

# Isabelle/HOL — Higher-Order Logic

April 19, 2009

## Contents

<b>1</b>	<b>HOL: The basis of Higher-Order Logic</b>	<b>17</b>
1.1	Primitive logic	17
1.1.1	Core syntax	17
1.1.2	Additional concrete syntax	18
1.1.3	Axioms and basic definitions	20
1.1.4	Generic classes and algebraic operations	21
1.2	Fundamental rules	23
1.2.1	Equality	23
1.2.2	Congruence rules for application	23
1.2.3	Equality of booleans – iff	24
1.2.4	True	24
1.2.5	Universal quantifier	24
1.2.6	False	25
1.2.7	Negation	25
1.2.8	Implication	26
1.2.9	Existential quantifier	26
1.2.10	Conjunction	27
1.2.11	Disjunction	27
1.2.12	Classical logic	28
1.2.13	Unique existence	28
1.2.14	THE: definite description operator	29
1.2.15	Classical intro rules for disjunction and existential quantifiers	30
1.2.16	Intuitionistic Reasoning	31
1.2.17	Atomizing meta-level connectives	32
1.2.18	Atomizing elimination rules	34
1.3	Package setup	34
1.3.1	Classical Reasoner setup	34
1.3.2	Simplifier	37
1.3.3	Generic cases and induction	46
1.3.4	Coherent logic	47

1.4	Other simple lemmas and lemma duplicates . . . . .	48
1.5	Basic ML bindings . . . . .	49
1.6	Code generator basics – see further theory <i>Code-Setup</i> . . . .	50
1.7	Nitpick hooks . . . . .	50
1.8	Legacy tactics and ML bindings . . . . .	51
<b>2</b>	<b>Code-Setup: Setup of code generators and related tools</b>	<b>52</b>
2.1	Generic code generator foundation . . . . .	52
2.2	Generic code generator preprocessor . . . . .	54
2.3	Generic code generator target languages . . . . .	54
2.4	SML code generator setup . . . . .	55
2.5	Evaluation and normalization by evaluation . . . . .	56
2.6	Quickcheck . . . . .	57
<b>3</b>	<b>Orderings: Abstract orderings</b>	<b>57</b>
3.1	Quasi orders . . . . .	57
3.2	Partial orders . . . . .	59
3.3	Linear (total) orders . . . . .	60
3.4	Reasoning tools setup . . . . .	62
3.5	Name duplicates . . . . .	68
3.6	Bounded quantifiers . . . . .	69
3.7	Transitivity reasoning . . . . .	71
3.8	Monotonicity, least value operator and min/max . . . . .	76
3.9	Top and bottom elements . . . . .	79
3.10	Dense orders . . . . .	79
3.11	Wellorders . . . . .	79
3.12	Order on bool . . . . .	80
3.13	Order on functions . . . . .	81
<b>4</b>	<b>Lattices: Abstract lattices</b>	<b>83</b>
4.1	Lattices . . . . .	83
4.1.1	Intro and elim rules . . . . .	84
4.1.2	Equational laws . . . . .	85
4.2	Distributive lattices . . . . .	87
4.3	Uniqueness of inf and sup . . . . .	88
4.4	<i>min/max</i> on linear orders as special case of <i>op</i> $\sqcap$ / <i>op</i> $\sqcup$ . . . .	88
4.5	Bool as lattice . . . . .	89
4.6	Fun as lattice . . . . .	89
<b>5</b>	<b>Set: Set theory for higher-order logic</b>	<b>90</b>
5.1	Basic syntax . . . . .	90
5.2	Additional concrete syntax . . . . .	93
5.2.1	Bounded quantifiers . . . . .	95
5.3	Rules and definitions . . . . .	97

5.4	Lemmas and proof tool setup . . . . .	99
5.4.1	Relating predicates and sets . . . . .	99
5.4.2	Bounded quantifiers . . . . .	99
5.4.3	Congruence rules . . . . .	101
5.4.4	Subsets . . . . .	101
5.4.5	Equality . . . . .	102
5.4.6	The universal set – UNIV . . . . .	103
5.4.7	The empty set . . . . .	103
5.4.8	The Powerset operator – Pow . . . . .	104
5.4.9	Set complement . . . . .	104
5.4.10	Binary union – Un . . . . .	105
5.4.11	Binary intersection – Int . . . . .	105
5.4.12	Set difference . . . . .	105
5.4.13	Augmenting a set – insert . . . . .	106
5.4.14	Singletons, using insert . . . . .	107
5.4.15	Unions of families . . . . .	107
5.4.16	Intersections of families . . . . .	108
5.4.17	Union . . . . .	108
5.4.18	Inter . . . . .	109
5.4.19	Set reasoning tools . . . . .	110
5.4.20	The “proper subset” relation . . . . .	111
5.5	Further set-theory lemmas . . . . .	112
5.5.1	Derived rules involving subsets. . . . .	112
5.5.2	Equalities involving union, intersection, inclusion, etc. . . . .	113
5.5.3	Monotonicity of various operations . . . . .	129
5.6	Inverse image of a function . . . . .	131
5.6.1	Basic rules . . . . .	131
5.6.2	Equations . . . . .	131
5.7	Getting the Contents of a Singleton Set . . . . .	133
5.8	Transitivity rules for calculational reasoning . . . . .	133
5.9	Least value operator . . . . .	133
5.10	Rudimentary code generation . . . . .	133
5.11	Complete lattices . . . . .	134
5.12	Bool as complete lattice . . . . .	136
5.13	Fun as complete lattice . . . . .	137
5.14	Set as lattice . . . . .	137
5.15	Misc theorem and ML bindings . . . . .	139
<b>6</b>	<b>Typedef: HOL type definitions</b>	<b>140</b>
<b>7</b>	<b>Fun: Notions about functions</b>	<b>142</b>
7.1	The Identity Function <i>id</i> . . . . .	143
7.2	The Composition Operator $f \circ g$ . . . . .	143
7.3	The Forward Composition Operator <i>fcomp</i> . . . . .	144

7.4	Injectivity and Surjectivity . . . . .	144
7.5	Function Updating . . . . .	149
7.6	<i>override-on</i> . . . . .	150
7.7	<i>swap</i> . . . . .	151
7.8	Proof tool setup . . . . .	152
7.9	Code generator setup . . . . .	153
<b>8</b>	<b>Sum-Type: The Disjoint Sum of Two Types</b>	<b>153</b>
8.1	Freeness Properties for <i>Inl</i> and <i>Inr</i> . . . . .	154
8.2	The Disjoint Sum of Sets . . . . .	155
8.3	The <i>Part</i> Primitive . . . . .	156
<b>9</b>	<b>Inductive: Knaster-Tarski Fixpoint Theorem and inductive definitions</b>	<b>158</b>
9.1	Least and greatest fixed points . . . . .	158
9.2	Proof of Knaster-Tarski Theorem using <i>lfp</i> . . . . .	158
9.3	General induction rules for least fixed points . . . . .	159
9.4	Proof of Knaster-Tarski Theorem using <i>gfp</i> . . . . .	160
9.5	Coinduction rules for greatest fixed points . . . . .	161
9.6	Even Stronger Coinduction Rule, by Martin Coen . . . . .	161
9.7	Inductive predicates and sets . . . . .	162
9.8	Inductive datatypes and primitive recursion . . . . .	164
<b>10</b>	<b>OrderedGroup: Ordered Groups</b>	<b>165</b>
10.1	Semigroups and Monoids . . . . .	165
10.2	Groups . . . . .	168
10.3	(Partially) Ordered Groups . . . . .	171
10.4	Support for reasoning about signs . . . . .	173
10.5	Lattice Ordered (Abelian) Groups . . . . .	183
10.6	Positive Part, Negative Part, Absolute Value . . . . .	185
10.7	Tools setup . . . . .	192
<b>11</b>	<b>Ring-and-Field: (Ordered) Rings and Fields</b>	<b>193</b>
11.1	Calculations with fractions . . . . .	221
11.1.1	Special Cancellation Simprules for Division . . . . .	222
11.2	Division and Unary Minus . . . . .	222
11.3	Ordered Fields . . . . .	222
11.4	Anti-Monotonicity of <i>inverse</i> . . . . .	224
11.5	Inverses and the Number One . . . . .	225
11.6	Simplification of Inequalities Involving Literal Divisors . . . . .	226
11.7	Field simplification . . . . .	229
11.8	Division and Signs . . . . .	229
11.9	Cancellation Laws for Division . . . . .	230
11.10	Division and the Number One . . . . .	231

11.11	Ordering Rules for Division . . . . .	231
11.12	Conditional Simplification Rules: No Case Splits . . . . .	233
11.13	Reasoning about inequalities with division . . . . .	234
11.14	Ordered Fields are Dense . . . . .	235
11.15	Absolute Value . . . . .	236
11.16	Bounds of products via negative and positive Part . . . . .	239
<b>12</b>	<b>Nat: Natural numbers</b>	<b>241</b>
12.1	Type <i>ind</i> . . . . .	241
12.2	Type <i>nat</i> . . . . .	241
12.3	Arithmetic operators . . . . .	243
12.3.1	Addition . . . . .	245
12.3.2	Difference . . . . .	246
12.3.3	Multiplication . . . . .	247
12.4	Orders on <i>nat</i> . . . . .	248
12.4.1	Operation definition . . . . .	248
12.4.2	Introduction properties . . . . .	250
12.4.3	Elimination properties . . . . .	250
12.4.4	Inductive (?) properties . . . . .	251
12.4.5	<i>min</i> and <i>max</i> . . . . .	254
12.4.6	Monotonicity of Addition . . . . .	255
12.4.7	Additional theorems about <i>op</i> $\leq$ . . . . .	256
12.4.8	More results about difference . . . . .	261
12.4.9	Monotonicity of Multiplication . . . . .	262
12.5	Embedding of the Naturals into any <i>semiring-1: of-nat</i> . . . . .	264
12.6	The Set of Natural Numbers . . . . .	266
12.7	Further Arithmetic Facts Concerning the Natural Numbers . . . . .	267
12.8	size of a datatype value . . . . .	271
<b>13</b>	<b>Product-Type: Cartesian products</b>	<b>271</b>
13.1	<i>bool</i> is a datatype . . . . .	272
13.2	Unit . . . . .	272
13.3	Pairs . . . . .	273
13.3.1	Product type, basic operations and concrete syntax . . . . .	273
13.3.2	Basic rules and proof tools . . . . .	277
13.3.3	<i>split</i> and <i>curry</i> . . . . .	278
13.4	Further cases/induct rules for tuples . . . . .	284
13.4.1	Derived operations . . . . .	284
13.4.2	Code generator setup . . . . .	289
13.5	Legacy bindings . . . . .	291
13.6	Further inductive packages . . . . .	293

<b>14 Datatype: Analogues of the Cartesian Product and Disjoint Sum for Datatypes</b>	<b>294</b>
14.1 Freeness: Distinctness of Constructors . . . . .	296
14.2 Set Constructions . . . . .	300
<b>15 Datatypes</b>	<b>304</b>
15.1 Representing sums . . . . .	304
<b>16 Power: Exponentiation</b>	<b>305</b>
16.1 Powers for Arbitrary Monoids . . . . .	306
16.2 Exponentiation for the Natural Numbers . . . . .	313
<b>17 Finite-Set: Finite sets</b>	<b>314</b>
17.1 Definition and basic properties . . . . .	314
17.1.1 Finiteness and set theoretic constructions . . . . .	317
17.2 Class <i>finite</i> . . . . .	322
17.3 A fold functional for finite sets . . . . .	323
17.3.1 From <i>fold-graph</i> to <i>fold</i> . . . . .	324
17.3.2 The derived combinator <i>fold-image</i> . . . . .	328
17.4 Generalized summation over a set . . . . .	331
17.4.1 Properties in more restricted classes of structures . . .	337
17.5 Generalized product over a set . . . . .	345
17.5.1 Properties in more restricted classes of structures . . .	349
17.6 Finite cardinality . . . . .	351
17.6.1 Cardinality of unions . . . . .	355
17.6.2 Cardinality of image . . . . .	356
17.6.3 Cardinality of products . . . . .	357
17.6.4 Cardinality of sums . . . . .	357
17.6.5 Cardinality of the Powerset . . . . .	357
17.6.6 Relating injectivity and surjectivity . . . . .	358
17.7 A fold functional for non-empty sets . . . . .	359
17.7.1 Determinacy for <i>fold1Set</i> . . . . .	362
17.7.2 Lemmas about <i>fold1</i> . . . . .	363
17.7.3 Fold1 in lattices with <i>inf</i> and <i>sup</i> . . . . .	363
17.7.4 Fold1 in linear orders with <i>min</i> and <i>max</i> . . . . .	368
<b>18 Relation: Relations</b>	<b>376</b>
18.1 Definitions . . . . .	376
18.2 The identity relation . . . . .	377
18.3 Diagonal: identity over a set . . . . .	378
18.4 Composition of two relations . . . . .	378
18.5 Reflexivity . . . . .	379
18.6 Antisymmetry . . . . .	380
18.7 Symmetry . . . . .	380

18.8	Transitivity	381
18.9	Irreflexivity	381
18.10	Totality	381
18.11	Converse	381
18.12	Domain	383
18.13	Range	384
18.14	Field	385
18.15	Image of a set under a relation	385
18.16	Single valued relations	387
18.17	Graphs given by <i>Collect</i>	387
18.18	Inverse image	387
18.19	Finiteness	388
18.20	Version of <i>lfp-induct</i> for binary relations	388
<b>19</b>	<b>Predicate: Predicates as relations and enumerations</b>	<b>388</b>
19.1	Predicates as (complete) lattices	389
19.1.1	<i>op</i> $\sqcup$ on <i>bool</i>	389
19.1.2	Equality and Subsets	389
19.1.3	Top and bottom elements	389
19.1.4	The empty set	389
19.1.5	Binary union	390
19.1.6	Binary intersection	390
19.1.7	Unions of families	391
19.1.8	Intersections of families	392
19.2	Predicates as relations	393
19.2.1	Composition	393
19.2.2	Converse	393
19.2.3	Domain	394
19.2.4	Range	394
19.2.5	Inverse image	394
19.2.6	Powerset	395
19.2.7	Properties of relations	395
19.3	Predicates as enumerations	395
19.3.1	The type of predicate enumerations (a monad)	395
19.3.2	Derived operations	397
19.3.3	Implementation	398
<b>20</b>	<b>Transitive-Closure: Reflexive and Transitive closure of a relation</b>	<b>401</b>
20.1	Reflexive closure	402
20.2	Reflexive-transitive closure	402
20.3	Transitive closure	407
20.4	Setup of transitivity reasoner	414

<b>21 Wellfounded: Well-founded Recursion</b>	<b>415</b>
21.1 Basic Definitions . . . . .	415
21.2 Basic Results . . . . .	417
21.3 Well-Foundedness Results for Unions . . . . .	420
21.3.1 acyclic . . . . .	422
21.4 Well-Founded Recursion . . . . .	423
21.5 Code generator setup . . . . .	424
21.6 <i>nat</i> is well-founded . . . . .	424
21.7 Accessible Part . . . . .	425
21.8 Tools for building wellfounded relations . . . . .	428
21.9 Weakly decreasing sequences (w.r.t. some well-founded order)	
stabilize. . . . .	433
21.10size of a datatype value . . . . .	433
<b>22 FunDef: Function Definitions and Termination Proofs</b>	<b>434</b>
22.1 Definitions with default value. . . . .	435
22.2 Measure Functions . . . . .	436
22.3 Congruence Rules . . . . .	437
22.4 Simp rules for termination proofs . . . . .	437
22.5 Decomposition . . . . .	438
22.6 Reduction Pairs . . . . .	438
22.7 Concrete orders for SCNP termination proofs . . . . .	438
22.8 Tool setup . . . . .	440
<b>23 Record: Extensible records with structural subtyping</b>	<b>441</b>
23.1 Concrete record syntax . . . . .	441
<b>24 Option: Datatype option</b>	<b>442</b>
24.0.1 Operations . . . . .	442
24.0.2 Code generator setup . . . . .	443
<b>25 Extraction: Program extraction for HOL</b>	<b>444</b>
25.1 Setup . . . . .	444
25.2 Type of extracted program . . . . .	445
25.3 Realizability . . . . .	446
25.4 Computational content of basic inference rules . . . . .	447
<b>26 Divides: The division operators <i>div</i> and <i>mod</i></b>	<b>453</b>
26.1 Syntactic division operations . . . . .	453
26.2 Abstract division in commutative semirings. . . . .	453
26.3 Division on <i>nat</i> . . . . .	461
26.3.1 Quotient . . . . .	465
26.3.2 Remainder . . . . .	466
26.3.3 Quotient and Remainder . . . . .	467



26.3.4	Cancellation of Common Factors in Division . . . . .	468
26.3.5	Further Facts about Quotient and Remainder . . . . .	468
26.3.6	The Divides Relation . . . . .	470
26.3.7	An “induction” law for modulus arithmetic. . . . .	474
<b>27</b>	<b>Plain: Plain HOL</b>	<b>476</b>
<b>28</b>	<b>Relation-Power: Powers of Relations and Functions</b>	<b>476</b>
<b>29</b>	<b>Equiv-Relations: Equivalence Relations in Higher-Order Set Theory</b>	<b>480</b>
29.1	Equivalence relations . . . . .	480
29.2	Equivalence classes . . . . .	481
29.3	Quotients . . . . .	482
29.4	Defining unary operations upon equivalence classes . . . . .	483
29.5	Defining binary operations upon equivalence classes . . . . .	484
29.6	Quotients and finiteness . . . . .	486
<b>30</b>	<b>Int: The Integers as Equivalence Classes over Pairs of Natural Numbers</b>	<b>486</b>
30.1	The equivalence relation underlying the integers . . . . .	487
30.2	Construction of the Integers . . . . .	488
30.3	Arithmetic Operations . . . . .	488
30.4	The $\leq$ Ordering . . . . .	490
30.5	Embedding of the Integers into any <i>ring-1: of-int</i> . . . . .	492
30.6	Magnitude of an Integer, as a Natural Number: <i>nat</i> . . . . .	494
30.7	Lemmas about the Function <i>of-nat</i> and Orderings . . . . .	496
30.8	Cases and induction . . . . .	497
30.9	Binary representation . . . . .	498
30.9.1	The constructors <i>Bit0</i> , <i>Bit1</i> , <i>Pls</i> and <i>Min</i> . . . . .	498
30.9.2	Successor and predecessor functions . . . . .	500
30.9.3	Binary arithmetic . . . . .	500
30.9.4	Binary comparisons . . . . .	502
30.10	Converting Numerals to Rings: <i>number-of</i> . . . . .	505
30.10.1	Equality of Binary Numbers . . . . .	507
30.10.2	Comparisons, for Ordered Rings . . . . .	508
30.10.3	The Less-Than Relation . . . . .	509
30.10.4	Simplification of arithmetic operations on integer constants. . . . .	510
30.10.5	Simplification of arithmetic when nested to the right. . . . .	510
30.11	The Set of Integers . . . . .	511
30.12	<i>setsum</i> and <i>setprod</i> . . . . .	513
30.13	Inequality Reasoning for the Arithmetic Simproc . . . . .	514
30.14	Special Arithmetic Rules for Abstract 0 and 1 . . . . .	514

30.15	Setting up simplification procedures . . . . .	516
30.16	Lemmas About Small Numerals . . . . .	516
30.17	More Inequality Reasoning . . . . .	517
30.18	The functions <i>nat</i> and <i>int</i> . . . . .	517
30.19	Induction principles for <i>int</i> . . . . .	519
30.20	Intermediate value theorems . . . . .	521
30.21	Products and 1, by T. M. Rasmussen . . . . .	522
30.22	Integer Powers . . . . .	523
30.23	Further theorems on numerals . . . . .	524
30.23.1	Special Simplification for Constants . . . . .	524
30.23.2	Optional Simplification Rules Involving Constants . . . . .	526
30.24	Configuration of the code generator . . . . .	526
30.25	Legacy theorems . . . . .	530
<b>31</b>	<b>IntDiv: The Division Operators <i>div</i> and <i>mod</i></b>	<b>531</b>
31.1	Uniqueness and Monotonicity of Quotients and Remainders . . . . .	533
31.2	Correctness of <i>posDivAlg</i> , the Algorithm for Non-Negative Dividends . . . . .	534
31.3	Correctness of <i>negDivAlg</i> , the Algorithm for Negative Dividends	535
31.4	Existence Shown by Proving the Division Algorithm to be Correct . . . . .	535
31.5	General Properties of <i>div</i> and <i>mod</i> . . . . .	537
31.6	Laws for <i>div</i> and <i>mod</i> with Unary Minus . . . . .	538
31.7	Division of a Number by Itself . . . . .	539
31.8	Computation of Division and Remainder . . . . .	540
31.9	Monotonicity in the First Argument (Dividend) . . . . .	543
31.10	Monotonicity in the Second Argument (Divisor) . . . . .	543
31.11	More Algebraic Laws for <i>div</i> and <i>mod</i> . . . . .	544
31.12	Proving $a \text{ div } (b * c) = a \text{ div } b \text{ div } c$ . . . . .	546
31.13	Cancellation of Common Factors in <i>div</i> . . . . .	547
31.14	Distribution of Factors over <i>mod</i> . . . . .	548
31.15	Splitting Rules for <i>div</i> and <i>mod</i> . . . . .	548
31.16	Speeding up the Division Algorithm with Shifting . . . . .	549
31.17	Computing <i>mod</i> by Shifting (proofs resemble those for <i>div</i> ) . . . . .	550
31.18	Quotients of Signs . . . . .	551
31.19	The Divides Relation . . . . .	552
31.20	Simproc setup . . . . .	558
31.21	Code generation . . . . .	558
<b>32</b>	<b>NatBin: Binary arithmetic for the natural numbers</b>	<b>559</b>
32.1	Predicate for negative binary numbers . . . . .	560
32.2	Function <i>nat</i> : Coercion from Type <i>int</i> to <i>nat</i> . . . . .	561
32.3	Function <i>int</i> : Coercion from Type <i>nat</i> to <i>int</i> . . . . .	561
32.3.1	Successor . . . . .	562

