keys m*)
⟨*proof*⟩

lemma *keys-map-values* [*simp*]: *keys* (*map-values f m*) = *keys m*
⟨*proof*⟩

lemma *keys-combine-with-key* [*simp*]:
Mapping.keys (*combine-with-key f m1 m2*) = *Mapping.keys m1* ∪ *Mapping.keys m2*
⟨*proof*⟩

lemma *keys-combine* [*simp*]: *Mapping.keys* (*combine f m1 m2*) = *Mapping.keys m1* ∪ *Mapping.keys m2*
⟨*proof*⟩

lemma *keys-tabulate* [*simp*]: *keys* (*tabulate ks f*) = *set ks*
⟨*proof*⟩

lemma *keys-of-alist* [*simp*]: *keys* (*of-alist xs*) = *set* (*List.map fst xs*)
⟨*proof*⟩

lemma *keys-bulkload* [*simp*]: *keys* (*bulkload xs*) = {*0*..<*length xs*}
⟨*proof*⟩

lemma *distinct-ordered-keys* [*simp*]: *distinct* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-infinite* [*simp*]: ¬ *finite* (*keys m*) ⟹ *ordered-keys m* = []
⟨*proof*⟩

lemma *ordered-keys-empty* [*simp*]: *ordered-keys empty* = []
⟨*proof*⟩

lemma *ordered-keys-update* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*update k v m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹
ordered-keys (*update k v m*) = *insort k* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-delete* [*simp*]: *ordered-keys* (*delete k m*) = *remove1 k* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-replace* [*simp*]: *ordered-keys* (*replace k v m*) = *ordered-keys m*
⟨*proof*⟩

lemma *ordered-keys-default* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*default k v m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹ *ordered-keys* (*default k v m*) = *insort k* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-map-entry* [*simp*]: *ordered-keys* (*map-entry k f m*) = *ordered-keys m*
⟨*proof*⟩

lemma *ordered-keys-map-default* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*map-default k v f m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹ *ordered-keys* (*map-default k v f m*) = *insort k* (*ordered-keys m*)
⟨*proof*⟩

lemma *ordered-keys-tabulate* [*simp*]: *ordered-keys* (*tabulate ks f*) = *sort* (*remdups ks*)
⟨*proof*⟩

lemma *ordered-keys-bulkload* [*simp*]: *ordered-keys* (*bulkload ks*) = [*0*..<*length ks*]
⟨*proof*⟩

lemma *tabulate-fold*: *tabulate xs f* = *fold* (λ*k m*. *update k* (*f k*) *m*) *xs empty*

⟨*proof*⟩

lemma *All-mapping-mono*:
(⋀*k v*. *k* ∈ *keys m* ⟹ *P k v* ⟹ *Q k v*) ⟹ *All-mapping m P* ⟹ *All-mapping m Q*
⟨*proof*⟩

lemma *All-mapping-empty* [*simp*]: *All-mapping Mapping.empty P*
⟨*proof*⟩

lemma *All-mapping-update-iff*:
All-mapping (*Mapping.update k v m*) *P* ⟷ *P k v* ∧ *All-mapping m* (λ*k' v'*. *k* = *k'* ∨ *P k' v'*)
⟨*proof*⟩

lemma *All-mapping-update*:
P k v ⟹ *All-mapping m* (λ*k' v'*. *k* = *k'* ∨ *P k' v'*) ⟹ *All-mapping* (*Mapping.update k v m*) *P*
⟨*proof*⟩

lemma *All-mapping-filter-iff*: *All-mapping* (*filter P m*) *Q* ⟷ *All-mapping m* (λ*k v*. *P k v* ⟶ *Q k v*)
⟨*proof*⟩

lemma *All-mapping-filter*: *All-mapping m Q* ⟹ *All-mapping* (*filter P m*) *Q*
⟨*proof*⟩

lemma *All-mapping-map-values*: *All-mapping* (*map-values f m*) *P* ⟷ *All-mapping m* (λ*k v*. *P k* (*f k v*))
⟨*proof*⟩

lemma *All-mapping-tabulate*: (∀ *x*∈*set xs*. *P x* (*f x*)) ⟹ *All-mapping* (*Mapping.tabulate xs f*) *P*
⟨*proof*⟩

lemma *All-mapping-alist*:
(⋀*k v*. (*k*, *v*) ∈ *set xs* ⟹ *P k v*) ⟹ *All-mapping* (*Mapping.of-alist xs*) *P*
⟨*proof*⟩

lemma *combine-empty* [*simp*]: *combine f Mapping.empty y* = *y combine f y Mapping.empty* = *y*
⟨*proof*⟩

lemma (**in** *abel-semigroup*) *comm-monoid-set-combine*: *comm-monoid-set* (*combine f*) *Mapping.empty*
⟨*proof*⟩

locale *combine-mapping-abel-semigroup* = *abel-semigroup*
begin

sublocale *combine*: *comm-monoid-set combine f Mapping.empty*
⟨*proof*⟩

lemma *fold-combine-code*:
combine.F g (*set xs*) = *foldr* (λ*x*. *combine f* (*g x*)) (*remdups xs*) *Mapping.empty*
⟨*proof*⟩

lemma *keys-fold-combine*: *finite A* ⟹ *Mapping.keys* (*combine.F g A*) = (⋃ *x*∈*A*. *Mapping.keys* (*g x*))
⟨*proof*⟩

end

53.6 Code generator setup

hide-const (**open**) *empty is-empty rep lookup lookup-default filter update delete ordered-keys*
keys size replace default map-entry map-default tabulate bulkload map map-values combine of-alist

end

54 Adhoc overloading of constants based on their types

theory *Adhoc-Overloading*
imports *Pure*
keywords
adhoc-overloading no-adhoc-overloading :: *thy-decl*
begin

⟨*ML*⟩

end

55 Monad notation for arbitrary types

theory *Monad-Syntax*
imports *Main* ~~/*src*/*Tools*/*Adhoc-Overloading*
begin

We provide a convenient do-notation for monadic expressions well-known from Haskell. *Let* is printed specially in do-expressions.

consts
bind :: ['*a*, '*b* ⇒ '*c*] ⇒ '*d* (**infixr** ⤜ *54*)

notation (*ASCII*)

bind (**infixr** *>>= 54*)

abbreviation (*do-notation*)
bind-do :: [*'a*, *'b* ⇒ *'c*] ⇒ *'d*
where *bind-do* ≡ *bind*

notation (**output**)
bind-do (**infixr** *≫= 54*)

notation (*ASCII* **output**)
bind-do (**infixr** *>>= 54*)

nonterminal *do-binds* **and** *do-bind*
syntax
-do-block :: *do-binds* ⇒ *'a* (*do* {//(*2* -)//} [*12*] *62*)
-do-bind :: [*pttrn*, *'a*] ⇒ *do-bind* ((*2*- ←/ -) *13*)
-do-let :: [*pttrn*, *'a*] ⇒ *do-bind* ((*2let* - =/ -) [*1000*, *13*] *13*)
-do-then :: *'a* ⇒ *do-bind* (- [*14*] *13*)
-do-final :: *'a* ⇒ *do-binds* (-)
-do-cons :: [*do-bind*, *do-binds*] ⇒ *do-binds* (-;//- [*13*, *12*] *12*)
-thenM :: [*'a*, *'b*] ⇒ *'c* (**infixr** *≫ 54*)

syntax (*ASCII*)
-do-bind :: [*pttrn*, *'a*] ⇒ *do-bind* ((*2*- <−/ -) *13*)
-thenM :: [*'a*, *'b*] ⇒ *'c* (**infixr** *>> 54*)

translations
-do-block (*-do-cons* (*-do-then t*) (*-do-final e*))
⇌ *CONST bind-do t* (*λ-. e*)
-do-block (*-do-cons* (*-do-bind p t*) (*-do-final e*))
⇌ *CONST bind-do t* (*λp. e*)
-do-block (*-do-cons* (*-do-let p t*) *bs*)
⇌ *let p = t in -do-block bs*
-do-block (*-do-cons b* (*-do-cons c cs*))
⇌ *-do-block* (*-do-cons b* (*-do-final* (*-do-block* (*-do-cons c cs*))))
-do-cons (*-do-let p t*) (*-do-final s*)
⇌ *-do-final* (*let p = t in s*)
-do-block (*-do-final e*) ⇀ *e*
(*m ≫ n*) ⇀ (*m ≫= λ-. n*))

adhoc-overloading
bind Set.bind Predicate.bind Option.bind List.bind

end

56 Less common functions on lists

theory *More-List*
imports *Main*
begin

definition *strip-while* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$
where
  $strip\text{-}while\ P = rev \circ dropWhile\ P \circ rev$

lemma *strip-while-rev* [*simp*]:
  $strip\text{-}while\ P\ (rev\ xs) = rev\ (dropWhile\ P\ xs)$
  ⟨*proof*⟩

lemma *strip-while-Nil* [*simp*]:
  $strip\text{-}while\ P\ [] = []$
  ⟨*proof*⟩

lemma *strip-while-append* [*simp*]:
  $\neg\ P\ x \Longrightarrow strip\text{-}while\ P\ (xs\ @\ [x]) = xs\ @\ [x]$
  ⟨*proof*⟩

lemma *strip-while-append-rec* [*simp*]:
  $P\ x \Longrightarrow strip\text{-}while\ P\ (xs\ @\ [x]) = strip\text{-}while\ P\ xs$
  ⟨*proof*⟩

lemma *strip-while-Cons* [*simp*]:
  $\neg\ P\ x \Longrightarrow strip\text{-}while\ P\ (x\ \#\ xs) = x\ \#\ strip\text{-}while\ P\ xs$
  ⟨*proof*⟩

lemma *strip-while-eq-Nil* [*simp*]:
  $strip\text{-}while\ P\ xs = [] \longleftrightarrow (\forall\, x{\in}set\ xs.\ P\ x)$
  ⟨*proof*⟩

lemma *strip-while-eq-Cons-rec*:
  $strip\text{-}while\ P\ (x\ \#\ xs) = x\ \#\ strip\text{-}while\ P\ xs \longleftrightarrow \neg\ (P\ x \wedge (\forall\, x{\in}set\ xs.\ P\ x))$
  ⟨*proof*⟩

lemma *split-strip-while-append*:
  fixes $xs :: 'a\ list$
  obtains $ys\ zs :: 'a\ list$
  where $strip\text{-}while\ P\ xs = ys$ **and** $\forall\, x{\in}set\ zs.\ P\ x$ **and** $xs = ys\ @\ zs$
⟨*proof*⟩

lemma *strip-while-snoc* [*simp*]:
  $strip\text{-}while\ P\ (xs\ @\ [x]) = (if\ P\ x\ then\ strip\text{-}while\ P\ xs\ else\ xs\ @\ [x])$
  ⟨*proof*⟩

lemma *strip-while-map*:

strip-while P (map f xs) = map f (strip-while (P ∘ f) xs)
⟨*proof*⟩

lemma *strip-while-dropWhile-commute*:
strip-while P (dropWhile Q xs) = dropWhile Q (strip-while P xs)
⟨*proof*⟩

lemma *dropWhile-strip-while-commute*:
dropWhile P (strip-while Q xs) = strip-while Q (dropWhile P xs)
⟨*proof*⟩

definition *no-leading* :: *('a ⇒ bool) ⇒ 'a list ⇒ bool*
where
no-leading P xs ⟷ (xs ≠ [] ⟶ ¬ P (hd xs))

lemma *no-leading-Nil* [*simp*, *intro*!]:
no-leading P []
⟨*proof*⟩

lemma *no-leading-Cons* [*simp*, *intro*!]:
no-leading P (x # xs) ⟷ ¬ P x
⟨*proof*⟩

lemma *no-leading-append* [*simp*]:
no-leading P (xs @ ys) ⟷ no-leading P xs ∧ (xs = [] ⟶ no-leading P ys)
⟨*proof*⟩

lemma *no-leading-dropWhile* [*simp*]:
no-leading P (dropWhile P xs)
⟨*proof*⟩

lemma *dropWhile-eq-obtain-leading*:
assumes *dropWhile P xs = ys*
obtains *zs* **where** *xs = zs @ ys* **and** *⋀z. z ∈ set zs ⟹ P z* **and** *no-leading P ys*
⟨*proof*⟩

lemma *dropWhile-idem-iff*:
dropWhile P xs = xs ⟷ no-leading P xs
⟨*proof*⟩

abbreviation *no-trailing* :: *('a ⇒ bool) ⇒ 'a list ⇒ bool*
where
no-trailing P xs ≡ no-leading P (rev xs)

lemma *no-trailing-unfold*:
no-trailing P xs ⟷ (xs ≠ [] ⟶ ¬ P (last xs))

⟨*proof*⟩

lemma *no-trailing-Nil* [*simp*, *intro*!]:
no-trailing P []
⟨*proof*⟩

lemma *no-trailing-Cons* [*simp*]:
no-trailing P (*x* # *xs*) ⟷ *no-trailing P xs* ∧ (*xs* = [] ⟶ ¬ *P x*)
⟨*proof*⟩

lemma *no-trailing-append*:
no-trailing P (*xs* @ *ys*) ⟷ *no-trailing P ys* ∧ (*ys* = [] ⟶ *no-trailing P xs*)
⟨*proof*⟩

lemma *no-trailing-append-Cons* [*simp*]:
no-trailing P (*xs* @ *y* # *ys*) ⟷ *no-trailing P* (*y* # *ys*)
⟨*proof*⟩

lemma *no-trailing-strip-while* [*simp*]:
no-trailing P (*strip-while P xs*)
⟨*proof*⟩

lemma *strip-while-idem* [*simp*]:
no-trailing P xs ⟹ *strip-while P xs* = *xs*
⟨*proof*⟩

lemma *strip-while-eq-obtain-trailing*:
assumes *strip-while P xs* = *ys*
obtains *zs* **where** *xs* = *ys* @ *zs* **and** ⋀*z*. *z* ∈ *set zs* ⟹ *P z* **and** *no-trailing P ys*
⟨*proof*⟩

lemma *strip-while-idem-iff*:
strip-while P xs = *xs* ⟷ *no-trailing P xs*
⟨*proof*⟩

lemma *no-trailing-map*:
no-trailing P (*map f xs*) ⟷ *no-trailing* (*P* ∘ *f*) *xs*
⟨*proof*⟩

lemma *no-trailing-drop* [*simp*]:
no-trailing P (*drop n xs*) **if** *no-trailing P xs*
⟨*proof*⟩

lemma *no-trailing-upt* [*simp*]:
no-trailing P [*n*..<*m*] ⟷ (*n* < *m* ⟶ ¬ *P* (*m* − *1*))
⟨*proof*⟩

definition *nth-default* :: *'a* ⇒ *'a list* ⇒ *nat* ⇒ *'a*
where
nth-default dflt xs n = (*if n* < *length xs then xs* ! *n else dflt*)

lemma *nth-default-nth*:
n < *length xs* ⟹ *nth-default dflt xs n* = *xs* ! *n*
⟨*proof*⟩

lemma *nth-default-beyond*:
length xs ≤ *n* ⟹ *nth-default dflt xs n* = *dflt*
⟨*proof*⟩

lemma *nth-default-Nil* [*simp*]:
nth-default dflt [] *n* = *dflt*
⟨*proof*⟩

lemma *nth-default-Cons*:
nth-default dflt (*x* # *xs*) *n* = (*case n of 0* ⇒ *x* | *Suc n'* ⇒ *nth-default dflt xs n'*)
⟨*proof*⟩

lemma *nth-default-Cons-0* [*simp*]:
nth-default dflt (*x* # *xs*) *0* = *x*
⟨*proof*⟩

lemma *nth-default-Cons-Suc* [*simp*]:
nth-default dflt (*x* # *xs*) (*Suc n*) = *nth-default dflt xs n*
⟨*proof*⟩

lemma *nth-default-replicate-dflt* [*simp*]:
nth-default dflt (*replicate n dflt*) *m* = *dflt*
⟨*proof*⟩

lemma *nth-default-append*:
nth-default dflt (*xs* @ *ys*) *n* =
(*if n* < *length xs then nth xs n else nth-default dflt ys* (*n* − *length xs*))
⟨*proof*⟩

lemma *nth-default-append-trailing* [*simp*]:
nth-default dflt (*xs* @ *replicate n dflt*) = *nth-default dflt xs*
⟨*proof*⟩

lemma *nth-default-snoc-default* [*simp*]:
nth-default dflt (*xs* @ [*dflt*]) = *nth-default dflt xs*
⟨*proof*⟩

lemma *nth-default-eq-dflt-iff*:
nth-default dflt xs k = *dflt* ⟷ (*k* < *length xs* ⟶ *xs* ! *k* = *dflt*)
⟨*proof*⟩

lemma *in-enumerate-iff-nth-default-eq*:
x ≠ *dflt* ⟹ (*n*, *x*) ∈ *set* (*enumerate 0 xs*) ⟷ *nth-default dflt xs n* = *x*
⟨*proof*⟩

lemma *last-conv-nth-default*:
assumes *xs* ≠ []
shows *last xs* = *nth-default dflt xs* (*length xs* − *1*)
⟨*proof*⟩

lemma *nth-default-map-eq*:
f dflt' = *dflt* ⟹ *nth-default dflt* (*map f xs*) *n* = *f* (*nth-default dflt' xs n*)
⟨*proof*⟩

lemma *finite-nth-default-neq-default* [*simp*]:
finite {*k*. *nth-default dflt xs k* ≠ *dflt*}
⟨*proof*⟩

lemma *sorted-list-of-set-nth-default*:
sorted-list-of-set {*k*. *nth-default dflt xs k* ≠ *dflt*} = *map fst* (*filter* (λ(-, *x*). *x* ≠ *dflt*) (*enumerate 0 xs*))
⟨*proof*⟩

lemma *map-nth-default*:
map (*nth-default x xs*) [*0*..<*length xs*] = *xs*
⟨*proof*⟩

lemma *range-nth-default* [*simp*]:
range (*nth-default dflt xs*) = *insert dflt* (*set xs*)
⟨*proof*⟩

lemma *nth-strip-while*:
assumes *n* < *length* (*strip-while P xs*)
shows *strip-while P xs* ! *n* = *xs* ! *n*
⟨*proof*⟩

lemma *length-strip-while-le*:
length (*strip-while P xs*) ≤ *length xs*
⟨*proof*⟩

lemma *nth-default-strip-while-dflt* [*simp*]:
nth-default dflt (*strip-while* (*op* = *dflt*) *xs*) = *nth-default dflt xs*
⟨*proof*⟩

lemma *nth-default-eq-iff*:
nth-default dflt xs = *nth-default dflt ys*
⟷ *strip-while* (*HOL.eq dflt*) *xs* = *strip-while* (*HOL.eq dflt*) *ys* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

end

theory *Cancellation*
imports *Main*
begin

named-theorems *cancelation-simproc-pre* ‹*These theorems are here to normalise the term. Special handling of constructors should be here. Remark that only the simproc @{term NO-MATCH} is also included.*›

named-theorems *cancelation-simproc-post* ‹*These theorems are here to normalise the term, after the cancelation simproc. Normalisation of* ‹*iterate-add*› *back to the normale representation should be put here.*›

named-theorems *cancelation-simproc-eq-elim* ‹*These theorems are here to help deriving contradiction (e.g.,* ‹*Suc - = 0*›*).*›

definition *iterate-add* :: ‹*nat* ⇒ *'a*::*cancel-comm-monoid-add* ⇒ *'a*› **where**
‹*iterate-add n a* = ((*op* + *a*) ^^ *n*) *0*›

lemma *iterate-add-simps*[*simp*]:
‹*iterate-add 0 a* = *0*›
‹*iterate-add* (*Suc n*) *a* = *a* + *iterate-add n a*›
⟨*proof*⟩

lemma *iterate-add-empty*[*simp*]: ‹*iterate-add n 0* = *0*›
⟨*proof*⟩

lemma *iterate-add-distrib*[*simp*]: ‹*iterate-add* (*m*+*n*) *a* = *iterate-add m a* + *iterate-add n a*›
⟨*proof*⟩

lemma *iterate-add-Numeral1*: ‹*iterate-add n Numeral1* = *of-nat n*›
⟨*proof*⟩

lemma *iterate-add-1*: ‹*iterate-add n 1* = *of-nat n*›
⟨*proof*⟩

lemma *iterate-add-eq-add-iff1*:
‹*i* ≤ *j* ⟹ (*iterate-add j u* + *m* = *iterate-add i u* + *n*) = (*iterate-add* (*j* − *i*) *u* + *m* = *n*)›
⟨*proof*⟩

lemma *iterate-add-eq-add-iff2*:

‹$i \leq j \Longrightarrow (\mathit{iterate\text{-}add}\ i\ u + m = \mathit{iterate\text{-}add}\ j\ u + n) = (m = \mathit{iterate\text{-}add}\ (j - i)\ u + n)$›
⟨*proof*⟩

lemma *iterate-add-less-iff1*:
$j \leq (i::nat) \Longrightarrow (\mathit{iterate\text{-}add}\ i\ (u::'a::\{\mathit{cancel\text{-}comm\text{-}monoid\text{-}add}, \mathit{ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le}\}) + m < \mathit{iterate\text{-}add}\ j\ u + n) = (\mathit{iterate\text{-}add}\ (i-j)\ u + m < n)$
⟨*proof*⟩

lemma *iterate-add-less-iff2*:
$i \leq (j::nat) \Longrightarrow (\mathit{iterate\text{-}add}\ i\ (u::'a::\{\mathit{cancel\text{-}comm\text{-}monoid\text{-}add}, \mathit{ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le}\}) + m < \mathit{iterate\text{-}add}\ j\ u + n) = (m < \mathit{iterate\text{-}add}\ (j - i)\ u + n)$
⟨*proof*⟩

lemma *iterate-add-less-eq-iff1*:
$j \leq (i::nat) \Longrightarrow (\mathit{iterate\text{-}add}\ i\ (u::'a::\{\mathit{cancel\text{-}comm\text{-}monoid\text{-}add}, \mathit{ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le}\}) + m \leq \mathit{iterate\text{-}add}\ j\ u + n) = (\mathit{iterate\text{-}add}\ (i-j)\ u + m \leq n)$
⟨*proof*⟩

lemma *iterate-add-less-eq-iff2*:
$i \leq (j::nat) \Longrightarrow (\mathit{iterate\text{-}add}\ i\ (u::'a::\{\mathit{cancel\text{-}comm\text{-}monoid\text{-}add}, \mathit{ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le}\}) + m \leq \mathit{iterate\text{-}add}\ j\ u + n) = (m \leq \mathit{iterate\text{-}add}\ (j - i)\ u + n)$
⟨*proof*⟩

lemma *iterate-add-add-eq1*:
$j \leq (i::nat) \Longrightarrow ((\mathit{iterate\text{-}add}\ i\ u + m) - (\mathit{iterate\text{-}add}\ j\ u + n)) = ((\mathit{iterate\text{-}add}\ (i-j)\ u + m) - n)$
⟨*proof*⟩

lemma *iterate-add-diff-add-eq2*:
$i \leq (j::nat) \Longrightarrow ((\mathit{iterate\text{-}add}\ i\ u + m) - (\mathit{iterate\text{-}add}\ j\ u + n)) = (m - (\mathit{iterate\text{-}add}\ (j-i)\ u + n))$
⟨*proof*⟩

56.1 Simproc Set-Up

⟨*ML*⟩

end

57 (Finite) Multisets

theory *Multiset*
imports *Cancellation*
begin

57.1 The type of multisets

definition $\mathit{multiset} = \{f :: 'a \Rightarrow nat.\ \mathit{finite}\ \{x.\ f\ x > 0\}\}$

typedef *'a multiset = multiset :: ('a ⇒ nat) set*
morphisms *count Abs-multiset*
⟨*proof*⟩

setup-lifting *type-definition-multiset*

lemma *multiset-eq-iff*: $M = N \longleftrightarrow (\forall a.\ count\ M\ a = count\ N\ a)$
⟨*proof*⟩

lemma *multiset-eqI*: $(\bigwedge x.\ count\ A\ x = count\ B\ x) \Longrightarrow A = B$
⟨*proof*⟩

Preservation of the representing set *multiset*.

lemma *const0-in-multiset*: $(\lambda a.\ 0) \in multiset$
⟨*proof*⟩

lemma *only1-in-multiset*: $(\lambda b.\ if\ b = a\ then\ n\ else\ 0) \in multiset$
⟨*proof*⟩

lemma *union-preserves-multiset*: $M \in multiset \Longrightarrow N \in multiset \Longrightarrow (\lambda a.\ M\ a + N\ a) \in multiset$
⟨*proof*⟩

lemma *diff-preserves-multiset*:
assumes $M \in multiset$
shows $(\lambda a.\ M\ a - N\ a) \in multiset$
⟨*proof*⟩

lemma *filter-preserves-multiset*:
assumes $M \in multiset$
shows $(\lambda x.\ if\ P\ x\ then\ M\ x\ else\ 0) \in multiset$
⟨*proof*⟩

lemmas *in-multiset = const0-in-multiset only1-in-multiset*
union-preserves-multiset diff-preserves-multiset filter-preserves-multiset

57.2 Representing multisets

Multiset enumeration

instantiation *multiset* :: (*type*) *cancel-comm-monoid-add*
begin

lift-definition *zero-multiset* :: *'a multiset* **is** $\lambda a.\ 0$
⟨*proof*⟩

abbreviation *Mempty* :: *'a multiset* ({#}) **where**
Mempty ≡ *0*

lift-definition *plus-multiset* :: *'a multiset ⇒ 'a multiset ⇒ 'a multiset* **is** *λM N. (λa. M a + N a)*
⟨*proof*⟩

lift-definition *minus-multiset* :: *'a multiset ⇒ 'a multiset ⇒ 'a multiset* **is** *λ M N. λa. M a − N a*
⟨*proof*⟩

instance
⟨*proof*⟩

end

context
begin

qualified definition *is-empty* :: *'a multiset ⇒ bool* **where**
[*code-abbrev*]: *is-empty A ⟷ A = {#}*

end

lemma *add-mset-in-multiset*:
assumes *M*: ‹*M ∈ multiset*›
shows ‹*(λb. if b = a then Suc (M b) else M b) ∈ multiset*›
⟨*proof*⟩

lift-definition *add-mset* :: *'a ⇒ 'a multiset ⇒ 'a multiset* **is**
λa M b. if b = a then Suc (M b) else M b
⟨*proof*⟩

syntax
-multiset :: *args ⇒ 'a multiset* ({#(-)#})
translations
{#x, xs#} == CONST add-mset x {#xs#}
{#x#} == CONST add-mset x {#}

lemma *count-empty* [*simp*]: *count {#} a = 0*
⟨*proof*⟩

lemma *count-add-mset* [*simp*]:
count (add-mset b A) a = (if b = a then Suc (count A a) else count A a)
⟨*proof*⟩

lemma *count-single*: *count {#b#} a = (if b = a then 1 else 0)*
⟨*proof*⟩

lemma
add-mset-not-empty [*simp*]: ‹*add-mset a A ≠ {#}*› **and**
empty-not-add-mset [*simp*]: *{#} ≠ add-mset a A*

⟨*proof*⟩

lemma *add-mset-add-mset-same-iff* [*simp*]:
add-mset a A = *add-mset a B* ⟷ *A* = *B*
⟨*proof*⟩

lemma *add-mset-commute*:
add-mset x (*add-mset y M*) = *add-mset y* (*add-mset x M*)
⟨*proof*⟩

57.3 Basic operations

57.3.1 Conversion to set and membership

definition *set-mset* :: *'a multiset* ⇒ *'a set*
where *set-mset M* = {*x*. *count M x* > *0*}

abbreviation *Melem* :: *'a* ⇒ *'a multiset* ⇒ *bool*
where *Melem a M* ≡ *a* ∈ *set-mset M*

notation
Melem (*op* ∈#) **and**
Melem ((-/ ∈# -) [*51*, *51*] *50*)

notation (*ASCII*)
Melem (*op* :#) **and**
Melem ((-/ :# -) [*51*, *51*] *50*)

abbreviation *not-Melem* :: *'a* ⇒ *'a multiset* ⇒ *bool*
where *not-Melem a M* ≡ *a* ∉ *set-mset M*

notation
not-Melem (*op* ∉#) **and**
not-Melem ((-/ ∉# -) [*51*, *51*] *50*)

notation (*ASCII*)
not-Melem (*op* ~:#) **and**
not-Melem ((-/ ~:# -) [*51*, *51*] *50*)

context
begin

qualified abbreviation *Ball* :: *'a multiset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where *Ball M* ≡ *Set.Ball* (*set-mset M*)

qualified abbreviation *Bex* :: *'a multiset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where *Bex M* ≡ *Set.Bex* (*set-mset M*)

end

syntax
-MBall :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* ((*3*∀ -∈#-./ -) [*0*, *0*, *10*] *10*)
-MBex :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* ((*3*∃ -∈#-./ -) [*0*, *0*, *10*] *10*)

syntax (*ASCII*)
-MBall :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* ((*3*ALL -:#-./ -) [*0*, *0*, *10*] *10*)
-MBex :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* ((*3*EX -:#-./ -) [*0*, *0*, *10*] *10*)

translations
∀ *x*∈#*A*. *P* ⇌ *CONST Multiset.Ball A* (λ*x*. *P*)
∃ *x*∈#*A*. *P* ⇌ *CONST Multiset.Bex A* (λ*x*. *P*)

lemma *count-eq-zero-iff*:
count M x = *0* ⟷ *x* ∉# *M*
⟨*proof*⟩

lemma *not-in-iff*:
x ∉# *M* ⟷ *count M x* = *0*
⟨*proof*⟩

lemma *count-greater-zero-iff* [*simp*]:
count M x > *0* ⟷ *x* ∈# *M*
⟨*proof*⟩

lemma *count-inI*:
assumes *count M x* = *0* ⟹ *False*
shows *x* ∈# *M*
⟨*proof*⟩

lemma *in-countE*:
assumes *x* ∈# *M*
obtains *n* **where** *count M x* = *Suc n*
⟨*proof*⟩

lemma *count-greater-eq-Suc-zero-iff* [*simp*]:
count M x ≥ *Suc 0* ⟷ *x* ∈# *M*
⟨*proof*⟩

lemma *count-greater-eq-one-iff* [*simp*]:
count M x ≥ *1* ⟷ *x* ∈# *M*
⟨*proof*⟩

lemma *set-mset-empty* [*simp*]:
set-mset {#} = {}
⟨*proof*⟩

lemma *set-mset-single*:
set-mset {#*b*#} = {*b*}
⟨*proof*⟩

lemma *set-mset-eq-empty-iff* [*simp*]:
set-mset M = {} ⟷ M = {#}
⟨*proof*⟩

lemma *finite-set-mset* [*iff*]:
finite (set-mset M)
⟨*proof*⟩

lemma *set-mset-add-mset-insert* [*simp*]: ‹*set-mset (add-mset a A) = insert a (set-mset A)*›
⟨*proof*⟩

lemma *multiset-nonemptyE* [*elim*]:
assumes *A ≠ {#}*
obtains *x* **where** *x ∈# A*
⟨*proof*⟩

57.3.2 Union

lemma *count-union* [*simp*]:
count (M + N) a = count M a + count N a
⟨*proof*⟩

lemma *set-mset-union* [*simp*]:
set-mset (M + N) = set-mset M ∪ set-mset N
⟨*proof*⟩

lemma *union-mset-add-mset-left* [*simp*]:
add-mset a A + B = add-mset a (A + B)
⟨*proof*⟩

lemma *union-mset-add-mset-right* [*simp*]:
A + add-mset a B = add-mset a (A + B)
⟨*proof*⟩

lemma *add-mset-add-single*: ‹*add-mset a A = A + {#a#}*›
⟨*proof*⟩

57.3.3 Difference

instance *multiset* :: (*type*) *comm-monoid-diff*
⟨*proof*⟩

lemma *count-diff* [*simp*]:
count (M − N) a = count M a − count N a
⟨*proof*⟩

lemma *add-mset-diff-bothsides*:
‹*add-mset a M − add-mset a A = M − A*›

⟨*proof*⟩

lemma *in-diff-count*:
$a \in\# M - N \longleftrightarrow count\ N\ a < count\ M\ a$
⟨*proof*⟩

lemma *count-in-diffI*:
assumes $\bigwedge n.\ count\ N\ x = n + count\ M\ x \Longrightarrow False$
shows $x \in\# M - N$
⟨*proof*⟩

lemma *in-diff-countE*:
assumes $x \in\# M - N$
obtains n **where** $count\ M\ x = Suc\ n + count\ N\ x$
⟨*proof*⟩

lemma *in-diffD*:
assumes $a \in\# M - N$
shows $a \in\# M$
⟨*proof*⟩

lemma *set-mset-diff*:
$set\text{-}mset\ (M - N) = \{a.\ count\ N\ a < count\ M\ a\}$
⟨*proof*⟩

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
⟨*proof*⟩

lemma *diff-cancel*: $A - A = \{\#\}$
⟨*proof*⟩

lemma *diff-union-cancelR*: $M + N - N = (M::'a\ multiset)$
⟨*proof*⟩

lemma *diff-union-cancelL*: $N + M - N = (M::'a\ multiset)$
⟨*proof*⟩

lemma *diff-right-commute*:
fixes $M\ N\ Q :: 'a\ multiset$
shows $M - N - Q = M - Q - N$
⟨*proof*⟩

lemma *diff-add*:
fixes $M\ N\ Q :: 'a\ multiset$
shows $M - (N + Q) = M - N - Q$
⟨*proof*⟩

lemma *insert-DiffM* [*simp*]: $x \in\# M \Longrightarrow add\text{-}mset\ x\ (M - \{\#x\#\}) = M$
⟨*proof*⟩

lemma *insert-DiffM2*: $x \in\# M \Longrightarrow (M - \{\#x\#\}) + \{\#x\#\} = M$
⟨*proof*⟩

lemma *diff-union-swap*: $a \neq b \Longrightarrow$ *add-mset* b $(M - \{\#a\#\}) =$ *add-mset* b $M - \{\#a\#\}$
⟨*proof*⟩

lemma *diff-add-mset-swap* [*simp*]: $b \notin\# A \Longrightarrow$ *add-mset* b $M - A =$ *add-mset* b $(M - A)$
⟨*proof*⟩

lemma *diff-union-swap2* [*simp*]: $y \in\# M \Longrightarrow$ *add-mset* x $M - \{\#y\#\} =$ *add-mset* x $(M - \{\#y\#\})$
⟨*proof*⟩

lemma *diff-diff-add-mset* [*simp*]: $(M::'a$ *multiset*$) - N - P = M - (N + P)$
⟨*proof*⟩

lemma *diff-union-single-conv*:
$a \in\# J \Longrightarrow I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
⟨*proof*⟩

lemma *mset-add* [*elim?*]:
assumes $a \in\# A$
obtains B **where** $A =$ *add-mset* a B
⟨*proof*⟩

lemma *union-iff*:
$a \in\# A + B \longleftrightarrow a \in\# A \vee a \in\# B$
⟨*proof*⟩

57.3.4 Min and Max

abbreviation *Min-mset* :: $'a$::*linorder multiset* ⇒ $'a$ **where**
Min-mset $m \equiv$ *Min* (*set-mset* m)

abbreviation *Max-mset* :: $'a$::*linorder multiset* ⇒ $'a$ **where**
Max-mset $m \equiv$ *Max* (*set-mset* m)

57.3.5 Equality of multisets

lemma *single-eq-single* [*simp*]: $\{\#a\#\} = \{\#b\#\} \longleftrightarrow a = b$
⟨*proof*⟩

lemma *union-eq-empty* [*iff*]: $M + N = \{\#\} \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
⟨*proof*⟩

lemma *empty-eq-union* [*iff*]: $\{\#\} = M + N \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
⟨*proof*⟩

lemma *multi-self-add-other-not-self* [*simp*]: $M = \textit{add-mset}\ x\ M \longleftrightarrow \textit{False}$
⟨*proof*⟩

lemma *add-mset-remove-trivial* [*simp*]: ‹*add-mset x M* − {#*x*#} = *M*›
⟨*proof*⟩

lemma *diff-single-trivial*: $\neg\ x \in\#\ M \Longrightarrow M - \{\#x\#\} = M$
⟨*proof*⟩

lemma *diff-single-eq-union*: $x \in\#\ M \Longrightarrow M - \{\#x\#\} = N \longleftrightarrow M = \textit{add-mset}\ x\ N$
⟨*proof*⟩

lemma *union-single-eq-diff*: $\textit{add-mset}\ x\ M = N \Longrightarrow M = N - \{\#x\#\}$
⟨*proof*⟩

lemma *union-single-eq-member*: $\textit{add-mset}\ x\ M = N \Longrightarrow x \in\#\ N$
⟨*proof*⟩

lemma *add-mset-remove-trivial-If*:
$\textit{add-mset}\ a\ (N - \{\#a\#\}) = (\textit{if}\ a \in\#\ N\ \textit{then}\ N\ \textit{else}\ \textit{add-mset}\ a\ N)$
⟨*proof*⟩

lemma *add-mset-remove-trivial-eq*: ‹*N* = *add-mset a* (*N* − {#*a*#}) ⟷ *a* ∈# *N*›
⟨*proof*⟩

lemma *union-is-single*:
$M + N = \{\#a\#\} \longleftrightarrow M = \{\#a\#\} \wedge N = \{\#\} \vee M = \{\#\} \wedge N = \{\#a\#\}$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *single-is-union*: $\{\#a\#\} = M + N \longleftrightarrow \{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N$
⟨*proof*⟩

lemma *add-eq-conv-diff*:
$\textit{add-mset}\ a\ M = \textit{add-mset}\ b\ N \longleftrightarrow M = N \wedge a = b \vee M = \textit{add-mset}\ b\ (N - \{\#a\#\}) \wedge N = \textit{add-mset}\ a\ (M - \{\#b\#\})$
(**is** *?lhs* ⟷ *?rhs*)

⟨*proof*⟩

lemma *add-mset-eq-single* [*iff*]: $\textit{add-mset}\ b\ M = \{\#a\#\} \longleftrightarrow b = a \wedge M = \{\#\}$
⟨*proof*⟩

lemma *single-eq-add-mset* [*iff*]: $\{\#a\#\} = \textit{add-mset}\ b\ M \longleftrightarrow b = a \wedge M = \{\#\}$
⟨*proof*⟩

lemma *insert-noteq-member*:
assumes *BC*: *add-mset b B = add-mset c C*
and *bnotc*: *b ≠ c*
shows *c ∈# B*
⟨*proof*⟩

lemma *add-eq-conv-ex*:
(*add-mset a M = add-mset b N*) =
(*M = N ∧ a = b ∨ (∃ K. M = add-mset b K ∧ N = add-mset a K*))
⟨*proof*⟩

lemma *multi-member-split*: *x ∈# M ⟹ ∃ A. M = add-mset x A*
⟨*proof*⟩

lemma *multiset-add-sub-el-shuffle*:
assumes *c ∈# B*
and *b ≠ c*
shows *add-mset b (B − {#c#}) = add-mset b B − {#c#}*
⟨*proof*⟩

lemma *add-mset-eq-singleton-iff*[*iff*]:
add-mset x M = {#y#} ⟷ M = {#} ∧ x = y
⟨*proof*⟩

57.3.6 Pointwise ordering induced by count

definition *subseteq-mset* :: *'a multiset ⇒ 'a multiset ⇒ bool* (**infix** *⊆# 50*)
where *A ⊆# B ⟷ (∀ a. count A a ≤ count B a)*

definition *subset-mset* :: *'a multiset ⇒ 'a multiset ⇒ bool* (**infix** *⊂# 50*)
where *A ⊂# B ⟷ A ⊆# B ∧ A ≠ B*

abbreviation (*input*) *supseteq-mset* :: *'a multiset ⇒ 'a multiset ⇒ bool* (**infix** *⊇# 50*)
where *supseteq-mset A B ≡ B ⊆# A*

abbreviation (*input*) *supset-mset* :: *'a multiset ⇒ 'a multiset ⇒ bool* (**infix** *⊃# 50*)
where *supset-mset A B ≡ B ⊂# A*

notation (*input*)
subseteq-mset (**infix** *≤# 50*) **and**
supseteq-mset (**infix** *≥# 50*)

notation (*ASCII*)
subseteq-mset (**infix** *<=# 50*) **and**
subset-mset (**infix** *<# 50*) **and**
supseteq-mset (**infix** *>=# 50*) **and**

supset-mset (**infix** $>\#$ *50*)

interpretation *subset-mset*: *ordered-ab-semigroup-add-imp-le op + op − op* $\subseteq\#$ *op* $\subset\#$
⟨*proof*⟩

interpretation *subset-mset*: *ordered-ab-semigroup-monoid-add-imp-le op + 0 op − op* $\subseteq\#$ *op* $\subset\#$
⟨*proof*⟩

lemma *mset-subset-eqI*:
$(\bigwedge a.\ count\ A\ a \leq count\ B\ a) \Longrightarrow A \subseteq\# B$
⟨*proof*⟩

lemma *mset-subset-eq-count*:
$A \subseteq\# B \Longrightarrow count\ A\ a \leq count\ B\ a$
⟨*proof*⟩

lemma *mset-subset-eq-exists-conv*: $(A::'a\ multiset) \subseteq\# B \longleftrightarrow (\exists C.\ B = A + C)$
⟨*proof*⟩

interpretation *subset-mset*: *ordered-cancel-comm-monoid-diff op + 0 op* $\subseteq\#$ *op* $\subset\#$ *op −*
⟨*proof*⟩

declare *subset-mset.add-diff-assoc*[*simp*] *subset-mset.add-diff-assoc2*[*simp*]

lemma *mset-subset-eq-mono-add-right-cancel*: $(A::'a\ multiset) + C \subseteq\# B + C \longleftrightarrow A \subseteq\# B$
⟨*proof*⟩

lemma *mset-subset-eq-mono-add-left-cancel*: $C + (A::'a\ multiset) \subseteq\# C + B \longleftrightarrow A \subseteq\# B$
⟨*proof*⟩

lemma *mset-subset-eq-mono-add*: $(A::'a\ multiset) \subseteq\# B \Longrightarrow C \subseteq\# D \Longrightarrow A + C \subseteq\# B + D$
⟨*proof*⟩

lemma *mset-subset-eq-add-left*: $(A::'a\ multiset) \subseteq\# A + B$
⟨*proof*⟩

lemma *mset-subset-eq-add-right*: $B \subseteq\# (A::'a\ multiset) + B$
⟨*proof*⟩

lemma *single-subset-iff* [*simp*]:
$\{\#a\#\} \subseteq\# M \longleftrightarrow a \in\# M$
⟨*proof*⟩

lemma *mset-subset-eq-single*: $a \in\# B \Longrightarrow \{\#a\#\} \subseteq\# B$
⟨*proof*⟩

lemma *mset-subset-eq-add-mset-cancel*: ‹*add-mset a A* $\subseteq\#$ *add-mset a B* $\longleftrightarrow$ *A* $\subseteq\#$ *B*›
⟨*proof*⟩

lemma *multiset-diff-union-assoc*:
fixes *A B C D* :: *'a multiset*
shows $C \subseteq\# B \Longrightarrow A + B - C = A + (B - C)$
⟨*proof*⟩

lemma *mset-subset-eq-multiset-union-diff-commute*:
fixes *A B C D* :: *'a multiset*
shows $B \subseteq\# A \Longrightarrow A - B + C = A + C - B$
⟨*proof*⟩

lemma *diff-subset-eq-self*[*simp*]:
(*M*::*'a multiset*) − *N* $\subseteq\#$ *M*
⟨*proof*⟩

lemma *mset-subset-eqD*:
assumes $A \subseteq\# B$ **and** $x \in\# A$
shows $x \in\# B$
⟨*proof*⟩

lemma *mset-subsetD*:
$A \subset\# B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
⟨*proof*⟩

lemma *set-mset-mono*:
$A \subseteq\# B \Longrightarrow$ *set-mset A* $\subseteq$ *set-mset B*
⟨*proof*⟩

lemma *mset-subset-eq-insertD*:
add-mset x A $\subseteq\# B \Longrightarrow x \in\# B \wedge A \subset\# B$
⟨*proof*⟩

lemma *mset-subset-insertD*:
add-mset x A $\subset\# B \Longrightarrow x \in\# B \wedge A \subset\# B$
⟨*proof*⟩

lemma *mset-subset-of-empty*[*simp*]: $A \subset\# \{\#\} \longleftrightarrow$ *False*
⟨*proof*⟩

lemma *empty-subset-add-mset*[*simp*]: $\{\#\} \subset\#$ *add-mset x M*
⟨*proof*⟩

lemma *empty-le*: $\{\#\} \subseteq\# A$

⟨*proof*⟩

lemma *insert-subset-eq-iff*:
add-mset a A ⊆# *B* ⟷ *a* ∈# *B* ∧ *A* ⊆# *B* − {#*a*#}
⟨*proof*⟩

lemma *insert-union-subset-iff*:
add-mset a A ⊂# *B* ⟷ *a* ∈# *B* ∧ *A* ⊂# *B* − {#*a*#}
⟨*proof*⟩

lemma *subset-eq-diff-conv*:
A − *C* ⊆# *B* ⟷ *A* ⊆# *B* + *C*
⟨*proof*⟩

lemma *multi-psub-of-add-self* [*simp*]: *A* ⊂# *add-mset x A*
⟨*proof*⟩

lemma *multi-psub-self*: *A* ⊂# *A* = *False*
⟨*proof*⟩

lemma *mset-subset-add-mset* [*simp*]: *add-mset x N* ⊂# *add-mset x M* ⟷ *N* ⊂# *M*
⟨*proof*⟩

lemma *mset-subset-diff-self*: *c* ∈# *B* ⟹ *B* − {#*c*#} ⊂# *B*
⟨*proof*⟩

lemma *Diff-eq-empty-iff-mset*: *A* − *B* = {#} ⟷ *A* ⊆# *B*
⟨*proof*⟩

lemma *add-mset-subseteq-single-iff* [*iff*]: *add-mset a M* ⊆# {#*b*#} ⟷ *M* = {#} ∧ *a* = *b*
⟨*proof*⟩

57.3.7 Intersection and bounded union

definition *inf-subset-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* (**infixl** ∩# *70*) **where**
multiset-inter-def: *inf-subset-mset A B* = *A* − (*A* − *B*)

interpretation *subset-mset*: *semilattice-inf inf-subset-mset op* ⊆# *op* ⊂#
⟨*proof*⟩

definition *sup-subset-mset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset*(**infixl** ∪# *70*)
where *sup-subset-mset A B* = *A* + (*B* − *A*) — FIXME irregular fact name

interpretation *subset-mset*: *semilattice-sup sup-subset-mset op* ⊆# *op* ⊂#
⟨*proof*⟩

interpretation *subset-mset*: *bounded-lattice-bot op* $\cap\#$ *op* $\subseteq\#$ *op* $\subset\#$ *op* $\cup\#$ $\{\#\}$
⟨*proof*⟩

57.3.8 Additional intersection facts

lemma *multiset-inter-count* [*simp*]:
fixes A B :: $'a$ *multiset*
shows *count* $(A \cap\# B)$ x = *min* (*count* A x) (*count* B x)
⟨*proof*⟩

lemma *set-mset-inter* [*simp*]:
set-mset $(A \cap\# B)$ = *set-mset* $A \cap$ *set-mset* B
⟨*proof*⟩

lemma *diff-intersect-left-idem* [*simp*]:
$M - M \cap\# N = M - N$
⟨*proof*⟩

lemma *diff-intersect-right-idem* [*simp*]:
$M - N \cap\# M = M - N$
⟨*proof*⟩

lemma *multiset-inter-single*[*simp*]: $a \neq b \Longrightarrow \{\#a\#\} \cap\# \{\#b\#\} = \{\#\}$
⟨*proof*⟩

lemma *multiset-union-diff-commute*:
assumes $B \cap\# C = \{\#\}$
shows $A + B - C = A - C + B$
⟨*proof*⟩

lemma *disjunct-not-in*:
$A \cap\# B = \{\#\} \longleftrightarrow (\forall a.\ a \notin\# A \vee a \notin\# B)$ (**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

lemma *inter-mset-empty-distrib-right*: $A \cap\# (B + C) = \{\#\} \longleftrightarrow A \cap\# B = \{\#\} \wedge A \cap\# C = \{\#\}$
⟨*proof*⟩

lemma *inter-mset-empty-distrib-left*: $(A + B) \cap\# C = \{\#\} \longleftrightarrow A \cap\# C = \{\#\} \wedge B \cap\# C = \{\#\}$
⟨*proof*⟩

lemma *add-mset-inter-add-mset*[*simp*]:
add-mset a $A \cap\#$ *add-mset* a B = *add-mset* a $(A \cap\# B)$
⟨*proof*⟩

lemma *add-mset-disjoint* [*simp*]:

add-mset a A $\cap\#$ *B* = {#} $\longleftrightarrow$ *a* $\notin\#$ *B* $\wedge$ *A* $\cap\#$ *B* = {#}
{#} = *add-mset a A* $\cap\#$ *B* $\longleftrightarrow$ *a* $\notin\#$ *B* $\wedge$ {#} = *A* $\cap\#$ *B*
⟨*proof*⟩

lemma *disjoint-add-mset* [*simp*]:
B $\cap\#$ *add-mset a A* = {#} $\longleftrightarrow$ *a* $\notin\#$ *B* $\wedge$ *B* $\cap\#$ *A* = {#}
{#} = *A* $\cap\#$ *add-mset b B* $\longleftrightarrow$ *b* $\notin\#$ *A* $\wedge$ {#} = *A* $\cap\#$ *B*
⟨*proof*⟩

lemma *inter-add-left1*: $\neg$ *x* $\in\#$ *N* $\Longrightarrow$ (*add-mset x M*) $\cap\#$ *N* = *M* $\cap\#$ *N*
⟨*proof*⟩

lemma *inter-add-left2*: *x* $\in\#$ *N* $\Longrightarrow$ (*add-mset x M*) $\cap\#$ *N* = *add-mset x* (*M* $\cap\#$ (*N* − {#*x*#}))
⟨*proof*⟩

lemma *inter-add-right1*: $\neg$ *x* $\in\#$ *N* $\Longrightarrow$ *N* $\cap\#$ (*add-mset x M*) = *N* $\cap\#$ *M*
⟨*proof*⟩

lemma *inter-add-right2*: *x* $\in\#$ *N* $\Longrightarrow$ *N* $\cap\#$ (*add-mset x M*) = *add-mset x* ((*N* − {#*x*#}) $\cap\#$ *M*)
⟨*proof*⟩

lemma *disjunct-set-mset-diff*:
assumes *M* $\cap\#$ *N* = {#}
shows *set-mset* (*M* − *N*) = *set-mset M*
⟨*proof*⟩

lemma *at-most-one-mset-mset-diff*:
assumes *a* $\notin\#$ *M* − {#*a*#}
shows *set-mset* (*M* − {#*a*#}) = *set-mset M* − {*a*}
⟨*proof*⟩

lemma *more-than-one-mset-mset-diff*:
assumes *a* $\in\#$ *M* − {#*a*#}
shows *set-mset* (*M* − {#*a*#}) = *set-mset M*
⟨*proof*⟩

lemma *inter-iff*:
a $\in\#$ *A* $\cap\#$ *B* $\longleftrightarrow$ *a* $\in\#$ *A* $\wedge$ *a* $\in\#$ *B*
⟨*proof*⟩

lemma *inter-union-distrib-left*:
A $\cap\#$ *B* + *C* = (*A* + *C*) $\cap\#$ (*B* + *C*)
⟨*proof*⟩

lemma *inter-union-distrib-right*:
C + *A* $\cap\#$ *B* = (*C* + *A*) $\cap\#$ (*C* + *B*)
⟨*proof*⟩

lemma *inter-subset-eq-union*:
$A \cap\# B \subseteq\# A + B$
⟨*proof*⟩

57.3.9 Additional bounded union facts

lemma *sup-subset-mset-count* [*simp*]: — FIXME irregular fact name
count $(A \cup\# B)$ x = *max* (*count* A x) (*count* B x)
⟨*proof*⟩

lemma *set-mset-sup* [*simp*]:
set-mset $(A \cup\# B)$ = *set-mset* $A \cup$ *set-mset* B
⟨*proof*⟩

lemma *sup-union-left1* [*simp*]: $\neg x \in\# N \Longrightarrow$ (*add-mset* x M) $\cup\# N$ = *add-mset* x $(M \cup\# N)$
⟨*proof*⟩

lemma *sup-union-left2*: $x \in\# N \Longrightarrow$ (*add-mset* x M) $\cup\# N$ = *add-mset* x $(M \cup\# (N - \{\#x\#\}))$
⟨*proof*⟩

lemma *sup-union-right1* [*simp*]: $\neg x \in\# N \Longrightarrow N \cup\#$ (*add-mset* x M) = *add-mset* x $(N \cup\# M)$
⟨*proof*⟩

lemma *sup-union-right2*: $x \in\# N \Longrightarrow N \cup\#$ (*add-mset* x M) = *add-mset* x $((N - \{\#x\#\}) \cup\# M)$
⟨*proof*⟩

lemma *sup-union-distrib-left*:
$A \cup\# B + C = (A + C) \cup\# (B + C)$
⟨*proof*⟩

lemma *union-sup-distrib-right*:
$C + A \cup\# B = (C + A) \cup\# (C + B)$
⟨*proof*⟩

lemma *union-diff-inter-eq-sup*:
$A + B - A \cap\# B = A \cup\# B$
⟨*proof*⟩

lemma *union-diff-sup-eq-inter*:
$A + B - A \cup\# B = A \cap\# B$
⟨*proof*⟩

lemma *add-mset-union*:
‹*add-mset* a $A \cup\#$ *add-mset* a B = *add-mset* a $(A \cup\# B)$›

⟨*proof*⟩

57.4 Replicate and repeat operations

definition *replicate-mset* :: *nat* ⇒ *'a* ⇒ *'a multiset* **where**
replicate-mset n x = (*add-mset x* ^^ *n*) {#}

lemma *replicate-mset-0*[*simp*]: *replicate-mset 0 x* = {#}
⟨*proof*⟩

lemma *replicate-mset-Suc* [*simp*]: *replicate-mset* (*Suc n*) *x* = *add-mset x* (*replicate-mset n x*)
⟨*proof*⟩

lemma *count-replicate-mset*[*simp*]: *count* (*replicate-mset n x*) *y* = (*if y* = *x then n else 0*)
⟨*proof*⟩

fun *repeat-mset* :: *nat* ⇒ *'a multiset* ⇒ *'a multiset* **where**
repeat-mset 0 - = {#} |
repeat-mset (*Suc n*) *A* = *A* + *repeat-mset n A*

lemma *count-repeat-mset* [*simp*]: *count* (*repeat-mset i A*) *a* = *i* ∗ *count A a*
⟨*proof*⟩

lemma *repeat-mset-right* [*simp*]: *repeat-mset a* (*repeat-mset b A*) = *repeat-mset* (*a* ∗ *b*) *A*
⟨*proof*⟩

lemma *left-diff-repeat-mset-distrib'*: ‹*repeat-mset* (*i* − *j*) *u* = *repeat-mset i u* − *repeat-mset j u*›
⟨*proof*⟩

lemma *left-add-mult-distrib-mset*:
repeat-mset i u + (*repeat-mset j u* + *k*) = *repeat-mset* (*i*+*j*) *u* + *k*
⟨*proof*⟩

lemma *repeat-mset-distrib*:
repeat-mset (*m* + *n*) *A* = *repeat-mset m A* + *repeat-mset n A*
⟨*proof*⟩

lemma *repeat-mset-distrib2*[*simp*]:
repeat-mset n (*A* + *B*) = *repeat-mset n A* + *repeat-mset n B*
⟨*proof*⟩

lemma *repeat-mset-replicate-mset*[*simp*]:
repeat-mset n {#*a*#} = *replicate-mset n a*
⟨*proof*⟩

lemma *repeat-mset-distrib-add-mset*[*simp*]:
repeat-mset n (*add-mset a A*) = *replicate-mset n a* + *repeat-mset n A*
⟨*proof*⟩

lemma *repeat-mset-empty*[*simp*]: *repeat-mset n* {#} = {#}
⟨*proof*⟩

57.4.1 Simprocs

lemma *repeat-mset-iterate-add*: ‹*repeat-mset n M* = *iterate-add n M*›
⟨*proof*⟩

lemma *mset-subseteq-add-iff1*:
j ≤ (*i*::*nat*) ⟹ (*repeat-mset i u* + *m* ⊆# *repeat-mset j u* + *n*) = (*repeat-mset* (*i*−*j*) *u* + *m* ⊆# *n*)
⟨*proof*⟩

lemma *mset-subseteq-add-iff2*:
i ≤ (*j*::*nat*) ⟹ (*repeat-mset i u* + *m* ⊆# *repeat-mset j u* + *n*) = (*m* ⊆# *repeat-mset* (*j*−*i*) *u* + *n*)
⟨*proof*⟩

lemma *mset-subset-add-iff1*:
j ≤ (*i*::*nat*) ⟹ (*repeat-mset i u* + *m* ⊂# *repeat-mset j u* + *n*) = (*repeat-mset* (*i*−*j*) *u* + *m* ⊂# *n*)
⟨*proof*⟩

lemma *mset-subset-add-iff2*:
i ≤ (*j*::*nat*) ⟹ (*repeat-mset i u* + *m* ⊂# *repeat-mset j u* + *n*) = (*m* ⊂# *repeat-mset* (*j*−*i*) *u* + *n*)
⟨*proof*⟩

⟨*ML*⟩

lemma *add-mset-replicate-mset-safe*[*cancelation-simproc-pre*]: ‹*NO-MATCH* {#} *M* ⟹ *add-mset a M* = {#*a*#} + *M*›
⟨*proof*⟩

declare *repeat-mset-iterate-add*[*cancelation-simproc-pre*]

declare *iterate-add-distrib*[*cancelation-simproc-pre*]
declare *repeat-mset-iterate-add*[*symmetric*, *cancelation-simproc-post*]

declare *add-mset-not-empty*[*cancelation-simproc-eq-elim*]
empty-not-add-mset[*cancelation-simproc-eq-elim*]
subset-mset.le-zero-eq[*cancelation-simproc-eq-elim*]
empty-not-add-mset[*cancelation-simproc-eq-elim*]
add-mset-not-empty[*cancelation-simproc-eq-elim*]
subset-mset.le-zero-eq[*cancelation-simproc-eq-elim*]

le-zero-eq[*cancelation-simproc-eq-elim*]

⟨*ML*⟩

57.4.2 Conditionally complete lattice

instantiation *multiset* :: (*type*) *Inf*
begin

lift-definition *Inf-multiset* :: *'a multiset set* ⇒ *'a multiset* **is**
λ*A i*. *if A* = {} *then 0 else Inf* ((λ*f*. *f i*) ‘ *A*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *Inf-multiset-empty*: *Inf* {} = {#}
⟨*proof*⟩

lemma *count-Inf-multiset-nonempty*: *A* ≠ {} ⟹ *count* (*Inf A*) *x* = *Inf* ((λ*X*. *count X x*) ‘ *A*)
⟨*proof*⟩

instantiation *multiset* :: (*type*) *Sup*
begin

definition *Sup-multiset* :: *'a multiset set* ⇒ *'a multiset* **where**
Sup-multiset A = (*if A* ≠ {} ∧ *subset-mset.bdd-above A then*
Abs-multiset (λ*i*. *Sup* ((λ*X*. *count X i*) ‘ *A*)) *else* {#})

lemma *Sup-multiset-empty*: *Sup* {} = {#}
⟨*proof*⟩

lemma *Sup-multiset-unbounded*: ¬*subset-mset.bdd-above A* ⟹ *Sup A* = {#}
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *bdd-above-multiset-imp-bdd-above-count*:
assumes *subset-mset.bdd-above* (*A* :: *'a multiset set*)
shows *bdd-above* ((λ*X*. *count X x*) ‘ *A*)
⟨*proof*⟩

lemma *bdd-above-multiset-imp-finite-support*:

assumes $A \neq \{\}$ *subset-mset.bdd-above* $(A :: 'a\ multiset\ set)$
shows *finite* $(\bigcup X{\in}A.\ \{x.\ count\ X\ x > 0\})$
⟨*proof*⟩

lemma *Sup-multiset-in-multiset*:
assumes $A \neq \{\}$ *subset-mset.bdd-above* A
shows $(\lambda i.\ SUP\ X{:}A.\ count\ X\ i) \in multiset$
⟨*proof*⟩

lemma *count-Sup-multiset-nonempty*:
assumes $A \neq \{\}$ *subset-mset.bdd-above* A
shows $count\ (Sup\ A)\ x = (SUP\ X{:}A.\ count\ X\ x)$
⟨*proof*⟩

interpretation *subset-mset*: *conditionally-complete-lattice Inf Sup op* $\cap\#$ *op* $\subseteq\#$ *op* $\subset\#$ *op* $\cup\#$
⟨*proof*⟩

lemma *set-mset-Inf*:
assumes $A \neq \{\}$
shows $set\text{-}mset\ (Inf\ A) = (\bigcap X{\in}A.\ set\text{-}mset\ X)$
⟨*proof*⟩

lemma *in-Inf-multiset-iff*:
assumes $A \neq \{\}$
shows $x \in\# Inf\ A \longleftrightarrow (\forall X{\in}A.\ x \in\# X)$
⟨*proof*⟩

lemma *in-Inf-multisetD*: $x \in\# Inf\ A \Longrightarrow X \in A \Longrightarrow x \in\# X$
⟨*proof*⟩

lemma *set-mset-Sup*:
assumes *subset-mset.bdd-above* A
shows $set\text{-}mset\ (Sup\ A) = (\bigcup X{\in}A.\ set\text{-}mset\ X)$
⟨*proof*⟩

lemma *in-Sup-multiset-iff*:
assumes *subset-mset.bdd-above* A
shows $x \in\# Sup\ A \longleftrightarrow (\exists X{\in}A.\ x \in\# X)$
⟨*proof*⟩

lemma *in-Sup-multisetD*:
assumes $x \in\# Sup\ A$
shows $\exists X{\in}A.\ x \in\# X$
⟨*proof*⟩

interpretation *subset-mset*: *distrib-lattice op* $\cap\#$ *op* $\subseteq\#$ *op* $\subset\#$ *op* $\cup\#$
⟨*proof*⟩

57.4.3 Filter (with comprehension syntax)

Multiset comprehension

lift-definition *filter-mset* :: $('a \Rightarrow bool) \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$
is $\lambda P\ M.\ \lambda x.\ if\ P\ x\ then\ M\ x\ else\ 0$
⟨*proof*⟩

syntax (*ASCII*)
-MCollect :: *pttrn* ⇒ *'a multiset* ⇒ *bool* ⇒ *'a multiset* ((*1*{#- :# -./ -#}))
syntax
-MCollect :: *pttrn* ⇒ *'a multiset* ⇒ *bool* ⇒ *'a multiset* ((*1*{#- ∈# -./ -#}))
translations
{#*x* ∈# *M*. *P*#} == *CONST filter-mset* (λ*x*. *P*) *M*

lemma *count-filter-mset* [*simp*]:
count (*filter-mset P M*) *a* = (*if P a then count M a else 0*)
⟨*proof*⟩

lemma *set-mset-filter* [*simp*]:
set-mset (*filter-mset P M*) = {*a* ∈ *set-mset M*. *P a*}
⟨*proof*⟩

lemma *filter-empty-mset* [*simp*]: *filter-mset P* {#} = {#}
⟨*proof*⟩

lemma *filter-single-mset*: *filter-mset P* {#*x*#} = (*if P x then* {#*x*#} *else* {#})
⟨*proof*⟩

lemma *filter-union-mset* [*simp*]: *filter-mset P* (*M* + *N*) = *filter-mset P M* + *filter-mset P N*
⟨*proof*⟩

lemma *filter-diff-mset* [*simp*]: *filter-mset P* (*M* − *N*) = *filter-mset P M* − *filter-mset P N*
⟨*proof*⟩

lemma *filter-inter-mset* [*simp*]: *filter-mset P* (*M* ∩# *N*) = *filter-mset P M* ∩# *filter-mset P N*
⟨*proof*⟩

lemma *filter-sup-mset*[*simp*]: *filter-mset P* (*A* ∪# *B*) = *filter-mset P A* ∪# *filter-mset P B*
⟨*proof*⟩

lemma *filter-mset-add-mset* [*simp*]:
filter-mset P (*add-mset x A*) =
(*if P x then add-mset x* (*filter-mset P A*) *else filter-mset P A*)
⟨*proof*⟩

lemma *multiset-filter-subset*[*simp*]: *filter-mset f M* $\subseteq\#$ *M*
⟨*proof*⟩

lemma *multiset-filter-mono*:
assumes $A \subseteq\# B$
shows *filter-mset f A* $\subseteq\#$ *filter-mset f B*
⟨*proof*⟩

lemma *filter-mset-eq-conv*:
filter-mset P M = *N* $\longleftrightarrow$ *N* $\subseteq\#$ *M* $\wedge$ ($\forall$ *b*$\in\#$*N*. *P b*) $\wedge$ ($\forall$ *a*$\in\#$*M* − *N*. $\neg$ *P a*)
(**is** *?P* $\longleftrightarrow$ *?Q*)
⟨*proof*⟩

lemma *filter-filter-mset*: *filter-mset P* (*filter-mset Q M*) = {#*x* $\in\#$ *M*. *Q x* $\wedge$ *P x*#}
⟨*proof*⟩

lemma
filter-mset-True[*simp*]: {#*y* $\in\#$ *M*. *True*#} = *M* **and**
filter-mset-False[*simp*]: {#*y* $\in\#$ *M*. *False*#} = {#}
⟨*proof*⟩

57.4.4 Size

definition *wcount* **where** *wcount f M* = (λ*x*. *count M x* ∗ *Suc* (*f x*))

lemma *wcount-union*: *wcount f* (*M* + *N*) *a* = *wcount f M a* + *wcount f N a*
⟨*proof*⟩

lemma *wcount-add-mset*:
wcount f (*add-mset x M*) *a* = (*if x* = *a then Suc* (*f a*) *else 0*) + *wcount f M a*
⟨*proof*⟩

definition *size-multiset* :: (′*a* ⇒ *nat*) ⇒ ′*a multiset* ⇒ *nat* **where**
size-multiset f M = *sum* (*wcount f M*) (*set-mset M*)

lemmas *size-multiset-eq* = *size-multiset-def*[*unfolded wcount-def*]

instantiation *multiset* :: (*type*) *size*
begin

definition *size-multiset* **where**
size-multiset-overloaded-def: *size-multiset* = *Multiset.size-multiset* (λ-. *0*)
instance ⟨*proof*⟩

end

lemmas *size-multiset-overloaded-eq* =
size-multiset-overloaded-def[*THEN fun-cong*, *unfolded size-multiset-eq*, *simplified*]

lemma *size-multiset-empty* [*simp*]: *size-multiset f* {#} = *0*
⟨*proof*⟩

lemma *size-empty* [*simp*]: *size* {#} = *0*
⟨*proof*⟩

lemma *size-multiset-single* : *size-multiset f* {#*b*#} = *Suc* (*f b*)
⟨*proof*⟩

lemma *size-single*: *size* {#*b*#} = *1*
⟨*proof*⟩

lemma *sum-wcount-Int*:
finite A ⟹ *sum* (*wcount f N*) (*A* ∩ *set-mset N*) = *sum* (*wcount f N*) *A*
⟨*proof*⟩

lemma *size-multiset-union* [*simp*]:
size-multiset f (*M* + *N*::'*a multiset*) = *size-multiset f M* + *size-multiset f N*
⟨*proof*⟩

lemma *size-multiset-add-mset* [*simp*]:
size-multiset f (*add-mset a M*) = *Suc* (*f a*) + *size-multiset f M*
⟨*proof*⟩

lemma *size-add-mset* [*simp*]: *size* (*add-mset a A*) = *Suc* (*size A*)
⟨*proof*⟩

lemma *size-union* [*simp*]: *size* (*M* + *N*::'*a multiset*) = *size M* + *size N*
⟨*proof*⟩

lemma *size-multiset-eq-0-iff-empty* [*iff*]:
size-multiset f M = *0* ⟷ *M* = {#}
⟨*proof*⟩

lemma *size-eq-0-iff-empty* [*iff*]: (*size M* = *0*) = (*M* = {#})
⟨*proof*⟩

lemma *nonempty-has-size*: (*S* ≠ {#}) = (*0* < *size S*)
⟨*proof*⟩

lemma *size-eq-Suc-imp-elem*: *size M* = *Suc n* ⟹ ∃ *a*. *a* ∈# *M*
⟨*proof*⟩

lemma *size-eq-Suc-imp-eq-union*:
assumes *size M* = *Suc n*
shows ∃ *a N*. *M* = *add-mset a N*
⟨*proof*⟩

lemma *size-mset-mono*:
fixes $A\ B :: 'a\ multiset$
assumes $A \subseteq\# B$
shows $size\ A \leq size\ B$
⟨*proof*⟩

lemma *size-filter-mset-lesseq*[*simp*]: $size\ (filter\text{-}mset\ f\ M) \leq size\ M$
⟨*proof*⟩

lemma *size-Diff-submset*:
$M \subseteq\# M' \Longrightarrow size\ (M' - M) = size\ M' - size(M::'a\ multiset)$
⟨*proof*⟩

57.5 Induction and case splits

theorem *multiset-induct* [*case-names empty add*, *induct type*: *multiset*]:
assumes *empty*: $P\ \{\#\}$
assumes *add*: $\bigwedge x\ M.\ P\ M \Longrightarrow P\ (add\text{-}mset\ x\ M)$
shows $P\ M$
⟨*proof*⟩

lemma *multiset-induct-min*[*case-names empty add*]:
fixes $M :: 'a::linorder\ multiset$
assumes
empty: $P\ \{\#\}$ **and**
add: $\bigwedge x\ M.\ P\ M \Longrightarrow (\forall y \in\# M.\ y \geq x) \Longrightarrow P\ (add\text{-}mset\ x\ M)$
shows $P\ M$
⟨*proof*⟩

lemma *multiset-induct-max*[*case-names empty add*]:
fixes $M :: 'a::linorder\ multiset$
assumes
empty: $P\ \{\#\}$ **and**
add: $\bigwedge x\ M.\ P\ M \Longrightarrow (\forall y \in\# M.\ y \leq x) \Longrightarrow P\ (add\text{-}mset\ x\ M)$
shows $P\ M$
⟨*proof*⟩

lemma *multi-nonempty-split*: $M \neq \{\#\} \Longrightarrow \exists A\ a.\ M = add\text{-}mset\ a\ A$
⟨*proof*⟩

lemma *multiset-cases* [*cases type*]:
obtains (*empty*) $M = \{\#\}$
| (*add*) $x\ N$ **where** $M = add\text{-}mset\ x\ N$
⟨*proof*⟩

lemma *multi-drop-mem-not-eq*: $c \in\# B \Longrightarrow B - \{\#c\#\} \neq B$
⟨*proof*⟩

lemma *multiset-partition*: $M = \{\#x \in\# M.\ P\ x\#\} + \{\#x \in\# M.\ \neg\ P\ x\#\}$

⟨*proof*⟩

lemma *mset-subset-size*: $A \subset\# B \Longrightarrow size\ A < size\ B$
⟨*proof*⟩

lemma *size-1-singleton-mset*: $size\ M = 1 \Longrightarrow \exists\, a.\ M = \{\#a\#\}$
⟨*proof*⟩

57.5.1 Strong induction and subset induction for multisets

Well-foundedness of strict subset relation

lemma *wf-subset-mset-rel*: *wf* $\{(M, N :: 'a\ multiset).\ M \subset\# N\}$
⟨*proof*⟩

lemma *full-multiset-induct* [*case-names less*]:
assumes *ih*: $\bigwedge B.\ \forall (A::'a\ multiset).\ A \subset\# B \longrightarrow P\ A \Longrightarrow P\ B$
shows $P\ B$
⟨*proof*⟩

lemma *multi-subset-induct* [*consumes 2*, *case-names empty add*]:
assumes $F \subseteq\# A$
and *empty*: $P\ \{\#\}$
and *insert*: $\bigwedge a\ F.\ a \in\# A \Longrightarrow P\ F \Longrightarrow P\ (add\text{-}mset\ a\ F)$
shows $P\ F$
⟨*proof*⟩

57.6 The fold combinator

definition *fold-mset* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ multiset \Rightarrow 'b$
where
$fold\text{-}mset\ f\ s\ M = Finite\text{-}Set.fold\ (\lambda x.\ f\ x\ \hat{}\hat{}\ count\ M\ x)\ s\ (set\text{-}mset\ M)$

lemma *fold-mset-empty* [*simp*]: $fold\text{-}mset\ f\ s\ \{\#\} = s$
⟨*proof*⟩

context *comp-fun-commute*
begin

lemma *fold-mset-add-mset* [*simp*]: $fold\text{-}mset\ f\ s\ (add\text{-}mset\ x\ M) = f\ x\ (fold\text{-}mset\ f\ s\ M)$
⟨*proof*⟩

corollary *fold-mset-single*: $fold\text{-}mset\ f\ s\ \{\#x\#\} = f\ x\ s$
⟨*proof*⟩

lemma *fold-mset-fun-left-comm*: $f\ x\ (fold\text{-}mset\ f\ s\ M) = fold\text{-}mset\ f\ (f\ x\ s)\ M$
⟨*proof*⟩

lemma *fold-mset-union* [*simp*]: *fold-mset f s* (*M* + *N*) = *fold-mset f* (*fold-mset f s M*) *N*
⟨*proof*⟩

lemma *fold-mset-fusion*:
assumes *comp-fun-commute g*
and ∗: ⋀*x y*. *h* (*g x y*) = *f x* (*h y*)
shows *h* (*fold-mset g w A*) = *fold-mset f* (*h w*) *A*
⟨*proof*⟩

end

lemma *union-fold-mset-add-mset*: *A* + *B* = *fold-mset add-mset A B*
⟨*proof*⟩

A note on code generation: When defining some function containing a subterm *fold-mset F*, code generation is not automatic. When interpreting locale *left-commutative* with *F*, the would be code thms for *fold-mset* become thms like *fold-mset F z* {#} = *z* where *F* is not a pattern but contains defined symbols, i.e. is not a code thm. Hence a separate constant with its own code thms needs to be introduced for *F*. See the image operator below.