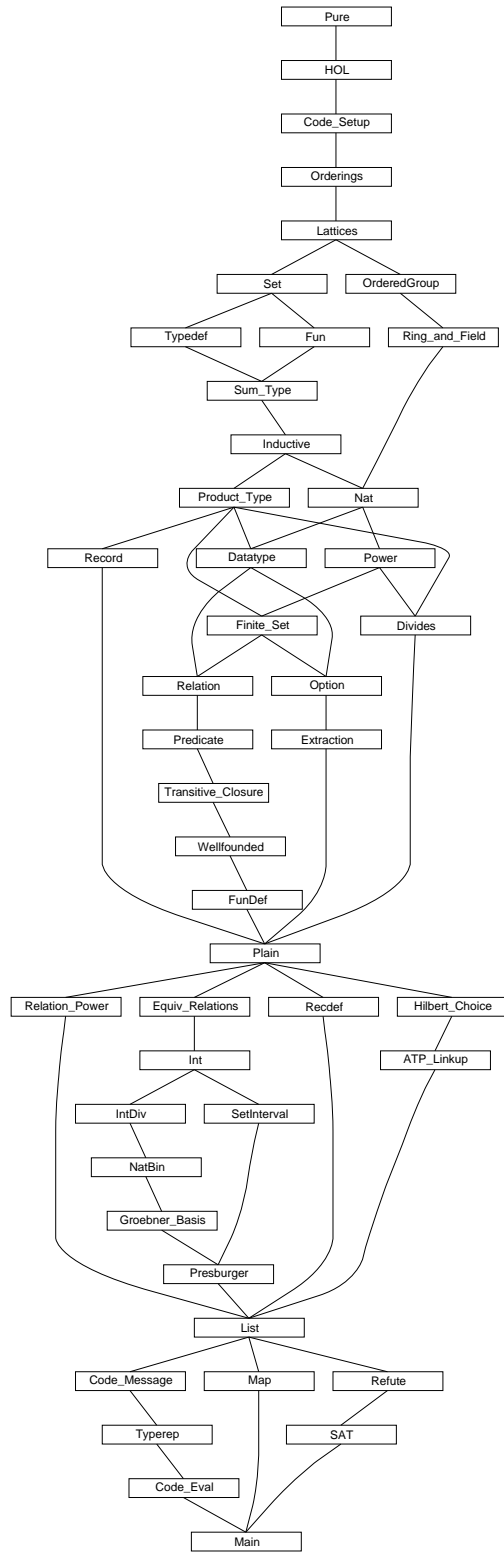
32.3.2	Addition . . . . .	562
32.3.3	Subtraction . . . . .	563
32.3.4	Multiplication . . . . .	563
32.3.5	Quotient . . . . .	563
32.3.6	Remainder . . . . .	563
32.3.7	Divisibility . . . . .	564
32.4	Comparisons . . . . .	564
32.4.1	Equals (=) . . . . .	564
32.4.2	Less-than (<) . . . . .	565
32.4.3	Less-than-or-equal . . . . .	565
32.5	Powers with Numeric Exponents . . . . .	565
32.5.1	Nat . . . . .	568
32.5.2	Arith . . . . .	568
32.6	Comparisons involving (0::nat) . . . . .	568
32.7	Comparisons involving <i>Suc</i> . . . . .	569
32.8	Max and Min Combined with <i>Suc</i> . . . . .	570
32.9	Literal arithmetic involving powers . . . . .	571
32.10	Literal arithmetic and <i>of-nat</i> . . . . .	573
32.11	Lemmas for the Combination and Cancellation Simprocs . . . . .	573
32.11.1	For <i>combine-numerals</i> . . . . .	574
32.11.2	For <i>cancel-numerals</i> . . . . .	574
32.11.3	For <i>cancel-numeral-factors</i> . . . . .	574
32.11.4	For <i>cancel-factor</i> . . . . .	575
32.12	Simprocs for the Naturals . . . . .	575
32.12.1	For simplifying $Suc\ m - K$ and $K - Suc\ m$ . . . . .	575
32.12.2	For <i>nat-case</i> and <i>nat-rec</i> . . . . .	576
32.12.3	Various Other Lemmas . . . . .	577
<b>33</b>	<b>Groebner-Basis: Semiring normalization and Groebner Bases</b>	<b>578</b>
33.1	Semiring normalization . . . . .	579
33.1.1	Declaring the abstract theory . . . . .	579
33.2	Groebner Bases . . . . .	584
33.3	Groebner Bases for fields . . . . .	588
<b>34</b>	<b>SetInterval: Set intervals</b>	<b>593</b>
34.1	Various equivalences . . . . .	594
34.2	Logical Equivalences for Set Inclusion and Equality . . . . .	595
34.3	Two-sided intervals . . . . .	596
34.3.1	Emptiness and singletons . . . . .	596
34.4	Intervals of natural numbers . . . . .	597
34.4.1	The Constant <i>lessThan</i> . . . . .	597
34.4.2	The Constant <i>greaterThan</i> . . . . .	597
34.4.3	The Constant <i>atLeast</i> . . . . .	597
34.4.4	The Constant <i>atMost</i> . . . . .	598

34.4.5	The Constant <i>atLeastLessThan</i> . . . . .	598
34.4.6	Intervals of nats with <i>Suc</i> . . . . .	598
34.4.7	Image . . . . .	599
34.4.8	Finiteness . . . . .	600
34.4.9	Cardinality . . . . .	601
34.5	Intervals of integers . . . . .	602
34.5.1	Finiteness . . . . .	602
34.5.2	Cardinality . . . . .	603
34.6	Lemmas useful with the summation operator <i>setsum</i> . . . . .	604
34.6.1	Disjoint Unions . . . . .	604
34.6.2	Disjoint Intersections . . . . .	605
34.6.3	Some Differences . . . . .	606
34.6.4	Some Subset Conditions . . . . .	606
34.7	Summation indexed over intervals . . . . .	606
34.8	Shifting bounds . . . . .	609
34.9	The formula for geometric sums . . . . .	609
34.10	The formula for arithmetic sums . . . . .	610
34.11	Products indexed over intervals . . . . .	611
<b>35</b>	<b>Presburger: Decision Procedure for Presburger Arithmetic</b>	<b>612</b>
35.1	The $-\infty$ and $+\infty$ Properties . . . . .	613
35.2	The A and B sets . . . . .	614
35.3	Cooper's Theorem $-\infty$ and $+\infty$ Version . . . . .	617
35.3.1	First some trivial facts about periodic sets or predicates	617
35.3.2	The $-\infty$ Version . . . . .	617
35.3.3	The $+\infty$ Version . . . . .	619
<b>36</b>	<b>Recdef: TFL: recursive function definitions</b>	<b>623</b>
<b>37</b>	<b>Hilbert-Choice: Hilbert's Epsilon-Operator and the Axiom of Choice</b>	<b>625</b>
37.1	Hilbert's epsilon . . . . .	626
37.2	Hilbert's Epsilon-operator . . . . .	626
37.3	Axiom of Choice, Proved Using the Description Operator . .	627
37.4	Function Inverse . . . . .	627
37.5	Inverse of a PI-function (restricted domain) . . . . .	630
37.6	Other Consequences of Hilbert's Epsilon . . . . .	631
37.7	Least value operator . . . . .	632
37.8	Greatest value operator . . . . .	633
37.9	The Meson proof procedure . . . . .	634
37.9.1	Negation Normal Form . . . . .	634
37.9.2	Pulling out the existential quantifiers . . . . .	635
37.9.3	Generating clauses for the Meson Proof Procedure . .	635
37.10	Lemmas for Meson, the Model Elimination Procedure . . . .	635

37.10.1 Lemmas for Forward Proof . . . . .	636
37.11 Meson package . . . . .	638
37.12 Specification package – Hilbertized version . . . . .	638
<b>38 ATP-Linkup: The Isabelle-ATP Linkup</b>	<b>638</b>
38.1 Setup of external ATPs . . . . .	640
38.2 The Metis prover . . . . .	640
<b>39 List: The datatype of finite lists</b>	<b>640</b>
39.1 Basic list processing functions . . . . .	641
39.1.1 List comprehension . . . . .	645
39.1.2 $[]$ and $op \#$ . . . . .	648
39.1.3 <i>length</i> . . . . .	648
39.1.4 $@$ – append . . . . .	650
39.1.5 <i>map</i> . . . . .	653
39.1.6 <i>rev</i> . . . . .	656
39.1.7 <i>set</i> . . . . .	657
39.1.8 <i>filter</i> . . . . .	660
39.1.9 List partitioning . . . . .	663
39.1.10 <i>concat</i> . . . . .	663
39.1.11 <i>nth</i> . . . . .	664
39.1.12 <i>list-update</i> . . . . .	666
39.1.13 <i>last</i> and <i>butlast</i> . . . . .	667
39.1.14 <i>take</i> and <i>drop</i> . . . . .	669
39.1.15 <i>takeWhile</i> and <i>dropWhile</i> . . . . .	674
39.1.16 <i>zip</i> . . . . .	675
39.1.17 <i>list-all2</i> . . . . .	677
39.1.18 <i>foldl</i> and <i>foldr</i> . . . . .	680
39.1.19 List summation: <i>listsum</i> and $\sum$ . . . . .	682
39.1.20 <i>upt</i> . . . . .	683
39.1.21 <i>distinct</i> and <i>remdups</i> . . . . .	686
39.1.22 <i>remove1</i> . . . . .	688
39.1.23 <i>removeAll</i> . . . . .	689
39.1.24 <i>replicate</i> . . . . .	690
39.1.25 <i>rotate1</i> and <i>rotate</i> . . . . .	692
39.1.26 <i>sublist</i> — a generalization of <i>nth</i> to sets . . . . .	694
39.1.27 <i>splice</i> . . . . .	696
39.1.28 Infiniteness . . . . .	696
39.2 Sorting . . . . .	696
39.2.1 <i>sorted-list-of-set</i> . . . . .	699
39.2.2 <i>upto</i> : the generic interval-list . . . . .	699
39.2.3 <i>lists</i> : the list-forming operator over sets . . . . .	701
39.2.4 Inductive definition for membership . . . . .	702
39.2.5 Lists as Cartesian products . . . . .	702

39.3	Relations on Lists . . . . .	703
39.3.1	Length Lexicographic Ordering . . . . .	703
39.3.2	Lexicographic Ordering . . . . .	704
39.4	Lexicographic combination of measure functions . . . . .	706
39.4.1	Lifting a Relation on List Elements to the Lists . . . . .	707
39.5	Miscellany . . . . .	708
39.5.1	Characters and strings . . . . .	708
39.6	Size function . . . . .	709
39.7	Code generator . . . . .	710
39.7.1	Setup . . . . .	710
39.7.2	Generation of efficient code . . . . .	714
<b>40</b>	<b>Code-Message: Monolithic strings (message strings) for code generation</b>	<b>719</b>
40.1	Datatype of messages . . . . .	719
40.2	ML interface . . . . .	719
40.3	Code serialization . . . . .	719
<b>41</b>	<b>Typerep: Reflecting Pure types into HOL</b>	<b>720</b>
<b>42</b>	<b>Code-Eval: Term evaluation using the generic code generator</b>	<b>722</b>
42.1	Term representation . . . . .	722
42.1.1	Terms and class <i>term-of</i> . . . . .	722
42.1.2	<i>term-of</i> instances . . . . .	723
42.1.3	Code generator setup . . . . .	725
42.2	Evaluation setup . . . . .	725
42.2.1	Syntax . . . . .	726
<b>43</b>	<b>Map: Maps</b>	<b>726</b>
43.1	<i>empty</i> . . . . .	728
43.2	<i>map-upd</i> . . . . .	728
43.3	<i>map-of</i> . . . . .	729
43.4	<i>Option.map</i> related . . . . .	731
43.5	<i>map-comp</i> related . . . . .	731
43.6	<i>++</i> . . . . .	732
43.7	<i>restrict-map</i> . . . . .	733
43.8	<i>map-upds</i> . . . . .	734
43.9	<i>dom</i> . . . . .	735
43.10	<i>ran</i> . . . . .	737
43.11	<i>map-le</i> . . . . .	737
<b>44</b>	<b>Refute: Refute</b>	<b>738</b>

45 SAT: Reconstructing external resolution proofs for propositional logic	740
46 Main: Main HOL	741





# 1 HOL: The basis of Higher-Order Logic

```

theory HOL
imports Pure
uses
  (Tools/hologic.ML)
  ~~ /src/Tools/IsaPlanner/zipper.ML
  ~~ /src/Tools/IsaPlanner/isand.ML
  ~~ /src/Tools/IsaPlanner/rw-tools.ML
  ~~ /src/Tools/IsaPlanner/rw-inst.ML
  ~~ /src/Tools/intuitionistic.ML
  ~~ /src/Tools/project-rule.ML
  ~~ /src/Provers/hypsubst.ML
  ~~ /src/Provers/splitter.ML
  ~~ /src/Provers/classical.ML
  ~~ /src/Provers/blast.ML
  ~~ /src/Provers/clasimp.ML
  ~~ /src/Tools/coherent.ML
  ~~ /src/Tools/eqsubst.ML
  ~~ /src/Provers/quantifier1.ML
  (Tools/simpdata.ML)
  ~~ /src/Tools/random-word.ML
  ~~ /src/Tools/atomize-elim.ML
  ~~ /src/Tools/induct.ML
  (~~ /src/Tools/induct-tacs.ML)
  ~~ /src/Tools/value.ML
  ~~ /src/Tools/code/code-name.ML
  ~~ /src/Tools/code/code-funcgr.ML
  ~~ /src/Tools/code/code-wellsorted.ML
  ~~ /src/Tools/code/code-thingol.ML
  ~~ /src/Tools/code/code-printer.ML
  ~~ /src/Tools/code/code-target.ML
  ~~ /src/Tools/code/code-ml.ML
  ~~ /src/Tools/code/code-haskell.ML
  ~~ /src/Tools/nbe.ML
  (Tools/recfun-codegen.ML)
begin

setup << Intuitionistic.method-setup iprover >>

1.1 Primitive logic

1.1.1 Core syntax

classes type
defaultsort type
setup << ObjectLogic.add-base-sort @{sort type} >>

arities
  fun :: (type, type) type

```

*itself* :: (type) type

**global**

**typeddecl** *bool*

**judgment**

*Trueprop* :: *bool* => *prop* ((-) 5)

**consts**

*Not* :: *bool* => *bool* ( $\sim$  - [40] 40)

*True* :: *bool*

*False* :: *bool*

*The* :: ('a => *bool*) => 'a

*All* :: ('a => *bool*) => *bool* (**binder** *ALL* 10)

*Ex* :: ('a => *bool*) => *bool* (**binder** *EX* 10)

*Ex1* :: ('a => *bool*) => *bool* (**binder** *EX!* 10)

*Let* :: ['a, 'a => 'b] => 'b

*op* = :: ['a, 'a] => *bool* (**infixl** = 50)

*op* & :: [*bool*, *bool*] => *bool* (**infixr** & 35)

*op* | :: [*bool*, *bool*] => *bool* (**infixr** | 30)

*op* --> :: [*bool*, *bool*] => *bool* (**infixr** --> 25)

**local**

**consts**

*If* :: [*bool*, 'a, 'a] => 'a ((if (-)/ then (-)/ else (-)) 10)

### 1.1.2 Additional concrete syntax

**notation** (output)

*op* = (**infix** = 50)

**abbreviation**

*not-equal* :: ['a, 'a] => *bool* (**infixl**  $\sim$  = 50) **where**

$x \sim = y == \sim (x = y)$

**notation** (output)

*not-equal* (**infix**  $\sim$  = 50)

**notation** (*xsymbols*)

*Not* ( $\neg$  - [40] 40) **and**

*op* & (**infixr**  $\wedge$  35) **and**

*op* | (**infixr**  $\vee$  30) **and**

*op* --> (**infixr**  $\longrightarrow$  25) **and**

*not-equal* (**infix**  $\neq$  50)

**notation** (*HTML output*)

*Not* ( $\neg$  - [40] 40) **and**  
*op* & (**infixr**  $\wedge$  35) **and**  
*op* | (**infixr**  $\vee$  30) **and**  
*not-equal* (**infix**  $\neq$  50)

**abbreviation** (*iff*)

*iff* :: [bool, bool] => bool (**infixr**  $\longleftrightarrow$  25) **where**  
 $A \longleftrightarrow B == A = B$

**notation** (*xsymbols*)

*iff* (**infixr**  $\longleftrightarrow$  25)

**nonterminals**

*letbinds* *letbind*  
*case-syn* *cases-syn*

**syntax**

*-The* :: [pttrn, bool] => 'a ((3THE -./ -) [0, 10] 10)

*-bind* :: [pttrn, 'a] => *letbind* ((2- =/ -) 10)  
 :: *letbind* => *letbinds* (-)

*-binds* :: [*letbind*, *letbinds*] => *letbinds* (-;/ -)

*-Let* :: [*letbinds*, 'a] => 'a ((let (-)/ in (-)) 10)

*-case-syntax*:: ['a, *cases-syn*] => 'b ((case - of / -) 10)

*-case1* :: ['a, 'b] => *case-syn* ((2- =>/ -) 10)  
 :: *case-syn* => *cases-syn* (-)

*-case2* :: [*case-syn*, *cases-syn*] => *cases-syn* (-/ | -)

**translations**

*THE*  $x. P == The (\%x. P)$   
*-Let* (*-binds*  $b$   $bs$ )  $e == -Let\ b\ (-Let\ bs\ e)$   
 $let\ x = a\ in\ e == Let\ a\ (\%x. e)$

**print-translation**  $\ll$ 

(\* To avoid eta-contraction of body: \*)  
 $\ll (The, fn [Abs\ abs] =>$   
 $\quad let\ val\ (x, t) = atomic-abs-tr'\ abs$   
 $\quad in\ Syntax.const\ -The\ \$\ x\ \$\ t\ end)]$   
 $\gg$

**syntax** (*xsymbols*)

*-case1* :: ['a, 'b] => *case-syn* ((2- =>/ -) 10)

**notation** (*xsymbols*)

*All* (**binder**  $\forall$  10) **and**  
*Ex* (**binder**  $\exists$  10) **and**

*Ex1* (**binder**  $\exists!$  10)

**notation** (*HTML output*)

*All* (**binder**  $\forall$  10) and

*Ex* (**binder**  $\exists$  10) and

*Ex1* (**binder**  $\exists!$  10)

**notation** (*HOL*)

*All* (**binder**  $!$  10) and

*Ex* (**binder**  $?$  10) and

*Ex1* (**binder**  $?!$  10)

### 1.1.3 Axioms and basic definitions

**axioms**

*refl*:  $t = (t::'a)$

*subst*:  $s = t \implies P\ s \implies P\ t$

*ext*:  $(!x::'a. (f\ x ::'b) = g\ x) \implies (\%x. f\ x) = (\%x. g\ x)$

— Extensionality is built into the meta-logic, and this rule expresses a related property. It is an eta-expanded version of the traditional rule, and similar to the ABS rule of HOL

*the-eq-trivial*:  $(THE\ x. x = a) = (a::'a)$

*impI*:  $(P \implies Q) \implies P \multimap Q$

*mp*:  $[P \multimap Q; P] \implies Q$

**defs**

*True-def*:  $True == ((\%x::bool. x) = (\%x. x))$

*All-def*:  $All(P) == (P = (\%x. True))$

*Ex-def*:  $Ex(P) == !Q. (!x. P\ x \multimap Q) \multimap Q$

*False-def*:  $False == (!P. P)$

*not-def*:  $\sim P == P \multimap False$

*and-def*:  $P \ \& \ Q == !R. (P \multimap Q \multimap R) \multimap R$

*or-def*:  $P \mid Q == !R. (P \multimap R) \multimap (Q \multimap R) \multimap R$

*Ex1-def*:  $Ex1(P) == ?\ x. P(x) \ \& \ (!\ y. P(y) \multimap y=x)$

**axioms**

*iff*:  $(P \multimap Q) \multimap (Q \multimap P) \multimap (P=Q)$

*True-or-False*:  $(P=True) \mid (P=False)$

**defs**

*Let-def*:  $Let\ s\ f == f(s)$

*if-def*:  $If\ P\ x\ y == THE\ z::'a. (P=True \multimap z=x) \ \& \ (P=False \multimap z=y)$

**finalconsts**

*op* =

*op* -->  
*The*

**axiomatization**

*undefined* :: 'a

**abbreviation** (*input*)

*arbitrary*  $\equiv$  *undefined*

**1.1.4 Generic classes and algebraic operations**

**class** *default* =

**fixes** *default* :: 'a

**class** *zero* =

**fixes** *zero* :: 'a (0)

**class** *one* =

**fixes** *one* :: 'a (1)

**hide** (**open**) *const zero one*

**class** *plus* =

**fixes** *plus* :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (**infixl** + 65)

**class** *minus* =

**fixes** *minus* :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (**infixl** - 65)

**class** *uminus* =

**fixes** *uminus* :: 'a  $\Rightarrow$  'a (- - [81] 80)

**class** *times* =

**fixes** *times* :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (**infixl** \* 70)

**class** *inverse* =

**fixes** *inverse* :: 'a  $\Rightarrow$  'a

**and** *divide* :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (**infixl** '/ 70)

**class** *abs* =

**fixes** *abs* :: 'a  $\Rightarrow$  'a

**begin**

**notation** (*xsymbols*)

*abs* (|·|)

**notation** (*HTML output*)

*abs* (|·|)

**end**

```

class sgn =
  fixes sgn :: 'a  $\Rightarrow$  'a

class ord =
  fixes less-eq :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
  and less :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
begin

notation
  less-eq (op  $\leq$ ) and
  less-eq ((-/  $\leq$  -) [51, 51] 50) and
  less (op  $<$ ) and
  less ((-/  $<$  -) [51, 51] 50)

notation (xsymbols)
  less-eq (op  $\leq$ ) and
  less-eq ((-/  $\leq$  -) [51, 51] 50)

notation (HTML output)
  less-eq (op  $\leq$ ) and
  less-eq ((-/  $\leq$  -) [51, 51] 50)

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

notation (input)
  greater-eq (infix  $\geq$  50)

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

end

syntax
  -index1 :: index (1)
translations
  (index)1  $\Rightarrow$  (index) $\diamond$ 

typed-print-translation  $\ll$ 
let
  fun tr' c = (c, fn show-sorts  $\Rightarrow$  fn T  $\Rightarrow$  fn ts  $\Rightarrow$ 
    if (not o null) ts orelse T = dummyT orelse not (! show-types) andalso can
    Term.dest-Type T then raise Match
    else Syntax.const Syntax.constrainC $ Syntax.const c $ Syntax.term-of-ty
    show-sorts T);
in map tr' [ @{const-syntax HOL.one}, @{const-syntax HOL.zero} ] end;

```

» — show types that are presumably too general

## 1.2 Fundamental rules

### 1.2.1 Equality

**lemma** *sym*:  $s = t \implies t = s$   
**by** (*erule subst*) (*rule refl*)

**lemma** *ssubst*:  $t = s \implies P\ s \implies P\ t$   
**by** (*drule sym*) (*erule subst*)

**lemma** *trans*:  $[| r=s; s=t |] \implies r=t$   
**by** (*erule subst*)

**lemma** *meta-eq-to-obj-eq*:  
**assumes** *meq*:  $A == B$   
**shows**  $A = B$   
**by** (*unfold meq*) (*rule refl*)

Useful with *erule* for proving equalities from known equalities.