57.7 Image

definition *image-mset* :: ('*a* ⇒ '*b*) ⇒ '*a multiset* ⇒ '*b multiset* **where**
image-mset f = *fold-mset* (*add-mset* ∘ *f*) {#}

lemma *comp-fun-commute-mset-image*: *comp-fun-commute* (*add-mset* ∘ *f*)
⟨*proof*⟩

lemma *image-mset-empty* [*simp*]: *image-mset f* {#} = {#}
⟨*proof*⟩

lemma *image-mset-single*: *image-mset f* {#*x*#} = {#*f x*#}
⟨*proof*⟩

lemma *image-mset-union* [*simp*]: *image-mset f* (*M* + *N*) = *image-mset f M* + *image-mset f N*
⟨*proof*⟩

corollary *image-mset-add-mset* [*simp*]:
image-mset f (*add-mset a M*) = *add-mset* (*f a*) (*image-mset f M*)
⟨*proof*⟩

lemma *set-image-mset* [*simp*]: *set-mset* (*image-mset f M*) = *image f* (*set-mset M*)
⟨*proof*⟩

lemma *size-image-mset* [*simp*]: *size* (*image-mset f M*) = *size M*

⟨*proof*⟩

lemma *image-mset-is-empty-iff* [*simp*]: *image-mset f M* = {#} ⟷ *M* = {#}
⟨*proof*⟩

lemma *image-mset-If*:
image-mset (λ*x*. *if P x then f x else g x*) *A* =
image-mset f (*filter-mset P A*) + *image-mset g* (*filter-mset* (λ*x*. ¬*P x*) *A*)
⟨*proof*⟩

lemma *image-mset-Diff*:
assumes *B* ⊆# *A*
shows *image-mset f* (*A* − *B*) = *image-mset f A* − *image-mset f B*
⟨*proof*⟩

lemma *count-image-mset*: *count* (*image-mset f A*) *x* = (∑ *y*∈*f* −‘ {*x*} ∩ *set-mset A*. *count A y*)
⟨*proof*⟩

lemma *image-mset-subseteq-mono*: *A* ⊆# *B* ⟹ *image-mset f A* ⊆# *image-mset f B*
⟨*proof*⟩

lemma *image-mset-subset-mono*: *M* ⊂# *N* ⟹ *image-mset f M* ⊂# *image-mset f N*
⟨*proof*⟩

syntax (*ASCII*)
-*comprehension-mset* :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *'a multiset* (({#-/. - :# -#}))
syntax
-*comprehension-mset* :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *'a multiset* (({#-/. - ∈# -#}))
translations
{#*e*. *x* ∈# *M*#} ⇌ *CONST image-mset* (λ*x*. *e*) *M*

syntax (*ASCII*)
-*comprehension-mset'* :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *bool* ⇒ *'a multiset* (({#-/ | - :# -./ -#}))
syntax
-*comprehension-mset'* :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *bool* ⇒ *'a multiset* (({#-/ | - ∈# -./ -#}))
translations
{#*e* | *x*∈#*M*. *P*#} ↽ {#*e*. *x* ∈# {# *x*∈#*M*. *P*#}#}

This allows to write not just filters like {#*x* ∈# *M*. *x* < *c*#} but also images like {#*x* + *x*. *x* ∈# *M*#} and {#*x*+*x*|*x*∈#*M*. *x*<*c*#}, where the latter is currently displayed as {#*x* + *x*. *x* ∈# {#*x* ∈# *M*. *x* < *c*#}#}.

lemma *in-image-mset*: *y* ∈# {#*f x*. *x* ∈# *M*#} ⟷ *y* ∈ *f* ‘ *set-mset M*
⟨*proof*⟩

functor *image-mset*: *image-mset*
⟨*proof*⟩

declare
image-mset.id [*simp*]
image-mset.identity [*simp*]

lemma *image-mset-id*[*simp*]: *image-mset id x* = *x*
⟨*proof*⟩

lemma *image-mset-cong*: (⋀*x*. *x* ∈# *M* ⟹ *f x* = *g x*) ⟹ {#*f x*. *x* ∈# *M*#}
= {#*g x*. *x* ∈# *M*#}
⟨*proof*⟩

lemma *image-mset-cong-pair*:
(∀ *x y*. (*x*, *y*) ∈# *M* ⟶ *f x y* = *g x y*) ⟹ {#*f x y*. (*x*, *y*) ∈# *M*#} = {#*g x*
y. (*x*, *y*) ∈# *M*#}
⟨*proof*⟩

lemma *image-mset-const-eq*:
{#*c*. *a* ∈# *M*#} = *replicate-mset* (*size M*) *c*
⟨*proof*⟩

57.8 Further conversions

primrec *mset* :: *'a list* ⇒ *'a multiset* **where**
mset [] = {#} |
mset (*a* # *x*) = *add-mset a* (*mset x*)

lemma *in-multiset-in-set*:
x ∈# *mset xs* ⟷ *x* ∈ *set xs*
⟨*proof*⟩

lemma *count-mset*:
count (*mset xs*) *x* = *length* (*filter* (λ*y*. *x* = *y*) *xs*)
⟨*proof*⟩

lemma *mset-zero-iff*[*simp*]: (*mset x* = {#}) = (*x* = [])
⟨*proof*⟩

lemma *mset-zero-iff-right*[*simp*]: ({#} = *mset x*) = (*x* = [])
⟨*proof*⟩

lemma *count-mset-gt-0*: *x* ∈ *set xs* ⟹ *count* (*mset xs*) *x* > *0*
⟨*proof*⟩

lemma *count-mset-0-iff* [*simp*]: *count* (*mset xs*) *x* = *0* ⟷ *x* ∉ *set xs*
⟨*proof*⟩

lemma *mset-single-iff*[*iff*]: *mset xs* = {#*x*#} ⟷ *xs* = [*x*]
⟨*proof*⟩

lemma *mset-single-iff-right*[*iff*]: {#*x*#} = *mset xs* ⟷ *xs* = [*x*]
⟨*proof*⟩

lemma *set-mset-mset*[*simp*]: *set-mset* (*mset xs*) = *set xs*
⟨*proof*⟩

lemma *set-mset-comp-mset* [*simp*]: *set-mset* ∘ *mset* = *set*
⟨*proof*⟩

lemma *size-mset* [*simp*]: *size* (*mset xs*) = *length xs*
⟨*proof*⟩

lemma *mset-append* [*simp*]: *mset* (*xs* @ *ys*) = *mset xs* + *mset ys*
⟨*proof*⟩

lemma *mset-filter*: *mset* (*filter P xs*) = {#*x* ∈# *mset xs*. *P x* #}
⟨*proof*⟩

lemma *mset-rev* [*simp*]:
mset (*rev xs*) = *mset xs*
⟨*proof*⟩

lemma *surj-mset*: *surj mset*
⟨*proof*⟩

lemma *distinct-count-atmost-1*:
distinct x = (∀ *a*. *count* (*mset x*) *a* = (*if a* ∈ *set x then 1 else 0*))
⟨*proof*⟩

lemma *mset-eq-setD*:
assumes *mset xs* = *mset ys*
shows *set xs* = *set ys*
⟨*proof*⟩

lemma *set-eq-iff-mset-eq-distinct*:
distinct x ⟹ *distinct y* ⟹
(*set x* = *set y*) = (*mset x* = *mset y*)
⟨*proof*⟩

lemma *set-eq-iff-mset-remdups-eq*:
(*set x* = *set y*) = (*mset* (*remdups x*) = *mset* (*remdups y*))
⟨*proof*⟩

lemma *mset-compl-union* [*simp*]: *mset* [*x*←*xs*. *P x*] + *mset* [*x*←*xs*. ¬*P x*] = *mset xs*
⟨*proof*⟩

lemma *nth-mem-mset*: $i < \mathit{length}\ \mathit{ls} \Longrightarrow (\mathit{ls}\ !\ i) \in\#\ \mathit{mset}\ \mathit{ls}$
⟨*proof*⟩

lemma *mset-remove1*[*simp*]: $\mathit{mset}\ (\mathit{remove1}\ a\ \mathit{xs}) = \mathit{mset}\ \mathit{xs} - \{\#a\#\}$
⟨*proof*⟩

lemma *mset-eq-length*:
assumes $\mathit{mset}\ \mathit{xs} = \mathit{mset}\ \mathit{ys}$
shows $\mathit{length}\ \mathit{xs} = \mathit{length}\ \mathit{ys}$
⟨*proof*⟩

lemma *mset-eq-length-filter*:
assumes $\mathit{mset}\ \mathit{xs} = \mathit{mset}\ \mathit{ys}$
shows $\mathit{length}\ (\mathit{filter}\ (\lambda x.\ z = x)\ \mathit{xs}) = \mathit{length}\ (\mathit{filter}\ (\lambda y.\ z = y)\ \mathit{ys})$
⟨*proof*⟩

lemma *fold-multiset-equiv*:
assumes f: $\bigwedge x\ y.\ x \in \mathit{set}\ \mathit{xs} \Longrightarrow y \in \mathit{set}\ \mathit{xs} \Longrightarrow f\ x \circ f\ y = f\ y \circ f\ x$
and *equiv*: $\mathit{mset}\ \mathit{xs} = \mathit{mset}\ \mathit{ys}$
shows $\mathit{List.fold}\ f\ \mathit{xs} = \mathit{List.fold}\ f\ \mathit{ys}$
⟨*proof*⟩

lemma *mset-shuffle*: $\mathit{zs} \in \mathit{shuffle}\ \mathit{xs}\ \mathit{ys} \Longrightarrow \mathit{mset}\ \mathit{zs} = \mathit{mset}\ \mathit{xs} + \mathit{mset}\ \mathit{ys}$
⟨*proof*⟩

lemma *mset-insort* [*simp*]: $\mathit{mset}\ (\mathit{insort}\ x\ \mathit{xs}) = \mathit{add\text{-}mset}\ x\ (\mathit{mset}\ \mathit{xs})$
⟨*proof*⟩

lemma *mset-map*[*simp*]: $\mathit{mset}\ (\mathit{map}\ f\ \mathit{xs}) = \mathit{image\text{-}mset}\ f\ (\mathit{mset}\ \mathit{xs})$
⟨*proof*⟩

global-interpretation *mset-set*: *folding add-mset* $\{\#\}$
defines $\mathit{mset\text{-}set} = \mathit{folding.F}\ \mathit{add\text{-}mset}\ \{\#\}$
⟨*proof*⟩

lemma *sum-multiset-singleton* [*simp*]: $\mathit{sum}\ (\lambda n.\ \{\#n\#\})\ A = \mathit{mset\text{-}set}\ A$
⟨*proof*⟩

lemma *count-mset-set* [*simp*]:
$\mathit{finite}\ A \Longrightarrow x \in A \Longrightarrow \mathit{count}\ (\mathit{mset\text{-}set}\ A)\ x = 1\ (\mathbf{is}\ \mathit{PROP}\ ?P)$
$\neg\ \mathit{finite}\ A \Longrightarrow \mathit{count}\ (\mathit{mset\text{-}set}\ A)\ x = 0\ (\mathbf{is}\ \mathit{PROP}\ ?Q)$
$x \notin A \Longrightarrow \mathit{count}\ (\mathit{mset\text{-}set}\ A)\ x = 0\ (\mathbf{is}\ \mathit{PROP}\ ?R)$
⟨*proof*⟩

lemma *elem-mset-set*[*simp*, *intro*]: $\mathit{finite}\ A \Longrightarrow x \in\#\ \mathit{mset\text{-}set}\ A \longleftrightarrow x \in A$
⟨*proof*⟩

lemma *mset-set-Union*:

finite A ⟹ finite B ⟹ A ∩ B = {} ⟹ mset-set (A ∪ B) = mset-set A + mset-set B
⟨*proof*⟩

lemma *filter-mset-mset-set* [*simp*]:
finite A ⟹ filter-mset P (mset-set A) = mset-set {x∈A. P x}
⟨*proof*⟩

lemma *mset-set-Diff*:
assumes *finite A B ⊆ A*
shows *mset-set (A − B) = mset-set A − mset-set B*
⟨*proof*⟩

lemma *mset-set-set*: *distinct xs ⟹ mset-set (set xs) = mset xs*
⟨*proof*⟩

lemma *count-mset-set'*: *count (mset-set A) x = (if finite A ∧ x ∈ A then 1 else 0)*
⟨*proof*⟩

lemma *subset-imp-msubset-mset-set*:
assumes *A ⊆ B finite B*
shows *mset-set A ⊆# mset-set B*
⟨*proof*⟩

lemma *mset-set-set-mset-msubset*: *mset-set (set-mset A) ⊆# A*
⟨*proof*⟩

context *linorder*
begin

definition *sorted-list-of-multiset* :: *'a multiset ⇒ 'a list*
where
sorted-list-of-multiset M = fold-mset insort [] M

lemma *sorted-list-of-multiset-empty* [*simp*]:
sorted-list-of-multiset {#} = []
⟨*proof*⟩

lemma *sorted-list-of-multiset-singleton* [*simp*]:
sorted-list-of-multiset {#x#} = [x]
⟨*proof*⟩

lemma *sorted-list-of-multiset-insert* [*simp*]:
sorted-list-of-multiset (add-mset x M) = List.insort x (sorted-list-of-multiset M)
⟨*proof*⟩

end

lemma *mset-sorted-list-of-multiset*[*simp*]: *mset* (*sorted-list-of-multiset M*) = *M*
⟨*proof*⟩

lemma *sorted-list-of-multiset-mset*[*simp*]: *sorted-list-of-multiset* (*mset xs*) = *sort xs*
⟨*proof*⟩

lemma *finite-set-mset-mset-set*[*simp*]: *finite A* ⟹ *set-mset* (*mset-set A*) = *A*
⟨*proof*⟩

lemma *mset-set-empty-iff*: *mset-set A* = {#} ⟷ *A* = {} ∨ *infinite A*
⟨*proof*⟩

lemma *infinite-set-mset-mset-set*: ¬ *finite A* ⟹ *set-mset* (*mset-set A*) = {}
⟨*proof*⟩

lemma *set-sorted-list-of-multiset* [*simp*]:
set (*sorted-list-of-multiset M*) = *set-mset M*
⟨*proof*⟩

lemma *sorted-list-of-mset-set* [*simp*]:
sorted-list-of-multiset (*mset-set A*) = *sorted-list-of-set A*
⟨*proof*⟩

lemma *mset-upt* [*simp*]: *mset* [*m*..<*n*] = *mset-set* {*m*..<*n*}
⟨*proof*⟩

lemma *image-mset-map-of*:
distinct (*map fst xs*) ⟹ {#*the* (*map-of xs i*). *i* ∈# *mset* (*map fst xs*)#} = *mset* (*map snd xs*)
⟨*proof*⟩

lemma *msubset-mset-set-iff*[*simp*]:
assumes *finite A finite B*
shows *mset-set A* ⊆# *mset-set B* ⟷ *A* ⊆ *B*
⟨*proof*⟩

lemma *mset-set-eq-iff*[*simp*]:
assumes *finite A finite B*
shows *mset-set A* = *mset-set B* ⟷ *A* = *B*
⟨*proof*⟩

lemma *image-mset-mset-set*:
assumes *inj-on f A*
shows *image-mset f* (*mset-set A*) = *mset-set* (*f* ‘ *A*)
⟨*proof*⟩

57.9 More properties of the replicate and repeat operations

lemma *in-replicate-mset*[*simp*]: $x \in\# \mathit{replicate\text{-}mset}\ n\ y \longleftrightarrow n > 0 \wedge x = y$
⟨*proof*⟩

lemma *set-mset-replicate-mset-subset*[*simp*]: $\mathit{set\text{-}mset}\ (\mathit{replicate\text{-}mset}\ n\ x) = (\mathit{if}\ n = 0\ \mathit{then}\ \{\}\ \mathit{else}\ \{x\})$
⟨*proof*⟩

lemma *size-replicate-mset*[*simp*]: $\mathit{size}\ (\mathit{replicate\text{-}mset}\ n\ M) = n$
⟨*proof*⟩

lemma *count-le-replicate-mset-subset-eq*: $n \leq \mathit{count}\ M\ x \longleftrightarrow \mathit{replicate\text{-}mset}\ n\ x \subseteq\# M$
⟨*proof*⟩

lemma *filter-eq-replicate-mset*: $\{\#y \in\# D.\ y = x\#\} = \mathit{replicate\text{-}mset}\ (\mathit{count}\ D\ x)\ x$
⟨*proof*⟩

lemma *replicate-count-mset-eq-filter-eq*: $\mathit{replicate}\ (\mathit{count}\ (\mathit{mset}\ xs)\ k)\ k = \mathit{filter}\ (\mathit{HOL.eq}\ k)\ xs$
⟨*proof*⟩

lemma *replicate-mset-eq-empty-iff* [*simp*]: $\mathit{replicate\text{-}mset}\ n\ a = \{\#\} \longleftrightarrow n = 0$
⟨*proof*⟩

lemma *replicate-mset-eq-iff*:
$\mathit{replicate\text{-}mset}\ m\ a = \mathit{replicate\text{-}mset}\ n\ b \longleftrightarrow m = 0 \wedge n = 0 \vee m = n \wedge a = b$
⟨*proof*⟩

lemma *repeat-mset-cancel1*: $\mathit{repeat\text{-}mset}\ a\ A = \mathit{repeat\text{-}mset}\ a\ B \longleftrightarrow A = B \vee a = 0$
⟨*proof*⟩

lemma *repeat-mset-cancel2*: $\mathit{repeat\text{-}mset}\ a\ A = \mathit{repeat\text{-}mset}\ b\ A \longleftrightarrow a = b \vee A = \{\#\}$
⟨*proof*⟩

lemma *repeat-mset-eq-empty-iff*: $\mathit{repeat\text{-}mset}\ n\ A = \{\#\} \longleftrightarrow n = 0 \vee A = \{\#\}$
⟨*proof*⟩

lemma *image-replicate-mset* [*simp*]:
$\mathit{image\text{-}mset}\ f\ (\mathit{replicate\text{-}mset}\ n\ a) = \mathit{replicate\text{-}mset}\ n\ (f\ a)$
⟨*proof*⟩

57.10 Big operators

locale *comm-monoid-mset* = *comm-monoid*
begin

interpretation *comp-fun-commute f*
⟨*proof*⟩

interpretation *comp?*: *comp-fun-commute f* ∘ *g*
⟨*proof*⟩

context
begin

definition *F* :: *'a multiset* ⇒ *'a*
where *eq-fold*: *F M* = *fold-mset f* **1** *M*

lemma *empty* [*simp*]: *F* {#} = **1**
⟨*proof*⟩

lemma *singleton* [*simp*]: *F* {#*x*#} = *x*
⟨*proof*⟩

lemma *union* [*simp*]: *F* (*M* + *N*) = *F M* ∗ *F N*
⟨*proof*⟩

lemma *add-mset* [*simp*]: *F* (*add-mset x N*) = *x* ∗ *F N*
⟨*proof*⟩

lemma *insert* [*simp*]:
shows *F* (*image-mset g* (*add-mset x A*)) = *g x* ∗ *F* (*image-mset g A*)
⟨*proof*⟩

lemma *remove*:
assumes *x* ∈# *A*
shows *F A* = *x* ∗ *F* (*A* − {#*x*#})
⟨*proof*⟩

lemma *neutral*:
∀ *x*∈#*A*. *x* = **1** ⟹ *F A* = **1**
⟨*proof*⟩

lemma *neutral-const* [*simp*]:
F (*image-mset* (λ-. **1**) *A*) = **1**
⟨*proof*⟩ **lemma** *F-image-mset-product*:
F {#*g x j* ∗ *F* {#*g i j*. *i* ∈# *A*#}. *j* ∈# *B*#} =
F (*image-mset* (*g x*) *B*) ∗ *F* {#*F* {#*g i j*. *i* ∈# *A*#}. *j* ∈# *B*#}
⟨*proof*⟩

lemma *commute*:
F (*image-mset* (λ*i*. *F* (*image-mset* (*g i*) *B*)) *A*) =
F (*image-mset* (λ*j*. *F* (*image-mset* (λ*i*. *g i j*) *A*)) *B*)
⟨*proof*⟩

lemma *distrib*: $F\ (\mathit{image\text{-}mset}\ (\lambda x.\ g\ x * h\ x)\ A) = F\ (\mathit{image\text{-}mset}\ g\ A) * F\ (\mathit{image\text{-}mset}\ h\ A)$
⟨*proof*⟩

lemma *union-disjoint*:
$A \cap\# B = \{\#\} \Longrightarrow F\ (\mathit{image\text{-}mset}\ g\ (A \cup\# B)) = F\ (\mathit{image\text{-}mset}\ g\ A) * F\ (\mathit{image\text{-}mset}\ g\ B)$
⟨*proof*⟩

end
end

lemma *comp-fun-commute-plus-mset*[*simp*]: *comp-fun-commute* (*op* + :: *'a multiset* ⇒ - ⇒ -)
⟨*proof*⟩

declare *comp-fun-commute.fold-mset-add-mset*[*OF comp-fun-commute-plus-mset*, *simp*]

lemma *in-mset-fold-plus-iff*[*iff*]: $x \in\# \mathit{fold\text{-}mset}\ (op +)\ M\ NN \longleftrightarrow x \in\# M \vee (\exists N.\ N \in\# NN \wedge x \in\# N)$
⟨*proof*⟩

context *comm-monoid-add*
begin

sublocale *sum-mset*: *comm-monoid-mset plus 0*
defines *sum-mset* = *sum-mset.F* ⟨*proof*⟩

lemma (**in** *semiring-1*) *sum-mset-replicate-mset* [*simp*]:
sum-mset (*replicate-mset n a*) = *of-nat n* ∗ *a*
⟨*proof*⟩

lemma *sum-unfold-sum-mset*:
sum f A = *sum-mset* (*image-mset f* (*mset-set A*))
⟨*proof*⟩

lemma *sum-mset-delta*: *sum-mset* (*image-mset* (λ*x*. *if x* = *y then c else 0*) *A*) = *c* ∗ *count A y*
⟨*proof*⟩

lemma *sum-mset-delta'*: *sum-mset* (*image-mset* (λ*x*. *if y* = *x then c else 0*) *A*) = *c* ∗ *count A y*
⟨*proof*⟩

end

lemma *of-nat-sum-mset* [*simp*]:

of-nat (*sum-mset M*) = *sum-mset* (*image-mset of-nat M*)
⟨*proof*⟩

lemma *sum-mset-0-iff* [*simp*]:
sum-mset M = (*0*::*'a*::*canonically-ordered-monoid-add*)
⟷ (∀ *x* ∈ *set-mset M*. *x* = *0*)
⟨*proof*⟩

lemma *sum-mset-diff*:
fixes *M N* :: (*'a* :: *ordered-cancel-comm-monoid-diff*) *multiset*
shows *N* ⊆# *M* ⟹ *sum-mset* (*M* − *N*) = *sum-mset M* − *sum-mset N*
⟨*proof*⟩

lemma *size-eq-sum-mset*: *size M* = *sum-mset* (*image-mset* (λ-. *1*) *M*)
⟨*proof*⟩

lemma *size-mset-set* [*simp*]: *size* (*mset-set A*) = *card A*
⟨*proof*⟩

lemma *sum-mset-sum-list*: *sum-mset* (*mset xs*) = *sum-list xs*
⟨*proof*⟩

syntax (*ASCII*)
-sum-mset-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-add* ((*3SUM -*:#*-.*
-) [*0*, *51*, *10*] *10*)
syntax
-sum-mset-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-add* ((*3*∑ *-*∈#*-.*
-) [*0*, *51*, *10*] *10*)
translations
∑ *i* ∈# *A*. *b* ⇌ *CONST sum-mset* (*CONST image-mset* (λ*i*. *b*) *A*)

lemma *sum-mset-distrib-left*:
fixes *f* :: *'a* ⇒ *'b*::*semiring-0*
shows *c* ∗ (∑ *x* ∈# *M*. *f x*) = (∑ *x* ∈# *M*. *c* ∗ *f*(*x*))
⟨*proof*⟩

lemma *sum-mset-distrib-right*:
fixes *f* :: *'a* ⇒ *'b*::*semiring-0*
shows (∑ *b* ∈# *B*. *f b*) ∗ *a* = (∑ *b* ∈# *B*. *f b* ∗ *a*)
⟨*proof*⟩

lemma *sum-mset-constant* [*simp*]:
fixes *y* :: *'b*::*semiring-1*
shows ‹(∑ *x*∈#*A*. *y*) = *of-nat* (*size A*) ∗ *y*›
⟨*proof*⟩

lemma (**in** *ordered-comm-monoid-add*) *sum-mset-mono*:
assumes ⋀*i*. *i* ∈# *K* ⟹ *f i* ≤ *g i*
shows *sum-mset* (*image-mset f K*) ≤ *sum-mset* (*image-mset g K*)

⟨*proof*⟩

lemma *sum-mset-product*:
fixes f :: $'a$::{*comm-monoid-add*,*times*} ⇒ $'b$::*semiring-0*
shows $(\sum i \in\# A.\ f\ i) * (\sum i \in\# B.\ g\ i) = (\sum i{\in}\#A.\ \sum j{\in}\#B.\ f\ i * g\ j)$
⟨*proof*⟩

abbreviation *Union-mset* :: $'a$ *multiset multiset* ⇒ $'a$ *multiset* $(\bigcup\#\text{-}\ [900]\ 900)$
where $\bigcup\#\ MM \equiv$ *sum-mset MM* — FIXME ambiguous notation – could likewise refer to $\bigsqcup\#$

lemma *set-mset-Union-mset*[*simp*]: *set-mset* $(\bigcup\#\ MM) = (\bigcup M \in$ *set-mset MM*. *set-mset M*)
⟨*proof*⟩

lemma *in-Union-mset-iff*[*iff*]: $x \in\# \bigcup\#\ MM \longleftrightarrow (\exists M.\ M \in\# MM \wedge x \in\# M)$
⟨*proof*⟩

lemma *count-sum*:
count (*sum f A*) *x* = *sum* (λ*a*. *count* (*f a*) *x*) *A*
⟨*proof*⟩

lemma *sum-eq-empty-iff*:
assumes *finite A*
shows *sum f A* = {#} ⟷ (∀ *a*∈*A*. *f a* = {#})
⟨*proof*⟩

lemma *Union-mset-empty-conv*[*simp*]: $\bigcup\#\ M = \{\#\} \longleftrightarrow (\forall i{\in}\#M.\ i = \{\#\})$
⟨*proof*⟩

context *comm-monoid-mult*
begin

sublocale *prod-mset*: *comm-monoid-mset times 1*
defines *prod-mset* = *prod-mset.F* ⟨*proof*⟩

lemma *prod-mset-empty*:
prod-mset {#} = *1*
⟨*proof*⟩

lemma *prod-mset-singleton*:
prod-mset {#*x*#} = *x*
⟨*proof*⟩

lemma *prod-mset-Un*:
prod-mset (*A* + *B*) = *prod-mset A* ∗ *prod-mset B*
⟨*proof*⟩

lemma *prod-mset-replicate-mset* [*simp*]:
prod-mset (*replicate-mset n a*) = *a* ^ *n*
⟨*proof*⟩

lemma *prod-unfold-prod-mset*:
prod f A = *prod-mset* (*image-mset f* (*mset-set A*))
⟨*proof*⟩

lemma *prod-mset-multiplicity*:
prod-mset M = *prod* (λ*x*. *x* ^ *count M x*) (*set-mset M*)
⟨*proof*⟩

lemma *prod-mset-delta*: *prod-mset* (*image-mset* (λ*x*. *if x* = *y then c else 1*) *A*) = *c* ^ *count A y*
⟨*proof*⟩

lemma *prod-mset-delta'*: *prod-mset* (*image-mset* (λ*x*. *if y* = *x then c else 1*) *A*) = *c* ^ *count A y*
⟨*proof*⟩

end

syntax (*ASCII*)
-prod-mset-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-mult* ((*3PROD* -:#-. -) [*0*, *51*, *10*] *10*)
syntax
-prod-mset-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-mult* ((*3*∏ -∈#-. -) [*0*, *51*, *10*] *10*)
translations
∏ *i* ∈# *A*. *b* ⇌ *CONST prod-mset* (*CONST image-mset* (λ*i*. *b*) *A*)

lemma *prod-mset-constant* [*simp*]: (∏ -∈#*A*. *c*) = *c* ^ *size A*
⟨*proof*⟩

lemma (**in** *comm-monoid-mult*) *prod-mset-subset-imp-dvd*:
assumes *A* ⊆# *B*
shows *prod-mset A dvd prod-mset B*
⟨*proof*⟩

lemma (**in** *comm-monoid-mult*) *dvd-prod-mset*:
assumes *x* ∈# *A*
shows *x dvd prod-mset A*
⟨*proof*⟩

lemma (**in** *semidom*) *prod-mset-zero-iff* [*iff*]:
prod-mset A = *0* ⟷ *0* ∈# *A*
⟨*proof*⟩

lemma (**in** *semidom-divide*) *prod-mset-diff*:

assumes $B \subseteq\# A$ **and** $0 \notin\# B$
shows *prod-mset* $(A - B) =$ *prod-mset* A *div prod-mset* B
⟨*proof*⟩

lemma (**in** *semidom-divide*) *prod-mset-minus*:
assumes $a \in\# A$ **and** $a \neq 0$
shows *prod-mset* $(A - \{\#a\#\}) =$ *prod-mset* A *div* a
⟨*proof*⟩

lemma (**in** *algebraic-semidom*) *is-unit-prod-mset-iff*:
is-unit (*prod-mset* A) $\longleftrightarrow$ $(\forall x \in\# A.$ *is-unit* $x)$
⟨*proof*⟩

lemma (**in** *normalization-semidom*) *normalize-prod-mset*:
normalize (*prod-mset* A) = *prod-mset* (*image-mset normalize* A)
⟨*proof*⟩

lemma (**in** *normalization-semidom*) *normalized-prod-msetI*:
assumes $\bigwedge a.\ a \in\# A \Longrightarrow$ *normalize* $a = a$
shows *normalize* (*prod-mset* A) = *prod-mset* A
⟨*proof*⟩

lemma *prod-mset-prod-list*: *prod-mset* (*mset xs*) = *prod-list xs*
⟨*proof*⟩

57.11 Alternative representations

57.11.1 Lists

context *linorder*
begin

lemma *mset-insort* [*simp*]:
mset (*insort-key k x xs*) = *add-mset x* (*mset xs*)
⟨*proof*⟩

lemma *mset-sort* [*simp*]:
mset (*sort-key k xs*) = *mset xs*
⟨*proof*⟩

This lemma shows which properties suffice to show that a function f with $f\ xs = ys$ behaves like sort.

lemma *properties-for-sort-key*:
assumes *mset ys* = *mset xs*
and $\bigwedge k.\ k \in$ *set ys* $\Longrightarrow$ *filter* $(\lambda x.\ f\ k = f\ x)\ ys =$ *filter* $(\lambda x.\ f\ k = f\ x)\ xs$
and *sorted* (*map f ys*)
shows *sort-key f xs* = *ys*
⟨*proof*⟩

lemma *properties-for-sort*:

assumes *multiset*: *mset ys = mset xs*
and *sorted ys*
shows *sort xs = ys*
⟨*proof*⟩

lemma *sort-key-inj-key-eq*:
assumes *mset-equal*: *mset xs = mset ys*
and *inj-on f (set xs)*
and *sorted (map f ys)*
shows *sort-key f xs = ys*
⟨*proof*⟩

lemma *sort-key-eq-sort-key*:
assumes *mset xs = mset ys*
and *inj-on f (set xs)*
shows *sort-key f xs = sort-key f ys*
⟨*proof*⟩

lemma *sort-key-by-quicksort*:
sort-key f xs = sort-key f [x←xs. f x < f (xs ! (length xs div 2))]
@ [x←xs. f x = f (xs ! (length xs div 2))]
@ sort-key f [x←xs. f x > f (xs ! (length xs div 2))] (**is** *sort-key f ?lhs = ?rhs*)
⟨*proof*⟩

lemma *sort-by-quicksort*:
sort xs = sort [x←xs. x < xs ! (length xs div 2)]
@ [x←xs. x = xs ! (length xs div 2)]
@ sort [x←xs. x > xs ! (length xs div 2)] (**is** *sort ?lhs = ?rhs*)
⟨*proof*⟩

A stable parametrized quicksort

definition *part* :: *('b ⇒ 'a) ⇒ 'a ⇒ 'b list ⇒ 'b list × 'b list × 'b list* **where**
part f pivot xs = ([x ← xs. f x < pivot], [x ← xs. f x = pivot], [x ← xs. pivot < f x])

lemma *part-code* [*code*]:
part f pivot [] = ([], [], [])
part f pivot (x # xs) = (let (lts, eqs, gts) = part f pivot xs; x' = f x in
if x' < pivot then (x # lts, eqs, gts)
else if x' > pivot then (lts, eqs, x # gts)
else (lts, x # eqs, gts))
⟨*proof*⟩

lemma *sort-key-by-quicksort-code* [*code*]:
sort-key f xs =
(case xs of
[] ⇒ []
| [x] ⇒ xs
| [x, y] ⇒ (if f x ≤ f y then xs else [y, x])

```
  | - ⇒
      let (lts, eqs, gts) = part f (f (xs ! (length xs div 2))) xs
      in sort-key f lts @ eqs @ sort-key f gts)
⟨proof⟩

end

hide-const (open) part

lemma mset-remdups-subset-eq: mset (remdups xs) ⊆# mset xs
  ⟨proof⟩

lemma mset-update:
  i < length ls ⟹ mset (ls[i := v]) = add-mset v (mset ls − {#ls ! i#})
⟨proof⟩

lemma mset-swap:
  i < length ls ⟹ j < length ls ⟹
    mset (ls[j := ls ! i, i := ls ! j]) = mset ls
  ⟨proof⟩
```

57.12 The multiset order

57.12.1 Well-foundedness

```
definition mult1 :: ('a × 'a) set ⇒ ('a multiset × 'a multiset) set where
  mult1 r = {(N, M). ∃ a M0 K. M = add-mset a M0 ∧ N = M0 + K ∧
      (∀ b. b ∈# K ⟶ (b, a) ∈ r)}

definition mult :: ('a × 'a) set ⇒ ('a multiset × 'a multiset) set where
  mult r = (mult1 r)+

lemma mult1I:
  assumes M = add-mset a M0 and N = M0 + K and ⋀b. b ∈# K ⟹ (b, a)
∈ r
  shows (N, M) ∈ mult1 r
  ⟨proof⟩

lemma mult1E:
  assumes (N, M) ∈ mult1 r
  obtains a M0 K where M = add-mset a M0 N = M0 + K ⋀b. b ∈# K ⟹
(b, a) ∈ r
  ⟨proof⟩

lemma mono-mult1:
  assumes r ⊆ r' shows mult1 r ⊆ mult1 r'
⟨proof⟩

lemma mono-mult:
  assumes r ⊆ r' shows mult r ⊆ mult r'
```

⟨*proof*⟩

lemma *not-less-empty* [*iff*]: $(M, \{\#\}) \notin$ *mult1* r
⟨*proof*⟩

lemma *less-add*:
assumes *mult1*: $(N,$ *add-mset a M0*$) \in$ *mult1* r
shows
$(\exists M.\ (M,\ M0) \in$ *mult1* $r \wedge N =$ *add-mset a* $M) \vee$
$(\exists K.\ (\forall b.\ b \in\# K \longrightarrow (b,\ a) \in r) \wedge N = M0 + K)$
⟨*proof*⟩

lemma *all-accessible*:
assumes *wf* r
shows $\forall M.\ M \in$ *Wellfounded.acc* (*mult1* r)
⟨*proof*⟩

theorem *wf-mult1*: *wf* $r \Longrightarrow$ *wf* (*mult1* r)
⟨*proof*⟩

theorem *wf-mult*: *wf* $r \Longrightarrow$ *wf* (*mult* r)
⟨*proof*⟩

57.12.2 Closure-free presentation

One direction.

lemma *mult-implies-one-step*:
assumes
trans: *trans* r **and**
MN: $(M,\ N) \in$ *mult* r
shows $\exists I\ J\ K.\ N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge (\forall k \in$ *set-mset* $K.\ \exists j \in$ *set-mset* $J.\ (k,\ j) \in r)$
⟨*proof*⟩

lemma *one-step-implies-mult*:
assumes
$J \neq \{\#\}$ **and**
$\forall k \in$ *set-mset* $K.\ \exists j \in$ *set-mset* $J.\ (k,\ j) \in r$
shows $(I + K,\ I + J) \in$ *mult* r
⟨*proof*⟩

lemma *subset-implies-mult*:
assumes *sub*: $A \subset\# B$
shows $(A,\ B) \in$ *mult* r
⟨*proof*⟩

57.13 The multiset extension is cancellative for multiset union

lemma *mult-cancel*:

assumes *trans s* **and** *irrefl s*
shows $(X + Z,\ Y + Z) \in \mathit{mult}\ s \longleftrightarrow (X,\ Y) \in \mathit{mult}\ s$ (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

lemmas *mult-cancel-add-mset* =
mult-cancel[*of* - - {#-#}, *unfolded union-mset-add-mset-right add.comm-neutral*]

lemma *mult-cancel-max*:
assumes *trans s* **and** *irrefl s*
shows $(X,\ Y) \in \mathit{mult}\ s \longleftrightarrow (X - X \cap\# Y,\ Y - X \cap\# Y) \in \mathit{mult}\ s$ (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

57.14 Quasi-executable version of the multiset extension

Predicate variants of *mult* and the reflexive closure of *mult*, which are executable whenever the given predicate P is. Together with the standard code equations for *op* $\cap\#$ and *op* $-$ this should yield quadratic (with respect to calls to P) implementations of *multp* and *multeqp*.

definition *multp* :: $('a \Rightarrow 'a \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathit{multiset} \Rightarrow 'a\ \mathit{multiset} \Rightarrow \mathit{bool}$ **where**
multp P N M =
($\mathit{let}\ Z = M \cap\# N;\ X = M - Z\ \mathit{in}$
$X \neq \{\#\} \wedge (\mathit{let}\ Y = N - Z\ \mathit{in}\ (\forall y \in \mathit{set\text{-}mset}\ Y.\ \exists x \in \mathit{set\text{-}mset}\ X.\ P\ y\ x))$)

definition *multeqp* :: $('a \Rightarrow 'a \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathit{multiset} \Rightarrow 'a\ \mathit{multiset} \Rightarrow \mathit{bool}$ **where**
multeqp P N M =
($\mathit{let}\ Z = M \cap\# N;\ X = M - Z;\ Y = N - Z\ \mathit{in}$
$(\forall y \in \mathit{set\text{-}mset}\ Y.\ \exists x \in \mathit{set\text{-}mset}\ X.\ P\ y\ x)$)

lemma *multp-iff*:
assumes *irrefl R* **and** *trans R* **and** [*simp*]: $\bigwedge x\ y.\ P\ x\ y \longleftrightarrow (x,\ y) \in R$
shows *multp P N M* $\longleftrightarrow (N,\ M) \in \mathit{mult}\ R$ (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

lemma *multeqp-iff*:
assumes *irrefl R* **and** *trans R* **and** $\bigwedge x\ y.\ P\ x\ y \longleftrightarrow (x,\ y) \in R$
shows *multeqp P N M* $\longleftrightarrow (N,\ M) \in (\mathit{mult}\ R)^=$
⟨*proof*⟩

57.14.1 Partial-order properties

lemma (**in** *preorder*) *mult1-lessE*:
assumes $(N,\ M) \in \mathit{mult1}\ \{(a,\ b).\ a < b\}$
obtains *a M0 K* **where** $M = \mathit{add\text{-}mset}\ a\ M0\ N = M0 + K$
$a \notin\# K\ \bigwedge b.\ b \in\# K \Longrightarrow b < a$
⟨*proof*⟩

instantiation *multiset* :: (*preorder*) *order*
begin

definition *less-multiset* :: *'a multiset ⇒ 'a multiset ⇒ bool*
  where $M' < M \longleftrightarrow (M', M) \in mult\ \{(x', x).\ x' < x\}$

definition *less-eq-multiset* :: *'a multiset ⇒ 'a multiset ⇒ bool*
  where *less-eq-multiset* $M'\ M \longleftrightarrow M' < M \vee M' = M$

instance
⟨*proof*⟩
end — FIXME avoid junk stemming from type class interpretation

lemma *mset-le-irrefl* [*elim*!]:
  fixes M :: *'a*::*preorder multiset*
  shows $M < M \Longrightarrow R$
  ⟨*proof*⟩

57.14.2 Monotonicity of multiset union

lemma *mult1-union*: $(B, D) \in mult1\ r \Longrightarrow (C + B, C + D) \in mult1\ r$
  ⟨*proof*⟩

lemma *union-le-mono2*: $B < D \Longrightarrow C + B < C + (D::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *union-le-mono1*: $B < D \Longrightarrow B + C < D + (C::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *union-less-mono*:
  fixes $A\ B\ C\ D$ :: *'a*::*preorder multiset*
  shows $A < C \Longrightarrow B < D \Longrightarrow A + B < C + D$
  ⟨*proof*⟩

instantiation *multiset* :: (*preorder*) *ordered-ab-semigroup-add*
begin
instance
  ⟨*proof*⟩
end

57.14.3 Termination proofs with multiset orders

lemma *multi-member-skip*: $x \in\# XS \Longrightarrow x \in\# \{\#\ y\ \#\} + XS$
  and *multi-member-this*: $x \in\# \{\#\ x\ \#\} + XS$
  and *multi-member-last*: $x \in\# \{\#\ x\ \#\}$
  ⟨*proof*⟩

definition *ms-strict* = *mult pair-less*
definition *ms-weak* = *ms-strict* ∪ *Id*

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
⟨*proof*⟩

lemma *smsI*:
(*set-mset A*, *set-mset B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-strict*
⟨*proof*⟩

lemma *wmsI*:
(*set-mset A*, *set-mset B*) ∈ *max-strict* ∨ *A* = {#} ∧ *B* = {#}
⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-weak*
⟨*proof*⟩

inductive *pw-leq*
where
pw-leq-empty: *pw-leq* {#} {#}
| *pw-leq-step*: ⟦(*x*,*y*) ∈ *pair-leq*; *pw-leq X Y* ⟧ ⟹ *pw-leq* ({#*x*#} + *X*) ({#*y*#} + *Y*)

lemma *pw-leq-lstep*:
(*x*, *y*) ∈ *pair-leq* ⟹ *pw-leq* {#*x*#} {#*y*#}
⟨*proof*⟩

lemma *pw-leq-split*:
assumes *pw-leq X Y*
shows ∃ *A B Z*. *X* = *A* + *Z* ∧ *Y* = *B* + *Z* ∧ ((*set-mset A*, *set-mset B*) ∈ *max-strict* ∨ (*B* = {#} ∧ *A* = {#}))
⟨*proof*⟩

lemma
assumes *pwleq*: *pw-leq Z Z′*
shows *ms-strictI*: (*set-mset A*, *set-mset B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z′* + *B*) ∈ *ms-strict*
and *ms-weakI1*: (*set-mset A*, *set-mset B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z′* + *B*) ∈ *ms-weak*
and *ms-weakI2*: (*Z* + {#}, *Z′* + {#}) ∈ *ms-weak*
⟨*proof*⟩

lemma *empty-neutral*: {#} + *x* = *x x* + {#} = *x*
and *nonempty-plus*: {# *x* #} + *rs* ≠ {#}
and *nonempty-single*: {# *x* #} ≠ {#}
⟨*proof*⟩

⟨*ML*⟩

57.15 Legacy theorem bindings

lemmas *multi-count-eq* = *multiset-eq-iff* [*symmetric*]

lemma *union-commute*: *M* + *N* = *N* + (*M*::′*a multiset*)
⟨*proof*⟩

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ multiset))$
⟨*proof*⟩

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
⟨*proof*⟩

lemmas *union-ac* = *union-assoc union-commute union-lcomm add-mset-commute*

lemma *union-right-cancel*: $M + K = N + K \longleftrightarrow M = (N::'a\ multiset)$
⟨*proof*⟩

lemma *union-left-cancel*: $K + M = K + N \longleftrightarrow M = (N::'a\ multiset)$
⟨*proof*⟩

lemma *multi-union-self-other-eq*: $(A::'a\ multiset) + X = A + Y \Longrightarrow X = Y$
⟨*proof*⟩

lemma *mset-subset-trans*: $(M::'a\ multiset) \subset\# K \Longrightarrow K \subset\# N \Longrightarrow M \subset\# N$
⟨*proof*⟩

lemma *multiset-inter-commute*: $A \cap\# B = B \cap\# A$
⟨*proof*⟩

lemma *multiset-inter-assoc*: $A \cap\# (B \cap\# C) = A \cap\# B \cap\# C$
⟨*proof*⟩

lemma *multiset-inter-left-commute*: $A \cap\# (B \cap\# C) = B \cap\# (A \cap\# C)$
⟨*proof*⟩

lemmas *multiset-inter-ac* =
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *mset-le-not-refl*: $\neg M < (M::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *mset-le-trans*: $K < M \Longrightarrow M < N \Longrightarrow K < (N::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *mset-le-not-sym*: $M < N \Longrightarrow \neg N < (M::'a::preorder\ multiset)$
⟨*proof*⟩

lemma *mset-le-asym*: $M < N \Longrightarrow (\neg P \Longrightarrow N < (M::'a::preorder\ multiset)) \Longrightarrow P$
⟨*proof*⟩

⟨*ML*⟩

57.16 Naive implementation using lists

code-datatype *mset*

lemma [*code*]: $\{\#\} = mset\ []$
⟨*proof*⟩

lemma [*code*]: *add-mset x* (*mset xs*) = *mset* (*x # xs*)
⟨*proof*⟩

lemma [*code*]: *Multiset.is-empty* (*mset xs*) ⟷ *List.null xs*
⟨*proof*⟩

lemma *union-code* [*code*]: *mset xs + mset ys* = *mset* (*xs @ ys*)
⟨*proof*⟩

lemma [*code*]: *image-mset f* (*mset xs*) = *mset* (*map f xs*)
⟨*proof*⟩

lemma [*code*]: *filter-mset f* (*mset xs*) = *mset* (*filter f xs*)
⟨*proof*⟩

lemma [*code*]: *mset xs − mset ys* = *mset* (*fold remove1 ys xs*)
⟨*proof*⟩

lemma [*code*]:
mset xs ∩# mset ys =
mset (*snd* (*fold* (λ*x* (*ys*, *zs*).
if x ∈ set ys then (*remove1 x ys*, *x # zs*) *else* (*ys*, *zs*)) *xs* (*ys*, [])))
⟨*proof*⟩

lemma [*code*]:
mset xs ∪# mset ys =
mset (*case-prod append* (*fold* (λ*x* (*ys*, *zs*). (*remove1 x ys*, *x # zs*)) *xs* (*ys*, [])))
⟨*proof*⟩

declare *in-multiset-in-set* [*code-unfold*]

lemma [*code*]: *count* (*mset xs*) *x* = *fold* (λ*y*. *if x* = *y then Suc else id*) *xs 0*
⟨*proof*⟩

declare *set-mset-mset* [*code*]

declare *sorted-list-of-multiset-mset* [*code*]

lemma [*code*]: — not very efficient, but representation-ignorant!
mset-set A = *mset* (*sorted-list-of-set A*)
⟨*proof*⟩

declare *size-mset* [*code*]

fun *subset-eq-mset-impl* :: *'a list* ⇒ *'a list* ⇒ *bool option* **where**
  subset-eq-mset-impl [] *ys* = *Some* (*ys* ≠ [])
| *subset-eq-mset-impl* (*Cons x xs*) *ys* = (*case List.extract* (*op* = *x*) *ys of*
    None ⇒ *None*
  | *Some* (*ys1*,-,*ys2*) ⇒ *subset-eq-mset-impl xs* (*ys1* @ *ys2*))

lemma *subset-eq-mset-impl*: (*subset-eq-mset-impl xs ys* = *None* ⟷ ¬ *mset xs* ⊆# *mset ys*) ∧
  (*subset-eq-mset-impl xs ys* = *Some True* ⟷ *mset xs* ⊂# *mset ys*) ∧
  (*subset-eq-mset-impl xs ys* = *Some False* ⟶ *mset xs* = *mset ys*)
⟨*proof*⟩

lemma [*code*]: *mset xs* ⊆# *mset ys* ⟷ *subset-eq-mset-impl xs ys* ≠ *None*
  ⟨*proof*⟩

lemma [*code*]: *mset xs* ⊂# *mset ys* ⟷ *subset-eq-mset-impl xs ys* = *Some True*
  ⟨*proof*⟩

instantiation *multiset* :: (*equal*) *equal*
begin

definition
  [*code del*]: *HOL.equal A* (*B* :: *'a multiset*) ⟷ *A* = *B*
lemma [*code*]: *HOL.equal* (*mset xs*) (*mset ys*) ⟷ *subset-eq-mset-impl xs ys* = *Some False*
  ⟨*proof*⟩

instance
  ⟨*proof*⟩

end

declare *sum-mset-sum-list* [*code*]

lemma [*code*]: *prod-mset* (*mset xs*) = *fold times xs 1*
⟨*proof*⟩

Exercise for the casual reader: add implementations for *op* ≤ and *op* < (multiset order).

Quickcheck generators

definition (**in** *term-syntax*)
  msetify :: *'a*::*typerep list* × (*unit* ⇒ *Code-Evaluation.term*)
    ⇒ *'a multiset* × (*unit* ⇒ *Code-Evaluation.term*) **where**
  [*code-unfold*]: *msetify xs* = *Code-Evaluation.valtermify mset* {·} *xs*

notation *fcomp* (**infixl** ∘> *60*)
notation *scomp* (**infixl** ∘→ *60*)

instantiation *multiset* :: (*random*) *random*
begin

definition
Quickcheck-Random.random i = Quickcheck-Random.random i ∘→ (λxs. Pair (msetify xs))

instance ⟨*proof*⟩

end

no-notation *fcomp* (**infixl** ∘> *60*)
no-notation *scomp* (**infixl** ∘→ *60*)

instantiation *multiset* :: (*full-exhaustive*) *full-exhaustive*
begin

definition *full-exhaustive-multiset* :: ('*a multiset* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where
full-exhaustive-multiset f i = Quickcheck-Exhaustive.full-exhaustive (λxs. f (msetify xs)) i

instance ⟨*proof*⟩

end

hide-const (**open**) *msetify*

57.17 BNF setup

definition *rel-mset* **where**
rel-mset R X Y ⟷ (∃xs ys. mset xs = X ∧ mset ys = Y ∧ list-all2 R xs ys)

lemma *mset-zip-take-Cons-drop-twice*:
assumes *length xs = length ys j ≤ length xs*
shows *mset (zip (take j xs @ x # drop j xs) (take j ys @ y # drop j ys)) =*
add-mset (x,y) (mset (zip xs ys))
⟨*proof*⟩

lemma *ex-mset-zip-left*:
assumes *length xs = length ys mset xs' = mset xs*
shows *∃ys'. length ys' = length xs' ∧ mset (zip xs' ys') = mset (zip xs ys)*
⟨*proof*⟩

lemma *list-all2-reorder-left-invariance*:
assumes *rel*: *list-all2 R xs ys* **and** *ms-x*: *mset xs' = mset xs*
shows *∃ys'. list-all2 R xs' ys' ∧ mset ys' = mset ys*
⟨*proof*⟩

lemma *ex-mset*: $\exists xs.\ mset\ xs = X$
⟨*proof*⟩

inductive *pred-mset* :: $('a \Rightarrow bool) \Rightarrow 'a\ multiset \Rightarrow bool$
where
pred-mset P {#}
| ⟦*P a*; *pred-mset P M*⟧ ⟹ *pred-mset P* (*add-mset a M*)

bnf *'a multiset*
map: *image-mset*
sets: *set-mset*
bd: *natLeq*
wits: {#}
rel: *rel-mset*
pred: *pred-mset*
⟨*proof*⟩

inductive *rel-mset'*
where
Zero[*intro*]: *rel-mset' R* {#} {#}
| *Plus*[*intro*]: ⟦*R a b*; *rel-mset' R M N*⟧ ⟹ *rel-mset' R* (*add-mset a M*) (*add-mset b N*)

lemma *rel-mset-Zero*: *rel-mset R* {#} {#}
⟨*proof*⟩

declare *multiset.count*[*simp*]
declare *Abs-multiset-inverse*[*simp*]
declare *multiset.count-inverse*[*simp*]
declare *union-preserves-multiset*[*simp*]

lemma *rel-mset-Plus*:
assumes *ab*: *R a b*
and *MN*: *rel-mset R M N*
shows *rel-mset R* (*add-mset a M*) (*add-mset b N*)
⟨*proof*⟩

lemma *rel-mset'-imp-rel-mset*: *rel-mset' R M N* ⟹ *rel-mset R M N*
⟨*proof*⟩

lemma *rel-mset-size*: *rel-mset R M N* ⟹ *size M* = *size N*
⟨*proof*⟩

lemma *multiset-induct2*[*case-names empty addL addR*]:
assumes *empty*: *P* {#} {#}
and *addL*: ⋀*a M N*. *P M N* ⟹ *P* (*add-mset a M*) *N*
and *addR*: ⋀*a M N*. *P M N* ⟹ *P M* (*add-mset a N*)
shows *P M N*

⟨*proof*⟩

lemma *multiset-induct2-size*[*consumes 1*, *case-names empty add*]:
assumes *c*: *size M* = *size N*
and *empty*: *P* {#} {#}
and *add*: ⋀*a b M N a b*. *P M N* ⟹ *P* (*add-mset a M*) (*add-mset b N*)
shows *P M N*
⟨*proof*⟩

lemma *msed-map-invL*:
assumes *image-mset f* (*add-mset a M*) = *N*
shows ∃ *N1*. *N* = *add-mset* (*f a*) *N1* ∧ *image-mset f M* = *N1*
⟨*proof*⟩

lemma *msed-map-invR*:
assumes *image-mset f M* = *add-mset b N*
shows ∃ *M1 a*. *M* = *add-mset a M1* ∧ *f a* = *b* ∧ *image-mset f M1* = *N*
⟨*proof*⟩

lemma *msed-rel-invL*:
assumes *rel-mset R* (*add-mset a M*) *N*
shows ∃ *N1 b*. *N* = *add-mset b N1* ∧ *R a b* ∧ *rel-mset R M N1*
⟨*proof*⟩

lemma *msed-rel-invR*:
assumes *rel-mset R M* (*add-mset b N*)
shows ∃ *M1 a*. *M* = *add-mset a M1* ∧ *R a b* ∧ *rel-mset R M1 N*
⟨*proof*⟩

lemma *rel-mset-imp-rel-mset′*:
assumes *rel-mset R M N*
shows *rel-mset′ R M N*
⟨*proof*⟩

lemma *rel-mset-rel-mset′*: *rel-mset R M N* = *rel-mset′ R M N*
⟨*proof*⟩

The main end product for *rel-mset*: inductive characterization:

lemmas *rel-mset-induct*[*case-names empty add*, *induct pred*: *rel-mset*] =
rel-mset′.induct[*unfolded rel-mset-rel-mset′*[*symmetric*]]

57.18 Size setup

lemma *multiset-size-o-map*: *size-multiset g* ∘ *image-mset f* = *size-multiset* (*g* ∘ *f*)
⟨*proof*⟩

⟨*ML*⟩

57.19 Lemmas about Size

lemma *size-mset-SucE*: *size A = Suc n* $\Longrightarrow$ ($\bigwedge$*a B*. *A = {#a#} + B* $\Longrightarrow$ *size B = n* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
⟨*proof*⟩

lemma *size-Suc-Diff1*: *x* $\in\#$ *M* $\Longrightarrow$ *Suc* (*size* (*M* $-$ {#*x*#})) = *size M*
⟨*proof*⟩

lemma *size-Diff-singleton*: *x* $\in\#$ *M* $\Longrightarrow$ *size* (*M* $-$ {#*x*#}) = *size M* $-$ *1*
⟨*proof*⟩

lemma *size-Diff-singleton-if*: *size* (*A* $-$ {#*x*#}) = (*if x* $\in\#$ *A then size A* $-$ *1 else size A*)
⟨*proof*⟩

lemma *size-Un-Int*: *size A + size B = size* (*A* $\cup\#$ *B*) + *size* (*A* $\cap\#$ *B*)
⟨*proof*⟩

lemma *size-Un-disjoint*: *A* $\cap\#$ *B* = {#} $\Longrightarrow$ *size* (*A* $\cup\#$ *B*) = *size A + size B*
⟨*proof*⟩

lemma *size-Diff-subset-Int*: *size* (*M* $-$ *M′*) = *size M* $-$ *size* (*M* $\cap\#$ *M′*)
⟨*proof*⟩

lemma *diff-size-le-size-Diff*: *size* (*M* :: - *multiset*) $-$ *size M′* $\leq$ *size* (*M* $-$ *M′*)
⟨*proof*⟩

lemma *size-Diff1-less*: *x*$\in\#$ *M* $\Longrightarrow$ *size* (*M* $-$ {#*x*#}) $<$ *size M*
⟨*proof*⟩

lemma *size-Diff2-less*: *x*$\in\#$ *M* $\Longrightarrow$ *y*$\in\#$ *M* $\Longrightarrow$ *size* (*M* $-$ {#*x*#} $-$ {#*y*#}) $<$ *size M*
⟨*proof*⟩

lemma *size-Diff1-le*: *size* (*M* $-$ {#*x*#}) $\leq$ *size M*
⟨*proof*⟩

lemma *size-psubset*: *M* $\subseteq\#$ *M′* $\Longrightarrow$ *size M* $<$ *size M′* $\Longrightarrow$ *M* $\subset\#$ *M′*
⟨*proof*⟩

hide-const (**open**) *wcount*

end

58 More Theorems about the Multiset Order

theory *Multiset-Order*
imports *Multiset*

begin

58.1 Alternative Characterizations

context *preorder*
begin

lemma *order-mult*: *class.order*
(λ*M N*. (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*} ∨ *M* = *N*)
(λ*M N*. (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*})
(**is** *class.order ?le ?less*)
⟨*proof*⟩

The Dershowitz–Manna ordering:

definition *less-multiset*$_{DM}$ **where**
less-multiset$_{DM}$ *M N* ⟷
(∃ *X Y*. *X* ≠ {#} ∧ *X* ⊆# *N* ∧ *M* = (*N* − *X*) + *Y* ∧ (∀ *k*. *k* ∈# *Y* ⟶ (∃ *a*. *a* ∈# *X* ∧ *k* < *a*)))

The Huet–Oppen ordering:

definition *less-multiset*$_{HO}$ **where**
less-multiset$_{HO}$ *M N* ⟷ *M* ≠ *N* ∧ (∀ *y*. *count N y* < *count M y* ⟶ (∃ *x*. *y* < *x* ∧ *count M x* < *count N x*))

lemma *mult-imp-less-multiset*$_{HO}$:
(*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*} ⟹ *less-multiset*$_{HO}$ *M N*
⟨*proof*⟩

lemma *less-multiset*$_{DM}$*-imp-mult*:
less-multiset$_{DM}$ *M N* ⟹ (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*}
⟨*proof*⟩

lemma *less-multiset*$_{HO}$*-imp-less-multiset*$_{DM}$: *less-multiset*$_{HO}$ *M N* ⟹ *less-multiset*$_{DM}$ *M N*
⟨*proof*⟩

lemma *mult-less-multiset*$_{DM}$: (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*} ⟷ *less-multiset*$_{DM}$ *M N*
⟨*proof*⟩

lemma *mult-less-multiset*$_{HO}$: (*M*, *N*) ∈ *mult* {(*x*, *y*). *x* < *y*} ⟷ *less-multiset*$_{HO}$ *M N*
⟨*proof*⟩

lemmas *mult*$_{DM}$ = *mult-less-multiset*$_{DM}$[*unfolded less-multiset*$_{DM}$*-def*]
lemmas *mult*$_{HO}$ = *mult-less-multiset*$_{HO}$[*unfolded less-multiset*$_{HO}$*-def*]

end

lemma *less-multiset-less-multiset*$_{HO}$:

$M < N \longleftrightarrow$ *less-multiset*$_{HO}$ M N
⟨*proof*⟩

lemmas *less-multiset*$_{DM}$ $=$ *mult*$_{DM}$[*folded less-multiset-def*]
lemmas *less-multiset*$_{HO}$ $=$ *mult*$_{HO}$[*folded less-multiset-def*]

lemma *subset-eq-imp-le-multiset*:
shows $M \subseteq\# N \Longrightarrow M \leq N$
⟨*proof*⟩

lemma *le-multiset-right-total*:
shows $M <$ *add-mset* x M
⟨*proof*⟩

lemma *less-eq-multiset-empty-left*[*simp*]:
shows $\{\#\} \leq M$
⟨*proof*⟩

lemma *ex-gt-imp-less-multiset*: $(\exists\, y.\ y \in\# N \wedge (\forall\, x.\ x \in\# M \longrightarrow x < y)) \Longrightarrow M < N$
⟨*proof*⟩

lemma *less-eq-multiset-empty-right*[*simp*]:
$M \neq \{\#\} \Longrightarrow \neg\ M \leq \{\#\}$
⟨*proof*⟩

lemma *le-multiset-empty-left*[*simp*]: $M \neq \{\#\} \Longrightarrow \{\#\} < M$
⟨*proof*⟩

lemma *le-multiset-empty-right*[*simp*]: $\neg\ M < \{\#\}$
⟨*proof*⟩

lemma *union-le-diff-plus*: $P \subseteq\# M \Longrightarrow N < P \Longrightarrow M - P + N < M$
⟨*proof*⟩

instantiation *multiset* :: (*preorder*) *ordered-ab-semigroup-monoid-add-imp-le*
begin

lemma *less-eq-multiset*$_{HO}$:
$M \leq N \longleftrightarrow (\forall\, y.\ count\ N\ y < count\ M\ y \longrightarrow (\exists\, x.\ y < x \wedge count\ M\ x < count\ N\ x))$
⟨*proof*⟩

instance ⟨*proof*⟩

lemma
fixes M N :: *'a multiset*
shows
less-eq-multiset-plus-left: $N \leq (M + N)$ **and**

less-eq-multiset-plus-right: $M \leq (M + N)$
⟨*proof*⟩

lemma
fixes $M\ N :: 'a\ multiset$
shows
le-multiset-plus-left-nonempty: $M \neq \{\#\} \Longrightarrow N < M + N$ **and**
le-multiset-plus-right-nonempty: $N \neq \{\#\} \Longrightarrow M < M + N$
⟨*proof*⟩

end

lemma *all-lt-Max-imp-lt-mset*: $N \neq \{\#\} \Longrightarrow (\forall\, a \in\# M.\ a < Max\ (set\text{-}mset\ N)) \Longrightarrow M < N$
⟨*proof*⟩

lemma *lt-imp-ex-count-lt*: $M < N \Longrightarrow \exists\, y.\ count\ M\ y < count\ N\ y$
⟨*proof*⟩

lemma *subset-imp-less-mset*: $A \subset\# B \Longrightarrow A < B$
⟨*proof*⟩

lemma *image-mset-strict-mono*:
assumes
mono-f: $\forall\, x \in set\text{-}mset\ M.\ \forall\, y \in set\text{-}mset\ N.\ x < y \longrightarrow f\ x < f\ y$ **and**
less: $M < N$
shows *image-mset f M* $<$ *image-mset f N*
⟨*proof*⟩

lemma *image-mset-mono*:
assumes
mono-f: $\forall\, x \in set\text{-}mset\ M.\ \forall\, y \in set\text{-}mset\ N.\ x < y \longrightarrow f\ x < f\ y$ **and**
less: $M \leq N$
shows *image-mset f M* $\leq$ *image-mset f N*
⟨*proof*⟩

lemma *mset-lt-single-right-iff* [*simp*]: $M < \{\#y\#\} \longleftrightarrow (\forall\, x \in\# M.\ x < y)$ **for** $y :: 'a{::}linorder$
⟨*proof*⟩

lemma *mset-le-single-right-iff* [*simp*]:
$M \leq \{\#y\#\} \longleftrightarrow M = \{\#y\#\} \vee (\forall\, x \in\# M.\ x < y)$ **for** $y :: 'a{::}linorder$
⟨*proof*⟩

58.2 Simprocs

lemma *mset-le-add-iff1*:
$j \leq (i{::}nat) \Longrightarrow (repeat\text{-}mset\ i\ u + m \leq repeat\text{-}mset\ j\ u + n) = (repeat\text{-}mset\ (i-j)\ u + m \leq n)$

⟨*proof*⟩

lemma *mset-le-add-iff2*:
$i \leq (j::nat) \Longrightarrow (repeat\text{-}mset\ i\ u + m \leq repeat\text{-}mset\ j\ u + n) = (m \leq repeat\text{-}mset\ (j-i)\ u + n)$
⟨*proof*⟩

⟨*ML*⟩

58.3 Additional facts and instantiations

lemma *ex-gt-count-imp-le-multiset*:
$(\forall y :: 'a :: order.\ y \in\# M + N \longrightarrow y \leq x) \Longrightarrow count\ M\ x < count\ N\ x \Longrightarrow M < N$
⟨*proof*⟩

lemma *mset-lt-single-iff*[*iff*]: $\{\#x\#\} < \{\#y\#\} \longleftrightarrow x < y$
⟨*proof*⟩

lemma *mset-le-single-iff*[*iff*]: $\{\#x\#\} \leq \{\#y\#\} \longleftrightarrow x \leq y$ **for** $x\ y :: 'a::order$
⟨*proof*⟩

instance *multiset* :: (*linorder*) *linordered-cancel-ab-semigroup-add*
⟨*proof*⟩

lemma *less-eq-multiset-total*:
fixes $M\ N :: 'a :: linorder\ multiset$
shows $\neg\ M \leq N \Longrightarrow N \leq M$
⟨*proof*⟩

instantiation *multiset* :: (*wellorder*) *wellorder*
begin

lemma *wf-less-multiset*: *wf* $\{(M :: 'a\ multiset,\ N).\ M < N\}$
⟨*proof*⟩

instance ⟨*proof*⟩

end

instantiation *multiset* :: (*preorder*) *order-bot*
begin

definition *bot-multiset* :: $'a\ multiset$ **where** *bot-multiset* $= \{\#\}$

instance ⟨*proof*⟩

end

instance *multiset* :: (*preorder*) *no-top*
⟨*proof*⟩

instance *multiset* :: (*preorder*) *ordered-cancel-comm-monoid-add*
⟨*proof*⟩

instantiation *multiset* :: (*linorder*) *distrib-lattice*
begin

definition *inf-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* **where**
inf-multiset A B = (*if A* < *B then A else B*)

definition *sup-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* **where**
sup-multiset A B = (*if B* > *A then B else A*)

instance
⟨*proof*⟩

end

end

59 Permutations, both general and specifically on finite sets.

theory *Permutations*
imports *Multiset Disjoint-Sets*
begin

59.1 Transpositions

lemma *swap-id-idempotent* [*simp*]: *Fun.swap a b id* ∘ *Fun.swap a b id* = *id*
⟨*proof*⟩

lemma *inv-swap-id*: *inv* (*Fun.swap a b id*) = *Fun.swap a b id*
⟨*proof*⟩

lemma *swap-id-eq*: *Fun.swap a b id x* = (*if x* = *a then b else if x* = *b then a else x*)
⟨*proof*⟩

lemma *bij-inv-eq-iff*: *bij p* ⟹ *x* = *inv p y* ⟷ *p x* = *y*
⟨*proof*⟩

lemma *bij-swap-comp*:
assumes *bij p*
shows *Fun.swap a b id* ∘ *p* = *Fun.swap* (*inv p a*) (*inv p b*) *p*
⟨*proof*⟩

lemma *bij-swap-compose-bij*:
assumes *bij p*
shows *bij* (*Fun.swap a b id* ∘ *p*)
⟨*proof*⟩