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

For calculational reasoning:

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

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

### 1.2.2 Congruence rules for application

**lemma** *fun-cong*:  $(f::'a \Rightarrow 'b) = g \implies f(x)=g(x)$   
**apply** (*erule subst*)  
**apply** (*rule refl*)  
**done**

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

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

**apply** (*rule refl*)  
**done**

**lemma cong**:  $[[f = g; (x::'a) = y]] \implies f(x) = g(y)$   
**apply** (*erule subst*)  
**apply** (*rule refl*)  
**done**

### 1.2.3 Equality of booleans – iff

**lemma iffI**: **assumes**  $P \implies Q$  **and**  $Q \implies P$  **shows**  $P = Q$   
**by** (*iprover intro: iff [THEN mp, THEN mp] impI assms*)

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

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

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

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

**lemma iffE**:  
**assumes major**:  $P = Q$   
**and minor**:  $[[P \longrightarrow Q; Q \longrightarrow P]] \implies R$   
**shows**  $R$   
**by** (*iprover intro: minor impI major [THEN iffD2] major [THEN iffD1]*)

### 1.2.4 True

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

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

**lemma eqTrueE**:  $P = \text{True} \implies P$   
**by** (*erule iffD2*) (*rule TrueI*)

### 1.2.5 Universal quantifier

**lemma allI**: **assumes**  $!!x::'a. P(x)$  **shows**  $\text{ALL } x. P(x)$   
**unfolding** *All-def* **by** (*iprover intro: ext eqTrueI assms*)

**lemma spec**:  $\text{ALL } x::'a. P(x) \implies P(x)$   
**apply** (*unfold All-def*)  
**apply** (*rule eqTrueE*)



**apply** (*erule fun-cong*)  
**done**

**lemma** *allE*:  
**assumes** *major*:  $ALL\ x.\ P(x)$   
**and** *minor*:  $P(x) \implies R$   
**shows**  $R$   
**by** (*iprover intro: minor major [THEN spec]*)

**lemma** *all-dupE*:  
**assumes** *major*:  $ALL\ x.\ P(x)$   
**and** *minor*:  $[P(x); ALL\ x.\ P(x)] \implies R$   
**shows**  $R$   
**by** (*iprover intro: minor major major [THEN spec]*)

### 1.2.6 False

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

**lemma** *FalseE*:  $False \implies P$   
**apply** (*unfold False-def*)  
**apply** (*erule spec*)  
**done**

**lemma** *False-neq-True*:  $False = True \implies P$   
**by** (*erule eqTrueE [THEN FalseE]*)

### 1.2.7 Negation

**lemma** *notI*:  
**assumes**  $P \implies False$   
**shows**  $\sim P$   
**apply** (*unfold not-def*)  
**apply** (*iprover intro: impI assms*)  
**done**

**lemma** *False-not-True*:  $False \sim = True$   
**apply** (*rule notI*)  
**apply** (*erule False-neq-True*)  
**done**

**lemma** *True-not-False*:  $True \sim = False$   
**apply** (*rule notI*)  
**apply** (*drule sym*)  
**apply** (*erule False-neq-True*)  
**done**

**lemma** *notE*:  $[ \sim P; P ] \implies R$   
**apply** (*unfold not-def*)

```

apply (erule mp [THEN FalseE])
apply assumption
done

```

```

lemma notI2:  $(P \implies \neg Pa) \implies (P \implies Pa) \implies \neg P$ 
by (erule notE [THEN notI]) (erule meta-mp)

```

### 1.2.8 Implication

```

lemma impE:
  assumes  $P \longrightarrow Q$   $P$   $Q \implies R$ 
  shows  $R$ 
by (iprover intro: assms mp)

```

```

lemma rev-mp:  $[P; P \longrightarrow Q] \implies Q$ 
by (iprover intro: mp)

```

```

lemma contrapos-nn:
  assumes major:  $\sim Q$ 
  and minor:  $P \implies Q$ 
  shows  $\sim P$ 
by (iprover intro: notI minor major [THEN notE])

```

```

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

```

```

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

```

```

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

```

```

lemma rev-contrapos:
  assumes pq:  $P \implies Q$ 
  and nq:  $\sim Q$ 
  shows  $\sim P$ 
apply (rule nq [THEN contrapos-nn])
apply (erule pq)
done

```

### 1.2.9 Existential quantifier

```

lemma exI:  $P\ x \implies EX\ x::'a.\ P\ x$ 
apply (unfold Ex-def)

```

```

apply (iprover intro: allI allE impI mp)
done

```

```

lemma exE:
  assumes major: EX x::'a. P(x)
  and minor: !!x. P(x) ==> Q
  shows Q
apply (rule major [unfolded Ex-def, THEN spec, THEN mp])
apply (iprover intro: impI [THEN allI] minor)
done

```

### 1.2.10 Conjunction

```

lemma conjI: [| P; Q |] ==> P & Q
apply (unfold and-def)
apply (iprover intro: impI [THEN allI] mp)
done

```

```

lemma conjunct1: [| P & Q |] ==> P
apply (unfold and-def)
apply (iprover intro: impI dest: spec mp)
done

```

```

lemma conjunct2: [| P & Q |] ==> Q
apply (unfold and-def)
apply (iprover intro: impI dest: spec mp)
done

```

```

lemma conjE:
  assumes major: P & Q
  and minor: [| P; Q |] ==> R
  shows R
apply (rule minor)
apply (rule major [THEN conjunct1])
apply (rule major [THEN conjunct2])
done

```

```

lemma context-conjI:
  assumes P P ==> Q shows P & Q
by (iprover intro: conjI assms)

```

### 1.2.11 Disjunction

```

lemma disjI1: P ==> P | Q
apply (unfold or-def)
apply (iprover intro: allI impI mp)
done

```

```

lemma disjI2: Q ==> P | Q
apply (unfold or-def)

```

```

apply (iprover intro: allI impI mp)
done

```

```

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

```

### 1.2.12 Classical logic

```

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

```

```

lemmas ccontr = FalseE [THEN classical, standard]

```

```

lemma rev-notE:
  assumes premp: P
    and premnot:  $\sim R \implies \sim P$ 
  shows R
apply (rule ccontr)
apply (erule notE [OF premnot premp])
done

```

```

lemma notnotD:  $\sim\sim P \implies P$ 
apply (rule classical)
apply (erule notE)
apply assumption
done

```

```

lemma contrapos-pp:
  assumes p1: Q
    and p2:  $\sim P \implies \sim Q$ 
  shows P
by (iprover intro: classical p1 p2 notE)

```

### 1.2.13 Unique existence

```

lemma ex1I:

```

```

assumes  $P\ a\ !!x. P(x) ==> x=a$ 
shows  $EX!\ x. P(x)$ 
by (unfold Ex1-def, iprover intro: assms exI conjI allI impI)

```

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

```

lemma ex-ex1I:
  assumes ex-prem:  $EX\ x. P(x)$ 
    and eq:  $!!x\ y. [P(x); P(y)] ==> x=y$ 
  shows  $EX!\ x. P(x)$ 
by (iprover intro: ex-prem [THEN exE] ex1I eq)

```

```

lemma ex1E:
  assumes major:  $EX!\ x. P(x)$ 
    and minor:  $!!x. [P(x); ALL\ y. P(y) --> y=x] ==> R$ 
  shows  $R$ 
apply (rule major [unfolded Ex1-def, THEN exE])
apply (erule conjE)
apply (iprover intro: minor)
done

```

```

lemma ex1-implies-ex:  $EX!\ x. P\ x ==> EX\ x. P\ x$ 
apply (erule ex1E)
apply (rule exI)
apply assumption
done

```

#### 1.2.14 THE: definite description operator

```

lemma the-equality:
  assumes prema:  $P\ a$ 
    and premx:  $!!x. P\ x ==> x=a$ 
  shows  $(THE\ x. P\ x) = a$ 
apply (rule trans [OF - the-eq-trivial])
apply (rule-tac f = The in arg-cong)
apply (rule ext)
apply (rule iffI)
  apply (erule premx)
apply (erule ssubst, rule prema)
done

```

```

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

```

```

lemma theI':  $EX!\ x. P\ x ==> P\ (THE\ x. P\ x)$ 
apply (erule ex1E)
apply (erule theI)
apply (erule allE)

```

```

apply (erule mp)
apply assumption
done

```

```

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

```

```

lemma the1I2: assumes  $EX!\ x. P\ x \wedge x. P\ x \implies Q\ x$  shows  $Q\ (THE\ x. P\ x)$ 
by(iprover intro:assms(2) theI2[where  $P=P$  and  $Q=Q$ ] ex1E[OF assms(1)]
    elim:allE impE)

```

```

lemma the1-equality [elim?]: [ $EX!\ x. P\ x; P\ a$ ]  $\implies (THE\ x. P\ x) = a$ 
apply (rule the-equality)
apply assumption
apply (erule ex1E)
apply (erule all-dupE)
apply (erule mp)
apply assumption
apply (erule ssubst)
apply (erule allE)
apply (erule mp)
apply assumption
done

```

```

lemma the-sym-eq-trivial:  $(THE\ y. x=y) = x$ 
apply (rule the-equality)
apply (rule refl)
apply (erule sym)
done

```

### 1.2.15 Classical intro rules for disjunction and existential quantifiers

```

lemma disjCI:
  assumes  $\sim Q \implies P$  shows  $P \mid Q$ 
apply (rule classical)
apply (iprover intro: assms disjI1 disjI2 notI elim: notE)
done

```

```

lemma excluded-middle:  $\sim P \mid P$ 
by (iprover intro: disjCI)

```

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

```

lemma case-split [case-names True False]:
  assumes  $prem1: P \implies Q$ 

```

```

    and prem2:  $\sim P \implies Q$ 
  shows  $Q$ 
  apply (rule excluded-middle [THEN disjE])
  apply (erule prem2)
  apply (erule prem1)
done

```

```

lemma impCE:
  assumes major:  $P \dashv\dashv Q$ 
    and minor:  $\sim P \implies R \implies Q \implies R$ 
  shows  $R$ 
  apply (rule excluded-middle [of P, THEN disjE])
  apply (iprover intro: minor major [THEN mp])
done

```

```

lemma impCE':
  assumes major:  $P \dashv\dashv Q$ 
    and minor:  $Q \implies R \implies \sim P \implies R$ 
  shows  $R$ 
  apply (rule excluded-middle [of P, THEN disjE])
  apply (iprover intro: minor major [THEN mp])
done

```

```

lemma iffCE:
  assumes major:  $P = Q$ 
    and minor:  $[P; Q] \implies R \implies [\sim P; \sim Q] \implies R$ 
  shows  $R$ 
  apply (rule major [THEN iffE])
  apply (iprover intro: minor elim: impCE notE)
done

```

```

lemma exCI:
  assumes ALL  $x. \sim P(x) \implies P(a)$ 
  shows EX  $x. P(x)$ 
  apply (rule ccontr)
  apply (iprover intro: asms exI allI notI notE [of  $\exists x. P x$ ])
done

```

### 1.2.16 Intuitionistic Reasoning

```

lemma impE':
  assumes 1:  $P \dashv\dashv Q$ 
    and 2:  $Q \implies R$ 
    and 3:  $P \dashv\dashv Q \implies P$ 
  shows  $R$ 
proof -

```

```

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

```

```

lemma allE':
  assumes 1: ALL x. P x
    and 2: P x ==> ALL x. P x ==> Q
  shows Q
proof -
  from 1 have P x by (rule spec)
  from this and 1 show Q by (rule 2)
qed

```

```

lemma notE':
  assumes 1: ~ P
    and 2: ~ P ==> P
  shows R
proof -
  from 2 and 1 have P .
  with 1 show R by (rule notE)
qed

```

```

lemma TrueE: True ==> P ==> P .
lemma notFalseE: ~ False ==> P ==> P .

```

```

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

```

```

lemmas [trans] = trans
and [sym] = sym not-sym
and [Pure.elim?] = iffD1 iffD2 impE

```

```

use Tools/hologic.ML

```

### 1.2.17 Atomizing meta-level connectives

axiomatization where

*eq-reflection*:  $x = y \implies x \equiv y$

```

lemma atomize-all [atomize]: (!!x. P x) == Trueprop (ALL x. P x)

```

```

proof
  assume !!x. P x
  then show ALL x. P x ..
next
  assume ALL x. P x
  then show !!x. P x by (rule allE)

```



qed

**lemma** *atomize-imp* [*atomize*]:  $(A ==> B) == \text{Trueprop } (A --> B)$

**proof**

**assume**  $r: A ==> B$

**show**  $A --> B$  **by** (*rule impI*) (*rule r*)

**next**

**assume**  $A --> B$  **and**  $A$

**then show**  $B$  **by** (*rule mp*)

qed

**lemma** *atomize-not*:  $(A ==> \text{False}) == \text{Trueprop } (\sim A)$

**proof**

**assume**  $r: A ==> \text{False}$

**show**  $\sim A$  **by** (*rule notI*) (*rule r*)

**next**

**assume**  $\sim A$  **and**  $A$

**then show**  $\text{False}$  **by** (*rule notE*)

qed

**lemma** *atomize-eq* [*atomize*]:  $(x == y) == \text{Trueprop } (x = y)$

**proof**

**assume**  $x == y$

**show**  $x = y$  **by** (*unfold*  $\langle x == y \rangle$ ) (*rule refl*)

**next**

**assume**  $x = y$

**then show**  $x == y$  **by** (*rule eq-reflection*)

qed

**lemma** *atomize-conj* [*atomize*]:  $(A \&\&\& B) == \text{Trueprop } (A \& B)$

**proof**

**assume** *conj*:  $A \&\&\& B$

**show**  $A \& B$

**proof** (*rule conjI*)

**from** *conj* **show**  $A$  **by** (*rule conjunctionD1*)

**from** *conj* **show**  $B$  **by** (*rule conjunctionD2*)

  qed

**next**

**assume** *conj*:  $A \& B$

**show**  $A \&\&\& B$

**proof** –

**from** *conj* **show**  $A$  ..

**from** *conj* **show**  $B$  ..

  qed

qed

**lemmas** [*symmetric*, *rulify*] = *atomize-all atomize-imp*

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

### 1.2.18 Atomizing elimination rules

**setup** *AtomizeElim.setup*

**lemma** *atomize-exL*[*atomize-elim*]:  $(!!x. P\ x ==> Q) == ((EX\ x. P\ x) ==> Q)$   
**by** *rule iprover+*

**lemma** *atomize-conjL*[*atomize-elim*]:  $(A ==> B ==> C) == (A \& B ==> C)$   
**by** *rule iprover+*

**lemma** *atomize-disjL*[*atomize-elim*]:  $((A ==> C) ==> (B ==> C) ==> C)$   
 $== ((A \mid B ==> C) ==> C)$   
**by** *rule iprover+*

**lemma** *atomize-elimL*[*atomize-elim*]:  $(!!B. (A ==> B) ==> B) == \text{Trueprop } A$   
**..**

## 1.3 Package setup

### 1.3.1 Classical Reasoner setup

**lemma** *imp-elim*:  $P \dashv\dashv Q ==> (\sim R ==> P) ==> (Q ==> R) ==> R$   
**by** (*rule classical*) *iprover*

**lemma** *swap*:  $\sim P ==> (\sim R ==> P) ==> R$   
**by** (*rule classical*) *iprover*

**lemma** *thin-refl*:  
 $\bigwedge X. \llbracket x=x; \text{PROP } W \rrbracket \implies \text{PROP } W$ .

**ML**  $\ll$   
 $\text{structure Hypsubst} = \text{HypsubstFun}(\text{struct}$   
 $\quad \text{structure Simplifier} = \text{Simplifier}$   
 $\quad \text{val dest-eq} = \text{HOLogic.dest-eq}$   
 $\quad \text{val dest-Trueprop} = \text{HOLogic.dest-Trueprop}$   
 $\quad \text{val dest-imp} = \text{HOLogic.dest-imp}$   
 $\quad \text{val eq-reflection} = @\{\text{thm eq-reflection}\}$   
 $\quad \text{val rev-eq-reflection} = @\{\text{thm meta-eq-to-obj-eq}\}$   
 $\quad \text{val imp-intr} = @\{\text{thm impI}\}$   
 $\quad \text{val rev-mp} = @\{\text{thm rev-mp}\}$   
 $\quad \text{val subst} = @\{\text{thm subst}\}$   
 $\quad \text{val sym} = @\{\text{thm sym}\}$   
 $\quad \text{val thin-refl} = @\{\text{thm thin-refl}\};$   
 $\quad \text{val prop-subst} = @\{\text{lemma PROP } P\ t ==> \text{PROP prop } (x = t ==> \text{PROP } P$   
 $\quad x)$   
 $\quad \text{by (unfold prop-def) (drule eq-reflection, unfold)}\}$   
 $\text{end});$   
 $\text{open Hypsubst};$

```

structure Classical = ClassicalFun(
struct
  val imp-elim = @{thm imp-elim}
  val not-elim = @{thm notE}
  val swap = @{thm swap}
  val classical = @{thm classical}
  val sizef = Drule.size-of-thm
  val hyp-subst-tacs = [Hypsubst.hyp-subst-tac]
end);

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

ML-Antiquote.value claset
(Scan.succeed Classical.local-claset-of (ML-Context.the-local-context ()));

structure ResAtpset = NamedThmsFun(val name = atp val description = ATP
rules);

structure ResBlacklist = NamedThmsFun(val name = noatp val description = the-
orems blacklisted for ATP);
>>

ResBlacklist holds theorems blacklisted to sledgehammer. These theorems
typically produce clauses that are prolific (match too many equality or mem-
bership literals) and relate to seldom-used facts. Some duplicate other rules.

setup <<
let
  (*prevent substitution on bool*)
  fun hyp-subst-tac' i thm = if i <= Thm.nprems-of thm andalso
    Term.exists-Const (fn (op =, Type (-, [T, -])) => T <> Type (bool, [])) | - =>
false)
    (nth (Thm.premis-of thm) (i - 1)) then Hypsubst.hyp-subst-tac i thm else
no-tac thm;
in
  Hypsubst.hypsubst-setup
  #> ContextRules.addSWrapper (fn tac => hyp-subst-tac' ORELSE' tac)
  #> Classical.setup
  #> ResAtpset.setup
  #> ResBlacklist.setup
end
>>

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

```

```

and refl [intro!]

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

declare ex-ex1I [intro!]
  and allI [intro!]
  and the-equality [intro]
  and exI [intro]

declare exE [elim!]
  allE [elim]

ML ⟨⟨ val HOL-cs = @{claset} ⟩⟩

lemma contrapos-np:  $\sim Q \implies (\sim P \implies Q) \implies P$ 
  apply (erule swap)
  apply (erule (1) meta-mp)
  done

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

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

lemma alt-ex1E [elim!]:
  assumes major:  $\exists!x. P\ x$ 
  and prem:  $\bigwedge x. \llbracket P\ x; \forall y\ y'. P\ y \wedge P\ y' \longrightarrow y = y' \rrbracket \implies R$ 
  shows R
  apply (rule ex1E [OF major])
  apply (rule prem)
  apply (tactic ⟨⟨ ares-tac @{thms allI} 1 ⟩⟩)+
  apply (tactic ⟨⟨ etac (Classical.dup-elim @{thm allE}) 1 ⟩⟩)
  apply iprover
  done

ML ⟨⟨
  structure Blast = BlastFun
  (
    type claset = Classical.claset
    val equality-name = @{const-name op =}
    val not-name = @{const-name Not}
    val notE = @{thm notE}
  )
  ⟩⟩

```

```

val ccontr = @{thm ccontr}
val contr-tac = Classical.contr-tac
val dup-intr = Classical.dup-intr
val hyp-subst-tac = Hypsubst.blast-hyp-subst-tac
val rep-cs = Classical.rep-cs
val cla-modifiers = Classical.cla-modifiers
val cla-meth' = Classical.cla-meth'
);
val blast-tac = Blast.blast-tac;
>>

```

```

setup Blast.setup

```

### 1.3.2 Simplifier

**lemma** *eta-contract-eq*:  $(\%s. f\ s) = f \ ..$

**lemma** *simp-thms*:

```

shows not-not:  $(\sim \sim P) = P$ 
and Not-eq-iff:  $((\sim P) = (\sim Q)) = (P = Q)$ 
and
   $(P \sim = Q) = (P = (\sim Q))$ 
   $(P \mid \sim P) = \text{True} \quad (\sim P \mid P) = \text{True}$ 
   $(x = x) = \text{True}$ 
and not-True-eq-False:  $(\neg \text{True}) = \text{False}$ 
and not-False-eq-True:  $(\neg \text{False}) = \text{True}$ 
and
   $(\sim P) \sim = P \quad P \sim = (\sim P)$ 
   $(\text{True} = P) = P$ 
and eq-True:  $(P = \text{True}) = P$ 
and  $(\text{False} = P) = (\sim P)$ 
and eq-False:  $(P = \text{False}) = (\neg P)$ 
and
   $(\text{True} \dashrightarrow P) = P \quad (\text{False} \dashrightarrow P) = \text{True}$ 
   $(P \dashrightarrow \text{True}) = \text{True} \quad (P \dashrightarrow P) = \text{True}$ 
   $(P \dashrightarrow \text{False}) = (\sim P) \quad (P \dashrightarrow \sim P) = (\sim P)$ 
   $(P \ \& \ \text{True}) = P \quad (\text{True} \ \& \ P) = P$ 
   $(P \ \& \ \text{False}) = \text{False} \quad (\text{False} \ \& \ P) = \text{False}$ 
   $(P \ \& \ P) = P \quad (P \ \& \ (P \ \& \ Q)) = (P \ \& \ Q)$ 
   $(P \ \& \ \sim P) = \text{False} \quad (\sim P \ \& \ P) = \text{False}$ 
   $(P \mid \text{True}) = \text{True} \quad (\text{True} \mid P) = \text{True}$ 
   $(P \mid \text{False}) = P \quad (\text{False} \mid P) = P$ 
   $(P \mid P) = P \quad (P \mid (P \mid Q)) = (P \mid Q)$  and
   $(\text{ALL } x. P) = P \quad (\text{EX } x. P) = P \quad \text{EX } x. x=t \quad \text{EX } x. t=x$ 
  — needed for the one-point-rule quantifier simplification procs
  — essential for termination!! and
  !!P.  $(\text{EX } x. x=t \ \& \ P(x)) = P(t)$ 
  !!P.  $(\text{EX } x. t=x \ \& \ P(x)) = P(t)$ 
  !!P.  $(\text{ALL } x. x=t \dashrightarrow P(x)) = P(t)$ 

```

**!!P.**  $(\text{ALL } x. t=x \dashv\dashv P(x)) = P(t)$   
**by**  $(\text{blast}, \text{blast}, \text{blast}, \text{blast}, \text{blast}, \text{iprover+})$

**lemma** *disj-absorb*:  $(A \mid A) = A$   
**by** *blast*

**lemma** *disj-left-absorb*:  $(A \mid (A \mid B)) = (A \mid B)$   
**by** *blast*

**lemma** *conj-absorb*:  $(A \ \& \ A) = A$   
**by** *blast*

**lemma** *conj-left-absorb*:  $(A \ \& \ (A \ \& \ B)) = (A \ \& \ B)$   
**by** *blast*

**lemma** *eq-ac*:  
**shows** *eq-commute*:  $(a=b) = (b=a)$   
**and** *eq-left-commute*:  $(P=(Q=R)) = (Q=(P=R))$   
**and** *eq-assoc*:  $((P=Q)=R) = (P=(Q=R))$  **by**  $(\text{iprover}, \text{blast+})$   
**lemma** *neg-commute*:  $(a^\sim=b) = (b^\sim=a)$  **by** *iprover*

**lemma** *conj-comms*:  
**shows** *conj-commute*:  $(P \ \& \ Q) = (Q \ \& \ P)$   
**and** *conj-left-commute*:  $(P \ \& \ (Q \ \& \ R)) = (Q \ \& \ (P \ \& \ R))$  **by** *iprover+*  
**lemma** *conj-assoc*:  $((P \ \& \ Q) \ \& \ R) = (P \ \& \ (Q \ \& \ R))$  **by** *iprover*

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

**lemma** *disj-comms*:  
**shows** *disj-commute*:  $(P \mid Q) = (Q \mid P)$   
**and** *disj-left-commute*:  $(P \mid (Q \mid R)) = (Q \mid (P \mid R))$  **by** *iprover+*  
**lemma** *disj-assoc*:  $((P \mid Q) \mid R) = (P \mid (Q \mid R))$  **by** *iprover*

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

**lemma** *conj-disj-distribL*:  $(P \ \& \ (Q \mid R)) = (P \ \& \ Q \mid P \ \& \ R)$  **by** *iprover*

**lemma** *conj-disj-distribR*:  $((P \mid Q) \ \& \ R) = (P \ \& \ R \mid Q \ \& \ R)$  **by** *iprover*

**lemma** *disj-conj-distribL*:  $(P \mid (Q \ \& \ R)) = ((P \mid Q) \ \& \ (P \mid R))$  **by** *iprover*

**lemma** *disj-conj-distribR*:  $((P \ \& \ Q) \mid R) = ((P \ \& \ R) \ \& \ (Q \ \& \ R))$  **by** *iprover*

**lemma** *imp-conjR*:  $(P \dashv\dashv (Q \ \& \ R)) = ((P \dashv\dashv Q) \ \& \ (P \dashv\dashv R))$  **by** *iprover*

**lemma** *imp-conjL*:  $((P \ \& \ Q) \dashv\dashv R) = (P \dashv\dashv (Q \dashv\dashv R))$  **by** *iprover*

**lemma** *imp-disjL*:  $((P \mid Q) \dashv\dashv R) = ((P \dashv\dashv R) \ \& \ (Q \dashv\dashv R))$  **by** *iprover*

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

**lemma** *imp-disj-not1*:  $(P \dashv\dashv Q \mid R) = (\sim Q \dashv\dashv P \dashv\dashv R)$  **by** *blast*

**lemma** *imp-disj-not2*:  $(P \dashv\dashv Q \mid R) = (\sim R \dashv\dashv P \dashv\dashv Q)$  **by** *blast*

**lemma** *imp-disj1*:  $((P \dashv\vdash Q) \mid R) = (P \dashv\vdash Q \mid R)$  **by** *blast*

**lemma** *imp-disj2*:  $(Q \mid (P \dashv\vdash R)) = (P \dashv\vdash Q \mid R)$  **by** *blast*

**lemma** *imp-cong*:  $(P = P') \implies (P' \implies (Q = Q')) \implies ((P \dashv\vdash Q) = (P' \dashv\vdash Q'))$   
**by** *iprover*

**lemma** *de-Morgan-disj*:  $(\sim(P \mid Q)) = (\sim P \ \& \ \sim Q)$  **by** *iprover*

**lemma** *de-Morgan-conj*:  $(\sim(P \ \& \ Q)) = (\sim P \mid \sim Q)$  **by** *blast*

**lemma** *not-imp*:  $(\sim(P \dashv\vdash Q)) = (P \ \& \ \sim Q)$  **by** *blast*

**lemma** *not-iff*:  $(P \sim Q) = (P = (\sim Q))$  **by** *blast*

**lemma** *disj-not1*:  $(\sim P \mid Q) = (P \dashv\vdash Q)$  **by** *blast*

**lemma** *disj-not2*:  $(P \mid \sim Q) = (Q \dashv\vdash P)$  — changes orientation :-(  
**by** *blast*

**lemma** *imp-conv-disj*:  $(P \dashv\vdash Q) = ((\sim P) \mid Q)$  **by** *blast*

**lemma** *iff-conv-conj-imp*:  $(P = Q) = ((P \dashv\vdash Q) \ \& \ (Q \dashv\vdash P))$  **by** *iprover*

**lemma** *cases-simp*:  $((P \dashv\vdash Q) \ \& \ (\sim P \dashv\vdash Q)) = Q$

— Avoids duplication of subgoals after *split-if*, when the true and false  
 — cases boil down to the same thing.

**by** *blast*

**lemma** *not-all*:  $(\sim (! x. P(x))) = (? x. \sim P(x))$  **by** *blast*

**lemma** *imp-all*:  $((! x. P x) \dashv\vdash Q) = (? x. P x \dashv\vdash Q)$  **by** *blast*

**lemma** *not-ex*:  $(\sim (? x. P(x))) = (! x. \sim P(x))$  **by** *iprover*

**lemma** *imp-ex*:  $((? x. P x) \dashv\vdash Q) = (! x. P x \dashv\vdash Q)$  **by** *iprover*

**lemma** *all-not-ex*:  $(ALL x. P x) = (\sim (EX x. \sim P x))$  **by** *blast*

**declare** *All-def* [*noatp*]

**lemma** *ex-disj-distrib*:  $(? x. P(x) \mid Q(x)) = ((? x. P(x)) \mid (? x. Q(x)))$  **by** *iprover*

**lemma** *all-conj-distrib*:  $(! x. P(x) \ \& \ Q(x)) = ((! x. P(x)) \ \& \ (! x. Q(x)))$  **by** *iprover*

The  $\&$  congruence rule: not included by default! May slow rewrite proofs down by as much as 50%

**lemma** *conj-cong*:

$(P = P') \implies (P' \implies (Q = Q')) \implies ((P \ \& \ Q) = (P' \ \& \ Q'))$

**by** *iprover*

**lemma** *rev-conj-cong*:

$(Q = Q') \implies (Q' \implies (P = P')) \implies ((P \ \& \ Q) = (P' \ \& \ Q'))$

**by** *iprover*

The  $\mid$  congruence rule: not included by default!

**lemma** *disj-cong*:

$(P = P') \implies (\sim P' \implies (Q = Q')) \implies ((P \mid Q) = (P' \mid Q'))$

**by** *blast*

if-then-else rules

**lemma** *if-True*:  $(\text{if } \text{True} \text{ then } x \text{ else } y) = x$   
**by** (*unfold if-def*) *blast*

**lemma** *if-False*:  $(\text{if } \text{False} \text{ then } x \text{ else } y) = y$   
**by** (*unfold if-def*) *blast*

**lemma** *if-P*:  $P \implies (\text{if } P \text{ then } x \text{ else } y) = x$   
**by** (*unfold if-def*) *blast*

**lemma** *if-not-P*:  $\sim P \implies (\text{if } P \text{ then } x \text{ else } y) = y$   
**by** (*unfold if-def*) *blast*

**lemma** *split-if*:  $P (\text{if } Q \text{ then } x \text{ else } y) = ((Q \longrightarrow P(x)) \ \& \ (\sim Q \longrightarrow P(y)))$   
**apply** (*rule case-split* [*of Q*])  
**apply** (*simplesubst if-P*)  
**prefer** 3 **apply** (*simplesubst if-not-P*, *blast*+) **done**

**lemma** *split-if-asm*:  $P (\text{if } Q \text{ then } x \text{ else } y) = (\sim((Q \ \& \ \sim P \ x) \mid (\sim Q \ \& \ \sim P \ y)))$   
**by** (*simplesubst split-if*, *blast*)

**lemmas** *if-splits* [*noatp*] = *split-if split-if-asm*

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

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

**lemma** *if-bool-eq-conj*:  $(\text{if } P \text{ then } Q \text{ else } R) = ((P \longrightarrow Q) \ \& \ (\sim P \longrightarrow R))$   
— This form is useful for expanding *ifs* on the RIGHT of the  $\implies$  symbol.  
**by** (*rule split-if*)

**lemma** *if-bool-eq-disj*:  $(\text{if } P \text{ then } Q \text{ else } R) = ((P \ \& \ Q) \mid (\sim P \ \& \ R))$   
— And this form is useful for expanding *ifs* on the LEFT.  
**apply** (*simplesubst split-if*, *blast*) **done**

**lemma** *Eq-TrueI*:  $P \implies P == \text{True}$  **by** (*unfold atomize-eq*) *iprover*

**lemma** *Eq-FalseI*:  $\sim P \implies P == \text{False}$  **by** (*unfold atomize-eq*) *iprover*

let rules for *simproc*

**lemma** *Let-folded*:  $f \ x \equiv g \ x \implies \text{Let } x \ f \equiv \text{Let } x \ g$   
**by** (*unfold Let-def*)



**lemma** *Let-unfold*:  $f\ x \equiv g \implies \text{Let } x\ f \equiv g$   
**by** (*unfold Let-def*)

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

**constdefs**

*simp-implies* :: [*prop*, *prop*] => *prop* (**infixr** =*simp=>* 1)  
[*code del*]: *simp-implies*  $\equiv$  *op* ==>

**lemma** *simp-impliesI*:

**assumes** *PQ*: (*PROP P*  $\implies$  *PROP Q*)  
**shows** *PROP P* =*simp=>* *PROP Q*  
**apply** (*unfold simp-implies-def*)  
**apply** (*rule PQ*)  
**apply** *assumption*  
**done**

**lemma** *simp-impliesE*:

**assumes** *PQ*: *PROP P* =*simp=>* *PROP Q*  
**and** *P*: *PROP P*  
**and** *QR*: *PROP Q*  $\implies$  *PROP R*  
**shows** *PROP R*  
**apply** (*rule QR*)  
**apply** (*rule PQ* [*unfolded simp-implies-def*])  
**apply** (*rule P*)  
**done**

**lemma** *simp-implies-cong*:

**assumes** *PP'*: *PROP P* == *PROP P'*  
**and** *P'QQ'*: *PROP P'* ==> (*PROP Q* == *PROP Q'*)  
**shows** (*PROP P* =*simp=>* *PROP Q*) == (*PROP P'* =*simp=>* *PROP Q'*)  
**proof** (*unfold simp-implies-def*, *rule equal-intr-rule*)  
**assume** *PQ*: *PROP P*  $\implies$  *PROP Q*  
**and** *P'*: *PROP P'*  
**from** *PP'* [*symmetric*] **and** *P'* **have** *PROP P*  
**by** (*rule equal-elim-rule1*)  
**then have** *PROP Q* **by** (*rule PQ*)  
**with** *P'QQ'* [*OF P'*] **show** *PROP Q'* **by** (*rule equal-elim-rule1*)  
**next**  
**assume** *P'Q'*: *PROP P'*  $\implies$  *PROP Q'*  
**and** *P*: *PROP P*  
**from** *PP'* **and** *P* **have** *P'*: *PROP P'* **by** (*rule equal-elim-rule1*)  
**then have** *PROP Q'* **by** (*rule P'Q'*)  
**with** *P'QQ'* [*OF P'*, *symmetric*] **show** *PROP Q*  
**by** (*rule equal-elim-rule1*)  
**qed**

**lemma** *uncurry*:

**assumes**  $P \longrightarrow Q \longrightarrow R$

**shows**  $P \wedge Q \longrightarrow R$   
**using** *assms* **by** *blast*

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

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

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

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

**use** *Tools/simpdata.ML*  
**ML**  $\ll$  *open Simpdata*  $\gg$

**setup**  $\ll$   
*Simplifier.method-setup Splitter.split-modifiers*  
 $\#>$  *Simplifier.map-simpset (K Simpdata.simpset-simprocs)*  
 $\#>$  *Splitter.setup*  
 $\#>$  *clasimp-setup*  
 $\#>$  *EqSubst.setup*  
 $\gg$

Simproc for proving  $(y = x) == \text{False}$  from premise  $\sim(x = y)$ :

**simproc-setup** *neq*  $(x = y) = \ll$  *fn*  $- =>$   
*let*  
 $\text{val neq-to-EQ-False} = @\{thm\ not\ sym\} RS @\{thm\ Eq\ FalseI\};$   
 $\text{fun is-neq eq lhs rhs thm} =$   
 $\text{(case Thm.prop-of thm of}$   
 $\text{ - } \$ (Not\ \$ (eq'\ \$ l'\ \$ r')) =>$   
 $\text{ Not} = HOLogic.Not\ andalso\ eq' = eq\ andalso$   
 $\text{ r' aconv lhs andalso l' aconv rhs}$   
 $\text{ | } - => \text{false});$   
 $\text{fun proc ss ct} =$   
 $\text{(case Thm.term-of ct of}$   
 $\text{ eq } \$ lhs\ \$ rhs =>$   
 $\text{ (case find-first (is-neq eq lhs rhs) (Simplifier.premis-of-ss ss) of}$   
 $\text{ SOME thm } => \text{SOME (thm RS neq-to-EQ-False)}$   
 $\text{ | NONE } => \text{NONE})}$   
 $\text{ | } - => \text{NONE});$

```
in proc end;
>>
```

```
simproc-setup let-simp (Let x f) = <<
let
  val (f-Let-unfold, x-Let-unfold) =
    let val [(- $ (f $ x) $ -)] = prems-of @{thm Let-unfold}
    in (cterm-of @{theory} f, cterm-of @{theory} x) end
  val (f-Let-folded, x-Let-folded) =
    let val [(- $ (f $ x) $ -)] = prems-of @{thm Let-folded}
    in (cterm-of @{theory} f, cterm-of @{theory} x) end;
  val g-Let-folded =
    let val [(- $ - $ (g $ -))] = prems-of @{thm Let-folded}
    in cterm-of @{theory} g end;
  fun count-loose (Bound i) k = if i >= k then 1 else 0
    | count-loose (s $ t) k = count-loose s k + count-loose t k
    | count-loose (Abs (-, -, t)) k = count-loose t (k + 1)
    | count-loose - = 0;
  fun is-trivial-let (Const (@{const-name Let}, -) $ x $ t) =
    case t
    of Abs (-, -, t') => count-loose t' 0 <= 1
    | - => true;
in fn - => fn ss => fn ct => if is-trivial-let (Thm.term-of ct)
then SOME @{thm Let-def} (*no or one occurrence of bound variable*)
else let (*Norbert Schirmer's case*)
  val ctxt = Simplifier.the-context ss;
  val thy = ProofContext.theory-of ctxt;
  val t = Thm.term-of ct;
  val ([t'], ctxt') = Variable.import-terms false [t] ctxt;
in Option.map (hd o Variable.export ctxt' ctxt o single)
  (case t' of Const (@{const-name Let}, -) $ x $ f => (* x and f are already in
normal form *)
  if is-Free x orelse is-Bound x orelse is-Const x
  then SOME @{thm Let-def}
  else
  let
    val n = case f of (Abs (x, -, -)) => x | - => x;
    val cx = cterm-of thy x;
    val {T = xT, ...} = rep-cterm cx;
    val cf = cterm-of thy f;
    val fx-g = Simplifier.rewrite ss (Thm.capply cf cx);
    val (- $ - $ g) = prop-of fx-g;
    val g' = abstract-over (x, g);
  in (if (g aconv g')
    then
      let
        val rl =
          cterm-instantiate [(f-Let-unfold, cf), (x-Let-unfold, cx)] @{thm
Let-unfold};
```

```

      in SOME (rl OF [fx-g]) end
      else if Term.betapply (f, x) aconv g then NONE (*avoid identity
conversion*)
      else let
        val abs-g' = Abs (n,xT,g');
        val g'x = abs-g'$x;
        val g-g'x = symmetric (beta-conversion false (cterm-of thy g'x));
        val rl = cterm-instantiate
          [(f-Let-folded, cterm-of thy f), (x-Let-folded, cx),
           (g-Let-folded, cterm-of thy abs-g')]
          @ {thm Let-folded};
        in SOME (rl OF [transitive fx-g g-g'x])
        end)
      end
    | - => NONE)
  end
end >>

```

**lemma** *True-implies-equals*:  $(\text{True} \implies \text{PROP } P) \equiv \text{PROP } P$

**proof**

**assume**  $\text{True} \implies \text{PROP } P$

**from** *this* [OF TrueI] **show**  $\text{PROP } P$  .

**next**

**assume**  $\text{PROP } P$

**then show**  $\text{PROP } P$  .

**qed**

**lemma** *ex-simps*:

!!P Q.  $(\text{EX } x. P \ x \ \& \ Q) = ((\text{EX } x. P \ x) \ \& \ Q)$

!!P Q.  $(\text{EX } x. P \ \& \ Q \ x) = (P \ \& \ (\text{EX } x. Q \ x))$

!!P Q.  $(\text{EX } x. P \ x \ | \ Q) = ((\text{EX } x. P \ x) \ | \ Q)$

!!P Q.  $(\text{EX } x. P \ | \ Q \ x) = (P \ | \ (\text{EX } x. Q \ x))$

!!P Q.  $(\text{EX } x. P \ x \ \longrightarrow \ Q) = ((\text{ALL } x. P \ x) \ \longrightarrow \ Q)$

!!P Q.  $(\text{EX } x. P \ \longrightarrow \ Q \ x) = (P \ \longrightarrow \ (\text{EX } x. Q \ x))$

— Miniscoping: pushing in existential quantifiers.

**by** (*iprover* | *blast*)+

**lemma** *all-simps*:

!!P Q.  $(\text{ALL } x. P \ x \ \& \ Q) = ((\text{ALL } x. P \ x) \ \& \ Q)$

!!P Q.  $(\text{ALL } x. P \ \& \ Q \ x) = (P \ \& \ (\text{ALL } x. Q \ x))$

!!P Q.  $(\text{ALL } x. P \ x \ | \ Q) = ((\text{ALL } x. P \ x) \ | \ Q)$

!!P Q.  $(\text{ALL } x. P \ | \ Q \ x) = (P \ | \ (\text{ALL } x. Q \ x))$

!!P Q.  $(\text{ALL } x. P \ x \ \longrightarrow \ Q) = ((\text{EX } x. P \ x) \ \longrightarrow \ Q)$

!!P Q.  $(\text{ALL } x. P \ \longrightarrow \ Q \ x) = (P \ \longrightarrow \ (\text{ALL } x. Q \ x))$

— Miniscoping: pushing in universal quantifiers.

**by** (*iprover* | *blast*)+

**lemmas** [*simp*] =

*triv-forall-equality*

*True-implies-equals*  
*if-True*  
*if-False*  
*if-cancel*  
*if-eq-cancel*  
*imp-disjL*

*conj-assoc*  
*disj-assoc*  
*de-Morgan-conj*  
*de-Morgan-disj*  
*imp-disj1*  
*imp-disj2*  
*not-imp*  
*disj-not1*  
*not-all*  
*not-ex*  
*cases-simp*  
*the-eq-trivial*  
*the-sym-eq-trivial*  
*ex-simps*  
*all-simps*  
*simp-thms*

**lemmas** [*cong*] = *imp-cong simp-implies-cong*  
**lemmas** [*split*] = *split-if*

**ML**  $\ll \text{val HOL-ss} = @\{\text{simpset}\} \gg$

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

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

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

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

Prevents simplification of *t*: much faster

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

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

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

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

This lemma restricts the effect of the rewrite rule  $u=v$  to the left-hand side of an equality. Used in  $\{Integ, Real\}/simproc.ML$

```
lemma restrict-to-left:
  assumes  $x = y$ 
  shows  $(x = z) = (y = z)$ 
  using assms by simp
```

### 1.3.3 Generic cases and induction

Rule projections:

```
ML <<
  structure ProjectRule = ProjectRuleFun
  (
    val conjunct1 = @{thm conjunct1}
    val conjunct2 = @{thm conjunct2}
    val mp = @{thm mp}
  )
  >>
```

```
constdefs
  induct-forall where induct-forall  $P == \forall x. P x$ 
  induct-implies where induct-implies  $A B == A \longrightarrow B$ 
  induct-equal where induct-equal  $x y == x = y$ 
  induct-conj where induct-conj  $A B == A \wedge B$ 
```

```
lemma induct-forall-eq:  $(!!x. P x) == \text{Trueprop } (\text{induct-forall } (\lambda x. P x))$ 
  by (unfold atomize-all induct-forall-def)
```

```
lemma induct-implies-eq:  $(A ==> B) == \text{Trueprop } (\text{induct-implies } A B)$ 
  by (unfold atomize-imp induct-implies-def)
```

```
lemma induct-equal-eq:  $(x == y) == \text{Trueprop } (\text{induct-equal } x y)$ 
  by (unfold atomize-eq induct-equal-def)
```

```
lemma induct-conj-eq:  $(A \&\&\& B) == \text{Trueprop } (\text{induct-conj } A B)$ 
  by (unfold atomize-conj induct-conj-def)
```

```
lemmas induct-atomize = induct-forall-eq induct-implies-eq induct-equal-eq induct-conj-eq
```

```

lemmas induct-rulify [symmetric, standard] = induct-atomize
lemmas induct-rulify-fallback =
  induct-forall-def induct-implies-def induct-equal-def induct-conj-def

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

lemma induct-implies-conj: induct-implies C (induct-conj A B) =
  induct-conj (induct-implies C A) (induct-implies C B)
by (unfold induct-implies-def induct-conj-def) iprover

lemma induct-conj-curry: (induct-conj A B ==> PROP C) == (A ==> B ==>
PROP C)
proof
  assume r: induct-conj A B ==> PROP C and A B
  show PROP C by (rule r) (simp add: induct-conj-def  $\langle A \rangle \langle B \rangle$ )
next
  assume r: A ==> B ==> PROP C and induct-conj A B
  show PROP C by (rule r) (simp-all add: induct-conj A B [unfolded induct-conj-def])
qed

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

hide const induct-forall induct-implies induct-equal induct-conj

Method setup.

ML <<
structure Induct = InductFun
(
  val cases-default = @{thm case-split}
  val atomize = @{thms induct-atomize}
  val rulify = @{thms induct-rulify}
  val rulify-fallback = @{thms induct-rulify-fallback}
)
>>

setup Induct.setup

use ~/src/Tools/induct-tacs.ML
setup InductTacs.setup

```

### 1.3.4 Coherent logic

```

ML <<
structure Coherent = CoherentFun
(
  val atomize-elimL = @{thm atomize-elimL}

```

```

val atomize-exL = @{thm atomize-exL}
val atomize-conjL = @{thm atomize-conjL}
val atomize-disjL = @{thm atomize-disjL}
val operator-names =
  [@{const-name op |}, @{const-name op &}, @{const-name Ex}]
);
>>

```

**setup** *Coherent.setup*

#### 1.4 Other simple lemmas and lemma duplicates

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

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

**lemma** *ex1-eq* [*iff*]: *EX! x. x = t EX! x. t = x*  
**by** *blast+*

**lemma** *choice-eq*:  $(\text{ALL } x. \text{EX! } y. P \ x \ y) = (\text{EX! } f. \text{ALL } x. P \ x \ (f \ x))$   
**apply** (*rule iffI*)  
**apply** (*rule-tac a = %x. THE y. P x y in ex1I*)  
**apply** (*fast dest!: theI'*)  
**apply** (*fast intro: ext the1-equality [symmetric]*)  
**apply** (*erule ex1E*)  
**apply** (*rule allI*)  
**apply** (*rule ex1I*)  
**apply** (*erule spec*)  
**apply** (*erule-tac x = %z. if z = x then y else f z in allE*)  
**apply** (*erule impE*)  
**apply** (*rule allI*)  
**apply** (*case-tac xa = x*)  
**apply** (*drule-tac [3] x = x in fun-cong, simp-all*)  
**done**

**lemma** *mk-left-commute*:  
**fixes** *f* (**infix**  $\otimes$  60)  
**assumes** *a*:  $\bigwedge x \ y \ z. (x \otimes y) \otimes z = x \otimes (y \otimes z)$  **and**  
*c*:  $\bigwedge x \ y. x \otimes y = y \otimes x$   
**shows**  $x \otimes (y \otimes z) = y \otimes (x \otimes z)$   
**by** (*rule trans [OF trans [OF c a] arg-cong [OF c, of f y]]*)

**lemmas** *eq-sym-conv* = *eq-commute*

**lemma** *nnf-simps*:  
 $(\neg(P \wedge Q)) = (\neg P \vee \neg Q) \ (\neg(P \vee Q)) = (\neg P \wedge \neg Q) \ (P \longrightarrow Q) = (\neg P \vee Q)$



$$(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg Q)) \quad (\neg(P = Q)) = ((P \wedge \neg Q) \vee (\neg P \wedge Q))$$

$$(\neg \neg(P)) = P$$

by blast+

## 1.5 Basic ML bindings

```

ML <<
val FalseE = @{thm FalseE}
val Let-def = @{thm Let-def}
val TrueI = @{thm TrueI}
val allE = @{thm allE}
val allI = @{thm allI}
val all-dupE = @{thm all-dupE}
val arg-cong = @{thm arg-cong}
val box-equals = @{thm box-equals}
val ccontr = @{thm ccontr}
val classical = @{thm classical}
val conjE = @{thm conjE}
val conjI = @{thm conjI}
val conjunct1 = @{thm conjunct1}
val conjunct2 = @{thm conjunct2}
val disjCI = @{thm disjCI}
val disjE = @{thm disjE}
val disjI1 = @{thm disjI1}
val disjI2 = @{thm disjI2}
val eq-reflection = @{thm eq-reflection}
val ex1E = @{thm ex1E}
val ex1I = @{thm ex1I}
val ex1-implies-ex = @{thm ex1-implies-ex}
val exE = @{thm exE}
val exI = @{thm exI}
val excluded-middle = @{thm excluded-middle}
val ext = @{thm ext}
val fun-cong = @{thm fun-cong}
val iffD1 = @{thm iffD1}
val iffD2 = @{thm iffD2}
val iffI = @{thm iffI}
val impE = @{thm impE}
val impI = @{thm impI}
val meta-eq-to-obj-eq = @{thm meta-eq-to-obj-eq}
val mp = @{thm mp}
val notE = @{thm notE}
val notI = @{thm notI}
val not-all = @{thm not-all}
val not-ex = @{thm not-ex}
val not-iff = @{thm not-iff}
val not-not = @{thm not-not}
val not-sym = @{thm not-sym}
val refl = @{thm refl}

```

```

val rev-mp = @{thm rev-mp}
val spec = @{thm spec}
val ssubst = @{thm ssubst}
val subst = @{thm subst}
val sym = @{thm sym}
val trans = @{thm trans}
>>

```

## 1.6 Code generator basics – see further theory *Code-Setup*

Equality

```

class eq =
  fixes eq :: 'a ⇒ 'a ⇒ bool
  assumes eq-equals: eq x y ⟷ x = y
begin

```

```

lemma eq: eq = (op =)
  by (rule ext eq-equals)+

```

```

lemma eq-refl: eq x x ⟷ True
  unfolding eq by rule+

```

end

Module setup

```

use Tools/recfun-codegen.ML

```

```

setup <<
  Code-ML.setup
  #> Code-Haskell.setup
  #> Nbe.setup
  #> Codegen.setup
  #> RecfunCodegen.setup
>>