59.2 Basic consequences of the definition

definition *permutes* (**infixr** *permutes 41*)
where (*p permutes S*) ⟷ (∀ *x*. *x* ∉ *S* ⟶ *p x* = *x*) ∧ (∀ *y*. ∃!*x*. *p x* = *y*)

lemma *permutes-in-image*: *p permutes S* ⟹ *p x* ∈ *S* ⟷ *x* ∈ *S*
⟨*proof*⟩

lemma *permutes-not-in*: *f permutes S* ⟹ *x* ∉ *S* ⟹ *f x* = *x*
⟨*proof*⟩

lemma *permutes-image*: *p permutes S* ⟹ *p* ‘ *S* = *S*
⟨*proof*⟩

lemma *permutes-inj*: *p permutes S* ⟹ *inj p*
⟨*proof*⟩

lemma *permutes-inj-on*: *f permutes S* ⟹ *inj-on f A*
⟨*proof*⟩

lemma *permutes-surj*: *p permutes s* ⟹ *surj p*
⟨*proof*⟩

lemma *permutes-bij*: *p permutes s* ⟹ *bij p*
⟨*proof*⟩

lemma *permutes-imp-bij*: *p permutes S* ⟹ *bij-betw p S S*
⟨*proof*⟩

lemma *bij-imp-permutes*: *bij-betw p S S* ⟹ (⋀*x*. *x* ∉ *S* ⟹ *p x* = *x*) ⟹ *p permutes S*
⟨*proof*⟩

lemma *permutes-inv-o*:
assumes *permutes*: *p permutes S*
shows *p* ∘ *inv p* = *id*
and *inv p* ∘ *p* = *id*
⟨*proof*⟩

lemma *permutes-inverses*:
fixes *p* :: *'a* ⇒ *'a*
assumes *permutes*: *p permutes S*
shows *p* (*inv p x*) = *x*

and *inv p (p x) = x*
⟨*proof*⟩

lemma *permutes-subset*: *p permutes S ⟹ S ⊆ T ⟹ p permutes T*
⟨*proof*⟩

lemma *permutes-empty*[*simp*]: *p permutes {} ⟷ p = id*
⟨*proof*⟩

lemma *permutes-sing*[*simp*]: *p permutes {a} ⟷ p = id*
⟨*proof*⟩

lemma *permutes-univ*: *p permutes UNIV ⟷ (∀ y. ∃!x. p x = y)*
⟨*proof*⟩

lemma *permutes-inv-eq*: *p permutes S ⟹ inv p y = x ⟷ p x = y*
⟨*proof*⟩

lemma *permutes-swap-id*: *a ∈ S ⟹ b ∈ S ⟹ Fun.swap a b id permutes S*
⟨*proof*⟩

lemma *permutes-superset*: *p permutes S ⟹ (∀ x ∈ S − T. p x = x) ⟹ p permutes T*
⟨*proof*⟩

lemma *permutes-bij-inv-into*:
fixes *A* :: *'a set*
and *B* :: *'b set*
assumes *p permutes A*
and *bij-betw f A B*
shows *(λx. if x ∈ B then f (p (inv-into A f x)) else x) permutes B*
⟨*proof*⟩

lemma *permutes-image-mset*:
assumes *p permutes A*
shows *image-mset p (mset-set A) = mset-set A*
⟨*proof*⟩

lemma *permutes-implies-image-mset-eq*:
assumes *p permutes A* ⋀*x. x ∈ A ⟹ f x = f' (p x)*
shows *image-mset f' (mset-set A) = image-mset f (mset-set A)*
⟨*proof*⟩

59.3 Group properties

lemma *permutes-id*: *id permutes S*
⟨*proof*⟩

lemma *permutes-compose*: *p permutes S* $\Longrightarrow$ *q permutes S* $\Longrightarrow$ *q* $\circ$ *p permutes S*
⟨*proof*⟩

lemma *permutes-inv*:
assumes *p permutes S*
shows *inv p permutes S*
⟨*proof*⟩

lemma *permutes-inv-inv*:
assumes *p permutes S*
shows *inv* (*inv p*) = *p*
⟨*proof*⟩

lemma *permutes-invI*:
assumes *perm*: *p permutes S*
and *inv*: $\bigwedge x.\ x \in S \Longrightarrow p'\ (p\ x) = x$
and *outside*: $\bigwedge x.\ x \notin S \Longrightarrow p'\ x = x$
shows *inv p* = *p'*
⟨*proof*⟩

lemma *permutes-vimage*: *f permutes A* $\Longrightarrow$ *f* −‘ *A* = *A*
⟨*proof*⟩

59.4 Mapping permutations with bijections

lemma *bij-betw-permutations*:
assumes *bij-betw f A B*
shows *bij-betw* (λπ *x*. *if x* ∈ *B then f* (π (*inv-into A f x*)) *else x*)
{π. π *permutes A*} {π. π *permutes B*} (**is** *bij-betw ?f* - -)
⟨*proof*⟩

lemma *bij-betw-derangements*:
assumes *bij-betw f A B*
shows *bij-betw* (λπ *x*. *if x* ∈ *B then f* (π (*inv-into A f x*)) *else x*)
{π. π *permutes A* ∧ (∀ *x*∈*A*. π *x* ≠ *x*)} {π. π *permutes B* ∧ (∀ *x*∈*B*. π *x* ≠ *x*)}
(**is** *bij-betw ?f* - -)
⟨*proof*⟩

59.5 The number of permutations on a finite set

lemma *permutes-insert-lemma*:
assumes *p permutes* (*insert a S*)
shows *Fun.swap a* (*p a*) *id* ∘ *p permutes S*
⟨*proof*⟩

lemma *permutes-insert*: {*p*. *p permutes* (*insert a S*)} =
(λ(*b*, *p*). *Fun.swap a b id* ∘ *p*) ‘ {(*b*, *p*). *b* ∈ *insert a S* ∧ *p* ∈ {*p*. *p permutes S*}}
⟨*proof*⟩

lemma *card-permutations*:
assumes *card S = n*
and *finite S*
shows *card {p. p permutes S} = fact n*
⟨*proof*⟩

lemma *finite-permutations*:
assumes *finite S*
shows *finite {p. p permutes S}*
⟨*proof*⟩

59.6 Permutations of index set for iterated operations

lemma (**in** *comm-monoid-set*) *permute*:
assumes *p permutes S*
shows *F g S = F (g ∘ p) S*
⟨*proof*⟩

59.7 Various combinations of transpositions with 2, 1 and 0 common elements

lemma *swap-id-common*: *a ≠ c ⟹ b ≠ c ⟹*
Fun.swap a b id ∘ Fun.swap a c id = Fun.swap b c id ∘ Fun.swap a b id
⟨*proof*⟩

lemma *swap-id-common'*: *a ≠ b ⟹ a ≠ c ⟹*
Fun.swap a c id ∘ Fun.swap b c id = Fun.swap b c id ∘ Fun.swap a b id
⟨*proof*⟩

lemma *swap-id-independent*: *a ≠ c ⟹ a ≠ d ⟹ b ≠ c ⟹ b ≠ d ⟹*
Fun.swap a b id ∘ Fun.swap c d id = Fun.swap c d id ∘ Fun.swap a b id
⟨*proof*⟩

59.8 Permutations as transposition sequences

inductive *swapidseq* :: *nat ⇒ ('a ⇒ 'a) ⇒ bool*
where
id[*simp*]: *swapidseq 0 id*
| *comp-Suc*: *swapidseq n p ⟹ a ≠ b ⟹ swapidseq (Suc n) (Fun.swap a b id ∘ p)*

declare *id*[*unfolded id-def*, *simp*]

definition *permutation p ⟷ (∃ n. swapidseq n p)*

59.9 Some closure properties of the set of permutations, with lengths

lemma *permutation-id*[*simp*]: *permutation id*

⟨*proof*⟩

declare *permutation-id*[*unfolded id-def*, *simp*]

lemma *swapidseq-swap*: *swapidseq* (*if a = b then 0 else 1*) (*Fun.swap a b id*)
⟨*proof*⟩

lemma *permutation-swap-id*: *permutation* (*Fun.swap a b id*)
⟨*proof*⟩

lemma *swapidseq-comp-add*: *swapidseq n p* ⟹ *swapidseq m q* ⟹ *swapidseq* (*n + m*) (*p* ∘ *q*)
⟨*proof*⟩

lemma *permutation-compose*: *permutation p* ⟹ *permutation q* ⟹ *permutation* (*p* ∘ *q*)
⟨*proof*⟩

lemma *swapidseq-endswap*: *swapidseq n p* ⟹ *a* ≠ *b* ⟹ *swapidseq* (*Suc n*) (*p* ∘ *Fun.swap a b id*)
⟨*proof*⟩

lemma *swapidseq-inverse-exists*: *swapidseq n p* ⟹ ∃ *q*. *swapidseq n q* ∧ *p* ∘ *q* = *id* ∧ *q* ∘ *p* = *id*
⟨*proof*⟩

lemma *swapidseq-inverse*:
assumes *swapidseq n p*
shows *swapidseq n* (*inv p*)
⟨*proof*⟩

lemma *permutation-inverse*: *permutation p* ⟹ *permutation* (*inv p*)
⟨*proof*⟩

59.10 The identity map only has even transposition sequences

lemma *symmetry-lemma*:
assumes ⋀*a b c d*. *P a b c d* ⟹ *P a b d c*
and ⋀*a b c d*. *a* ≠ *b* ⟹ *c* ≠ *d* ⟹
a = *c* ∧ *b* = *d* ∨ *a* = *c* ∧ *b* ≠ *d* ∨ *a* ≠ *c* ∧ *b* = *d* ∨ *a* ≠ *c* ∧ *a* ≠ *d* ∧ *b* ≠ *c* ∧ *b* ≠ *d* ⟹
P a b c d
shows ⋀*a b c d*. *a* ≠ *b* ⟶ *c* ≠ *d* ⟶ *P a b c d*
⟨*proof*⟩

lemma *swap-general*: *a* ≠ *b* ⟹ *c* ≠ *d* ⟹
Fun.swap a b id ∘ *Fun.swap c d id* = *id* ∨
(∃ *x y z*. *x* ≠ *a* ∧ *y* ≠ *a* ∧ *z* ≠ *a* ∧ *x* ≠ *y* ∧

Fun.swap a b id ∘ Fun.swap c d id = Fun.swap x y id ∘ Fun.swap a z id)
⟨*proof*⟩

lemma *swapidseq-id-iff*[*simp*]: *swapidseq 0 p* ⟷ *p = id*
⟨*proof*⟩

lemma *swapidseq-cases*: *swapidseq n p* ⟷
n = 0 ∧ p = id ∨ (∃ a b q m. n = Suc m ∧ p = Fun.swap a b id ∘ q ∧ swapidseq m q ∧ a ≠ b)
⟨*proof*⟩

lemma *fixing-swapidseq-decrease*:
assumes *swapidseq n p*
and *a ≠ b*
and (*Fun.swap a b id ∘ p*) *a = a*
shows *n ≠ 0 ∧ swapidseq* (*n − 1*) (*Fun.swap a b id ∘ p*)
⟨*proof*⟩

lemma *swapidseq-identity-even*:
assumes *swapidseq n* (*id* :: *'a ⇒ 'a*)
shows *even n*
⟨*proof*⟩

59.11 Therefore we have a welldefined notion of parity

definition *evenperm p = even* (*SOME n. swapidseq n p*)

lemma *swapidseq-even-even*:
assumes *m*: *swapidseq m p*
and *n*: *swapidseq n p*
shows *even m* ⟷ *even n*
⟨*proof*⟩

lemma *evenperm-unique*:
assumes *p*: *swapidseq n p*
and *n*:*even n = b*
shows *evenperm p = b*
⟨*proof*⟩

59.12 And it has the expected composition properties

lemma *evenperm-id*[*simp*]: *evenperm id = True*
⟨*proof*⟩

lemma *evenperm-swap*: *evenperm* (*Fun.swap a b id*) = (*a = b*)
⟨*proof*⟩

lemma *evenperm-comp*:
assumes *permutation p permutation q*
shows *evenperm* (*p ∘ q*) ⟷ *evenperm p = evenperm q*

⟨*proof*⟩

lemma *evenperm-inv*:
  assumes *permutation p*
  shows *evenperm* (*inv p*) = *evenperm p*
⟨*proof*⟩

59.13 A more abstract characterization of permutations

lemma *bij-iff*: *bij f* ⟷ (∀ *x*. ∃!*y*. *f y* = *x*)
  ⟨*proof*⟩

lemma *permutation-bijective*:
  assumes *permutation p*
  shows *bij p*
⟨*proof*⟩

lemma *permutation-finite-support*:
  assumes *permutation p*
  shows *finite* {*x*. *p x* ≠ *x*}
⟨*proof*⟩

lemma *permutation-lemma*:
  assumes *finite S*
    and *bij p*
    and ∀ *x*. *x*∉ *S* ⟶ *p x* = *x*
  shows *permutation p*
  ⟨*proof*⟩

lemma *permutation*: *permutation p* ⟷ *bij p* ∧ *finite* {*x*. *p x* ≠ *x*}
  (**is** *?lhs* ⟷ *?b* ∧ *?f*)
⟨*proof*⟩

lemma *permutation-inverse-works*:
  assumes *permutation p*
  shows *inv p* ∘ *p* = *id*
    and *p* ∘ *inv p* = *id*
  ⟨*proof*⟩

lemma *permutation-inverse-compose*:
  assumes *p*: *permutation p*
    and *q*: *permutation q*
  shows *inv* (*p* ∘ *q*) = *inv q* ∘ *inv p*
⟨*proof*⟩

59.14 Relation to *permutes*

lemma *permutation-permutes*: *permutation p* ⟷ (∃ *S*. *finite S* ∧ *p permutes S*)
  ⟨*proof*⟩

59.15 Hence a sort of induction principle composing by swaps

lemma *permutes-induct*: *finite* $S \Longrightarrow P\ id \Longrightarrow$
$(\bigwedge a\ b\ p.\ a \in S \Longrightarrow b \in S \Longrightarrow P\ p \Longrightarrow P\ p \Longrightarrow permutation\ p \Longrightarrow P\ (Fun.swap\ a\ b\ id \circ p)) \Longrightarrow$
$(\bigwedge p.\ p\ permutes\ S \Longrightarrow P\ p)$
⟨*proof*⟩

59.16 Sign of a permutation as a real number

definition $sign\ p = (if\ evenperm\ p\ then\ (1::int)\ else\ -1)$

lemma *sign-nz*: $sign\ p \neq 0$
⟨*proof*⟩

lemma *sign-id*: $sign\ id = 1$
⟨*proof*⟩

lemma *sign-inverse*: $permutation\ p \Longrightarrow sign\ (inv\ p) = sign\ p$
⟨*proof*⟩

lemma *sign-compose*: $permutation\ p \Longrightarrow permutation\ q \Longrightarrow sign\ (p \circ q) = sign\ p * sign\ q$
⟨*proof*⟩

lemma *sign-swap-id*: $sign\ (Fun.swap\ a\ b\ id) = (if\ a = b\ then\ 1\ else\ -1)$
⟨*proof*⟩

lemma *sign-idempotent*: $sign\ p * sign\ p = 1$
⟨*proof*⟩

59.17 Permuting a list

This function permutes a list by applying a permutation to the indices.

definition $permute\text{-}list :: (nat \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow 'a\ list$
where $permute\text{-}list\ f\ xs = map\ (\lambda i.\ xs\ !\ (f\ i))\ [0..<length\ xs]$

lemma *permute-list-map*:
assumes $f\ permutes\ \{..<length\ xs\}$
shows $permute\text{-}list\ f\ (map\ g\ xs) = map\ g\ (permute\text{-}list\ f\ xs)$
⟨*proof*⟩

lemma *permute-list-nth*:
assumes $f\ permutes\ \{..<length\ xs\}\ i < length\ xs$
shows $permute\text{-}list\ f\ xs\ !\ i = xs\ !\ f\ i$
⟨*proof*⟩

lemma *permute-list-Nil* [*simp*]: $permute\text{-}list\ f\ [] = []$
⟨*proof*⟩

lemma *length-permute-list* [*simp*]: *length* (*permute-list f xs*) = *length xs*
⟨*proof*⟩

lemma *permute-list-compose*:
assumes *g permutes* {..<*length xs*}
shows *permute-list* (*f* ∘ *g*) *xs* = *permute-list g* (*permute-list f xs*)
⟨*proof*⟩

lemma *permute-list-ident* [*simp*]: *permute-list* (λ*x*. *x*) *xs* = *xs*
⟨*proof*⟩

lemma *permute-list-id* [*simp*]: *permute-list id xs* = *xs*
⟨*proof*⟩

lemma *mset-permute-list* [*simp*]:
fixes *xs* :: *'a list*
assumes *f permutes* {..<*length xs*}
shows *mset* (*permute-list f xs*) = *mset xs*
⟨*proof*⟩

lemma *set-permute-list* [*simp*]:
assumes *f permutes* {..<*length xs*}
shows *set* (*permute-list f xs*) = *set xs*
⟨*proof*⟩

lemma *distinct-permute-list* [*simp*]:
assumes *f permutes* {..<*length xs*}
shows *distinct* (*permute-list f xs*) = *distinct xs*
⟨*proof*⟩

lemma *permute-list-zip*:
assumes *f permutes A A* = {..<*length xs*}
assumes [*simp*]: *length xs* = *length ys*
shows *permute-list f* (*zip xs ys*) = *zip* (*permute-list f xs*) (*permute-list f ys*)
⟨*proof*⟩

lemma *map-of-permute*:
assumes *σ permutes fst ' set xs*
shows *map-of xs* ∘ *σ* = *map-of* (*map* (λ(*x*,*y*). (*inv σ x*, *y*)) *xs*)
(**is** - = *map-of* (*map ?f* -))
⟨*proof*⟩

59.18 More lemmas about permutations

The following few lemmas were contributed by Lukas Bulwahn.

lemma *count-image-mset-eq-card-vimage*:
assumes *finite A*
shows *count* (*image-mset f* (*mset-set A*)) *b* = *card* {*a* ∈ *A*. *f a* = *b*}
⟨*proof*⟩

lemma *image-mset-eq-implies-permutes*:
fixes $f :: 'a \Rightarrow 'b$
assumes *finite A*
and *mset-eq*: *image-mset f* (*mset-set A*) = *image-mset f'* (*mset-set A*)
obtains *p* **where** *p permutes A* **and** $\forall x{\in}A.\ f\ x = f'\ (p\ x)$
⟨*proof*⟩

lemma *mset-set-upto-eq-mset-upto*: *mset-set* {..<*n*} = *mset* [*0*..<*n*]
⟨*proof*⟩

lemma *mset-eq-permutation*:
fixes *xs ys* :: *'a list*
assumes *mset-eq*: *mset xs* = *mset ys*
obtains *p* **where** *p permutes* {..<*length ys*} *permute-list p ys* = *xs*
⟨*proof*⟩

lemma *permutes-natset-le*:
fixes *S* :: *'a*::*wellorder set*
assumes *p permutes S*
and $\forall\, i \in S.\ p\ i \leq i$
shows *p* = *id*
⟨*proof*⟩

lemma *permutes-natset-ge*:
fixes *S* :: *'a*::*wellorder set*
assumes *p*: *p permutes S*
and *le*: $\forall\, i \in S.\ p\ i \geq i$
shows *p* = *id*
⟨*proof*⟩

lemma *image-inverse-permutations*: {*inv p* |*p*. *p permutes S*} = {*p*. *p permutes S*}
⟨*proof*⟩

lemma *image-compose-permutations-left*:
assumes *q permutes S*
shows {*q* ∘ *p* |*p*. *p permutes S*} = {*p*. *p permutes S*}
⟨*proof*⟩

lemma *image-compose-permutations-right*:
assumes *q permutes S*
shows {*p* ∘ *q* | *p*. *p permutes S*} = {*p* . *p permutes S*}
⟨*proof*⟩

lemma *permutes-in-seg*: *p permutes* {*1* ..*n*} ⟹ *i* ∈ {*1*..*n*} ⟹ *1* ≤ *p i* ∧ *p i* ≤ *n*

⟨*proof*⟩

lemma *sum-permutations-inverse*: *sum f* {*p*. *p permutes S*} = *sum* (λ*p*. *f*(*inv p*)) {*p*. *p permutes S*}
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *setum-permutations-compose-left*:
assumes *q*: *q permutes S*
shows *sum f* {*p*. *p permutes S*} = *sum* (λ*p*. *f*(*q* ∘ *p*)) {*p*. *p permutes S*}
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *sum-permutations-compose-right*:
assumes *q*: *q permutes S*
shows *sum f* {*p*. *p permutes S*} = *sum* (λ*p*. *f*(*p* ∘ *q*)) {*p*. *p permutes S*}
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

59.19 Sum over a set of permutations (could generalize to iteration)

lemma *sum-over-permutations-insert*:
assumes *fS*: *finite S*
and *aS*: *a* ∉ *S*
shows *sum f* {*p*. *p permutes* (*insert a S*)} =
sum (λ*b*. *sum* (λ*q*. *f* (*Fun.swap a b id* ∘ *q*)) {*p*. *p permutes S*}) (*insert a S*)
⟨*proof*⟩

59.20 Constructing permutations from association lists

definition *list-permutes* :: ('*a* × '*a*) *list* ⇒ '*a set* ⇒ *bool*
where *list-permutes xs A* ⟷
set (*map fst xs*) ⊆ *A* ∧
set (*map snd xs*) = *set* (*map fst xs*) ∧
distinct (*map fst xs*) ∧
distinct (*map snd xs*)

lemma *list-permutesI* [*simp*]:
assumes *set* (*map fst xs*) ⊆ *A set* (*map snd xs*) = *set* (*map fst xs*) *distinct* (*map fst xs*)
shows *list-permutes xs A*
⟨*proof*⟩

definition *permutation-of-list* :: ('*a* × '*a*) *list* ⇒ '*a* ⇒ '*a*
where *permutation-of-list xs x* = (*case map-of xs x of None* ⇒ *x* | *Some y* ⇒ *y*)

lemma *permutation-of-list-Cons*:
permutation-of-list ((*x*, *y*) # *xs*) *x'* = (*if x* = *x'* *then y else permutation-of-list*

xs x')
⟨*proof*⟩

fun *inverse-permutation-of-list* :: ('*a* × '*a*) *list* ⇒ '*a* ⇒ '*a*
where
inverse-permutation-of-list [] *x* = *x*
| *inverse-permutation-of-list* ((*y*, *x'*) # *xs*) *x* =
(*if x* = *x' then y else inverse-permutation-of-list xs x*)

declare *inverse-permutation-of-list.simps* [*simp del*]

lemma *inj-on-map-of*:
assumes *distinct* (*map snd xs*)
shows *inj-on* (*map-of xs*) (*set* (*map fst xs*))
⟨*proof*⟩

lemma *inj-on-the*: *None* ∉ *A* ⟹ *inj-on the A*
⟨*proof*⟩

lemma *inj-on-map-of'*:
assumes *distinct* (*map snd xs*)
shows *inj-on* (*the* ∘ *map-of xs*) (*set* (*map fst xs*))
⟨*proof*⟩

lemma *image-map-of*:
assumes *distinct* (*map fst xs*)
shows *map-of xs* ‘ *set* (*map fst xs*) = *Some* ‘ *set* (*map snd xs*)
⟨*proof*⟩

lemma *the-Some-image* [*simp*]: *the* ‘ *Some* ‘ *A* = *A*
⟨*proof*⟩

lemma *image-map-of'*:
assumes *distinct* (*map fst xs*)
shows (*the* ∘ *map-of xs*) ‘ *set* (*map fst xs*) = *set* (*map snd xs*)
⟨*proof*⟩

lemma *permutation-of-list-permutes* [*simp*]:
assumes *list-permutes xs A*
shows *permutation-of-list xs permutes A*
(**is** *?f permutes* -)
⟨*proof*⟩

lemma *eval-permutation-of-list* [*simp*]:
permutation-of-list [] *x* = *x*
x = *x'* ⟹ *permutation-of-list* ((*x'*,*y*)#*xs*) *x* = *y*
x ≠ *x'* ⟹ *permutation-of-list* ((*x'*,*y'*)#*xs*) *x* = *permutation-of-list xs x*
⟨*proof*⟩

lemma *eval-inverse-permutation-of-list* [*simp*]:
inverse-permutation-of-list [] *x* = *x*
x = *x'* ⟹ *inverse-permutation-of-list* ((*y*,*x'*)#*xs*) *x* = *y*
x ≠ *x'* ⟹ *inverse-permutation-of-list* ((*y'*,*x'*)#*xs*) *x* = *inverse-permutation-of-list xs x*
⟨*proof*⟩

lemma *permutation-of-list-id*: *x* ∉ *set* (*map fst xs*) ⟹ *permutation-of-list xs x* = *x*
⟨*proof*⟩

lemma *permutation-of-list-unique'*:
distinct (*map fst xs*) ⟹ (*x*, *y*) ∈ *set xs* ⟹ *permutation-of-list xs x* = *y*
⟨*proof*⟩

lemma *permutation-of-list-unique*:
list-permutes xs A ⟹ (*x*, *y*) ∈ *set xs* ⟹ *permutation-of-list xs x* = *y*
⟨*proof*⟩

lemma *inverse-permutation-of-list-id*:
x ∉ *set* (*map snd xs*) ⟹ *inverse-permutation-of-list xs x* = *x*
⟨*proof*⟩

lemma *inverse-permutation-of-list-unique'*:
distinct (*map snd xs*) ⟹ (*x*, *y*) ∈ *set xs* ⟹ *inverse-permutation-of-list xs y* = *x*
⟨*proof*⟩

lemma *inverse-permutation-of-list-unique*:
list-permutes xs A ⟹ (*x*,*y*) ∈ *set xs* ⟹ *inverse-permutation-of-list xs y* = *x*
⟨*proof*⟩

lemma *inverse-permutation-of-list-correct*:
fixes *A* :: *'a set*
assumes *list-permutes xs A*
shows *inverse-permutation-of-list xs* = *inv* (*permutation-of-list xs*)
⟨*proof*⟩

end

60 Permutations of a Multiset

theory *Multiset-Permutations*
imports
Complex-Main
Multiset
Permutations
begin

lemma *mset-tl*: $xs \neq [] \Longrightarrow mset\ (tl\ xs) = mset\ xs - \{\#hd\ xs\#\}$
⟨*proof*⟩

lemma *mset-set-image-inj*:
assumes *inj-on f A*
shows *mset-set* $(f\ `\ A) =$ *image-mset f* (*mset-set A*)
⟨*proof*⟩

lemma *multiset-remove-induct* [*case-names empty remove*]:
assumes $P\ \{\#\}\ \bigwedge A.\ A \neq \{\#\} \Longrightarrow (\bigwedge x.\ x \in\# A \Longrightarrow P\ (A - \{\#x\#\})) \Longrightarrow P\ A$
shows *P A*
⟨*proof*⟩

lemma *map-list-bind*: *map g* (*List.bind xs f*) = *List.bind xs* (*map g* ∘ *f*)
⟨*proof*⟩

lemma *mset-eq-mset-set-imp-distinct*:
finite A ⟹ *mset-set A* = *mset xs* ⟹ *distinct xs*
⟨*proof*⟩

60.1 Permutations of a multiset

definition *permutations-of-multiset* :: *'a multiset* ⇒ *'a list set* **where**
permutations-of-multiset A = {*xs*. *mset xs* = *A*}

lemma *permutations-of-multisetI*: *mset xs* = *A* ⟹ *xs* ∈ *permutations-of-multiset A*
⟨*proof*⟩

lemma *permutations-of-multisetD*: *xs* ∈ *permutations-of-multiset A* ⟹ *mset xs* = *A*
⟨*proof*⟩

lemma *permutations-of-multiset-Cons-iff*:
x # xs ∈ *permutations-of-multiset A* ⟷ *x* ∈# *A* ∧ *xs* ∈ *permutations-of-multiset* (*A* − {#*x*#})
⟨*proof*⟩

lemma *permutations-of-multiset-empty* [*simp*]: *permutations-of-multiset* {#} = {[]}
⟨*proof*⟩

lemma *permutations-of-multiset-nonempty*:
assumes *nonempty*: *A* ≠ {#}
shows *permutations-of-multiset A* =
(⋃*x*∈*set-mset A*. (*op # x*) ` *permutations-of-multiset* (*A* − {#*x*#}))
(**is** - = *?rhs*)

⟨*proof*⟩

lemma *permutations-of-multiset-singleton* [*simp*]: *permutations-of-multiset* {#*x*#} = {[*x*]}
⟨*proof*⟩

lemma *permutations-of-multiset-doubleton*:
permutations-of-multiset {#*x*,*y*#} = {[*x*,*y*], [*y*,*x*]}
⟨*proof*⟩

lemma *rev-permutations-of-multiset* [*simp*]:
rev ' *permutations-of-multiset A* = *permutations-of-multiset A*
⟨*proof*⟩

lemma *length-finite-permutations-of-multiset*:
xs ∈ *permutations-of-multiset A* ⟹ *length xs* = *size A*
⟨*proof*⟩

lemma *permutations-of-multiset-lists*: *permutations-of-multiset A* ⊆ *lists* (*set-mset A*)
⟨*proof*⟩

lemma *finite-permutations-of-multiset* [*simp*]: *finite* (*permutations-of-multiset A*)
⟨*proof*⟩

lemma *permutations-of-multiset-not-empty* [*simp*]: *permutations-of-multiset A* ≠ {}
⟨*proof*⟩

lemma *permutations-of-multiset-image*:
permutations-of-multiset (*image-mset f A*) = *map f* ' *permutations-of-multiset A*
⟨*proof*⟩

60.2 Cardinality of permutations

In this section, we prove some basic facts about the number of permutations of a multiset.

context
begin

private lemma *multiset-prod-fact-insert*:
(∏ *y*∈*set-mset* (*A*+{#*x*#}). *fact* (*count* (*A*+{#*x*#}) *y*)) =
(*count A x* + *1*) * (∏ *y*∈*set-mset A*. *fact* (*count A y*))
⟨*proof*⟩ **lemma** *multiset-prod-fact-remove*:
x ∈# *A* ⟹ (∏ *y*∈*set-mset A*. *fact* (*count A y*)) =
count A x * (∏ *y*∈*set-mset* (*A*−{#*x*#}). *fact* (*count* (*A*−{#*x*#}) *y*))
⟨*proof*⟩

lemma *card-permutations-of-multiset-aux*:
card (*permutations-of-multiset A*) ∗ (∏ *x*∈*set-mset A*. *fact* (*count A x*)) = *fact* (*size A*)
⟨*proof*⟩

theorem *card-permutations-of-multiset*:
card (*permutations-of-multiset A*) = *fact* (*size A*) *div* (∏ *x*∈*set-mset A*. *fact* (*count A x*))
(∏ *x*∈*set-mset A*. *fact* (*count A x*) :: *nat*) *dvd fact* (*size A*)
⟨*proof*⟩

lemma *card-permutations-of-multiset-insert-aux*:
card (*permutations-of-multiset* (*A* + {#*x*#})) ∗ (*count A x* + *1*) =
(*size A* + *1*) ∗ *card* (*permutations-of-multiset A*)
⟨*proof*⟩

lemma *card-permutations-of-multiset-remove-aux*:
assumes *x* ∈# *A*
shows *card* (*permutations-of-multiset A*) ∗ *count A x* =
size A ∗ *card* (*permutations-of-multiset* (*A* − {#*x*#}))
⟨*proof*⟩

lemma *real-card-permutations-of-multiset-remove*:
assumes *x* ∈# *A*
shows *real* (*card* (*permutations-of-multiset* (*A* − {#*x*#}))) =
real (*card* (*permutations-of-multiset A*) ∗ *count A x*) / *real* (*size A*)
⟨*proof*⟩

lemma *real-card-permutations-of-multiset-remove′*:
assumes *x* ∈# *A*
shows *real* (*card* (*permutations-of-multiset A*)) =
real (*size A* ∗ *card* (*permutations-of-multiset* (*A* − {#*x*#}))) / *real* (*count A x*)
⟨*proof*⟩

end

60.3 Permutations of a set

definition *permutations-of-set* :: *'a set* ⇒ *'a list set* **where**
permutations-of-set A = {*xs*. *set xs* = *A* ∧ *distinct xs*}

lemma *permutations-of-set-altdef*:
finite A ⟹ *permutations-of-set A* = *permutations-of-multiset* (*mset-set A*)
⟨*proof*⟩

lemma *permutations-of-setI* [*intro*]:
assumes *set xs* = *A distinct xs*
shows *xs* ∈ *permutations-of-set A*

⟨*proof*⟩

lemma *permutations-of-setD*:
assumes $xs \in$ *permutations-of-set* A
shows *set* $xs = A$ *distinct* xs
⟨*proof*⟩

lemma *permutations-of-set-lists*: *permutations-of-set* $A \subseteq$ *lists* A
⟨*proof*⟩

lemma *permutations-of-set-empty* [*simp*]: *permutations-of-set* $\{\} = \{[]\}$
⟨*proof*⟩

lemma *UN-set-permutations-of-set* [*simp*]:
finite $A \Longrightarrow (\bigcup xs \in$ *permutations-of-set* A. *set* $xs) = A$
⟨*proof*⟩

lemma *permutations-of-set-infinite*:
$\neg$*finite* $A \Longrightarrow$ *permutations-of-set* $A = \{\}$
⟨*proof*⟩

lemma *permutations-of-set-nonempty*:
$A \neq \{\} \Longrightarrow$ *permutations-of-set* $A =$
$(\bigcup x \in A.\ (\lambda xs.\ x\ \#\ xs)\ `$ *permutations-of-set* $(A - \{x\}))$
⟨*proof*⟩

lemma *permutations-of-set-singleton* [*simp*]: *permutations-of-set* $\{x\} = \{[x]\}$
⟨*proof*⟩

lemma *permutations-of-set-doubleton*:
$x \neq y \Longrightarrow$ *permutations-of-set* $\{x,y\} = \{[x,y], [y,x]\}$
⟨*proof*⟩

lemma *rev-permutations-of-set* [*simp*]:
rev ` *permutations-of-set* A = *permutations-of-set* A
⟨*proof*⟩

lemma *length-finite-permutations-of-set*:
$xs \in$ *permutations-of-set* $A \Longrightarrow$ *length* xs = *card* A
⟨*proof*⟩

lemma *finite-permutations-of-set* [*simp*]: *finite* (*permutations-of-set* A)
⟨*proof*⟩

lemma *permutations-of-set-empty-iff* [*simp*]:
permutations-of-set $A = \{\} \longleftrightarrow \neg$*finite* A
⟨*proof*⟩

lemma *card-permutations-of-set* [*simp*]:

finite A $\Longrightarrow$ *card* (*permutations-of-set A*) = *fact* (*card A*)
⟨*proof*⟩

lemma *permutations-of-set-image-inj*:
assumes *inj*: *inj-on f A*
shows *permutations-of-set* (*f* ‘ *A*) = *map f* ‘ *permutations-of-set A*
⟨*proof*⟩

lemma *permutations-of-set-image-permutes*:
σ permutes A $\Longrightarrow$ *map σ* ‘ *permutations-of-set A* = *permutations-of-set A*
⟨*proof*⟩

60.4 Code generation

First, we give code an implementation for permutations of lists.

declare *length-remove1* [*termination-simp*]

fun *permutations-of-list-impl* **where**
permutations-of-list-impl xs = (*if xs* = [] *then* [[]] *else*
List.bind (*remdups xs*) (λ*x*. *map* (*op* # *x*) (*permutations-of-list-impl* (*remove1 x xs*))))

fun *permutations-of-list-impl-aux* **where**
permutations-of-list-impl-aux acc xs = (*if xs* = [] *then* [*acc*] *else*
List.bind (*remdups xs*) (λ*x*. *permutations-of-list-impl-aux* (*x*#*acc*) (*remove1 x xs*)))

declare *permutations-of-list-impl-aux.simps* [*simp del*]
declare *permutations-of-list-impl.simps* [*simp del*]

lemma *permutations-of-list-impl-Nil* [*simp*]:
permutations-of-list-impl [] = [[]]
⟨*proof*⟩

lemma *permutations-of-list-impl-nonempty*:
xs ≠ [] $\Longrightarrow$ *permutations-of-list-impl xs* =
List.bind (*remdups xs*) (λ*x*. *map* (*op* # *x*) (*permutations-of-list-impl* (*remove1 x xs*)))
⟨*proof*⟩

lemma *set-permutations-of-list-impl*:
set (*permutations-of-list-impl xs*) = *permutations-of-multiset* (*mset xs*)
⟨*proof*⟩

lemma *distinct-permutations-of-list-impl*:
distinct (*permutations-of-list-impl xs*)
⟨*proof*⟩

lemma *permutations-of-list-impl-aux-correct′*:

permutations-of-list-impl-aux acc xs =
map (λ*xs*. *rev xs* @ *acc*) (*permutations-of-list-impl xs*)
⟨*proof*⟩

lemma *permutations-of-list-impl-aux-correct*:
permutations-of-list-impl-aux [] *xs* = *map rev* (*permutations-of-list-impl xs*)
⟨*proof*⟩

lemma *distinct-permutations-of-list-impl-aux*:
distinct (*permutations-of-list-impl-aux acc xs*)
⟨*proof*⟩

lemma *set-permutations-of-list-impl-aux*:
set (*permutations-of-list-impl-aux* [] *xs*) = *permutations-of-multiset* (*mset xs*)
⟨*proof*⟩

declare *set-permutations-of-list-impl-aux* [*symmetric*, *code*]

value [*code*] *permutations-of-multiset* {#*1*,*2*,*3*,*4*::*int*#}

Now we turn to permutations of sets. We define an auxiliary version with an accumulator to avoid having to map over the results.

function *permutations-of-set-aux* **where**
permutations-of-set-aux acc A =
(*if* ¬*finite A then* {} *else if A* = {} *then* {*acc*} *else*
(⋃*x*∈*A*. *permutations-of-set-aux* (*x*#*acc*) (*A* − {*x*})))
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *permutations-of-set-aux-altdef*:
permutations-of-set-aux acc A = (λ*xs*. *rev xs* @ *acc*) ‘ *permutations-of-set A*
⟨*proof*⟩

declare *permutations-of-set-aux.simps* [*simp del*]

lemma *permutations-of-set-aux-correct*:
permutations-of-set-aux [] *A* = *permutations-of-set A*
⟨*proof*⟩

In another refinement step, we define a version on lists.

declare *length-remove1* [*termination-simp*]

fun *permutations-of-set-aux-list* **where**
permutations-of-set-aux-list acc xs =
(*if xs* = [] *then* [*acc*] *else*
List.bind xs (λ*x*. *permutations-of-set-aux-list* (*x*#*acc*) (*List.remove1 x xs*)))

definition *permutations-of-set-list* **where**
permutations-of-set-list xs = *permutations-of-set-aux-list* [] *xs*

declare *permutations-of-set-aux-list.simps* [*simp del*]

lemma *permutations-of-set-aux-list-refine*:
assumes *distinct xs*
shows *set* (*permutations-of-set-aux-list acc xs*) = *permutations-of-set-aux acc* (*set xs*)
⟨*proof*⟩

The permutation lists contain no duplicates if the inputs contain no duplicates. Therefore, these functions can easily be used when working with a representation of sets by distinct lists. The same approach should generalise to any kind of set implementation that supports a monadic bind operation, and since the results are disjoint, merging should be cheap.

lemma *distinct-permutations-of-set-aux-list*:
distinct xs ⟹ *distinct* (*permutations-of-set-aux-list acc xs*)
⟨*proof*⟩

lemma *distinct-permutations-of-set-list*:
distinct xs ⟹ *distinct* (*permutations-of-set-list xs*)
⟨*proof*⟩

lemma *permutations-of-list*:
permutations-of-set (*set xs*) = *set* (*permutations-of-set-list* (*remdups xs*))
⟨*proof*⟩

lemma *permutations-of-list-code* [*code*]:
permutations-of-set (*set xs*) = *set* (*permutations-of-set-list* (*remdups xs*))
permutations-of-set (*List.coset xs*) =
Code.abort (*STR ''Permutation of set complement not supported''*)
(λ-. *permutations-of-set* (*List.coset xs*))
⟨*proof*⟩

value [*code*] *permutations-of-set* (*set ''abcd''*)

end

61 Non-negative, non-positive integers and reals

theory *Nonpos-Ints*
imports *Complex-Main*
begin

61.1 Non-positive integers

The set of non-positive integers on a ring. (in analogy to the set of non-negative integers ℕ) This is useful e.g. for the Gamma function.

definition *nonpos-Ints* ($\mathbb{Z}_{\leq 0}$) **where** $\mathbb{Z}_{\leq 0}$ = {*of-int n* |*n*. *n* ≤ *0*}

lemma *zero-in-nonpos-Ints* [*simp*,*intro*]: $0 \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *neg-one-in-nonpos-Ints* [*simp*,*intro*]: $-1 \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *neg-numeral-in-nonpos-Ints* [*simp*,*intro*]: $-numeral\ n \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *one-notin-nonpos-Ints* [*simp*]: $(1 :: 'a :: ring\text{-}char\text{-}0) \notin \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *numeral-notin-nonpos-Ints* [*simp*]: $(numeral\ n :: 'a :: ring\text{-}char\text{-}0) \notin \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *minus-of-nat-in-nonpos-Ints* [*simp*, *intro*]: $-\ of\text{-}nat\ n \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *of-nat-in-nonpos-Ints-iff*: $(of\text{-}nat\ n :: 'a :: \{ring\text{-}1, ring\text{-}char\text{-}0\}) \in \mathbb{Z}_{\leq 0}$
$\longleftrightarrow n = 0$
⟨*proof*⟩

lemma *nonpos-Ints-of-int*: $n \leq 0 \Longrightarrow of\text{-}int\ n \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-IntsI*:
$x \in \mathbb{Z} \Longrightarrow x \leq 0 \Longrightarrow (x :: 'a :: linordered\text{-}idom) \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Ints-subset-Ints*: $\mathbb{Z}_{\leq 0} \subseteq \mathbb{Z}$
⟨*proof*⟩

lemma *nonpos-Ints-nonpos* [*dest*]: $x \in \mathbb{Z}_{\leq 0} \Longrightarrow x \leq (0 :: 'a :: linordered\text{-}idom)$
⟨*proof*⟩

lemma *nonpos-Ints-Int* [*dest*]: $x \in \mathbb{Z}_{\leq 0} \Longrightarrow x \in \mathbb{Z}$
⟨*proof*⟩

lemma *nonpos-Ints-cases*:
assumes $x \in \mathbb{Z}_{\leq 0}$
obtains n **where** $x = of\text{-}int\ n\ n \leq 0$
⟨*proof*⟩

lemma *nonpos-Ints-cases'*:
assumes $x \in \mathbb{Z}_{\leq 0}$
obtains n **where** $x = -of\text{-}nat\ n$
⟨*proof*⟩

lemma *of-real-in-nonpos-Ints-iff*: $(\mathit{of\text{-}real}\ x :: 'a :: \mathit{real\text{-}algebra\text{-}1}) \in \mathbb{Z}_{\leq 0} \longleftrightarrow x \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Ints-altdef*: $\mathbb{Z}_{\leq 0} = \{n \in \mathbb{Z}.\ (n :: 'a :: \mathit{linordered\text{-}idom}) \leq 0\}$
⟨*proof*⟩

lemma *uminus-in-Nats-iff*: $-x \in \mathbb{N} \longleftrightarrow x \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *uminus-in-nonpos-Ints-iff*: $-x \in \mathbb{Z}_{\leq 0} \longleftrightarrow x \in \mathbb{N}$
⟨*proof*⟩

lemma *nonpos-Ints-mult*: $x \in \mathbb{Z}_{\leq 0} \Longrightarrow y \in \mathbb{Z}_{\leq 0} \Longrightarrow x * y \in \mathbb{N}$
⟨*proof*⟩

lemma *Nats-mult-nonpos-Ints*: $x \in \mathbb{N} \Longrightarrow y \in \mathbb{Z}_{\leq 0} \Longrightarrow x * y \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Ints-mult-Nats*:
$x \in \mathbb{Z}_{\leq 0} \Longrightarrow y \in \mathbb{N} \Longrightarrow x * y \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Ints-add*:
$x \in \mathbb{Z}_{\leq 0} \Longrightarrow y \in \mathbb{Z}_{\leq 0} \Longrightarrow x + y \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Ints-diff-Nats*:
$x \in \mathbb{Z}_{\leq 0} \Longrightarrow y \in \mathbb{N} \Longrightarrow x - y \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

lemma *Nats-diff-nonpos-Ints*:
$x \in \mathbb{N} \Longrightarrow y \in \mathbb{Z}_{\leq 0} \Longrightarrow x - y \in \mathbb{N}$
⟨*proof*⟩

lemma *plus-of-nat-eq-0-imp*: $z + \mathit{of\text{-}nat}\ n = 0 \Longrightarrow z \in \mathbb{Z}_{\leq 0}$
⟨*proof*⟩

61.2 Non-negative reals

definition *nonneg-Reals* :: $'a$::*real-algebra-1 set* $(\mathbb{R}_{\geq 0})$
where $\mathbb{R}_{\geq 0} = \{\mathit{of\text{-}real}\ r \mid r.\ r \geq 0\}$

lemma *nonneg-Reals-of-real-iff* [*simp*]: $\mathit{of\text{-}real}\ r \in \mathbb{R}_{\geq 0} \longleftrightarrow r \geq 0$
⟨*proof*⟩

lemma *nonneg-Reals-subset-Reals*: $\mathbb{R}_{\geq 0} \subseteq \mathbb{R}$
⟨*proof*⟩

lemma *nonneg-Reals-Real* [*dest*]: $x \in \mathbb{R}_{\geq 0} \Longrightarrow x \in \mathbb{R}$
⟨*proof*⟩

lemma *nonneg-Reals-of-nat-I* [*simp*]: *of-nat n* $\in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-cases*:
assumes $x \in \mathbb{R}_{\geq 0}$
obtains *r* **where** $x =$ *of-real r* $r \geq 0$
⟨*proof*⟩

lemma *nonneg-Reals-zero-I* [*simp*]: $0 \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-one-I* [*simp*]: $1 \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-minus-one-I* [*simp*]: $-1 \notin \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-numeral-I* [*simp*]: *numeral w* $\in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-minus-numeral-I* [*simp*]: $-$ *numeral w* $\notin \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-add-I* [*simp*]: $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a + b \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-mult-I* [*simp*]: $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a * b \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-inverse-I* [*simp*]:
fixes *a* :: *'a*::*real-div-algebra*
shows $a \in \mathbb{R}_{\geq 0} \Longrightarrow$ *inverse a* $\in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-divide-I* [*simp*]:
fixes *a* :: *'a*::*real-div-algebra*
shows $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a\ /\ b \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonneg-Reals-pow-I* [*simp*]: $a \in \mathbb{R}_{\geq 0} \Longrightarrow a\hat{}n \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *complex-nonneg-Reals-iff*: $z \in \mathbb{R}_{\geq 0} \longleftrightarrow$ *Re z* $\geq 0 \wedge$ *Im z* $= 0$
⟨*proof*⟩

lemma *ii-not-nonneg-Reals* [*iff*]: $\mathrm{i} \notin \mathbb{R}_{\geq 0}$

⟨*proof*⟩

61.3 Non-positive reals

definition *nonpos-Reals* :: *'a::real-algebra-1 set* ($\mathbb{R}_{\leq 0}$)
where $\mathbb{R}_{\leq 0} = \{of\text{-}real\ r \mid r.\ r \leq 0\}$

lemma *nonpos-Reals-of-real-iff* [*simp*]: $of\text{-}real\ r \in \mathbb{R}_{\leq 0} \longleftrightarrow r \leq 0$
⟨*proof*⟩

lemma *nonpos-Reals-subset-Reals*: $\mathbb{R}_{\leq 0} \subseteq \mathbb{R}$
⟨*proof*⟩

lemma *nonpos-Ints-subset-nonpos-Reals*: $\mathbb{Z}_{\leq 0} \subseteq \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-of-nat-iff* [*simp*]: $of\text{-}nat\ n \in \mathbb{R}_{\leq 0} \longleftrightarrow n=0$
⟨*proof*⟩

lemma *nonpos-Reals-Real* [*dest*]: $x \in \mathbb{R}_{\leq 0} \Longrightarrow x \in \mathbb{R}$
⟨*proof*⟩

lemma *nonpos-Reals-cases*:
assumes $x \in \mathbb{R}_{\leq 0}$
obtains r **where** $x = of\text{-}real\ r\ r \leq 0$
⟨*proof*⟩

lemma *uminus-nonneg-Reals-iff* [*simp*]: $-x \in \mathbb{R}_{\geq 0} \longleftrightarrow x \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *uminus-nonpos-Reals-iff* [*simp*]: $-x \in \mathbb{R}_{\leq 0} \longleftrightarrow x \in \mathbb{R}_{\geq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-zero-I* [*simp*]: $0 \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-one-I* [*simp*]: $1 \notin \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-numeral-I* [*simp*]: $numeral\ w \notin \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-add-I* [*simp*]: $[\![a \in \mathbb{R}_{\leq 0};\ b \in \mathbb{R}_{\leq 0}]\!] \Longrightarrow a + b \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-mult-I1*: $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\leq 0}]\!] \Longrightarrow a * b \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-mult-I2*: $[\![a \in \mathbb{R}_{\leq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a * b \in \mathbb{R}_{\leq 0}$

⟨*proof*⟩

lemma *nonpos-Reals-mult-of-nat-iff*:
fixes *a*:: *'a* :: *real-div-algebra* **shows** $a * \mathit{of\text{-}nat}\ n \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0} \vee n{=}0$
⟨*proof*⟩

lemma *nonpos-Reals-inverse-I*:
fixes *a* :: *'a*::*real-div-algebra*
shows $a \in \mathbb{R}_{\leq 0} \Longrightarrow \mathit{inverse}\ a \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-divide-I1*:
fixes *a* :: *'a*::*real-div-algebra*
shows $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\leq 0}]\!] \Longrightarrow a\ /\ b \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-divide-I2*:
fixes *a* :: *'a*::*real-div-algebra*
shows $[\![a \in \mathbb{R}_{\leq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a\ /\ b \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *nonpos-Reals-divide-of-nat-iff*:
fixes *a*:: *'a* :: *real-div-algebra* **shows** $a\ /\ \mathit{of\text{-}nat}\ n \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0} \vee n{=}0$
⟨*proof*⟩

lemma *nonpos-Reals-pow-I*: $[\![a \in \mathbb{R}_{\leq 0};\ \mathit{odd}\ n]\!] \Longrightarrow a\hat{}n \in \mathbb{R}_{\leq 0}$
⟨*proof*⟩

lemma *complex-nonpos-Reals-iff*: $z \in \mathbb{R}_{\leq 0} \longleftrightarrow \mathit{Re}\ z \leq 0 \wedge \mathit{Im}\ z = 0$
⟨*proof*⟩

lemma *ii-not-nonpos-Reals* [*iff*]: $\mathrm{i} \notin \mathbb{R}_{\leq 0}$
⟨*proof*⟩

end

62 A generic phantom type

theory *Phantom-Type*
imports *Main*
begin

datatype (*'a*, *'b*) *phantom* = *phantom* (*of-phantom*: *'b*)

lemma *type-definition-phantom'*: *type-definition of-phantom phantom UNIV*
⟨*proof*⟩

lemma *phantom-comp-of-phantom* [*simp*]: *phantom* ∘ *of-phantom* = *id*
and *of-phantom-comp-phantom* [*simp*]: *of-phantom* ∘ *phantom* = *id*

⟨*proof*⟩

syntax -*Phantom* :: *type* ⇒ *logic* ((*1Phantom*/(*1* '(-'))))
translations
Phantom('*t*) => *CONST phantom* :: - ⇒ ('*t*, -) *phantom*

⟨*ML*⟩

lemma *of-phantom-inject* [*simp*]:
of-phantom x = *of-phantom y* ⟷ *x* = *y*
⟨*proof*⟩

end

63 Cardinality of types

theory *Cardinality*
imports *Phantom-Type*
begin

63.1 Preliminary lemmas

lemma (**in** *type-definition*) *univ*:
UNIV = *Abs* ' *A*
⟨*proof*⟩

lemma (**in** *type-definition*) *card*: *card* (*UNIV* :: '*b set*) = *card A*
⟨*proof*⟩

lemma *finite-range-Some*: *finite* (*range* (*Some* :: '*a* ⇒ '*a option*)) = *finite* (*UNIV* :: '*a set*)
⟨*proof*⟩

lemma *infinite-literal*: ¬ *finite* (*UNIV* :: *String.literal set*)
⟨*proof*⟩

63.2 Cardinalities of types

syntax -*type-card* :: *type* => *nat* ((*1CARD*/(*1* '(-'))))

translations *CARD*('*t*) => *CONST card* (*CONST UNIV* :: '*t set*)

⟨*ML*⟩

lemma *card-prod* [*simp*]: *CARD*('*a* × '*b*) = *CARD*('*a*) ∗ *CARD*('*b*)
⟨*proof*⟩

lemma *card-UNIV-sum*: *CARD*('*a* + '*b*) = (*if CARD*('*a*) ≠ *0* ∧ *CARD*('*b*) ≠ *0* *then CARD*('*a*) + *CARD*('*b*) *else 0*)

⟨*proof*⟩

lemma *card-sum* [*simp*]: *CARD*('*a* + '*b*) = *CARD*('*a*::*finite*) + *CARD*('*b*::*finite*)
⟨*proof*⟩

lemma *card-UNIV-option*: *CARD*('*a option*) = (*if CARD*('*a*) = *0 then 0 else CARD*('*a*) + *1*)
⟨*proof*⟩

lemma *card-option* [*simp*]: *CARD*('*a option*) = *Suc CARD*('*a*::*finite*)
⟨*proof*⟩

lemma *card-UNIV-set*: *CARD*('*a set*) = (*if CARD*('*a*) = *0 then 0 else 2* ^ *CARD*('*a*))
⟨*proof*⟩

lemma *card-set* [*simp*]: *CARD*('*a set*) = *2* ^ *CARD*('*a*::*finite*)
⟨*proof*⟩

lemma *card-nat* [*simp*]: *CARD*(*nat*) = *0*
⟨*proof*⟩

lemma *card-fun*: *CARD*('*a* ⇒ '*b*) = (*if CARD*('*a*) ≠ *0* ∧ *CARD*('*b*) ≠ *0* ∨ *CARD*('*b*) = *1 then CARD*('*b*) ^ *CARD*('*a*) *else 0*)
⟨*proof*⟩

corollary *finite-UNIV-fun*:
finite (*UNIV* :: ('*a* ⇒ '*b*) *set*) ⟷
finite (*UNIV* :: '*a set*) ∧ *finite* (*UNIV* :: '*b set*) ∨ *CARD*('*b*) = *1*
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *card-literal*: *CARD*(*String.literal*) = *0*
⟨*proof*⟩

63.3 Classes with at least 1 and 2

Class finite already captures "at least 1"

lemma *zero-less-card-finite* [*simp*]: *0* < *CARD*('*a*::*finite*)
⟨*proof*⟩

lemma *one-le-card-finite* [*simp*]: *Suc 0* ≤ *CARD*('*a*::*finite*)
⟨*proof*⟩

Class for cardinality "at least 2"

class *card2* = *finite* +
assumes *two-le-card*: *2* ≤ *CARD*('*a*)

lemma *one-less-card*: *Suc 0* < *CARD*('*a*::*card2*)
⟨*proof*⟩

lemma *one-less-int-card*: *1 < int CARD('a::card2)*
⟨*proof*⟩

63.4 A type class for deciding finiteness of types

type-synonym *'a finite-UNIV = ('a, bool) phantom*

class *finite-UNIV* =
fixes *finite-UNIV* :: *('a, bool) phantom*
assumes *finite-UNIV*: *finite-UNIV = Phantom('a) (finite (UNIV :: 'a set))*

lemma *finite-UNIV-code* [*code-unfold*]:
finite (UNIV :: 'a :: finite-UNIV set)
⟷ of-phantom (finite-UNIV :: 'a finite-UNIV)
⟨*proof*⟩

63.5 A type class for computing the cardinality of types

definition *is-list-UNIV :: 'a list ⇒ bool*
where *is-list-UNIV xs = (let c = CARD('a) in if c = 0 then False else size (remdups xs) = c)*

lemma *is-list-UNIV-iff*: *is-list-UNIV xs ⟷ set xs = UNIV*
⟨*proof*⟩

type-synonym *'a card-UNIV = ('a, nat) phantom*

class *card-UNIV = finite-UNIV +*
fixes *card-UNIV :: 'a card-UNIV*
assumes *card-UNIV*: *card-UNIV = Phantom('a) CARD('a)*

63.6 Instantiations for *card-UNIV*

instantiation *nat :: card-UNIV* **begin**
definition *finite-UNIV = Phantom(nat) False*
definition *card-UNIV = Phantom(nat) 0*
instance ⟨*proof*⟩
end

instantiation *int :: card-UNIV* **begin**
definition *finite-UNIV = Phantom(int) False*
definition *card-UNIV = Phantom(int) 0*
instance ⟨*proof*⟩
end

instantiation *natural :: card-UNIV* **begin**
definition *finite-UNIV = Phantom(natural) False*
definition *card-UNIV = Phantom(natural) 0*
instance

⟨*proof*⟩
end

instantiation *integer* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*integer*) *False*
definition *card-UNIV* = *Phantom*(*integer*) *0*
instance
⟨*proof*⟩
end

instantiation *list* :: (*type*) *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a list*) *False*
definition *card-UNIV* = *Phantom*('*a list*) *0*
instance ⟨*proof*⟩
end

instantiation *unit* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*unit*) *True*
definition *card-UNIV* = *Phantom*(*unit*) *1*
instance ⟨*proof*⟩
end

instantiation *bool* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*bool*) *True*
definition *card-UNIV* = *Phantom*(*bool*) *2*
instance ⟨*proof*⟩
end

instantiation *char* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*char*) *True*
definition *card-UNIV* = *Phantom*(*char*) *256*
instance ⟨*proof*⟩
end

instantiation *prod* :: (*finite-UNIV*, *finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a* × '*b*)
(*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*) ∧ *of-phantom* (*finite-UNIV* :: '*b finite-UNIV*))
instance ⟨*proof*⟩
end

instantiation *prod* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a* × '*b*)
(*of-phantom* (*card-UNIV* :: '*a card-UNIV*) ∗ *of-phantom* (*card-UNIV* :: '*b card-UNIV*))
instance ⟨*proof*⟩
end

instantiation *sum* :: (*finite-UNIV*, *finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a* + '*b*)

(*of-phantom* (*finite-UNIV* :: *'a finite-UNIV*) ∧ *of-phantom* (*finite-UNIV* :: *'b finite-UNIV*))
instance
⟨*proof*⟩
end

instantiation *sum* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*(*'a* + *'b*)
(*let ca* = *of-phantom* (*card-UNIV* :: *'a card-UNIV*);
cb = *of-phantom* (*card-UNIV* :: *'b card-UNIV*)
in if ca ≠ *0* ∧ *cb* ≠ *0 then ca* + *cb else 0*)
instance ⟨*proof*⟩
end

instantiation *fun* :: (*finite-UNIV*, *card-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*'a* ⇒ *'b*)
(*let cb* = *of-phantom* (*card-UNIV* :: *'b card-UNIV*)
in cb = *1* ∨ *of-phantom* (*finite-UNIV* :: *'a finite-UNIV*) ∧ *cb* ≠ *0*)
instance
⟨*proof*⟩
end

instantiation *fun* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*(*'a* ⇒ *'b*)
(*let ca* = *of-phantom* (*card-UNIV* :: *'a card-UNIV*);
cb = *of-phantom* (*card-UNIV* :: *'b card-UNIV*)
in if ca ≠ *0* ∧ *cb* ≠ *0* ∨ *cb* = *1 then cb* ^ *ca else 0*)
instance ⟨*proof*⟩
end

instantiation *option* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*'a option*) (*of-phantom* (*finite-UNIV* :: *'a finite-UNIV*))
instance ⟨*proof*⟩
end

instantiation *option* :: (*card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*(*'a option*)
(*let c* = *of-phantom* (*card-UNIV* :: *'a card-UNIV*) *in if c* ≠ *0 then Suc c else 0*)
instance ⟨*proof*⟩
end

instantiation *String.literal* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*String.literal*) *False*
definition *card-UNIV* = *Phantom*(*String.literal*) *0*
instance
⟨*proof*⟩
end

instantiation *set* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a set*) (*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*))
instance ⟨*proof*⟩
end

instantiation *set* :: (*card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a set*)
(*let c* = *of-phantom* (*card-UNIV* :: '*a card-UNIV*) *in if c* = *0 then 0 else 2* ^ *c*)
instance ⟨*proof*⟩
end

lemma *UNIV-finite-1*: *UNIV* = *set* [*finite-1*.a_1]
⟨*proof*⟩

lemma *UNIV-finite-2*: *UNIV* = *set* [*finite-2*.a_1, *finite-2*.a_2]
⟨*proof*⟩

lemma *UNIV-finite-3*: *UNIV* = *set* [*finite-3*.a_1, *finite-3*.a_2, *finite-3*.a_3]
⟨*proof*⟩

lemma *UNIV-finite-4*: *UNIV* = *set* [*finite-4*.a_1, *finite-4*.a_2, *finite-4*.a_3, *finite-4*.a_4]
⟨*proof*⟩

lemma *UNIV-finite-5*:
UNIV = *set* [*finite-5*.a_1, *finite-5*.a_2, *finite-5*.a_3, *finite-5*.a_4, *finite-5*.a_5]
⟨*proof*⟩

instantiation *Enum.finite-1* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-1*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-1*) *1*
instance
⟨*proof*⟩
end

instantiation *Enum.finite-2* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-2*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-2*) *2*
instance
⟨*proof*⟩
end

instantiation *Enum.finite-3* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-3*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-3*) *3*
instance
⟨*proof*⟩
end

instantiation *Enum.finite-4* :: *card-UNIV* **begin**

definition *finite-UNIV* = *Phantom*(*Enum.finite-4*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-4*) *4*
instance
⟨*proof*⟩
end

instantiation *Enum.finite-5* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-5*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-5*) *5*
instance
⟨*proof*⟩
end

63.7 Code setup for sets

Implement *CARD*('*a*) via *card-UNIV-class.card-UNIV* and provide implementations for *finite*, *card*, *op* ⊆, and *op* =if the calling context already provides *finite-UNIV* and *card-UNIV* instances. If we implemented the latter always via *card-UNIV-class.card-UNIV*, we would require instances of essentially all element types, i.e., a lot of instantiation proofs and – at run time – possibly slow dictionary constructions.

context
begin

qualified definition *card-UNIV'* :: '*a card-UNIV*
where [*code del*]: *card-UNIV'* = *Phantom*('*a*) *CARD*('*a*)

lemma *CARD-code* [*code-unfold*]:
CARD('*a*) = *of-phantom* (*card-UNIV'* :: '*a card-UNIV*)
⟨*proof*⟩

lemma *card-UNIV'-code* [*code*]:
card-UNIV' = *card-UNIV*
⟨*proof*⟩

end

lemma *card-Compl*:
finite A ⟹ *card* (− *A*) = *card* (*UNIV* :: '*a set*) − *card* (*A* :: '*a set*)
⟨*proof*⟩

context fixes *xs* :: '*a* :: *finite-UNIV list*
begin

qualified definition *finite'* :: '*a set* ⇒ *bool*
where [*simp*, *code del*, *code-abbrev*]: *finite'* = *finite*

lemma *finite'-code* [*code*]:

finite' (*set xs*) ⟷ *True*
finite' (*List.coset xs*) ⟷ *of-phantom* (*finite-UNIV* :: '*a finite-UNIV*)
⟨*proof*⟩

end

context fixes *xs* :: '*a* :: *card-UNIV list*
begin

qualified definition *card'* :: '*a set* ⇒ *nat*
where [*simp*, *code del*, *code-abbrev*]: *card'* = *card*

lemma *card'-code* [*code*]:
card' (*set xs*) = *length* (*remdups xs*)
card' (*List.coset xs*) = *of-phantom* (*card-UNIV* :: '*a card-UNIV*) − *length* (*remdups xs*)
⟨*proof*⟩ **definition** *subset'* :: '*a set* ⇒ '*a set* ⇒ *bool*
where [*simp*, *code del*, *code-abbrev*]: *subset'* = *op* ⊆

lemma *subset'-code* [*code*]:
subset' *A* (*List.coset ys*) ⟷ (∀ *y* ∈ *set ys*. *y* ∉ *A*)
subset' (*set ys*) *B* ⟷ (∀ *y* ∈ *set ys*. *y* ∈ *B*)
subset' (*List.coset xs*) (*set ys*) ⟷ (*let n* = *CARD*('*a*) *in n* > *0* ∧ *card*(*set* (*xs* @ *ys*)) = *n*)
⟨*proof*⟩ **definition** *eq-set* :: '*a set* ⇒ '*a set* ⇒ *bool*
where [*simp*, *code del*, *code-abbrev*]: *eq-set* = *op* =

lemma *eq-set-code* [*code*]:
fixes *ys*
defines *rhs* ≡
let n = *CARD*('*a*)
in if n = *0 then False else*
let xs' = *remdups xs*; *ys'* = *remdups ys*
in length xs' + *length ys'* = *n* ∧ (∀ *x* ∈ *set xs'*. *x* ∉ *set ys'*) ∧ (∀ *y* ∈ *set ys'*. *y* ∉ *set xs'*)
shows *eq-set* (*List.coset xs*) (*set ys*) ⟷ *rhs*
and *eq-set* (*set ys*) (*List.coset xs*) ⟷ *rhs*
and *eq-set* (*set xs*) (*set ys*) ⟷ (∀ *x* ∈ *set xs*. *x* ∈ *set ys*) ∧ (∀ *y* ∈ *set ys*. *y* ∈ *set xs*)
and *eq-set* (*List.coset xs*) (*List.coset ys*) ⟷ (∀ *x* ∈ *set xs*. *x* ∈ *set ys*) ∧ (∀ *y* ∈ *set ys*. *y* ∈ *set xs*)
⟨*proof*⟩

end

Provide more informative exceptions than Match for non-rewritten cases. If generated code raises one these exceptions, then a code equation calls the mentioned operator for an element type that is not an instance of *card-UNIV* and is therefore not implemented via *card-UNIV-class.card-UNIV*. Con-

strain the element type with sort *card-UNIV* to change this.

lemma *card-coset-error* [*code*]:
card (*List.coset xs*) =
Code.abort (*STR ''card* (*List.coset* -) *requires type class instance card-UNIV''*)
(λ-. *card* (*List.coset xs*))
⟨*proof*⟩

lemma *coset-subseteq-set-code* [*code*]:
List.coset xs ⊆ *set ys* ⟷
(*if xs* = [] ∧ *ys* = [] *then False*
else Code.abort
(*STR ''subset-eq* (*List.coset* -) (*List.set* -) *requires type class instance card-UNIV''*)
(λ-. *List.coset xs* ⊆ *set ys*))
⟨*proof*⟩

notepad begin — test code setup
⟨*proof*⟩
end

end

64 Numeral Syntax for Types

theory *Numeral-Type*
imports *Cardinality*
begin

64.1 Numeral Types

typedef *num0* = *UNIV* :: *nat set* ⟨*proof*⟩
typedef *num1* = *UNIV* :: *unit set* ⟨*proof*⟩

typedef *'a bit0* = {*0* ..< *2* ∗ *int CARD*('*a*::*finite*)}
⟨*proof*⟩

typedef *'a bit1* = {*0* ..< *1* + *2* ∗ *int CARD*('*a*::*finite*)}
⟨*proof*⟩

lemma *card-num0* [*simp*]: *CARD* (*num0*) = *0*
⟨*proof*⟩

lemma *infinite-num0*: ¬ *finite* (*UNIV* :: *num0 set*)
⟨*proof*⟩

lemma *card-num1* [*simp*]: *CARD*(*num1*) = *1*
⟨*proof*⟩

lemma *card-bit0* [*simp*]: *CARD*('*a bit0*) = *2* ∗ *CARD*('*a*::*finite*)
⟨*proof*⟩

lemma *card-bit1* [*simp*]: $CARD('a\ bit1) = Suc\ (2 * CARD('a::finite))$
⟨*proof*⟩

instance *num1* :: *finite*
⟨*proof*⟩

instance *bit0* :: (*finite*) *card2*
⟨*proof*⟩

instance *bit1* :: (*finite*) *card2*
⟨*proof*⟩

64.2 Locales for for modular arithmetic subtypes

locale *mod-type* =
fixes n :: *int*
and *Rep* :: $'a::\{zero,one,plus,times,uminus,minus\} \Rightarrow int$
and *Abs* :: $int \Rightarrow 'a::\{zero,one,plus,times,uminus,minus\}$
assumes *type*: *type-definition Rep Abs* $\{0..<n\}$
and *size1*: $1 < n$
and *zero-def*: $0 = Abs\ 0$
and *one-def*: $1 = Abs\ 1$
and *add-def*: $x + y = Abs\ ((Rep\ x + Rep\ y)\ mod\ n)$
and *mult-def*: $x * y = Abs\ ((Rep\ x * Rep\ y)\ mod\ n)$
and *diff-def*: $x - y = Abs\ ((Rep\ x - Rep\ y)\ mod\ n)$
and *minus-def*: $- x = Abs\ ((- Rep\ x)\ mod\ n)$
begin

lemma *size0*: $0 < n$
⟨*proof*⟩

lemmas *definitions* =
zero-def one-def add-def mult-def minus-def diff-def

lemma *Rep-less-n*: $Rep\ x < n$
⟨*proof*⟩

lemma *Rep-le-n*: $Rep\ x \leq n$
⟨*proof*⟩

lemma *Rep-inject-sym*: $x = y \longleftrightarrow Rep\ x = Rep\ y$
⟨*proof*⟩

lemma *Rep-inverse*: $Abs\ (Rep\ x) = x$
⟨*proof*⟩

lemma *Abs-inverse*: $m \in \{0..<n\} \Longrightarrow Rep\ (Abs\ m) = m$
⟨*proof*⟩

lemma *Rep-Abs-mod*: *Rep* (*Abs* (*m mod n*)) = *m mod n*
⟨*proof*⟩

lemma *Rep-Abs-0*: *Rep* (*Abs 0*) = *0*
⟨*proof*⟩

lemma *Rep-0*: *Rep 0* = *0*
⟨*proof*⟩

lemma *Rep-Abs-1*: *Rep* (*Abs 1*) = *1*
⟨*proof*⟩

lemma *Rep-1*: *Rep 1* = *1*
⟨*proof*⟩

lemma *Rep-mod*: *Rep x mod n* = *Rep x*
⟨*proof*⟩

lemmas *Rep-simps* =
Rep-inject-sym Rep-inverse Rep-Abs-mod Rep-mod Rep-Abs-0 Rep-Abs-1

lemma *comm-ring-1*: *OFCLASS*('*a*, *comm-ring-1-class*)
⟨*proof*⟩

end

locale *mod-ring* = *mod-type n Rep Abs*
for *n* :: *int*
and *Rep* :: '*a*::{*comm-ring-1*} ⇒ *int*
and *Abs* :: *int* ⇒ '*a*::{*comm-ring-1*}
begin

lemma *of-nat-eq*: *of-nat k* = *Abs* (*int k mod n*)
⟨*proof*⟩

lemma *of-int-eq*: *of-int z* = *Abs* (*z mod n*)
⟨*proof*⟩

lemma *Rep-numeral*:
Rep (*numeral w*) = *numeral w mod n*
⟨*proof*⟩

lemma *iszero-numeral*:
iszero (*numeral w*::'*a*) ⟷ *numeral w mod n* = *0*
⟨*proof*⟩

lemma *cases*:
assumes *1*: ⋀*z*. ⟦(*x*::'*a*) = *of-int z*; *0* ≤ *z*; *z* < *n*⟧ ⟹ *P*

shows *P*
⟨*proof*⟩

lemma *induct*:
(⋀*z*. ⟦*0* ≤ *z*; *z* < *n*⟧ ⟹ *P* (*of-int z*)) ⟹ *P* (*x*::*'a*)
⟨*proof*⟩

end

64.3 Ring class instances

Unfortunately *ring-1* instance is not possible for *num1*, since 0 and 1 are not distinct.

instantiation *num1* :: {*comm-ring*,*comm-monoid-mult*,*numeral*}
begin

lemma *num1-eq-iff*: (*x*::*num1*) = (*y*::*num1*) ⟷ *True*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation
bit0 **and** *bit1* :: (*finite*) {*zero*,*one*,*plus*,*times*,*uminus*,*minus*}
begin

definition *Abs-bit0'* :: *int* ⇒ *'a bit0* **where**
Abs-bit0' x = *Abs-bit0* (*x mod int CARD*(*'a bit0*))

definition *Abs-bit1'* :: *int* ⇒ *'a bit1* **where**
Abs-bit1' x = *Abs-bit1* (*x mod int CARD*(*'a bit1*))

definition *0* = *Abs-bit0 0*
definition *1* = *Abs-bit0 1*
definition *x* + *y* = *Abs-bit0'* (*Rep-bit0 x* + *Rep-bit0 y*)
definition *x* ∗ *y* = *Abs-bit0'* (*Rep-bit0 x* ∗ *Rep-bit0 y*)
definition *x* − *y* = *Abs-bit0'* (*Rep-bit0 x* − *Rep-bit0 y*)
definition − *x* = *Abs-bit0'* (− *Rep-bit0 x*)

definition *0* = *Abs-bit1 0*
definition *1* = *Abs-bit1 1*
definition *x* + *y* = *Abs-bit1'* (*Rep-bit1 x* + *Rep-bit1 y*)
definition *x* ∗ *y* = *Abs-bit1'* (*Rep-bit1 x* ∗ *Rep-bit1 y*)
definition *x* − *y* = *Abs-bit1'* (*Rep-bit1 x* − *Rep-bit1 y*)
definition − *x* = *Abs-bit1'* (− *Rep-bit1 x*)

instance ⟨*proof*⟩

end

interpretation *bit0*:
  mod-type int CARD('*a*::*finite bit0*)
    Rep-bit0 :: '*a*::*finite bit0* ⇒ *int*
    Abs-bit0 :: *int* ⇒ '*a*::*finite bit0*
⟨*proof*⟩

interpretation *bit1*:
  mod-type int CARD('*a*::*finite bit1*)
    Rep-bit1 :: '*a*::*finite bit1* ⇒ *int*
    Abs-bit1 :: *int* ⇒ '*a*::*finite bit1*
⟨*proof*⟩

instance *bit0* :: (*finite*) *comm-ring-1*
  ⟨*proof*⟩

instance *bit1* :: (*finite*) *comm-ring-1*
  ⟨*proof*⟩

interpretation *bit0*:
  mod-ring int CARD('*a*::*finite bit0*)
    Rep-bit0 :: '*a*::*finite bit0* ⇒ *int*
    Abs-bit0 :: *int* ⇒ '*a*::*finite bit0*
  ⟨*proof*⟩

interpretation *bit1*:
  mod-ring int CARD('*a*::*finite bit1*)
    Rep-bit1 :: '*a*::*finite bit1* ⇒ *int*
    Abs-bit1 :: *int* ⇒ '*a*::*finite bit1*
  ⟨*proof*⟩

Set up cases, induction, and arithmetic

lemmas *bit0-cases* [*case-names of-int*, *cases type*: *bit0*] = *bit0.cases*
lemmas *bit1-cases* [*case-names of-int*, *cases type*: *bit1*] = *bit1.cases*

lemmas *bit0-induct* [*case-names of-int*, *induct type*: *bit0*] = *bit0.induct*
lemmas *bit1-induct* [*case-names of-int*, *induct type*: *bit1*] = *bit1.induct*

lemmas *bit0-iszero-numeral* [*simp*] = *bit0.iszero-numeral*
lemmas *bit1-iszero-numeral* [*simp*] = *bit1.iszero-numeral*

lemmas [*simp*] = *eq-numeral-iff-iszero* [**where** '*a*='*a bit0*] **for** *dummy* :: '*a*::*finite*
lemmas [*simp*] = *eq-numeral-iff-iszero* [**where** '*a*='*a bit1*] **for** *dummy* :: '*a*::*finite*

64.4 Order instances

instantiation *bit0* **and** *bit1* :: (*finite*) *linorder* **begin**
definition *a* < *b* ⟷ *Rep-bit0 a* < *Rep-bit0 b*

definition $a \leq b \longleftrightarrow$ *Rep-bit0 a* $\leq$ *Rep-bit0 b*
definition $a < b \longleftrightarrow$ *Rep-bit1 a* $<$ *Rep-bit1 b*
definition $a \leq b \longleftrightarrow$ *Rep-bit1 a* $\leq$ *Rep-bit1 b*

instance
⟨*proof*⟩
end

lemma (**in** *preorder*) *tranclp-less*: *op* $<^{++}$ = *op* $<$
⟨*proof*⟩

instance *bit0* **and** *bit1* :: (*finite*) *wellorder*
⟨*proof*⟩

64.5 Code setup and type classes for code generation

Code setup for *num0* and *num1*

definition *Num0* :: *num0* **where** *Num0* = *Abs-num0 0*
code-datatype *Num0*

instantiation *num0* :: *equal* **begin**
definition *equal-num0* :: *num0* ⇒ *num0* ⇒ *bool*
where *equal-num0* = *op* =
instance ⟨*proof*⟩
end

lemma *equal-num0-code* [*code*]:
equal-class.equal Num0 Num0 = *True*
⟨*proof*⟩

code-datatype *1* :: *num1*

instantiation *num1* :: *equal* **begin**
definition *equal-num1* :: *num1* ⇒ *num1* ⇒ *bool*
where *equal-num1* = *op* =
instance ⟨*proof*⟩
end

lemma *equal-num1-code* [*code*]:
equal-class.equal (*1* :: *num1*) *1* = *True*
⟨*proof*⟩

instantiation *num1* :: *enum* **begin**
definition *enum-class.enum* = [*1* :: *num1*]
definition *enum-class.enum-all P* = *P* (*1* :: *num1*)
definition *enum-class.enum-ex P* = *P* (*1* :: *num1*)
instance
⟨*proof*⟩
end

instantiation *num0* **and** *num1* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*num0*) *False*
definition *card-UNIV* = *Phantom*(*num0*) *0*
definition *finite-UNIV* = *Phantom*(*num1*) *True*
definition *card-UNIV* = *Phantom*(*num1*) *1*
instance
⟨*proof*⟩
end

Code setup for *'a bit0* and *'a bit1*

declare
bit0.Rep-inverse[*code abstype*]
bit0.Rep-0[*code abstract*]
bit0.Rep-1[*code abstract*]

lemma *Abs-bit0'-code* [*code abstract*]:
Rep-bit0 (*Abs-bit0'* *x* :: *'a* :: *finite bit0*) = *x mod int* (*CARD*(*'a bit0*))
⟨*proof*⟩

lemma *inj-on-Abs-bit0*:
inj-on (*Abs-bit0* :: *int* ⇒ *'a bit0*) {*0..<2* ∗ *int CARD*(*'a* :: *finite*)}
⟨*proof*⟩

declare
bit1.Rep-inverse[*code abstype*]
bit1.Rep-0[*code abstract*]
bit1.Rep-1[*code abstract*]

lemma *Abs-bit1'-code* [*code abstract*]:
Rep-bit1 (*Abs-bit1'* *x* :: *'a* :: *finite bit1*) = *x mod int* (*CARD*(*'a bit1*))
⟨*proof*⟩

lemma *inj-on-Abs-bit1*:
inj-on (*Abs-bit1* :: *int* ⇒ *'a bit1*) {*0..<1* + *2* ∗ *int CARD*(*'a* :: *finite*)}
⟨*proof*⟩

instantiation *bit0* **and** *bit1* :: (*finite*) *equal* **begin**

definition *equal-class.equal* *x* *y* ⟷ *Rep-bit0* *x* = *Rep-bit0* *y*
definition *equal-class.equal* *x* *y* ⟷ *Rep-bit1* *x* = *Rep-bit1* *y*

instance
⟨*proof*⟩

end

instantiation *bit0* :: (*finite*) *enum* **begin**
definition (*enum-class.enum* :: *'a bit0 list*) = *map* (*Abs-bit0'* ∘ *int*) (*upt 0* (*CARD*(*'a*

bit0)))
definition *enum-class.enum-all P* = (∀ *b* :: *'a bit0* ∈ *set enum-class.enum*. *P b*)
definition *enum-class.enum-ex P* = (∃ *b* :: *'a bit0* ∈ *set enum-class.enum*. *P b*)

instance
⟨*proof*⟩

end

instantiation *bit1* :: (*finite*) *enum* **begin**
definition (*enum-class.enum* :: *'a bit1 list*) = *map* (*Abs-bit1'* ∘ *int*) (*upt 0* (*CARD*('*a bit1*)))
definition *enum-class.enum-all P* = (∀ *b* :: *'a bit1* ∈ *set enum-class.enum*. *P b*)
definition *enum-class.enum-ex P* = (∃ *b* :: *'a bit1* ∈ *set enum-class.enum*. *P b*)

instance
⟨*proof*⟩

end

instantiation *bit0* **and** *bit1* :: (*finite*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a bit0*) *True*
definition *finite-UNIV* = *Phantom*('*a bit1*) *True*
instance ⟨*proof*⟩
end

instantiation *bit0* **and** *bit1* :: ({*finite*,*card-UNIV*}) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a bit0*) (*2* ∗ *of-phantom* (*card-UNIV* :: *'a card-UNIV*))
definition *card-UNIV* = *Phantom*('*a bit1*) (*1* + *2* ∗ *of-phantom* (*card-UNIV* :: *'a card-UNIV*))
instance ⟨*proof*⟩
end

64.6 Syntax

syntax
-NumeralType :: *num-token* => *type* (-)
-NumeralType0 :: *type* (*0*)
-NumeralType1 :: *type* (*1*)

translations
(*type*) *1* == (*type*) *num1*
(*type*) *0* == (*type*) *num0*

⟨*ML*⟩

64.7 Examples

lemma *CARD*(*0*) = *0* ⟨*proof*⟩

lemma *CARD(17) = 17* ⟨*proof*⟩
lemma *8 * 11 ^ 3 − 6 = (2::5)* ⟨*proof*⟩

end

65 ω-words

theory *Omega-Words-Fun*

imports *Infinite-Set*
begin

Note: This theory is based on Stefan Merz's work.

Automata recognize languages, which are sets of words. For the theory of ω-automata, we are mostly interested in ω-words, but it is sometimes useful to reason about finite words, too. We are modeling finite words as lists; this lets us benefit from the existing library. Other formalizations could be investigated, such as representing words as functions whose domains are initial intervals of the natural numbers.

65.1 Type declaration and elementary operations

We represent ω-words as functions from the natural numbers to the alphabet type. Other possible formalizations include a coinductive definition or a uniform encoding of finite and infinite words, as studied by Müller et al.

type-synonym
'a word = nat ⇒ 'a

We can prefix a finite word to an ω-word, and a way to obtain an ω-word from a finite, non-empty word is by ω-iteration.

definition
conc :: ['a list, 'a word] ⇒ 'a word (**infixr** ⌢ *65*)
where *w ⌢ x == λn. if n < length w then w!n else x (n − length w)*

definition
iter :: 'a list ⇒ 'a word ((-^ω) [*1000*])
where *iter w == if w = [] then undefined else (λn. w!(n mod (length w)))*

lemma *conc-empty*[*simp*]: *[] ⌢ w = w*
⟨*proof*⟩

lemma *conc-fst*[*simp*]: *n < length w ⟹ (w ⌢ x) n = w!n*
⟨*proof*⟩

lemma *conc-snd*[*simp*]: *¬(n < length w) ⟹ (w ⌢ x) n = x (n − length w)*
⟨*proof*⟩

lemma *iter-nth* [*simp*]: $0 < length\ w \Longrightarrow w^\omega\ n = w!(n\ mod\ (length\ w))$
⟨*proof*⟩

lemma *conc-conc*[*simp*]: $u \frown v \frown w = (u\ @\ v) \frown w$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *range-conc*[*simp*]: $range\ (w_1 \frown w_2) = set\ w_1 \cup range\ w_2$
⟨*proof*⟩

lemma *iter-unroll*: $0 < length\ w \Longrightarrow w^\omega = w \frown w^\omega$
⟨*proof*⟩

65.2 Subsequence, Prefix, and Suffix

definition *suffix* :: [*nat*, *'a word*] ⇒ *'a word*
where $suffix\ k\ x \equiv \lambda n.\ x\ (k+n)$

definition *subsequence* :: *'a word* ⇒ *nat* ⇒ *nat* ⇒ *'a list* (- [- → -] *900*)
where $subsequence\ w\ i\ j \equiv map\ w\ [i..<j]$

abbreviation *prefix* :: *nat* ⇒ *'a word* ⇒ *'a list*
where $prefix\ n\ w \equiv subsequence\ w\ 0\ n$

lemma *suffix-nth* [*simp*]: $(suffix\ k\ x)\ n = x\ (k+n)$
⟨*proof*⟩

lemma *suffix-0* [*simp*]: $suffix\ 0\ x = x$
⟨*proof*⟩

lemma *suffix-suffix* [*simp*]: $suffix\ m\ (suffix\ k\ x) = suffix\ (k+m)\ x$
⟨*proof*⟩

lemma *subsequence-append*: $prefix\ (i + j)\ w = prefix\ i\ w\ @\ (w\ [i \rightarrow i + j])$
⟨*proof*⟩

lemma *subsequence-drop*[*simp*]: $drop\ i\ (w\ [j \rightarrow k]) = w\ [j + i \rightarrow k]$
⟨*proof*⟩

lemma *subsequence-empty*[*simp*]: $w\ [i \rightarrow j] = [] \longleftrightarrow j \leq i$
⟨*proof*⟩

lemma *subsequence-length*[*simp*]: $length\ (subsequence\ w\ i\ j) = j - i$
⟨*proof*⟩

lemma *subsequence-nth*[*simp*]: $k < j - i \Longrightarrow (w\ [i \rightarrow j])\ !\ k = w\ (i + k)$
⟨*proof*⟩

lemma *subseq-to-zero*[*simp*]: $w[i \rightarrow 0] = []$

⟨*proof*⟩

lemma *subseq-to-smaller*[*simp*]: $i \geq j \Longrightarrow w[i \rightarrow j] = []$
⟨*proof*⟩

lemma *subseq-to-Suc*[*simp*]: $i \leq j \Longrightarrow w\ [i \rightarrow Suc\ j] = w\ [\ i \rightarrow j\ ] \ @\ [w\ j]$
⟨*proof*⟩

lemma *subsequence-singleton*[*simp*]: $w\ [i \rightarrow Suc\ i] = [w\ i]$
⟨*proof*⟩

lemma *subsequence-prefix-suffix*: $prefix\ (j - i)\ (suffix\ i\ w) = w\ [i \rightarrow j]$
⟨*proof*⟩

lemma *prefix-suffix*: $x = prefix\ n\ x \frown (suffix\ n\ x)$
⟨*proof*⟩

declare *prefix-suffix*[*symmetric*, *simp*]

lemma *word-split*: **obtains** $v_1\ v_2$ **where** $v = v_1 \frown v_2\ length\ v_1 = k$
⟨*proof*⟩

lemma *set-subsequence*[*simp*]: $set\ (w[i \rightarrow j]) = w`\{i..<j\}$
⟨*proof*⟩

lemma *subsequence-take*[*simp*]: $take\ i\ (w\ [j \rightarrow k]) = w\ [j \rightarrow min\ (j + i)\ k]$
⟨*proof*⟩

lemma *subsequence-shift*[*simp*]: $(suffix\ i\ w)\ [j \rightarrow k] = w\ [i + j \rightarrow i + k]$
⟨*proof*⟩

lemma *suffix-subseq-join*[*simp*]: $i \leq j \Longrightarrow v\ [i \rightarrow j] \frown suffix\ j\ v = suffix\ i\ v$
⟨*proof*⟩

lemma *prefix-conc-fst*[*simp*]:
assumes $j \leq length\ w$
shows $prefix\ j\ (w \frown w') = take\ j\ w$
⟨*proof*⟩

lemma *prefix-conc-snd*[*simp*]:
assumes $n \geq length\ u$
shows $prefix\ n\ (u \frown v) = u\ @\ prefix\ (n - length\ u)\ v$
⟨*proof*⟩

lemma *prefix-conc-length*[*simp*]: $prefix\ (length\ w)\ (w \frown w') = w$
⟨*proof*⟩

lemma *suffix-conc-fst*[*simp*]:
assumes $n \leq$ *length* u
shows *suffix* n $(u \frown v) =$ *drop* n $u \frown v$
⟨*proof*⟩

lemma *suffix-conc-snd*[*simp*]:
assumes $n \geq$ *length* u
shows *suffix* n $(u \frown v) =$ *suffix* $(n -$ *length* $u)$ v
⟨*proof*⟩

lemma *suffix-conc-length*[*simp*]: *suffix* (*length* w) $(w \frown w') = w'$
⟨*proof*⟩

lemma *concat-eq*[*iff*]:
assumes *length* $v_1 =$ *length* v_2
shows $v_1 \frown u_1 = v_2 \frown u_2 \longleftrightarrow v_1 = v_2 \wedge u_1 = u_2$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *same-concat-eq*[*iff*]: $u \frown v = u \frown w \longleftrightarrow v = w$
⟨*proof*⟩

lemma *comp-concat*[*simp*]: $f \circ u \frown v =$ *map* f $u \frown (f \circ v)$
⟨*proof*⟩

65.3 Prepending

primrec *build* :: $'a \Rightarrow 'a$ *word* $\Rightarrow 'a$ *word* (**infixr** *##* 65)
where $(a$ *##* $w)$ *0* $= a \mid (a$ *##* $w)$ (*Suc* $i) = w\ i$

lemma *build-eq*[*iff*]: a_1 *##* $w_1 = a_2$ *##* $w_2 \longleftrightarrow a_1 = a_2 \wedge w_1 = w_2$
⟨*proof*⟩

lemma *build-cons*[*simp*]: $(a$ *#* $u) \frown v = a$ *##* $u \frown v$
⟨*proof*⟩

lemma *build-append*[*simp*]: $(w$ *@* a *#* $u) \frown v = w \frown a$ *##* $u \frown v$
⟨*proof*⟩

lemma *build-first*[*simp*]: w *0* *##* *suffix* (*Suc 0*) $w = w$
⟨*proof*⟩

lemma *build-split*[*intro*]: $w = w$ *0* *##* *suffix 1* w
⟨*proof*⟩

lemma *build-range*[*simp*]: *range* $(a$ *##* $w) =$ *insert* a (*range* w)
⟨*proof*⟩

lemma *suffix-singleton-suffix*[*simp*]: *w i ## suffix* (*Suc i*) *w* = *suffix i w*
⟨*proof*⟩

Find the first occurrence of a letter from a given set

lemma *word-first-split-set*:
assumes *A* ∩ *range w* ≠ {}
obtains *u a v* **where** *w* = *u* ⌢ [*a*] ⌢ *v A* ∩ *set u* = {} *a* ∈ *A*
⟨*proof*⟩

65.4 The limit set of an ω-word

The limit set (also called infinity set) of an ω-word is the set of letters that appear infinitely often in the word. This set plays an important role in defining acceptance conditions of ω-automata.

definition *limit* :: *'a word* ⇒ *'a set*
where *limit x* ≡ {*a* . ∃ ∞*n* . *x n* = *a*}

lemma *limit-iff-frequent*: *a* ∈ *limit x* ⟷ (∃ ∞*n* . *x n* = *a*)
⟨*proof*⟩

The following is a different way to define the limit, using the reverse image, making the laws about reverse image applicable to the limit set. (Might want to change the definition above?)

lemma *limit-vimage*: (*a* ∈ *limit x*) = *infinite* (*x* −‘ {*a*})
⟨*proof*⟩

lemma *two-in-limit-iff*:
({*a*, *b*} ⊆ *limit x*) =
((∃ *n*. *x n* =*a*) ∧ (∀ *n*. *x n* = *a* ⟶ (∃ *m*>*n*. *x m* = *b*)) ∧ (∀ *m*. *x m* = *b* ⟶ (∃ *n*>*m*. *x n* = *a*)))
(**is** *?lhs* = (*?r1* ∧ *?r2* ∧ *?r3*))
⟨*proof*⟩

For ω-words over a finite alphabet, the limit set is non-empty. Moreover, from some position onward, any such word contains only letters from its limit set.

lemma *limit-nonempty*:
assumes *fin*: *finite* (*range x*)
shows ∃ *a*. *a* ∈ *limit x*
⟨*proof*⟩

lemmas *limit-nonemptyE* = *limit-nonempty*[*THEN exE*]

lemma *limit-inter-INF*:
assumes *hyp*: *limit w* ∩ *S* ≠ {}
shows ∃ ∞ *n*. *w n* ∈ *S*
⟨*proof*⟩

The reverse implication is true only if *S* is finite.

lemma *INF-limit-inter*:
assumes *hyp*: $\exists_\infty\ n.\ w\ n \in S$
and *fin*: *finite* $(S \cap range\ w)$
shows $\exists\, a.\ a \in limit\ w \cap S$
⟨*proof*⟩

lemma *fin-ex-inf-eq-limit*: *finite* $A \Longrightarrow (\exists_\infty i.\ w\ i \in A) \longleftrightarrow limit\ w \cap A \neq \{\}$
⟨*proof*⟩

lemma *limit-in-range-suffix*: $limit\ x \subseteq range\ (suffix\ k\ x)$
⟨*proof*⟩

lemma *limit-in-range*: $limit\ r \subseteq range\ r$
⟨*proof*⟩

lemmas *limit-in-range-suffixD* = *limit-in-range-suffix*[*THEN subsetD*]

lemma *limit-subset*: $limit\ f \subseteq f\ `\ \{n..\}$
⟨*proof*⟩

theorem *limit-is-suffix*:
assumes *fin*: *finite* $(range\ x)$
shows $\exists\, k.\ limit\ x = range\ (suffix\ k\ x)$
⟨*proof*⟩

lemmas *limit-is-suffixE* = *limit-is-suffix*[*THEN exE*]

The limit set enjoys some simple algebraic laws with respect to concatenation, suffixes, iteration, and renaming.

theorem *limit-conc* [*simp*]: $limit\ (w \frown x) = limit\ x$
⟨*proof*⟩

theorem *limit-suffix* [*simp*]: $limit\ (suffix\ n\ x) = limit\ x$
⟨*proof*⟩

theorem *limit-iter* [*simp*]:
assumes *nempty*: $0 < length\ w$
shows $limit\ w^\omega = set\ w$
⟨*proof*⟩

lemma *limit-o* [*simp*]:
assumes *a*: $a \in limit\ w$
shows $f\ a \in limit\ (f \circ w)$
⟨*proof*⟩

The converse relation is not true in general: $f(a)$ can be in the limit of $f \circ w$ even though a is not in the limit of w. However, *limit* commutes with renaming if the function is injective. More generally, if $f(a)$ is the image of only finitely many elements, some of these must be in the limit of w.

lemma *limit-o-inv*:
  assumes *fin*: *finite* $(f -` \{x\})$
    and x: $x \in$ *limit* $(f \circ w)$
  shows $\exists\, a \in (f -` \{x\}).\ a \in$ *limit* w
⟨*proof*⟩

theorem *limit-inj* [*simp*]:
  assumes *inj*: *inj* f
  shows *limit* $(f \circ w) = f$ ` (*limit* w)
⟨*proof*⟩

lemma *limit-inter-empty*:
  assumes *fin*: *finite* (*range* w)
  assumes *hyp*: *limit* $w \cap S = \{\}$
  shows $\forall_\infty n.\ w\ n \notin S$
⟨*proof*⟩

If the limit is the suffix of the sequence's range, we may increase the suffix index arbitrarily

lemma *limit-range-suffix-incr*:
  assumes *limit* $r =$ *range* (*suffix* i r)
  assumes $j \geq i$
  shows *limit* $r =$ *range* (*suffix* j r)
    (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

For two finite sequences, we can find a common suffix index such that the limits can be represented as these suffixes' ranges.

lemma *common-range-limit*:
  assumes *finite* (*range* x)
    and *finite* (*range* y)
  obtains i **where** *limit* $x =$ *range* (*suffix* i x)
    and *limit* $y =$ *range* (*suffix* i y)
⟨*proof*⟩

65.5 Index sequences and piecewise definitions

A word can be defined piecewise: given a sequence of words $w_0, w_1, \ldots$ and a strictly increasing sequence of integers $i_0, i_1, \ldots$ where $i_0 = 0$, a single word is obtained by concatenating subwords of the w_n as given by the integers: the resulting word is

$$(w_0)_{i_0} \ldots (w_0)_{i_1-1} (w_1)_{i_1} \ldots (w_1)_{i_2-1} \ldots$$

We prepare the field by proving some trivial facts about such sequences of indexes.

definition *idx-sequence* :: *nat word* ⇒ *bool*
  where *idx-sequence idx* ≡ (*idx 0* = *0*) ∧ (∀ *n*. *idx n* < *idx* (*Suc n*))

lemma *idx-sequence-less*:
assumes *iseq*: *idx-sequence idx*
shows *idx n* < *idx* (*Suc*(*n*+*k*))
⟨*proof*⟩

lemma *idx-sequence-inj*:
assumes *iseq*: *idx-sequence idx*
and *eq*: *idx m* = *idx n*
shows *m* = *n*
⟨*proof*⟩

lemma *idx-sequence-mono*:
assumes *iseq*: *idx-sequence idx*
and *m*: *m* ≤ *n*
shows *idx m* ≤ *idx n*
⟨*proof*⟩

Given an index sequence, every natural number is contained in the interval defined by two adjacent indexes, and in fact this interval is determined uniquely.

lemma *idx-sequence-idx*:
assumes *idx-sequence idx*
shows *idx k* ∈ {*idx k* ..< *idx* (*Suc k*)}
⟨*proof*⟩

lemma *idx-sequence-interval*:
assumes *iseq*: *idx-sequence idx*
shows ∃ *k*. *n* ∈ {*idx k* ..< *idx* (*Suc k*) }
(**is** *?P n* **is** ∃ *k*. *?in n k*)
⟨*proof*⟩

lemma *idx-sequence-interval-unique*:
assumes *iseq*: *idx-sequence idx*
and *k*: *n* ∈ {*idx k* ..< *idx* (*Suc k*)}
and *m*: *n* ∈ {*idx m* ..< *idx* (*Suc m*)}
shows *k* = *m*
⟨*proof*⟩

lemma *idx-sequence-unique-interval*:
assumes *iseq*: *idx-sequence idx*
shows ∃! *k*. *n* ∈ {*idx k* ..< *idx* (*Suc k*) }
⟨*proof*⟩

Now we can define the piecewise construction of a word using an index sequence.

definition *merge* :: *'a word word* ⇒ *nat word* ⇒ *'a word*
where *merge ws idx* ≡ λ*n*. *let i* = *THE i*. *n* ∈ {*idx i* ..< *idx* (*Suc i*) } *in ws i n*

lemma *merge*:
assumes *idx*: *idx-sequence idx*
and *n*: $n \in \{idx\ i\ ..<\ idx\ (Suc\ i)\}$
shows *merge ws idx n* = *ws i n*
⟨*proof*⟩

lemma *merge0*:
assumes *idx*: *idx-sequence idx*
shows *merge ws idx 0* = *ws 0 0*
⟨*proof*⟩

lemma *merge-Suc*:
assumes *idx*: *idx-sequence idx*
and *n*: $n \in \{idx\ i\ ..<\ idx\ (Suc\ i)\}$
shows *merge ws idx* (*Suc n*) = (*if Suc n* = *idx* (*Suc i*) *then ws* (*Suc i*) *else ws i*) (*Suc n*)
⟨*proof*⟩

end

66 Combinator syntax for generic, open state monads (single-threaded monads)

theory *Open-State-Syntax*
imports *Main*
begin

66.1 Motivation

The logic HOL has no notion of constructor classes, so it is not possible to model monads the Haskell way in full genericity in Isabelle/HOL.

However, this theory provides substantial support for a very common class of monads: *state monads* (or *single-threaded monads*, since a state is transformed single-threadedly).

To enter from the Haskell world, http://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm makes a good motivating start. Here we just sketch briefly how those monads enter the game of Isabelle/HOL.

66.2 State transformations and combinators

We classify functions operating on states into two categories:

transformations with type signature $\sigma \Rightarrow \sigma'$, transforming a state.

"yielding" transformations with type signature $\sigma \Rightarrow \alpha \times \sigma'$, "yielding" a side result while transforming a state.

queries with type signature $\sigma \Rightarrow \alpha$, computing a result dependent on a state.

By convention we write σ for types representing states and α, β, γ, ... for types representing side results. Type changes due to transformations are not excluded in our scenario.

We aim to assert that values of any state type σ are used in a single-threaded way: after application of a transformation on a value of type σ, the former value should not be used again. To achieve this, we use a set of monad combinators:

notation *fcomp* (**infixl** $\circ>$ *60*)
notation *scomp* (**infixl** $\circ\!\!\rightarrow$ *60*)

Given two transformations f and g, they may be directly composed using the *op* $\circ>$ combinator, forming a forward composition: $(f \circ> g)\ s = f\ (g\ s)$.

After any yielding transformation, we bind the side result immediately using a lambda abstraction. This is the purpose of the *op* $\circ\!\!\rightarrow$ combinator: $(f \circ\!\!\rightarrow (\lambda x.\ g))\ s = (let\ (x,\ s') = f\ s\ in\ g\ s')$.

For queries, the existing *Let* is appropriate.

Naturally, a computation may yield a side result by pairing it to the state from the left; we introduce the suggestive abbreviation *return* for this purpose.

The most crucial distinction to Haskell is that we do not need to introduce distinguished type constructors for different kinds of state. This has two consequences:

- The monad model does not state anything about the kind of state; the model for the state is completely orthogonal and may be specified completely independently.
- There is no distinguished type constructor encapsulating away the state transformation, i.e. transformations may be applied directly without using any lifting or providing and dropping units ("open monad").
- The type of states may change due to a transformation.

66.3 Monad laws

The common monadic laws hold and may also be used as normalization rules for monadic expressions:

lemmas *monad-simp = Pair-scomp scomp-Pair id-fcomp fcomp-id scomp-scomp scomp-fcomp fcomp-scomp fcomp-assoc*

Evaluation of monadic expressions by force:

lemmas *monad-collapse = monad-simp fcomp-apply scomp-apply split-beta*

66.4 Do-syntax

nonterminal *sdo-binds* **and** *sdo-bind*

syntax
-sdo-block :: *sdo-binds* ⇒ *'a* (*exec* {//(*2* -)//} [*12*] *62*)
-sdo-bind :: [*pttrn*, *'a*] ⇒ *sdo-bind* ((- ←/ -) *13*)
-sdo-let :: [*pttrn*, *'a*] ⇒ *sdo-bind* ((*2let* - =/ -) [*1000*, *13*] *13*)
-sdo-then :: *'a* ⇒ *sdo-bind* (- [*14*] *13*)
-sdo-final :: *'a* ⇒ *sdo-binds* (-)
-sdo-cons :: [*sdo-bind*, *sdo-binds*] ⇒ *sdo-binds* (-;//- [*13*, *12*] *12*)

syntax (*ASCII*)
-sdo-bind :: [*pttrn*, *'a*] ⇒ *sdo-bind* ((- <−/ -) *13*)

translations
-sdo-block (*-sdo-cons* (*-sdo-bind p t*) (*-sdo-final e*))
⇌ *CONST scomp t* (*λp. e*)
-sdo-block (*-sdo-cons* (*-sdo-then t*) (*-sdo-final e*))
=> *CONST fcomp t e*
-sdo-final (*-sdo-block* (*-sdo-cons* (*-sdo-then t*) (*-sdo-final e*)))
<= *-sdo-final* (*CONST fcomp t e*)
-sdo-block (*-sdo-cons* (*-sdo-then t*) *e*)
<= *CONST fcomp t* (*-sdo-block e*)
-sdo-block (*-sdo-cons* (*-sdo-let p t*) *bs*)
⇌ *let p = t in -sdo-block bs*
-sdo-block (*-sdo-cons b* (*-sdo-cons c cs*))
⇌ *-sdo-block* (*-sdo-cons b* (*-sdo-final* (*-sdo-block* (*-sdo-cons c cs*))))
-sdo-cons (*-sdo-let p t*) (*-sdo-final s*)
⇌ *-sdo-final* (*let p = t in s*)
-sdo-block (*-sdo-final e*) => *e*

For an example, see `~~/src/HOL/Proofs/Extraction/Higman_Extraction.thy`.

end

67 Canonical order on option type

theory *Option-ord*
imports *HOL.Option Main*
begin

notation
bot (⊥) **and**
top (⊤) **and**
inf (**infixl** ⊓ *70*) **and**
sup (**infixl** ⊔ *65*) **and**
Inf (⨅- [*900*] *900*) **and**
Sup (⨆- [*900*] *900*)

syntax
-INF1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨅ -./ -) [*0*, *10*] *10*)
-INF :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨅ -∈-./ -) [*0*, *0*, *10*] *10*)
-SUP1 :: *pttrns* ⇒ *'b* ⇒ *'b* ((*3*⨆ -./ -) [*0*, *10*] *10*)
-SUP :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨆ -∈-./ -) [*0*, *0*, *10*] *10*)

instantiation *option* :: (*preorder*) *preorder*
begin

definition *less-eq-option* **where**
x ≤ *y* ⟷ (*case x of None* ⇒ *True* | *Some x* ⇒ (*case y of None* ⇒ *False* | *Some y* ⇒ *x* ≤ *y*))

definition *less-option* **where**
x < *y* ⟷ (*case y of None* ⇒ *False* | *Some y* ⇒ (*case x of None* ⇒ *True* | *Some x* ⇒ *x* < *y*))

lemma *less-eq-option-None* [*simp*]: *None* ≤ *x*
⟨*proof*⟩

lemma *less-eq-option-None-code* [*code*]: *None* ≤ *x* ⟷ *True*
⟨*proof*⟩

lemma *less-eq-option-None-is-None*: *x* ≤ *None* ⟹ *x* = *None*
⟨*proof*⟩

lemma *less-eq-option-Some-None* [*simp*, *code*]: *Some x* ≤ *None* ⟷ *False*
⟨*proof*⟩

lemma *less-eq-option-Some* [*simp*, *code*]: *Some x* ≤ *Some y* ⟷ *x* ≤ *y*
⟨*proof*⟩

lemma *less-option-None* [*simp*, *code*]: *x* < *None* ⟷ *False*
⟨*proof*⟩

lemma *less-option-None-is-Some*: *None* < *x* ⟹ ∃ *z*. *x* = *Some z*
⟨*proof*⟩

lemma *less-option-None-Some* [*simp*]: *None* < *Some x*
⟨*proof*⟩

lemma *less-option-None-Some-code* [*code*]: *None* < *Some x* ⟷ *True*
⟨*proof*⟩

lemma *less-option-Some* [*simp*, *code*]: *Some x* < *Some y* ⟷ *x* < *y*
⟨*proof*⟩

instance

⟨*proof*⟩

end

instance *option* :: (*order*) *order*
⟨*proof*⟩

instance *option* :: (*linorder*) *linorder*
⟨*proof*⟩

instantiation *option* :: (*order*) *order-bot*
begin

definition *bot-option* **where** $\bot$ = *None*

instance
⟨*proof*⟩

end

instantiation *option* :: (*order-top*) *order-top*
begin

definition *top-option* **where** $\top$ = *Some* $\top$

instance
⟨*proof*⟩

end

instance *option* :: (*wellorder*) *wellorder*
⟨*proof*⟩

instantiation *option* :: (*inf*) *inf*
begin

definition *inf-option* **where**
x $\sqcap$ *y* = (*case x of None* ⇒ *None* | *Some x* ⇒ (*case y of None* ⇒ *None* | *Some y* ⇒ *Some* (*x* $\sqcap$ *y*)))

lemma *inf-None-1* [*simp*, *code*]: *None* $\sqcap$ *y* = *None*
⟨*proof*⟩

lemma *inf-None-2* [*simp*, *code*]: *x* $\sqcap$ *None* = *None*
⟨*proof*⟩

lemma *inf-Some* [*simp*, *code*]: *Some x* $\sqcap$ *Some y* = *Some* (*x* $\sqcap$ *y*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

instantiation *option* :: (*sup*) *sup*
begin

definition *sup-option* **where**
x ⊔ *y* = (*case x of None* ⇒ *y* | *Some x'* ⇒ (*case y of None* ⇒ *x* | *Some y* ⇒ *Some* (*x'* ⊔ *y*)))

lemma *sup-None-1* [*simp*, *code*]: *None* ⊔ *y* = *y*
⟨*proof*⟩

lemma *sup-None-2* [*simp*, *code*]: *x* ⊔ *None* = *x*
⟨*proof*⟩

lemma *sup-Some* [*simp*, *code*]: *Some x* ⊔ *Some y* = *Some* (*x* ⊔ *y*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *option* :: (*semilattice-inf*) *semilattice-inf*
⟨*proof*⟩

instance *option* :: (*semilattice-sup*) *semilattice-sup*
⟨*proof*⟩

instance *option* :: (*lattice*) *lattice* ⟨*proof*⟩

instance *option* :: (*lattice*) *bounded-lattice-bot* ⟨*proof*⟩

instance *option* :: (*bounded-lattice-top*) *bounded-lattice-top* ⟨*proof*⟩

instance *option* :: (*bounded-lattice-top*) *bounded-lattice* ⟨*proof*⟩

instance *option* :: (*distrib-lattice*) *distrib-lattice*
⟨*proof*⟩

instantiation *option* :: (*complete-lattice*) *complete-lattice*
begin

definition *Inf-option* :: *'a option set* ⇒ *'a option* **where**
⨅*A* = (*if None* ∈ *A then None else Some* (⨅ *Option.these A*))

lemma *None-in-Inf* [*simp*]: *None* ∈ *A* ⟹ ⨅*A* = *None*
⟨*proof*⟩

definition *Sup-option* :: *'a option set* ⇒ *'a option* **where**
  ⨆ *A* = (*if A* = {} ∨ *A* = {*None*} *then None else Some* (⨆ *Option.these A*))

lemma *empty-Sup* [*simp*]: ⨆ {} = *None*
  ⟨*proof*⟩

lemma *singleton-None-Sup* [*simp*]: ⨆ {*None*} = *None*
  ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *Some-Inf*:
  Some (⨅ *A*) = ⨅ (*Some* ‘ *A*)
  ⟨*proof*⟩

lemma *Some-Sup*:
  A ≠ {} ⟹ *Some* (⨆ *A*) = ⨆ (*Some* ‘ *A*)
  ⟨*proof*⟩

lemma *Some-INF*:
  Some (⨅ *x*∈*A*. *f x*) = (⨅ *x*∈*A*. *Some* (*f x*))
  ⟨*proof*⟩

lemma *Some-SUP*:
  A ≠ {} ⟹ *Some* (⨆ *x*∈*A*. *f x*) = (⨆ *x*∈*A*. *Some* (*f x*))
  ⟨*proof*⟩

instance *option* :: (*complete-distrib-lattice*) *complete-distrib-lattice*
⟨*proof*⟩

instance *option* :: (*complete-linorder*) *complete-linorder* ⟨*proof*⟩

no-notation
  bot (⊥) **and**
  top (⊤) **and**
  inf (**infixl** ⊓ *70*) **and**
  sup (**infixl** ⊔ *65*) **and**
  Inf (⨅ - [*900*] *900*) **and**
  Sup (⨆ - [*900*] *900*)

no-syntax
  -INF1 :: *pttrns* ⇒ *'b* ⇒ *'b* (⨅ -./ -) [*0*, *10*] *10*)
  -INF :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (⨅ -∈-./ -) [*0*, *0*, *10*] *10*)
  -SUP1 :: *pttrns* ⇒ *'b* ⇒ *'b* (⨆ -./ -) [*0*, *10*] *10*)

-SUP :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* ((*3*⨆ -∈-./ -) [*0*, *0*, *10*] *10*)

end

68 Futures and parallel lists for code generated towards Isabelle/ML

theory *Parallel*
imports *Main*
begin

68.1 Futures

datatype *'a future* = *fork unit* ⇒ *'a*

primrec *join* :: *'a future* ⇒ *'a* **where**
join (*fork f*) = *f* ()

lemma *future-eqI* [*intro*!]:
assumes *join f* = *join g*
shows *f* = *g*
⟨*proof*⟩

code-printing
type-constructor *future* ⇀ (*Eval*) - *future*
| **constant** *fork* ⇀ (*Eval*) *Future.fork*
| **constant** *join* ⇀ (*Eval*) *Future.join*

code-reserved *Eval Future future*

68.2 Parallel lists

definition *map* :: (*'a* ⇒ *'b*) ⇒ *'a list* ⇒ *'b list* **where**
[*simp*]: *map* = *List.map*

definition *forall* :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool* **where**
forall = *list-all*

lemma *forall-all* [*simp*]:
forall P xs ⟷ (∀ *x*∈*set xs*. *P x*)
⟨*proof*⟩

definition *exists* :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool* **where**
exists = *list-ex*

lemma *exists-ex* [*simp*]:
exists P xs ⟷ (∃ *x*∈*set xs*. *P x*)
⟨*proof*⟩

code-printing
  constant *map* ⇀ (*Eval*) *Par'-List.map*
| **constant** *forall* ⇀ (*Eval*) *Par'-List.forall*
| **constant** *exists* ⇀ (*Eval*) *Par'-List.exists*

code-reserved *Eval Par-List*

hide-const (**open**) *fork join map exists forall*

end

69 Input syntax for pattern aliases (or "as-patterns" in Haskell)

theory *Pattern-Aliases*
imports *Main*
begin

Most functional languages (Haskell, ML, Scala) support aliases in patterns. This allows to refer to a subpattern with a variable name. This theory implements this using a check phase. It works well for function definitions (see usage below). All features are packed into a **bundle**.

The following caveats should be kept in mind:

- The translation expects a term of the form $f\ x\ y = rhs$, where x and y are patterns that may contain aliases. The result of the translation is a nested *Let*-expression on the right hand side. The code generator *does not* print Isabelle pattern aliases to target language pattern aliases.
- The translation does not process nested equalities; only the top-level equality is translated.
- Terms that do not adhere to the above shape may either stay untranslated or produce an error message. The **fun** command will complain if pattern aliases are left untranslated. In particular, there are no checks whether the patterns are wellformed or linear.
- The corresponding uncheck phase attempts to reverse the translation (no guarantee). The additionally introduced variables are bound using a "fake quantifier" that does not appear in the output.
- To obtain reasonable induction principles in function definitions, the bundle also declares a custom congruence rule for *Let* that only affects **fun**. This congruence rule might lead to an explosion in term size (although that is rare)! In some circumstances (using *let* to destructure tuples), the internal construction of functions stumbles over this rule and prints an error. To mitigate this, either

  - activate the bundle locally (**context includes ... begin**) or
  - rewrite the *let*-expression to use *case*: *let* $(a,\ b) = x$ *in* $(b,\ a)$ becomes *case* x *of* $(a,\ b) \Rightarrow (b,\ a)$.

- The bundle also adds the *Let ?s ?f* $\equiv$ *?f ?s* rule to the simpset.

69.1 Definition

consts
as :: $'a \Rightarrow 'a \Rightarrow 'a$
fake-quant :: $('a \Rightarrow prop) \Rightarrow prop$

lemma *let-cong-unfolding*: $M = N \Longrightarrow f\ N = g\ N \Longrightarrow Let\ M\ f = Let\ N\ g$
⟨*proof*⟩

translations $P <= CONST\ fake\text{-}quant\ (\lambda x.\ P)$

⟨*ML*⟩

bundle *pattern-aliases* **begin**

notation *as* (**infixr** =: *1*)

⟨*ML*⟩

declare *let-cong-unfolding* [*fundef-cong*]
declare *Let-def* [*simp*]

end

hide-const *as*
hide-const *fake-quant*

69.2 Usage

context includes *pattern-aliases* **begin**

Not very useful for plain definitions, but works anyway.

private definition *test-1* $x\ (y =: z) = y + z$

lemma *test-1* $x\ y = y + y$
⟨*proof*⟩

Very useful for function definitions.

private fun *test-2* **where**
test-2 $(y\ \#\ (y'\ \#\ ys =: x') =: x) = x\ @\ x'\ @\ x'\ |$
test-2 $- = []$

lemma *test-2* $(y \# y' \# ys) = (y \# y' \# ys) @ (y' \# ys) @ (y' \# ys)$
⟨*proof*⟩

⟨*ML*⟩

end

end

70 Periodic Functions

theory *Periodic-Fun*
imports *Complex-Main*
begin

A locale for periodic functions. The idea is that one proves $f(x + p) = f(x)$ for some period p and gets derived results like $f(x - p) = f(x)$ and $f(x + 2p) = f(x)$ for free.

g and *gm* are "plus/minus k periods" functions. *g1* and *gn1* are "plus/minus one period" functions. This is useful e.g. if the period is one; the lemmas one gets are then $f\ (x + (1::'b)) = f\ x$ instead of $f\ (x + (1::'b) * (1::'b)) = f\ x$ etc.

locale *periodic-fun* =
  fixes $f :: ('a :: \{ring\text{-}1\}) \Rightarrow 'b$ **and** $g\ gm :: 'a \Rightarrow 'a \Rightarrow 'a$ **and** $g1\ gn1 :: 'a \Rightarrow 'a$
  assumes *plus-1*: $f\ (g1\ x) = f\ x$
  assumes *periodic-arg-plus-0*: $g\ x\ 0 = x$
  assumes *periodic-arg-plus-distrib*: $g\ x\ (of\text{-}int\ (m + n)) = g\ (g\ x\ (of\text{-}int\ n))\ (of\text{-}int\ m)$
  assumes *plus-1-eq*: $g\ x\ 1 = g1\ x$ **and** *minus-1-eq*: $g\ x\ (-1) = gn1\ x$
    and *minus-eq*: $g\ x\ (-y) = gm\ x\ y$
begin

lemma *plus-of-nat*: $f\ (g\ x\ (of\text{-}nat\ n)) = f\ x$
  ⟨*proof*⟩

lemma *minus-of-nat*: $f\ (gm\ x\ (of\text{-}nat\ n)) = f\ x$
⟨*proof*⟩

lemma *plus-of-int*: $f\ (g\ x\ (of\text{-}int\ n)) = f\ x$
  ⟨*proof*⟩

lemma *minus-of-int*: $f\ (gm\ x\ (of\text{-}int\ n)) = f\ x$
  ⟨*proof*⟩

lemma *plus-numeral*: $f\ (g\ x\ (numeral\ n)) = f\ x$
  ⟨*proof*⟩

lemma *minus-numeral*: $f\ (gm\ x\ (numeral\ n)) = f\ x$
⟨*proof*⟩

lemma *minus-1*: $f\ (gn1\ x) = f\ x$
⟨*proof*⟩

lemmas *periodic-simps = plus-of-nat minus-of-nat plus-of-int minus-of-int plus-numeral minus-numeral plus-1 minus-1*

end

Specialised case of the *periodic-fun* locale for periods that are not 1. Gives lemmas $f\ (x - period) = f\ x$ etc.

locale *periodic-fun-simple* =
fixes $f :: ('a :: \{ring\text{-}1\}) \Rightarrow 'b$ **and** $period :: 'a$
assumes *plus-period*: $f\ (x + period) = f\ x$
begin
sublocale *periodic-fun f* $\lambda z\ x.\ z + x * period$ $\lambda z\ x.\ z - x * period$ $\lambda z.\ z + period$ $\lambda z.\ z - period$
⟨*proof*⟩
end

Specialised case of the *periodic-fun* locale for period 1. Gives lemmas $f\ (x - (1::'b)) = f\ x$ etc.

locale *periodic-fun-simple'* =
fixes $f :: ('a :: \{ring\text{-}1\}) \Rightarrow 'b$
assumes *plus-period*: $f\ (x + 1) = f\ x$
begin
sublocale *periodic-fun f* $\lambda z\ x.\ z + x$ $\lambda z\ x.\ z - x$ $\lambda z.\ z + 1$ $\lambda z.\ z - 1$
⟨*proof*⟩

lemma *of-nat*: $f\ (of\text{-}nat\ n) = f\ 0$ ⟨*proof*⟩
lemma *uminus-of-nat*: $f\ (-of\text{-}nat\ n) = f\ 0$ ⟨*proof*⟩
lemma *of-int*: $f\ (of\text{-}int\ n) = f\ 0$ ⟨*proof*⟩
lemma *uminus-of-int*: $f\ (-of\text{-}int\ n) = f\ 0$ ⟨*proof*⟩
lemma *of-numeral*: $f\ (numeral\ n) = f\ 0$ ⟨*proof*⟩
lemma *of-neg-numeral*: $f\ (-numeral\ n) = f\ 0$ ⟨*proof*⟩
lemma *of-1*: $f\ 1 = f\ 0$ ⟨*proof*⟩
lemma *of-neg-1*: $f\ (-1) = f\ 0$ ⟨*proof*⟩

lemmas *periodic-simps'* =
of-nat uminus-of-nat of-int uminus-of-int of-numeral of-neg-numeral of-1 of-neg-1

end

lemma *sin-plus-pi*: $sin\ ((z :: 'a :: \{real\text{-}normed\text{-}field, banach\}) + of\text{-}real\ pi) = -sin\ z$
⟨*proof*⟩

lemma *cos-plus-pi*: *cos* ((*z* :: '*a* :: {*real-normed-field*,*banach*}) + *of-real pi*) = − *cos z*
⟨*proof*⟩

interpretation *sin*: *periodic-fun-simple sin* 2 ∗ *of-real pi* :: '*a* :: {*real-normed-field*,*banach*}
⟨*proof*⟩

interpretation *cos*: *periodic-fun-simple cos* 2 ∗ *of-real pi* :: '*a* :: {*real-normed-field*,*banach*}
⟨*proof*⟩

interpretation *tan*: *periodic-fun-simple tan* 2 ∗ *of-real pi* :: '*a* :: {*real-normed-field*,*banach*}
⟨*proof*⟩

interpretation *cot*: *periodic-fun-simple cot* 2 ∗ *of-real pi* :: '*a* :: {*real-normed-field*,*banach*}
⟨*proof*⟩

end

71 Permutations as abstract type

theory *Perm*
imports *Main*
begin

This theory introduces basics about permutations, i.e. almost everywhere fix bijections. But it is by no means complete. Grieviously missing are cycles since these would require more elaboration, e.g. the concept of distinct lists equivalent under rotation, which maybe would also deserve its own theory. But see theory *src/HOL/ex/Perm-Fragments.thy* for fragments on that.

71.1 Abstract type of permutations

typedef '*a perm* = {*f* :: '*a* ⇒ '*a*. *bij f* ∧ *finite* {*a*. *f a* ≠ *a*}}
morphisms *apply Perm*
⟨*proof*⟩

setup-lifting *type-definition-perm*

notation *apply* (**infixl** ⟨$⟩ *999*)
no-notation *apply* (*op* ⟨$⟩)

lemma *bij-apply* [*simp*]:
bij (*apply f*)
⟨*proof*⟩

lemma *perm-eqI*:
assumes ⋀*a*. *f* ⟨$⟩ *a* = *g* ⟨$⟩ *a*

shows $f = g$
⟨*proof*⟩

lemma *perm-eq-iff*:
$f = g \longleftrightarrow (\forall a.\ f\ \langle\$\rangle\ a = g\ \langle\$\rangle\ a)$
⟨*proof*⟩

lemma *apply-inj*:
$f\ \langle\$\rangle\ a = f\ \langle\$\rangle\ b \longleftrightarrow a = b$
⟨*proof*⟩

lift-definition *affected* :: *'a perm* ⇒ *'a set*
is $\lambda f.\ \{a.\ f\ a \neq a\}$ ⟨*proof*⟩

lemma *in-affected*:
$a \in \mathit{affected}\ f \longleftrightarrow f\ \langle\$\rangle\ a \neq a$
⟨*proof*⟩

lemma *finite-affected* [*simp*]:
finite (*affected f*)
⟨*proof*⟩

lemma *apply-affected* [*simp*]:
$f\ \langle\$\rangle\ a \in \mathit{affected}\ f \longleftrightarrow a \in \mathit{affected}\ f$
⟨*proof*⟩

lemma *card-affected-not-one*:
card (*affected f*) ≠ *1*
⟨*proof*⟩

71.2 Identity, composition and inversion

instantiation *Perm.perm* :: (*type*) {*monoid-mult*, *inverse*}
begin

lift-definition *one-perm* :: *'a perm*
is *id*
⟨*proof*⟩

lemma *apply-one* [*simp*]:
apply 1 = *id*
⟨*proof*⟩

lemma *affected-one* [*simp*]:
affected 1 = {}
⟨*proof*⟩

lemma *affected-empty-iff* [*simp*]:
affected f = {} ⟷ *f* = *1*

⟨*proof*⟩

lift-definition *times-perm* :: *'a perm* ⇒ *'a perm* ⇒ *'a perm*
is *comp*
⟨*proof*⟩

lemma *apply-times*:
apply (*f* ∗ *g*) = *apply f* ∘ *apply g*
⟨*proof*⟩

lemma *apply-sequence*:
f ⟨$⟩ (*g* ⟨$⟩ *a*) = *apply* (*f* ∗ *g*) *a*
⟨*proof*⟩

lemma *affected-times* [*simp*]:
affected (*f* ∗ *g*) ⊆ *affected f* ∪ *affected g*
⟨*proof*⟩

lift-definition *inverse-perm* :: *'a perm* ⇒ *'a perm*
is *inv*
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *apply-inverse*:
apply (*inverse f*) = *inv* (*apply f*)
⟨*proof*⟩

lemma *affected-inverse* [*simp*]:
affected (*inverse f*) = *affected f*
⟨*proof*⟩

global-interpretation *perm*: *group times 1*::*'a perm inverse*
⟨*proof*⟩

declare *perm.inverse-distrib-swap* [*simp*]

lemma *perm-mult-commute*:
assumes *affected f* ∩ *affected g* = {}
shows *g* ∗ *f* = *f* ∗ *g*
⟨*proof*⟩

lemma *apply-power*:
apply (*f* ^ *n*) = *apply f* ^^ *n*
⟨*proof*⟩

lemma *perm-power-inverse*:
inverse f ^ n = inverse ((f :: 'a perm) ^ n)
⟨*proof*⟩

71.3 Orbit and order of elements

definition *orbit* :: *'a perm ⇒ 'a ⇒ 'a set*
where
orbit f a = range (λn. (f ^ n) ⟨$⟩ a)

lemma *in-orbitI*:
assumes *(f ^ n) ⟨$⟩ a = b*
shows *b ∈ orbit f a*
⟨*proof*⟩

lemma *apply-power-self-in-orbit* [*simp*]:
(f ^ n) ⟨$⟩ a ∈ orbit f a
⟨*proof*⟩

lemma *in-orbit-self* [*simp*]:
a ∈ orbit f a
⟨*proof*⟩

lemma *apply-self-in-orbit* [*simp*]:
f ⟨$⟩ a ∈ orbit f a
⟨*proof*⟩

lemma *orbit-not-empty* [*simp*]:
orbit f a ≠ {}
⟨*proof*⟩

lemma *not-in-affected-iff-orbit-eq-singleton*:
a ∉ affected f ⟷ orbit f a = {a} (**is** *?P ⟷ ?Q*)
⟨*proof*⟩

definition *order* :: *'a perm ⇒ 'a ⇒ nat*
where
order f = card ∘ orbit f

lemma *orbit-subset-eq-affected*:
assumes *a ∈ affected f*
shows *orbit f a ⊆ affected f*
⟨*proof*⟩

lemma *finite-orbit* [*simp*]:
finite (orbit f a)
⟨*proof*⟩

lemma *orbit-1* [*simp*]:

orbit 1 a = {a}
⟨*proof*⟩

lemma *order-1* [*simp*]:
order 1 a = 1
⟨*proof*⟩

lemma *card-orbit-eq* [*simp*]:
card (orbit f a) = order f a
⟨*proof*⟩

lemma *order-greater-zero* [*simp*]:
order f a > 0
⟨*proof*⟩

lemma *order-eq-one-iff*:
order f a = Suc 0 ⟷ a ∉ affected f (**is** *?P ⟷ ?Q*)
⟨*proof*⟩

lemma *order-greater-eq-two-iff*:
order f a ≥ 2 ⟷ a ∈ affected f
⟨*proof*⟩

lemma *order-less-eq-affected*:
assumes *f ≠ 1*
shows *order f a ≤ card (affected f)*
⟨*proof*⟩

lemma *affected-order-greater-eq-two*:
assumes *a ∈ affected f*
shows *order f a ≥ 2*
⟨*proof*⟩

lemma *order-witness-unfold*:
assumes *n > 0* **and** *(f ^ n) ⟨$⟩ a = a*
shows *order f a = card ((λm. (f ^ m) ⟨$⟩ a) ` {0..<n})*
⟨*proof*⟩

lemma *inj-on-apply-range*:
inj-on (λm. (f ^ m) ⟨$⟩ a) {..<order f a}
⟨*proof*⟩

lemma *orbit-unfold-image*:
orbit f a = (λn. (f ^ n) ⟨$⟩ a) ` {..<order f a} (**is** *- = ?A*)
⟨*proof*⟩

lemma *in-orbitE*:
assumes *b ∈ orbit f a*
obtains *n* **where** *b = (f ^ n) ⟨$⟩ a* **and** *n < order f a*

⟨*proof*⟩

lemma *apply-power-order* [*simp*]:
(*f* ^ *order f a*) ⟨$⟩ *a* = *a*
⟨*proof*⟩

lemma *apply-power-left-mult-order* [*simp*]:
(*f* ^ (*n* ∗ *order f a*)) ⟨$⟩ *a* = *a*
⟨*proof*⟩

lemma *apply-power-right-mult-order* [*simp*]:
(*f* ^ (*order f a* ∗ *n*)) ⟨$⟩ *a* = *a*
⟨*proof*⟩

lemma *apply-power-mod-order-eq* [*simp*]:
(*f* ^ (*n mod order f a*)) ⟨$⟩ *a* = (*f* ^ *n*) ⟨$⟩ *a*
⟨*proof*⟩

lemma *apply-power-eq-iff*:
(*f* ^ *m*) ⟨$⟩ *a* = (*f* ^ *n*) ⟨$⟩ *a* ⟷ *m mod order f a* = *n mod order f a* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *apply-inverse-eq-apply-power-order-minus-one*:
(*inverse f*) ⟨$⟩ *a* = (*f* ^ (*order f a* − *1*)) ⟨$⟩ *a*
⟨*proof*⟩

lemma *apply-inverse-self-in-orbit* [*simp*]:
(*inverse f*) ⟨$⟩ *a* ∈ *orbit f a*
⟨*proof*⟩

lemma *apply-inverse-power-eq*:
(*inverse* (*f* ^ *n*)) ⟨$⟩ *a* = (*f* ^ (*order f a* − *n mod order f a*)) ⟨$⟩ *a*
⟨*proof*⟩

lemma *apply-power-eq-self-iff*:
(*f* ^ *n*) ⟨$⟩ *a* = *a* ⟷ *order f a dvd n*
⟨*proof*⟩

lemma *orbit-equiv*:
assumes *b* ∈ *orbit f a*
shows *orbit f b* = *orbit f a* (**is** *?B* = *?A*)
⟨*proof*⟩

lemma *orbit-apply* [*simp*]:
orbit f (*f* ⟨$⟩ *a*) = *orbit f a*
⟨*proof*⟩

lemma *order-apply* [*simp*]:

order f (f ⟨$⟩ a) = order f a
⟨*proof*⟩

lemma *orbit-apply-inverse* [*simp*]:
orbit f (inverse f ⟨$⟩ a) = orbit f a
⟨*proof*⟩

lemma *order-apply-inverse* [*simp*]:
order f (inverse f ⟨$⟩ a) = order f a
⟨*proof*⟩

lemma *orbit-apply-power* [*simp*]:
orbit f ((f ^ n) ⟨$⟩ a) = orbit f a
⟨*proof*⟩

lemma *order-apply-power* [*simp*]:
order f ((f ^ n) ⟨$⟩ a) = order f a
⟨*proof*⟩

lemma *orbit-inverse* [*simp*]:
orbit (inverse f) = orbit f
⟨*proof*⟩

lemma *order-inverse* [*simp*]:
order (inverse f) = order f
⟨*proof*⟩

lemma *orbit-disjoint*:
assumes *orbit f a ≠ orbit f b*
shows *orbit f a ∩ orbit f b = {}*
⟨*proof*⟩

71.4 Swaps

lift-definition *swap* :: *'a ⇒ 'a ⇒ 'a perm* (⟨-↔-⟩)
is *λa b. Fun.swap a b id*
⟨*proof*⟩

lemma *apply-swap-simp* [*simp*]:
⟨a↔b⟩ ⟨$⟩ a = b
⟨a↔b⟩ ⟨$⟩ b = a
⟨*proof*⟩

lemma *apply-swap-same* [*simp*]:
c ≠ a ⟹ c ≠ b ⟹ ⟨a↔b⟩ ⟨$⟩ c = c
⟨*proof*⟩

lemma *apply-swap-eq-iff* [*simp*]:
⟨a↔b⟩ ⟨$⟩ c = a ⟷ c = b

$\langle a{\leftrightarrow}b\rangle\ \langle\$\rangle\ c = b \longleftrightarrow c = a$
⟨*proof*⟩

lemma *swap-1* [*simp*]:
$\langle a{\leftrightarrow}a\rangle = 1$
⟨*proof*⟩

lemma *swap-sym*:
$\langle b{\leftrightarrow}a\rangle = \langle a{\leftrightarrow}b\rangle$
⟨*proof*⟩

lemma *swap-self* [*simp*]:
$\langle a{\leftrightarrow}b\rangle * \langle a{\leftrightarrow}b\rangle = 1$
⟨*proof*⟩

lemma *affected-swap*:
$a \neq b \Longrightarrow \mathit{affected}\ \langle a{\leftrightarrow}b\rangle = \{a, b\}$
⟨*proof*⟩

lemma *inverse-swap* [*simp*]:
$\mathit{inverse}\ \langle a{\leftrightarrow}b\rangle = \langle a{\leftrightarrow}b\rangle$
⟨*proof*⟩

71.5 Permutations specified by cycles

fun *cycle* :: *'a list* ⇒ *'a perm* (⟨-⟩)
where
$\langle[]\rangle = 1$
| $\langle[a]\rangle = 1$
| $\langle a \# b \# as\rangle = \langle a \# as\rangle * \langle a{\leftrightarrow}b\rangle$

We do not continue and restrict ourselves to syntax from here. See also introductory note.