```

## 1.7 Nitpick hooks

This will be relocated once Nitpick is moved to HOL.

```

ML <<
  structure Nitpick-Const-Def-Thms = NamedThmsFun
  (
    val name = nitpick-const-def
    val description = alternative definitions of constants as needed by Nitpick
  )
  structure Nitpick-Const-Simp-Thms = NamedThmsFun
  (
    val name = nitpick-const-simp
    val description = equational specification of constants as needed by Nitpick
  )

```

```

)
structure Nitpick-Const-Psimp-Thms = NamedThmsFun
(
  val name = nitpick-const-psimp
  val description = partial equational specification of constants as needed by Nitpick
)
structure Nitpick-Ind-Intro-Thms = NamedThmsFun
(
  val name = nitpick-ind-intro
  val description = introduction rules for (co)inductive predicates as needed by
Nitpick
)
)
)
setup << Nitpick-Const-Def-Thms.setup
  #> Nitpick-Const-Simp-Thms.setup
  #> Nitpick-Const-Psimp-Thms.setup
  #> Nitpick-Ind-Intro-Thms.setup >>

```

## 1.8 Legacy tactics and ML bindings

```

ML <<
fun strip-tac i = REPEAT (resolve-tac [impI, allI] i);

(* combination of (spec RS spec RS ...(j times) ... spec RS mp) *)
local
  fun wrong-prem (Const (All, -) $ (Abs (-, -, t))) = wrong-prem t
    | wrong-prem (Bound -) = true
    | wrong-prem - = false;
  val filter-right = filter (not o wrong-prem o HOLogic.dest-Trueprop o hd o Thm.premsof);
in
  fun smp i = funpow i (fn m => filter-right ([spec] RL m)) ([mp]);
  fun smp-tac j = EVERY [dresolve-tac (smp j), atac];
end;

val all-conj-distrib = thm all-conj-distrib;
val all-simps = thms all-simps;
val atomize-not = thm atomize-not;
val case-split = thm case-split;
val cases-simp = thm cases-simp;
val choice-eq = thm choice-eq
val cong = thm cong
val conj-comms = thms conj-comms;
val conj-cong = thm conj-cong;
val de-Morgan-conj = thm de-Morgan-conj;
val de-Morgan-disj = thm de-Morgan-disj;
val disj-assoc = thm disj-assoc;
val disj-comms = thms disj-comms;
val disj-cong = thm disj-cong;
val eq-ac = thms eq-ac;

```

```

val eq-cong2 = thm eq-cong2
val Eq-FalseI = thm Eq-FalseI;
val Eq-TrueI = thm Eq-TrueI;
val Ex1-def = thm Ex1-def
val ex-disj-distrib = thm ex-disj-distrib;
val ex-simps = thms ex-simps;
val if-cancel = thm if-cancel;
val if-eq-cancel = thm if-eq-cancel;
val if-False = thm if-False;
val iff-conv-conj-imp = thm iff-conv-conj-imp;
val iff = thm iff
val if-splits = thms if-splits;
val if-True = thm if-True;
val if-weak-cong = thm if-weak-cong
val imp-all = thm imp-all;
val imp-cong = thm imp-cong;
val imp-conjL = thm imp-conjL;
val imp-conjR = thm imp-conjR;
val imp-conv-disj = thm imp-conv-disj;
val simp-implies-def = thm simp-implies-def;
val simp-thms = thms simp-thms;
val split-if = thm split-if;
val the1-equality = thm the1-equality
val theI = thm theI
val theI' = thm theI'
val True-implies-equals = thm True-implies-equals;
val nnf-conv = Simplifier.rewrite (HOL-basic-ss addsimps simp-thms @ @{thms
nnf-simps})

>>

end

```

## 2 Code-Setup: Setup of code generators and related tools

```

theory Code-Setup
imports HOL
begin

```

### 2.1 Generic code generator foundation

Datatypes

```
code-datatype True False
```

```
code-datatype TYPE('a::{})
```

**code-datatype** *Trueprop prop*

Code equations

**lemma** [*code*]:  
**shows**  $(True \implies PROP\ P) \equiv PROP\ P$   
**and**  $(False \implies Q) \equiv Trueprop\ True$   
**and**  $(PROP\ P \implies True) \equiv Trueprop\ True$   
**and**  $(Q \implies False) \equiv Trueprop\ (\neg Q)$  **by** (*auto intro! equal-intr-rule*)

**lemma** [*code*]:  
**shows**  $False \wedge x \longleftrightarrow False$   
**and**  $True \wedge x \longleftrightarrow x$   
**and**  $x \wedge False \longleftrightarrow False$   
**and**  $x \wedge True \longleftrightarrow x$  **by** *simp-all*

**lemma** [*code*]:  
**shows**  $False \vee x \longleftrightarrow x$   
**and**  $True \vee x \longleftrightarrow True$   
**and**  $x \vee False \longleftrightarrow x$   
**and**  $x \vee True \longleftrightarrow True$  **by** *simp-all*

**lemma** [*code*]:  
**shows**  $\neg True \longleftrightarrow False$   
**and**  $\neg False \longleftrightarrow True$  **by** (*rule HOL.simp-thms*)<sup>+</sup>

**lemmas** [*code*] = *Let-def if-True if-False*

**lemmas** [*code, code unfold, symmetric, code post*] = *imp-conv-disj*

Equality

**context** *eq*  
**begin**

**lemma** *equals-eq* [*code inline, code*]:  $op = \equiv eq$   
**by** (*rule eq-reflection*) (*rule ext, rule ext, rule sym, rule eq-equals*)

**declare** *eq* [*code unfold, code inline del*]

**declare** *equals-eq* [*symmetric, code post*]

**end**

**declare** *simp-thms*(6) [*code nbe*]

**hide** (**open**) *const eq*  
**hide** *const eq*

**setup**  $\ll$   
*Code-Unit.add-const-alias* @{*thm equals-eq*}

»

Cases

**lemma** *Let-case-cert*:

**assumes**  $CASE \equiv (\lambda x. \text{Let } x \text{ } f)$

**shows**  $CASE \ x \equiv f \ x$

**using** *assms* **by** *simp-all*

**lemma** *If-case-cert*:

**assumes**  $CASE \equiv (\lambda b. \text{If } b \text{ } f \text{ } g)$

**shows**  $(CASE \ \text{True} \equiv f) \ \&\&\& \ (CASE \ \text{False} \equiv g)$

**using** *assms* **by** *simp-all*

**setup** «

*Code.add-case* @{*thm Let-case-cert*}

#> *Code.add-case* @{*thm If-case-cert*}

#> *Code.add-undefined* @{*const-name undefined*}

»

**code-abort** *undefined*

## 2.2 Generic code generator preprocessor

**setup** «

*Code.map-pre* (*K HOL-basic-ss*)

#> *Code.map-post* (*K HOL-basic-ss*)

»

## 2.3 Generic code generator target languages

type bool

**code-type** *bool*

(*SML bool*)

(*OCaml bool*)

(*Haskell Bool*)

**code-const** *True and False and Not and op & and op | and If*

(*SML true and false and not*

**and** **infixl** 1 *andalso and infixl* 0 *orelse*

**and** **!**(*if* (-)/ *then* (-)/ *else* (-)))

(*OCaml true and false and not*

**and** **infixl** 4 *&& and infixl* 2 *||*

**and** **!**(*if* (-)/ *then* (-)/ *else* (-)))

(*Haskell True and False and not*

**and** **infixl** 3 *&& and infixl* 2 *||*

**and** **!**(*if* (-)/ *then* (-)/ *else* (-)))

**code-reserved** *SML*

*bool true false not*

**code-reserved** *OCaml*

*bool not*

using built-in Haskell equality

**code-class** *eq*

*(Haskell Eq)*

**code-const** *eq-class.eq*

*(Haskell infixl 4 ==)*

**code-const** *op =*

*(Haskell infixl 4 ==)*

undefined

**code-const** *undefined*

*(SML !(raise/ Fail/ undefined))*

*(OCaml failwith/ undefined)*

*(Haskell error/ undefined)*

## 2.4 SML code generator setup

**types-code**

*bool (bool)*

**attach** *(term-of)*  $\ll$

*fun term-of-bool b = if b then HLogic.true-const else HLogic.false-const;*

$\gg$

**attach** *(test)*  $\ll$

*fun gen-bool i =*

*let val b = one-of [false, true]*

*in (b, fn () => term-of-bool b) end;*

$\gg$

*prop (bool)*

**attach** *(term-of)*  $\ll$

*fun term-of-prop b =*

*HLogic.mk-Trueprop (if b then HLogic.true-const else HLogic.false-const);*

$\gg$

**consts-code**

*Trueprop ((-))*

*True (true)*

*False (false)*

*Not (Bool.not)*

*op | ((- orelse/ -))*

*op & ((- andalso/ -))*

*If ((if -/ then -/ else -))*

**setup**  $\ll$

*let*

```

fun eq-codegen thy defs dep thyname b t gr =
  (case strip-comb t of
    (Const (op =, Type (-, [Type (fun, -), -])), -) => NONE
  | (Const (op =, -), [t, u]) =>
    let
      val (pt, gr') = Codegen.invoke-codegen thy defs dep thyname false t gr;
      val (pu, gr'') = Codegen.invoke-codegen thy defs dep thyname false u gr';
      val (-, gr''') = Codegen.invoke-tycodegen thy defs dep thyname false
        HLogic.boolT gr'';
    in
      SOME (Codegen.parens
        (Pretty.block [pt, Codegen.str =, Pretty.brk 1, pu]), gr''')
    end
  | (t as Const (op =, -), ts) => SOME (Codegen.invoke-codegen
    thy defs dep thyname b (Codegen.eta-expand t ts 2) gr)
  | - => NONE);

in
  Codegen.add-codegen eq-codegen eq-codegen
end
>>

```

## 2.5 Evaluation and normalization by evaluation

```

setup <<
  Value.add-evaluator (SML, Codegen.eval-term o ProofContext.theory-of)
>>

```

```

ML <<
  structure Eval-Method =
  struct

```

```

    val eval-ref : (unit -> bool) option ref = ref NONE;

```

```

  end;
>>

```

```

oracle eval-oracle = << fn ct =>
  let
    val thy = Thm.theory-of-cterm ct;
    val t = Thm.term-of ct;
    val dummy = @{cprop True};
  in case try HLogic.dest-Trueprop t
    of SOME t' => if Code-ML.eval-term
      (Eval-Method.eval-ref, Eval-Method.eval-ref) thy t' []
      then Thm.capply (Thm.capply @{cterm op ≡ :: prop ⇒ prop} ct)
        dummy
      else dummy
  end

```



```

    | NONE => dummy
  end
>>

ML <<
fun gen-eval-method conv ctxt = SIMPLE-METHOD'
  (CONVERSION (Conv.params-conv (~1) (K (Conv.concl-conv (~1) conv))
  ctxt)
  THEN' rtac TrueI)
>>

method-setup eval = << Scan.succeed (gen-eval-method eval-oracle) >>
  solve goal by evaluation

method-setup evaluation = << Scan.succeed (gen-eval-method Codegen.evaluation-conv)
>>
  solve goal by evaluation

method-setup normalization = <<
  Scan.succeed (K (SIMPLE-METHOD' (CONVERSION Nbe.norm-conv THEN'
  (fn k => TRY (rtac TrueI k)))))
>> solve goal by normalization

2.6 Quickcheck

setup <<
  Quickcheck.add-generator (SML, Codegen.test-term)
>>

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

end

```

### 3 Orderings: Abstract orderings

```

theory Orderings
imports Code-Setup
uses ~~/src/Provers/order.ML
begin

```

#### 3.1 Quasi orders

```

class preorder = ord +
  assumes less-le-not-le:  $x < y \longleftrightarrow x \leq y \wedge \neg (y \leq x)$ 
  and order-refl [iff]:  $x \leq x$ 
  and order-trans:  $x \leq y \implies y \leq z \implies x \leq z$ 
begin

```

Reflexivity.

**lemma** *eq-refl*:  $x = y \implies x \leq y$

— This form is useful with the classical reasoner.

**by** (*erule ssubst*) (*rule order-refl*)

**lemma** *less-irrefl* [*iff*]:  $\neg x < x$

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

**lemma** *less-imp-le*:  $x < y \implies x \leq y$

**unfolding** *less-le-not-le* **by** *blast*

Asymmetry.

**lemma** *less-not-sym*:  $x < y \implies \neg (y < x)$

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

**lemma** *less-asm*:  $x < y \implies (\neg P \implies y < x) \implies P$

**by** (*drule less-not-sym, erule contrapos-np*) *simp*

Transitivity.

**lemma** *less-trans*:  $x < y \implies y < z \implies x < z$

**by** (*auto simp add: less-le-not-le intro: order-trans*)

**lemma** *le-less-trans*:  $x \leq y \implies y < z \implies x < z$

**by** (*auto simp add: less-le-not-le intro: order-trans*)

**lemma** *less-le-trans*:  $x < y \implies y \leq z \implies x < z$

**by** (*auto simp add: less-le-not-le intro: order-trans*)

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

**lemma** *less-imp-not-less*:  $x < y \implies (\neg y < x) \longleftrightarrow \text{True}$

**by** (*blast elim: less-asm*)

**lemma** *less-imp-triv*:  $x < y \implies (y < x \longrightarrow P) \longleftrightarrow \text{True}$

**by** (*blast elim: less-asm*)

Transitivity rules for calculational reasoning

**lemma** *less-asm'*:  $a < b \implies b < a \implies P$

**by** (*rule less-asm*)

Dual order

**lemma** *dual-preorder*:

*preorder* (*op*  $\geq$ ) (*op*  $>$ )

**proof qed** (*auto simp add: less-le-not-le intro: order-trans*)

**end**

### 3.2 Partial orders

**class** *order* = *preorder* +  
**assumes** *antisym*:  $x \leq y \implies y \leq x \implies x = y$   
**begin**

Reflexivity.

**lemma** *less-le*:  $x < y \iff x \leq y \wedge x \neq y$   
**by** (*auto simp add: less-le-not-le intro: antisym*)

**lemma** *le-less*:  $x \leq y \iff x < y \vee x = y$   
 — NOT suitable for iff, since it can cause PROOF FAILED.  
**by** (*simp add: less-le*) *blast*

**lemma** *le-imp-less-or-eq*:  $x \leq y \implies x < y \vee x = y$   
**unfolding** *less-le* **by** *blast*

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

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

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

Transitivity rules for calculational reasoning

**lemma** *neq-le-trans*:  $a \neq b \implies a \leq b \implies a < b$   
**by** (*simp add: less-le*)

**lemma** *le-neq-trans*:  $a \leq b \implies a \neq b \implies a < b$   
**by** (*simp add: less-le*)

Asymmetry.

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

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

**lemma** *less-imp-neq*:  $x < y \implies x \neq y$   
**by** (*erule contrapos-pn, erule subst, rule less-irrefl*)

Least value operator

**definition** (*in ord*)  
*Least* :: ( $'a \Rightarrow \text{bool}$ )  $\Rightarrow 'a$  (**binder** *LEAST* 10) **where**  
*Least* *P* = (*THE* *x*.  $P\ x \wedge (\forall y. P\ y \longrightarrow x \leq y)$ )

**lemma** *Least-equality*:  
**assumes**  $P\ x$   
**and**  $\bigwedge y. P\ y \implies x \leq y$

**shows**  $\text{Least } P = x$   
**unfolding**  $\text{Least-def}$  **by** (rule the-equality)  
 (blast intro: assms antisym)+

**lemma**  $\text{LeastI2-order}$ :  
**assumes**  $P x$   
**and**  $\bigwedge y. P y \implies x \leq y$   
**and**  $\bigwedge x. P x \implies \forall y. P y \longrightarrow x \leq y \implies Q x$   
**shows**  $Q (\text{Least } P)$   
**unfolding**  $\text{Least-def}$  **by** (rule theI2)  
 (blast intro: assms antisym)+

Dual order

**lemma**  $\text{dual-order}$ :  
 $\text{order } (op \geq) (op >)$   
**by** (intro-locales, rule dual-preorder) (unfold-locales, rule antisym)  
**end**

### 3.3 Linear (total) orders

**class**  $\text{linorder} = \text{order} +$   
**assumes**  $\text{linear}: x \leq y \vee y \leq x$   
**begin**

**lemma**  $\text{less-linear}: x < y \vee x = y \vee y < x$   
**unfolding**  $\text{less-le}$  **using**  $\text{less-le linear}$  **by** blast

**lemma**  $\text{le-less-linear}: x \leq y \vee y < x$   
**by** (simp add: le-less less-linear)

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

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

**lemma**  $\text{not-less}: \neg x < y \longleftrightarrow y \leq x$   
**apply** (simp add: less-le)  
**using**  $\text{linear}$  **apply** (blast intro: antisym)  
**done**

**lemma**  $\text{not-less-iff-gr-or-eq}$ :  
 $\neg(x < y) \longleftrightarrow (x > y \mid x = y)$   
**apply**(simp add: not-less le-less)  
**apply** blast  
**done**

**lemma** *not-le*:  $\neg x \leq y \longleftrightarrow y < x$   
**apply** (*simp add: less-le*)  
**using** *linear* **apply** (*blast intro: antisym*)  
**done**

**lemma** *neq-iff*:  $x \neq y \longleftrightarrow x < y \vee y < x$   
**by** (*cut-tac x = x and y = y in less-linear, auto*)

**lemma** *neqE*:  $x \neq y \Longrightarrow (x < y \Longrightarrow R) \Longrightarrow (y < x \Longrightarrow R) \Longrightarrow R$   
**by** (*simp add: neq-iff*) *blast*

**lemma** *antisym-conv1*:  $\neg x < y \Longrightarrow x \leq y \longleftrightarrow x = y$   
**by** (*blast intro: antisym dest: not-less [THEN iffD1]*)

**lemma** *antisym-conv2*:  $x \leq y \Longrightarrow \neg x < y \longleftrightarrow x = y$   
**by** (*blast intro: antisym dest: not-less [THEN iffD1]*)

**lemma** *antisym-conv3*:  $\neg y < x \Longrightarrow \neg x < y \longleftrightarrow x = y$   
**by** (*blast intro: antisym dest: not-less [THEN iffD1]*)

**lemma** *leI*:  $\neg x < y \Longrightarrow y \leq x$   
**unfolding** *not-less* .

**lemma** *leD*:  $y \leq x \Longrightarrow \neg x < y$   
**unfolding** *not-less* .

**lemma** *not-leE*:  $\neg y \leq x \Longrightarrow x < y$   
**unfolding** *not-le* .

Dual order

**lemma** *dual-linorder*:  
*linorder* (*op*  $\geq$ ) (*op*  $>$ )  
**by** (*rule linorder.intro, rule dual-order*) (*unfold-locales, rule linear*)

min/max

**definition** (*in ord*) *min* ::  $'a \Rightarrow 'a \Rightarrow 'a$  **where**  
 $[code\ del]: min\ a\ b = (if\ a \leq b\ then\ a\ else\ b)$

**definition** (*in ord*) *max* ::  $'a \Rightarrow 'a \Rightarrow 'a$  **where**  
 $[code\ del]: max\ a\ b = (if\ a \leq b\ then\ b\ else\ a)$

**lemma** *min-le-iff-disj*:  
 $min\ x\ y \leq z \longleftrightarrow x \leq z \vee y \leq z$   
**unfolding** *min-def* **using** *linear* **by** (*auto intro: order-trans*)

**lemma** *le-max-iff-disj*:  
 $z \leq max\ x\ y \longleftrightarrow z \leq x \vee z \leq y$

**unfolding** *max-def* **using** *linear* **by** (*auto intro: order-trans*)

**lemma** *min-less-iff-disj*:

$$\min x y < z \iff x < z \vee y < z$$

**unfolding** *min-def le-less* **using** *less-linear* **by** (*auto intro: less-trans*)

**lemma** *less-max-iff-disj*:

$$z < \max x y \iff z < x \vee z < y$$

**unfolding** *max-def le-less* **using** *less-linear* **by** (*auto intro: less-trans*)

**lemma** *min-less-iff-conj* [*simp*]:

$$z < \min x y \iff z < x \wedge z < y$$

**unfolding** *min-def le-less* **using** *less-linear* **by** (*auto intro: less-trans*)

**lemma** *max-less-iff-conj* [*simp*]:

$$\max x y < z \iff x < z \wedge y < z$$

**unfolding** *max-def le-less* **using** *less-linear* **by** (*auto intro: less-trans*)

**lemma** *split-min* [*noatp*]:

$$P (\min i j) \iff (i \leq j \longrightarrow P i) \wedge (\neg i \leq j \longrightarrow P j)$$

**by** (*simp add: min-def*)

**lemma** *split-max* [*noatp*]:

$$P (\max i j) \iff (i \leq j \longrightarrow P j) \wedge (\neg i \leq j \longrightarrow P i)$$

**by** (*simp add: max-def*)

**end**

Explicit dictionaries for code generation

**lemma** *min-ord-min* [*code, code unfold, code inline del*]:

$$\min = \text{ord.min } (op \leq)$$

**by** (*rule ext*) + (*simp add: min-def ord.min-def*)

**declare** *ord.min-def* [*code*]

**lemma** *max-ord-max* [*code, code unfold, code inline del*]:

$$\max = \text{ord.max } (op \leq)$$

**by** (*rule ext*) + (*simp add: max-def ord.max-def*)

**declare** *ord.max-def* [*code*]

### 3.4 Reasoning tools setup

ML  $\ll$

*signature* *ORDERS* =

*sig*

*val* *print-structures*: *Proof.context*  $\rightarrow$  *unit*

*val* *setup*: *theory*  $\rightarrow$  *theory*

```

    val order-tac: thm list -> Proof.context -> int -> tactic
end;

structure Orders: ORDERS =
struct

(** Theory and context data **)

fun struct-eq ((s1: string, ts1), (s2, ts2)) =
  (s1 = s2) andalso eq-list (op aconv) (ts1, ts2);

structure Data = GenericDataFun
(
  type T = ((string * term list) * Order-Tac.less-arith) list;
  (* Order structures:
    identifier of the structure, list of operations and record of theorems
    needed to set up the transitivity reasoner,
    identifier and operations identify the structure uniquely. *)
  val empty = [];
  val extend = I;
  fun merge - = AList.join struct-eq (K fst);
);

fun print-structures ctxt =
  let
    val structs = Data.get (Context.Proof ctxt);
    fun pretty-term t = Pretty.block
      [Pretty.quote (Syntax.pretty-term ctxt t), Pretty.brk 1,
       Pretty.str ::, Pretty.brk 1,
       Pretty.quote (Syntax.pretty-typ ctxt (type-of t))];
    fun pretty-struct ((s, ts), _) = Pretty.block
      [Pretty.str s, Pretty.str :, Pretty.brk 1,
       Pretty.enclose ( ) (Pretty.breaks (map pretty-term ts))];
  in
    Pretty.writeln (Pretty.big-list Order structures: (map pretty-struct structs))
  end;

(** Method **)

fun struct-tac ((s, [eq, le, less]), thms) prems =
  let
    fun decomp thy (@{const Trueprop} $ t) =
      let
        fun excluded t =
          (* exclude numeric types: linear arithmetic subsumes transitivity *)
          let val T = type-of t
            in
              T = HOLogic.natT orelse T = HOLogic.intT orelse T = HOLogic.realT
            end
        end
      in
        decomp thy (t)
      end
  end