71.6 Syntax

bundle *no-permutation-syntax*
begin
no-notation *swap* (⟨-↔-⟩)
no-notation *cycle* (⟨-⟩)
no-notation *apply* (**infixl** ⟨$⟩ *999*)
end

bundle *permutation-syntax*
begin
notation *swap* (⟨-↔-⟩)
notation *cycle* (⟨-⟩)
notation *apply* (**infixl** ⟨$⟩ *999*)
no-notation *apply* (*op* ⟨$⟩)
end

unbundle *no-permutation-syntax*

end

72 Permutations

theory *Permutation*
imports *Multiset*
begin

inductive *perm* :: *'a list* ⇒ *'a list* ⇒ *bool* (- <~~> - [*50*, *50*] *50*)
where
Nil [*intro*!]: [] <~~> []
| *swap* [*intro*!]: *y* # *x* # *l* <~~> *x* # *y* # *l*
| *Cons* [*intro*!]: *xs* <~~> *ys* ⟹ *z* # *xs* <~~> *z* # *ys*
| *trans* [*intro*]: *xs* <~~> *ys* ⟹ *ys* <~~> *zs* ⟹ *xs* <~~> *zs*

proposition *perm-refl* [*iff*]: *l* <~~> *l*
⟨*proof*⟩

72.1 Some examples of rule induction on permutations

proposition *xperm-empty-imp*: [] <~~> *ys* ⟹ *ys* = []
⟨*proof*⟩

This more general theorem is easier to understand!

proposition *perm-length*: *xs* <~~> *ys* ⟹ *length xs* = *length ys*
⟨*proof*⟩

proposition *perm-empty-imp*: [] <~~> *xs* ⟹ *xs* = []
⟨*proof*⟩

proposition *perm-sym*: *xs* <~~> *ys* ⟹ *ys* <~~> *xs*
⟨*proof*⟩

72.2 Ways of making new permutations

We can insert the head anywhere in the list.

proposition *perm-append-Cons*: *a* # *xs* @ *ys* <~~> *xs* @ *a* # *ys*
⟨*proof*⟩

proposition *perm-append-swap*: *xs* @ *ys* <~~> *ys* @ *xs*
⟨*proof*⟩

proposition *perm-append-single*: *a* # *xs* <~~> *xs* @ [*a*]
⟨*proof*⟩

proposition *perm-rev*: $rev\ xs <\sim\sim> xs$
⟨*proof*⟩

proposition *perm-append1*: $xs <\sim\sim> ys \Longrightarrow l\ @\ xs <\sim\sim> l\ @\ ys$
⟨*proof*⟩

proposition *perm-append2*: $xs <\sim\sim> ys \Longrightarrow xs\ @\ l <\sim\sim> ys\ @\ l$
⟨*proof*⟩

72.3 Further results

proposition *perm-empty* [*iff*]: $[] <\sim\sim> xs \longleftrightarrow xs = []$
⟨*proof*⟩

proposition *perm-empty2* [*iff*]: $xs <\sim\sim> [] \longleftrightarrow xs = []$
⟨*proof*⟩

proposition *perm-sing-imp*: $ys <\sim\sim> xs \Longrightarrow xs = [y] \Longrightarrow ys = [y]$
⟨*proof*⟩

proposition *perm-sing-eq* [*iff*]: $ys <\sim\sim> [y] \longleftrightarrow ys = [y]$
⟨*proof*⟩

proposition *perm-sing-eq2* [*iff*]: $[y] <\sim\sim> ys \longleftrightarrow ys = [y]$
⟨*proof*⟩

72.4 Removing elements

proposition *perm-remove*: $x \in set\ ys \Longrightarrow ys <\sim\sim> x\ \#\ remove1\ x\ ys$
⟨*proof*⟩

Congruence rule

proposition *perm-remove-perm*: $xs <\sim\sim> ys \Longrightarrow remove1\ z\ xs <\sim\sim> remove1\ z\ ys$
⟨*proof*⟩

proposition *remove-hd* [*simp*]: $remove1\ z\ (z\ \#\ xs) = xs$
⟨*proof*⟩

proposition *cons-perm-imp-perm*: $z\ \#\ xs <\sim\sim> z\ \#\ ys \Longrightarrow xs <\sim\sim> ys$
⟨*proof*⟩

proposition *cons-perm-eq* [*iff*]: $z\#xs <\sim\sim> z\#ys \longleftrightarrow xs <\sim\sim> ys$
⟨*proof*⟩

proposition *append-perm-imp-perm*: $zs\ @\ xs <\sim\sim> zs\ @\ ys \Longrightarrow xs <\sim\sim> ys$
⟨*proof*⟩

proposition *perm-append1-eq* [*iff*]: $zs\ @\ xs <\sim\sim> zs\ @\ ys \longleftrightarrow xs <\sim\sim> ys$

⟨*proof*⟩

proposition *perm-append2-eq* [*iff*]: *xs* @ *zs* <~~> *ys* @ *zs* ⟷ *xs* <~~> *ys*
⟨*proof*⟩

theorem *mset-eq-perm*: *mset xs* = *mset ys* ⟷ *xs* <~~> *ys*
⟨*proof*⟩

proposition *mset-le-perm-append*: *mset xs* ⊆# *mset ys* ⟷ (∃ *zs*. *xs* @ *zs* <~~> *ys*)
⟨*proof*⟩

proposition *perm-set-eq*: *xs* <~~> *ys* ⟹ *set xs* = *set ys*
⟨*proof*⟩

proposition *perm-distinct-iff*: *xs* <~~> *ys* ⟹ *distinct xs* = *distinct ys*
⟨*proof*⟩

theorem *eq-set-perm-remdups*: *set xs* = *set ys* ⟹ *remdups xs* <~~> *remdups ys*
⟨*proof*⟩

proposition *perm-remdups-iff-eq-set*: *remdups x* <~~> *remdups y* ⟷ *set x* = *set y*
⟨*proof*⟩

theorem *permutation-Ex-bij*:
assumes *xs* <~~> *ys*
shows ∃ *f*. *bij-betw f* {..<*length xs*} {..<*length ys*} ∧ (∀ *i*<*length xs*. *xs* ! *i* = *ys* ! (*f i*))
⟨*proof*⟩

proposition *perm-finite*: *finite* {*B*. *B* <~~> *A*}
⟨*proof*⟩

proposition *perm-swap*:
assumes *i* < *length xs j* < *length xs*
shows *xs*[*i* := *xs* ! *j*, *j* := *xs* ! *i*] <~~> *xs*
⟨*proof*⟩

end

73 Additive group operations on product types

theory *Product-Plus*
imports *Main*
begin

73.1 Operations

instantiation *prod* :: (*zero*, *zero*) *zero*
begin

definition *zero-prod-def*: $0 = (0, 0)$

instance ⟨*proof*⟩
end

instantiation *prod* :: (*plus*, *plus*) *plus*
begin

definition *plus-prod-def*:
$x + y = (fst\ x + fst\ y, snd\ x + snd\ y)$

instance ⟨*proof*⟩
end

instantiation *prod* :: (*minus*, *minus*) *minus*
begin

definition *minus-prod-def*:
$x - y = (fst\ x - fst\ y, snd\ x - snd\ y)$

instance ⟨*proof*⟩
end

instantiation *prod* :: (*uminus*, *uminus*) *uminus*
begin

definition *uminus-prod-def*:
$- x = (- fst\ x, - snd\ x)$

instance ⟨*proof*⟩
end

lemma *fst-zero* [*simp*]: $fst\ 0 = 0$
⟨*proof*⟩

lemma *snd-zero* [*simp*]: $snd\ 0 = 0$
⟨*proof*⟩

lemma *fst-add* [*simp*]: $fst\ (x + y) = fst\ x + fst\ y$
⟨*proof*⟩

lemma *snd-add* [*simp*]: $snd\ (x + y) = snd\ x + snd\ y$
⟨*proof*⟩

lemma *fst-diff* [*simp*]: $fst\ (x - y) = fst\ x - fst\ y$

⟨*proof*⟩

lemma *snd-diff* [*simp*]: *snd* (*x* − *y*) = *snd x* − *snd y*
⟨*proof*⟩

lemma *fst-uminus* [*simp*]: *fst* (− *x*) = − *fst x*
⟨*proof*⟩

lemma *snd-uminus* [*simp*]: *snd* (− *x*) = − *snd x*
⟨*proof*⟩

lemma *add-Pair* [*simp*]: (*a*, *b*) + (*c*, *d*) = (*a* + *c*, *b* + *d*)
⟨*proof*⟩

lemma *diff-Pair* [*simp*]: (*a*, *b*) − (*c*, *d*) = (*a* − *c*, *b* − *d*)
⟨*proof*⟩

lemma *uminus-Pair* [*simp*, *code*]: − (*a*, *b*) = (− *a*, − *b*)
⟨*proof*⟩

73.2 Class instances

instance *prod* :: (*semigroup-add*, *semigroup-add*) *semigroup-add*
⟨*proof*⟩

instance *prod* :: (*ab-semigroup-add*, *ab-semigroup-add*) *ab-semigroup-add*
⟨*proof*⟩

instance *prod* :: (*monoid-add*, *monoid-add*) *monoid-add*
⟨*proof*⟩

instance *prod* :: (*comm-monoid-add*, *comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instance *prod* :: (*cancel-semigroup-add*, *cancel-semigroup-add*) *cancel-semigroup-add*
⟨*proof*⟩

instance *prod* :: (*cancel-ab-semigroup-add*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
⟨*proof*⟩

instance *prod* :: (*cancel-comm-monoid-add*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add*
⟨*proof*⟩

instance *prod* :: (*group-add*, *group-add*) *group-add*
⟨*proof*⟩

instance *prod* :: (*ab-group-add*, *ab-group-add*) *ab-group-add*
⟨*proof*⟩

lemma *fst-sum*: $fst\ (\sum x{\in}A.\ f\ x) = (\sum x{\in}A.\ fst\ (f\ x))$
⟨*proof*⟩

lemma *snd-sum*: $snd\ (\sum x{\in}A.\ f\ x) = (\sum x{\in}A.\ snd\ (f\ x))$
⟨*proof*⟩

lemma *sum-prod*: $(\sum x{\in}A.\ (f\ x,\ g\ x)) = (\sum x{\in}A.\ f\ x,\ \sum x{\in}A.\ g\ x)$
⟨*proof*⟩

end

74 Roots of real quadratics

theory *Quadratic-Discriminant*
imports *Complex-Main*
begin

definition *discrim* :: *real* ⇒ *real* ⇒ *real* ⇒ *real*
  where $discrim\ a\ b\ c \equiv b^2 - 4 * a * c$

lemma *complete-square*:
  fixes $a\ b\ c\ x$ :: *real*
  assumes $a \neq 0$
  shows $a * x^2 + b * x + c = 0 \longleftrightarrow (2 * a * x + b)^2 = discrim\ a\ b\ c$
⟨*proof*⟩

lemma *discriminant-negative*:
  fixes $a\ b\ c\ x$ :: *real*
  assumes $a \neq 0$
    and $discrim\ a\ b\ c < 0$
  shows $a * x^2 + b * x + c \neq 0$
⟨*proof*⟩

lemma *plus-or-minus-sqrt*:
  fixes $x\ y$ :: *real*
  assumes $y \geq 0$
  shows $x^2 = y \longleftrightarrow x = sqrt\ y \vee x = -\ sqrt\ y$
⟨*proof*⟩

lemma *divide-non-zero*:
  fixes $x\ y\ z$ :: *real*
  assumes $x \neq 0$
  shows $x * y = z \longleftrightarrow y = z\ /\ x$
⟨*proof*⟩

lemma *discriminant-nonneg*:
  fixes $a\ b\ c\ x$ :: *real*
  assumes $a \neq 0$
    and $discrim\ a\ b\ c \geq 0$

shows $a * x^2 + b * x + c = 0 \longleftrightarrow$
$x = (-b + sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a) \vee$
$x = (-b - sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a)$
⟨*proof*⟩

lemma *discriminant-zero*:
fixes $a\ b\ c\ x :: real$
assumes $a \neq 0$
and $discrim\ a\ b\ c = 0$
shows $a * x^2 + b * x + c = 0 \longleftrightarrow x = -b\ /\ (2 * a)$
⟨*proof*⟩

theorem *discriminant-iff*:
fixes $a\ b\ c\ x :: real$
assumes $a \neq 0$
shows $a * x^2 + b * x + c = 0 \longleftrightarrow$
$discrim\ a\ b\ c \geq 0 \wedge$
$(x = (-b + sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a) \vee$
$x = (-b - sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a))$
⟨*proof*⟩

lemma *discriminant-nonneg-ex*:
fixes $a\ b\ c :: real$
assumes $a \neq 0$
and $discrim\ a\ b\ c \geq 0$
shows $\exists\ x.\ a * x^2 + b * x + c = 0$
⟨*proof*⟩

lemma *discriminant-pos-ex*:
fixes $a\ b\ c :: real$
assumes $a \neq 0$
and $discrim\ a\ b\ c > 0$
shows $\exists x\ y.\ x \neq y \wedge a * x^2 + b * x + c = 0 \wedge a * y^2 + b * y + c = 0$
⟨*proof*⟩

lemma *discriminant-pos-distinct*:
fixes $a\ b\ c\ x :: real$
assumes $a \neq 0$
and $discrim\ a\ b\ c > 0$
shows $\exists\ y.\ x \neq y \wedge a * y^2 + b * y + c = 0$
⟨*proof*⟩

end

75 Pretty syntax for Quotient operations

theory *Quotient-Syntax*
imports *Main*
begin

notation
rel-conj (**infixr** *OOO 75*) **and**
map-fun (**infixr** *−−−> 55*) **and**
rel-fun (**infixr** *===> 55*)

end

76 Quotient infrastructure for the set type

theory *Quotient-Set*
imports *Quotient-Syntax*
begin

76.1 Contravariant set map (vimage) and set relator, rules for the Quotient package

definition *rel-vset R xs ys* ≡ ∀ *x y*. *R x y* ⟶ *x* ∈ *xs* ⟷ *y* ∈ *ys*

lemma *rel-vset-eq* [*id-simps*]:
rel-vset op = = *op* =
⟨*proof*⟩

lemma *rel-vset-equivp*:
assumes *e*: *equivp R*
shows *rel-vset R xs ys* ⟷ *xs* = *ys* ∧ (∀ *x y*. *x* ∈ *xs* ⟶ *R x y* ⟶ *y* ∈ *xs*)
⟨*proof*⟩

lemma *set-quotient* [*quot-thm*]:
assumes *Quotient3 R Abs Rep*
shows *Quotient3* (*rel-vset R*) (*vimage Rep*) (*vimage Abs*)
⟨*proof*⟩

declare [[*mapQ3 set* = (*rel-vset*, *set-quotient*)]]

lemma *empty-set-rsp*[*quot-respect*]:
rel-vset R {} {}
⟨*proof*⟩

lemma *collect-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows ((*R* ===> *op* =) ===> *rel-vset R*) *Collect Collect*
⟨*proof*⟩

lemma *collect-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows ((*Abs* −−−> *id*) −−−> *op* −‘ *Rep*) *Collect* = *Collect*
⟨*proof*⟩

lemma *union-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows (*rel-vset R* ===> *rel-vset R* ===> *rel-vset R*) *op* ∪ *op* ∪
⟨*proof*⟩

lemma *union-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*op* −‘ *Abs* −−−> *op* −‘ *Abs* −−−> *op* −‘ *Rep*) *op* ∪ = *op* ∪
⟨*proof*⟩

lemma *diff-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows (*rel-vset R* ===> *rel-vset R* ===> *rel-vset R*) *op* − *op* −
⟨*proof*⟩

lemma *diff-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*op* −‘ *Abs* −−−> *op* −‘ *Abs* −−−> *op* −‘ *Rep*) *op* − = *op* −
⟨*proof*⟩

lemma *inter-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows (*rel-vset R* ===> *rel-vset R* ===> *rel-vset R*) *op* ∩ *op* ∩
⟨*proof*⟩

lemma *inter-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*op* −‘ *Abs* −−−> *op* −‘ *Abs* −−−> *op* −‘ *Rep*) *op* ∩ = *op* ∩
⟨*proof*⟩

lemma *mem-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*Rep* −−−> *op* −‘ *Abs* −−−> *id*) *op* ∈ = *op* ∈
⟨*proof*⟩

lemma *mem-rsp*[*quot-respect*]:
shows (*R* ===> *rel-vset R* ===> *op* =) *op* ∈ *op* ∈
⟨*proof*⟩

end

77 Quotient infrastructure for the product type

theory *Quotient-Product*
imports *Quotient-Syntax*
begin

77.1 Rules for the Quotient package

lemma *map-prod-id* [*id-simps*]:
shows *map-prod id id = id*
⟨*proof*⟩

lemma *rel-prod-eq* [*id-simps*]:
shows *rel-prod (op =) (op =) = (op =)*
⟨*proof*⟩

lemma *prod-equivp* [*quot-equiv*]:
assumes *equivp R1*
assumes *equivp R2*
shows *equivp (rel-prod R1 R2)*
⟨*proof*⟩

lemma *prod-quotient* [*quot-thm*]:
assumes *Quotient3 R1 Abs1 Rep1*
assumes *Quotient3 R2 Abs2 Rep2*
shows *Quotient3 (rel-prod R1 R2) (map-prod Abs1 Abs2) (map-prod Rep1 Rep2)*
⟨*proof*⟩

declare [[*mapQ3 prod = (rel-prod, prod-quotient)*]]

lemma *Pair-rsp* [*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows *(R1 ===> R2 ===> rel-prod R1 R2) Pair Pair*
⟨*proof*⟩

lemma *Pair-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows *(Rep1 ---> Rep2 ---> (map-prod Abs1 Abs2)) Pair = Pair*
⟨*proof*⟩

lemma *fst-rsp* [*quot-respect*]:
assumes *Quotient3 R1 Abs1 Rep1*
assumes *Quotient3 R2 Abs2 Rep2*
shows *(rel-prod R1 R2 ===> R1) fst fst*
⟨*proof*⟩

lemma *fst-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows *(map-prod Rep1 Rep2 ---> Abs1) fst = fst*
⟨*proof*⟩

lemma *snd-rsp* [*quot-respect*]:
assumes *Quotient3 R1 Abs1 Rep1*

assumes *Quotient3 R2 Abs2 Rep2*
shows (*rel-prod R1 R2 ===> R2*) *snd snd*
⟨*proof*⟩

lemma *snd-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*map-prod Rep1 Rep2 ---> Abs2*) *snd* = *snd*
⟨*proof*⟩

lemma *case-prod-rsp* [*quot-respect*]:
shows ((*R1 ===> R2 ===> (op =)*) *===> (rel-prod R1 R2) ===> (op =)*) *case-prod case-prod*
⟨*proof*⟩

lemma *split-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows (((*Abs1 ---> Abs2 ---> id*) *---> map-prod Rep1 Rep2 ---> id*) *case-prod*) = *case-prod*
⟨*proof*⟩

lemma [*quot-respect*]:
shows ((*R2 ===> R2 ===> op =*) *===> (R1 ===> R1 ===> op =) ===>*
rel-prod R2 R1 ===> rel-prod R2 R1 ===> op =) *rel-prod rel-prod*
⟨*proof*⟩

lemma [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 abs1 rep1*
and *q2*: *Quotient3 R2 abs2 rep2*
shows ((*abs1 ---> abs1 ---> id*) *---> (abs2 ---> abs2 ---> id) ---> map-prod rep1 rep2 ---> map-prod rep1 rep2 ---> id*) *rel-prod* = *rel-prod*
⟨*proof*⟩

lemma [*quot-preserve*]:
shows(*rel-prod* ((*rep1 ---> rep1 ---> id*) *R1*) ((*rep2 ---> rep2 ---> id*) *R2*)
(*l1*, *l2*) (*r1*, *r2*)) = (*R1* (*rep1 l1*) (*rep1 r1*) ∧ *R2* (*rep2 l2*) (*rep2 r2*))
⟨*proof*⟩

declare *prod.inject*[*quot-preserve*]

end

78 Quotient infrastructure for the option type

theory *Quotient-Option*

imports *Quotient-Syntax*
begin

78.1 Rules for the Quotient package

lemma *rel-option-map1*:
rel-option R (map-option f x) y ⟷ rel-option (λx. R (f x)) x y
⟨*proof*⟩

lemma *rel-option-map2*:
rel-option R x (map-option f y) ⟷ rel-option (λx y. R x (f y)) x y
⟨*proof*⟩

declare
map-option.id [*id-simps*]
option.rel-eq [*id-simps*]

lemma *reflp-rel-option*:
reflp R ⟹ reflp (rel-option R)
⟨*proof*⟩

lemma *option-symp*:
symp R ⟹ symp (rel-option R)
⟨*proof*⟩

lemma *option-transp*:
transp R ⟹ transp (rel-option R)
⟨*proof*⟩

lemma *option-equivp* [*quot-equiv*]:
equivp R ⟹ equivp (rel-option R)
⟨*proof*⟩

lemma *option-quotient* [*quot-thm*]:
assumes *Quotient3 R Abs Rep*
shows *Quotient3 (rel-option R) (map-option Abs) (map-option Rep)*
⟨*proof*⟩

declare [[*mapQ3 option* = (*rel-option*, *option-quotient*)]]

lemma *option-None-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *rel-option R None None*
⟨*proof*⟩

lemma *option-Some-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*R* ===> *rel-option R*) *Some Some*
⟨*proof*⟩

lemma *option-None-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *map-option Abs None = None*
⟨*proof*⟩

lemma *option-Some-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*Rep* −−−> *map-option Abs*) *Some = Some*
⟨*proof*⟩

end

79 Quotient infrastructure for the list type

theory *Quotient-List*
imports *Quotient-Set Quotient-Product Quotient-Option*
begin

79.1 Rules for the Quotient package

lemma *map-id* [*id-simps*]:
map id = id
⟨*proof*⟩

lemma *list-all2-eq* [*id-simps*]:
list-all2 (*op =*) = (*op =*)
⟨*proof*⟩

lemma *reflp-list-all2*:
assumes *reflp R*
shows *reflp* (*list-all2 R*)
⟨*proof*⟩

lemma *list-symp*:
assumes *symp R*
shows *symp* (*list-all2 R*)
⟨*proof*⟩

lemma *list-transp*:
assumes *transp R*
shows *transp* (*list-all2 R*)
⟨*proof*⟩

lemma *list-equivp* [*quot-equiv*]:
equivp R ⟹ *equivp* (*list-all2 R*)
⟨*proof*⟩

lemma *list-quotient3* [*quot-thm*]:

assumes *Quotient3 R Abs Rep*
shows *Quotient3* (*list-all2 R*) (*map Abs*) (*map Rep*)
⟨*proof*⟩

declare [[*mapQ3 list* = (*list-all2*, *list-quotient3*)]]

lemma *cons-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*Rep* −−−> (*map Rep*) −−−> (*map Abs*)) (*op #*) = (*op #*)
⟨*proof*⟩

lemma *cons-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*R* ===> *list-all2 R* ===> *list-all2 R*) (*op #*) (*op #*)
⟨*proof*⟩

lemma *nil-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *map Abs* [] = []
⟨*proof*⟩

lemma *nil-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *list-all2 R* [] []
⟨*proof*⟩

lemma *map-prs-aux*:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows (*map abs2*) (*map* ((*abs1* −−−> *rep2*) *f*) (*map rep1 l*)) = *map f l*
⟨*proof*⟩

lemma *map-prs* [*quot-preserve*]:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows ((*abs1* −−−> *rep2*) −−−> (*map rep1*) −−−> (*map abs2*)) *map* = *map*
and ((*abs1* −−−> *id*) −−−> *map rep1* −−−> *id*) *map* = *map*
⟨*proof*⟩

lemma *map-rsp* [*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows ((*R1* ===> *R2*) ===> (*list-all2 R1*) ===> *list-all2 R2*) *map map*
and ((*R1* ===> *op* =) ===> (*list-all2 R1*) ===> *op* =) *map map*
⟨*proof*⟩

lemma *foldr-prs-aux*:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*

shows *abs2* (*foldr* ((*abs1* −−−> *abs2* −−−> *rep2*) *f*) (*map rep1 l*) (*rep2 e*)) = *foldr f l e*
⟨*proof*⟩

lemma *foldr-prs* [*quot-preserve*]:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows ((*abs1* −−−> *abs2* −−−> *rep2*) −−−> (*map rep1*) −−−> *rep2* −−−> *abs2*) *foldr* = *foldr*
⟨*proof*⟩

lemma *foldl-prs-aux*:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows *abs1* (*foldl* ((*abs1* −−−> *abs2* −−−> *rep1*) *f*) (*rep1 e*) (*map rep2 l*)) = *foldl f e l*
⟨*proof*⟩

lemma *foldl-prs* [*quot-preserve*]:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows ((*abs1* −−−> *abs2* −−−> *rep1*) −−−> *rep1* −−−> (*map rep2*) −−−> *abs1*) *foldl* = *foldl*
⟨*proof*⟩

lemma *foldl-rsp*[*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows ((*R1* ===> *R2* ===> *R1*) ===> *R1* ===> *list-all2 R2* ===> *R1*) *foldl foldl*
⟨*proof*⟩

lemma *foldr-rsp*[*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows ((*R1* ===> *R2* ===> *R2*) ===> *list-all2 R1* ===> *R2* ===> *R2*) *foldr foldr*
⟨*proof*⟩

lemma *list-all2-rsp*:
assumes *r*: ∀ *x y*. *R x y* ⟶ (∀ *a b*. *R a b* ⟶ *S x a* = *T y b*)
and *l1*: *list-all2 R x y*
and *l2*: *list-all2 R a b*
shows *list-all2 S x a* = *list-all2 T y b*
⟨*proof*⟩

lemma [*quot-respect*]:
((*R* ===> *R* ===> *op* =) ===> *list-all2 R* ===> *list-all2 R* ===> *op* =) *list-all2 list-all2*

⟨*proof*⟩

lemma [*quot-preserve*]:
assumes *a*: *Quotient3 R abs1 rep1*
shows ((*abs1* −−−> *abs1* −−−> *id*) −−−> *map rep1* −−−> *map rep1* −−−> *id*) *list-all2* = *list-all2*
⟨*proof*⟩

lemma [*quot-preserve*]:
assumes *a*: *Quotient3 R abs1 rep1*
shows (*list-all2* ((*rep1* −−−> *rep1* −−−> *id*) *R*) *l m*) = (*l* = *m*)
⟨*proof*⟩

lemma *list-all2-find-element*:
assumes *a*: *x* ∈ *set a*
and *b*: *list-all2 R a b*
shows ∃ *y*. (*y* ∈ *set b* ∧ *R x y*)
⟨*proof*⟩

lemma *list-all2-refl*:
assumes *a*: ⋀*x y*. *R x y* = (*R x* = *R y*)
shows *list-all2 R x x*
⟨*proof*⟩

end

80 Quotient infrastructure for the sum type

theory *Quotient-Sum*
imports *Quotient-Syntax*
begin

80.1 Rules for the Quotient package

lemma *rel-sum-map1*:
rel-sum R1 R2 (*map-sum f1 f2 x*) *y* ⟷ *rel-sum* (λ*x*. *R1* (*f1 x*)) (λ*x*. *R2* (*f2 x*)) *x y*
⟨*proof*⟩

lemma *rel-sum-map2*:
rel-sum R1 R2 x (*map-sum f1 f2 y*) ⟷ *rel-sum* (λ*x y*. *R1 x* (*f1 y*)) (λ*x y*. *R2 x* (*f2 y*)) *x y*
⟨*proof*⟩

lemma *map-sum-id* [*id-simps*]:
map-sum id id = *id*
⟨*proof*⟩

lemma *rel-sum-eq* [*id-simps*]:

rel-sum (*op* =) (*op* =) = (*op* =)
⟨*proof*⟩

lemma *reflp-rel-sum*:
reflp R1 ⟹ *reflp R2* ⟹ *reflp* (*rel-sum R1 R2*)
⟨*proof*⟩

lemma *sum-symp*:
symp R1 ⟹ *symp R2* ⟹ *symp* (*rel-sum R1 R2*)
⟨*proof*⟩

lemma *sum-transp*:
transp R1 ⟹ *transp R2* ⟹ *transp* (*rel-sum R1 R2*)
⟨*proof*⟩

lemma *sum-equivp* [*quot-equiv*]:
equivp R1 ⟹ *equivp R2* ⟹ *equivp* (*rel-sum R1 R2*)
⟨*proof*⟩

lemma *sum-quotient* [*quot-thm*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows *Quotient3* (*rel-sum R1 R2*) (*map-sum Abs1 Abs2*) (*map-sum Rep1 Rep2*)
⟨*proof*⟩

declare [[*mapQ3 sum* = (*rel-sum*, *sum-quotient*)]]

lemma *sum-Inl-rsp* [*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*R1* ===> *rel-sum R1 R2*) *Inl Inl*
⟨*proof*⟩

lemma *sum-Inr-rsp* [*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*R2* ===> *rel-sum R1 R2*) *Inr Inr*
⟨*proof*⟩

lemma *sum-Inl-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*Rep1* ---> *map-sum Abs1 Abs2*) *Inl* = *Inl*
⟨*proof*⟩

lemma *sum-Inr-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*Rep2* ---> *map-sum Abs1 Abs2*) *Inr* = *Inr*

⟨*proof*⟩

end

81 Quotient types

theory *Quotient-Type*
imports *Main*
begin

We introduce the notion of quotient types over equivalence relations via type classes.

81.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations $\sim :: 'a \Rightarrow 'a \Rightarrow bool$.

class *eqv* =
  fixes *eqv* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infixl** $\sim$ *50*)

class *equiv* = *eqv* +
  assumes *equiv-refl* [*intro*]: $x \sim x$
    and *equiv-trans* [*trans*]: $x \sim y \Longrightarrow y \sim z \Longrightarrow x \sim z$
    and *equiv-sym* [*sym*]: $x \sim y \Longrightarrow y \sim x$
begin

lemma *equiv-not-sym* [*sym*]: $\neg\ x \sim y \Longrightarrow \neg\ y \sim x$
⟨*proof*⟩

lemma *not-equiv-trans1* [*trans*]: $\neg\ x \sim y \Longrightarrow y \sim z \Longrightarrow \neg\ x \sim z$
⟨*proof*⟩

lemma *not-equiv-trans2* [*trans*]: $x \sim y \Longrightarrow \neg\ y \sim z \Longrightarrow \neg\ x \sim z$
⟨*proof*⟩

end

The quotient type $'a$ *quot* consists of all *equivalence classes* over elements of the base type $'a$.

definition (**in** *eqv*) *quot* = $\{\{x.\ a \sim x\} \mid a.\ True\}$

typedef (**overloaded**) $'a$ *quot* = *quot* :: $'a$::*eqv set set*
  ⟨*proof*⟩

lemma *quotI* [*intro*]: $\{x.\ a \sim x\} \in quot$
  ⟨*proof*⟩

lemma *quotE* [*elim*]:
  assumes $R \in quot$

obtains a **where** $R = \{x.\ a \sim x\}$
⟨*proof*⟩

Abstracted equivalence classes are the canonical representation of elements of a quotient type.

definition *class* :: *'a::equiv* ⇒ *'a quot* (⌊-⌋)
where $\lfloor a \rfloor = Abs\text{-}quot\ \{x.\ a \sim x\}$

theorem *quot-exhaust*: $\exists a.\ A = \lfloor a \rfloor$
⟨*proof*⟩

lemma *quot-cases* [*cases type*: *quot*]:
obtains a **where** $A = \lfloor a \rfloor$
⟨*proof*⟩

81.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

theorem *quot-equality* [*iff?*]: $\lfloor a \rfloor = \lfloor b \rfloor \longleftrightarrow a \sim b$
⟨*proof*⟩

81.3 Picking representing elements

definition *pick* :: *'a::equiv quot* ⇒ *'a*
where $pick\ A = (SOME\ a.\ A = \lfloor a \rfloor)$

theorem *pick-equiv* [*intro*]: $pick\ \lfloor a \rfloor \sim a$
⟨*proof*⟩

theorem *pick-inverse* [*intro*]: $\lfloor pick\ A \rfloor = A$
⟨*proof*⟩

The following rules support canonical function definitions on quotient types (with up to two arguments). Note that the stripped-down version without additional conditions is sufficient most of the time.

theorem *quot-cond-function*:
assumes *eq*: $\bigwedge X\ Y.\ P\ X\ Y \Longrightarrow f\ X\ Y \equiv g\ (pick\ X)\ (pick\ Y)$
and *cong*: $\bigwedge x\ x'\ y\ y'.\ \lfloor x \rfloor = \lfloor x' \rfloor \Longrightarrow \lfloor y \rfloor = \lfloor y' \rfloor$
$\Longrightarrow P\ \lfloor x \rfloor\ \lfloor y \rfloor \Longrightarrow P\ \lfloor x' \rfloor\ \lfloor y' \rfloor \Longrightarrow g\ x\ y = g\ x'\ y'$
and *P*: $P\ \lfloor a \rfloor\ \lfloor b \rfloor$
shows $f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
⟨*proof*⟩

theorem *quot-function*:
assumes $\bigwedge X\ Y.\ f\ X\ Y \equiv g\ (pick\ X)\ (pick\ Y)$
and $\bigwedge x\ x'\ y\ y'.\ \lfloor x \rfloor = \lfloor x' \rfloor \Longrightarrow \lfloor y \rfloor = \lfloor y' \rfloor \Longrightarrow g\ x\ y = g\ x'\ y'$
shows $f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
⟨*proof*⟩

theorem *quot-function'*:
(⋀*X Y*. *f X Y* ≡ *g* (*pick X*) (*pick Y*)) ⟹
(⋀*x x' y y'*. *x* ∼ *x'* ⟹ *y* ∼ *y'* ⟹ *g x y* = *g x' y'*) ⟹
f ⌊*a*⌋ ⌊*b*⌋ = *g a b*
⟨*proof*⟩

end

82 Ramsey's Theorem

theory *Ramsey*
imports *Infinite-Set*
begin

82.1 Finite Ramsey theorem(s)

To distinguish the finite and infinite ones, lower and upper case names are used.

This is the most basic version in terms of cliques and independent sets, i.e. the version for graphs and 2 colours.

definition *clique V E* ⟷ (∀ *v*∈*V*. ∀ *w*∈*V*. *v* ≠ *w* ⟶ {*v*, *w*} ∈ *E*)
definition *indep V E* ⟷ (∀ *v*∈*V*. ∀ *w*∈*V*. *v* ≠ *w* ⟶ {*v*, *w*} ∉ *E*)

lemma *ramsey2*:
∃ *r*≥*1*. ∀ (*V*::'*a set*) (*E*::'*a set set*). *finite V* ∧ *card V* ≥ *r* ⟶
(∃ *R* ⊆ *V*. *card R* = *m* ∧ *clique R E* ∨ *card R* = *n* ∧ *indep R E*)
(**is** ∃ *r*≥*1*. *?R m n r*)
⟨*proof*⟩

82.2 Preliminaries

82.2.1 "Axiom" of Dependent Choice

primrec *choice* :: ('*a* ⇒ *bool*) ⇒ ('*a* × '*a*) *set* ⇒ *nat* ⇒ '*a*
where — An integer-indexed chain of choices
choice-0: *choice P r 0* = (*SOME x*. *P x*)
| *choice-Suc*: *choice P r* (*Suc n*) = (*SOME y*. *P y* ∧ (*choice P r n*, *y*) ∈ *r*)

lemma *choice-n*:
assumes *P0*: *P x0*
and *Pstep*: ⋀*x*. *P x* ⟹ ∃ *y*. *P y* ∧ (*x*, *y*) ∈ *r*
shows *P* (*choice P r n*)
⟨*proof*⟩

lemma *dependent-choice*:
assumes *trans*: *trans r*
and *P0*: *P x0*
and *Pstep*: ⋀*x*. *P x* ⟹ ∃ *y*. *P y* ∧ (*x*, *y*) ∈ *r*

obtains $f :: nat \Rightarrow 'a$ **where** $\bigwedge n.\ P\ (f\ n)$ **and** $\bigwedge n\ m.\ n < m \Longrightarrow (f\ n, f\ m) \in r$
⟨*proof*⟩

82.2.2 Partitions of a Set

definition *part* :: $nat \Rightarrow nat \Rightarrow 'a\ set \Rightarrow ('a\ set \Rightarrow nat) \Rightarrow bool$
— the function f partitions the r-subsets of the typically infinite set Y into s distinct categories.
where $part\ r\ s\ Y\ f \longleftrightarrow (\forall X.\ X \subseteq Y \wedge finite\ X \wedge card\ X = r \longrightarrow f\ X < s)$

For induction, we decrease the value of r in partitions.

lemma *part-Suc-imp-part*:
$[\![infinite\ Y;\ part\ (Suc\ r)\ s\ Y\ f;\ y \in Y]\!] \Longrightarrow part\ r\ s\ (Y - \{y\})\ (\lambda u.\ f\ (insert\ y\ u))$
⟨*proof*⟩

lemma *part-subset*: $part\ r\ s\ YY\ f \Longrightarrow Y \subseteq YY \Longrightarrow part\ r\ s\ Y\ f$
⟨*proof*⟩

82.3 Ramsey's Theorem: Infinitary Version

lemma *Ramsey-induction*:
fixes $s\ r :: nat$
and $YY :: 'a\ set$
and $f :: 'a\ set \Rightarrow nat$
assumes *infinite YY part r s YY f*
shows $\exists Y'\ t'.\ Y' \subseteq YY \wedge infinite\ Y' \wedge t' < s \wedge (\forall X.\ X \subseteq Y' \wedge finite\ X \wedge card\ X = r \longrightarrow f\ X = t')$
⟨*proof*⟩

theorem *Ramsey*:
fixes $s\ r :: nat$
and $Z :: 'a\ set$
and $f :: 'a\ set \Rightarrow nat$
shows
$[\![infinite\ Z;$
$\forall X.\ X \subseteq Z \wedge finite\ X \wedge card\ X = r \longrightarrow f\ X < s]\!]$
$\Longrightarrow \exists Y\ t.\ Y \subseteq Z \wedge infinite\ Y \wedge t < s$
$\wedge (\forall X.\ X \subseteq Y \wedge finite\ X \wedge card\ X = r \longrightarrow f\ X = t)$
⟨*proof*⟩

corollary *Ramsey2*:
fixes $s :: nat$
and $Z :: 'a\ set$
and $f :: 'a\ set \Rightarrow nat$
assumes *infZ*: *infinite Z*
and *part*: $\forall x{\in}Z.\ \forall y{\in}Z.\ x \neq y \longrightarrow f\ \{x, y\} < s$

shows $\exists\, Y\, t.\ Y \subseteq Z \wedge \mathit{infinite}\ Y \wedge t < s \wedge (\forall\, x{\in}Y.\ \forall\, y{\in}Y.\ x{\neq}y \longrightarrow f\ \{x, y\} = t)$
⟨*proof*⟩

82.4 Disjunctive Well-Foundedness

An application of Ramsey's theorem to program termination. See [2].

definition *disj-wf* :: $('a \times 'a)\ \mathit{set} \Rightarrow \mathit{bool}$
where $\mathit{disj\text{-}wf}\ r \longleftrightarrow (\exists\, T.\ \exists\, n{::}\mathit{nat}.\ (\forall\, i{<}n.\ \mathit{wf}\ (T\ i)) \wedge r = (\bigcup i{<}n.\ T\ i))$

definition *transition-idx* :: $(\mathit{nat} \Rightarrow 'a) \Rightarrow (\mathit{nat} \Rightarrow ('a \times 'a)\ \mathit{set}) \Rightarrow \mathit{nat}\ \mathit{set} \Rightarrow \mathit{nat}$
where $\mathit{transition\text{-}idx}\ s\ T\ A = (\mathit{LEAST}\ k.\ \exists\, i\ j.\ A = \{i, j\} \wedge i < j \wedge (s\ j, s\ i) \in T\ k)$

lemma *transition-idx-less*:
assumes $i < j\ (s\ j, s\ i) \in T\ k\ k < n$
shows $\mathit{transition\text{-}idx}\ s\ T\ \{i, j\} < n$
⟨*proof*⟩

lemma *transition-idx-in*:
assumes $i < j\ (s\ j, s\ i) \in T\ k$
shows $(s\ j, s\ i) \in T\ (\mathit{transition\text{-}idx}\ s\ T\ \{i, j\})$
⟨*proof*⟩

To be equal to the union of some well-founded relations is equivalent to being the subset of such a union.

lemma *disj-wf*: $\mathit{disj\text{-}wf}\ r \longleftrightarrow (\exists\, T.\ \exists\, n{::}\mathit{nat}.\ (\forall\, i{<}n.\ \mathit{wf}(T\ i)) \wedge r \subseteq (\bigcup i{<}n.\ T\ i))$
⟨*proof*⟩

theorem *trans-disj-wf-implies-wf*:
assumes *trans r*
and *disj-wf r*
shows *wf r*
⟨*proof*⟩

end

83 Generic reflection and reification

theory *Reflection*
imports *Main*
begin

⟨*ML*⟩

end

theory *Rewrite*
imports *Main*
begin

consts *rewrite-HOLE* :: *'a*::{} (⌑)

lemma *eta-expand*:
fixes *f* :: *'a*::{} ⇒ *'b*::{}
shows *f* ≡ λ*x*. *f x* ⟨*proof*⟩

lemma *rewr-imp*:
assumes *PROP A* ≡ *PROP B*
shows (*PROP A* ⟹ *PROP C*) ≡ (*PROP B* ⟹ *PROP C*)
⟨*proof*⟩

lemma *imp-cong-eq*:
(*PROP A* ⟹ (*PROP B* ⟹ *PROP C*) ≡ (*PROP B'* ⟹ *PROP C'*)) ≡
((*PROP B* ⟹ *PROP A* ⟹ *PROP C*) ≡ (*PROP B'* ⟹ *PROP A* ⟹ *PROP C'*))
⟨*proof*⟩

⟨*ML*⟩

end

84 Assigning lengths to types by type classes

theory *Type-Length*
imports *Numeral-Type*
begin

The aim of this is to allow any type as index type, but to provide a default instantiation for numeral types. This independence requires some duplication with the definitions in `Numeral_Type.thy`.

class *len0* =
fixes *len-of* :: *'a itself* ⇒ *nat*

syntax *-type-length* :: *type* ⇒ *nat* ((*1LENGTH*/(*1'*(*-'*))))

translations *LENGTH*('*a*) ⇀
CONST len-of (*CONST Pure.type* :: *'a itself*)

⟨*ML*⟩

Some theorems are only true on words with length greater 0.

class *len* = *len0* +
assumes *len-gt-0* [*iff*]: *0* < *LENGTH*('*a*)

instantiation *num0* **and** *num1* :: *len0*
begin

definition *len-num0*: *len-of* (- :: *num0 itself*) = *0*
definition *len-num1*: *len-of* (- :: *num1 itself*) = *1*

instance ⟨*proof*⟩

end

instantiation *bit0* **and** *bit1* :: (*len0*) *len0*
begin

definition *len-bit0*: *len-of* (- :: *'a*::*len0 bit0 itself*) = *2* ∗ *LENGTH*(*'a*)
definition *len-bit1*: *len-of* (- :: *'a*::*len0 bit1 itself*) = *2* ∗ *LENGTH*(*'a*) + *1*

instance ⟨*proof*⟩

end

lemmas *len-of-numeral-defs* [*simp*] = *len-num0 len-num1 len-bit0 len-bit1*

instance *num1* :: *len*
⟨*proof*⟩
instance *bit0* :: (*len*) *len*
⟨*proof*⟩
instance *bit1* :: (*len0*) *len*
⟨*proof*⟩

end

85 Saturated arithmetic

theory *Saturated*
imports *Numeral-Type Type-Length*
begin

85.1 The type of saturated naturals

typedef (**overloaded**) (*'a*::*len*) *sat* = {.. *len-of TYPE*(*'a*)}
morphisms *nat-of Abs-sat*
⟨*proof*⟩

lemma *sat-eqI*:
nat-of m = *nat-of n* ⟹ *m* = *n*
⟨*proof*⟩

lemma *sat-eq-iff*:

$m = n \longleftrightarrow nat\text{-}of\ m = nat\text{-}of\ n$
⟨*proof*⟩

lemma *Abs-sat-nat-of* [*code abstype*]:
$Abs\text{-}sat\ (nat\text{-}of\ n) = n$
⟨*proof*⟩

definition $Abs\text{-}sat' :: nat \Rightarrow 'a{::}len\ sat$ **where**
$Abs\text{-}sat'\ n = Abs\text{-}sat\ (min\ (len\text{-}of\ TYPE('a))\ n)$

lemma *nat-of-Abs-sat'* [*simp*]:
$nat\text{-}of\ (Abs\text{-}sat'\ n :: ('a{::}len)\ sat) = min\ (len\text{-}of\ TYPE('a))\ n$
⟨*proof*⟩

lemma *nat-of-le-len-of* [*simp*]:
$nat\text{-}of\ (n :: ('a{::}len)\ sat) \leq len\text{-}of\ TYPE('a)$
⟨*proof*⟩

lemma *min-len-of-nat-of* [*simp*]:
$min\ (len\text{-}of\ TYPE('a))\ (nat\text{-}of\ (n{::}('a{::}len)\ sat)) = nat\text{-}of\ n$
⟨*proof*⟩

lemma *min-nat-of-len-of* [*simp*]:
$min\ (nat\text{-}of\ (n{::}('a{::}len)\ sat))\ (len\text{-}of\ TYPE('a)) = nat\text{-}of\ n$
⟨*proof*⟩

lemma *Abs-sat'-nat-of* [*simp*]:
$Abs\text{-}sat'\ (nat\text{-}of\ n) = n$
⟨*proof*⟩

instantiation $sat :: (len)\ linorder$
begin

definition
less-eq-sat-def: $x \leq y \longleftrightarrow nat\text{-}of\ x \leq nat\text{-}of\ y$

definition
less-sat-def: $x < y \longleftrightarrow nat\text{-}of\ x < nat\text{-}of\ y$

instance
⟨*proof*⟩

end

instantiation $sat :: (len)\ \{minus, comm\text{-}semiring\text{-}1\}$
begin

definition
$0 = Abs\text{-}sat'\ 0$

definition
1 = Abs-sat' 1

lemma *nat-of-zero-sat* [*simp*, *code abstract*]:
nat-of 0 = 0
⟨*proof*⟩

lemma *nat-of-one-sat* [*simp*, *code abstract*]:
nat-of 1 = min 1 (len-of TYPE('a))
⟨*proof*⟩

definition
x + y = Abs-sat' (nat-of x + nat-of y)

lemma *nat-of-plus-sat* [*simp*, *code abstract*]:
nat-of (x + y) = min (nat-of x + nat-of y) (len-of TYPE('a))
⟨*proof*⟩

definition
x − y = Abs-sat' (nat-of x − nat-of y)

lemma *nat-of-minus-sat* [*simp*, *code abstract*]:
nat-of (x − y) = nat-of x − nat-of y
⟨*proof*⟩

definition
*x * y = Abs-sat' (nat-of x * nat-of y)*

lemma *nat-of-times-sat* [*simp*, *code abstract*]:
*nat-of (x * y) = min (nat-of x * nat-of y) (len-of TYPE('a))*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *sat* :: (*len*) *ordered-comm-semiring*
begin

instance
⟨*proof*⟩

end

lemma *Abs-sat'-eq-of-nat*: *Abs-sat' n = of-nat n*
⟨*proof*⟩

abbreviation *Sat* :: *nat* ⇒ *'a*::*len sat* **where**
Sat ≡ *of-nat*

lemma *nat-of-Sat* [*simp*]:
nat-of (*Sat n* :: (*'a*::*len*) *sat*) = *min* (*len-of TYPE*(*'a*)) *n*
⟨*proof*⟩

lemma [*code-abbrev*]:
of-nat (*numeral k*) = (*numeral k* :: *'a*::*len sat*)
⟨*proof*⟩

context
begin

qualified definition *sat-of-nat* :: *nat* ⇒ (*'a*::*len*) *sat*
where [*code-abbrev*]: *sat-of-nat* = *of-nat*

lemma [*code abstract*]:
nat-of (*sat-of-nat n* :: (*'a*::*len*) *sat*) = *min* (*len-of TYPE*(*'a*)) *n*
⟨*proof*⟩

end

instance *sat* :: (*len*) *finite*
⟨*proof*⟩

instantiation *sat* :: (*len*) *equal*
begin

definition *HOL.equal A B* ⟷ *nat-of A* = *nat-of B*

instance
⟨*proof*⟩

end

instantiation *sat* :: (*len*) {*bounded-lattice*, *distrib-lattice*}
begin

definition (*inf* :: *'a sat* ⇒ *'a sat* ⇒ *'a sat*) = *min*
definition (*sup* :: *'a sat* ⇒ *'a sat* ⇒ *'a sat*) = *max*
definition *bot* = (*0* :: *'a sat*)
definition *top* = *Sat* (*len-of TYPE*(*'a*))

instance
⟨*proof*⟩

end

instantiation *sat* :: (*len*) {*Inf*, *Sup*}
begin

definition *Inf* = (*semilattice-neutr-set.F min top* :: *'a sat set* ⇒ *'a sat*)
definition *Sup* = (*semilattice-neutr-set.F max bot* :: *'a sat set* ⇒ *'a sat*)

instance ⟨*proof*⟩

end

interpretation *Inf-sat*: *semilattice-neutr-set min top* :: *'a*::*len sat*
  rewrites *semilattice-neutr-set.F min* (*top* :: *'a sat*) = *Inf*
⟨*proof*⟩

interpretation *Sup-sat*: *semilattice-neutr-set max bot* :: *'a*::*len sat*
  rewrites *semilattice-neutr-set.F max* (*bot* :: *'a sat*) = *Sup*
⟨*proof*⟩

instance *sat* :: (*len*) *complete-lattice*
⟨*proof*⟩

end

86 State monad

theory *State-Monad*
imports *Monad-Syntax*
begin

datatype (*'s*, *'a*) *state* = *State* (*run-state*: *'s* ⇒ (*'a* × *'s*))

lemma *set-state-iff*: *x* ∈ *set-state m* ⟷ (∃ *s s'*. *run-state m s* = (*x*, *s'*))
⟨*proof*⟩

lemma *pred-stateI*[*intro*]:
  assumes ⋀*a s s'*. *run-state m s* = (*a*, *s'*) ⟹ *P a*
  shows *pred-state P m*
⟨*proof*⟩

lemma *pred-stateD*[*dest*]:
  assumes *pred-state P m run-state m s* = (*a*, *s'*)
  shows *P a*
⟨*proof*⟩

lemma *pred-state-run-state*: *pred-state P m* ⟹ *P* (*fst* (*run-state m s*))
⟨*proof*⟩

definition *state-io-rel* :: (*'s* ⇒ *'s* ⇒ *bool*) ⇒ (*'s*, *'a*) *state* ⇒ *bool* **where**
state-io-rel P m = (∀ *s*. *P s* (*snd* (*run-state m s*)))

lemma *state-io-relI*[*intro*]:
assumes $\bigwedge a\ s\ s'.\ run\text{-}state\ m\ s = (a,\ s') \Longrightarrow P\ s\ s'$
shows *state-io-rel P m*
⟨*proof*⟩

lemma *state-io-relD*[*dest*]:
assumes *state-io-rel P m run-state m s* = (*a*, *s*′)
shows *P s s*′
⟨*proof*⟩

lemma *state-io-rel-mono*[*mono*]: $P \leq Q \Longrightarrow state\text{-}io\text{-}rel\ P \leq state\text{-}io\text{-}rel\ Q$
⟨*proof*⟩

lemma *state-ext*:
assumes $\bigwedge s.\ run\text{-}state\ m\ s = run\text{-}state\ n\ s$
shows $m = n$
⟨*proof*⟩

context begin

qualified definition *return* :: *'a* ⇒ (*'s*, *'a*) *state* **where**
return a = *State* (*Pair a*)

lemma *run-state-return*[*simp*]: *run-state* (*return x*) *s* = (*x*, *s*)
⟨*proof*⟩ **definition** *ap* :: (*'s*, *'a* ⇒ *'b*) *state* ⇒ (*'s*, *'a*) *state* ⇒ (*'s*, *'b*) *state* **where**
ap f x = *State* (λ*s*. *case run-state f s of* (*g*, *s*′) ⇒ *case run-state x s*′ *of* (*y*, *s*′′) ⇒ (*g y*, *s*′′))

qualified definition *bind* :: (*'s*, *'a*) *state* ⇒ (*'a* ⇒ (*'s*, *'b*) *state*) ⇒ (*'s*, *'b*) *state* **where**
bind x f = *State* (λ*s*. *case run-state x s of* (*a*, *s*′) ⇒ *run-state* (*f a*) *s*′)

adhoc-overloading *Monad-Syntax.bind bind*

lemma *bind-left-identity*[*simp*]: *bind* (*return a*) *f* = *f a*
⟨*proof*⟩

lemma *bind-right-identity*[*simp*]: *bind m return* = *m*
⟨*proof*⟩

lemma *bind-assoc*[*simp*]: *bind* (*bind m f*) *g* = *bind m* (λ*x*. *bind* (*f x*) *g*)
⟨*proof*⟩

lemma *bind-predI*[*intro*]:
assumes *pred-state* (λ*x*. *pred-state P* (*f x*)) *m*
shows *pred-state P* (*bind m f*)
⟨*proof*⟩ **definition** *get* :: (*'s*, *'s*) *state* **where**
get = *State* (λ*s*. (*s*, *s*))

qualified definition *set* :: *'s* ⇒ (*'s*, *unit*) *state* **where**
set s' = *State* (λ-. ((), *s'*))

lemma *get-set*[*simp*]: *bind get set* = *return* ()
⟨*proof*⟩

lemma *set-set*[*simp*]: *bind* (*set s*) (λ-. *set s'*) = *set s'*
⟨*proof*⟩

lemma *get-bind-set*[*simp*]: *bind get* (λ*s*. *bind* (*set s*) (*f s*)) = *bind get* (λ*s*. *f s* ())
⟨*proof*⟩

lemma *get-const*[*simp*]: *bind get* (λ-. *m*) = *m*
⟨*proof*⟩

fun *traverse-list* :: (*'a* ⇒ (*'b*, *'c*) *state*) ⇒ *'a list* ⇒ (*'b*, *'c list*) *state* **where**
traverse-list - [] = *return* [] |
traverse-list f (*x* # *xs*) = *do* {
  x ← *f x*;
  xs ← *traverse-list f xs*;
  return (*x* # *xs*)
}

lemma *traverse-list-app*[*simp*]: *traverse-list f* (*xs* @ *ys*) = *do* {
  xs ← *traverse-list f xs*;
  ys ← *traverse-list f ys*;
  return (*xs* @ *ys*)
}
⟨*proof*⟩

lemma *traverse-comp*[*simp*]: *traverse-list* (*g* ∘ *f*) *xs* = *traverse-list g* (*map f xs*)
⟨*proof*⟩

abbreviation *mono-state* :: (*'s*::*preorder*, *'a*) *state* ⇒ *bool* **where**
mono-state ≡ *state-io-rel* (*op* ≤)

abbreviation *strict-mono-state* :: (*'s*::*preorder*, *'a*) *state* ⇒ *bool* **where**
strict-mono-state ≡ *state-io-rel* (*op* <)

corollary *strict-mono-implies-mono*: *strict-mono-state m* ⟹ *mono-state m*
⟨*proof*⟩

lemma *return-mono*[*simp*, *intro*]: *mono-state* (*return x*)
⟨*proof*⟩

lemma *get-mono*[*simp*, *intro*]: *mono-state get*
⟨*proof*⟩

lemma *put-mono*:
assumes $\bigwedge x.\ s' \geq x$
shows *mono-state* (*set* s')
⟨*proof*⟩

lemma *map-mono*[*intro*]: *mono-state* $m \Longrightarrow$ *mono-state* (*map-state* f m)
⟨*proof*⟩

lemma *map-strict-mono*[*intro*]: *strict-mono-state* $m \Longrightarrow$ *strict-mono-state* (*map-state* f m)
⟨*proof*⟩

lemma *bind-mono-strong*:
assumes *mono-state* m
assumes $\bigwedge x\ s\ s'.$ *run-state* $m\ s = (x,\ s') \Longrightarrow$ *mono-state* $(f\ x)$
shows *mono-state* (*bind* m f)
⟨*proof*⟩

lemma *bind-strict-mono-strong1*:
assumes *mono-state* m
assumes $\bigwedge x\ s\ s'.$ *run-state* $m\ s = (x,\ s') \Longrightarrow$ *strict-mono-state* $(f\ x)$
shows *strict-mono-state* (*bind* m f)
⟨*proof*⟩

lemma *bind-strict-mono-strong2*:
assumes *strict-mono-state* m
assumes $\bigwedge x\ s\ s'.$ *run-state* $m\ s = (x,\ s') \Longrightarrow$ *mono-state* $(f\ x)$
shows *strict-mono-state* (*bind* m f)
⟨*proof*⟩

corollary *bind-strict-mono-strong*:
assumes *strict-mono-state* m
assumes $\bigwedge x\ s\ s'.$ *run-state* $m\ s = (x,\ s') \Longrightarrow$ *strict-mono-state* $(f\ x)$
shows *strict-mono-state* (*bind* m f)
⟨*proof*⟩ **definition** *update* :: $('s \Rightarrow 's) \Rightarrow ('s,\ unit)$ *state* **where**
update f = *bind get* (*set* $\circ$ f)

lemma *update-id*[*simp*]: *update* $(\lambda x.\ x)$ = *return* ()
⟨*proof*⟩

lemma *update-comp*[*simp*]: *bind* (*update* f) $(\lambda\text{-}.$ *update* $g)$ = *update* $(g \circ f)$
⟨*proof*⟩

lemma *set-update*[*simp*]: *bind* (*set* s) $(\lambda\text{-}.$ *update* $f)$ = *set* $(f\ s)$
⟨*proof*⟩

lemma *set-bind-update*[*simp*]: *bind* (*set* s) $(\lambda\text{-}.$ *bind* (*update* f) $g)$ = *bind* (*set* (f s)) g
⟨*proof*⟩

lemma *update-mono*:
  assumes $\bigwedge x.\ x \leq f\ x$
  shows *mono-state* (*update f*)
⟨*proof*⟩

lemma *update-strict-mono*:
  assumes $\bigwedge x.\ x < f\ x$
  shows *strict-mono-state* (*update f*)
⟨*proof*⟩

end

end

87 Stirling numbers of first and second kind

theory *Stirling*
imports *Main*
begin

87.1 Stirling numbers of the second kind

fun *Stirling* :: *nat* ⇒ *nat* ⇒ *nat*
  where
    Stirling 0 0 = 1
  | *Stirling 0* (*Suc k*) = *0*
  | *Stirling* (*Suc n*) *0 = 0*
  | *Stirling* (*Suc n*) (*Suc k*) = *Suc k* ∗ *Stirling n* (*Suc k*) + *Stirling n k*

lemma *Stirling-1* [*simp*]: *Stirling* (*Suc n*) (*Suc 0*) = *1*
  ⟨*proof*⟩

lemma *Stirling-less* [*simp*]: *n < k* ⟹ *Stirling n k = 0*
  ⟨*proof*⟩

lemma *Stirling-same* [*simp*]: *Stirling n n = 1*
  ⟨*proof*⟩

lemma *Stirling-2-2*: *Stirling* (*Suc* (*Suc n*)) (*Suc* (*Suc 0*)) = *2* ^ *Suc n − 1*
⟨*proof*⟩

lemma *Stirling-2*: *Stirling* (*Suc n*) (*Suc* (*Suc 0*)) = *2* ^ *n − 1*
  ⟨*proof*⟩

87.2 Stirling numbers of the first kind

fun *stirling* :: *nat* ⇒ *nat* ⇒ *nat*
  where

stirling 0 0 = 1
| *stirling 0 (Suc k) = 0*
| *stirling (Suc n) 0 = 0*
| *stirling (Suc n) (Suc k) = n * stirling n (Suc k) + stirling n k*

lemma *stirling-0* [*simp*]: *n > 0 ⟹ stirling n 0 = 0*
⟨*proof*⟩

lemma *stirling-less* [*simp*]: *n < k ⟹ stirling n k = 0*
⟨*proof*⟩

lemma *stirling-same* [*simp*]: *stirling n n = 1*
⟨*proof*⟩

lemma *stirling-Suc-n-1*: *stirling (Suc n) (Suc 0) = fact n*
⟨*proof*⟩

lemma *stirling-Suc-n-n*: *stirling (Suc n) n = Suc n choose 2*
⟨*proof*⟩

lemma *stirling-Suc-n-2*:
assumes *n ≥ Suc 0*
shows *stirling (Suc n) 2 = (∑ k=1..n. fact n div k)*
⟨*proof*⟩

lemma *of-nat-stirling-Suc-n-2*:
assumes *n ≥ Suc 0*
shows *(of-nat (stirling (Suc n) 2)::'a::field-char-0) = fact n * (∑ k=1..n. (1 / of-nat k))*
⟨*proof*⟩

lemma *sum-stirling*: *(∑ k≤n. stirling n k) = fact n*
⟨*proof*⟩

lemma *stirling-pochhammer*:
*(∑ k≤n. of-nat (stirling n k) * x ^ k) = (pochhammer x n :: 'a::comm-semiring-1)*
⟨*proof*⟩

A row of the Stirling number triangle

definition *stirling-row* :: *nat ⇒ nat list*
where *stirling-row n = [stirling n k. k ← [0..<Suc n]]*

lemma *nth-stirling-row*: *k ≤ n ⟹ stirling-row n ! k = stirling n k*
⟨*proof*⟩

lemma *length-stirling-row* [*simp*]: *length (stirling-row n) = Suc n*
⟨*proof*⟩

lemma *stirling-row-nonempty* [*simp*]: *stirling-row n ≠ []*

⟨*proof*⟩

lemma *list-ext*:
assumes *length xs = length ys*
assumes $\bigwedge i.\ i < length\ xs \Longrightarrow xs\ !\ i = ys\ !\ i$
shows *xs = ys*
⟨*proof*⟩

87.2.1 Efficient code

Naively using the defining equations of the Stirling numbers of the first kind to compute them leads to exponential run time due to repeated computations. We can use memoisation to compute them row by row without repeating computations, at the cost of computing a few unneeded values.

As a bonus, this is very efficient for applications where an entire row of Stirling numbers is needed.

definition *zip-with-prev* :: $('a \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'b\ list$
where *zip-with-prev f x xs = map* (λ(*x*,*y*). *f x y*) (*zip* (*x* # *xs*) *xs*)

lemma *zip-with-prev-altdef*:
zip-with-prev f x xs =
(*if xs* = [] *then* [] *else f x* (*hd xs*) # [*f* (*xs*!*i*) (*xs*!(*i*+*1*)). *i* ← [*0*..<*length xs* − *1*]])
⟨*proof*⟩

primrec *stirling-row-aux*
where
stirling-row-aux n y [] = [*1*]
| *stirling-row-aux n y* (*x*#*xs*) = (*y* + *n* ∗ *x*) # *stirling-row-aux n x xs*

lemma *stirling-row-aux-correct*:
stirling-row-aux n y xs = zip-with-prev (λ*a b*. *a* + *n* ∗ *b*) *y xs* @ [*1*]
⟨*proof*⟩

lemma *stirling-row-code* [*code*]:
stirling-row 0 = [*1*]
stirling-row (*Suc n*) = *stirling-row-aux n 0* (*stirling-row n*)
⟨*proof*⟩

lemma *stirling-code* [*code*]:
stirling n k =
(*if k* = *0 then* (*if n* = *0 then 1 else 0*)
else if k > *n then 0*
else if k = *n then 1*
else stirling-row n ! *k*)
⟨*proof*⟩

end

88 A decision procedure for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming

theory *Sum-of-Squares*
imports *Complex-Main*
begin

⟨*ML*⟩

end

89 A table-based implementation of the reflexive transitive closure

theory *Transitive-Closure-Table*
imports *Main*
begin

inductive *rtrancl-path* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow bool$
  for $r :: 'a \Rightarrow 'a \Rightarrow bool$
where
  base: *rtrancl-path r x* [] *x*
| *step*: *r x y* $\Longrightarrow$ *rtrancl-path r y ys z* $\Longrightarrow$ *rtrancl-path r x* $(y \# ys)$ *z*

lemma *rtranclp-eq-rtrancl-path*: $r^{**}\ x\ y \longleftrightarrow (\exists xs.$ *rtrancl-path r x xs y*$)$
⟨*proof*⟩

lemma *rtrancl-path-trans*:
  assumes *xy*: *rtrancl-path r x xs y*
    and *yz*: *rtrancl-path r y ys z*
  shows *rtrancl-path r x* $(xs\ @\ ys)$ *z* ⟨*proof*⟩

lemma *rtrancl-path-appendE*:
  assumes *xz*: *rtrancl-path r x* $(xs\ @\ y \# ys)$ *z*
  obtains *rtrancl-path r x* $(xs\ @\ [y])$ *y* **and** *rtrancl-path r y ys z*
  ⟨*proof*⟩

lemma *rtrancl-path-distinct*:
  assumes *xy*: *rtrancl-path r x xs y*
  obtains xs' **where** *rtrancl-path r x* xs' *y* **and** *distinct* $(x \# xs')$ **and** *set* $xs' \subseteq$ *set xs*

⟨*proof*⟩

inductive *rtrancl-tab* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a* ⇒ '*a* ⇒ *bool*
  for *r* :: '*a* ⇒ '*a* ⇒ *bool*
where
  base: *rtrancl-tab r xs x x*
| *step*: *x* ∉ *set xs* ⟹ *r x y* ⟹ *rtrancl-tab r* (*x* # *xs*) *y z* ⟹ *rtrancl-tab r xs x z*

lemma *rtrancl-path-imp-rtrancl-tab*:
  assumes *path*: *rtrancl-path r x xs y*
    and *x*: *distinct* (*x* # *xs*)
    and *ys*: ({*x*} ∪ *set xs*) ∩ *set ys* = {}
  shows *rtrancl-tab r ys x y*
  ⟨*proof*⟩

lemma *rtrancl-tab-imp-rtrancl-path*:
  assumes *tab*: *rtrancl-tab r ys x y*
  obtains *xs* **where** *rtrancl-path r x xs y*
  ⟨*proof*⟩

lemma *rtranclp-eq-rtrancl-tab-nil*: *r*** *x y* ⟷ *rtrancl-tab r* [] *x y*
⟨*proof*⟩

declare *rtranclp-rtrancl-eq* [*code del*]
declare *rtranclp-eq-rtrancl-tab-nil* [*THEN iffD2*, *code-pred-intro*]

code-pred *rtranclp*
  ⟨*proof*⟩

lemma *rtrancl-path-Range*: ⟦ *rtrancl-path R x xs y*; *z* ∈ *set xs* ⟧ ⟹ *Rangep R z*
⟨*proof*⟩

lemma *rtrancl-path-Range-end*: ⟦ *rtrancl-path R x xs y*; *xs* ≠ [] ⟧ ⟹ *Rangep R y*
⟨*proof*⟩

lemma *rtrancl-path-nth*:
  ⟦ *rtrancl-path R x xs y*; *i* < *length xs* ⟧ ⟹ *R* ((*x* # *xs*) ! *i*) (*xs* ! *i*)
⟨*proof*⟩

lemma *rtrancl-path-last*: ⟦ *rtrancl-path R x xs y*; *xs* ≠ [] ⟧ ⟹ *last xs* = *y*
⟨*proof*⟩

lemma *rtrancl-path-mono*:
  ⟦ *rtrancl-path R x p y*; ⋀*x y*. *R x y* ⟹ *S x y* ⟧ ⟹ *rtrancl-path S x p y*
⟨*proof*⟩

end

90 Binary Tree

theory *Tree*
imports *Main*
begin

datatype *'a tree* =
Leaf (⟨⟩) |
Node 'a tree (*root-val*: *'a*) *'a tree* ((*1*⟨-,/ -,/ -⟩))
datatype-compat *tree*

Can be seen as counting the number of leaves rather than nodes:

definition *size1* :: *'a tree* ⇒ *nat* **where**
size1 t = *size t* + *1*

fun *subtrees* :: *'a tree* ⇒ *'a tree set* **where**
subtrees ⟨⟩ = {⟨⟩} |
subtrees (⟨*l*, *a*, *r*⟩) = *insert* ⟨*l*, *a*, *r*⟩ (*subtrees l* ∪ *subtrees r*)

fun *mirror* :: *'a tree* ⇒ *'a tree* **where**
mirror ⟨⟩ = *Leaf* |
mirror ⟨*l*,*x*,*r*⟩ = ⟨*mirror r*, *x*, *mirror l*⟩

class *height* = **fixes** *height* :: *'a* ⇒ *nat*

instantiation *tree* :: (*type*)*height*
begin

fun *height-tree* :: *'a tree* => *nat* **where**
height Leaf = *0* |
height (*Node t1 a t2*) = *max* (*height t1*) (*height t2*) + *1*

instance ⟨*proof*⟩

end

fun *min-height* :: *'a tree* ⇒ *nat* **where**
min-height Leaf = *0* |
min-height (*Node l - r*) = *min* (*min-height l*) (*min-height r*) + *1*

fun *complete* :: *'a tree* ⇒ *bool* **where**
complete Leaf = *True* |
complete (*Node l x r*) = (*complete l* ∧ *complete r* ∧ *height l* = *height r*)

definition *balanced* :: *'a tree* ⇒ *bool* **where**
balanced t = (*height t* − *min-height t* ≤ *1*)

Weight balanced:

fun *wbalanced* :: *'a tree* ⇒ *bool* **where**
wbalanced Leaf = *True* |

wbalanced (Node l x r) = (abs(int(size l) − int(size r)) ≤ 1 ∧ wbalanced l ∧ wbalanced r)

Internal path length:

fun *ipl* :: *'a tree ⇒ nat* **where**
ipl Leaf = 0 |
ipl (Node l - r) = ipl l + size l + ipl r + size r

fun *preorder* :: *'a tree ⇒ 'a list* **where**
preorder ⟨⟩ = [] |
preorder ⟨l, x, r⟩ = x # preorder l @ preorder r

fun *inorder* :: *'a tree ⇒ 'a list* **where**
inorder ⟨⟩ = [] |
inorder ⟨l, x, r⟩ = inorder l @ [x] @ inorder r

A linear version avoiding append:

fun *inorder2* :: *'a tree ⇒ 'a list ⇒ 'a list* **where**
inorder2 ⟨⟩ xs = xs |
inorder2 ⟨l, x, r⟩ xs = inorder2 l (x # inorder2 r xs)

fun *postorder* :: *'a tree ⇒ 'a list* **where**
postorder ⟨⟩ = [] |
postorder ⟨l, x, r⟩ = postorder l @ postorder r @ [x]

Binary Search Tree:

fun *bst-wrt* :: *('a ⇒ 'a ⇒ bool) ⇒ 'a tree ⇒ bool* **where**
bst-wrt P ⟨⟩ ⟷ True |
bst-wrt P ⟨l, a, r⟩ ⟷
bst-wrt P l ∧ bst-wrt P r ∧ (∀ x∈set-tree l. P x a) ∧ (∀ x∈set-tree r. P a x)

abbreviation *bst* :: *('a::linorder) tree ⇒ bool* **where**
bst ≡ bst-wrt (op <)

fun (**in** *linorder*) *heap* :: *'a tree ⇒ bool* **where**
heap Leaf = True |
heap (Node l m r) =
(heap l ∧ heap r ∧ (∀ x ∈ set-tree l ∪ set-tree r. m ≤ x))

90.1 *map-tree*

lemma *eq-map-tree-Leaf*[*simp*]: *map-tree f t = Leaf ⟷ t = Leaf*
⟨*proof*⟩

lemma *eq-Leaf-map-tree*[*simp*]: *Leaf = map-tree f t ⟷ t = Leaf*
⟨*proof*⟩

90.2 *size*

lemma *size1-simps*[*simp*]:

size1 $\langle\rangle = 1$
size1 $\langle l, x, r\rangle =$ *size1* $l +$ *size1* r
⟨*proof*⟩

lemma *size1-ge0*[*simp*]: $0 <$ *size1* t
⟨*proof*⟩

lemma *eq-size-0*[*simp*]: *size* $t = 0 \longleftrightarrow t =$ *Leaf*
⟨*proof*⟩

lemma *eq-0-size*[*simp*]: $0 =$ *size* $t \longleftrightarrow t =$ *Leaf*
⟨*proof*⟩

lemma *neq-Leaf-iff*: $(t \neq \langle\rangle) = (\exists\, l\ a\ r.\ t = \langle l, a, r\rangle)$
⟨*proof*⟩

lemma *size-map-tree*[*simp*]: *size* (*map-tree* $f\ t$) = *size* t
⟨*proof*⟩

lemma *size1-map-tree*[*simp*]: *size1* (*map-tree* $f\ t$) = *size1* t
⟨*proof*⟩

90.3 *set-tree*

lemma *eq-set-tree-empty*[*simp*]: *set-tree* $t = \{\} \longleftrightarrow t =$ *Leaf*
⟨*proof*⟩

lemma *eq-empty-set-tree*[*simp*]: $\{\}$ = *set-tree* $t \longleftrightarrow t =$ *Leaf*
⟨*proof*⟩

lemma *finite-set-tree*[*simp*]: *finite*(*set-tree* t)
⟨*proof*⟩

90.4 *subtrees*

lemma *neq-subtrees-empty*[*simp*]: *subtrees* $t \neq \{\}$
⟨*proof*⟩

lemma *neq-empty-subtrees*[*simp*]: $\{\} \neq$ *subtrees* t
⟨*proof*⟩

lemma *set-treeE*: $a \in$ *set-tree* $t \Longrightarrow \exists\, l\ r.\ \langle l, a, r\rangle \in$ *subtrees* t
⟨*proof*⟩

lemma *Node-notin-subtrees-if*[*simp*]: $a \notin$ *set-tree* $t \Longrightarrow$ *Node* $l\ a\ r \notin$ *subtrees* t
⟨*proof*⟩

lemma *in-set-tree-if*: $\langle l, a, r\rangle \in$ *subtrees* $t \Longrightarrow a \in$ *set-tree* t
⟨*proof*⟩

90.5 *height* and *min-height*

lemma *eq-height-0*[*simp*]: *height t = 0 ⟷ t = Leaf*
⟨*proof*⟩

lemma *eq-0-height*[*simp*]: *0 = height t ⟷ t = Leaf*
⟨*proof*⟩

lemma *height-map-tree*[*simp*]: *height (map-tree f t) = height t*
⟨*proof*⟩

lemma *height-le-size-tree*: *height t ≤ size (t::'a tree)*
⟨*proof*⟩

lemma *size1-height*: *size1 t ≤ 2 ^ height (t::'a tree)*
⟨*proof*⟩

corollary *size-height*: *size t ≤ 2 ^ height (t::'a tree) − 1*
⟨*proof*⟩

lemma *height-subtrees*: *s ∈ subtrees t ⟹ height s ≤ height t*
⟨*proof*⟩

lemma *min-height-le-height*: *min-height t ≤ height t*
⟨*proof*⟩

lemma *min-height-map-tree*[*simp*]: *min-height (map-tree f t) = min-height t*
⟨*proof*⟩

lemma *min-height-size1*: *2 ^ min-height t ≤ size1 t*
⟨*proof*⟩

90.6 *complete*

lemma *complete-iff-height*: *complete t ⟷ (min-height t = height t)*
⟨*proof*⟩

lemma *size1-if-complete*: *complete t ⟹ size1 t = 2 ^ height t*
⟨*proof*⟩

lemma *size-if-complete*: *complete t ⟹ size t = 2 ^ height t − 1*
⟨*proof*⟩

lemma *complete-if-size1-height*: *size1 t = 2 ^ height t ⟹ complete t*
⟨*proof*⟩

The following proof involves ≥/> chains rather than the standard ≤/< chains. To chain the elements together the transitivity rules *xtrans* are used.

lemma *complete-if-size1-min-height*: *size1 t = 2 ^ min-height t ⟹ complete t*

⟨*proof*⟩

lemma *complete-iff-size1*: *complete t* ⟷ *size1 t* = *2* ^ *height t*
⟨*proof*⟩

Better bounds for incomplete trees:

lemma *size1-height-if-incomplete*:
¬ *complete t* ⟹ *size1 t* < *2* ^ *height t*
⟨*proof*⟩

lemma *min-height-size1-if-incomplete*:
¬ *complete t* ⟹ *2* ^ *min-height t* < *size1 t*
⟨*proof*⟩

90.7 *balanced*

lemma *balanced-subtreeL*: *balanced* (*Node l x r*) ⟹ *balanced l*
⟨*proof*⟩

lemma *balanced-subtreeR*: *balanced* (*Node l x r*) ⟹ *balanced r*
⟨*proof*⟩

lemma *balanced-subtrees*: ⟦ *balanced t*; *s* ∈ *subtrees t* ⟧ ⟹ *balanced s*
⟨*proof*⟩

Balanced trees have optimal height:

lemma *balanced-optimal*:
fixes *t* :: *'a tree* **and** *t'* :: *'b tree*
assumes *balanced t size t* ≤ *size t'* **shows** *height t* ≤ *height t'*
⟨*proof*⟩

90.8 *wbalanced*

lemma *wbalanced-subtrees*: ⟦ *wbalanced t*; *s* ∈ *subtrees t* ⟧ ⟹ *wbalanced s*
⟨*proof*⟩

90.9 *ipl*

The internal path length of a tree:

lemma *ipl-if-complete-int*:
complete t ⟹ *int*(*ipl t*) = (*int*(*height t*) − *2*) ∗ *2*^(*height t*) + *2*
⟨*proof*⟩

90.10 List of entries

lemma *eq-inorder-Nil*[*simp*]: *inorder t* = [] ⟷ *t* = *Leaf*
⟨*proof*⟩

lemma *eq-Nil-inorder*[*simp*]: [] = *inorder t* ⟷ *t* = *Leaf*

⟨*proof*⟩

lemma *set-inorder*[*simp*]: *set* (*inorder t*) = *set-tree t*
⟨*proof*⟩

lemma *set-preorder*[*simp*]: *set* (*preorder t*) = *set-tree t*
⟨*proof*⟩

lemma *set-postorder*[*simp*]: *set* (*postorder t*) = *set-tree t*
⟨*proof*⟩

lemma *length-preorder*[*simp*]: *length* (*preorder t*) = *size t*
⟨*proof*⟩

lemma *length-inorder*[*simp*]: *length* (*inorder t*) = *size t*
⟨*proof*⟩

lemma *length-postorder*[*simp*]: *length* (*postorder t*) = *size t*
⟨*proof*⟩

lemma *preorder-map*: *preorder* (*map-tree f t*) = *map f* (*preorder t*)
⟨*proof*⟩

lemma *inorder-map*: *inorder* (*map-tree f t*) = *map f* (*inorder t*)
⟨*proof*⟩

lemma *postorder-map*: *postorder* (*map-tree f t*) = *map f* (*postorder t*)
⟨*proof*⟩

lemma *inorder2-inorder*: *inorder2 t xs* = *inorder t* @ *xs*
⟨*proof*⟩

90.11 Binary Search Tree

lemma *bst-wrt-mono*: (⋀*x y*. *P x y* ⟹ *Q x y*) ⟹ *bst-wrt P t* ⟹ *bst-wrt Q t*
⟨*proof*⟩

lemma *bst-wrt-le-if-bst*: *bst t* ⟹ *bst-wrt* (*op* ≤) *t*
⟨*proof*⟩

lemma *bst-wrt-le-iff-sorted*: *bst-wrt* (*op* ≤) *t* ⟷ *sorted* (*inorder t*)
⟨*proof*⟩

lemma *bst-iff-sorted-wrt-less*: *bst t* ⟷ *sorted-wrt* (*op* <) (*inorder t*)
⟨*proof*⟩

90.12 *heap*

90.13 *mirror*

lemma *mirror-Leaf*[*simp*]: *mirror t* = ⟨⟩ ⟷ *t* = ⟨⟩
⟨*proof*⟩

lemma *Leaf-mirror*[*simp*]: ⟨⟩ = *mirror t* ⟷ *t* = ⟨⟩
⟨*proof*⟩

lemma *size-mirror*[*simp*]: *size*(*mirror t*) = *size t*
⟨*proof*⟩

lemma *size1-mirror*[*simp*]: *size1*(*mirror t*) = *size1 t*
⟨*proof*⟩

lemma *height-mirror*[*simp*]: *height*(*mirror t*) = *height t*
⟨*proof*⟩

lemma *inorder-mirror*: *inorder*(*mirror t*) = *rev*(*inorder t*)
⟨*proof*⟩

lemma *map-mirror*: *map-tree f* (*mirror t*) = *mirror* (*map-tree f t*)
⟨*proof*⟩

lemma *mirror-mirror*[*simp*]: *mirror*(*mirror t*) = *t*
⟨*proof*⟩

end

91 Multiset of Elements of Binary Tree

theory *Tree-Multiset*
imports *Multiset Tree*
begin

Kept separate from theory *Tree* to avoid importing all of theory *Multiset* into *Tree*. Should be merged if *Multiset* ever becomes part of *Main*.

fun *mset-tree* :: *'a tree* ⇒ *'a multiset* **where**
mset-tree Leaf = {#} |
mset-tree (*Node l a r*) = {#*a*#} + *mset-tree l* + *mset-tree r*

fun *subtrees-mset* :: *'a tree* ⇒ *'a tree multiset* **where**
subtrees-mset Leaf = {#*Leaf*#} |
subtrees-mset (*Node l x r*) = *add-mset* (*Node l x r*) (*subtrees-mset l* + *subtrees-mset r*)

lemma *mset-tree-empty-iff*[*simp*]: *mset-tree t* = {#} ⟷ *t* = *Leaf*

⟨*proof*⟩

lemma *set-mset-tree*[*simp*]: *set-mset* (*mset-tree t*) = *set-tree t*
⟨*proof*⟩

lemma *size-mset-tree*[*simp*]: *size*(*mset-tree t*) = *size t*
⟨*proof*⟩

lemma *mset-map-tree*: *mset-tree* (*map-tree f t*) = *image-mset f* (*mset-tree t*)
⟨*proof*⟩

lemma *mset-iff-set-tree*: *x* ∈# *mset-tree t* ⟷ *x* ∈ *set-tree t*
⟨*proof*⟩

lemma *mset-preorder*[*simp*]: *mset* (*preorder t*) = *mset-tree t*
⟨*proof*⟩

lemma *mset-inorder*[*simp*]: *mset* (*inorder t*) = *mset-tree t*
⟨*proof*⟩

lemma *map-mirror*: *mset-tree* (*mirror t*) = *mset-tree t*
⟨*proof*⟩

lemma *in-subtrees-mset-iff*[*simp*]: *s* ∈# *subtrees-mset t* ⟷ *s* ∈ *subtrees t*
⟨*proof*⟩

end

theory *Tree-Real*
imports
Complex-Main
Tree
begin

This theory is separate from *Tree* because the former is discrete and builds on *Main* whereas this theory builds on *Complex-Main*.

lemma *size1-height-log*: *log 2* (*size1 t*) ≤ *height t*
⟨*proof*⟩

lemma *min-height-size1-log*: *min-height t* ≤ *log 2* (*size1 t*)
⟨*proof*⟩

lemma *size1-log-if-complete*: *complete t* ⟹ *height t* = *log 2* (*size1 t*)
⟨*proof*⟩

lemma *min-height-size1-log-if-incomplete*:
¬ *complete t* ⟹ *min-height t* < *log 2* (*size1 t*)
⟨*proof*⟩

lemma *min-height-balanced*: **assumes** *balanced t*
shows *min-height t = nat(floor(log 2 (size1 t)))*
⟨*proof*⟩

lemma *height-balanced*: **assumes** *balanced t*
shows *height t = nat(ceiling(log 2 (size1 t)))*
⟨*proof*⟩

lemma *balanced-Node-if-wbal1*:
assumes *balanced l balanced r size l = size r + 1*
shows *balanced ⟨l, x, r⟩*
⟨*proof*⟩

lemma *balanced-sym*: *balanced ⟨l, x, r⟩ ⟹ balanced ⟨r, y, l⟩*
⟨*proof*⟩

lemma *balanced-Node-if-wbal2*:
assumes *balanced l balanced r abs(int(size l) − int(size r)) ≤ 1*
shows *balanced ⟨l, x, r⟩*
⟨*proof*⟩

lemma *balanced-if-wbalanced*: *wbalanced t ⟹ balanced t*
⟨*proof*⟩

end

92 Unordered pairs

theory *Uprod* **imports** *Main* **begin**

typedef *('a, 'b) commute = {f :: 'a ⇒ 'a ⇒ 'b. ∀ x y. f x y = f y x}*
morphisms *apply-commute Abs-commute*
⟨*proof*⟩

setup-lifting *type-definition-commute*

lemma *apply-commute-commute*: *apply-commute f x y = apply-commute f y x*
⟨*proof*⟩

context includes *lifting-syntax* **begin**

lift-definition *rel-commute :: ('a ⇒ 'b ⇒ bool) ⇒ ('c ⇒ 'd ⇒ bool) ⇒ ('a, 'c) commute ⇒ ('b, 'd) commute ⇒ bool*
is *λA B. A ===> A ===> B* ⟨*proof*⟩

end

definition *eq-upair* :: $('a \times 'a) \Rightarrow ('a \times 'a) \Rightarrow bool$
where $eq\text{-}upair = (\lambda(a,\ b)\ (c,\ d).\ a = c \wedge b = d \vee a = d \wedge b = c)$

lemma *eq-upair-simps* [*simp*]:
$eq\text{-}upair\ (a,\ b)\ (c,\ d) \longleftrightarrow a = c \wedge b = d \vee a = d \wedge b = c$
⟨*proof*⟩

lemma *equivp-eq-upair*: *equivp eq-upair*
⟨*proof*⟩

quotient-type $'a\ uprod = 'a \times 'a\ /\ eq\text{-}upair$ ⟨*proof*⟩

lift-definition *Upair* :: $'a \Rightarrow 'a \Rightarrow 'a\ uprod$ **is** *Pair* **parametric** *Pair-transfer*[*of A A* **for** *A*] ⟨*proof*⟩

lemma *uprod-exhaust* [*case-names Upair*, *cases type*: *uprod*]:
obtains *a b* **where** $x = Upair\ a\ b$
⟨*proof*⟩

lemma *Upair-inject* [*simp*]: $Upair\ a\ b = Upair\ c\ d \longleftrightarrow a = c \wedge b = d \vee a = d \wedge b = c$
⟨*proof*⟩

code-datatype *Upair*

lift-definition *case-uprod* :: $('a,\ 'b)\ commute \Rightarrow 'a\ uprod \Rightarrow 'b$ **is** *case-prod*
parametric *case-prod-transfer*[*of A A* **for** *A*] ⟨*proof*⟩

lemma *case-uprod-simps* [*simp*, *code*]: $case\text{-}uprod\ f\ (Upair\ x\ y) = apply\text{-}commute\ f\ x\ y$
⟨*proof*⟩

lemma *uprod-split*: $P\ (case\text{-}uprod\ f\ x) \longleftrightarrow (\forall\ a\ b.\ x = Upair\ a\ b \longrightarrow P\ (apply\text{-}commute\ f\ a\ b))$
⟨*proof*⟩

lemma *uprod-split-asm*: $P\ (case\text{-}uprod\ f\ x) \longleftrightarrow \neg\ (\exists\ a\ b.\ x = Upair\ a\ b \wedge \neg\ P\ (apply\text{-}commute\ f\ a\ b))$
⟨*proof*⟩

lift-definition *not-equal* :: $('a,\ bool)\ commute$ **is** $op \neq$ ⟨*proof*⟩

lemma *apply-not-equal* [*simp*]: $apply\text{-}commute\ not\text{-}equal\ x\ y \longleftrightarrow x \neq y$
⟨*proof*⟩

definition *proper-uprod* :: $'a\ uprod \Rightarrow bool$
where $proper\text{-}uprod = case\text{-}uprod\ not\text{-}equal$

lemma *proper-uprod-simps* [*simp*, *code*]: $proper\text{-}uprod\ (Upair\ x\ y) \longleftrightarrow x \neq y$

⟨*proof*⟩

context includes *lifting-syntax* **begin**

private lemma *set-uprod-parametric'*:
(*rel-prod A A* ===> *rel-set A*) (λ(*a*, *b*). {*a*, *b*}) (λ(*a*, *b*). {*a*, *b*})
⟨*proof*⟩

lift-definition *set-uprod* :: *'a uprod* ⇒ *'a set* **is** λ(*a*, *b*). {*a*, *b*}
parametric *set-uprod-parametric'* ⟨*proof*⟩

lemma *set-uprod-simps* [*simp*, *code*]: *set-uprod* (*Upair x y*) = {*x*, *y*}
⟨*proof*⟩

lemma *finite-set-uprod* [*simp*]: *finite* (*set-uprod x*)
⟨*proof*⟩ **lemma** *map-uprod-parametric'*:
((*A* ===> *B*) ===> *rel-prod A A* ===> *rel-prod B B*) (λ*f*. *map-prod f f*) (λ*f*. *map-prod f f*)
⟨*proof*⟩

lift-definition *map-uprod* :: (*'a* ⇒ *'b*) ⇒ *'a uprod* ⇒ *'b uprod* **is** λ*f*. *map-prod f f*
parametric *map-uprod-parametric'* ⟨*proof*⟩

lemma *map-uprod-simps* [*simp*, *code*]: *map-uprod f* (*Upair x y*) = *Upair* (*f x*) (*f y*)
⟨*proof*⟩ **lemma** *rel-uprod-transfer'*:
((*A* ===> *B* ===> *op* =) ===> *rel-prod A A* ===> *rel-prod B B* ===> *op* =)
(λ*R* (*a*, *b*) (*c*, *d*). *R a c* ∧ *R b d* ∨ *R a d* ∧ *R b c*) (λ*R* (*a*, *b*) (*c*, *d*). *R a c* ∧ *R b d* ∨ *R a d* ∧ *R b c*)
⟨*proof*⟩

lift-definition *rel-uprod* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a uprod* ⇒ *'b uprod* ⇒ *bool*
is λ*R* (*a*, *b*) (*c*, *d*). *R a c* ∧ *R b d* ∨ *R a d* ∧ *R b c* **parametric** *rel-uprod-transfer'*
⟨*proof*⟩

lemma *rel-uprod-simps* [*simp*, *code*]:
rel-uprod R (*Upair a b*) (*Upair c d*) ⟷ *R a c* ∧ *R b d* ∨ *R a d* ∧ *R b c*
⟨*proof*⟩

lemma *Upair-parametric* [*transfer-rule*]: (*A* ===> *A* ===> *rel-uprod A*) *Upair Upair*
⟨*proof*⟩

lemma *case-uprod-parametric* [*transfer-rule*]:
(*rel-commute A B* ===> *rel-uprod A* ===> *B*) *case-uprod case-uprod*
⟨*proof*⟩

end

bnf *uprod*: *'a uprod*
map: *map-uprod*
sets: *set-uprod*
bd: *natLeq*
rel: *rel-uprod*
⟨*proof*⟩

lemma *pred-uprod-code* [*simp*, *code*]: *pred-uprod P* (*Upair x y*) ⟷ *P x* ∧ *P y*
⟨*proof*⟩

instantiation *uprod* :: (*equal*) *equal* **begin**

definition *equal-uprod* :: *'a uprod* ⇒ *'a uprod* ⇒ *bool*
where *equal-uprod* = *op* =

lemma *equal-uprod-code* [*code*]:
HOL.equal (*Upair x y*) (*Upair z u*) ⟷ *x* = *z* ∧ *y* = *u* ∨ *x* = *u* ∧ *y* = *z*
⟨*proof*⟩

instance ⟨*proof*⟩
end

quickcheck-generator *uprod constructors*: *Upair*

lemma *UNIV-uprod*: *UNIV* = (λ*x*. *Upair x x*) ‘ *UNIV* ∪ (λ(*x*, *y*). *Upair x y*) ‘ *Sigma UNIV* (λ*x*. *UNIV* − {*x*})
⟨*proof*⟩

context begin
private lift-definition *upair-inv* :: *'a uprod* ⇒ *'a*
is λ(*x*, *y*). *if x* = *y then x else undefined* ⟨*proof*⟩

lemma *finite-UNIV-prod* [*simp*]:
finite (*UNIV* :: *'a uprod set*) ⟷ *finite* (*UNIV* :: *'a set*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

lemma *card-UNIV-uprod*:
card (*UNIV* :: *'a uprod set*) = *card* (*UNIV* :: *'a set*) ∗ (*card* (*UNIV* :: *'a set*) + *1*) *div 2*
(**is** *?UPROD* = *?A* ∗ - *div* -)
⟨*proof*⟩

end

93 Pointwise order on product types

theory *Product-Order*
imports *Product-Plus*
begin

93.1 Pointwise ordering

instantiation *prod* :: (*ord*, *ord*) *ord*
begin

definition
$x \leq y \longleftrightarrow fst\ x \leq fst\ y \wedge snd\ x \leq snd\ y$

definition
$(x::'a \times 'b) < y \longleftrightarrow x \leq y \wedge \neg\ y \leq x$

instance ⟨*proof*⟩

end

lemma *fst-mono*: $x \leq y \Longrightarrow fst\ x \leq fst\ y$
⟨*proof*⟩

lemma *snd-mono*: $x \leq y \Longrightarrow snd\ x \leq snd\ y$
⟨*proof*⟩

lemma *Pair-mono*: $x \leq x' \Longrightarrow y \leq y' \Longrightarrow (x, y) \leq (x', y')$
⟨*proof*⟩

lemma *Pair-le* [*simp*]: $(a, b) \leq (c, d) \longleftrightarrow a \leq c \wedge b \leq d$
⟨*proof*⟩

instance *prod* :: (*preorder*, *preorder*) *preorder*
⟨*proof*⟩

instance *prod* :: (*order*, *order*) *order*
⟨*proof*⟩

93.2 Binary infimum and supremum

instantiation *prod* :: (*inf*, *inf*) *inf*
begin

definition $inf\ x\ y = (inf\ (fst\ x)\ (fst\ y), inf\ (snd\ x)\ (snd\ y))$

lemma *inf-Pair-Pair* [*simp*]: $inf\ (a, b)\ (c, d) = (inf\ a\ c, inf\ b\ d)$
⟨*proof*⟩

lemma *fst-inf* [*simp*]: $fst\ (inf\ x\ y) = inf\ (fst\ x)\ (fst\ y)$

⟨*proof*⟩

lemma *snd-inf* [*simp*]: *snd* (*inf x y*) = *inf* (*snd x*) (*snd y*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *prod* :: (*semilattice-inf*, *semilattice-inf*) *semilattice-inf*
⟨*proof*⟩

instantiation *prod* :: (*sup*, *sup*) *sup*
begin

definition
sup x y = (*sup* (*fst x*) (*fst y*), *sup* (*snd x*) (*snd y*))

lemma *sup-Pair-Pair* [*simp*]: *sup* (*a*, *b*) (*c*, *d*) = (*sup a c*, *sup b d*)
⟨*proof*⟩

lemma *fst-sup* [*simp*]: *fst* (*sup x y*) = *sup* (*fst x*) (*fst y*)
⟨*proof*⟩

lemma *snd-sup* [*simp*]: *snd* (*sup x y*) = *sup* (*snd x*) (*snd y*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *prod* :: (*semilattice-sup*, *semilattice-sup*) *semilattice-sup*
⟨*proof*⟩

instance *prod* :: (*lattice*, *lattice*) *lattice* ⟨*proof*⟩

instance *prod* :: (*distrib-lattice*, *distrib-lattice*) *distrib-lattice*
⟨*proof*⟩

93.3 Top and bottom elements

instantiation *prod* :: (*top*, *top*) *top*
begin

definition
top = (*top*, *top*)

instance ⟨*proof*⟩

end

lemma *fst-top* [*simp*]: *fst top* = *top*
⟨*proof*⟩

lemma *snd-top* [*simp*]: *snd top* = *top*
⟨*proof*⟩

lemma *Pair-top-top*: (*top*, *top*) = *top*
⟨*proof*⟩

instance *prod* :: (*order-top*, *order-top*) *order-top*
⟨*proof*⟩

instantiation *prod* :: (*bot*, *bot*) *bot*
begin

definition
bot = (*bot*, *bot*)

instance ⟨*proof*⟩

end

lemma *fst-bot* [*simp*]: *fst bot* = *bot*
⟨*proof*⟩

lemma *snd-bot* [*simp*]: *snd bot* = *bot*
⟨*proof*⟩

lemma *Pair-bot-bot*: (*bot*, *bot*) = *bot*
⟨*proof*⟩

instance *prod* :: (*order-bot*, *order-bot*) *order-bot*
⟨*proof*⟩

instance *prod* :: (*bounded-lattice*, *bounded-lattice*) *bounded-lattice* ⟨*proof*⟩

instance *prod* :: (*boolean-algebra*, *boolean-algebra*) *boolean-algebra*
⟨*proof*⟩

93.4 Complete lattice operations

instantiation *prod* :: (*Inf*, *Inf*) *Inf*
begin

definition *Inf A* = (*INF x*:*A*. *fst x*, *INF x*:*A*. *snd x*)

instance ⟨*proof*⟩

end

instantiation *prod* :: (*Sup*, *Sup*) *Sup*
begin

definition *Sup A* = (*SUP x*:*A*. *fst x*, *SUP x*:*A*. *snd x*)

instance ⟨*proof*⟩

end

instance *prod* :: (*conditionally-complete-lattice*, *conditionally-complete-lattice*)
conditionally-complete-lattice
⟨*proof*⟩

instance *prod* :: (*complete-lattice*, *complete-lattice*) *complete-lattice*
⟨*proof*⟩

lemma *fst-Sup*: *fst* (*Sup A*) = (*SUP x*:*A*. *fst x*)
⟨*proof*⟩

lemma *snd-Sup*: *snd* (*Sup A*) = (*SUP x*:*A*. *snd x*)
⟨*proof*⟩

lemma *fst-Inf*: *fst* (*Inf A*) = (*INF x*:*A*. *fst x*)
⟨*proof*⟩

lemma *snd-Inf*: *snd* (*Inf A*) = (*INF x*:*A*. *snd x*)
⟨*proof*⟩

lemma *fst-SUP*: *fst* (*SUP x*:*A*. *f x*) = (*SUP x*:*A*. *fst* (*f x*))
⟨*proof*⟩

lemma *snd-SUP*: *snd* (*SUP x*:*A*. *f x*) = (*SUP x*:*A*. *snd* (*f x*))
⟨*proof*⟩

lemma *fst-INF*: *fst* (*INF x*:*A*. *f x*) = (*INF x*:*A*. *fst* (*f x*))
⟨*proof*⟩

lemma *snd-INF*: *snd* (*INF x*:*A*. *f x*) = (*INF x*:*A*. *snd* (*f x*))
⟨*proof*⟩

lemma *SUP-Pair*: (*SUP x*:*A*. (*f x*, *g x*)) = (*SUP x*:*A*. *f x*, *SUP x*:*A*. *g x*)
⟨*proof*⟩

lemma *INF-Pair*: (*INF x*:*A*. (*f x*, *g x*)) = (*INF x*:*A*. *f x*, *INF x*:*A*. *g x*)
⟨*proof*⟩

Alternative formulations for set infima and suprema over the product of two complete lattices:

lemma *INF-prod-alt-def*:
INFIMUM A f = (*INFIMUM A* (*fst o f*), *INFIMUM A* (*snd o f*))
⟨*proof*⟩

lemma *SUP-prod-alt-def*:
SUPREMUM A f = (*SUPREMUM A* (*fst o f*), *SUPREMUM A* (*snd o f*))
⟨*proof*⟩

93.5 Complete distributive lattices

instance *prod* :: (*complete-distrib-lattice*, *complete-distrib-lattice*) *complete-distrib-lattice*
⟨*proof*⟩

93.6 Bekic's Theorem

Simultaneous fixed points over pairs can be written in terms of separate fixed points. Transliterated from HOLCF.Fix by Peter Gammie

lemma *lfp-prod*:
fixes *F* :: *'a*::*complete-lattice* × *'b*::*complete-lattice* ⇒ *'a* × *'b*
assumes *mono F*
shows *lfp F* = (*lfp* (λ*x*. *fst* (*F* (*x*, *lfp* (λ*y*. *snd* (*F* (*x*, *y*)))))),
(*lfp* (λ*y*. *snd* (*F* (*lfp* (λ*x*. *fst* (*F* (*x*, *lfp* (λ*y*. *snd* (*F* (*x*, *y*)))))), *y*)))))
(**is** *lfp F* = (*?x*, *?y*))
⟨*proof*⟩

lemma *gfp-prod*:
fixes *F* :: *'a*::*complete-lattice* × *'b*::*complete-lattice* ⇒ *'a* × *'b*
assumes *mono F*
shows *gfp F* = (*gfp* (λ*x*. *fst* (*F* (*x*, *gfp* (λ*y*. *snd* (*F* (*x*, *y*)))))),
(*gfp* (λ*y*. *snd* (*F* (*gfp* (λ*x*. *fst* (*F* (*x*, *gfp* (λ*y*. *snd* (*F* (*x*, *y*)))))),
y)))))
(**is** *gfp F* = (*?x*, *?y*))
⟨*proof*⟩

end

theory *Finite-Lattice*
imports *Product-Order*
begin

A non-empty finite lattice is a complete lattice. Since types are never empty in Isabelle/HOL, a type of classes *finite* and *lattice* should also have class *complete-lattice*. A type class is defined that extends classes *finite* and *lattice* with the operators *bot*, *top*, *Inf*, and *Sup*, along with assumptions that

define these operators in terms of the ones of classes *finite* and *lattice*. The resulting class is a subclass of *complete-lattice*.

class *finite-lattice-complete* = *finite* + *lattice* + *bot* + *top* + *Inf* + *Sup* +
  assumes *bot-def*: *bot* = *Inf-fin UNIV*
  assumes *top-def*: *top* = *Sup-fin UNIV*
  assumes *Inf-def*: *Inf A* = *Finite-Set.fold inf top A*
  assumes *Sup-def*: *Sup A* = *Finite-Set.fold sup bot A*

The definitional assumptions on the operators *bot* and *top* of class *finite-lattice-complete* ensure that they yield bottom and top.

lemma *finite-lattice-complete-bot-least*: (*bot*::'*a*::*finite-lattice-complete*) ≤ *x*
  ⟨*proof*⟩

instance *finite-lattice-complete* ⊆ *order-bot*
  ⟨*proof*⟩

lemma *finite-lattice-complete-top-greatest*: (*top*::'*a*::*finite-lattice-complete*) ≥ *x*
  ⟨*proof*⟩

instance *finite-lattice-complete* ⊆ *order-top*
  ⟨*proof*⟩

instance *finite-lattice-complete* ⊆ *bounded-lattice* ⟨*proof*⟩

The definitional assumptions on the operators *Inf* and *Sup* of class *finite-lattice-complete* ensure that they yield infimum and supremum.

lemma *finite-lattice-complete-Inf-empty*: *Inf* {} = (*top* :: '*a*::*finite-lattice-complete*)
  ⟨*proof*⟩

lemma *finite-lattice-complete-Sup-empty*: *Sup* {} = (*bot* :: '*a*::*finite-lattice-complete*)
  ⟨*proof*⟩

lemma *finite-lattice-complete-Inf-insert*:
  fixes *A* :: '*a*::*finite-lattice-complete set*
  shows *Inf* (*insert x A*) = *inf x* (*Inf A*)
⟨*proof*⟩

lemma *finite-lattice-complete-Sup-insert*:
  fixes *A* :: '*a*::*finite-lattice-complete set*
  shows *Sup* (*insert x A*) = *sup x* (*Sup A*)
⟨*proof*⟩

lemma *finite-lattice-complete-Inf-lower*:
  (*x*::'*a*::*finite-lattice-complete*) ∈ *A* ⟹ *Inf A* ≤ *x*
  ⟨*proof*⟩

lemma *finite-lattice-complete-Inf-greatest*:
  ∀ *x*::'*a*::*finite-lattice-complete* ∈ *A*. *z* ≤ *x* ⟹ *z* ≤ *Inf A*
  ⟨*proof*⟩

lemma *finite-lattice-complete-Sup-upper*:
$(x::'a::\textit{finite-lattice-complete}) \in A \Longrightarrow \textit{Sup}\ A \geq x$
⟨*proof*⟩

lemma *finite-lattice-complete-Sup-least*:
$\forall\, x::'a::\textit{finite-lattice-complete} \in A.\ z \geq x \Longrightarrow z \geq \textit{Sup}\ A$
⟨*proof*⟩

instance *finite-lattice-complete* ⊆ *complete-lattice*
⟨*proof*⟩

The product of two finite lattices is already a finite lattice.

lemma *finite-bot-prod*:
$(\textit{bot} :: ('a::\textit{finite-lattice-complete} \times 'b::\textit{finite-lattice-complete})) =$
Inf-fin UNIV
⟨*proof*⟩

lemma *finite-top-prod*:
$(\textit{top} :: ('a::\textit{finite-lattice-complete} \times 'b::\textit{finite-lattice-complete})) =$
Sup-fin UNIV
⟨*proof*⟩

lemma *finite-Inf-prod*:
$\textit{Inf}(A :: ('a::\textit{finite-lattice-complete} \times 'b::\textit{finite-lattice-complete})\ \textit{set}) =$
Finite-Set.fold inf top A
⟨*proof*⟩

lemma *finite-Sup-prod*:
$\textit{Sup}\ (A :: ('a::\textit{finite-lattice-complete} \times 'b::\textit{finite-lattice-complete})\ \textit{set}) =$
Finite-Set.fold sup bot A
⟨*proof*⟩

instance *prod* :: (*finite-lattice-complete*, *finite-lattice-complete*) *finite-lattice-complete*
⟨*proof*⟩

Functions with a finite domain and with a finite lattice as codomain already form a finite lattice.

lemma *finite-bot-fun*: $(\textit{bot} :: ('a::\textit{finite} \Rightarrow 'b::\textit{finite-lattice-complete})) = \textit{Inf-fin UNIV}$
⟨*proof*⟩

lemma *finite-top-fun*: $(\textit{top} :: ('a::\textit{finite} \Rightarrow 'b::\textit{finite-lattice-complete})) = \textit{Sup-fin UNIV}$
⟨*proof*⟩

lemma *finite-Inf-fun*:
$\textit{Inf}\ (A::('a::\textit{finite} \Rightarrow 'b::\textit{finite-lattice-complete})\ \textit{set}) =$
Finite-Set.fold inf top A

⟨*proof*⟩

lemma *finite-Sup-fun*:
Sup (*A*::(*'a*::*finite* ⇒ *'b*::*finite-lattice-complete*) *set*) =
Finite-Set.fold sup bot A
⟨*proof*⟩

instance *fun* :: (*finite*, *finite-lattice-complete*) *finite-lattice-complete*
⟨*proof*⟩

93.7 Finite Distributive Lattices

A finite distributive lattice is a complete lattice whose *inf* and *sup* operators distribute over *Sup* and *Inf*.

class *finite-distrib-lattice-complete* =
distrib-lattice + *finite-lattice-complete*

lemma *finite-distrib-lattice-complete-sup-Inf*:
sup (*x*::*'a*::*finite-distrib-lattice-complete*) (*Inf A*) = (*INF y*:*A*. *sup x y*)
⟨*proof*⟩

lemma *finite-distrib-lattice-complete-inf-Sup*:
inf (*x*::*'a*::*finite-distrib-lattice-complete*) (*Sup A*) = (*SUP y*:*A*. *inf x y*)
⟨*proof*⟩

instance *finite-distrib-lattice-complete* ⊆ *complete-distrib-lattice*
⟨*proof*⟩

The product of two finite distributive lattices is already a finite distributive lattice.

instance *prod* ::
(*finite-distrib-lattice-complete*, *finite-distrib-lattice-complete*)
finite-distrib-lattice-complete
⟨*proof*⟩

Functions with a finite domain and with a finite distributive lattice as codomain already form a finite distributive lattice.

instance *fun* ::
(*finite*, *finite-distrib-lattice-complete*) *finite-distrib-lattice-complete*
⟨*proof*⟩

93.8 Linear Orders

A linear order is a distributive lattice. A type class is defined that extends class *linorder* with the operators *inf* and *sup*, along with assumptions that define these operators in terms of the ones of class *linorder*. The resulting class is a subclass of *distrib-lattice*.

class *linorder-lattice* = *linorder* + *inf* + *sup* +

assumes *inf-def*: $inf\ x\ y = (if\ x \leq y\ then\ x\ else\ y)$
assumes *sup-def*: $sup\ x\ y = (if\ x \geq y\ then\ x\ else\ y)$

The definitional assumptions on the operators *inf* and *sup* of class *linorder-lattice* ensure that they yield infimum and supremum and that they distribute over each other.

lemma *linorder-lattice-inf-le1*: $inf\ (x::'a::linorder\text{-}lattice)\ y \leq x$
⟨*proof*⟩

lemma *linorder-lattice-inf-le2*: $inf\ (x::'a::linorder\text{-}lattice)\ y \leq y$
⟨*proof*⟩

lemma *linorder-lattice-inf-greatest*:
$(x::'a::linorder\text{-}lattice) \leq y \Longrightarrow x \leq z \Longrightarrow x \leq inf\ y\ z$
⟨*proof*⟩

lemma *linorder-lattice-sup-ge1*: $sup\ (x::'a::linorder\text{-}lattice)\ y \geq x$
⟨*proof*⟩

lemma *linorder-lattice-sup-ge2*: $sup\ (x::'a::linorder\text{-}lattice)\ y \geq y$
⟨*proof*⟩

lemma *linorder-lattice-sup-least*:
$(x::'a::linorder\text{-}lattice) \geq y \Longrightarrow x \geq z \Longrightarrow x \geq sup\ y\ z$
⟨*proof*⟩

lemma *linorder-lattice-sup-inf-distrib1*:
$sup\ (x::'a::linorder\text{-}lattice)\ (inf\ y\ z) = inf\ (sup\ x\ y)\ (sup\ x\ z)$
⟨*proof*⟩

instance *linorder-lattice* ⊆ *distrib-lattice*
⟨*proof*⟩

93.9 Finite Linear Orders

A (non-empty) finite linear order is a complete linear order.

class *finite-linorder-complete* = *linorder-lattice* + *finite-lattice-complete*

instance *finite-linorder-complete* ⊆ *complete-linorder* ⟨*proof*⟩

A (non-empty) finite linear order is a complete lattice whose *inf* and *sup* operators distribute over *Sup* and *Inf*.

instance *finite-linorder-complete* ⊆ *finite-distrib-lattice-complete* ⟨*proof*⟩

end

94 Lexicographic order on lists

theory *List-lexord*
imports *Main*
begin

instantiation *list* :: (*ord*) *ord*
begin

definition
list-less-def: $xs < ys \longleftrightarrow (xs, ys) \in lexord\ \{(u, v).\ u < v\}$

definition
list-le-def: $(xs :: \text{-}\ list) \leq ys \longleftrightarrow xs < ys \vee xs = ys$

instance ⟨*proof*⟩

end

instance *list* :: (*order*) *order*
⟨*proof*⟩

instance *list* :: (*linorder*) *linorder*
⟨*proof*⟩

instantiation *list* :: (*linorder*) *distrib-lattice*
begin

definition $(inf :: 'a\ list \Rightarrow \text{-}) = min$

definition $(sup :: 'a\ list \Rightarrow \text{-}) = max$

instance
⟨*proof*⟩

end

lemma *not-less-Nil* [*simp*]: $\neg\ x < []$
⟨*proof*⟩

lemma *Nil-less-Cons* [*simp*]: $[] < a \# x$
⟨*proof*⟩

lemma *Cons-less-Cons* [*simp*]: $a \# x < b \# y \longleftrightarrow a < b \vee a = b \wedge x < y$
⟨*proof*⟩

lemma *le-Nil* [*simp*]: $x \leq [] \longleftrightarrow x = []$
⟨*proof*⟩

lemma *Nil-le-Cons* [*simp*]: $[] \leq x$
⟨*proof*⟩

lemma *Cons-le-Cons* [*simp*]: $a \# x \leq b \# y \longleftrightarrow a < b \vee a = b \wedge x \leq y$
⟨*proof*⟩

instantiation *list* :: (*order*) *order-bot*
begin

definition *bot* = []

instance
⟨*proof*⟩

end

lemma *less-list-code* [*code*]:
$xs < ([]::'a::\{equal, order\}\ list) \longleftrightarrow False$
$[] < (x::'a::\{equal, order\}) \# xs \longleftrightarrow True$
$(x::'a::\{equal, order\}) \# xs < y \# ys \longleftrightarrow x < y \vee x = y \wedge xs < ys$
⟨*proof*⟩

lemma *less-eq-list-code* [*code*]:
$x \# xs \leq ([]::'a::\{equal, order\}\ list) \longleftrightarrow False$
$[] \leq (xs::'a::\{equal, order\}\ list) \longleftrightarrow True$
$(x::'a::\{equal, order\}) \# xs \leq y \# ys \longleftrightarrow x < y \vee x = y \wedge xs \leq ys$
⟨*proof*⟩

end

95 Prefix order on lists as order class instance

theory *Prefix-Order*
imports *Sublist*
begin

instantiation *list* :: (*type*) *order*
begin

definition $xs \leq ys \equiv prefix\ xs\ ys$ **for** $xs\ ys :: 'a\ list$
definition $xs < ys \equiv xs \leq ys \wedge \neg (ys \leq xs)$ **for** $xs\ ys :: 'a\ list$

instance
⟨*proof*⟩

end

lemma *less-list-def'*: $xs < ys \longleftrightarrow strict\text{-}prefix\ xs\ ys$ **for** $xs\ ys :: 'a\ list$
⟨*proof*⟩

lemmas *prefixI* [*intro?*] = *prefixI* [*folded less-eq-list-def*]
lemmas *prefixE* [*elim?*] = *prefixE* [*folded less-eq-list-def*]
lemmas *strict-prefixI'* [*intro?*] = *strict-prefixI'* [*folded less-list-def'*]
lemmas *strict-prefixE'* [*elim?*] = *strict-prefixE'* [*folded less-list-def'*]
lemmas *strict-prefixI* [*intro?*] = *strict-prefixI* [*folded less-list-def'*]
lemmas *strict-prefixE* [*elim?*] = *strict-prefixE* [*folded less-list-def'*]
lemmas *Nil-prefix* [*iff*] = *Nil-prefix* [*folded less-eq-list-def*]
lemmas *prefix-Nil* [*simp*] = *prefix-Nil* [*folded less-eq-list-def*]
lemmas *prefix-snoc* [*simp*] = *prefix-snoc* [*folded less-eq-list-def*]
lemmas *Cons-prefix-Cons* [*simp*] = *Cons-prefix-Cons* [*folded less-eq-list-def*]
lemmas *same-prefix-prefix* [*simp*] = *same-prefix-prefix* [*folded less-eq-list-def*]
lemmas *same-prefix-nil* [*iff*] = *same-prefix-nil* [*folded less-eq-list-def*]
lemmas *prefix-prefix* [*simp*] = *prefix-prefix* [*folded less-eq-list-def*]
lemmas *prefix-Cons* = *prefix-Cons* [*folded less-eq-list-def*]
lemmas *prefix-length-le* = *prefix-length-le* [*folded less-eq-list-def*]
lemmas *strict-prefix-simps* [*simp*, *code*] = *strict-prefix-simps* [*folded less-list-def'*]
lemmas *not-prefix-induct* [*consumes 1*, *case-names Nil Neq Eq*] =
not-prefix-induct [*folded less-eq-list-def*]

end

96 Lexicographic order on product types

theory *Product-Lexorder*
imports *Main*
begin

instantiation *prod* :: (*ord*, *ord*) *ord*
begin

definition
$x \leq y \longleftrightarrow fst\ x < fst\ y \vee fst\ x \leq fst\ y \wedge snd\ x \leq snd\ y$

definition
$x < y \longleftrightarrow fst\ x < fst\ y \vee fst\ x \leq fst\ y \wedge snd\ x < snd\ y$

instance ⟨*proof*⟩

end

lemma *less-eq-prod-simp* [*simp*, *code*]:
$(x1, y1) \leq (x2, y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 \leq y2$
⟨*proof*⟩

lemma *less-prod-simp* [*simp*, *code*]:
$(x1, y1) < (x2, y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 < y2$
⟨*proof*⟩

A stronger version for partial orders.

lemma *less-prod-def'*:
fixes $x\ y :: 'a::order \times 'b::ord$
shows $x < y \longleftrightarrow fst\ x < fst\ y \vee fst\ x = fst\ y \wedge snd\ x < snd\ y$
⟨*proof*⟩

instance *prod* :: (*preorder*, *preorder*) *preorder*
⟨*proof*⟩

instance *prod* :: (*order*, *order*) *order*
⟨*proof*⟩

instance *prod* :: (*linorder*, *linorder*) *linorder*
⟨*proof*⟩

instantiation *prod* :: (*linorder*, *linorder*) *distrib-lattice*
begin

definition
$(inf :: 'a \times 'b \Rightarrow - \Rightarrow -) = min$

definition
$(sup :: 'a \times 'b \Rightarrow - \Rightarrow -) = max$

instance
⟨*proof*⟩

end

instantiation *prod* :: (*bot*, *bot*) *bot*
begin

definition
$bot = (bot, bot)$

instance ⟨*proof*⟩

end

instance *prod* :: (*order-bot*, *order-bot*) *order-bot*
⟨*proof*⟩

instantiation *prod* :: (*top*, *top*) *top*
begin

definition
$top = (top, top)$

instance ⟨*proof*⟩

end

instance *prod* :: (*order-top*, *order-top*) *order-top*
⟨*proof*⟩

instance *prod* :: (*wellorder*, *wellorder*) *wellorder*
⟨*proof*⟩

Legacy lemma bindings

lemmas *prod-le-def* = *less-eq-prod-def*
lemmas *prod-less-def* = *less-prod-def*
lemmas *prod-less-eq* = *less-prod-def'*

end

97 Subsequence Ordering

theory *Subseq-Order*
imports *Sublist*
begin

This theory defines subsequence ordering on lists. A list *ys* is a subsequence of a list *xs*, iff one obtains *ys* by erasing some elements from *xs*.

97.1 Definitions and basic lemmas

instantiation *list* :: (*type*) *ord*
begin

definition $xs \leq ys \longleftrightarrow$ *subseq xs ys* **for** *xs ys* :: *'a list*
definition $xs < ys \longleftrightarrow xs \leq ys \wedge \neg\ ys \leq xs$ **for** *xs ys* :: *'a list*

instance ⟨*proof*⟩

end

instance *list* :: (*type*) *order*
⟨*proof*⟩

lemmas *less-eq-list-induct* [*consumes 1*, *case-names empty drop take*] =
list-emb.induct [*of op* =, *folded less-eq-list-def*]
lemmas *less-eq-list-drop* = *list-emb.list-emb-Cons* [*of op* =, *folded less-eq-list-def*]
lemmas *le-list-Cons2-iff* [*simp*, *code*] = *subseq-Cons2-iff* [*folded less-eq-list-def*]
lemmas *le-list-map* = *subseq-map* [*folded less-eq-list-def*]
lemmas *le-list-filter* = *subseq-filter* [*folded less-eq-list-def*]
lemmas *le-list-length* = *list-emb-length* [*of op* =, *folded less-eq-list-def*]

lemma *less-list-length*: $xs < ys \Longrightarrow$ *length xs* < *length ys*

⟨*proof*⟩

lemma *less-list-empty* [*simp*]: [] < *xs* ⟷ *xs* ≠ []
⟨*proof*⟩

lemma *less-list-below-empty* [*simp*]: *xs* < [] ⟷ *False*
⟨*proof*⟩

lemma *less-list-drop*: *xs* < *ys* ⟹ *xs* < *x* # *ys*
⟨*proof*⟩

lemma *less-list-take-iff*: *x* # *xs* < *x* # *ys* ⟷ *xs* < *ys*
⟨*proof*⟩

lemma *less-list-drop-many*: *xs* < *ys* ⟹ *xs* < *zs* @ *ys*
⟨*proof*⟩

lemma *less-list-take-many-iff*: *zs* @ *xs* < *zs* @ *ys* ⟷ *xs* < *ys*
⟨*proof*⟩

lemma *less-list-rev-take*: *xs* @ *zs* < *ys* @ *zs* ⟷ *xs* < *ys*
⟨*proof*⟩

end

98 Implementation of mappings with Association Lists

theory *AList-Mapping*
imports *AList Mapping*
begin

lift-definition *Mapping* :: ('*a* × '*b*) *list* ⇒ ('*a*, '*b*) *mapping* **is** *map-of* ⟨*proof*⟩

code-datatype *Mapping*

lemma *lookup-Mapping* [*simp*, *code*]: *Mapping.lookup* (*Mapping xs*) = *map-of xs*
⟨*proof*⟩

lemma *keys-Mapping* [*simp*, *code*]: *Mapping.keys* (*Mapping xs*) = *set* (*map fst xs*)
⟨*proof*⟩

lemma *empty-Mapping* [*code*]: *Mapping.empty* = *Mapping* []
⟨*proof*⟩

lemma *is-empty-Mapping* [*code*]: *Mapping.is-empty* (*Mapping xs*) ⟷ *List.null xs*
⟨*proof*⟩

lemma *update-Mapping* [*code*]: *Mapping.update k v* (*Mapping xs*) = *Mapping* (*AList.update k v xs*)
⟨*proof*⟩

lemma *delete-Mapping* [*code*]: *Mapping.delete k* (*Mapping xs*) = *Mapping* (*AList.delete k xs*)
⟨*proof*⟩

lemma *ordered-keys-Mapping* [*code*]:
Mapping.ordered-keys (*Mapping xs*) = *sort* (*remdups* (*map fst xs*))
⟨*proof*⟩

lemma *size-Mapping* [*code*]: *Mapping.size* (*Mapping xs*) = *length* (*remdups* (*map fst xs*))
⟨*proof*⟩

lemma *tabulate-Mapping* [*code*]: *Mapping.tabulate ks f* = *Mapping* (*map* (λ*k*. (*k*, *f k*)) *ks*)
⟨*proof*⟩

lemma *bulkload-Mapping* [*code*]:
Mapping.bulkload vs = *Mapping* (*map* (λ*n*. (*n*, *vs* ! *n*)) [*0*..<*length vs*])
⟨*proof*⟩

lemma *equal-Mapping* [*code*]:
HOL.equal (*Mapping xs*) (*Mapping ys*) ⟷
(*let ks* = *map fst xs*; *ls* = *map fst ys*
in (∀ *l*∈*set ls*. *l* ∈ *set ks*) ∧ (∀ *k*∈*set ks*. *k* ∈ *set ls* ∧ *map-of xs k* = *map-of ys k*))
⟨*proof*⟩

lemma *map-values-Mapping* [*code*]:
Mapping.map-values f (*Mapping xs*) = *Mapping* (*map* (λ(*x*,*y*). (*x*, *f x y*)) *xs*)
for *f* :: *'c* ⇒ *'a* ⇒ *'b* **and** *xs* :: (*'c* × *'a*) *list*
⟨*proof*⟩

lemma *combine-with-key-code* [*code*]:
Mapping.combine-with-key f (*Mapping xs*) (*Mapping ys*) =
Mapping.tabulate (*remdups* (*map fst xs* @ *map fst ys*))
(λ*x*. *the* (*combine-options* (*f x*) (*map-of xs x*) (*map-of ys x*)))
⟨*proof*⟩

lemma *combine-code* [*code*]:
Mapping.combine f (*Mapping xs*) (*Mapping ys*) =
Mapping.tabulate (*remdups* (*map fst xs* @ *map fst ys*))
(λ*x*. *the* (*combine-options f* (*map-of xs x*) (*map-of ys x*)))
⟨*proof*⟩

lemma *map-of-filter-distinct*:
assumes *distinct* (*map fst xs*)
shows *map-of* (*filter P xs*) *x* =
(*case map-of xs x of*
None ⇒ *None*
| *Some y* ⇒ *if P* (*x*,*y*) *then Some y else None*)
⟨*proof*⟩

lemma *filter-Mapping* [*code*]:
Mapping.filter P (*Mapping xs*) = *Mapping* (*filter* (λ(*k*,*v*). *P k v*) (*AList.clearjunk xs*))
⟨*proof*⟩

lemma [*code nbe*]: *HOL.equal* (*x* :: ('*a*, '*b*) *mapping*) *x* ⟷ *True*
⟨*proof*⟩

end

99 Avoidance of pattern matching on natural numbers

theory *Code-Abstract-Nat*
imports *Main*
begin

When natural numbers are implemented in another than the conventional inductive *0*/*Suc* representation, it is necessary to avoid all pattern matching on natural numbers altogether. This is accomplished by this theory (up to a certain extent).

99.1 Case analysis

Case analysis on natural numbers is rephrased using a conditional expression:

lemma [*code*, *code-unfold*]:
case-nat = (λ*f g n*. *if n* = *0 then f else g* (*n* − *1*))
⟨*proof*⟩

99.2 Preprocessors

The term *Suc n* is no longer a valid pattern. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a code equation) must be eliminated. This can be accomplished – as far as possible – by applying the following transformation rule:

lemma *Suc-if-eq*:
assumes ⋀*n*. *f* (*Suc n*) ≡ *h n*

assumes $f\ 0 \equiv g$
shows $f\ n \equiv \mathit{if}\ n = 0\ \mathit{then}\ g\ \mathit{else}\ h\ (n - 1)$
⟨*proof*⟩

The rule above is built into a preprocessor that is plugged into the code generator.

⟨*ML*⟩

end

100 Implementation of natural numbers as binary numerals

theory *Code-Binary-Nat*
imports *Code-Abstract-Nat*
begin

When generating code for functions on natural numbers, the canonical representation using *0* and *Suc* is unsuitable for computations involving large numbers. This theory refines the representation of natural numbers for code generation to use binary numerals, which do not grow linear in size but logarithmic.

100.1 Representation

code-datatype *0*::*nat nat-of-num*

lemma [*code*]:
num-of-nat 0 = *Num.One*
num-of-nat (*nat-of-num k*) = *k*
⟨*proof*⟩

lemma [*code*]:
(*1*::*nat*) = *Numeral1*
⟨*proof*⟩

lemma [*code-abbrev*]: *Numeral1* = (*1*::*nat*)
⟨*proof*⟩

lemma [*code*]:
Suc n = *n* + *1*
⟨*proof*⟩

100.2 Basic arithmetic

context
begin

declare [[*code drop*: *plus* :: *nat* ⇒ -]]

lemma *plus-nat-code* [*code*]:
nat-of-num k + *nat-of-num l* = *nat-of-num* (*k* + *l*)
m + *0* = (*m*::*nat*)
0 + *n* = (*n*::*nat*)
⟨*proof*⟩

Bounded subtraction needs some auxiliary

qualified definition *dup* :: *nat* ⇒ *nat* **where**
dup n = *n* + *n*

lemma *dup-code* [*code*]:
dup 0 = *0*
dup (*nat-of-num k*) = *nat-of-num* (*Num.Bit0 k*)
⟨*proof*⟩ **definition** *sub* :: *num* ⇒ *num* ⇒ *nat option* **where**
sub k l = (*if k* ≥ *l then Some* (*numeral k* − *numeral l*) *else None*)

lemma *sub-code* [*code*]:
sub Num.One Num.One = *Some 0*
sub (*Num.Bit0 m*) *Num.One* = *Some* (*nat-of-num* (*Num.BitM m*))
sub (*Num.Bit1 m*) *Num.One* = *Some* (*nat-of-num* (*Num.Bit0 m*))
sub Num.One (*Num.Bit0 n*) = *None*
sub Num.One (*Num.Bit1 n*) = *None*
sub (*Num.Bit0 m*) (*Num.Bit0 n*) = *map-option dup* (*sub m n*)
sub (*Num.Bit1 m*) (*Num.Bit1 n*) = *map-option dup* (*sub m n*)
sub (*Num.Bit1 m*) (*Num.Bit0 n*) = *map-option* (λ*q*. *dup q* + *1*) (*sub m n*)
sub (*Num.Bit0 m*) (*Num.Bit1 n*) = (*case sub m n of None* ⇒ *None*
| *Some q* ⇒ *if q* = *0 then None else Some* (*dup q* − *1*))
⟨*proof*⟩

declare [[*code drop*: *minus* :: *nat* ⇒ -]]

lemma *minus-nat-code* [*code*]:
nat-of-num k − *nat-of-num l* = (*case sub k l of None* ⇒ *0* | *Some j* ⇒ *j*)
m − *0* = (*m*::*nat*)
0 − *n* = (*0*::*nat*)
⟨*proof*⟩

declare [[*code drop*: *times* :: *nat* ⇒ -]]

lemma *times-nat-code* [*code*]:
nat-of-num k ∗ *nat-of-num l* = *nat-of-num* (*k* ∗ *l*)
m ∗ *0* = (*0*::*nat*)
0 ∗ *n* = (*0*::*nat*)
⟨*proof*⟩

declare [[*code drop*: *HOL.equal* :: *nat* ⇒ -]]

lemma *equal-nat-code* [*code*]:
HOL.equal 0 (*0*::*nat*) ⟷ *True*
HOL.equal 0 (*nat-of-num l*) ⟷ *False*
HOL.equal (*nat-of-num k*) *0* ⟷ *False*
HOL.equal (*nat-of-num k*) (*nat-of-num l*) ⟷ *HOL.equal k l*
⟨*proof*⟩

lemma *equal-nat-refl* [*code nbe*]:
HOL.equal (*n*::*nat*) *n* ⟷ *True*
⟨*proof*⟩

declare [[*code drop*: *less-eq* :: *nat* ⇒ -]]

lemma *less-eq-nat-code* [*code*]:
0 ≤ (*n*::*nat*) ⟷ *True*
nat-of-num k ≤ *0* ⟷ *False*
nat-of-num k ≤ *nat-of-num l* ⟷ *k* ≤ *l*
⟨*proof*⟩

declare [[*code drop*: *less* :: *nat* ⇒ -]]

lemma *less-nat-code* [*code*]:
(*m*::*nat*) < *0* ⟷ *False*
0 < *nat-of-num l* ⟷ *True*
nat-of-num k < *nat-of-num l* ⟷ *k* < *l*
⟨*proof*⟩

declare [[*code drop*: *Divides.divmod-nat*]]

lemma *divmod-nat-code* [*code*]:
Divides.divmod-nat (*nat-of-num k*) (*nat-of-num l*) = *divmod k l*
Divides.divmod-nat m 0 = (*0*, *m*)
Divides.divmod-nat 0 n = (*0*, *0*)
⟨*proof*⟩

end

100.3 Conversions

declare [[*code drop*: *of-nat*]]

lemma *of-nat-code* [*code*]:
of-nat 0 = *0*
of-nat (*nat-of-num k*) = *numeral k*
⟨*proof*⟩

code-identifier
code-module *Code-Binary-Nat* ⇀

(*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

101 Code generation of pretty characters (and strings)

theory *Code-Char*
imports *Main Char-ord*
begin

code-printing
type-constructor *char* ⇀
(*SML*) *char*
and (*OCaml*) *char*
and (*Haskell*) *Prelude.Char*
and (*Scala*) *Char*

⟨*ML*⟩

code-printing
constant *integer-of-char* ⇀
(*SML*) !(*IntInf.fromInt o Char.ord*)
and (*OCaml*) *Big'-int.big'-int'-of'-int* (*Char.code* -)
and (*Haskell*) *Prelude.toInteger* (*Prelude.fromEnum* (- :: *Prelude.Char*))
and (*Scala*) *BigInt*(-.*toInt*)
| **constant** *char-of-integer* ⇀
(*SML*) !(*Char.chr o IntInf.toInt*)
and (*OCaml*) *Char.chr* (*Big'-int.int'-of'-big'-int* -)
and (*Haskell*) !(*let chr k* | (*0* <= *k* && *k* < *256*) = *Prelude.toEnum k* :: *Prelude.Char in chr* . *Prelude.fromInteger*)
and (*Scala*) !((*k*: *BigInt*) => *if* (*BigInt*(*0*) <= *k* && *k* < *BigInt*(*256*)) *k.charValue else sys.error*(*character value out of range*))
| **class-instance** *char* :: *equal* ⇀
(*Haskell*) –
| **constant** *HOL.equal* :: *char* ⇒ *char* ⇒ *bool* ⇀
(*SML*) !((- : *char*) = -)
and (*OCaml*) !((- : *char*) = -)
and (*Haskell*) **infix** *4* ==
and (*Scala*) **infixl** *5* ==
| **constant** *Code-Evaluation.term-of* :: *char* ⇒ *term* ⇀
(*Eval*) *HOLogic.mk'-char*/ (*IntInf.fromInt*/ (*Char.ord*/ -))

code-reserved *SML*
char

code-reserved *OCaml*
char

code-reserved *Scala*

char

code-reserved *SML String*

code-printing
constant *String.implode* ⇀
(*SML*) *String.implode*
and (*OCaml*) !(*let l = - in let res = String.create* (*List.length l*) *in let rec imp i = function* | [] -> *res* | *c* :: *l* -> *String.set res i c*; *imp* (*i* + *1*) *l in imp 0 l*)
and (*Haskell*) -
and (*Scala*) !(++/ -)
| **constant** *String.explode* ⇀
(*SML*) *String.explode*
and (*OCaml*) !(*let s = - in let rec exp i l = if i* < *0 then l else exp* (*i* − *1*) (*String.get s i* :: *l*) *in exp* (*String.length s* − *1*) [])
and (*Haskell*) -
and (*Scala*) !(-.*toList*)

code-printing
constant *Orderings.less-eq* :: *char* ⇒ *char* ⇒ *bool* ⇀
(*SML*) !((- : *char*) <= -)
and (*OCaml*) !((- : *char*) <= -)
and (*Haskell*) **infix** *4* <=
and (*Scala*) **infixl** *4* <=
and (*Eval*) **infixl** *6* <=
| **constant** *Orderings.less* :: *char* ⇒ *char* ⇒ *bool* ⇀
(*SML*) !((- : *char*) < -)
and (*OCaml*) !((- : *char*) < -)
and (*Haskell*) **infix** *4* <
and (*Scala*) **infixl** *4* <
and (*Eval*) **infixl** *6* <
| **constant** *Orderings.less-eq* :: *String.literal* ⇒ *String.literal* ⇒ *bool* ⇀
(*SML*) !((- : *string*) <= -)
and (*OCaml*) !((- : *string*) <= -)
— 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 (*Haskell*) **infix** *4* <=
and (*Scala*) **infixl** *4* <=
and (*Eval*) **infixl** *6* <=
| **constant** *Orderings.less* :: *String.literal* ⇒ *String.literal* ⇒ *bool* ⇀
(*SML*) !((- : *string*) < -)
and (*OCaml*) !((- : *string*) < -)
and (*Haskell*) **infix** *4* <
and (*Scala*) **infixl** *4* <
and (*Eval*) **infixl** *6* <

end

102 Code generation of prolog programs

theory *Code-Prolog*
imports *Main*
keywords *values-prolog* :: *diag*
begin

⟨*ML*⟩

103 Setup for Numerals

⟨*ML*⟩

end

104 Implementation of integer numbers by target-language integers

theory *Code-Target-Int*
imports *Main*
begin

code-datatype *int-of-integer*

declare [[*code drop*: *integer-of-int*]]

context
includes *integer.lifting*
begin

lemma [*code*]:
integer-of-int (*int-of-integer k*) = *k*
⟨*proof*⟩

lemma [*code*]:
Int.Pos = *int-of-integer* ∘ *integer-of-num*
⟨*proof*⟩

lemma [*code*]:
Int.Neg = *int-of-integer* ∘ *uminus* ∘ *integer-of-num*
⟨*proof*⟩

lemma [*code-abbrev*]:
int-of-integer (*numeral k*) = *Int.Pos k*
⟨*proof*⟩

lemma [*code-abbrev*]:
int-of-integer (− *numeral k*) = *Int.Neg k*

⟨*proof*⟩

context
begin

qualified definition *positive* :: *num* ⇒ *int*
where [*simp*]: *positive* = *numeral*

qualified definition *negative* :: *num* ⇒ *int*
where [*simp*]: *negative* = *uminus* ∘ *numeral*

lemma [*code-computation-unfold*]:
numeral = *positive*
Int.Pos = *positive*
Int.Neg = *negative*
⟨*proof*⟩

end

lemma [*code*, *symmetric*, *code-post*]:
0 = *int-of-integer 0*
⟨*proof*⟩

lemma [*code*, *symmetric*, *code-post*]:
1 = *int-of-integer 1*
⟨*proof*⟩

lemma [*code-post*]:
int-of-integer (− *1*) = − *1*
⟨*proof*⟩

lemma [*code*]:
k + *l* = *int-of-integer* (*of-int k* + *of-int l*)
⟨*proof*⟩

lemma [*code*]:
− *k* = *int-of-integer* (− *of-int k*)
⟨*proof*⟩

lemma [*code*]:
k − *l* = *int-of-integer* (*of-int k* − *of-int l*)
⟨*proof*⟩

lemma [*code*]:
Int.dup k = *int-of-integer* (*Code-Numeral.dup* (*of-int k*))
⟨*proof*⟩

declare [[*code drop*: *Int.sub*]]

lemma [*code*]:
k ∗ l = int-of-integer (of-int k ∗ of-int l)
⟨*proof*⟩

lemma [*code*]:
k div l = int-of-integer (of-int k div of-int l)
⟨*proof*⟩

lemma [*code*]:
k mod l = int-of-integer (of-int k mod of-int l)
⟨*proof*⟩

lemma [*code*]:
divmod m n = map-prod int-of-integer int-of-integer (divmod m n)
⟨*proof*⟩

lemma [*code*]:
HOL.equal k l = HOL.equal (of-int k :: integer) (of-int l)
⟨*proof*⟩

lemma [*code*]:
k ≤ l ⟷ (of-int k :: integer) ≤ of-int l
⟨*proof*⟩

lemma [*code*]:
k < l ⟷ (of-int k :: integer) < of-int l
⟨*proof*⟩

declare [[*code drop*: *gcd* :: *int* ⇒ - *lcm* :: *int* ⇒ -]]

lemma *gcd-int-of-integer* [*code*]:
gcd (int-of-integer x) (int-of-integer y) = int-of-integer (gcd x y)
⟨*proof*⟩

lemma *lcm-int-of-integer* [*code*]:
lcm (int-of-integer x) (int-of-integer y) = int-of-integer (lcm x y)
⟨*proof*⟩

end

lemma (**in** *ring-1*) *of-int-code-if*:
of-int k = (if k = 0 then 0
else if k < 0 then − of-int (− k)
else let
l = 2 ∗ of-int (k div 2);
j = k mod 2
in if j = 0 then l else l + 1)
⟨*proof*⟩

declare *of-int-code-if* [*code*]

lemma [*code*]:
nat = *nat-of-integer* ∘ *of-int*
including *integer.lifting* ⟨*proof*⟩

code-identifier
code-module *Code-Target-Int* ⇀
(*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

theory *Code-Real-Approx-By-Float*
imports *Complex-Main Code-Target-Int*
begin

WARNING! This theory implements mathematical reals by machine reals (floats). This is inconsistent. See the proof of False at the end of the theory, where an equality on mathematical reals is (incorrectly) disproved by mapping it to machine reals.

The **value** command cannot display real results yet.

The only legitimate use of this theory is as a tool for code generation purposes.

code-printing
type-constructor *real* ⇀
(*SML*) *real*
and (*OCaml*) *float*

code-printing
constant *Ratreal* ⇀
(*SML*) *error/ Bad constant*: *Ratreal*

code-printing
constant *0* :: *real* ⇀
(*SML*) *0.0*
and (*OCaml*) *0.0*

code-printing
constant *1* :: *real* ⇀
(*SML*) *1.0*
and (*OCaml*) *1.0*

code-printing
constant *HOL.equal* :: *real* ⇒ *real* ⇒ *bool* ⇀
(*SML*) *Real.==* ((-), (-))
and (*OCaml*) *Pervasives.*(=)

code-printing

constant *Orderings.less-eq* :: *real* ⇒ *real* ⇒ *bool* ⇀
(*SML*) *Real.<=* ((-), (-))
and (*OCaml*) *Pervasives.*(<=)

code-printing
constant *Orderings.less* :: *real* ⇒ *real* ⇒ *bool* ⇀
(*SML*) *Real.<* ((-), (-))
and (*OCaml*) *Pervasives.*(<)

code-printing
constant *op +* :: *real* ⇒ *real* ⇒ *real* ⇀
(*SML*) *Real.+* ((-), (-))
and (*OCaml*) *Pervasives.*(+.)

code-printing
constant *op ∗* :: *real* ⇒ *real* ⇒ *real* ⇀
(*SML*) *Real.∗* ((-), (-))
and (*OCaml*) *Pervasives.*(∗.)

code-printing
constant *op −* :: *real* ⇒ *real* ⇒ *real* ⇀
(*SML*) *Real.−* ((-), (-))
and (*OCaml*) *Pervasives.*(−.)

code-printing
constant *uminus* :: *real* ⇒ *real* ⇀
(*SML*) *Real.~*
and (*OCaml*) *Pervasives.*(~−.)

code-printing
constant *op /* :: *real* ⇒ *real* ⇒ *real* ⇀
(*SML*) *Real.'/* ((-), (-))
and (*OCaml*) *Pervasives.*('/.)

code-printing
constant *HOL.equal* :: *real* ⇒ *real* ⇒ *bool* ⇀
(*SML*) *Real.==* ((-:*real*), (-))

code-printing
constant *sqrt* :: *real* ⇒ *real* ⇀
(*SML*) *Math.sqrt*
and (*OCaml*) *Pervasives.sqrt*
declare *sqrt-def*[*code del*]

context
begin

qualified definition *real-exp* :: *real* ⇒ *real*
where *real-exp* = *exp*

lemma *exp-eq-real-exp* [*code-unfold*]: *exp* = *real-exp*
⟨*proof*⟩

end

code-printing
constant *Code-Real-Approx-By-Float.real-exp* ⇀
(*SML*) *Math.exp*
and (*OCaml*) *Pervasives.exp*
declare *Code-Real-Approx-By-Float.real-exp-def* [*code del*]
declare *exp-def* [*code del*]

code-printing
constant *ln* ⇀
(*SML*) *Math.ln*
and (*OCaml*) *Pervasives.ln*
declare *ln-real-def* [*code del*]

code-printing
constant *cos* ⇀
(*SML*) *Math.cos*
and (*OCaml*) *Pervasives.cos*
declare *cos-def* [*code del*]

code-printing
constant *sin* ⇀
(*SML*) *Math.sin*
and (*OCaml*) *Pervasives.sin*
declare *sin-def* [*code del*]

code-printing
constant *pi* ⇀
(*SML*) *Math.pi*
and (*OCaml*) *Pervasives.pi*
declare *pi-def* [*code del*]

code-printing
constant *arctan* ⇀
(*SML*) *Math.atan*
and (*OCaml*) *Pervasives.atan*
declare *arctan-def* [*code del*]

code-printing
constant *arccos* ⇀
(*SML*) *Math.scos*
and (*OCaml*) *Pervasives.acos*
declare *arccos-def* [*code del*]

code-printing
  constant *arcsin* ⇀
    (*SML*) *Math.asin*
    and (*OCaml*) *Pervasives.asin*
declare *arcsin-def*[*code del*]

definition *real-of-integer* :: *integer* ⇒ *real*
  where *real-of-integer* = *of-int* ∘ *int-of-integer*

code-printing
  constant *real-of-integer* ⇀
    (*SML*) *Real.fromInt*
    and (*OCaml*) *Pervasives.float* (*Big'-int.int'-of'-big'-int* (-))

context
begin

qualified definition *real-of-int* :: *int* ⇒ *real*
  where [*code-abbrev*]: *real-of-int* = *of-int*

lemma [*code*]: *real-of-int* = *real-of-integer* ∘ *integer-of-int*
  ⟨*proof*⟩

lemma [*code-unfold del*]: *0* ≡ (*of-rat 0* :: *real*)
  ⟨*proof*⟩

lemma [*code-unfold del*]: *1* ≡ (*of-rat 1* :: *real*)
  ⟨*proof*⟩

lemma [*code-unfold del*]: *numeral k* ≡ (*of-rat* (*numeral k*) :: *real*)
  ⟨*proof*⟩

lemma [*code-unfold del*]: − *numeral k* ≡ (*of-rat* (− *numeral k*) :: *real*)
  ⟨*proof*⟩

end

code-printing
  constant *Ratreal* ⇀ (*SML*)

definition *Realfract* :: *int* ⇒ *int* ⇒ *real*
  where *Realfract p q* = *of-int p* / *of-int q*

code-datatype *Realfract*

code-printing
  constant *Realfract* ⇀ (*SML*) *Real.fromInt* -/ *'// Real.fromInt* -

lemma [*code*]: *Ratreal r* = *case-prod Realfract* (*quotient-of r*)

⟨*proof*⟩

declare [[*code drop*: *HOL.equal* :: *real* ⇒ *real* ⇒ *bool*
plus :: *real* ⇒ *real* ⇒ *real*
uminus :: *real* ⇒ *real*
minus :: *real* ⇒ *real* ⇒ *real*
times :: *real* ⇒ *real* ⇒ *real*
divide :: *real* ⇒ *real* ⇒ *real*]]

lemma [*code*]: *inverse r* = *1* / *r* **for** *r* :: *real*
⟨*proof*⟩

notepad
begin
⟨*proof*⟩
end

end

105 Implementation of natural numbers by target-language integers

theory *Code-Target-Nat*
imports *Code-Abstract-Nat*
begin

105.1 Implementation for *nat*

context
includes *natural.lifting integer.lifting*
begin

lift-definition *Nat* :: *integer* ⇒ *nat*
is *nat*
⟨*proof*⟩

lemma [*code-post*]:
Nat 0 = *0*
Nat 1 = *1*
Nat (*numeral k*) = *numeral k*
⟨*proof*⟩

lemma [*code-abbrev*]:
integer-of-nat = *of-nat*
⟨*proof*⟩

lemma [*code-unfold*]:
Int.nat (*int-of-integer k*) = *nat-of-integer k*

⟨*proof*⟩

lemma [*code abstype*]:
Code-Target-Nat.Nat (*integer-of-nat n*) = *n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*nat-of-integer k*) = *max 0 k*
⟨*proof*⟩

lemma [*code-abbrev*]:
nat-of-integer (*numeral k*) = *nat-of-num k*
⟨*proof*⟩

context
begin

qualified definition *natural* :: *num* ⇒ *nat*
where [*simp*]: *natural* = *nat-of-num*

lemma [*code-computation-unfold*]:
numeral = *natural*
nat-of-num = *natural*
⟨*proof*⟩

end

lemma [*code abstract*]:
integer-of-nat (*nat-of-num n*) = *integer-of-num n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat 0 = *0*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat 1 = *1*
⟨*proof*⟩

lemma [*code*]:
Suc n = *n* + *1*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m* + *n*) = *of-nat m* + *of-nat n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m* − *n*) = *max 0* (*of-nat m* − *of-nat n*)

⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m* ∗ *n*) = *of-nat m* ∗ *of-nat n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m div n*) = *of-nat m div of-nat n*
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*m mod n*) = *of-nat m mod of-nat n*
⟨*proof*⟩

lemma [*code*]:
Divides.divmod-nat m n = (*m div n*, *m mod n*)
⟨*proof*⟩

lemma [*code*]:
divmod m n = *map-prod nat-of-integer nat-of-integer* (*divmod m n*)
⟨*proof*⟩

lemma [*code*]:
HOL.equal m n = *HOL.equal* (*of-nat m* :: *integer*) (*of-nat n*)
⟨*proof*⟩

lemma [*code*]:
m ≤ *n* ⟷ (*of-nat m* :: *integer*) ≤ *of-nat n*
⟨*proof*⟩

lemma [*code*]:
m < *n* ⟷ (*of-nat m* :: *integer*) < *of-nat n*
⟨*proof*⟩

lemma *num-of-nat-code* [*code*]:
num-of-nat = *num-of-integer* ∘ *of-nat*
⟨*proof*⟩

end

lemma (**in** *semiring-1*) *of-nat-code-if*:
of-nat n = (*if n* = *0 then 0*
else let
(*m*, *q*) = *Divides.divmod-nat n 2*;
m′ = *2* ∗ *of-nat m*
in if q = *0 then m*′ *else m*′ + *1*)
⟨*proof*⟩

declare *of-nat-code-if* [*code*]

definition *int-of-nat* :: *nat* ⇒ *int* **where**
[*code-abbrev*]: *int-of-nat* = *of-nat*

lemma [*code*]:
int-of-nat n = *int-of-integer* (*of-nat n*)
⟨*proof*⟩

lemma [*code abstract*]:
integer-of-nat (*nat k*) = *max 0* (*integer-of-int k*)
including *integer.lifting* ⟨*proof*⟩

lemma *term-of-nat-code* [*code*]:
— Use *nat-of-integer* in term reconstruction instead of *Code-Target-Nat.Nat* such that reconstructed terms can be fed back to the code generator
term-of-class.term-of n =
Code-Evaluation.App
(*Code-Evaluation.Const* (*STR ''Code-Numeral.nat-of-integer''*)
(*typerep.Typerep* (*STR ''fun''*)
[*typerep.Typerep* (*STR ''Code-Numeral.integer''*) [],
typerep.Typerep (*STR ''Nat.nat''*) []]))
(*term-of-class.term-of* (*integer-of-nat n*))
⟨*proof*⟩

lemma *nat-of-integer-code-post* [*code-post*]:
nat-of-integer 0 = *0*
nat-of-integer 1 = *1*
nat-of-integer (*numeral k*) = *numeral k*
including *integer.lifting* ⟨*proof*⟩

code-identifier
code-module *Code-Target-Nat* ⇀
(*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

106 Implementation of natural and integer numbers by target-language integers

theory *Code-Target-Numeral*
imports *Code-Target-Int Code-Target-Nat*
begin

end

107 Abstract type of association lists with unique keys

theory *DAList*
imports *AList*
begin

This was based on some existing fragments in the AFP-Collection framework.