```

```

    end;
  fun rel (bin-op $ t1 $ t2) =
    if excluded t1 then NONE
    else if Pattern.matches thy (eq, bin-op) then SOME (t1, =, t2)
    else if Pattern.matches thy (le, bin-op) then SOME (t1, <=, t2)
    else if Pattern.matches thy (less, bin-op) then SOME (t1, <, t2)
    else NONE
  | rel - = NONE;
  fun dec (Const (@{const-name Not}, -) $ t) = (case rel t
    of NONE => NONE
    | SOME (t1, rel, t2) => SOME (t1, ~ ^ rel, t2))
    | dec x = rel x;
  in dec t end
  | decomp thy - = NONE;
in
  case s of
    order => Order-Tac.partial-tac decomp thms prems
  | linorder => Order-Tac.linear-tac decomp thms prems
  | - => error (Unknown kind of order ' ^ s ^ ' encountered in transitivity
    reasoner.)
  end

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

(** Attribute **)

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

  val attrib-setup =
    Attrib.setup @{binding order}
      (Scan.lift ((Args.add -- Args.name >> (fn (-, s) => SOME s) || Args.del
    >> K NONE) --|
      Args.colon (* FIXME || Scan.succeed true *) ) -- Scan.lift Args.name --
      Scan.repeat Args.term
    >> (fn ((SOME tag, n), ts) => add-struct-thm (n, ts) tag
      | ((NONE, n), ts) => del-struct (n, ts)))
    theorems controlling transitivity reasoner;

(** Diagnostic command **)

```



```

val - =
  OuterSyntax.improper-command print-orders
    print order structures available to transitivity reasoner OuterKeyword.diag
    (Scan.succeed (Toplevel.no-timing o Toplevel.unknown-context o
      Toplevel.keep (print-structures o Toplevel.context-of)));

(** Setup **)

val setup =
  Method.setup @{binding order} (Scan.succeed (SIMPLE-METHOD' o order-tac
    []))
    transitivity reasoner #>
    attrib-setup;

end;

>>

setup Orders.setup

Declarations to set up transitivity reasoner of partial and linear orders.

context order
begin

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

declare order-refl [order add le-refl: order op = :: 'a => 'a => bool op <= op
  <]

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

declare antisym [order add eqI: order op = :: 'a => 'a => bool op <= op <]

declare eq-refl [order add eqD1: order op = :: 'a => 'a => bool op <= op <]

declare sym [THEN eq-refl, order add eqD2: order op = :: 'a => 'a => bool op
  <= op <]

declare less-trans [order add less-trans: order op = :: 'a => 'a => bool op <=
  op <]

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

```

```

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

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

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

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

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

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

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

end

context linorder
begin

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

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

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

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

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

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

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

declare not-le [THEN iffD1, order add not-leD: linorder op = :: 'a => 'a =>
bool op <= op <]

```

```

declare antisym [order add eqI: linorder op = :: 'a => 'a => bool op <= op <]

declare eq-refl [order add eqD1: linorder op = :: 'a => 'a => bool op <= op <]

declare sym [THEN eq-refl, order add eqD2: linorder op = :: 'a => 'a => bool
op <= op <]

declare less-trans [order add less-trans: linorder op = :: 'a => 'a => bool op <=
op <]

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

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

declare order-trans [order add le-trans: linorder op = :: 'a => 'a => bool op <=
op <]

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

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

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

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

declare not-sym [order add not-sym: linorder op = :: 'a => 'a => bool op <=
op <]

end

setup <<
  let

  fun prp t thm = (#prop (rep-thm thm) = t);

  fun prove-antisym-le sg ss ((le as Const(-,T)) $ r $ s) =
    let val prems = prems-of-ss ss;
      val less = Const (@{const-name less}, T);
      val t = HOLogic.mk-Trueprop(le $ s $ r);
    in case find-first (prp t) prems of
      NONE =>

```

```

    let val t = HOLogic.mk-Trueprop(HOLogic.Not $ (less $ r $ s))
    in case find-first (prp t) prems of
        NONE => NONE
      | SOME thm => SOME(mk-meta-eq(thm RS @{thm linorder-class.antisym-conv1}))
    end
  | SOME thm => SOME(mk-meta-eq(thm RS @{thm order-class.antisym-conv}))
end
handle THM - => NONE;

fun prove-antisym-less sg ss (NotC $ ((less as Const(-,T)) $ r $ s)) =
  let val prems = prems-of-ss ss;
      val le = Const (@{const-name less-eq}, T);
      val t = HOLogic.mk-Trueprop(le $ r $ s);
  in case find-first (prp t) prems of
      NONE =>
        let val t = HOLogic.mk-Trueprop(NotC $ (less $ s $ r))
        in case find-first (prp t) prems of
            NONE => NONE
          | SOME thm => SOME(mk-meta-eq(thm RS @{thm linorder-class.antisym-conv3}))
        end
      | SOME thm => SOME(mk-meta-eq(thm RS @{thm linorder-class.antisym-conv2}))
    end
  handle THM - => NONE;

fun add-simprocs procs thy =
  Simplifier.map-simpset (fn ss => ss
    addsimprocs (map (fn (name, raw-ts, proc) =>
      Simplifier.simproc thy name raw-ts proc) procs)) thy;
fun add-solver name tac =
  Simplifier.map-simpset (fn ss => ss addSolver
    mk-solver' name (fn ss => tac (Simplifier.prems-of-ss ss) (Simplifier.the-context
ss)));

in
  add-simprocs [
    (antisym le, [(x::'a::order) <= y], prove-antisym-le),
    (antisym less, [~ (x::'a::linorder) < y], prove-antisym-less)
  ]
#> add-solver Transitivity Orders.order-tac
(* Adding the transitivity reasoners also as safe solvers showed a slight
speed up, but the reasoning strength appears to be not higher (at least
no breaking of additional proofs in the entire HOL distribution, as
of 5 March 2004, was observed). *)
end
>>

```

### 3.5 Name duplicates

lemmas order-less-le = less-le

```

lemmas order-eq-refl = preorder-class.eq-refl
lemmas order-less-irrefl = preorder-class.less-irrefl
lemmas order-le-less = order-class.le-less
lemmas order-le-imp-less-or-eq = order-class.le-imp-less-or-eq
lemmas order-less-imp-le = preorder-class.less-imp-le
lemmas order-less-imp-not-eq = order-class.less-imp-not-eq
lemmas order-less-imp-not-eq2 = order-class.less-imp-not-eq2
lemmas order-neq-le-trans = order-class.neq-le-trans
lemmas order-le-neq-trans = order-class.le-neq-trans

lemmas order-antisym = antisym
lemmas order-less-not-sym = preorder-class.less-not-sym
lemmas order-less-asy = preorder-class.less-asy
lemmas order-eq-iff = order-class.eq-iff
lemmas order-antisym-conv = order-class.antisym-conv
lemmas order-less-trans = preorder-class.less-trans
lemmas order-le-less-trans = preorder-class.le-less-trans
lemmas order-less-le-trans = preorder-class.less-le-trans
lemmas order-less-imp-not-less = preorder-class.less-imp-not-less
lemmas order-less-imp-triv = preorder-class.less-imp-triv
lemmas order-less-asy' = preorder-class.less-asy'

lemmas linorder-linear = linear
lemmas linorder-less-linear = linorder-class.less-linear
lemmas linorder-le-less-linear = linorder-class.le-less-linear
lemmas linorder-le-cases = linorder-class.le-cases
lemmas linorder-not-less = linorder-class.not-less
lemmas linorder-not-le = linorder-class.not-le
lemmas linorder-neq-iff = linorder-class.neq-iff
lemmas linorder-neqE = linorder-class.neqE
lemmas linorder-antisym-conv1 = linorder-class.antisym-conv1
lemmas linorder-antisym-conv2 = linorder-class.antisym-conv2
lemmas linorder-antisym-conv3 = linorder-class.antisym-conv3

```

### 3.6 Bounded quantifiers

#### syntax

```

-All-less :: [idt, 'a, bool] => bool  (( $\exists$ ALL -<./ -) [0, 0, 10] 10)
-Ex-less :: [idt, 'a, bool] => bool  (( $\exists$ EX -<./ -) [0, 0, 10] 10)
-All-less-eq :: [idt, 'a, bool] => bool  (( $\exists$ ALL -<=./ -) [0, 0, 10] 10)
-Ex-less-eq :: [idt, 'a, bool] => bool  (( $\exists$ EX -<=./ -) [0, 0, 10] 10)

-All-greater :: [idt, 'a, bool] => bool  (( $\exists$ ALL ->./ -) [0, 0, 10] 10)
-Ex-greater :: [idt, 'a, bool] => bool  (( $\exists$ EX ->./ -) [0, 0, 10] 10)
-All-greater-eq :: [idt, 'a, bool] => bool  (( $\exists$ ALL ->=./ -) [0, 0, 10] 10)
-Ex-greater-eq :: [idt, 'a, bool] => bool  (( $\exists$ EX ->=./ -) [0, 0, 10] 10)

```

#### syntax (xsymbols)

```

-All-less :: [idt, 'a, bool] => bool  (( $\forall$ -<./ -) [0, 0, 10] 10)

```

```

-Ex-less :: [idt, 'a, bool] => bool    (( $\exists \exists$  -<-./ -) [0, 0, 10] 10)
-All-less-eq :: [idt, 'a, bool] => bool  (( $\exists \forall$  -≤-./ -) [0, 0, 10] 10)
-Ex-less-eq :: [idt, 'a, bool] => bool   (( $\exists \exists$  -≤-./ -) [0, 0, 10] 10)

-All-greater :: [idt, 'a, bool] => bool  (( $\exists \forall$  ->-./ -) [0, 0, 10] 10)
-Ex-greater :: [idt, 'a, bool] => bool   (( $\exists \exists$  ->-./ -) [0, 0, 10] 10)
-All-greater-eq :: [idt, 'a, bool] => bool (( $\exists \forall$  -≥-./ -) [0, 0, 10] 10)
-Ex-greater-eq :: [idt, 'a, bool] => bool (( $\exists \exists$  -≥-./ -) [0, 0, 10] 10)

```

**syntax (HOL)**

```

-All-less :: [idt, 'a, bool] => bool    (( $\exists !$  -<-./ -) [0, 0, 10] 10)
-Ex-less :: [idt, 'a, bool] => bool     (( $\exists ?$  -<-./ -) [0, 0, 10] 10)
-All-less-eq :: [idt, 'a, bool] => bool  (( $\exists !$  -<= -./ -) [0, 0, 10] 10)
-Ex-less-eq :: [idt, 'a, bool] => bool   (( $\exists ?$  -<= -./ -) [0, 0, 10] 10)

```

**syntax (HTML output)**

```

-All-less :: [idt, 'a, bool] => bool    (( $\exists \forall$  -<-./ -) [0, 0, 10] 10)
-Ex-less :: [idt, 'a, bool] => bool     (( $\exists \exists$  -<-./ -) [0, 0, 10] 10)
-All-less-eq :: [idt, 'a, bool] => bool  (( $\exists \forall$  -≤-./ -) [0, 0, 10] 10)
-Ex-less-eq :: [idt, 'a, bool] => bool   (( $\exists \exists$  -≤-./ -) [0, 0, 10] 10)

-All-greater :: [idt, 'a, bool] => bool  (( $\exists \forall$  ->-./ -) [0, 0, 10] 10)
-Ex-greater :: [idt, 'a, bool] => bool   (( $\exists \exists$  ->-./ -) [0, 0, 10] 10)
-All-greater-eq :: [idt, 'a, bool] => bool (( $\exists \forall$  -≥-./ -) [0, 0, 10] 10)
-Ex-greater-eq :: [idt, 'a, bool] => bool (( $\exists \exists$  -≥-./ -) [0, 0, 10] 10)

```

**translations**

```

ALL x<y. P  =>  ALL x. x < y → P
EX x<y. P   =>  EX x. x < y ∧ P
ALL x<=y. P =>  ALL x. x <= y → P
EX x<=y. P  =>  EX x. x <= y ∧ P
ALL x>y. P  =>  ALL x. x > y → P
EX x>y. P   =>  EX x. x > y ∧ P
ALL x>=y. P =>  ALL x. x >= y → P
EX x>=y. P  =>  EX x. x >= y ∧ P

```

**print-translation**  $\ll$ 

let

```

val All-binder = Syntax.binder-name @{const-syntax All};
val Ex-binder  = Syntax.binder-name @{const-syntax Ex};
val impl = @{const-syntax op -->};
val conj  = @{const-syntax op &};
val less  = @{const-syntax less};
val less-eq = @{const-syntax less-eq};

```

val trans =

```

(((All-binder, impl, less), (-All-less, -All-greater)),
 ((All-binder, impl, less-eq), (-All-less-eq, -All-greater-eq)),
 ((Ex-binder, conj, less), (-Ex-less, -Ex-greater)),

```

```

((Ex-binder, conj, less-eq), (-Ex-less-eq, -Ex-greater-eq));

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

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

```

### 3.7 Transitivity reasoning

**context** *ord*

**begin**

**lemma** *ord-le-eq-trans*:  $a \leq b \implies b = c \implies a \leq c$   
**by** (*rule subst*)

**lemma** *ord-eq-le-trans*:  $a = b \implies b \leq c \implies a \leq c$   
**by** (*rule ssubst*)

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

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

**end**

**lemma** *order-less-subst2*:  $(a::'a::order) < b \implies f\ b < (c::'c::order) \implies$   
 $(!!x\ y. x < y \implies f\ x < f\ y) \implies f\ a < c$

**proof** –

**assume**  $r$ :  $!!x\ y. x < y \implies f\ x < f\ y$   
**assume**  $a < b$  **hence**  $f\ a < f\ b$  **by** (*rule r*)  
**also assume**  $f\ b < c$   
**finally** (*order-less-trans*) **show** ?thesis .

**qed**

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

**proof** –

assume  $r: !!x\ y. x < y ==> f\ x < f\ y$

assume  $a < f\ b$

also assume  $b < c$  hence  $f\ b < f\ c$  by (rule  $r$ )

finally (order-less-trans) show ?thesis .

qed

**lemma** *order-le-less-subst2*:  $(a::'a::order) <= b ==> f\ b < (c::'c::order) ==>$   
 $(!!x\ y. x <= y ==> f\ x <= f\ y) ==> f\ a < c$

**proof** –

assume  $r: !!x\ y. x <= y ==> f\ x <= f\ y$

assume  $a <= b$  hence  $f\ a <= f\ b$  by (rule  $r$ )

also assume  $f\ b < c$

finally (order-le-less-trans) show ?thesis .

qed

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

**proof** –

assume  $r: !!x\ y. x < y ==> f\ x < f\ y$

assume  $a <= f\ b$

also assume  $b < c$  hence  $f\ b < f\ c$  by (rule  $r$ )

finally (order-le-less-trans) show ?thesis .

qed

**lemma** *order-less-le-subst2*:  $(a::'a::order) < b ==> f\ b <= (c::'c::order) ==>$   
 $(!!x\ y. x < y ==> f\ x < f\ y) ==> f\ a < c$

**proof** –

assume  $r: !!x\ y. x < y ==> f\ x < f\ y$

assume  $a < b$  hence  $f\ a < f\ b$  by (rule  $r$ )

also assume  $f\ b <= c$

finally (order-less-le-trans) show ?thesis .

qed

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

**proof** –

assume  $r: !!x\ y. x <= y ==> f\ x <= f\ y$

assume  $a < f\ b$

also assume  $b <= c$  hence  $f\ b <= f\ c$  by (rule  $r$ )

finally (order-less-le-trans) show ?thesis .

qed

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

**proof** –

assume  $r: !!x\ y. x <= y ==> f\ x <= f\ y$



assume  $a \leq f b$   
 also assume  $b \leq c$  hence  $f b \leq f c$  by (rule  $r$ )  
 finally (order-trans) show ?thesis .  
 qed

**lemma** order-subst2:  $(a::'a::order) \leq b \implies f b \leq (c::'c::order) \implies$   
 $(!!x y. x \leq y \implies f x \leq f y) \implies f a \leq c$   
**proof** –  
 assume  $r: !!x y. x \leq y \implies f x \leq f y$   
 assume  $a \leq b$  hence  $f a \leq f b$  by (rule  $r$ )  
 also assume  $f b \leq c$   
 finally (order-trans) show ?thesis .  
 qed

**lemma** ord-le-eq-subst:  $a \leq b \implies f b = c \implies$   
 $(!!x y. x \leq y \implies f x \leq f y) \implies f a \leq c$   
**proof** –  
 assume  $r: !!x y. x \leq y \implies f x \leq f y$   
 assume  $a \leq b$  hence  $f a \leq f b$  by (rule  $r$ )  
 also assume  $f b = c$   
 finally (ord-le-eq-trans) show ?thesis .  
 qed

**lemma** ord-eq-le-subst:  $a = f b \implies b \leq c \implies$   
 $(!!x y. x \leq y \implies f x \leq f y) \implies a \leq f c$   
**proof** –  
 assume  $r: !!x y. x \leq y \implies f x \leq f y$   
 assume  $a = f b$   
 also assume  $b \leq c$  hence  $f b \leq f c$  by (rule  $r$ )  
 finally (ord-eq-le-trans) show ?thesis .  
 qed

**lemma** ord-less-eq-subst:  $a < b \implies f b = c \implies$   
 $(!!x y. x < y \implies f x < f y) \implies f a < c$   
**proof** –  
 assume  $r: !!x y. x < y \implies f x < f y$   
 assume  $a < b$  hence  $f a < f b$  by (rule  $r$ )  
 also assume  $f b = c$   
 finally (ord-less-eq-trans) show ?thesis .  
 qed

**lemma** ord-eq-less-subst:  $a = f b \implies b < c \implies$   
 $(!!x y. x < y \implies f x < f y) \implies a < f c$   
**proof** –  
 assume  $r: !!x y. x < y \implies f x < f y$   
 assume  $a = f b$   
 also assume  $b < c$  hence  $f b < f c$  by (rule  $r$ )  
 finally (ord-eq-less-trans) show ?thesis .  
 qed

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

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

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

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

**lemmas** (**in** *order*) [*trans*] =  
*antisym*

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

**lemmas** [*trans*] =  
*trans*

**lemmas** *order-trans-rules* =  
*order-less-subst2*  
*order-less-subst1*  
*order-le-less-subst2*  
*order-le-less-subst1*  
*order-less-le-subst2*  
*order-less-le-subst1*

```

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

```

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

**lemma** *xt1*:

```

a = b ==> b > c ==> a > c
a > b ==> b = c ==> a > c
a = b ==> b >= c ==> a >= c
a >= b ==> b = c ==> a >= c
(x::'a::order) >= y ==> y >= x ==> x = y
(x::'a::order) >= y ==> y >= z ==> x >= z
(x::'a::order) > y ==> y >= z ==> x > z
(x::'a::order) >= y ==> y > z ==> x > z
(a::'a::order) > b ==> b > a ==> P
(x::'a::order) > y ==> y > z ==> x > z
(a::'a::order) >= b ==> a ~ b ==> a > b
(a::'a::order) ~ b ==> a >= b ==> a > b
a = f b ==> b > c ==> (!!x y. x > y ==> f x > f y) ==> a > f c
a > b ==> f b = c ==> (!!x y. x > y ==> f x > f y) ==> f a > c
a = f b ==> b >= c ==> (!!x y. x >= y ==> f x >= f y) ==> a >= f c
a >= b ==> f b = c ==> (!!x y. x >= y ==> f x >= f y) ==> f a >= c
by auto

```

**lemma** *xt2*:

```

(a::'a::order) >= f b ==> b >= c ==> (!!x y. x >= y ==> f x >= f y) ==>
a >= f c
by (subgoal-tac f b >= f c, force, force)

```

**lemma** *xt3*:  $(a :: 'a :: \text{order}) \geq b \implies (f\ b :: 'b :: \text{order}) \geq c \implies$   
 $(!!x\ y. x \geq y \implies f\ x \geq f\ y) \implies f\ a \geq c$   
**by** (*subgoal-tac*  $f\ a \geq f\ b$ , *force*, *force*)

**lemma** *xt4*:  $(a :: 'a :: \text{order}) > f\ b \implies (b :: 'b :: \text{order}) \geq c \implies$   
 $(!!x\ y. x \geq y \implies f\ x \geq f\ y) \implies a > f\ c$   
**by** (*subgoal-tac*  $f\ b \geq f\ c$ , *force*, *force*)

**lemma** *xt5*:  $(a :: 'a :: \text{order}) > b \implies (f\ b :: 'b :: \text{order}) \geq c \implies$   
 $(!!x\ y. x > y \implies f\ x > f\ y) \implies f\ a > c$   
**by** (*subgoal-tac*  $f\ a > f\ b$ , *force*, *force*)

**lemma** *xt6*:  $(a :: 'a :: \text{order}) \geq f\ b \implies b > c \implies$   
 $(!!x\ y. x > y \implies f\ x > f\ y) \implies a > f\ c$   
**by** (*subgoal-tac*  $f\ b > f\ c$ , *force*, *force*)

**lemma** *xt7*:  $(a :: 'a :: \text{order}) \geq b \implies (f\ b :: 'b :: \text{order}) > c \implies$   
 $(!!x\ y. x \geq y \implies f\ x \geq f\ y) \implies f\ a > c$   
**by** (*subgoal-tac*  $f\ a \geq f\ b$ , *force*, *force*)

**lemma** *xt8*:  $(a :: 'a :: \text{order}) > f\ b \implies (b :: 'b :: \text{order}) > c \implies$   
 $(!!x\ y. x > y \implies f\ x > f\ y) \implies a > f\ c$   
**by** (*subgoal-tac*  $f\ b > f\ c$ , *force*, *force*)

**lemma** *xt9*:  $(a :: 'a :: \text{order}) > b \implies (f\ b :: 'b :: \text{order}) > c \implies$   
 $(!!x\ y. x > y \implies f\ x > f\ y) \implies f\ a > c$   
**by** (*subgoal-tac*  $f\ a > f\ b$ , *force*, *force*)

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

### 3.8 Monotonicity, least value operator and min/max

**context** *order*

**begin**

**definition** *mono* ::  $('a \Rightarrow 'b :: \text{order}) \Rightarrow \text{bool}$  **where**  
 $\text{mono}\ f \longleftrightarrow (\forall x\ y. x \leq y \longrightarrow f\ x \leq f\ y)$

**lemma** *monoI* [*intro?*]:  
**fixes**  $f :: 'a \Rightarrow 'b :: \text{order}$   
**shows**  $(\bigwedge x\ y. x \leq y \implies f\ x \leq f\ y) \implies \text{mono}\ f$   
**unfolding** *mono-def* **by** *iprover*

**lemma** *monoD* [*dest?*]:  
**fixes**  $f :: 'a \Rightarrow 'b :: \text{order}$   
**shows**  $\text{mono}\ f \implies x \leq y \implies f\ x \leq f\ y$   
**unfolding** *mono-def* **by** *iprover*

**definition** *strict-mono* :: (*'a*  $\Rightarrow$  *'b::order*)  $\Rightarrow$  *bool* **where**  
*strict-mono* *f*  $\longleftrightarrow (\forall x\ y. x < y \longrightarrow f\ x < f\ y)$

**lemma** *strict-monoI* [*intro?*]:  
**assumes**  $\bigwedge x\ y. x < y \Longrightarrow f\ x < f\ y$   
**shows** *strict-mono* *f*  
**using** *assms* **unfolding** *strict-mono-def* **by** *auto*

**lemma** *strict-monoD* [*dest?*]:  
*strict-mono* *f*  $\Longrightarrow x < y \Longrightarrow f\ x < f\ y$   
**unfolding** *strict-mono-def* **by** *auto*

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

**context** *linorder*  
**begin**

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

```

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

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

lemma min-of-mono:
  fixes  $f :: 'a \Rightarrow 'b::linorder$ 
  shows  $\text{mono } f \implies \min (f\ m) (f\ n) = f\ (\min\ m\ n)$ 
  by (auto simp: mono-def Orderings.min-def min-def intro: Orderings.antisym)

lemma max-of-mono:
  fixes  $f :: 'a \Rightarrow 'b::linorder$ 
  shows  $\text{mono } f \implies \max (f\ m) (f\ n) = f\ (\max\ m\ n)$ 
  by (auto simp: mono-def Orderings.max-def max-def intro: Orderings.antisym)

end

lemma min-leastL:  $(\forall x. \text{least } x \leq x) \implies \min\ \text{least } x = \text{least } x$ 
by (simp add: min-def)

lemma max-leastL:  $(\forall x. \text{least } x \leq x) \implies \max\ \text{least } x = x$ 
by (simp add: max-def)

lemma min-leastR:  $(\bigwedge x::'a::order. \text{least } x \leq x) \implies \min\ x\ \text{least} = \text{least}$ 
apply (simp add: min-def)
apply (blast intro: order-antisym)
done

lemma max-leastR:  $(\bigwedge x::'a::order. \text{least } x \leq x) \implies \max\ x\ \text{least} = x$ 
apply (simp add: max-def)
apply (blast intro: order-antisym)
done