107.1 Preliminaries

lemma *distinct-map-fst-filter*:
distinct (*map fst xs*) ⟹ *distinct* (*map fst* (*List.filter P xs*))
⟨*proof*⟩

107.2 Type ('*key*, '*value*) *alist*

typedef ('*key*, '*value*) *alist* = {*xs* :: ('*key* × '*value*) *list*. (*distinct* ∘ *map fst*) *xs*}
morphisms *impl-of Alist*
⟨*proof*⟩

setup-lifting *type-definition-alist*

lemma *alist-ext*: *impl-of xs* = *impl-of ys* ⟹ *xs* = *ys*
⟨*proof*⟩

lemma *alist-eq-iff*: *xs* = *ys* ⟷ *impl-of xs* = *impl-of ys*
⟨*proof*⟩

lemma *impl-of-distinct* [*simp*, *intro*]: *distinct* (*map fst* (*impl-of xs*))
⟨*proof*⟩

lemma *Alist-impl-of* [*code abstype*]: *Alist* (*impl-of xs*) = *xs*
⟨*proof*⟩

107.3 Primitive operations

lift-definition *lookup* :: ('*key*, '*value*) *alist* ⇒ '*key* ⇒ '*value option* **is** *map-of*
⟨*proof*⟩

lift-definition *empty* :: ('*key*, '*value*) *alist* **is** []
⟨*proof*⟩

lift-definition *update* :: '*key* ⇒ '*value* ⇒ ('*key*, '*value*) *alist* ⇒ ('*key*, '*value*) *alist*
is *AList.update*
⟨*proof*⟩

lift-definition *delete* :: *'key* ⇒ (*'key*, *'value*) *alist* ⇒ (*'key*, *'value*) *alist*
is *AList.delete*
⟨*proof*⟩

lift-definition *map-entry* ::
'key ⇒ (*'value* ⇒ *'value*) ⇒ (*'key*, *'value*) *alist* ⇒ (*'key*, *'value*) *alist*
is *AList.map-entry*
⟨*proof*⟩

lift-definition *filter* :: (*'key* × *'value* ⇒ *bool*) ⇒ (*'key*, *'value*) *alist* ⇒ (*'key*, *'value*) *alist*
is *List.filter*
⟨*proof*⟩

lift-definition *map-default* ::
'key ⇒ *'value* ⇒ (*'value* ⇒ *'value*) ⇒ (*'key*, *'value*) *alist* ⇒ (*'key*, *'value*) *alist*
is *AList.map-default*
⟨*proof*⟩

107.4 Abstract operation properties

lemma *lookup-empty* [*simp*]: *lookup empty k* = *None*
⟨*proof*⟩

lemma *lookup-update*:
lookup (*update k1 v xs*) *k2* = (*if k1* = *k2 then Some v else lookup xs k2*)
⟨*proof*⟩

lemma *lookup-update-eq* [*simp*]:
k1 = *k2* ⟹ *lookup* (*update k1 v xs*) *k2* = *Some v*
⟨*proof*⟩

lemma *lookup-update-neq* [*simp*]:
k1 ≠ *k2* ⟹ *lookup* (*update k1 v xs*) *k2* = *lookup xs k2*
⟨*proof*⟩

lemma *update-update-eq* [*simp*]:
k1 = *k2* ⟹ *update k2 v2* (*update k1 v1 xs*) = *update k2 v2 xs*
⟨*proof*⟩

lemma *lookup-delete* [*simp*]: *lookup* (*delete k al*) = (*lookup al*)(*k* := *None*)
⟨*proof*⟩

107.5 Further operations

107.5.1 Equality

instantiation *alist* :: (*equal*, *equal*) *equal*
begin

definition *HOL.equal* (*xs* :: ('*a*, '*b*) *alist*) *ys* == *impl-of xs* = *impl-of ys*

instance
⟨*proof*⟩

end

107.5.2 Size

instantiation *alist* :: (*type*, *type*) *size*
begin

definition *size* (*al* :: ('*a*, '*b*) *alist*) = *length* (*impl-of al*)

instance ⟨*proof*⟩

end

107.6 Quickcheck generators

notation *fcomp* (**infixl** ∘> *60*)
notation *scomp* (**infixl** ∘→ *60*)

definition (**in** *term-syntax*)
valterm-empty :: ('*key* :: *typerep*, '*value* :: *typerep*) *alist* × (*unit* ⇒ *Code-Evaluation.term*)
where *valterm-empty* = *Code-Evaluation.valtermify empty*

definition (**in** *term-syntax*)
valterm-update :: '*key* :: *typerep* × (*unit* ⇒ *Code-Evaluation.term*) ⇒
'*value* :: *typerep* × (*unit* ⇒ *Code-Evaluation.term*) ⇒
('*key*, '*value*) *alist* × (*unit* ⇒ *Code-Evaluation.term*) ⇒
('*key*, '*value*) *alist* × (*unit* ⇒ *Code-Evaluation.term*) **where**
[*code-unfold*]: *valterm-update k v a* = *Code-Evaluation.valtermify update* {·} *k* {·}
v {·}*a*

fun (**in** *term-syntax*) *random-aux-alist*
where
random-aux-alist i j =
(*if i* = *0 then Pair valterm-empty*
else Quickcheck-Random.collapse
(*Random.select-weight*
[(*i*, *Quickcheck-Random.random j* ∘→ (λ*k*. *Quickcheck-Random.random j*
∘→
(λ*v*. *random-aux-alist* (*i* − *1*) *j* ∘→ (λ*a*. *Pair* (*valterm-update k v a*))))),
(*1*, *Pair valterm-empty*)]))

instantiation *alist* :: (*random*, *random*) *random*
begin

definition *random-alist*

```
where
  random-alist i = random-aux-alist i i

instance ⟨proof⟩

end

no-notation fcomp (infixl ∘> 60)
no-notation scomp (infixl ∘→ 60)

instantiation alist :: (exhaustive, exhaustive) exhaustive
begin

fun exhaustive-alist ::
  (('a, 'b) alist ⇒ (bool × term list) option) ⇒ natural ⇒ (bool × term list) option
where
  exhaustive-alist f i =
    (if i = 0 then None
     else
      case f empty of
        Some ts ⇒ Some ts
      | None ⇒
          exhaustive-alist
            (λa. Quickcheck-Exhaustive.exhaustive
              (λk. Quickcheck-Exhaustive.exhaustive (λv. f (update k v a)) (i − 1))
(i − 1))
            (i − 1))

instance ⟨proof⟩

end

instantiation alist :: (full-exhaustive, full-exhaustive) full-exhaustive
begin

fun full-exhaustive-alist ::
  (('a, 'b) alist × (unit ⇒ term) ⇒ (bool × term list) option) ⇒ natural ⇒
    (bool × term list) option
where
  full-exhaustive-alist f i =
    (if i = 0 then None
     else
      case f valterm-empty of
        Some ts ⇒ Some ts
      | None ⇒
          full-exhaustive-alist
            (λa.
              Quickcheck-Exhaustive.full-exhaustive
               (λk. Quickcheck-Exhaustive.full-exhaustive (λv. f (valterm-update k v
```

a)) (*i* − *1*))
(*i* − *1*))
(*i* − *1*))

instance ⟨*proof*⟩

end

108 alist is a BNF

lift-bnf (*dead* *'k*, *set*: *'v*) *alist* [*wits*: [] :: (*'k* × *'v*) *list*] **for** *map*: *map rel*: *rel*
⟨*proof*⟩

hide-const *valterm-empty valterm-update random-aux-alist*

hide-fact (**open**) *lookup-def empty-def update-def delete-def map-entry-def filter-def map-default-def*
hide-const (**open**) *impl-of lookup empty update delete map-entry filter map-default map set rel*

end

109 Multisets partially implemented by association lists

theory *DAList-Multiset*
imports *Multiset DAList*
begin

Delete prexisting code equations

declare [[*code drop*: {#} *Multiset.is-empty add-mset*
plus :: *'a multiset* ⇒ *- minus* :: *'a multiset* ⇒ *-*
inf-subset-mset sup-subset-mset image-mset filter-mset count
size :: *- multiset* ⇒ *nat sum-mset prod-mset*
set-mset sorted-list-of-multiset subset-mset subseteq-mset
equal-multiset-inst.equal-multiset]]

Raw operations on lists

definition *join-raw* ::
(*'key* ⇒ *'val* × *'val* ⇒ *'val*) ⇒
(*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where *join-raw f xs ys* = *foldr* (λ(*k*, *v*). *map-default k v* (λ*v'*. *f k* (*v'*, *v*))) *ys xs*

lemma *join-raw-Nil* [*simp*]: *join-raw f xs* [] = *xs*
⟨*proof*⟩

lemma *join-raw-Cons* [*simp*]:
join-raw f xs ((*k*, *v*) # *ys*) = *map-default k v* (λ*v'*. *f k* (*v'*, *v*)) (*join-raw f xs ys*)

⟨*proof*⟩

lemma *map-of-join-raw*:
assumes *distinct* (*map fst ys*)
shows *map-of* (*join-raw f xs ys*) *x* =
(*case map-of xs x of*
None ⇒ *map-of ys x*
| *Some v* ⇒ (*case map-of ys x of None* ⇒ *Some v* | *Some v′* ⇒ *Some* (*f x* (*v*, *v′*))))
⟨*proof*⟩

lemma *distinct-join-raw*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*join-raw f xs ys*))
⟨*proof*⟩

definition *subtract-entries-raw xs ys* = *foldr* (λ(*k*, *v*). *AList.map-entry k* (λ*v′*. *v′* − *v*)) *ys xs*

lemma *map-of-subtract-entries-raw*:
assumes *distinct* (*map fst ys*)
shows *map-of* (*subtract-entries-raw xs ys*) *x* =
(*case map-of xs x of*
None ⇒ *None*
| *Some v* ⇒ (*case map-of ys x of None* ⇒ *Some v* | *Some v′* ⇒ *Some* (*v* − *v′*)))
⟨*proof*⟩

lemma *distinct-subtract-entries-raw*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*subtract-entries-raw xs ys*))
⟨*proof*⟩

Operations on alists with distinct keys

lift-definition *join* :: ('*a* ⇒ '*b* × '*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *alist* ⇒ ('*a*, '*b*) *alist* ⇒ ('*a*, '*b*) *alist*
is *join-raw*
⟨*proof*⟩

lift-definition *subtract-entries* :: ('*a*, ('*b* :: *minus*)) *alist* ⇒ ('*a*, '*b*) *alist* ⇒ ('*a*, '*b*) *alist*
is *subtract-entries-raw*
⟨*proof*⟩

Implementing multisets by means of association lists

definition *count-of* :: ('*a* × *nat*) *list* ⇒ '*a* ⇒ *nat*
where *count-of xs x* = (*case map-of xs x of None* ⇒ *0* | *Some n* ⇒ *n*)

lemma *count-of-multiset*: *count-of xs* ∈ *multiset*
⟨*proof*⟩

lemma *count-simps* [*simp*]:
count-of [] = (λ-. *0*)
count-of ((*x*, *n*) # *xs*) = (λ*y*. *if x = y then n else count-of xs y*)
⟨*proof*⟩

lemma *count-of-empty*: *x* ∉ *fst* ' *set xs* ⟹ *count-of xs x = 0*
⟨*proof*⟩

lemma *count-of-filter*: *count-of* (*List.filter* (*P* ∘ *fst*) *xs*) *x* = (*if P x then count-of xs x else 0*)
⟨*proof*⟩

lemma *count-of-map-default* [*simp*]:
count-of (*map-default x b* (λ*x*. *x* + *b*) *xs*) *y* =
(*if x = y then count-of xs x* + *b else count-of xs y*)
⟨*proof*⟩

lemma *count-of-join-raw*:
distinct (*map fst ys*) ⟹
count-of xs x + *count-of ys x* = *count-of* (*join-raw* (λ*x* (*x*, *y*). *x* + *y*) *xs ys*) *x*
⟨*proof*⟩

lemma *count-of-subtract-entries-raw*:
distinct (*map fst ys*) ⟹
count-of xs x − *count-of ys x* = *count-of* (*subtract-entries-raw xs ys*) *x*
⟨*proof*⟩

Code equations for multiset operations

definition *Bag* :: ('*a*, *nat*) *alist* ⇒ '*a multiset*
where *Bag xs* = *Abs-multiset* (*count-of* (*DAList.impl-of xs*))

code-datatype *Bag*

lemma *count-Bag* [*simp*, *code*]: *count* (*Bag xs*) = *count-of* (*DAList.impl-of xs*)
⟨*proof*⟩

lemma *Mempty-Bag* [*code*]: {#} = *Bag* (*DAList.empty*)
⟨*proof*⟩

lift-definition *is-empty-Bag-impl* :: ('*a*, *nat*) *alist* ⇒ *bool* **is**
λ*xs*. *list-all* (λ*x*. *snd x = 0*) *xs* ⟨*proof*⟩

lemma *is-empty-Bag* [*code*]: *Multiset.is-empty* (*Bag xs*) ⟷ *is-empty-Bag-impl xs*
⟨*proof*⟩

lemma *union-Bag* [*code*]: *Bag xs* + *Bag ys* = *Bag* (*join* (λ*x* (*n1*, *n2*). *n1* + *n2*) *xs ys*)
⟨*proof*⟩

lemma *add-mset-Bag* [*code*]: *add-mset x* (*Bag xs*) =
Bag (*join* (λ*x* (*n1*, *n2*). *n1* + *n2*) (*DAList.update x 1 DAList.empty*) *xs*)
⟨*proof*⟩

lemma *minus-Bag* [*code*]: *Bag xs* − *Bag ys* = *Bag* (*subtract-entries xs ys*)
⟨*proof*⟩

lemma *filter-Bag* [*code*]: *filter-mset P* (*Bag xs*) = *Bag* (*DAList.filter* (*P* ∘ *fst*) *xs*)
⟨*proof*⟩

lemma *mset-eq* [*code*]: *HOL.equal* (*m1*::'*a*::*equal multiset*) *m2* ⟷ *m1* ⊆# *m2* ∧ *m2* ⊆# *m1*
⟨*proof*⟩

By default the code for < is (*xs* < *ys*) = (*xs* ≤ *ys* ∧ *xs* ≠ *ys*). With equality implemented by ≤, this leads to three calls of ≤. Here is a more efficient version:

lemma *mset-less*[*code*]: *xs* ⊂# (*ys* :: '*a multiset*) ⟷ *xs* ⊆# *ys* ∧ ¬ *ys* ⊆# *xs*
⟨*proof*⟩

lemma *mset-less-eq-Bag0*:
Bag xs ⊆# *A* ⟷ (∀(*x*, *n*) ∈ *set* (*DAList.impl-of xs*). *count-of* (*DAList.impl-of xs*) *x* ≤ *count A x*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *mset-less-eq-Bag* [*code*]:
Bag xs ⊆# (*A* :: '*a multiset*) ⟷ (∀(*x*, *n*) ∈ *set* (*DAList.impl-of xs*). *n* ≤ *count A x*)
⟨*proof*⟩

declare *multiset-inter-def* [*code*]
declare *sup-subset-mset-def* [*code*]
declare *mset.simps* [*code*]

fun *fold-impl* :: ('*a* ⇒ *nat* ⇒ '*b* ⇒ '*b*) ⇒ '*b* ⇒ ('*a* × *nat*) *list* ⇒ '*b*
where
fold-impl fn e ((*a*,*n*) # *ms*) = (*fold-impl fn* ((*fn a n*) *e*) *ms*)
| *fold-impl fn e* [] = *e*

context
begin

qualified definition *fold* :: ('*a* ⇒ *nat* ⇒ '*b* ⇒ '*b*) ⇒ '*b* ⇒ ('*a*, *nat*) *alist* ⇒ '*b*
where *fold f e al* = *fold-impl f e* (*DAList.impl-of al*)

end

context *comp-fun-commute*
begin

lemma *DAList-Multiset-fold*:
assumes *fn*: $\bigwedge a\ n\ x.\ fn\ a\ n\ x = (f\ a\ \hat{}\hat{}\ n)\ x$
shows *fold-mset f e* (*Bag al*) = *DAList-Multiset.fold fn e al*
⟨*proof*⟩

end

context
begin

private lift-definition *single-alist-entry* :: $'a \Rightarrow 'b \Rightarrow ('a, 'b)$ *alist* **is** $\lambda a\ b.\ [(a, b)]$
⟨*proof*⟩

lemma *image-mset-Bag* [*code*]:
image-mset f (*Bag ms*) =
DAList-Multiset.fold ($\lambda a\ n\ m.$ *Bag* (*single-alist-entry* (*f a*) *n*) + *m*) {#} *ms*
⟨*proof*⟩

end

lemma *sum-mset-Bag*[*code*]: *sum-mset* (*Bag ms*) = *DAList-Multiset.fold* ($\lambda a\ n.$ ((*op* + *a*) $\hat{}\hat{}$ *n*)) *0 ms*
⟨*proof*⟩

lemma *prod-mset-Bag*[*code*]: *prod-mset* (*Bag ms*) = *DAList-Multiset.fold* ($\lambda a\ n.$ ((*op* ∗ *a*) $\hat{}\hat{}$ *n*)) *1 ms*
⟨*proof*⟩

lemma *size-fold*: *size A* = *fold-mset* (λ-. *Suc*) *0 A* (**is** - = *fold-mset ?f* - -)
⟨*proof*⟩

lemma *size-Bag*[*code*]: *size* (*Bag ms*) = *DAList-Multiset.fold* ($\lambda a\ n.$ *op* + *n*) *0 ms*
⟨*proof*⟩

lemma *set-mset-fold*: *set-mset A* = *fold-mset insert* {} *A* (**is** - = *fold-mset ?f* - -)
⟨*proof*⟩

lemma *set-mset-Bag*[*code*]:
set-mset (*Bag ms*) = *DAList-Multiset.fold* ($\lambda a\ n.$ (*if n* = *0 then* ($\lambda m.$ *m*) *else*

insert a)) {} *ms*
⟨*proof*⟩

instantiation *multiset* :: (*exhaustive*) *exhaustive*
begin

definition *exhaustive-multiset* ::
('*a multiset* ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where *exhaustive-multiset f i* = *Quickcheck-Exhaustive.exhaustive* (λ*xs*. *f* (*Bag xs*)) *i*

instance ⟨*proof*⟩

end

end

110 Implementation of Red-Black Trees

theory *RBT-Impl*
imports *Main*
begin

For applications, you should use theory *RBT* which defines an abstract type of red-black tree obeying the invariant.

110.1 Datatype of RB trees

datatype *color* = *R* | *B*
datatype ('*a*, '*b*) *rbt* = *Empty* | *Branch color* ('*a*, '*b*) *rbt* '*a* '*b* ('*a*, '*b*) *rbt*

lemma *rbt-cases*:
obtains (*Empty*) *t* = *Empty*
| (*Red*) *l k v r* **where** *t* = *Branch R l k v r*
| (*Black*) *l k v r* **where** *t* = *Branch B l k v r*
⟨*proof*⟩

110.2 Tree properties

110.2.1 Content of a tree

primrec *entries* :: ('*a*, '*b*) *rbt* ⇒ ('*a* × '*b*) *list*
where
entries Empty = []
| *entries* (*Branch - l k v r*) = *entries l* @ (*k*,*v*) # *entries r*

abbreviation (*input*) *entry-in-tree* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *rbt* ⇒ *bool*
where

entry-in-tree k v t ≡ (*k*, *v*) ∈ *set* (*entries t*)

definition *keys* :: ('*a*, '*b*) *rbt* ⇒ '*a list* **where**
keys t = *map fst* (*entries t*)

lemma *keys-simps* [*simp*, *code*]:
keys Empty = []
keys (*Branch c l k v r*) = *keys l* @ *k* # *keys r*
⟨*proof*⟩

lemma *entry-in-tree-keys*:
assumes (*k*, *v*) ∈ *set* (*entries t*)
shows *k* ∈ *set* (*keys t*)
⟨*proof*⟩

lemma *keys-entries*:
k ∈ *set* (*keys t*) ⟷ (∃ *v*. (*k*, *v*) ∈ *set* (*entries t*))
⟨*proof*⟩

lemma *non-empty-rbt-keys*:
t ≠ *rbt.Empty* ⟹ *keys t* ≠ []
⟨*proof*⟩

110.2.2 Search tree properties

context *ord* **begin**

definition *rbt-less* :: '*a* ⇒ ('*a*, '*b*) *rbt* ⇒ *bool*
where
rbt-less-prop: *rbt-less k t* ⟷ (∀ *x*∈*set* (*keys t*). *x* < *k*)

abbreviation *rbt-less-symbol* (**infix** |≪ *50*)
where *t* |≪ *x* ≡ *rbt-less x t*

definition *rbt-greater* :: '*a* ⇒ ('*a*, '*b*) *rbt* ⇒ *bool* (**infix** ≪| *50*)
where
rbt-greater-prop: *rbt-greater k t* = (∀ *x*∈*set* (*keys t*). *k* < *x*)

lemma *rbt-less-simps* [*simp*]:
Empty |≪ *k* = *True*
Branch c lt kt v rt |≪ *k* ⟷ *kt* < *k* ∧ *lt* |≪ *k* ∧ *rt* |≪ *k*
⟨*proof*⟩

lemma *rbt-greater-simps* [*simp*]:
k ≪| *Empty* = *True*
k ≪| (*Branch c lt kt v rt*) ⟷ *k* < *kt* ∧ *k* ≪| *lt* ∧ *k* ≪| *rt*
⟨*proof*⟩

lemmas *rbt-ord-props* = *rbt-less-prop rbt-greater-prop*

lemmas *rbt-greater-nit = rbt-greater-prop entry-in-tree-keys*
lemmas *rbt-less-nit = rbt-less-prop entry-in-tree-keys*

lemma (**in** *order*)
  shows *rbt-less-eq-trans*: $l \mid\ll u \Longrightarrow u \leq v \Longrightarrow l \mid\ll v$
  and *rbt-less-trans*: $t \mid\ll x \Longrightarrow x < y \Longrightarrow t \mid\ll y$
  and *rbt-greater-eq-trans*: $u \leq v \Longrightarrow v \ll\mid r \Longrightarrow u \ll\mid r$
  and *rbt-greater-trans*: $x < y \Longrightarrow y \ll\mid t \Longrightarrow x \ll\mid t$
  ⟨*proof*⟩

primrec *rbt-sorted* :: $('a, 'b)$ *rbt* ⇒ *bool*
where
  rbt-sorted Empty = True
| *rbt-sorted* (*Branch c l k v r*) = (*l* $\mid\ll$ *k* ∧ *k* $\ll\mid$ *r* ∧ *rbt-sorted l* ∧ *rbt-sorted r*)

end

context *linorder* **begin**

lemma *rbt-sorted-entries*:
  rbt-sorted t ⟹ *List.sorted* (*map fst* (*entries t*))
⟨*proof*⟩

lemma *distinct-entries*:
  rbt-sorted t ⟹ *distinct* (*map fst* (*entries t*))
⟨*proof*⟩

lemma *distinct-keys*:
  rbt-sorted t ⟹ *distinct* (*keys t*)
  ⟨*proof*⟩

110.2.3 Tree lookup

primrec (**in** *ord*) *rbt-lookup* :: $('a, 'b)$ *rbt* ⇒ $'a \rightharpoonup 'b$
where
  rbt-lookup Empty k = None
| *rbt-lookup* (*Branch - l x y r*) *k* =
   (*if k < x then rbt-lookup l k else if x < k then rbt-lookup r k else Some y*)

lemma *rbt-lookup-keys*: *rbt-sorted t* ⟹ *dom* (*rbt-lookup t*) = *set* (*keys t*)
  ⟨*proof*⟩

lemma *dom-rbt-lookup-Branch*:
  rbt-sorted (*Branch c t1 k v t2*) ⟹
    dom (*rbt-lookup* (*Branch c t1 k v t2*))
    = *Set.insert k* (*dom* (*rbt-lookup t1*) ∪ *dom* (*rbt-lookup t2*))
⟨*proof*⟩

lemma *finite-dom-rbt-lookup* [*simp*, *intro*!]: *finite* (*dom* (*rbt-lookup t*))
⟨*proof*⟩

end

context *ord* **begin**

lemma *rbt-lookup-rbt-less*[*simp*]: *t* |≪ *k* ⟹ *rbt-lookup t k* = *None*
⟨*proof*⟩

lemma *rbt-lookup-rbt-greater*[*simp*]: *k* ≪| *t* ⟹ *rbt-lookup t k* = *None*
⟨*proof*⟩

lemma *rbt-lookup-Empty*: *rbt-lookup Empty* = *empty*
⟨*proof*⟩

end

context *linorder* **begin**

lemma *map-of-entries*:
rbt-sorted t ⟹ *map-of* (*entries t*) = *rbt-lookup t*
⟨*proof*⟩

lemma *rbt-lookup-in-tree*: *rbt-sorted t* ⟹ *rbt-lookup t k* = *Some v* ⟷ (*k*, *v*) ∈ *set* (*entries t*)
⟨*proof*⟩

lemma *set-entries-inject*:
assumes *rbt-sorted*: *rbt-sorted t1 rbt-sorted t2*
shows *set* (*entries t1*) = *set* (*entries t2*) ⟷ *entries t1* = *entries t2*
⟨*proof*⟩

lemma *entries-eqI*:
assumes *rbt-sorted*: *rbt-sorted t1 rbt-sorted t2*
assumes *rbt-lookup*: *rbt-lookup t1* = *rbt-lookup t2*
shows *entries t1* = *entries t2*
⟨*proof*⟩

lemma *entries-rbt-lookup*:
assumes *rbt-sorted t1 rbt-sorted t2*
shows *entries t1* = *entries t2* ⟷ *rbt-lookup t1* = *rbt-lookup t2*
⟨*proof*⟩

lemma *rbt-lookup-from-in-tree*:
assumes *rbt-sorted t1 rbt-sorted t2*
and ⋀*v*. (*k*, *v*) ∈ *set* (*entries t1*) ⟷ (*k*, *v*) ∈ *set* (*entries t2*)
shows *rbt-lookup t1 k* = *rbt-lookup t2 k*
⟨*proof*⟩

end

110.2.4 Red-black properties

primrec *color-of* :: ('*a*, '*b*) *rbt* ⇒ *color*
where
color-of Empty = *B*
| *color-of* (*Branch c - - - -*) = *c*

primrec *bheight* :: ('*a*,'*b*) *rbt* ⇒ *nat*
where
bheight Empty = *0*
| *bheight* (*Branch c lt k v rt*) = (*if c* = *B then Suc* (*bheight lt*) *else bheight lt*)

primrec *inv1* :: ('*a*, '*b*) *rbt* ⇒ *bool*
where
inv1 Empty = *True*
| *inv1* (*Branch c lt k v rt*) ⟷ *inv1 lt* ∧ *inv1 rt* ∧ (*c* = *B* ∨ *color-of lt* = *B* ∧ *color-of rt* = *B*)

primrec *inv1l* :: ('*a*, '*b*) *rbt* ⇒ *bool* — Weaker version
where
inv1l Empty = *True*
| *inv1l* (*Branch c l k v r*) = (*inv1 l* ∧ *inv1 r*)
lemma [*simp*]: *inv1 t* ⟹ *inv1l t* ⟨*proof*⟩

primrec *inv2* :: ('*a*, '*b*) *rbt* ⇒ *bool*
where
inv2 Empty = *True*
| *inv2* (*Branch c lt k v rt*) = (*inv2 lt* ∧ *inv2 rt* ∧ *bheight lt* = *bheight rt*)

context *ord* **begin**

definition *is-rbt* :: ('*a*, '*b*) *rbt* ⇒ *bool* **where**
is-rbt t ⟷ *inv1 t* ∧ *inv2 t* ∧ *color-of t* = *B* ∧ *rbt-sorted t*

lemma *is-rbt-rbt-sorted* [*simp*]:
is-rbt t ⟹ *rbt-sorted t* ⟨*proof*⟩

theorem *Empty-is-rbt* [*simp*]:
is-rbt Empty ⟨*proof*⟩

end

110.3 Insertion

The function definitions are based on the book by Okasaki.

fun

balance :: ('*a*,'*b*) *rbt* ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*
where
balance (*Branch R a w x b*) *s t* (*Branch R c y z d*) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance (*Branch R* (*Branch R a w x b*) *s t c*) *y z d* = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance (*Branch R a w x* (*Branch R b s t c*)) *y z d* = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a w x (*Branch R b s t* (*Branch R c y z d*)) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a w x (*Branch R* (*Branch R b s t c*) *y z d*) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a s t b = *Branch B a s t b*

lemma *balance-inv1*: ⟦*inv1l l*; *inv1l r*⟧ ⟹ *inv1* (*balance l k v r*)
⟨*proof*⟩

lemma *balance-bheight*: *bheight l* = *bheight r* ⟹ *bheight* (*balance l k v r*) = *Suc* (*bheight l*)
⟨*proof*⟩

lemma *balance-inv2*:
assumes *inv2 l inv2 r bheight l* = *bheight r*
shows *inv2* (*balance l k v r*)
⟨*proof*⟩

context *ord* **begin**

lemma *balance-rbt-greater*[*simp*]: (*v* ≪| *balance a k x b*) = (*v* ≪| *a* ∧ *v* ≪| *b* ∧ *v* < *k*)
⟨*proof*⟩

lemma *balance-rbt-less*[*simp*]: (*balance a k x b* |≪ *v*) = (*a* |≪ *v* ∧ *b* |≪ *v* ∧ *k* < *v*)
⟨*proof*⟩

end

lemma (**in** *linorder*) *balance-rbt-sorted*:
fixes *k* :: '*a*
assumes *rbt-sorted l rbt-sorted r l* |≪ *k k* ≪| *r*
shows *rbt-sorted* (*balance l k v r*)
⟨*proof*⟩

lemma *entries-balance* [*simp*]:
entries (*balance l k v r*) = *entries l* @ (*k*, *v*) # *entries r*
⟨*proof*⟩

lemma *keys-balance* [*simp*]:
keys (*balance l k v r*) = *keys l* @ *k* # *keys r*

⟨*proof*⟩

lemma *balance-in-tree*:
entry-in-tree k x (*balance l v y r*) ⟷ *entry-in-tree k x l* ∨ *k* = *v* ∧ *x* = *y* ∨ *entry-in-tree k x r*
⟨*proof*⟩

lemma (**in** *linorder*) *rbt-lookup-balance*[*simp*]:
fixes *k* :: *'a*
assumes *rbt-sorted l rbt-sorted r l* |≪ *k k* ≪| *r*
shows *rbt-lookup* (*balance l k v r*) *x* = *rbt-lookup* (*Branch B l k v r*) *x*
⟨*proof*⟩

primrec *paint* :: *color* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
paint c Empty = *Empty*
| *paint c* (*Branch - l k v r*) = *Branch c l k v r*

lemma *paint-inv1l*[*simp*]: *inv1l t* ⟹ *inv1l* (*paint c t*) ⟨*proof*⟩
lemma *paint-inv1*[*simp*]: *inv1l t* ⟹ *inv1* (*paint B t*) ⟨*proof*⟩
lemma *paint-inv2*[*simp*]: *inv2 t* ⟹ *inv2* (*paint c t*) ⟨*proof*⟩
lemma *paint-color-of*[*simp*]: *color-of* (*paint B t*) = *B* ⟨*proof*⟩
lemma *paint-in-tree*[*simp*]: *entry-in-tree k x* (*paint c t*) = *entry-in-tree k x t* ⟨*proof*⟩

context *ord* **begin**

lemma *paint-rbt-sorted*[*simp*]: *rbt-sorted t* ⟹ *rbt-sorted* (*paint c t*) ⟨*proof*⟩
lemma *paint-rbt-lookup*[*simp*]: *rbt-lookup* (*paint c t*) = *rbt-lookup t* ⟨*proof*⟩
lemma *paint-rbt-greater*[*simp*]: (*v* ≪| *paint c t*) = (*v* ≪| *t*) ⟨*proof*⟩
lemma *paint-rbt-less*[*simp*]: (*paint c t* |≪ *v*) = (*t* |≪ *v*) ⟨*proof*⟩

fun
rbt-ins :: (*'a* ⇒ *'b* ⇒ *'b* ⇒ *'b*) ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
rbt-ins f k v Empty = *Branch R Empty k v Empty* |
rbt-ins f k v (*Branch B l x y r*) = (*if k* < *x then balance* (*rbt-ins f k v l*) *x y r*
else if k > *x then balance l x y* (*rbt-ins f k v r*)
else Branch B l x (*f k y v*) *r*) |
rbt-ins f k v (*Branch R l x y r*) = (*if k* < *x then Branch R* (*rbt-ins f k v l*) *x y r*
else if k > *x then Branch R l x y* (*rbt-ins f k v r*)
else Branch R l x (*f k y v*) *r*)

lemma *ins-inv1-inv2*:
assumes *inv1 t inv2 t*
shows *inv2* (*rbt-ins f k x t*) *bheight* (*rbt-ins f k x t*) = *bheight t*
color-of t = *B* ⟹ *inv1* (*rbt-ins f k x t*) *inv1l* (*rbt-ins f k x t*)
⟨*proof*⟩

end

context *linorder* **begin**

lemma *ins-rbt-greater*[*simp*]: $(v \ll| \textit{rbt-ins}\ f\ (k :: 'a)\ x\ t) = (v \ll| t \wedge k > v)$
⟨*proof*⟩
lemma *ins-rbt-less*[*simp*]: $(\textit{rbt-ins}\ f\ k\ x\ t \mid\ll v) = (t \mid\ll v \wedge k < v)$
⟨*proof*⟩
lemma *ins-rbt-sorted*[*simp*]: *rbt-sorted* $t \Longrightarrow$ *rbt-sorted* (*rbt-ins f k x t*)
⟨*proof*⟩

lemma *keys-ins*: *set* (*keys* (*rbt-ins f k v t*)) = { *k* } ∪ *set* (*keys t*)
⟨*proof*⟩

lemma *rbt-lookup-ins*:
fixes $k :: 'a$
assumes *rbt-sorted t*
shows *rbt-lookup* (*rbt-ins f k v t*) *x* = ((*rbt-lookup t*)(*k* |−> *case rbt-lookup t k of None* ⇒ *v*
| *Some w* ⇒ *f k w v*)) *x*
⟨*proof*⟩

end

context *ord* **begin**

definition *rbt-insert-with-key* :: ('*a* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*
where *rbt-insert-with-key f k v t* = *paint B* (*rbt-ins f k v t*)

definition *rbt-insertw-def*: *rbt-insert-with f* = *rbt-insert-with-key* (λ-. *f*)

definition *rbt-insert* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* **where**
rbt-insert = *rbt-insert-with-key* (λ- - *nv*. *nv*)

end

context *linorder* **begin**

lemma *rbt-insertwk-rbt-sorted*: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-insert-with-key f* (*k* :: '*a*) *x t*)
⟨*proof*⟩

theorem *rbt-insertwk-is-rbt*:
assumes *inv*: *is-rbt t*
shows *is-rbt* (*rbt-insert-with-key f k x t*)
⟨*proof*⟩

lemma *rbt-lookup-rbt-insertwk*:
assumes *rbt-sorted t*

shows *rbt-lookup* (*rbt-insert-with-key f k v t*) *x* = ((*rbt-lookup t*)(*k* |–> *case rbt-lookup t k of None* ⇒ *v*
| *Some w* ⇒ *f k w v*)) *x*
⟨*proof*⟩

lemma *rbt-insertw-rbt-sorted*: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-insert-with f k v t*)
⟨*proof*⟩
theorem *rbt-insertw-is-rbt*: *is-rbt t* ⟹ *is-rbt* (*rbt-insert-with f k v t*)
⟨*proof*⟩

lemma *rbt-lookup-rbt-insertw*:
is-rbt t ⟹
rbt-lookup (*rbt-insert-with f k v t*) =
(*rbt-lookup t*)(*k* ↦ (*if k* ∈ *dom* (*rbt-lookup t*) *then f* (*the* (*rbt-lookup t k*)) *v* *else v*))
⟨*proof*⟩

lemma *rbt-insert-rbt-sorted*: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-insert k v t*)
⟨*proof*⟩
theorem *rbt-insert-is-rbt* [*simp*]: *is-rbt t* ⟹ *is-rbt* (*rbt-insert k v t*)
⟨*proof*⟩

lemma *rbt-lookup-rbt-insert*: *is-rbt t* ⟹ *rbt-lookup* (*rbt-insert k v t*) = (*rbt-lookup t*)(*k*↦*v*)
⟨*proof*⟩

end

110.4 Deletion

lemma *bheight-paintR'*[*simp*]: *color-of t* = *B* ⟹ *bheight* (*paint R t*) = *bheight t* − *1*
⟨*proof*⟩

The function definitions are based on the Haskell code by Stefan Kahrs at http://www.cs.ukc.ac.uk/people/staff/smk/redblack/rb.html.

fun
balance-left :: ('*a*,'*b*) *rbt* ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*
where
balance-left (*Branch R a k x b*) *s y c* = *Branch R* (*Branch B a k x b*) *s y c* |
balance-left bl k x (*Branch B a s y b*) = *balance bl k x* (*Branch R a s y b*) |
balance-left bl k x (*Branch R* (*Branch B a s y b*) *t z c*) = *Branch R* (*Branch B bl k x a*) *s y* (*balance b t z* (*paint R c*)) |
balance-left t k x s = *Empty*

lemma *balance-left-inv2-with-inv1*:
assumes *inv2 lt inv2 rt bheight lt* + *1* = *bheight rt inv1 rt*
shows *bheight* (*balance-left lt k v rt*) = *bheight lt* + *1*
and *inv2* (*balance-left lt k v rt*)

⟨*proof*⟩

lemma *balance-left-inv2-app*:
assumes *inv2 lt inv2 rt bheight lt + 1 = bheight rt color-of rt = B*
shows *inv2* (*balance-left lt k v rt*)
bheight (*balance-left lt k v rt*) = *bheight rt*
⟨*proof*⟩

lemma *balance-left-inv1*: ⟦*inv1l a*; *inv1 b*; *color-of b = B*⟧ ⟹ *inv1* (*balance-left a k x b*)
⟨*proof*⟩

lemma *balance-left-inv1l*: ⟦ *inv1l lt*; *inv1 rt* ⟧ ⟹ *inv1l* (*balance-left lt k x rt*)
⟨*proof*⟩

lemma (**in** *linorder*) *balance-left-rbt-sorted*:
⟦ *rbt-sorted l*; *rbt-sorted r*; *rbt-less k l*; *k* ≪| *r* ⟧ ⟹ *rbt-sorted* (*balance-left l k v r*)
⟨*proof*⟩

context *order* **begin**

lemma *balance-left-rbt-greater*:
fixes *k* :: *'a*
assumes *k* ≪| *a k* ≪| *b k* < *x*
shows *k* ≪| *balance-left a x t b*
⟨*proof*⟩

lemma *balance-left-rbt-less*:
fixes *k* :: *'a*
assumes *a* |≪ *k b* |≪ *k x* < *k*
shows *balance-left a x t b* |≪ *k*
⟨*proof*⟩

end

lemma *balance-left-in-tree*:
assumes *inv1l l inv1 r bheight l + 1 = bheight r*
shows *entry-in-tree k v* (*balance-left l a b r*) = (*entry-in-tree k v l* ∨ *k = a* ∧ *v = b* ∨ *entry-in-tree k v r*)
⟨*proof*⟩

fun
balance-right :: (*'a*,*'b*) *rbt* ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
balance-right a k x (*Branch R b s y c*) = *Branch R a k x* (*Branch B b s y c*) |
balance-right (*Branch B a k x b*) *s y bl* = *balance* (*Branch R a k x b*) *s y bl* |
balance-right (*Branch R a k x* (*Branch B b s y c*)) *t z bl* = *Branch R* (*balance* (*paint R a*) *k x b*) *s y* (*Branch B c t z bl*) |

balance-right t k x s = Empty

lemma *balance-right-inv2-with-inv1*:
assumes *inv2 lt inv2 rt bheight lt = bheight rt + 1 inv1 lt*
shows *inv2 (balance-right lt k v rt) ∧ bheight (balance-right lt k v rt) = bheight lt*
⟨*proof*⟩

lemma *balance-right-inv1*: ⟦*inv1 a*; *inv1l b*; *color-of a = B*⟧ ⟹ *inv1 (balance-right a k x b)*
⟨*proof*⟩

lemma *balance-right-inv1l*: ⟦ *inv1 lt*; *inv1l rt* ⟧ ⟹*inv1l (balance-right lt k x rt)*
⟨*proof*⟩

lemma (**in** *linorder*) *balance-right-rbt-sorted*:
⟦ *rbt-sorted l*; *rbt-sorted r*; *rbt-less k l*; *k ≪| r* ⟧ ⟹ *rbt-sorted (balance-right l k v r)*
⟨*proof*⟩

context *order* **begin**

lemma *balance-right-rbt-greater*:
fixes *k* :: *'a*
assumes *k ≪| a k ≪| b k < x*
shows *k ≪| balance-right a x t b*
⟨*proof*⟩

lemma *balance-right-rbt-less*:
fixes *k* :: *'a*
assumes *a |≪ k b |≪ k x < k*
shows *balance-right a x t b |≪ k*
⟨*proof*⟩

end

lemma *balance-right-in-tree*:
assumes *inv1 l inv1l r bheight l = bheight r + 1 inv2 l inv2 r*
shows *entry-in-tree x y (balance-right l k v r) = (entry-in-tree x y l ∨ x = k ∧ y = v ∨ entry-in-tree x y r)*
⟨*proof*⟩

fun
combine :: *('a,'b) rbt ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt*
where
combine Empty x = x
| *combine x Empty = x*
| *combine (Branch R a k x b) (Branch R c s y d) = (case (combine b c) of*
Branch R b2 t z c2 ⇒ (Branch R (Branch R a k x

b2) *t z* (*Branch R c2 s y d*)) |
bc ⇒ *Branch R a k x* (*Branch R bc s y d*))
| *combine* (*Branch B a k x b*) (*Branch B c s y d*) = (*case* (*combine b c*) *of*
Branch R b2 t z c2 ⇒ *Branch R* (*Branch B a k x b2*)
t z (*Branch B c2 s y d*) |
bc ⇒ *balance-left a k x* (*Branch B bc s y d*))
| *combine a* (*Branch R b k x c*) = *Branch R* (*combine a b*) *k x c*
| *combine* (*Branch R a k x b*) *c* = *Branch R a k x* (*combine b c*)

lemma *combine-inv2*:
assumes *inv2 lt inv2 rt bheight lt* = *bheight rt*
shows *bheight* (*combine lt rt*) = *bheight lt inv2* (*combine lt rt*)
⟨*proof*⟩

lemma *combine-inv1*:
assumes *inv1 lt inv1 rt*
shows *color-of lt* = *B* ⟹ *color-of rt* = *B* ⟹ *inv1* (*combine lt rt*)
inv1l (*combine lt rt*)
⟨*proof*⟩

context *linorder* **begin**

lemma *combine-rbt-greater*[*simp*]:
fixes *k* :: *'a*
assumes *k* ≪| *l k* ≪| *r*
shows *k* ≪| *combine l r*
⟨*proof*⟩

lemma *combine-rbt-less*[*simp*]:
fixes *k* :: *'a*
assumes *l* |≪ *k r* |≪ *k*
shows *combine l r* |≪ *k*
⟨*proof*⟩

lemma *combine-rbt-sorted*:
fixes *k* :: *'a*
assumes *rbt-sorted l rbt-sorted r l* |≪ *k k* ≪| *r*
shows *rbt-sorted* (*combine l r*)
⟨*proof*⟩

end

lemma *combine-in-tree*:
assumes *inv2 l inv2 r bheight l* = *bheight r inv1 l inv1 r*
shows *entry-in-tree k v* (*combine l r*) = (*entry-in-tree k v l* ∨ *entry-in-tree k v r*)
⟨*proof*⟩

context *ord* **begin**

fun
rbt-del-from-left :: $'a \Rightarrow ('a,'b)\ rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a,'b)\ rbt \Rightarrow ('a,'b)\ rbt$ **and**
rbt-del-from-right :: $'a \Rightarrow ('a,'b)\ rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a,'b)\ rbt \Rightarrow ('a,'b)\ rbt$ **and**
rbt-del :: $'a \Rightarrow ('a,'b)\ rbt \Rightarrow ('a,'b)\ rbt$
where
rbt-del x Empty = Empty |
rbt-del x (Branch c a y s b) =
(if x < y then rbt-del-from-left x a y s b
else (if x > y then rbt-del-from-right x a y s b else combine a b)) |
rbt-del-from-left x (Branch B lt z v rt) y s b = balance-left (rbt-del x (Branch B lt z v rt)) y s b |
rbt-del-from-left x a y s b = Branch R (rbt-del x a) y s b |
rbt-del-from-right x a y s (Branch B lt z v rt) = balance-right a y s (rbt-del x (Branch B lt z v rt)) |
rbt-del-from-right x a y s b = Branch R a y s (rbt-del x b)

end

context *linorder* **begin**

lemma
assumes *inv2 lt inv1 lt*
shows
$[\![$*inv2 rt*; *bheight lt = bheight rt*; *inv1 rt*$]\!] \Longrightarrow$
inv2 (rbt-del-from-left x lt k v rt) ∧
bheight (rbt-del-from-left x lt k v rt) = bheight lt ∧
(color-of lt = B ∧ color-of rt = B ∧ inv1 (rbt-del-from-left x lt k v rt) ∨
(color-of lt ≠ B ∨ color-of rt ≠ B) ∧ inv1l (rbt-del-from-left x lt k v rt))
and $[\![$*inv2 rt*; *bheight lt = bheight rt*; *inv1 rt*$]\!] \Longrightarrow$
inv2 (rbt-del-from-right x lt k v rt) ∧
bheight (rbt-del-from-right x lt k v rt) = bheight lt ∧
(color-of lt = B ∧ color-of rt = B ∧ inv1 (rbt-del-from-right x lt k v rt) ∨
(color-of lt ≠ B ∨ color-of rt ≠ B) ∧ inv1l (rbt-del-from-right x lt k v rt))
and *rbt-del-inv1-inv2*: *inv2 (rbt-del x lt) ∧ (color-of lt = R ∧ bheight (rbt-del x lt) = bheight lt ∧ inv1 (rbt-del x lt)*
∨ color-of lt = B ∧ bheight (rbt-del x lt) = bheight lt − 1 ∧ inv1l (rbt-del x lt))
⟨*proof*⟩

lemma
rbt-del-from-left-rbt-less: $[\![$ *lt* $|\ll$ *v*; *rt* $|\ll$ *v*; *k < v*$]\!] \Longrightarrow$ *rbt-del-from-left x lt k y rt* $|\ll$ *v*
and *rbt-del-from-right-rbt-less*: $[\![$*lt* $|\ll$ *v*; *rt* $|\ll$ *v*; *k < v*$]\!] \Longrightarrow$ *rbt-del-from-right x lt k y rt* $|\ll$ *v*
and *rbt-del-rbt-less*: *lt* $|\ll$ *v* $\Longrightarrow$ *rbt-del x lt* $|\ll$ *v*
⟨*proof*⟩

lemma *rbt-del-from-left-rbt-greater*: $[\![$*v* $\ll|$ *lt*; *v* $\ll|$ *rt*; *k > v*$]\!] \Longrightarrow$ *v* $\ll|$ *rbt-del-from-left x lt k y rt*

and *rbt-del-from-right-rbt-greater*: $[\![v \ll| lt; v \ll| rt; k > v]\!] \Longrightarrow v \ll| rbt\text{-}del\text{-}from\text{-}right\ x\ lt\ k\ y\ rt$
and *rbt-del-rbt-greater*: $v \ll| lt \Longrightarrow v \ll| rbt\text{-}del\ x\ lt$
⟨*proof*⟩

lemma $[\![rbt\text{-}sorted\ lt; rbt\text{-}sorted\ rt; lt \mid\ll k; k \ll| rt]\!] \Longrightarrow rbt\text{-}sorted\ (rbt\text{-}del\text{-}from\text{-}left\ x\ lt\ k\ y\ rt)$
and $[\![rbt\text{-}sorted\ lt; rbt\text{-}sorted\ rt; lt \mid\ll k; k \ll| rt]\!] \Longrightarrow rbt\text{-}sorted\ (rbt\text{-}del\text{-}from\text{-}right\ x\ lt\ k\ y\ rt)$
and *rbt-del-rbt-sorted*: $rbt\text{-}sorted\ lt \Longrightarrow rbt\text{-}sorted\ (rbt\text{-}del\ x\ lt)$
⟨*proof*⟩

lemma $[\![rbt\text{-}sorted\ lt; rbt\text{-}sorted\ rt; lt \mid\ll kt; kt \ll| rt; inv1\ lt; inv1\ rt; inv2\ lt; inv2\ rt; bheight\ lt = bheight\ rt; x < kt]\!] \Longrightarrow entry\text{-}in\text{-}tree\ k\ v\ (rbt\text{-}del\text{-}from\text{-}left\ x\ lt\ kt\ y\ rt) = (False \vee (x \neq k \wedge entry\text{-}in\text{-}tree\ k\ v\ (Branch\ c\ lt\ kt\ y\ rt)))$
and $[\![rbt\text{-}sorted\ lt; rbt\text{-}sorted\ rt; lt \mid\ll kt; kt \ll| rt; inv1\ lt; inv1\ rt; inv2\ lt; inv2\ rt; bheight\ lt = bheight\ rt; x > kt]\!] \Longrightarrow entry\text{-}in\text{-}tree\ k\ v\ (rbt\text{-}del\text{-}from\text{-}right\ x\ lt\ kt\ y\ rt) = (False \vee (x \neq k \wedge entry\text{-}in\text{-}tree\ k\ v\ (Branch\ c\ lt\ kt\ y\ rt)))$
and *rbt-del-in-tree*: $[\![rbt\text{-}sorted\ t; inv1\ t; inv2\ t]\!] \Longrightarrow entry\text{-}in\text{-}tree\ k\ v\ (rbt\text{-}del\ x\ t) = (False \vee (x \neq k \wedge entry\text{-}in\text{-}tree\ k\ v\ t))$
⟨*proof*⟩

definition (**in** *ord*) *rbt-delete* **where**
$rbt\text{-}delete\ k\ t = paint\ B\ (rbt\text{-}del\ k\ t)$

theorem *rbt-delete-is-rbt* [*simp*]: **assumes** *is-rbt t* **shows** *is-rbt* $(rbt\text{-}delete\ k\ t)$
⟨*proof*⟩

lemma *rbt-delete-in-tree*:
assumes *is-rbt t*
shows $entry\text{-}in\text{-}tree\ k\ v\ (rbt\text{-}delete\ x\ t) = (x \neq k \wedge entry\text{-}in\text{-}tree\ k\ v\ t)$
⟨*proof*⟩

lemma *rbt-lookup-rbt-delete*:
assumes *is-rbt*: *is-rbt t*
shows $rbt\text{-}lookup\ (rbt\text{-}delete\ k\ t) = (rbt\text{-}lookup\ t) |`(-\{k\})$
⟨*proof*⟩

end

110.5 Modifying existing entries

context *ord* **begin**

primrec
$rbt\text{-}map\text{-}entry :: 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$
where
$rbt\text{-}map\text{-}entry\ k\ f\ Empty = Empty$
$\mid rbt\text{-}map\text{-}entry\ k\ f\ (Branch\ c\ lt\ x\ v\ rt) =$

(*if* $k < x$ *then Branch* c (*rbt-map-entry* k f lt) x v rt
else if $k > x$ *then* (*Branch* c lt x v (*rbt-map-entry* k f rt))
else Branch c lt x (f v) rt)

lemma *rbt-map-entry-color-of*: *color-of* (*rbt-map-entry* k f t) = *color-of* t ⟨*proof*⟩
lemma *rbt-map-entry-inv1*: *inv1* (*rbt-map-entry* k f t) = *inv1* t ⟨*proof*⟩
lemma *rbt-map-entry-inv2*: *inv2* (*rbt-map-entry* k f t) = *inv2* t *bheight* (*rbt-map-entry* k f t) = *bheight* t ⟨*proof*⟩
lemma *rbt-map-entry-rbt-greater*: *rbt-greater* a (*rbt-map-entry* k f t) = *rbt-greater* a t ⟨*proof*⟩
lemma *rbt-map-entry-rbt-less*: *rbt-less* a (*rbt-map-entry* k f t) = *rbt-less* a t ⟨*proof*⟩
lemma *rbt-map-entry-rbt-sorted*: *rbt-sorted* (*rbt-map-entry* k f t) = *rbt-sorted* t
⟨*proof*⟩

theorem *rbt-map-entry-is-rbt* [*simp*]: *is-rbt* (*rbt-map-entry* k f t) = *is-rbt* t
⟨*proof*⟩

end

theorem (**in** *linorder*) *rbt-lookup-rbt-map-entry*:
rbt-lookup (*rbt-map-entry* k f t) = (*rbt-lookup* t)(k := *map-option* f (*rbt-lookup* t k))
⟨*proof*⟩

110.6 Mapping all entries

primrec
map :: ('a ⇒ 'b ⇒ 'c) ⇒ ('a, 'b) *rbt* ⇒ ('a, 'c) *rbt*
where
map f *Empty* = *Empty*
| *map* f (*Branch* c lt k v rt) = *Branch* c (*map* f lt) k (f k v) (*map* f rt)

lemma *map-entries* [*simp*]: *entries* (*map* f t) = *List.map* (λ(k, v). (k, f k v)) (*entries* t)
⟨*proof*⟩
lemma *map-keys* [*simp*]: *keys* (*map* f t) = *keys* t ⟨*proof*⟩
lemma *map-color-of*: *color-of* (*map* f t) = *color-of* t ⟨*proof*⟩
lemma *map-inv1*: *inv1* (*map* f t) = *inv1* t ⟨*proof*⟩
lemma *map-inv2*: *inv2* (*map* f t) = *inv2* t *bheight* (*map* f t) = *bheight* t ⟨*proof*⟩

context *ord* **begin**

lemma *map-rbt-greater*: *rbt-greater* k (*map* f t) = *rbt-greater* k t ⟨*proof*⟩
lemma *map-rbt-less*: *rbt-less* k (*map* f t) = *rbt-less* k t ⟨*proof*⟩
lemma *map-rbt-sorted*: *rbt-sorted* (*map* f t) = *rbt-sorted* t ⟨*proof*⟩
theorem *map-is-rbt* [*simp*]: *is-rbt* (*map* f t) = *is-rbt* t
⟨*proof*⟩

end

theorem (**in** *linorder*) *rbt-lookup-map*: *rbt-lookup* (*map f t*) *x* = *map-option* (*f x*) (*rbt-lookup t x*)
⟨*proof*⟩

hide-const (**open**) *map*

110.7 Folding over entries

definition *fold* :: ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*c*) ⇒ ('*a*, '*b*) *rbt* ⇒ '*c* ⇒ '*c* **where**
fold f t = *List.fold* (*case-prod f*) (*entries t*)

lemma *fold-simps* [*simp*]:
fold f Empty = *id*
fold f (*Branch c lt k v rt*) = *fold f rt* ∘ *f k v* ∘ *fold f lt*
⟨*proof*⟩

lemma *fold-code* [*code*]:
fold f Empty x = *x*
fold f (*Branch c lt k v rt*) *x* = *fold f rt* (*f k v* (*fold f lt x*))
⟨*proof*⟩

fun *foldi* :: ('*c* ⇒ *bool*) ⇒ ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*c*) ⇒ ('*a* :: *linorder*, '*b*) *rbt* ⇒ '*c* ⇒ '*c*
where
foldi c f Empty s = *s* |
foldi c f (*Branch col l k v r*) *s* = (
if (*c s*) *then*
let s' = *foldi c f l s in*
if (*c s'*) *then*
foldi c f r (*f k v s'*)
else s'
else
s
)

110.8 Bulkloading a tree

definition (**in** *ord*) *rbt-bulkload* :: ('*a* × '*b*) *list* ⇒ ('*a*, '*b*) *rbt* **where**
rbt-bulkload xs = *foldr* (λ(*k*, *v*). *rbt-insert k v*) *xs Empty*

context *linorder* **begin**

lemma *rbt-bulkload-is-rbt* [*simp*, *intro*]:
is-rbt (*rbt-bulkload xs*)
⟨*proof*⟩

lemma *rbt-lookup-rbt-bulkload*:
rbt-lookup (*rbt-bulkload xs*) = *map-of xs*
⟨*proof*⟩

end

110.9 Building a RBT from a sorted list

These functions have been adapted from Andrew W. Appel, Efficient Verified Red-Black Trees (September 2011)

fun *rbtreeify-f* :: *nat* ⇒ (*'a* × *'b*) *list* ⇒ (*'a*, *'b*) *rbt* × (*'a* × *'b*) *list*
and *rbtreeify-g* :: *nat* ⇒ (*'a* × *'b*) *list* ⇒ (*'a*, *'b*) *rbt* × (*'a* × *'b*) *list*
where
rbtreeify-f n kvs =
(*if n* = *0 then* (*Empty*, *kvs*)
else if n = *1 then*
case kvs of (*k*, *v*) # *kvs'* ⇒ (*Branch R Empty k v Empty*, *kvs'*)
else if (*n mod 2* = *0*) *then*
case rbtreeify-f (*n div 2*) *kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g* (*n div 2*) *kvs'*)
else case rbtreeify-f (*n div 2*) *kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-f* (*n div 2*) *kvs'*))

| *rbtreeify-g n kvs* =
(*if n* = *0* ∨ *n* = *1 then* (*Empty*, *kvs*)
else if n mod 2 = *0 then*
case rbtreeify-g (*n div 2*) *kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g* (*n div 2*) *kvs'*)
else case rbtreeify-f (*n div 2*) *kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g* (*n div 2*) *kvs'*))

definition *rbtreeify* :: (*'a* × *'b*) *list* ⇒ (*'a*, *'b*) *rbt*
where *rbtreeify kvs* = *fst* (*rbtreeify-g* (*Suc* (*length kvs*)) *kvs*)

declare *rbtreeify-f.simps* [*simp del*] *rbtreeify-g.simps* [*simp del*]

lemma *rbtreeify-f-code* [*code*]:
rbtreeify-f n kvs =
(*if n* = *0 then* (*Empty*, *kvs*)
else if n = *1 then*
case kvs of (*k*, *v*) # *kvs'* ⇒
(*Branch R Empty k v Empty*, *kvs'*)
else let (*n'*, *r*) = *Divides.divmod-nat n 2 in*
if r = *0 then*
case rbtreeify-f n' kvs of (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g n' kvs'*)
else case rbtreeify-f n' kvs of (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-f n' kvs'*))

⟨*proof*⟩

lemma *rbtreeify-g-code* [*code*]:
rbtreeify-g n kvs =
(*if n* = *0* ∨ *n* = *1 then* (*Empty*, *kvs*)
else let (*n'*, *r*) = *Divides.divmod-nat n 2 in*
if r = *0 then*
case rbtreeify-g n' kvs of (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g n' kvs'*)
else case rbtreeify-f n' kvs of (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g n' kvs'*))
⟨*proof*⟩

lemma *Suc-double-half*: *Suc* (*2* ∗ *n*) *div 2* = *n*
⟨*proof*⟩

lemma *div2-plus-div2*: *n div 2* + *n div 2* = (*n* :: *nat*) − *n mod 2*
⟨*proof*⟩

lemma *rbtreeify-f-rec-aux-lemma*:
⟦*k* − *n div 2* = *Suc k'*; *n* ≤ *k*; *n mod 2* = *Suc 0*⟧
⟹ *k'* − *n div 2* = *k* − *n*
⟨*proof*⟩

lemma *rbtreeify-f-simps*:
rbtreeify-f 0 kvs = (*Empty*, *kvs*)
rbtreeify-f (*Suc 0*) ((*k*, *v*) # *kvs*) =
(*Branch R Empty k v Empty*, *kvs*)
0 < *n* ⟹ *rbtreeify-f* (*2* ∗ *n*) *kvs* =
(*case rbtreeify-f n kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g n kvs'*))
0 < *n* ⟹ *rbtreeify-f* (*Suc* (*2* ∗ *n*)) *kvs* =
(*case rbtreeify-f n kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-f n kvs'*))
⟨*proof*⟩

lemma *rbtreeify-g-simps*:
rbtreeify-g 0 kvs = (*Empty*, *kvs*)
rbtreeify-g (*Suc 0*) *kvs* = (*Empty*, *kvs*)
0 < *n* ⟹ *rbtreeify-g* (*2* ∗ *n*) *kvs* =
(*case rbtreeify-g n kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g n kvs'*))
0 < *n* ⟹ *rbtreeify-g* (*Suc* (*2* ∗ *n*)) *kvs* =
(*case rbtreeify-f n kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g n kvs'*))
⟨*proof*⟩

declare *rbtreeify-f-simps*[*simp*] *rbtreeify-g-simps*[*simp*]

lemma *length-rbtreeify-f*: $n \leq$ *length kvs*
$\Longrightarrow$ *length* (*snd* (*rbtreeify-f n kvs*)) = *length kvs* − *n*
and *length-rbtreeify-g*:⟦ *0* < *n*; *n* ≤ *Suc* (*length kvs*) ⟧
$\Longrightarrow$ *length* (*snd* (*rbtreeify-g n kvs*)) = *Suc* (*length kvs*) − *n*
⟨*proof*⟩

lemma *rbtreeify-induct* [*consumes 1*, *case-names f-0 f-1 f-even f-odd g-0 g-1 g-even g-odd*]:
fixes *P Q*
defines *f0* == (⋀*kvs*. *P 0 kvs*)
and *f1* == (⋀*k v kvs*. *P* (*Suc 0*) ((*k*, *v*) # *kvs*))
and *feven* ==
(⋀*n kvs t k v kvs'*. ⟦ *n* > *0*; *n* ≤ *length kvs*; *P n kvs*;
rbtreeify-f n kvs = (*t*, (*k*, *v*) # *kvs'*); *n* ≤ *Suc* (*length kvs'*); *Q n kvs'* ⟧
$\Longrightarrow$ *P* (*2* ∗ *n*) *kvs*)
and *fodd* ==
(⋀*n kvs t k v kvs'*. ⟦ *n* > *0*; *n* ≤ *length kvs*; *P n kvs*;
rbtreeify-f n kvs = (*t*, (*k*, *v*) # *kvs'*); *n* ≤ *length kvs'*; *P n kvs'* ⟧
$\Longrightarrow$ *P* (*Suc* (*2* ∗ *n*)) *kvs*)
and *g0* == (⋀*kvs*. *Q 0 kvs*)
and *g1* == (⋀*kvs*. *Q* (*Suc 0*) *kvs*)
and *geven* ==
(⋀*n kvs t k v kvs'*. ⟦ *n* > *0*; *n* ≤ *Suc* (*length kvs*); *Q n kvs*;
rbtreeify-g n kvs = (*t*, (*k*, *v*) # *kvs'*); *n* ≤ *Suc* (*length kvs'*); *Q n kvs'* ⟧
$\Longrightarrow$ *Q* (*2* ∗ *n*) *kvs*)
and *godd* ==
(⋀*n kvs t k v kvs'*. ⟦ *n* > *0*; *n* ≤ *length kvs*; *P n kvs*;
rbtreeify-f n kvs = (*t*, (*k*, *v*) # *kvs'*); *n* ≤ *Suc* (*length kvs'*); *Q n kvs'* ⟧
$\Longrightarrow$ *Q* (*Suc* (*2* ∗ *n*)) *kvs*)
shows ⟦ *n* ≤ *length kvs*;
PROP f0; *PROP f1*; *PROP feven*; *PROP fodd*;
PROP g0; *PROP g1*; *PROP geven*; *PROP godd* ⟧
$\Longrightarrow$ *P n kvs*
and ⟦ *n* ≤ *Suc* (*length kvs*);
PROP f0; *PROP f1*; *PROP feven*; *PROP fodd*;
PROP g0; *PROP g1*; *PROP geven*; *PROP godd* ⟧
$\Longrightarrow$ *Q n kvs*
⟨*proof*⟩

lemma *inv1-rbtreeify-f*: *n* ≤ *length kvs*
$\Longrightarrow$ *inv1* (*fst* (*rbtreeify-f n kvs*))
and *inv1-rbtreeify-g*: *n* ≤ *Suc* (*length kvs*)
$\Longrightarrow$ *inv1* (*fst* (*rbtreeify-g n kvs*))
⟨*proof*⟩

fun *plog2* :: *nat* ⇒ *nat*
where *plog2 n* = (*if n* ≤ *1 then 0 else plog2* (*n div 2*) + *1*)

declare *plog2.simps* [*simp del*]

lemma *plog2-simps* [*simp*]:
plog2 0 = 0 plog2 (Suc 0) = 0
*0 < n ⟹ plog2 (2 * n) = 1 + plog2 n*
*0 < n ⟹ plog2 (Suc (2 * n)) = 1 + plog2 n*
⟨*proof*⟩

lemma *bheight-rbtreeify-f*: *n ≤ length kvs*
⟹ bheight (fst (rbtreeify-f n kvs)) = plog2 n
and *bheight-rbtreeify-g*: *n ≤ Suc (length kvs)*
⟹ bheight (fst (rbtreeify-g n kvs)) = plog2 n
⟨*proof*⟩

lemma *bheight-rbtreeify-f-eq-plog2I*:
⟦ *rbtreeify-f n kvs = (t, kvs′)*; *n ≤ length kvs* ⟧
⟹ bheight t = plog2 n
⟨*proof*⟩

lemma *bheight-rbtreeify-g-eq-plog2I*:
⟦ *rbtreeify-g n kvs = (t, kvs′)*; *n ≤ Suc (length kvs)* ⟧
⟹ bheight t = plog2 n
⟨*proof*⟩

hide-const (**open**) *plog2*

lemma *inv2-rbtreeify-f*: *n ≤ length kvs*
⟹ inv2 (fst (rbtreeify-f n kvs))
and *inv2-rbtreeify-g*: *n ≤ Suc (length kvs)*
⟹ inv2 (fst (rbtreeify-g n kvs))
⟨*proof*⟩

lemma *n ≤ length kvs ⟹ True*
and *color-of-rbtreeify-g*:
⟦ *n ≤ Suc (length kvs)*; *0 < n* ⟧
⟹ color-of (fst (rbtreeify-g n kvs)) = B
⟨*proof*⟩

lemma *entries-rbtreeify-f-append*:
n ≤ length kvs
⟹ entries (fst (rbtreeify-f n kvs)) @ snd (rbtreeify-f n kvs) = kvs
and *entries-rbtreeify-g-append*:
n ≤ Suc (length kvs)
⟹ entries (fst (rbtreeify-g n kvs)) @ snd (rbtreeify-g n kvs) = kvs
⟨*proof*⟩

lemma *length-entries-rbtreeify-f*:
n ≤ length kvs ⟹ length (entries (fst (rbtreeify-f n kvs))) = n
and *length-entries-rbtreeify-g*:
n ≤ Suc (length kvs) ⟹ length (entries (fst (rbtreeify-g n kvs))) = n − 1