```

### 3.9 Top and bottom elements

```
class top = preorder +
  fixes top :: 'a
  assumes top-greatest [simp]:  $x \leq top$ 
```

```
class bot = preorder +
  fixes bot :: 'a
  assumes bot-least [simp]:  $bot \leq x$ 
```

### 3.10 Dense orders

```
class dense-linear-order = linorder +
  assumes gt-ex:  $\exists y. x < y$ 
  and lt-ex:  $\exists y. y < x$ 
  and dense:  $x < y \implies (\exists z. x < z \wedge z < y)$ 
```

### 3.11 Wellorders

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

**lemma** *wellorder-Least-lemma*:

```
  fixes k :: 'a
  assumes P k
  shows  $P (LEAST x. P x)$  and  $(LEAST x. P x) \leq k$ 
proof -
  have  $P (LEAST x. P x) \wedge (LEAST x. P x) \leq k$ 
  using assms proof (induct k rule: less-induct)
    case (less x) then have  $P x$  by simp
  show ?case proof (rule classical)
    assume asm:  $\neg (P (LEAST a. P a) \wedge (LEAST a. P a) \leq x)$ 
    have  $\bigwedge y. P y \implies x \leq y$ 
    proof (rule classical)
      fix y
      assume  $P y$  and  $\neg x \leq y$ 
      with less have  $P (LEAST a. P a)$  and  $(LEAST a. P a) \leq y$ 
      by (auto simp add: not-le)
      with asm have  $x < (LEAST a. P a)$  and  $(LEAST a. P a) \leq y$ 
      by auto
      then show  $x \leq y$  by auto
    qed
  with  $\langle P x \rangle$  have Least:  $(LEAST a. P a) = x$ 
  by (rule Least-equality)
  with  $\langle P x \rangle$  show ?thesis by simp
qed
qed
then show  $P (LEAST x. P x)$  and  $(LEAST x. P x) \leq k$  by auto
```

qed

**lemmas** *LeastI* = *wellorder-Least-lemma*(1)

**lemmas** *Least-le* = *wellorder-Least-lemma*(2)

— The following 3 lemmas are due to Brian Huffman

**lemma** *LeastI-ex*:  $\exists x. P\ x \implies P\ (\text{Least } P)$

**by** (*erule exE*) (*erule LeastI*)

**lemma** *LeastI2*:

$P\ a \implies (\bigwedge x. P\ x \implies Q\ x) \implies Q\ (\text{Least } P)$

**by** (*blast intro: LeastI*)

**lemma** *LeastI2-ex*:

$\exists a. P\ a \implies (\bigwedge x. P\ x \implies Q\ x) \implies Q\ (\text{Least } P)$

**by** (*blast intro: LeastI-ex*)

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

**apply** (*simp* (*no-asm-use*) *add: not-le [symmetric]*)

**apply** (*erule contrapos-nn*)

**apply** (*erule Least-le*)

**done**

**end**

### 3.12 Order on bool

**instantiation** *bool* :: {*order*, *top*, *bot*}

**begin**

**definition**

*le-bool-def* [*code del*]:  $P \leq Q \longleftrightarrow P \longrightarrow Q$

**definition**

*less-bool-def* [*code del*]:  $(P::\text{bool}) < Q \longleftrightarrow \neg P \wedge Q$

**definition**

*top-bool-eq*:  $\text{top} = \text{True}$

**definition**

*bot-bool-eq*:  $\text{bot} = \text{False}$

**instance** **proof**

**qed** (*auto simp add: le-bool-def less-bool-def top-bool-eq bot-bool-eq*)

**end**

**lemma** *le-boolI*:  $(P \implies Q) \implies P \leq Q$

**by** (*simp add: le-bool-def*)



**lemma** *le-boolI'*:  $P \longrightarrow Q \Longrightarrow P \leq Q$

**by** (*simp add: le-bool-def*)

**lemma** *le-boolE*:  $P \leq Q \Longrightarrow P \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$

**by** (*simp add: le-bool-def*)

**lemma** *le-boolD*:  $P \leq Q \Longrightarrow P \longrightarrow Q$

**by** (*simp add: le-bool-def*)

**lemma** [*code*]:

$False \leq b \longleftrightarrow True$

$True \leq b \longleftrightarrow b$

$False < b \longleftrightarrow b$

$True < b \longleftrightarrow False$

**unfolding** *le-bool-def less-bool-def* **by** *simp-all*

### 3.13 Order on functions

**instantiation** *fun* :: (*type*, *ord*) *ord*

**begin**

**definition**

*le-fun-def* [*code del*]:  $f \leq g \longleftrightarrow (\forall x. f\ x \leq g\ x)$

**definition**

*less-fun-def* [*code del*]:  $(f :: 'a \Rightarrow 'b) < g \longleftrightarrow f \leq g \wedge \neg (g \leq f)$

**instance** ..

**end**

**instance** *fun* :: (*type*, *preorder*) *preorder* **proof**

**qed** (*auto simp add: le-fun-def less-fun-def*

*intro: order-trans order-antisym intro!: ext*)

**instance** *fun* :: (*type*, *order*) *order* **proof**

**qed** (*auto simp add: le-fun-def intro: order-antisym ext*)

**instantiation** *fun* :: (*type*, *top*) *top*

**begin**

**definition**

*top-fun-eq*:  $top = (\lambda x. top)$

**instance** **proof**

**qed** (*simp add: top-fun-eq le-fun-def*)

**end**

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

**definition**  
*bot-fun-eq*: *bot* = ( $\lambda x. bot$ )

**instance proof**  
**qed** (*simp add: bot-fun-eq le-fun-def*)

**end**

**lemma** *le-funI*:  $(\bigwedge x. f\ x \leq g\ x) \implies f \leq g$   
**unfolding** *le-fun-def* **by** *simp*

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

**lemma** *le-funD*:  $f \leq g \implies f\ x \leq g\ x$   
**unfolding** *le-fun-def* **by** *simp*

Handy introduction and elimination rules for  $\leq$  on unary and binary predicates

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

**lemma** *predicate1D* [*Pure.dest*, *dest*]:  $P \leq Q \implies P\ x \implies Q\ x$   
**apply** (*erule le-funE*)  
**apply** (*erule le-boolE*)  
**apply** *assumption* +  
**done**

**lemma** *predicate2I* [*Pure.intro!*, *intro!*]:  
**assumes** *PQ*:  $\bigwedge x\ y. P\ x\ y \implies Q\ x\ y$   
**shows**  $P \leq Q$   
**apply** (*rule le-funI*) +  
**apply** (*rule le-boolI*)  
**apply** (*rule PQ*)  
**apply** *assumption*  
**done**

**lemma** *predicate2D* [*Pure.dest*, *dest*]:  $P \leq Q \implies P\ x\ y \implies Q\ x\ y$   
**apply** (*erule le-funE*) +

```

apply (erule le-boolE)
apply assumption+
done

lemma rev-predicate1D:  $P\ x ==> P\ <= Q ==> Q\ x$ 
  by (rule predicate1D)

lemma rev-predicate2D:  $P\ x\ y ==> P\ <= Q ==> Q\ x\ y$ 
  by (rule predicate2D)

end

```

## 4 Lattices: Abstract lattices

```

theory Lattices
imports Orderings
begin

```

### 4.1 Lattices

```

notation
  less-eq (infix  $\sqsubseteq$  50) and
  less (infix  $\sqsubset$  50)

class lower-semilattice = order +
  fixes inf :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixl  $\sqcap$  70)
  assumes inf-le1 [simp]:  $x\ \sqcap\ y\ \sqsubseteq x$ 
  and inf-le2 [simp]:  $x\ \sqcap\ y\ \sqsubseteq y$ 
  and inf-greatest:  $x\ \sqsubseteq y \Longrightarrow x\ \sqsubseteq z \Longrightarrow x\ \sqsubseteq y\ \sqcap\ z$ 

class upper-semilattice = order +
  fixes sup :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixl  $\sqcup$  65)
  assumes sup-ge1 [simp]:  $x\ \sqsubseteq x\ \sqcup\ y$ 
  and sup-ge2 [simp]:  $y\ \sqsubseteq x\ \sqcup\ y$ 
  and sup-least:  $y\ \sqsubseteq x \Longrightarrow z\ \sqsubseteq x \Longrightarrow y\ \sqcup\ z\ \sqsubseteq x$ 
begin

Dual lattice

lemma dual-lattice:
  lower-semilattice (op  $\geq$ ) (op  $>$ ) sup
by (rule lower-semilattice.intro, rule dual-order)
  (unfold-locales, simp-all add: sup-least)

end

class lattice = lower-semilattice + upper-semilattice

```

## 4.1.1 Intro and elim rules

**context** *lower-semilattice*

**begin**

**lemma** *le-infI1*[*intro*]:

assumes  $a \sqsubseteq x$

shows  $a \sqcap b \sqsubseteq x$

**proof** (*rule order-trans*)

from *assms* **show**  $a \sqsubseteq x$  .

**show**  $a \sqcap b \sqsubseteq a$  **by** *simp*

**qed**

**lemmas** (**in**  $-$ ) [*rule del*] = *le-infI1*

**lemma** *le-infI2*[*intro*]:

assumes  $b \sqsubseteq x$

shows  $a \sqcap b \sqsubseteq x$

**proof** (*rule order-trans*)

from *assms* **show**  $b \sqsubseteq x$  .

**show**  $a \sqcap b \sqsubseteq b$  **by** *simp*

**qed**

**lemmas** (**in**  $-$ ) [*rule del*] = *le-infI2*

**lemma** *le-infI*[*intro!*]:  $x \sqsubseteq a \implies x \sqsubseteq b \implies x \sqsubseteq a \sqcap b$

**by** (*blast intro: inf-greatest*)

**lemmas** (**in**  $-$ ) [*rule del*] = *le-infI*

**lemma** *le-infE* [*elim!*]:  $x \sqsubseteq a \sqcap b \implies (x \sqsubseteq a \implies x \sqsubseteq b \implies P) \implies P$

**by** (*blast intro: order-trans*)

**lemmas** (**in**  $-$ ) [*rule del*] = *le-infE*

**lemma** *le-inf-iff* [*simp*]:

$x \sqsubseteq y \sqcap z = (x \sqsubseteq y \wedge x \sqsubseteq z)$

**by** *blast*

**lemma** *le-iff-inf*:  $(x \sqsubseteq y) = (x \sqcap y = x)$

**by** (*blast intro: antisym dest: eq-iff [THEN iffD1]*)

**lemma** *mono-inf*:

**fixes**  $f :: 'a \Rightarrow 'b :: \text{lower-semilattice}$

**shows**  $\text{mono } f \implies f (A \sqcap B) \leq f A \sqcap f B$

**by** (*auto simp add: mono-def intro: Lattices.inf-greatest*)

**end**

**context** *upper-semilattice*

**begin**

**lemma** *le-supI1*[*intro*]:  $x \sqsubseteq a \implies x \sqsubseteq a \sqcup b$

**by** (*rule order-trans*) *auto*

```

lemmas (in -) [rule del] = le-supI1

lemma le-supI2[intro]:  $x \sqsubseteq b \implies x \sqsubseteq a \sqcup b$ 
  by (rule order-trans) auto
lemmas (in -) [rule del] = le-supI2

lemma le-supI[intro!]:  $a \sqsubseteq x \implies b \sqsubseteq x \implies a \sqcup b \sqsubseteq x$ 
  by (blast intro: sup-least)
lemmas (in -) [rule del] = le-supI

lemma le-supE[elim!]:  $a \sqcup b \sqsubseteq x \implies (a \sqsubseteq x \implies b \sqsubseteq x \implies P) \implies P$ 
  by (blast intro: order-trans)
lemmas (in -) [rule del] = le-supE

lemma ge-sup-conv[simp]:
   $x \sqcup y \sqsubseteq z = (x \sqsubseteq z \wedge y \sqsubseteq z)$ 
by blast

lemma le-iff-sup:  $(x \sqsubseteq y) = (x \sqcup y = y)$ 
  by (blast intro: antisym dest: eq-iff [THEN iffD1])

lemma mono-sup:
  fixes f :: 'a  $\Rightarrow$  'b::upper-semilattice
  shows mono f  $\implies f A \sqcup f B \leq f (A \sqcup B)$ 
  by (auto simp add: mono-def intro: Lattices.sup-least)

end

4.1.2 Equational laws

context lower-semilattice
begin

lemma inf-commute:  $(x \sqcap y) = (y \sqcap x)$ 
  by (blast intro: antisym)

lemma inf-assoc:  $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$ 
  by (blast intro: antisym)

lemma inf-idem[simp]:  $x \sqcap x = x$ 
  by (blast intro: antisym)

lemma inf-left-idem[simp]:  $x \sqcap (x \sqcap y) = x \sqcap y$ 
  by (blast intro: antisym)

lemma inf-absorb1:  $x \sqsubseteq y \implies x \sqcap y = x$ 
  by (blast intro: antisym)

lemma inf-absorb2:  $y \sqsubseteq x \implies x \sqcap y = y$ 

```

```

    by (blast intro: antisym)

lemma inf-left-commute:  $x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z)$ 
  by (blast intro: antisym)

lemmas inf-ACI = inf-commute inf-assoc inf-left-commute inf-left-idem

end

context upper-semilattice
begin

lemma sup-commute:  $(x \sqcup y) = (y \sqcup x)$ 
  by (blast intro: antisym)

lemma sup-assoc:  $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$ 
  by (blast intro: antisym)

lemma sup-idem[simp]:  $x \sqcup x = x$ 
  by (blast intro: antisym)

lemma sup-left-idem[simp]:  $x \sqcup (x \sqcup y) = x \sqcup y$ 
  by (blast intro: antisym)

lemma sup-absorb1:  $y \sqsubseteq x \implies x \sqcup y = x$ 
  by (blast intro: antisym)

lemma sup-absorb2:  $x \sqsubseteq y \implies x \sqcup y = y$ 
  by (blast intro: antisym)

lemma sup-left-commute:  $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z)$ 
  by (blast intro: antisym)

lemmas sup-ACI = sup-commute sup-assoc sup-left-commute sup-left-idem

end

context lattice
begin

lemma inf-sup-absorb:  $x \sqcap (x \sqcup y) = x$ 
  by (blast intro: antisym inf-le1 inf-greatest sup-ge1)

lemma sup-inf-absorb:  $x \sqcup (x \sqcap y) = x$ 
  by (blast intro: antisym sup-ge1 sup-least inf-le1)

lemmas ACI = inf-ACI sup-ACI

```

**lemmas** *inf-sup-ord* = *inf-le1 inf-le2 sup-ge1 sup-ge2*

Towards distributivity

**lemma** *distrib-sup-le*:  $x \sqcup (y \sqcap z) \sqsubseteq (x \sqcup y) \sqcap (x \sqcup z)$   
**by** *blast*

**lemma** *distrib-inf-le*:  $(x \sqcap y) \sqcup (x \sqcap z) \sqsubseteq x \sqcap (y \sqcup z)$   
**by** *blast*

If you have one of them, you have them all.

**lemma** *distrib-imp1*:

**assumes** *D*:  $\forall x y z. x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$

**shows**  $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

**proof**–

**have**  $x \sqcup (y \sqcap z) = (x \sqcup (x \sqcap z)) \sqcup (y \sqcap z)$  **by**(*simp add:sup-inf-absorb*)  
**also have**  $\dots = x \sqcup (z \sqcap (x \sqcup y))$  **by**(*simp add:D inf-commute sup-assoc*)  
**also have**  $\dots = ((x \sqcup y) \sqcap x) \sqcup ((x \sqcup y) \sqcap z)$   
**by**(*simp add:inf-sup-absorb inf-commute*)  
**also have**  $\dots = (x \sqcup y) \sqcap (x \sqcup z)$  **by**(*simp add:D*)  
**finally show** *?thesis* .

**qed**

**lemma** *distrib-imp2*:

**assumes** *D*:  $\forall x y z. x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

**shows**  $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$

**proof**–

**have**  $x \sqcap (y \sqcup z) = (x \sqcap (x \sqcup z)) \sqcap (y \sqcup z)$  **by**(*simp add:inf-sup-absorb*)  
**also have**  $\dots = x \sqcap (z \sqcup (x \sqcap y))$  **by**(*simp add:D sup-commute inf-assoc*)  
**also have**  $\dots = ((x \sqcap y) \sqcup x) \sqcap ((x \sqcap y) \sqcup z)$   
**by**(*simp add:sup-inf-absorb sup-commute*)  
**also have**  $\dots = (x \sqcap y) \sqcup (x \sqcap z)$  **by**(*simp add:D*)  
**finally show** *?thesis* .

**qed**

**lemma** *modular-le*:  $x \sqsubseteq z \implies x \sqcup (y \sqcap z) \sqsubseteq (x \sqcup y) \sqcap z$   
**by** *blast*

**end**

## 4.2 Distributive lattices

**class** *distrib-lattice* = *lattice* +

**assumes** *sup-inf-distrib1*:  $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

**context** *distrib-lattice*

**begin**

**lemma** *sup-inf-distrib2*:

$(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$   
**by**(*simp add:ACI sup-inf-distrib1*)

**lemma** *inf-sup-distrib1*:  
 $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$   
**by**(*rule distrib-imp2[OF sup-inf-distrib1]*)

**lemma** *inf-sup-distrib2*:  
 $(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$   
**by**(*simp add:ACI inf-sup-distrib1*)

**lemmas** *distrib* =  
*sup-inf-distrib1 sup-inf-distrib2 inf-sup-distrib1 inf-sup-distrib2*

**end**

### 4.3 Uniqueness of inf and sup

**lemma** (*in lower-semilattice*) *inf-unique*:  
**fixes** *f* (**infixl**  $\triangle$  70)  
**assumes** *le1*:  $\bigwedge x y. x \triangle y \leq x$  **and** *le2*:  $\bigwedge x y. x \triangle y \leq y$   
**and** *greatest*:  $\bigwedge x y z. x \leq y \implies x \leq z \implies x \leq y \triangle z$   
**shows**  $x \sqcap y = x \triangle y$   
**proof** (*rule antisym*)  
**show**  $x \triangle y \leq x \sqcap y$  **by** (*rule le-infI*) (*rule le1, rule le2*)  
**next**  
**have** *leI*:  $\bigwedge x y z. x \leq y \implies x \leq z \implies x \leq y \triangle z$  **by** (*blast intro: greatest*)  
**show**  $x \sqcap y \leq x \triangle y$  **by** (*rule leI*) *simp-all*  
**qed**

**lemma** (*in upper-semilattice*) *sup-unique*:  
**fixes** *f* (**infixl**  $\nabla$  70)  
**assumes** *ge1* [*simp*]:  $\bigwedge x y. x \leq x \nabla y$  **and** *ge2*:  $\bigwedge x y. y \leq x \nabla y$   
**and** *least*:  $\bigwedge x y z. y \leq x \implies z \leq x \implies y \nabla z \leq x$   
**shows**  $x \sqcup y = x \nabla y$   
**proof** (*rule antisym*)  
**show**  $x \sqcup y \leq x \nabla y$  **by** (*rule le-supI*) (*rule ge1, rule ge2*)  
**next**  
**have** *leI*:  $\bigwedge x y z. x \leq z \implies y \leq z \implies x \nabla y \leq z$  **by** (*blast intro: least*)  
**show**  $x \nabla y \leq x \sqcup y$  **by** (*rule leI*) *simp-all*  
**qed**

### 4.4 *min/max* on linear orders as special case of $op \sqcap / op \sqcup$

**lemma** (*in linorder*) *distrib-lattice-min-max*:  
*distrib-lattice* ( $op \leq$ ) ( $op <$ ) *min max*  
**proof**  
**have** *aux*:  $\bigwedge x y :: 'a. x < y \implies y \leq x \implies x = y$   
**by** (*auto simp add: less-le antisym*)  
**fix**  $x y z$



```

show  $\max x (\min y z) = \min (\max x y) (\max x z)$ 
unfolding min-def max-def
by auto
qed (auto simp add: min-def max-def not-le less-imp-le)

interpretation min-max: distrib-lattice op ≤ :: 'a::linorder ⇒ 'a ⇒ bool op <
min max
by (rule distrib-lattice-min-max)

lemma inf-min: inf = (min :: 'a::{lower-semilattice, linorder} ⇒ 'a ⇒ 'a)
by (rule ext)+ (auto intro: antisym)

lemma sup-max: sup = (max :: 'a::{upper-semilattice, linorder} ⇒ 'a ⇒ 'a)
by (rule ext)+ (auto intro: antisym)

lemmas le-maxI1 = min-max.sup-ge1
lemmas le-maxI2 = min-max.sup-ge2

lemmas max-ac = min-max.sup-assoc min-max.sup-commute
mk-left-commute [of max, OF min-max.sup-assoc min-max.sup-commute]

lemmas min-ac = min-max.inf-assoc min-max.inf-commute
mk-left-commute [of min, OF min-max.inf-assoc min-max.inf-commute]

Now we have inherited antisymmetry as an intro-rule on all linear orders.
This is a problem because it applies to bool, which is undesirable.

lemmas [rule del] = min-max.le-infI min-max.le-supI
min-max.le-supE min-max.le-infE min-max.le-supI1 min-max.le-supI2
min-max.le-infI1 min-max.le-infI2

```

## 4.5 Bool as lattice

```

instantiation bool :: distrib-lattice
begin

```

**definition**

```

inf-bool-eq: P □ Q ⟷ P ∧ Q

```

**definition**

```

sup-bool-eq: P ⊔ Q ⟷ P ∨ Q

```

**instance**

```

by intro-classes (auto simp add: inf-bool-eq sup-bool-eq le-bool-def)

```

**end**

## 4.6 Fun as lattice

```

instantiation fun :: (type, lattice) lattice

```

**begin**

**definition**

*inf-fun-eq* [code del]:  $f \sqcap g = (\lambda x. f\ x \sqcap g\ x)$

**definition**

*sup-fun-eq* [code del]:  $f \sqcup g = (\lambda x. f\ x \sqcup g\ x)$

**instance**

**apply** *intro-classes*

**unfolding** *inf-fun-eq sup-fun-eq*

**apply** (*auto intro: le-funI*)

**apply** (*rule le-funI*)

**apply** (*auto dest: le-funD*)

**apply** (*rule le-funI*)

**apply** (*auto dest: le-funD*)

**done**

**end**

**instance** *fun* :: (*type*, *distrib-lattice*) *distrib-lattice*

**by** *default* (*auto simp add: inf-fun-eq sup-fun-eq sup-inf-distrib1*)

redundant bindings

**lemmas** *inf-aci* = *inf-ACI*

**lemmas** *sup-aci* = *sup-ACI*

**no-notation**

*less-eq* (**infix**  $\sqsubseteq$  50) **and**

*less* (**infix**  $\sqsubset$  50) **and**

*inf* (**infixl**  $\sqcap$  70) **and**

*sup* (**infixl**  $\sqcup$  65)

**end**

## 5 Set: Set theory for higher-order logic

**theory** *Set*

**imports** *Lattices*

**begin**

A set in HOL is simply a predicate.