⟨*proof*⟩

lemma *rbtreeify-f-conv-drop*:
  $n \leq$ *length kvs* $\Longrightarrow$ *snd* (*rbtreeify-f n kvs*) = *drop n kvs*
⟨*proof*⟩

lemma *rbtreeify-g-conv-drop*:
  $n \leq$ *Suc* (*length kvs*) $\Longrightarrow$ *snd* (*rbtreeify-g n kvs*) = *drop* (*n* − *1*) *kvs*
⟨*proof*⟩

lemma *entries-rbtreeify-f* [*simp*]:
  $n \leq$ *length kvs* $\Longrightarrow$ *entries* (*fst* (*rbtreeify-f n kvs*)) = *take n kvs*
⟨*proof*⟩

lemma *entries-rbtreeify-g* [*simp*]:
  $n \leq$ *Suc* (*length kvs*) $\Longrightarrow$
  entries (*fst* (*rbtreeify-g n kvs*)) = *take* (*n* − *1*) *kvs*
⟨*proof*⟩

lemma *keys-rbtreeify-f* [*simp*]: $n \leq$ *length kvs*
  $\Longrightarrow$ *keys* (*fst* (*rbtreeify-f n kvs*)) = *take n* (*map fst kvs*)
⟨*proof*⟩

lemma *keys-rbtreeify-g* [*simp*]: $n \leq$ *Suc* (*length kvs*)
  $\Longrightarrow$ *keys* (*fst* (*rbtreeify-g n kvs*)) = *take* (*n* − *1*) (*map fst kvs*)
⟨*proof*⟩

lemma *rbtreeify-fD*:
  ⟦ *rbtreeify-f n kvs* = (*t*, *kvs'*); $n \leq$ *length kvs* ⟧
  $\Longrightarrow$ *entries t* = *take n kvs* ∧ *kvs'* = *drop n kvs*
⟨*proof*⟩

lemma *rbtreeify-gD*:
  ⟦ *rbtreeify-g n kvs* = (*t*, *kvs'*); $n \leq$ *Suc* (*length kvs*) ⟧
  $\Longrightarrow$ *entries t* = *take* (*n* − *1*) *kvs* ∧ *kvs'* = *drop* (*n* − *1*) *kvs*
⟨*proof*⟩

lemma *entries-rbtreeify* [*simp*]: *entries* (*rbtreeify kvs*) = *kvs*
⟨*proof*⟩

context *linorder* **begin**

lemma *rbt-sorted-rbtreeify-f*:
  ⟦ $n \leq$ *length kvs*; *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧
  $\Longrightarrow$ *rbt-sorted* (*fst* (*rbtreeify-f n kvs*))
  and *rbt-sorted-rbtreeify-g*:
  ⟦ $n \leq$ *Suc* (*length kvs*); *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧
  $\Longrightarrow$ *rbt-sorted* (*fst* (*rbtreeify-g n kvs*))
⟨*proof*⟩

lemma *rbt-sorted-rbtreeify*:
⟦ *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧ ⟹ *rbt-sorted* (*rbtreeify kvs*)
⟨*proof*⟩

lemma *is-rbt-rbtreeify*:
⟦ *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧
⟹ *is-rbt* (*rbtreeify kvs*)
⟨*proof*⟩

lemma *rbt-lookup-rbtreeify*:
⟦ *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧ ⟹
rbt-lookup (*rbtreeify kvs*) = *map-of kvs*
⟨*proof*⟩

end

Functions to compare the height of two rbt trees, taken from Andrew W. Appel, Efficient Verified Red-Black Trees (September 2011)

fun *skip-red* :: ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt*
where
skip-red (*Branch color.R l k v r*) = *l*
| *skip-red t* = *t*

definition *skip-black* :: ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt*
where
skip-black t = (*let t'* = *skip-red t in case t' of Branch color.B l k v r* ⇒ *l* | - ⇒ *t'*)

datatype *compare* = *LT* | *GT* | *EQ*

partial-function (*tailrec*) *compare-height* :: ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ *compare*
where
compare-height sx s t tx =
(*case* (*skip-red sx*, *skip-red s*, *skip-red t*, *skip-red tx*) *of*
(*Branch* - *sx'* - - -, *Branch* - *s'* - - -, *Branch* - *t'* - - -, *Branch* - *tx'* - - -) ⇒
compare-height (*skip-black sx'*) *s'* *t'* (*skip-black tx'*)
| (-, *rbt.Empty*, -, *Branch* - - - - -) ⇒ *LT*
| (*Branch* - - - - -, -, *rbt.Empty*, -) ⇒ *GT*
| (*Branch* - *sx'* - - -, *Branch* - *s'* - - -, *Branch* - *t'* - - -, *rbt.Empty*) ⇒
compare-height (*skip-black sx'*) *s'* *t'* *rbt.Empty*
| (*rbt.Empty*, *Branch* - *s'* - - -, *Branch* - *t'* - - -, *Branch* - *tx'* - - -) ⇒
compare-height rbt.Empty s' *t'* (*skip-black tx'*)
| - ⇒ *EQ*)

declare *compare-height.simps* [*code*]

hide-type (**open**) *compare*

hide-const (**open**)
compare-height skip-black skip-red LT GT EQ case-compare rec-compare
Abs-compare Rep-compare
hide-fact (**open**)
Abs-compare-cases Abs-compare-induct Abs-compare-inject Abs-compare-inverse
Rep-compare Rep-compare-cases Rep-compare-induct Rep-compare-inject Rep-compare-inverse
compare.simps compare.exhaust compare.induct compare.rec compare.simps
compare.size compare.case-cong compare.case-cong-weak compare.case
compare.nchotomy compare.split compare.split-asm compare.eq.refl compare.eq.simps
equal-compare-def
skip-red.simps skip-red.cases skip-red.induct
skip-black-def
compare-height.simps

110.10 union and intersection of sorted associative lists

context *ord* **begin**

function *sunion-with* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a \times 'b)\ list \Rightarrow ('a \times 'b)\ list \Rightarrow ('a \times 'b)\ list$
where
sunion-with f $((k, v) \# as)$ $((k', v') \# bs) =$
(*if* $k > k'$ *then* $(k', v') \# sunion\text{-}with\ f\ ((k, v) \# as)\ bs$
else if $k < k'$ *then* $(k, v) \# sunion\text{-}with\ f\ as\ ((k', v') \# bs)$
else $(k, f\ k\ v\ v') \# sunion\text{-}with\ f\ as\ bs$)
| *sunion-with f* [] *bs* = *bs*
| *sunion-with f as* [] = *as*
⟨*proof*⟩
termination ⟨*proof*⟩

function *sinter-with* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a \times 'b)\ list \Rightarrow ('a \times 'b)\ list \Rightarrow ('a \times 'b)\ list$
where
sinter-with f $((k, v) \# as)$ $((k', v') \# bs) =$
(*if* $k > k'$ *then sinter-with f* $((k, v) \# as)\ bs$
else if $k < k'$ *then sinter-with f as* $((k', v') \# bs)$
else $(k, f\ k\ v\ v') \# sinter\text{-}with\ f\ as\ bs$)
| *sinter-with f* [] - = []
| *sinter-with f* - [] = []
⟨*proof*⟩
termination ⟨*proof*⟩

end

declare *ord.sunion-with.simps* [*code*] *ord.sinter-with.simps*[*code*]

context *linorder* **begin**

lemma *set-fst-sunion-with*:

set (map fst (sunion-with f xs ys)) = set (map fst xs) ∪ set (map fst ys)
⟨*proof*⟩

lemma *sorted-sunion-with* [*simp*]:
⟦ sorted (map fst xs); sorted (map fst ys) ⟧
⟹ sorted (map fst (sunion-with f xs ys))
⟨*proof*⟩

lemma *distinct-sunion-with* [*simp*]:
⟦ distinct (map fst xs); distinct (map fst ys); sorted (map fst xs); sorted (map fst ys) ⟧
⟹ distinct (map fst (sunion-with f xs ys))
⟨*proof*⟩

lemma *map-of-sunion-with*:
⟦ sorted (map fst xs); sorted (map fst ys) ⟧
⟹ map-of (sunion-with f xs ys) k =
(case map-of xs k of None ⇒ map-of ys k
| Some v ⇒ case map-of ys k of None ⇒ Some v
| Some w ⇒ Some (f k v w))
⟨*proof*⟩

lemma *set-fst-sinter-with* [*simp*]:
⟦ sorted (map fst xs); sorted (map fst ys) ⟧
⟹ set (map fst (sinter-with f xs ys)) = set (map fst xs) ∩ set (map fst ys)
⟨*proof*⟩

lemma *set-fst-sinter-with-subset1*:
set (map fst (sinter-with f xs ys)) ⊆ set (map fst xs)
⟨*proof*⟩

lemma *set-fst-sinter-with-subset2*:
set (map fst (sinter-with f xs ys)) ⊆ set (map fst ys)
⟨*proof*⟩

lemma *sorted-sinter-with* [*simp*]:
⟦ sorted (map fst xs); sorted (map fst ys) ⟧
⟹ sorted (map fst (sinter-with f xs ys))
⟨*proof*⟩

lemma *distinct-sinter-with* [*simp*]:
⟦ distinct (map fst xs); distinct (map fst ys) ⟧
⟹ distinct (map fst (sinter-with f xs ys))
⟨*proof*⟩

lemma *map-of-sinter-with*:
⟦ sorted (map fst xs); sorted (map fst ys) ⟧
⟹ map-of (sinter-with f xs ys) k =
(case map-of xs k of None ⇒ None | Some v ⇒ map-option (f k v) (map-of ys

k))
⟨*proof*⟩

end

lemma *distinct-map-of-rev*: *distinct* (*map fst xs*) ⟹ *map-of* (*rev xs*) = *map-of xs*
⟨*proof*⟩

lemma *map-map-filter*:
map f (*List.map-filter g xs*) = *List.map-filter* (*map-option f* ∘ *g*) *xs*
⟨*proof*⟩

lemma *map-filter-map-option-const*:
List.map-filter (λ*x*. *map-option* (λ*y*. *f x*) (*g* (*f x*))) *xs* = *filter* (λ*x*. *g x* ≠ *None*) (*map f xs*)
⟨*proof*⟩

lemma *set-map-filter*: *set* (*List.map-filter P xs*) = *the* ‘ (*P* ‘ *set xs* − {*None*})
⟨*proof*⟩

context *ord* **begin**

definition *rbt-union-with-key* :: ('*a* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt*
where
rbt-union-with-key f t1 t2 =
(*case RBT-Impl.compare-height t1 t1 t2 t2*
of compare.EQ ⇒ *rbtreeify* (*sunion-with f* (*entries t1*) (*entries t2*))
| *compare.LT* ⇒ *fold* (*rbt-insert-with-key* (λ*k v w*. *f k w v*)) *t1 t2*
| *compare.GT* ⇒ *fold* (*rbt-insert-with-key f*) *t2 t1*)

definition *rbt-union-with* **where**
rbt-union-with f = *rbt-union-with-key* (λ-. *f*)

definition *rbt-union* **where**
rbt-union = *rbt-union-with-key* (%- - *rv*. *rv*)

definition *rbt-inter-with-key* :: ('*a* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt*
where
rbt-inter-with-key f t1 t2 =
(*case RBT-Impl.compare-height t1 t1 t2 t2*
of compare.EQ ⇒ *rbtreeify* (*sinter-with f* (*entries t1*) (*entries t2*))
| *compare.LT* ⇒ *rbtreeify* (*List.map-filter* (λ(*k*, *v*). *map-option* (λ*w*. (*k*, *f k v w*)) (*rbt-lookup t2 k*)) (*entries t1*))
| *compare.GT* ⇒ *rbtreeify* (*List.map-filter* (λ(*k*, *v*). *map-option* (λ*w*. (*k*, *f k w v*)) (*rbt-lookup t1 k*)) (*entries t2*)))

definition *rbt-inter-with* **where**
rbt-inter-with f = rbt-inter-with-key (λ-. f)

definition *rbt-inter* **where**
rbt-inter = rbt-inter-with-key (λ- - rv. rv)

end

context *linorder* **begin**

lemma *rbt-sorted-entries-right-unique*:
⟦ (k, v) ∈ set (entries t); (k, v′) ∈ set (entries t); rbt-sorted t ⟧ ⟹ v = v′
⟨*proof*⟩

lemma *rbt-sorted-fold-rbt-insertwk*:
rbt-sorted t ⟹ rbt-sorted (List.fold (λ(k, v). rbt-insert-with-key f k v) xs t)
⟨*proof*⟩

lemma *is-rbt-fold-rbt-insertwk*:
assumes *is-rbt t1*
shows *is-rbt (fold (rbt-insert-with-key f) t2 t1)*
⟨*proof*⟩

lemma *rbt-lookup-fold-rbt-insertwk*:
assumes *t1*: *rbt-sorted t1* **and** *t2*: *rbt-sorted t2*
shows *rbt-lookup (fold (rbt-insert-with-key f) t1 t2) k =*
(case rbt-lookup t1 k of None ⇒ rbt-lookup t2 k
| Some v ⇒ case rbt-lookup t2 k of None ⇒ Some v
| Some w ⇒ Some (f k w v))
⟨*proof*⟩

lemma *is-rbt-rbt-unionwk* [*simp*]:
⟦ is-rbt t1; is-rbt t2 ⟧ ⟹ is-rbt (rbt-union-with-key f t1 t2)
⟨*proof*⟩

lemma *rbt-lookup-rbt-unionwk*:
⟦ rbt-sorted t1; rbt-sorted t2 ⟧
⟹ rbt-lookup (rbt-union-with-key f t1 t2) k =
(case rbt-lookup t1 k of None ⇒ rbt-lookup t2 k
| Some v ⇒ case rbt-lookup t2 k of None ⇒ Some v
| Some w ⇒ Some (f k v w))
⟨*proof*⟩

lemma *rbt-unionw-is-rbt*: *⟦ is-rbt lt; is-rbt rt ⟧ ⟹ is-rbt (rbt-union-with f lt rt)*
⟨*proof*⟩

lemma *rbt-union-is-rbt*: *⟦ is-rbt lt; is-rbt rt ⟧ ⟹ is-rbt (rbt-union lt rt)*
⟨*proof*⟩

lemma *rbt-lookup-rbt-union*:
⟦ *rbt-sorted s*; *rbt-sorted t* ⟧ ⟹
rbt-lookup (*rbt-union s t*) = *rbt-lookup s* ++ *rbt-lookup t*
⟨*proof*⟩

lemma *rbt-interwk-is-rbt* [*simp*]:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧ ⟹ *is-rbt* (*rbt-inter-with-key f t1 t2*)
⟨*proof*⟩

lemma *rbt-interw-is-rbt*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧ ⟹ *is-rbt* (*rbt-inter-with f t1 t2*)
⟨*proof*⟩

lemma *rbt-inter-is-rbt*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧ ⟹ *is-rbt* (*rbt-inter t1 t2*)
⟨*proof*⟩

lemma *rbt-lookup-rbt-interwk*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧
⟹ *rbt-lookup* (*rbt-inter-with-key f t1 t2*) *k* =
(*case rbt-lookup t1 k of None* ⇒ *None*
| *Some v* ⇒ *case rbt-lookup t2 k of None* ⇒ *None*
| *Some w* ⇒ *Some* (*f k v w*))
⟨*proof*⟩

lemma *rbt-lookup-rbt-inter*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧
⟹ *rbt-lookup* (*rbt-inter t1 t2*) = *rbt-lookup t2* |‘ *dom* (*rbt-lookup t1*)
⟨*proof*⟩

end

110.11 Code generator setup

lemmas [*code*] =
ord.rbt-less-prop
ord.rbt-greater-prop
ord.rbt-sorted.simps
ord.rbt-lookup.simps
ord.is-rbt-def
ord.rbt-ins.simps
ord.rbt-insert-with-key-def
ord.rbt-insertw-def
ord.rbt-insert-def
ord.rbt-del-from-left.simps
ord.rbt-del-from-right.simps
ord.rbt-del.simps
ord.rbt-delete-def

ord.sunion-with.simps
ord.sinter-with.simps
ord.rbt-union-with-key-def
ord.rbt-union-with-def
ord.rbt-union-def
ord.rbt-inter-with-key-def
ord.rbt-inter-with-def
ord.rbt-inter-def
ord.rbt-map-entry.simps
ord.rbt-bulkload-def

More efficient implementations for *entries* and *keys*

definition *gen-entries* ::
(('*a* × '*b*) × ('*a*, '*b*) *rbt*) *list* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a* × '*b*) *list*
where
gen-entries kvts t = *entries t* @ *concat* (*map* (λ(*kv*, *t*). *kv* # *entries t*) *kvts*)

lemma *gen-entries-simps* [*simp*, *code*]:
gen-entries [] *Empty* = []
gen-entries ((*kv*, *t*) # *kvts*) *Empty* = *kv* # *gen-entries kvts t*
gen-entries kvts (*Branch c l k v r*) = *gen-entries* (((*k*, *v*), *r*) # *kvts*) *l*
⟨*proof*⟩

lemma *entries-code* [*code*]:
entries = *gen-entries* []
⟨*proof*⟩

definition *gen-keys* :: ('*a* × ('*a*, '*b*) *rbt*) *list* ⇒ ('*a*, '*b*) *rbt* ⇒ '*a list*
where *gen-keys kts t* = *RBT-Impl.keys t* @ *concat* (*List.map* (λ(*k*, *t*). *k* # *keys t*) *kts*)

lemma *gen-keys-simps* [*simp*, *code*]:
gen-keys [] *Empty* = []
gen-keys ((*k*, *t*) # *kts*) *Empty* = *k* # *gen-keys kts t*
gen-keys kts (*Branch c l k v r*) = *gen-keys* ((*k*, *r*) # *kts*) *l*
⟨*proof*⟩

lemma *keys-code* [*code*]:
keys = *gen-keys* []
⟨*proof*⟩

Restore original type constraints for constants

⟨*ML*⟩

hide-const (**open**) *R B Empty entries keys fold gen-keys gen-entries*

end

111 Abstract type of RBT trees

theory *RBT*
imports *Main RBT-Impl*
begin

111.1 Type definition

typedef (**overloaded**) $('a, 'b)$ *rbt* = $\{t :: ('a::linorder, 'b)$ *RBT-Impl.rbt*. *is-rbt* $t\}$
  morphisms *impl-of RBT*
⟨*proof*⟩

lemma *rbt-eq-iff*:
  t1 = *t2* ⟷ *impl-of t1* = *impl-of t2*
  ⟨*proof*⟩

lemma *rbt-eqI*:
  impl-of t1 = *impl-of t2* ⟹ *t1* = *t2*
  ⟨*proof*⟩

lemma *is-rbt-impl-of* [*simp*, *intro*]:
  is-rbt (*impl-of t*)
  ⟨*proof*⟩

lemma *RBT-impl-of* [*simp*, *code abstype*]:
  RBT (*impl-of t*) = *t*
  ⟨*proof*⟩

111.2 Primitive operations

setup-lifting *type-definition-rbt*

lift-definition *lookup* :: $('a::linorder, 'b)$ *rbt* ⇒ $'a \rightharpoonup 'b$ **is** *rbt-lookup* ⟨*proof*⟩

lift-definition *empty* :: $('a::linorder, 'b)$ *rbt* **is** *RBT-Impl.Empty*
⟨*proof*⟩

lift-definition *insert* :: $'a::linorder \Rightarrow 'b \Rightarrow ('a, 'b)$ *rbt* ⇒ $('a, 'b)$ *rbt* **is** *rbt-insert*

⟨*proof*⟩

lift-definition *delete* :: $'a::linorder \Rightarrow ('a, 'b)$ *rbt* ⇒ $('a, 'b)$ *rbt* **is** *rbt-delete*
⟨*proof*⟩

lift-definition *entries* :: $('a::linorder, 'b)$ *rbt* ⇒ $('a \times 'b)$ *list* **is** *RBT-Impl.entries*
⟨*proof*⟩

lift-definition *keys* :: $('a::linorder, 'b)$ *rbt* ⇒ $'a$ *list* **is** *RBT-Impl.keys* ⟨*proof*⟩

lift-definition *bulkload* :: (*'a*::*linorder* × *'b*) *list* ⇒ (*'a*, *'b*) *rbt* **is** *rbt-bulkload*
⟨*proof*⟩

lift-definition *map-entry* :: *'a* ⇒ (*'b* ⇒ *'b*) ⇒ (*'a*::*linorder*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt*
is *rbt-map-entry*
⟨*proof*⟩

lift-definition *map* :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ (*'a*::*linorder*, *'b*) *rbt* ⇒ (*'a*, *'c*) *rbt* **is**
RBT-Impl.map
⟨*proof*⟩

lift-definition *fold* :: (*'a* ⇒ *'b* ⇒ *'c* ⇒ *'c*) ⇒ (*'a*::*linorder*, *'b*) *rbt* ⇒ *'c* ⇒ *'c* **is**
RBT-Impl.fold ⟨*proof*⟩

lift-definition *union* :: (*'a*::*linorder*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **is**
rbt-union
⟨*proof*⟩

lift-definition *foldi* :: (*'c* ⇒ *bool*) ⇒ (*'a* ⇒ *'b* ⇒ *'c* ⇒ *'c*) ⇒ (*'a* :: *linorder*, *'b*)
rbt ⇒ *'c* ⇒ *'c*
is *RBT-Impl.foldi* ⟨*proof*⟩

lift-definition *combine-with-key* :: (*'a* ⇒ *'b* ⇒ *'b* ⇒ *'b*) ⇒ (*'a*::*linorder*, *'b*) *rbt*
⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt*
is *RBT-Impl.rbt-union-with-key* ⟨*proof*⟩

lift-definition *combine* :: (*'b* ⇒ *'b* ⇒ *'b*) ⇒ (*'a*::*linorder*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt*
⇒ (*'a*, *'b*) *rbt*
is *RBT-Impl.rbt-union-with* ⟨*proof*⟩

111.3 Derived operations

definition *is-empty* :: (*'a*::*linorder*, *'b*) *rbt* ⇒ *bool* **where**
[*code*]: *is-empty t* = (*case impl-of t of RBT-Impl.Empty* ⇒ *True* | - ⇒ *False*)

definition *filter* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ (*'a*::*linorder*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **where**
[*code*]: *filter P t* = *fold* (λ*k v t*. *if P k v then insert k v t else t*) *t empty*

111.4 Abstract lookup properties

lemma *lookup-RBT*:
is-rbt t ⟹ *lookup* (*RBT t*) = *rbt-lookup t*
⟨*proof*⟩

lemma *lookup-impl-of*:
rbt-lookup (*impl-of t*) = *lookup t*
⟨*proof*⟩

lemma *entries-impl-of*:

RBT-Impl.entries (*impl-of t*) = *entries t*
⟨*proof*⟩

lemma *keys-impl-of*:
RBT-Impl.keys (*impl-of t*) = *keys t*
⟨*proof*⟩

lemma *lookup-keys*:
dom (*lookup t*) = *set* (*keys t*)
⟨*proof*⟩

lemma *lookup-empty* [*simp*]:
lookup empty = *Map.empty*
⟨*proof*⟩

lemma *lookup-insert* [*simp*]:
lookup (*insert k v t*) = (*lookup t*)(*k* ↦ *v*)
⟨*proof*⟩

lemma *lookup-delete* [*simp*]:
lookup (*delete k t*) = (*lookup t*)(*k* := *None*)
⟨*proof*⟩

lemma *map-of-entries* [*simp*]:
map-of (*entries t*) = *lookup t*
⟨*proof*⟩

lemma *entries-lookup*:
entries t1 = *entries t2* ⟷ *lookup t1* = *lookup t2*
⟨*proof*⟩

lemma *lookup-bulkload* [*simp*]:
lookup (*bulkload xs*) = *map-of xs*
⟨*proof*⟩

lemma *lookup-map-entry* [*simp*]:
lookup (*map-entry k f t*) = (*lookup t*)(*k* := *map-option f* (*lookup t k*))
⟨*proof*⟩

lemma *lookup-map* [*simp*]:
lookup (*map f t*) *k* = *map-option* (*f k*) (*lookup t k*)
⟨*proof*⟩

lemma *lookup-combine-with-key* [*simp*]:
lookup (*combine-with-key f t1 t2*) *k* = *combine-options* (*f k*) (*lookup t1 k*) (*lookup t2 k*)
⟨*proof*⟩

lemma *combine-altdef*: *combine f t1 t2* = *combine-with-key* (λ-. *f*) *t1 t2*

⟨*proof*⟩

lemma *lookup-combine* [*simp*]:
lookup (*combine f t1 t2*) *k* = *combine-options f* (*lookup t1 k*) (*lookup t2 k*)
⟨*proof*⟩

lemma *fold-fold*:
fold f t = *List.fold* (*case-prod f*) (*entries t*)
⟨*proof*⟩

lemma *impl-of-empty*:
impl-of empty = *RBT-Impl.Empty*
⟨*proof*⟩

lemma *is-empty-empty* [*simp*]:
is-empty t ⟷ *t* = *empty*
⟨*proof*⟩

lemma *RBT-lookup-empty* [*simp*]:
rbt-lookup t = *Map.empty* ⟷ *t* = *RBT-Impl.Empty*
⟨*proof*⟩

lemma *lookup-empty-empty* [*simp*]:
lookup t = *Map.empty* ⟷ *t* = *empty*
⟨*proof*⟩

lemma *sorted-keys* [*iff*]:
sorted (*keys t*)
⟨*proof*⟩

lemma *distinct-keys* [*iff*]:
distinct (*keys t*)
⟨*proof*⟩

lemma *finite-dom-lookup* [*simp*, *intro*!]: *finite* (*dom* (*lookup t*))
⟨*proof*⟩

lemma *lookup-union*: *lookup* (*union s t*) = *lookup s* ++ *lookup t*
⟨*proof*⟩

lemma *lookup-in-tree*: (*lookup t k* = *Some v*) = ((*k*, *v*) ∈ *set* (*entries t*))
⟨*proof*⟩

lemma *keys-entries*: (*k* ∈ *set* (*keys t*)) = (∃ *v*. (*k*, *v*) ∈ *set* (*entries t*))
⟨*proof*⟩

lemma *fold-def-alt*:
fold f t = *List.fold* (*case-prod f*) (*entries t*)
⟨*proof*⟩

lemma *distinct-entries*: *distinct* (*List.map fst* (*entries t*))
⟨*proof*⟩

lemma *non-empty-keys*: $t \neq empty \Longrightarrow keys\ t \neq []$
⟨*proof*⟩

lemma *keys-def-alt*:
keys t = *List.map fst* (*entries t*)
⟨*proof*⟩

context
begin

private lemma *lookup-filter-aux*:
assumes *distinct* (*List.map fst xs*)
shows *lookup* (*List.fold* (λ(*k*, *v*) *t*. *if P k v then insert k v t else t*) *xs t*) *k* =
(*case map-of xs k of*
None ⇒ *lookup t k*
| *Some v* ⇒ *if P k v then Some v else lookup t k*)
⟨*proof*⟩

lemma *lookup-filter*:
lookup (*filter P t*) *k* =
(*case lookup t k of None* ⇒ *None* | *Some v* ⇒ *if P k v then Some v else None*)
⟨*proof*⟩

end

111.5 Quickcheck generators

quickcheck-generator *rbt predicate*: *is-rbt constructors*: *empty*, *insert*

111.6 Hide implementation details

lifting-update *rbt.lifting*
lifting-forget *rbt.lifting*

hide-const (**open**) *impl-of empty lookup keys entries bulkload delete map fold union insert map-entry foldi*
is-empty filter
hide-fact (**open**) *empty-def lookup-def keys-def entries-def bulkload-def delete-def map-def fold-def*
union-def insert-def map-entry-def foldi-def is-empty-def filter-def

end

112 Implementation of mappings with Red-Black Trees

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*p*

This theory defines abstract red-black trees as an efficient representation of finite maps, backed by the implementation in *RBT-Impl*.

112.1 Data type and invariant

The type $('k, 'v)$ *RBT-Impl.rbt* denotes red-black trees with keys of type $'k$ and values of type $'v$. To function properly, the key type musorted belong to the *linorder* class.

A value t of this type is a valid red-black tree if it satisfies the invariant *is-rbt t*. The abstract type $('k, 'v)$ *RBT.rbt* always obeys this invariant, and for this reason you should only use this in our application. Going back to $('k, 'v)$ *RBT-Impl.rbt* may be necessary in proofs if not yet proven properties about the operations must be established.

The interpretation function *RBT.lookup* returns the partial map represented by a red-black tree:

RBT.lookup::$('a, 'b)$ *RBT.rbt* $\Rightarrow$ $'a \Rightarrow$ $'b$ *option*

This function should be used for reasoning about the semantics of the RBT operations. Furthermore, it implements the lookup functionality for the data structure: It is executable and the lookup is performed in $O(\log n)$.

112.2 Operations

Currently, the following operations are supported:

RBT.empty::$('a, 'b)$ *RBT.rbt*

Returns the empty tree. $O(1)$

RBT.insert::$'a \Rightarrow$ $'b \Rightarrow$ $('a, 'b)$ *RBT.rbt* $\Rightarrow$ $('a, 'b)$ *RBT.rbt*

Updates the map at a given position. $O(\log n)$

RBT.delete::$'a \Rightarrow$ $('a, 'b)$ *RBT.rbt* $\Rightarrow$ $('a, 'b)$ *RBT.rbt*

Deletes a map entry at a given position. $O(\log n)$

RBT.entries::$('a, 'b)$ *RBT.rbt* $\Rightarrow$ $('a \times 'b)$ *list*

Return a corresponding key-value list for a tree.

RBT.bulkload::$('a \times 'b)$ *list* $\Rightarrow$ $('a, 'b)$ *RBT.rbt*

Builds a tree from a key-value list.

RBT.map-entry::$'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b)$ *RBT.rbt* $\Rightarrow ('a, 'b)$ *RBT.rbt*

Maps a single entry in a tree.

RBT.map::$('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'b)$ *RBT.rbt* $\Rightarrow ('a, 'c)$ *RBT.rbt*

Maps all values in a tree. $O(n)$

RBT.fold::$('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a, 'b)$ *RBT.rbt* $\Rightarrow 'c \Rightarrow 'c$

Folds over all entries in a tree. $O(n)$

112.3 Invariant preservation

is-rbt rbt.Empty (*Empty-is-rbt*)
is-rbt ?t $\Longrightarrow$ *is-rbt* (*rbt-insert ?k ?v ?t*) (*rbt-insert-is-rbt*)
is-rbt ?t $\Longrightarrow$ *is-rbt* (*rbt-delete ?k ?t*) (*delete-is-rbt*)
is-rbt (*rbt-bulkload ?xs*) (*bulkload-is-rbt*)
is-rbt (*rbt-map-entry ?k ?f ?t*) = *is-rbt ?t* (*map-entry-is-rbt*)
is-rbt (*RBT-Impl.map ?f ?t*) = *is-rbt ?t* (*map-is-rbt*)
$[\![$*is-rbt ?lt*; *is-rbt ?rt*$]\!] \Longrightarrow$ *is-rbt* (*rbt-union ?lt ?rt*) (*union-is-rbt*)

112.4 Map Semantics

lookup-empty

Mapping.lookup Mapping.empty ?k = None

lookup-insert

RBT.lookup (*RBT.insert ?k ?v ?t*) = *RBT.lookup ?t*(*?k* $\mapsto$ *?v*)

lookup-delete

RBT.lookup (*RBT.delete ?k ?t*) = (*RBT.lookup ?t*)(*?k* := *None*)

lookup-bulkload

RBT.lookup (*RBT.bulkload ?xs*) = *map-of ?xs*

lookup-map

RBT.lookup (*RBT.map ?f ?t*) *?k* = *map-option* (*?f ?k*) (*RBT.lookup ?t ?k*)

end

113 Implementation of sets using RBT trees

theory *RBT-Set*
imports *RBT Product-Lexorder*
begin

114 Definition of code datatype constructors

definition *Set* :: (*'a*::*linorder*, *unit*) *rbt* ⇒ *'a set*
  where *Set t* = {*x* . *RBT.lookup t x* = *Some* ()}

definition *Coset* :: (*'a*::*linorder*, *unit*) *rbt* ⇒ *'a set*
  where [*simp*]: *Coset t* = − *Set t*

115 Deletion of already existing code equations

declare [[*code drop*: *Set.empty Set.is-empty uminus-set-inst.uminus-set*
  Set.member Set.insert Set.remove UNIV Set.filter image
  Set.subset-eq Ball Bex can-select Set.union minus-set-inst.minus-set Set.inter
  card the-elem Pow sum prod Product-Type.product Id-on
  Image trancl relcomp wf Min Inf-fin Max Sup-fin
  (*Inf* :: *'a set set* ⇒ *'a set*) (*Sup* :: *'a set set* ⇒ *'a set*)
  sorted-list-of-set List.map-project List.Bleast]]

116 Lemmas

116.1 Auxiliary lemmas

lemma [*simp*]: *x* ≠ *Some* () ⟷ *x* = *None*
⟨*proof*⟩

lemma *Set-set-keys*: *Set x* = *dom* (*RBT.lookup x*)
⟨*proof*⟩

lemma *finite-Set* [*simp*, *intro*!]: *finite* (*Set x*)
⟨*proof*⟩

lemma *set-keys*: *Set t* = *set*(*RBT.keys t*)
⟨*proof*⟩

116.2 fold and filter

lemma *finite-fold-rbt-fold-eq*:
  assumes *comp-fun-commute f*
  shows *Finite-Set.fold f A* (*set* (*RBT.entries t*)) = *RBT.fold* (*curry f*) *t A*
⟨*proof*⟩

definition *fold-keys* :: (*'a* :: *linorder* ⇒ *'b* ⇒ *'b*) ⇒ (*'a*, -) *rbt* ⇒ *'b* ⇒ *'b*

where [*code-unfold*]:*fold-keys f t A* = *RBT.fold* (λ*k* - *t*. *f k t*) *t A*

lemma *fold-keys-def-alt*:
fold-keys f t s = *List.fold f* (*RBT.keys t*) *s*
⟨*proof*⟩

lemma *finite-fold-fold-keys*:
assumes *comp-fun-commute f*
shows *Finite-Set.fold f A* (*Set t*) = *fold-keys f t A*
⟨*proof*⟩

definition *rbt-filter* :: ('*a* :: *linorder* ⇒ *bool*) ⇒ ('*a*, '*b*) *rbt* ⇒ '*a set* **where**
rbt-filter P t = *RBT.fold* (λ*k* - *A'*. *if P k then Set.insert k A' else A'*) *t* {}

lemma *Set-filter-rbt-filter*:
Set.filter P (*Set t*) = *rbt-filter P t*
⟨*proof*⟩

116.3 foldi and Ball

lemma *Ball-False*: *RBT-Impl.fold* (λ*k v s*. *s* ∧ *P k*) *t False* = *False*
⟨*proof*⟩

lemma *rbt-foldi-fold-conj*:
RBT-Impl.foldi (λ*s*. *s* = *True*) (λ*k v s*. *s* ∧ *P k*) *t val* = *RBT-Impl.fold* (λ*k v s*. *s* ∧ *P k*) *t val*
⟨*proof*⟩

lemma *foldi-fold-conj*: *RBT.foldi* (λ*s*. *s* = *True*) (λ*k v s*. *s* ∧ *P k*) *t val* = *fold-keys* (λ*k s*. *s* ∧ *P k*) *t val*
⟨*proof*⟩ **including** *rbt.lifting* ⟨*proof*⟩

116.4 foldi and Bex

lemma *Bex-True*: *RBT-Impl.fold* (λ*k v s*. *s* ∨ *P k*) *t True* = *True*
⟨*proof*⟩

lemma *rbt-foldi-fold-disj*:
RBT-Impl.foldi (λ*s*. *s* = *False*) (λ*k v s*. *s* ∨ *P k*) *t val* = *RBT-Impl.fold* (λ*k v s*. *s* ∨ *P k*) *t val*
⟨*proof*⟩

lemma *foldi-fold-disj*: *RBT.foldi* (λ*s*. *s* = *False*) (λ*k v s*. *s* ∨ *P k*) *t val* = *fold-keys* (λ*k s*. *s* ∨ *P k*) *t val*
⟨*proof*⟩ **including** *rbt.lifting* ⟨*proof*⟩

116.5 folding over non empty trees and selecting the minimal and maximal element

definition *rbt-fold1-keys* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('a{::}linorder, 'b)$ *RBT-Impl.rbt* $\Rightarrow 'a$
where *rbt-fold1-keys f t* = *List.fold f* (*tl*(*RBT-Impl.keys t*)) (*hd*(*RBT-Impl.keys t*))

definition *rbt-min* :: $('a{::}linorder, unit)$ *RBT-Impl.rbt* $\Rightarrow 'a$
where *rbt-min t* = *rbt-fold1-keys min t*

lemma *key-le-right*: *rbt-sorted* (*Branch c lt k v rt*) $\Longrightarrow$ ($\bigwedge x.\ x \in$*set* (*RBT-Impl.keys rt*) $\Longrightarrow$ $k \leq x$)
⟨*proof*⟩

lemma *left-le-key*: *rbt-sorted* (*Branch c lt k v rt*) $\Longrightarrow$ ($\bigwedge x.\ x \in$*set* (*RBT-Impl.keys lt*) $\Longrightarrow$ $x \leq k$)
⟨*proof*⟩

lemma *fold-min-triv*:
fixes k :: - :: *linorder*
shows ($\forall x \in$*set xs*. $k \leq x$) $\Longrightarrow$ *List.fold min xs k* = k
⟨*proof*⟩

lemma *rbt-min-simps*:
is-rbt (*Branch c RBT-Impl.Empty k v rt*) $\Longrightarrow$ *rbt-min* (*Branch c RBT-Impl.Empty k v rt*) = k
⟨*proof*⟩

fun *rbt-min-opt* **where**
rbt-min-opt (*Branch c RBT-Impl.Empty k v rt*) = k |
rbt-min-opt (*Branch c* (*Branch lc llc lk lv lrt*) *k v rt*) = *rbt-min-opt* (*Branch lc llc lk lv lrt*)

lemma *rbt-min-opt-Branch*:
t1 $\neq$ *rbt.Empty* $\Longrightarrow$ *rbt-min-opt* (*Branch c t1 k* () *t2*) = *rbt-min-opt t1*
⟨*proof*⟩

lemma *rbt-min-opt-induct* [*case-names empty left-empty left-non-empty*]:
fixes t :: ($'a$ :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *P rbt.Empty*
assumes $\bigwedge$*color t1 a b t2*. *P t1* $\Longrightarrow$ *P t2* $\Longrightarrow$ *t1* = *rbt.Empty* $\Longrightarrow$ *P* (*Branch color t1 a b t2*)
assumes $\bigwedge$*color t1 a b t2*. *P t1* $\Longrightarrow$ *P t2* $\Longrightarrow$ *t1* $\neq$ *rbt.Empty* $\Longrightarrow$ *P* (*Branch color t1 a b t2*)
shows *P t*
⟨*proof*⟩

lemma *rbt-min-opt-in-set*:
fixes $t :: ('a :: linorder, unit)\ RBT\text{-}Impl.rbt$
assumes $t \neq rbt.Empty$
shows $rbt\text{-}min\text{-}opt\ t \in set\ (RBT\text{-}Impl.keys\ t)$
⟨*proof*⟩

lemma *rbt-min-opt-is-min*:
fixes $t :: ('a :: linorder, unit)\ RBT\text{-}Impl.rbt$
assumes *rbt-sorted t*
assumes $t \neq rbt.Empty$
shows $\bigwedge y.\ y \in set\ (RBT\text{-}Impl.keys\ t) \Longrightarrow y \geq rbt\text{-}min\text{-}opt\ t$
⟨*proof*⟩

lemma *rbt-min-eq-rbt-min-opt*:
assumes $t \neq RBT\text{-}Impl.Empty$
assumes *is-rbt t*
shows $rbt\text{-}min\ t = rbt\text{-}min\text{-}opt\ t$
⟨*proof*⟩

definition $rbt\text{-}max :: ('a::linorder, unit)\ RBT\text{-}Impl.rbt \Rightarrow 'a$
where $rbt\text{-}max\ t = rbt\text{-}fold1\text{-}keys\ max\ t$

lemma *fold-max-triv*:
fixes $k :: - :: linorder$
shows $(\forall x \in set\ xs.\ x \leq k) \Longrightarrow List.fold\ max\ xs\ k = k$
⟨*proof*⟩

lemma *fold-max-rev-eq*:
fixes $xs :: ('a :: linorder)\ list$
assumes $xs \neq []$
shows $List.fold\ max\ (tl\ xs)\ (hd\ xs) = List.fold\ max\ (tl\ (rev\ xs))\ (hd\ (rev\ xs))$
⟨*proof*⟩

lemma *rbt-max-simps*:
assumes $is\text{-}rbt\ (Branch\ c\ lt\ k\ v\ RBT\text{-}Impl.Empty)$
shows $rbt\text{-}max\ (Branch\ c\ lt\ k\ v\ RBT\text{-}Impl.Empty) = k$
⟨*proof*⟩

fun *rbt-max-opt* **where**
$rbt\text{-}max\text{-}opt\ (Branch\ c\ lt\ k\ v\ RBT\text{-}Impl.Empty) = k\ |$
$rbt\text{-}max\text{-}opt\ (Branch\ c\ lt\ k\ v\ (Branch\ rc\ rlc\ rk\ rv\ rrt)) = rbt\text{-}max\text{-}opt\ (Branch\ rc\ rlc\ rk\ rv\ rrt)$

lemma *rbt-max-opt-Branch*:
$t2 \neq rbt.Empty \Longrightarrow rbt\text{-}max\text{-}opt\ (Branch\ c\ t1\ k\ ()\ t2) = rbt\text{-}max\text{-}opt\ t2$
⟨*proof*⟩

lemma *rbt-max-opt-induct* [*case-names empty right-empty right-non-empty*]:
fixes *t* :: (*'a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *P rbt.Empty*
assumes $\bigwedge$*color t1 a b t2*. *P t1* $\Longrightarrow$ *P t2* $\Longrightarrow$ *t2* = *rbt.Empty* $\Longrightarrow$ *P* (*Branch color t1 a b t2*)
assumes $\bigwedge$*color t1 a b t2*. *P t1* $\Longrightarrow$ *P t2* $\Longrightarrow$ *t2* $\neq$ *rbt.Empty* $\Longrightarrow$ *P* (*Branch color t1 a b t2*)
shows *P t*
⟨*proof*⟩

lemma *rbt-max-opt-in-set*:
fixes *t* :: (*'a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *t* $\neq$ *rbt.Empty*
shows *rbt-max-opt t* $\in$ *set* (*RBT-Impl.keys t*)
⟨*proof*⟩

lemma *rbt-max-opt-is-max*:
fixes *t* :: (*'a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *rbt-sorted t*
assumes *t* $\neq$ *rbt.Empty*
shows $\bigwedge$*y*. *y* $\in$ *set* (*RBT-Impl.keys t*) $\Longrightarrow$ *y* $\leq$ *rbt-max-opt t*
⟨*proof*⟩

lemma *rbt-max-eq-rbt-max-opt*:
assumes *t* $\neq$ *RBT-Impl.Empty*
assumes *is-rbt t*
shows *rbt-max t* = *rbt-max-opt t*
⟨*proof*⟩

context includes *rbt.lifting* **begin**
lift-definition *fold1-keys* :: (*'a* $\Rightarrow$ *'a* $\Rightarrow$ *'a*) $\Rightarrow$ (*'a*::*linorder*, *'b*) *rbt* $\Rightarrow$ *'a*
is *rbt-fold1-keys* ⟨*proof*⟩

lemma *fold1-keys-def-alt*:
fold1-keys f t = *List.fold f* (*tl* (*RBT.keys t*)) (*hd* (*RBT.keys t*))
⟨*proof*⟩

lemma *finite-fold1-fold1-keys*:
assumes *semilattice f*
assumes ¬ *RBT.is-empty t*
shows *semilattice-set.F f* (*Set t*) = *fold1-keys f t*
⟨*proof*⟩

lift-definition *r-min* :: (*'a* :: *linorder*, *unit*) *rbt* $\Rightarrow$ *'a* **is** *rbt-min* ⟨*proof*⟩

lift-definition *r-min-opt* :: (*'a* :: *linorder*, *unit*) *rbt* ⇒ *'a* **is** *rbt-min-opt* ⟨*proof*⟩

lemma *r-min-alt-def*: *r-min t* = *fold1-keys min t*
⟨*proof*⟩

lemma *r-min-eq-r-min-opt*:
assumes ¬ (*RBT.is-empty t*)
shows *r-min t* = *r-min-opt t*
⟨*proof*⟩

lemma *fold-keys-min-top-eq*:
fixes *t* :: (*'a*::{*linorder*,*bounded-lattice-top*}, *unit*) *rbt*
assumes ¬ (*RBT.is-empty t*)
shows *fold-keys min t top* = *fold1-keys min t*
⟨*proof*⟩

lift-definition *r-max* :: (*'a* :: *linorder*, *unit*) *rbt* ⇒ *'a* **is** *rbt-max* ⟨*proof*⟩

lift-definition *r-max-opt* :: (*'a* :: *linorder*, *unit*) *rbt* ⇒ *'a* **is** *rbt-max-opt* ⟨*proof*⟩

lemma *r-max-alt-def*: *r-max t* = *fold1-keys max t*
⟨*proof*⟩

lemma *r-max-eq-r-max-opt*:
assumes ¬ (*RBT.is-empty t*)
shows *r-max t* = *r-max-opt t*
⟨*proof*⟩

lemma *fold-keys-max-bot-eq*:
fixes *t* :: (*'a*::{*linorder*,*bounded-lattice-bot*}, *unit*) *rbt*
assumes ¬ (*RBT.is-empty t*)
shows *fold-keys max t bot* = *fold1-keys max t*
⟨*proof*⟩

end

117 Code equations

code-datatype *Set Coset*

declare *list.set*[*code*]

lemma *empty-Set* [*code*]:
Set.empty = *Set RBT.empty*
⟨*proof*⟩

lemma *UNIV-Coset* [*code*]:
UNIV = *Coset RBT.empty*
⟨*proof*⟩

lemma *is-empty-Set* [*code*]:
Set.is-empty (*Set t*) = *RBT.is-empty t*
⟨*proof*⟩

lemma *compl-code* [*code*]:
− *Set xs* = *Coset xs*
− *Coset xs* = *Set xs*
⟨*proof*⟩

lemma *member-code* [*code*]:
x ∈ (*Set t*) = (*RBT.lookup t x* = *Some* ())
x ∈ (*Coset t*) = (*RBT.lookup t x* = *None*)
⟨*proof*⟩

lemma *insert-code* [*code*]:
Set.insert x (*Set t*) = *Set* (*RBT.insert x* () *t*)
Set.insert x (*Coset t*) = *Coset* (*RBT.delete x t*)
⟨*proof*⟩

lemma *remove-code* [*code*]:
Set.remove x (*Set t*) = *Set* (*RBT.delete x t*)
Set.remove x (*Coset t*) = *Coset* (*RBT.insert x* () *t*)
⟨*proof*⟩

lemma *union-Set* [*code*]:
Set t ∪ *A* = *fold-keys Set.insert t A*
⟨*proof*⟩

lemma *inter-Set* [*code*]:
A ∩ *Set t* = *rbt-filter* (λ*k*. *k* ∈ *A*) *t*
⟨*proof*⟩

lemma *minus-Set* [*code*]:
A − *Set t* = *fold-keys Set.remove t A*
⟨*proof*⟩

lemma *union-Coset* [*code*]:
Coset t ∪ *A* = − *rbt-filter* (λ*k*. *k* ∉ *A*) *t*
⟨*proof*⟩

lemma *union-Set-Set* [*code*]:
Set t1 ∪ *Set t2* = *Set* (*RBT.union t1 t2*)
⟨*proof*⟩

lemma *inter-Coset* [*code*]:

$A \cap Coset\ t = fold\text{-}keys\ Set.remove\ t\ A$
⟨*proof*⟩

lemma *inter-Coset-Coset* [*code*]:
$Coset\ t1 \cap Coset\ t2 = Coset\ (RBT.union\ t1\ t2)$
⟨*proof*⟩

lemma *minus-Coset* [*code*]:
$A - Coset\ t = rbt\text{-}filter\ (\lambda k.\ k \in A)\ t$
⟨*proof*⟩

lemma *filter-Set* [*code*]:
$Set.filter\ P\ (Set\ t) = (rbt\text{-}filter\ P\ t)$
⟨*proof*⟩

lemma *image-Set* [*code*]:
$image\ f\ (Set\ t) = fold\text{-}keys\ (\lambda k\ A.\ Set.insert\ (f\ k)\ A)\ t\ \{\}$
⟨*proof*⟩

lemma *Ball-Set* [*code*]:
$Ball\ (Set\ t)\ P \longleftrightarrow RBT.foldi\ (\lambda s.\ s = True)\ (\lambda k\ v\ s.\ s \wedge P\ k)\ t\ True$
⟨*proof*⟩

lemma *Bex-Set* [*code*]:
$Bex\ (Set\ t)\ P \longleftrightarrow RBT.foldi\ (\lambda s.\ s = False)\ (\lambda k\ v\ s.\ s \vee P\ k)\ t\ False$
⟨*proof*⟩

lemma *subset-code* [*code*]:
$Set\ t \leq B \longleftrightarrow (\forall\, x{\in}Set\ t.\ x \in B)$
$A \leq Coset\ t \longleftrightarrow (\forall\, y{\in}Set\ t.\ y \notin A)$
⟨*proof*⟩

lemma *subset-Coset-empty-Set-empty* [*code*]:
$Coset\ t1 \leq Set\ t2 \longleftrightarrow (case\ (RBT.impl\text{-}of\ t1,\ RBT.impl\text{-}of\ t2)\ of$
$(rbt.Empty,\ rbt.Empty) \Rightarrow False\ |$
$(\text{-},\ \text{-}) \Rightarrow Code.abort\ (STR\ ''non\text{-}empty\text{-}trees'')\ (\lambda\text{-}.\ Coset\ t1 \leq Set\ t2))$
⟨*proof*⟩

A frequent case – avoid intermediate sets

lemma [*code-unfold*]:
$Set\ t1 \subseteq Set\ t2 \longleftrightarrow RBT.foldi\ (\lambda s.\ s = True)\ (\lambda k\ v\ s.\ s \wedge k \in Set\ t2)\ t1\ True$
⟨*proof*⟩

lemma *card-Set* [*code*]:
$card\ (Set\ t) = fold\text{-}keys\ (\lambda\text{-}\ n.\ n + 1)\ t\ 0$
⟨*proof*⟩

lemma *sum-Set* [*code*]:
$sum\ f\ (Set\ xs) = fold\text{-}keys\ (plus\ o\ f)\ xs\ 0$

⟨*proof*⟩

lemma *the-elem-set* [*code*]:
fixes *t* :: ('*a* :: *linorder*, *unit*) *rbt*
shows *the-elem* (*Set t*) = (*case RBT.impl-of t of*
(*Branch RBT-Impl.B RBT-Impl.Empty x* () *RBT-Impl.Empty*) ⇒ *x*
| - ⇒ *Code.abort* (*STR* ''*not-a-singleton-tree*'') (λ-. *the-elem* (*Set t*)))
⟨*proof*⟩

lemma *Pow-Set* [*code*]: *Pow* (*Set t*) = *fold-keys* (λ*x A*. *A* ∪ *Set.insert x* ' *A*) *t* {{}}
⟨*proof*⟩

lemma *product-Set* [*code*]:
Product-Type.product (*Set t1*) (*Set t2*) =
fold-keys (λ*x A*. *fold-keys* (λ*y*. *Set.insert* (*x*, *y*)) *t2 A*) *t1* {}
⟨*proof*⟩

lemma *Id-on-Set* [*code*]: *Id-on* (*Set t*) = *fold-keys* (λ*x*. *Set.insert* (*x*, *x*)) *t* {}
⟨*proof*⟩

lemma *Image-Set* [*code*]:
(*Set t*) `` *S* = *fold-keys* (λ(*x*,*y*) *A*. *if x* ∈ *S then Set.insert y A else A*) *t* {}
⟨*proof*⟩

lemma *trancl-set-ntrancl* [*code*]:
trancl (*Set t*) = *ntrancl* (*card* (*Set t*) − *1*) (*Set t*)
⟨*proof*⟩

lemma *relcomp-Set*[*code*]:
(*Set t1*) *O* (*Set t2*) = *fold-keys*
(λ(*x*,*y*) *A*. *fold-keys* (λ(*w*,*z*) *A'*. *if y* = *w then Set.insert* (*x*,*z*) *A'* *else A'*) *t2 A*) *t1* {}
⟨*proof*⟩

lemma *wf-set* [*code*]:
wf (*Set t*) = *acyclic* (*Set t*)
⟨*proof*⟩

lemma *Min-fin-set-fold* [*code*]:
Min (*Set t*) =
(*if RBT.is-empty t*
then Code.abort (*STR* ''*not-non-empty-tree*'') (λ-. *Min* (*Set t*))
else r-min-opt t)
⟨*proof*⟩

lemma *Inf-fin-set-fold* [*code*]:
Inf-fin (*Set t*) = *Min* (*Set t*)
⟨*proof*⟩

lemma *Inf-Set-fold*:
fixes $t :: ('a :: \{linorder, complete\text{-}lattice\}, unit)\ rbt$
shows $Inf\ (Set\ t) = (if\ RBT.is\text{-}empty\ t\ then\ top\ else\ r\text{-}min\text{-}opt\ t)$
⟨*proof*⟩

lemma *Max-fin-set-fold* [*code*]:
$Max\ (Set\ t) =$
$(if\ RBT.is\text{-}empty\ t$
$\ then\ Code.abort\ (STR\ ''not\text{-}non\text{-}empty\text{-}tree'')\ (\lambda\text{-}.\ Max\ (Set\ t))$
$\ else\ r\text{-}max\text{-}opt\ t)$
⟨*proof*⟩

lemma *Sup-fin-set-fold* [*code*]:
$Sup\text{-}fin\ (Set\ t) = Max\ (Set\ t)$
⟨*proof*⟩

lemma *Sup-Set-fold*:
fixes $t :: ('a :: \{linorder, complete\text{-}lattice\}, unit)\ rbt$
shows $Sup\ (Set\ t) = (if\ RBT.is\text{-}empty\ t\ then\ bot\ else\ r\text{-}max\text{-}opt\ t)$
⟨*proof*⟩

context
begin

qualified definition $Inf' :: 'a :: \{linorder, complete\text{-}lattice\}\ set \Rightarrow 'a$
where [*code-abbrev*]: $Inf' = Inf$

lemma *Inf'-Set-fold* [*code*]:
$Inf'\ (Set\ t) = (if\ RBT.is\text{-}empty\ t\ then\ top\ else\ r\text{-}min\text{-}opt\ t)$
⟨*proof*⟩ **definition** $Sup' :: 'a :: \{linorder, complete\text{-}lattice\}\ set \Rightarrow 'a$
where [*code-abbrev*]: $Sup' = Sup$

lemma *Sup'-Set-fold* [*code*]:
$Sup'\ (Set\ t) = (if\ RBT.is\text{-}empty\ t\ then\ bot\ else\ r\text{-}max\text{-}opt\ t)$
⟨*proof*⟩

lemma [*code drop*: *Gcd-fin*, *code*]:
$Gcd_{fin}\ (Set\ t) = fold\text{-}keys\ gcd\ t\ (0::'a::\{semiring\text{-}gcd, linorder\})$
⟨*proof*⟩

lemma [*code drop*: $Gcd :: - \Rightarrow nat$, *code*]:
$Gcd\ (Set\ t) = (Gcd_{fin}\ (Set\ t) :: nat)$
⟨*proof*⟩

lemma [*code drop*: $Gcd :: - \Rightarrow int$, *code*]:
$Gcd\ (Set\ t) = (Gcd_{fin}\ (Set\ t) :: int)$
⟨*proof*⟩

lemma [*code drop*: *Lcm-fin*,*code*]:
Lcm$_{fin}$ (*Set t*) = *fold-keys lcm t* (*1*::*'a*::{*semiring-gcd*, *linorder*})
⟨*proof*⟩ **definition** *Lcm'* :: *'a* :: *semiring-Gcd set* ⇒ *'a*
where [*code-abbrev*]: *Lcm'* = *Lcm*

lemma [*code drop*: *Lcm* :: - ⇒ *nat*, *code*]:
Lcm (*Set t*) = (*Lcm*$_{fin}$ (*Set t*) :: *nat*)
⟨*proof*⟩

lemma [*code drop*: *Lcm* :: - ⇒ *int*, *code*]:
Lcm (*Set t*) = (*Lcm*$_{fin}$ (*Set t*) :: *int*)
⟨*proof*⟩

end

lemma *sorted-list-set*[*code*]: *sorted-list-of-set* (*Set t*) = *RBT.keys t*
⟨*proof*⟩

lemma *Bleast-code* [*code*]:
Bleast (*Set t*) *P* =
(*case List.filter P* (*RBT.keys t*) *of*
x # *xs* ⇒ *x*
| [] ⇒ *abort-Bleast* (*Set t*) *P*)
⟨*proof*⟩

hide-const (**open**) *RBT-Set.Set RBT-Set.Coset*

end

⟨*ML*⟩

theory *Predicate-Compile-Alternative-Defs*
imports *Main*
begin

118 Common constants

declare *HOL.if-bool-eq-disj*[*code-pred-inline*]

declare *bool-diff-def*[*code-pred-inline*]
declare *inf-bool-def*[*abs-def*, *code-pred-inline*]
declare *less-bool-def*[*abs-def*, *code-pred-inline*]
declare *le-bool-def*[*abs-def*, *code-pred-inline*]

lemma *min-bool-eq* [*code-pred-inline*]: (*min* :: *bool* => *bool* => *bool*) == (*op* &)
⟨*proof*⟩

lemma [*code-pred-inline*]:

((A::bool) ~= (B::bool)) = ((A & ~ B) | (B & ~ A))
⟨proof⟩

⟨ML⟩

119 Pairs

⟨ML⟩

120 Filters

⟨ML⟩

121 Bounded quantifiers

declare *Ball-def*[*code-pred-inline*]
declare *Bex-def*[*code-pred-inline*]

122 Operations on Predicates

lemma *Diff*[*code-pred-inline*]:
(A − B) = (%x. A x ∧ ¬ B x)
⟨proof⟩

lemma *subset-eq*[*code-pred-inline*]:
(P :: 'a => bool) < (Q :: 'a => bool) == ((∃x. Q x ∧ (¬ P x)) ∧ (∀ x. P x −−> Q x))
⟨proof⟩

lemma *set-equality*[*code-pred-inline*]:
A = B ⟷ (∀x. A x ⟶ B x) ∧ (∀x. B x ⟶ A x)
⟨proof⟩

123 Setup for Numerals

⟨ML⟩

124 Arithmetic operations

124.1 Arithmetic on naturals and integers

definition *plus-eq-nat* :: nat => nat => nat => bool
where
plus-eq-nat x y z = (x + y = z)

definition *minus-eq-nat* :: nat => nat => nat => bool

where
minus-eq-nat x y z = (*x* − *y* = *z*)

definition *plus-eq-int* :: *int* => *int* => *int* => *bool*
where
plus-eq-int x y z = (*x* + *y* = *z*)

definition *minus-eq-int* :: *int* => *int* => *int* => *bool*
where
minus-eq-int x y z = (*x* − *y* = *z*)

definition *subtract*
where
[*code-unfold*]: *subtract x y* = *y* − *x*

⟨*ML*⟩

124.2 Inductive definitions for ordering on naturals

inductive *less-nat*
where
less-nat 0 (*Suc y*)
| *less-nat x y* ==> *less-nat* (*Suc x*) (*Suc y*)

lemma *less-nat*[*code-pred-inline*]:
x < *y* = *less-nat x y*
⟨*proof*⟩

inductive *less-eq-nat*
where
less-eq-nat 0 y
| *less-eq-nat x y* ==> *less-eq-nat* (*Suc x*) (*Suc y*)

lemma [*code-pred-inline*]:
x <= *y* = *less-eq-nat x y*
⟨*proof*⟩

125 Alternative list definitions

125.1 Alternative rules for *length*

definition *size-list'* :: *'a list* => *nat*
where *size-list'* = *size*

lemma *size-list'-simps*:
size-list' [] = *0*
size-list' (*x* # *xs*) = *Suc* (*size-list' xs*)
⟨*proof*⟩

declare *size-list'-simps*[*code-pred-def*]
declare *size-list'-def*[*symmetric*, *code-pred-inline*]

125.2 Alternative rules for *list-all2*

lemma *list-all2-NilI* [*code-pred-intro*]: *list-all2 P* [] []
⟨*proof*⟩

lemma *list-all2-ConsI* [*code-pred-intro*]: *list-all2 P xs ys* ==> *P x y* ==> *list-all2 P* (*x*#*xs*) (*y*#*ys*)
⟨*proof*⟩

code-pred [*skip-proof*] *list-all2*
⟨*proof*⟩

125.3 Alternative rules for membership in lists

declare *in-set-member*[*code-pred-inline*]

lemma *member-intros* [*code-pred-intro*]:
List.member (*x*#*xs*) *x*
List.member xs x ⟹ *List.member* (*y*#*xs*) *x*
⟨*proof*⟩

code-pred *List.member*
⟨*proof*⟩

code-identifier constant *member-i-i*
⇀ (*SML*) *List.member-i-i*
and (*OCaml*) *List.member-i-i*
and (*Haskell*) *List.member-i-i*
and (*Scala*) *List.member-i-i*

code-identifier constant *member-i-o*
⇀ (*SML*) *List.member-i-o*
and (*OCaml*) *List.member-i-o*
and (*Haskell*) *List.member-i-o*
and (*Scala*) *List.member-i-o*

126 Setup for String.literal

⟨*ML*⟩

127 Simplification rules for optimisation

lemma [*code-pred-simp*]: ¬ *False* == *True*
⟨*proof*⟩

lemma [*code-pred-simp*]: ¬ *True* == *False*
⟨*proof*⟩

lemma *less-nat-k-0* [*code-pred-simp*]: *less-nat k 0* == *False*
⟨*proof*⟩

end

128 A Prototype of Quickcheck based on the Predicate Compiler

theory *Predicate-Compile-Quickcheck*
imports *Predicate-Compile-Alternative-Defs*
begin

⟨*ML*⟩

end

129 TFL: recursive function definitions

theory *Old-Recdef*
imports *Main*
keywords
recdef :: *thy-decl* **and**
permissive congs hints
begin

129.1 Lemmas for TFL

lemma *tfl-wf-induct*: *ALL R. wf R* −−>
(*ALL P.* (*ALL x.* (*ALL y.* (*y*,*x*):*R* −−> *P y*) −−> *P x*) −−> (*ALL x. P x*))
⟨*proof*⟩

lemma *tfl-cut-def*: *cut f r x* ≡ (λ*y. if* (*y*,*x*) ∈ *r then f y else undefined*)
⟨*proof*⟩

lemma *tfl-cut-apply*: *ALL f R.* (*x*,*a*):*R* −−> (*cut f R a*)(*x*) = *f*(*x*)
⟨*proof*⟩

lemma *tfl-wfrec*:
ALL M R f. (*f*=*wfrec R M*) −−> *wf R* −−> (*ALL x. f x* = *M* (*cut f R x*) *x*)
⟨*proof*⟩

lemma *tfl-eq-True*: (*x* = *True*) −−> *x*
⟨*proof*⟩

lemma *tfl-rev-eq-mp*: $(x = y) \longrightarrow y \longrightarrow x$
⟨*proof*⟩

lemma *tfl-simp-thm*: $(x \longrightarrow y) \longrightarrow (x = x') \longrightarrow (x' \longrightarrow y)$
⟨*proof*⟩

lemma *tfl-P-imp-P-iff-True*: $P \Longrightarrow P = True$
⟨*proof*⟩

lemma *tfl-imp-trans*: $(A \longrightarrow B) \Longrightarrow (B \longrightarrow C) \Longrightarrow (A \longrightarrow C)$
⟨*proof*⟩

lemma *tfl-disj-assoc*: $(a \vee b) \vee c \equiv a \vee (b \vee c)$
⟨*proof*⟩

lemma *tfl-disjE*: $P \vee Q \Longrightarrow P \longrightarrow R \Longrightarrow Q \longrightarrow R \Longrightarrow R$
⟨*proof*⟩

lemma *tfl-exE*: $\exists x.\ P\ x \Longrightarrow \forall x.\ P\ x \longrightarrow Q \Longrightarrow Q$
⟨*proof*⟩

⟨*ML*⟩

129.2 Rule setup

lemmas [*recdef-simp*] =
inv-image-def
measure-def
lex-prod-def
same-fst-def
less-Suc-eq [*THEN iffD2*]

lemmas [*recdef-cong*] =
if-cong let-cong image-cong INF-cong SUP-cong bex-cong ball-cong imp-cong
map-cong filter-cong takeWhile-cong dropWhile-cong foldl-cong foldr-cong

lemmas [*recdef-wf*] =
wf-trancl
wf-less-than
wf-lex-prod
wf-inv-image
wf-measure
wf-measures
wf-pred-nat
wf-same-fst
wf-empty

end

130 Refute

theory *Refute*
imports *Main*
keywords
refute :: *diag* **and**
refute-params :: *thy-decl*
begin

⟨*ML*⟩

refute-params
[*itself* = *1*,
minsize = *1*,
maxsize = *8*,
maxvars = *10000*,
maxtime = *60*,
satsolver = *auto*,
no-assms = *false*]

```
(* ------------------------------------------------------------------------- *)
(* REFUTE                                                                    *)
(*                                                                           *)
(* We use a SAT solver to search for a (finite) model that refutes a given   *)
(* HOL formula.                                                              *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* NOTE                                                                      *)
(*                                                                           *)
(* I strongly recommend that you install a stand-alone SAT solver if you     *)
(* want to use 'refute'.  For details see 'HOL/Tools/sat_solver.ML'.  If you *)
(* have installed (a supported version of) zChaff, simply set 'ZCHAFF_HOME'  *)
(* in 'etc/settings'.                                                        *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* USAGE                                                                     *)
(*                                                                           *)
(* See the file 'HOL/ex/Refute_Examples.thy' for examples.  The supported    *)
(* parameters are explained below.                                           *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* CURRENT LIMITATIONS                                                       *)
(*                                                                           *)
(* 'refute' currently accepts formulas of higher-order predicate logic (with *)
(* equality), including free/bound/schematic variables, lambda abstractions, *)
(* sets and set membership, "arbitrary", "The", "Eps", records and           *)
```

```
(* inductively defined sets.  Constants are unfolded automatically, and sort *)
(* axioms are added as well.  Other, user-asserted axioms however are        *)
(* ignored.  Inductive datatypes and recursive functions are supported, but  *)
(* may lead to spurious countermodels.                                       *)
(*                                                                           *)
(* The (space) complexity of the algorithm is non-elementary.                *)
(*                                                                           *)
(* Schematic type variables are not supported.                               *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* PARAMETERS                                                                *)
(*                                                                           *)
(* The following global parameters are currently supported (and required,    *)
(* except for "expect"):                                                     *)
(*                                                                           *)
(* Name          Type    Description                                         *)
(*                                                                           *)
(* "minsize"     int     Only search for models with size at least           *)
(*                       'minsize'.                                          *)
(* "maxsize"     int     If >0, only search for models with size at most     *)
(*                       'maxsize'.                                          *)
(* "maxvars"     int     If >0, use at most 'maxvars' boolean variables      *)
(*                       when transforming the term into a propositional     *)
(*                       formula.                                            *)
(* "maxtime"     int     If >0, terminate after at most 'maxtime' seconds.   *)
(*                       This value is ignored under some ML compilers.      *)
(* "satsolver"   string  Name of the SAT solver to be used.                  *)
(* "no_assms"    bool    If "true", assumptions in structured proofs are     *)
(*                       not considered.                                     *)
(* "expect"      string  Expected result ("genuine", "potential", "none", or *)
(*                       "unknown").                                         *)
(*                                                                           *)
(* The size of particular types can be specified in the form type=size       *)
(* (where 'type' is a string, and 'size' is an int).  Examples:              *)
(* "'a"=1                                                                    *)
(* "List.list"=2                                                             *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* FILES                                                                     *)
(*                                                                           *)
(* HOL/Tools/prop_logic.ML     Propositional logic                           *)
(* HOL/Tools/sat_solver.ML     SAT solvers                                   *)
(* HOL/Tools/refute.ML         Translation HOL -> propositional logic and    *)
(*                             Boolean assignment -> HOL model               *)
(* HOL/Refute.thy              This file: loads the ML files, basic setup,   *)
(*                             documentation                                 *)
(* HOL/SAT.thy                 Sets default parameters                       *)
```

```
(* HOL/ex/Refute_Examples.thy  Examples                                      *)
(* ------------------------------------------------------------------------- *)
```

end

References

[1] J. Avigad and K. Donnelly. Formalizing O notation in Isabelle/HOL. In D. Basin and M. Rusiowitch, editors, *Automated Reasoning: second international conference, IJCAR 2004*, pages 357–371. Springer, 2004.

[2] A. Podelski and A. Rybalchenko. Transition invariants. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41, 2004.