### 5.1 Basic syntax

**global**

**types**  $'a \text{ set} = 'a \Rightarrow \text{bool}$

**consts**

$\text{Collect} \quad :: ('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ set} \quad \text{— comprehension}$   
 $\text{op} : \quad :: 'a \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \quad \text{— membership}$   
 $\text{insert} \quad :: 'a \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$   
 $\text{Ball} \quad :: 'a \text{ set} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool} \quad \text{— bounded universal quantifiers}$   
  
 $\text{Bex} \quad :: 'a \text{ set} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool} \quad \text{— bounded existential}$   
 quantifiers  
 $\text{Bex1} \quad :: 'a \text{ set} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow \text{bool} \quad \text{— bounded unique existential}$   
 quantifiers  
 $\text{Pow} \quad :: 'a \text{ set} \Rightarrow 'a \text{ set set} \quad \text{— powerset}$   
 $\text{image} \quad :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set} \quad (\text{infixr } '90)$

**local**

**notation**

$\text{op} : (\text{op} :) \text{ and}$   
 $\text{op} : ((-/ : -) [50, 51] 50)$

**abbreviation**

$\text{not-mem } x \ A == \sim (x : A) \text{ — non-membership}$

**notation**

$\text{not-mem } (\text{op } \sim :) \text{ and}$   
 $\text{not-mem } ((-/ \sim : -) [50, 51] 50)$

**notation** (*xsymbols*)

$\text{op} : (\text{op } \in) \text{ and}$   
 $\text{op} : ((-/ \in -) [50, 51] 50) \text{ and}$   
 $\text{not-mem } (\text{op } \notin) \text{ and}$   
 $\text{not-mem } ((-/ \notin -) [50, 51] 50)$

**notation** (*HTML output*)

$\text{op} : (\text{op } \in) \text{ and}$   
 $\text{op} : ((-/ \in -) [50, 51] 50) \text{ and}$   
 $\text{not-mem } (\text{op } \notin) \text{ and}$   
 $\text{not-mem } ((-/ \notin -) [50, 51] 50)$

**syntax**

$@Coll \quad :: \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \text{ set} \quad ((1\{-/-\}))$

**translations**

$\{x. P\} \quad == \text{Collect } (\%x. P)$

**definition**  $\text{empty} :: 'a \text{ set } (\{\})$  **where**

$\text{empty} \equiv \{x. \text{False}\}$

**definition**  $UNIV :: 'a \text{ set}$  **where**  
 $UNIV \equiv \{x. \text{True}\}$

**syntax**

$@Finset \quad :: \text{args} \Rightarrow 'a \text{ set} \quad (\{(-)\})$

**translations**

$\{x, xs\} \quad == \text{insert } x \ \{xs\}$   
 $\{x\} \quad == \text{insert } x \ \{\}$

**definition**  $Int :: 'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$  (**infixl**  $Int \ 70$ ) **where**  
 $A \ Int \ B \equiv \{x. x \in A \wedge x \in B\}$

**definition**  $Un :: 'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$  (**infixl**  $Un \ 65$ ) **where**  
 $A \ Un \ B \equiv \{x. x \in A \vee x \in B\}$

**notation** ( $xsymbols$ )

$Int$  (**infixl**  $\cap \ 70$ ) **and**  
 $Un$  (**infixl**  $\cup \ 65$ )

**notation** ( $HTML \text{ output}$ )

$Int$  (**infixl**  $\cap \ 70$ ) **and**  
 $Un$  (**infixl**  $\cup \ 65$ )

**syntax**

$-Ball \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists ALL \text{ :-./ } -) [0, 0, 10] \ 10)$   
 $-Bex \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists EX \text{ :-./ } -) [0, 0, 10] \ 10)$   
 $-Bex1 \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists EX! \text{ :-./ } -) [0, 0, 10] \ 10)$   
 $-Bleast \quad :: \text{id} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow 'a \quad ((\exists LEAST \text{ :-./ } -) [0, 0, 10] \ 10)$

**syntax** ( $HOL$ )

$-Ball \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists! \text{ :-./ } -) [0, 0, 10] \ 10)$   
 $-Bex \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists? \text{ :-./ } -) [0, 0, 10] \ 10)$   
 $-Bex1 \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists?! \text{ :-./ } -) [0, 0, 10] \ 10)$

**syntax** ( $xsymbols$ )

$-Ball \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists \forall \text{ -}\in\text{./ } -) [0, 0, 10] \ 10)$   
 $-Bex \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists \exists \text{ -}\in\text{./ } -) [0, 0, 10] \ 10)$   
 $-Bex1 \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists \exists! \text{ -}\in\text{./ } -) [0, 0, 10] \ 10)$   
 $-Bleast \quad :: \text{id} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow 'a \quad ((\exists LEAST \text{ -}\in\text{./ } -) [0, 0, 10] \ 10)$

**syntax** ( $HTML \text{ output}$ )

$-Ball \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists \forall \text{ -}\in\text{./ } -) [0, 0, 10] \ 10)$   
 $-Bex \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists \exists \text{ -}\in\text{./ } -) [0, 0, 10] \ 10)$   
 $-Bex1 \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow \text{bool} \Rightarrow \text{bool} \quad ((\exists \exists! \text{ -}\in\text{./ } -) [0, 0, 10] \ 10)$

**translations**

$ALL\ x:A. P == Ball\ A\ (\%x. P)$   
 $EX\ x:A. P == Bex\ A\ (\%x. P)$   
 $EX!\ x:A. P == Bex1\ A\ (\%x. P)$   
 $LEAST\ x:A. P => LEAST\ x. x:A \ \&\ P$

**definition** *INTER* :: 'a set  $\Rightarrow$  ('a  $\Rightarrow$  'b set)  $\Rightarrow$  'b set **where**  
 $INTER\ A\ B \equiv \{y. \forall x \in A. y \in B\ x\}$

**definition** *UNION* :: 'a set  $\Rightarrow$  ('a  $\Rightarrow$  'b set)  $\Rightarrow$  'b set **where**  
 $UNION\ A\ B \equiv \{y. \exists x \in A. y \in B\ x\}$

**definition** *Inter* :: 'a set set  $\Rightarrow$  'a set **where**  
 $Inter\ S \equiv INTER\ S\ (\lambda x. x)$

**definition** *Union* :: 'a set set  $\Rightarrow$  'a set **where**  
 $Union\ S \equiv UNION\ S\ (\lambda x. x)$

**notation** (*xsymbols*)  
 $Inter\ (\bigcap - [90] 90)$  **and**  
 $Union\ (\bigcup - [90] 90)$

## 5.2 Additional concrete syntax

### syntax

$@SetCompr :: 'a \Rightarrow idts \Rightarrow bool \Rightarrow 'a\ set \quad ((1\ \{-\ |/\ -/\ -\}))$   
 $@Collect :: idt \Rightarrow 'a\ set \Rightarrow bool \Rightarrow 'a\ set \quad ((1\ \{-\ :/\ -/\ -\}))$   
 $@INTER1 :: pptrns \Rightarrow 'b\ set \Rightarrow 'b\ set \quad ((3INT\ -/\ -) [0, 10] 10)$   
 $@UNION1 :: pptrns \Rightarrow 'b\ set \Rightarrow 'b\ set \quad ((3UN\ -/\ -) [0, 10] 10)$   
 $@INTER :: pptrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow 'b\ set \quad ((3INT\ :-/\ -) [0, 10] 10)$   
 $@UNION :: pptrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow 'b\ set \quad ((3UN\ :-/\ -) [0, 10] 10)$

### syntax (*xsymbols*)

$@Collect :: idt \Rightarrow 'a\ set \Rightarrow bool \Rightarrow 'a\ set \quad ((1\ \{-\ \in/\ -/\ -\}))$   
 $@INTER1 :: pptrns \Rightarrow 'b\ set \Rightarrow 'b\ set \quad ((3\bigcap\ -/\ -) [0, 10] 10)$   
 $@UNION1 :: pptrns \Rightarrow 'b\ set \Rightarrow 'b\ set \quad ((3\bigcup\ -/\ -) [0, 10] 10)$   
 $@INTER :: pptrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow 'b\ set \quad ((3\bigcap\ -\in/\ -) [0, 10] 10)$   
 $@UNION :: pptrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow 'b\ set \quad ((3\bigcup\ -\in/\ -) [0, 10] 10)$

### syntax (*latex output*)

$@INTER1 :: pptrns \Rightarrow 'b\ set \Rightarrow 'b\ set \quad ((3\bigcap\ (00\ -)/\ -) [0, 10] 10)$   
 $@UNION1 :: pptrns \Rightarrow 'b\ set \Rightarrow 'b\ set \quad ((3\bigcup\ (00\ -)/\ -) [0, 10] 10)$   
 $@INTER :: pptrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow 'b\ set \quad ((3\bigcap\ (00\ -\in)/\ -) [0, 10] 10)$   
 $@UNION :: pptrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow 'b\ set \quad ((3\bigcup\ (00\ -\in)/\ -) [0, 10] 10)$

### translations

$\{x:A. P\} \Rightarrow \{x. x:A \ \&\ P\}$   
 $INT\ x\ y. B == INT\ x. INT\ y. B$

$$\begin{aligned}
INT\ x.\ B &== CONST\ INTER\ CONST\ UNIV\ (\%x.\ B) \\
INT\ x.\ B &== INT\ x:CONST\ UNIV.\ B \\
INT\ x:A.\ B &== CONST\ INTER\ A\ (\%x.\ B) \\
UN\ x\ y.\ B &== UN\ x.\ UN\ y.\ B \\
UN\ x.\ B &== CONST\ UNION\ CONST\ UNIV\ (\%x.\ B) \\
UN\ x.\ B &== UN\ x:CONST\ UNIV.\ B \\
UN\ x:A.\ B &== CONST\ UNION\ A\ (\%x.\ B)
\end{aligned}$$

Note the difference between ordinary xsymbol syntax of indexed unions and intersections (e.g.  $\bigcup_{a_1 \in A_1} B$ ) and their L<sup>A</sup>T<sub>E</sub>X rendition:  $\bigcup_{a_1 \in A_1} B$ . The former does not make the index expression a subscript of the union/intersection symbol because this leads to problems with nested subscripts in Proof General.

#### abbreviation

*subset* :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool **where**  
*subset*  $\equiv$  less

#### abbreviation

*subset-eq* :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool **where**  
*subset-eq*  $\equiv$  less-eq

#### notation (output)

*subset* (op <) **and**  
*subset* ((-/ < -) [50, 51] 50) **and**  
*subset-eq* (op <=) **and**  
*subset-eq* ((-/ <= -) [50, 51] 50)

#### notation (xsymbols)

*subset* (op  $\subset$ ) **and**  
*subset* ((-/  $\subset$  -) [50, 51] 50) **and**  
*subset-eq* (op  $\subseteq$ ) **and**  
*subset-eq* ((-/  $\subseteq$  -) [50, 51] 50)

#### notation (HTML output)

*subset* (op  $\subset$ ) **and**  
*subset* ((-/  $\subset$  -) [50, 51] 50) **and**  
*subset-eq* (op  $\subseteq$ ) **and**  
*subset-eq* ((-/  $\subseteq$  -) [50, 51] 50)

#### abbreviation (input)

*supset* :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool **where**  
*supset*  $\equiv$  greater

#### abbreviation (input)

*supset-eq* :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool **where**  
*supset-eq*  $\equiv$  greater-eq

#### notation (xsymbols)

*supset* (op  $\supset$ ) **and**

*supset*  $((-/ \supset -) [50, 51] 50)$  **and**  
*supset-eq*  $(op \supseteq)$  **and**  
*supset-eq*  $((-/ \supseteq -) [50, 51] 50)$

**abbreviation**

*range*  $:: ('a \Rightarrow 'b) \Rightarrow 'b$  **set where** — of function  
*range*  $f == f \text{ ' UNIV}$

**5.2.1 Bounded quantifiers****syntax (output)**

*-setlessAll*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists ALL \text{ -<-./ -}) [0, 0, 10] 10)$   
*-setlessEx*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists EX \text{ -<-./ -}) [0, 0, 10] 10)$   
*-setleAll*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists ALL \text{ -<=.-./ -}) [0, 0, 10] 10)$   
*-setleEx*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists EX \text{ -<=.-./ -}) [0, 0, 10] 10)$   
*-setleEx1*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists EX! \text{ -<=.-./ -}) [0, 0, 10] 10)$

**syntax (xsymbols)**

*-setlessAll*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists \forall \text{ -<-./ -}) [0, 0, 10] 10)$   
*-setlessEx*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists \exists \text{ -<-./ -}) [0, 0, 10] 10)$   
*-setleAll*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists \forall \text{ -<=.-./ -}) [0, 0, 10] 10)$   
*-setleEx*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists \exists \text{ -<=.-./ -}) [0, 0, 10] 10)$   
*-setleEx1*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists \exists! \text{ -<=.-./ -}) [0, 0, 10] 10)$

**syntax (HOL output)**

*-setlessAll*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists! \text{ -<-./ -}) [0, 0, 10] 10)$   
*-setlessEx*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists? \text{ -<-./ -}) [0, 0, 10] 10)$   
*-setleAll*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists! \text{ -<=.-./ -}) [0, 0, 10] 10)$   
*-setleEx*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists? \text{ -<=.-./ -}) [0, 0, 10] 10)$   
*-setleEx1*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists?! \text{ -<=.-./ -}) [0, 0, 10] 10)$

**syntax (HTML output)**

*-setlessAll*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists \forall \text{ -<-./ -}) [0, 0, 10] 10)$   
*-setlessEx*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists \exists \text{ -<-./ -}) [0, 0, 10] 10)$   
*-setleAll*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists \forall \text{ -<=.-./ -}) [0, 0, 10] 10)$   
*-setleEx*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists \exists \text{ -<=.-./ -}) [0, 0, 10] 10)$   
*-setleEx1*  $:: [idt, 'a, bool] \Rightarrow bool \ ((\exists \exists! \text{ -<=.-./ -}) [0, 0, 10] 10)$

**translations**

$\forall A \subset B. P \Rightarrow ALL A. A \subset B \dashrightarrow P$   
 $\exists A \subset B. P \Rightarrow EX A. A \subset B \ \& \ P$   
 $\forall A \subseteq B. P \Rightarrow ALL A. A \subseteq B \dashrightarrow P$   
 $\exists A \subseteq B. P \Rightarrow EX A. A \subseteq B \ \& \ P$   
 $\exists! A \subseteq B. P \Rightarrow EX! A. A \subseteq B \ \& \ P$

**print-translation**  $\ll$ 

*let*

*val* *Type* (*set-type*, *-*) =  $@\{typ \ 'a \ set\};$   
*val* *All-binder* = *Syntax.binder-name*  $@\{const-syntax \ All\};$

```

val Ex-binder = Syntax.binder-name @ {const-syntax Ex};
val impl = @ {const-syntax op -->};
val conj = @ {const-syntax op &};
val sbset = @ {const-syntax subset};
val sbset-eq = @ {const-syntax subset-eq};

val trans =
  [((All-binder, impl, sbset), -setlessAll),
   ((All-binder, impl, sbset-eq), -settleAll),
   ((Ex-binder, conj, sbset), -setlessEx),
   ((Ex-binder, conj, sbset-eq), -settleEx)];

fun mk v v' c n P =
  if v = v' andalso not (Term.exists-subterm (fn Free (x, -) => x = v | - =>
false) n)
  then Syntax.const c $ Syntax.mark-bound v' $ n $ P else raise Match;

fun tr' q = (q,
  fn [Const (-bound, -) $ Free (v, Type (T, -)), Const (c, -) $ (Const (d, -) $
(Const (-bound, -) $ Free (v', -)) $ n) $ P] =>
    if T = (set-type) then case AList.lookup (op =) trans (q, c, d)
      of NONE => raise Match
        | SOME l => mk v v' l n P
    else raise Match
  | - => raise Match);
in
  [tr' All-binder, tr' Ex-binder]
end
>>

Translate between  $\{e \mid x1...xn. P\}$  and  $\{u. EX\ x1..xn. u = e \ \& \ P\}$ ;  $\{y. EX\ x1..xn. y = e \ \& \ P\}$  is only translated if  $[0..n]$  subset  $bvs(e)$ .

parse-translation <<
  let
    val ex-tr = snd (mk-binder-tr (EX , Ex));

    fun nvars (Const (-idts, -) $ - $ idts) = nvars idts + 1
      | nvars - = 1;

    fun setcompr-tr [e, idts, b] =
      let
        val eq = Syntax.const op = $ Bound (nvars idts) $ e;
        val P = Syntax.const op & $ eq $ b;
        val exP = ex-tr [idts, P];
      in Syntax.const Collect $ Term.absdummy (dummyT, exP) end;

  in [(@SetCompr, setcompr-tr)] end;
>>

```



```

print-translation ⟨⟨
  let
    fun btr' syn [A, Abs abs] =
      let val (x, t) = atomic-abs-tr' abs
      in Syntax.const syn $ x $ A $ t end
  in
    [(@{const-syntax Ball}, btr' -Ball), (@{const-syntax Bex}, btr' -Bex),
     (@{const-syntax UNION}, btr' @UNION), (@{const-syntax INTER}, btr' @INTER)]
  end
⟩⟩

```

```

print-translation ⟨⟨
  let
    val ex-tr' = snd (mk-binder-tr' (Ex, DUMMY));

    fun setcompr-tr' [Abs (abs as (-, -, P))] =
      let
        fun check (Const (Ex, -) $ Abs (-, -, P), n) = check (P, n + 1)
          | check (Const (op &, -) $ (Const (op =, -) $ Bound m $ e) $ P, n) =
              n > 0 andalso m = n andalso not (loose-bvar1 (P, n)) andalso
              ((0 upto (n - 1)) subset add-loose-bnos (e, 0, []))
          | check - = false

        fun tr' (- $ abs) =
          let val - $ idts $ (- $ (- $ - $ e) $ Q) = ex-tr' [abs]
          in Syntax.const @SetCompr $ e $ idts $ Q end;
      in if check (P, 0) then tr' P
        else let val (x as - $ Free(xN, -), t) = atomic-abs-tr' abs
          val M = Syntax.const @Coll $ x $ t
          in case t of
              Const(op &, -)
                $ (Const(op :-, -) $ (Const(-bound, -) $ Free(yN, -)) $ A)
                $ P =>
              if xN=yN then Syntax.const @Collect $ x $ A $ P else M
            | - => M
          end
        end;
      in [(Collect, setcompr-tr')] end;
    ⟩⟩

```

### 5.3 Rules and definitions

Isomorphisms between predicates and sets.

**defs**

```

  mem-def [code]: x : S == S x
  Collect-def [code]: Collect P == P

```

**defs**

*Ball-def:*      $Ball\ A\ P \quad ==\ ALL\ x.\ x:A \dashrightarrow P(x)$   
*Bex-def:*      $Bex\ A\ P \quad ==\ EX\ x.\ x:A \ \&\ P(x)$   
*Bex1-def:*     $Bex1\ A\ P \quad ==\ EX!\ x.\ x:A \ \&\ P(x)$

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

**definition**

*fun-diff-def:*  $A - B = (\%x.\ A\ x - B\ x)$

**instance** ..

**end**

**instantiation** *bool* :: *minus*  
**begin**

**definition**

*bool-diff-def:*  $A - B = (A \ \&\ \sim B)$

**instance** ..

**end**

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

**definition**

*fun-Compl-def:*  $-\ A = (\%x.\ -\ A\ x)$

**instance** ..

**end**

**instantiation** *bool* :: *uminus*  
**begin**

**definition**

*bool-Compl-def:*  $-\ A = (\sim A)$

**instance** ..

**end**

**defs**

*Pow-def:*      $Pow\ A \quad ==\ \{B.\ B \leq A\}$   
*insert-def:*    $insert\ a\ B \quad ==\ \{x.\ x=a\} \cup B$   
*image-def:*     $f^{\ast}A \quad ==\ \{y.\ EX\ x:A.\ y = f(x)\}$

## 5.4 Lemmas and proof tool setup

### 5.4.1 Relating predicates and sets

**lemma** *mem-Collect-eq* [*iff*]:  $(a : \{x. P(x)\}) = P(a)$   
**by** (*simp add: Collect-def mem-def*)

**lemma** *Collect-mem-eq* [*simp*]:  $\{x. x:A\} = A$   
**by** (*simp add: Collect-def mem-def*)

**lemma** *CollectI*:  $P(a) ==> a : \{x. P(x)\}$   
**by** *simp*

**lemma** *CollectD*:  $a : \{x. P(x)\} ==> P(a)$   
**by** *simp*

**lemma** *Collect-cong*:  $(!!x. P x = Q x) ==> \{x. P(x)\} = \{x. Q(x)\}$   
**by** *simp*

**lemmas** *CollectE* = *CollectD* [*elim-format*]

### 5.4.2 Bounded quantifiers

**lemma** *ballI* [*intro!*]:  $(!!x. x:A ==> P x) ==> ALL x:A. P x$   
**by** (*simp add: Ball-def*)

**lemmas** *strip* = *impI allI ballI*

**lemma** *bspec* [*dest?*]:  $ALL x:A. P x ==> x:A ==> P x$   
**by** (*simp add: Ball-def*)

**lemma** *ballE* [*elim*]:  $ALL x:A. P x ==> (P x ==> Q) ==> (x \sim: A ==> Q) ==> Q$   
**by** (*unfold Ball-def*) *blast*

**ML**  $\ll \text{bind-thm (rev-ballE, permute-prems 1 1 @\{thm ballE\})} \gg$

This tactic takes assumptions  $\forall x \in A. P x$  and  $a \in A$ ; creates assumption  $P a$ .

**ML**  $\ll$   
*fun ball-tac i = etac @\{thm ballE\} i THEN contr-tac (i + 1)*  
 $\gg$

Gives better instantiation for bound:

**declaration**  $\ll \text{fn - ==>}$   
*Classical.map-cs (fn cs ==> cs addbefore (bspec, datac @\{thm bspec\} 1))*  
 $\gg$

**lemma** *beXI* [*intro*]:  $P x ==> x:A ==> EX x:A. P x$

— Normally the best argument order:  $P\ x$  constrains the choice of  $x \in A$ .  
**by** (*unfold Bex-def*) *blast*

**lemma** *rev-bexI* [*intro?*]:  $x:A \implies P\ x \implies EX\ x:A. P\ x$   
 — The best argument order when there is only one  $x \in A$ .  
**by** (*unfold Bex-def*) *blast*

**lemma** *bexCI*:  $(ALL\ x:A. \sim P\ x \implies P\ a) \implies a:A \implies EX\ x:A. P\ x$   
**by** (*unfold Bex-def*) *blast*

**lemma** *bexE* [*elim!*]:  $EX\ x:A. P\ x \implies (!x. x:A \implies P\ x \implies Q) \implies Q$   
**by** (*unfold Bex-def*) *blast*

**lemma** *ball-triv* [*simp*]:  $(ALL\ x:A. P) = ((EX\ x. x:A) \longrightarrow P)$   
 — Trivial rewrite rule.  
**by** (*simp add: Ball-def*)

**lemma** *bex-triv* [*simp*]:  $(EX\ x:A. P) = ((EX\ x. x:A) \& P)$   
 — Dual form for existentials.  
**by** (*simp add: Bex-def*)

**lemma** *bex-triv-one-point1* [*simp*]:  $(EX\ x:A. x = a) = (a:A)$   
**by** *blast*

**lemma** *bex-triv-one-point2* [*simp*]:  $(EX\ x:A. a = x) = (a:A)$   
**by** *blast*

**lemma** *bex-one-point1* [*simp*]:  $(EX\ x:A. x = a \& P\ x) = (a:A \& P\ a)$   
**by** *blast*

**lemma** *bex-one-point2* [*simp*]:  $(EX\ x:A. a = x \& P\ x) = (a:A \& P\ a)$   
**by** *blast*

**lemma** *ball-one-point1* [*simp*]:  $(ALL\ x:A. x = a \longrightarrow P\ x) = (a:A \longrightarrow P\ a)$   
**by** *blast*

**lemma** *ball-one-point2* [*simp*]:  $(ALL\ x:A. a = x \longrightarrow P\ x) = (a:A \longrightarrow P\ a)$   
**by** *blast*

**ML**  $\ll$   
*local*  
*val* *unfold-bex-tac* = *unfold-tac* @{*thms Bex-def*};  
*fun* *prove-bex-tac* *ss* = *unfold-bex-tac* *ss* *THEN* *Quantifier1.prove-one-point-ex-tac*;  
*val* *rearrange-bex* = *Quantifier1.rearrange-bex* *prove-bex-tac*;  
  
*val* *unfold-ball-tac* = *unfold-tac* @{*thms Ball-def*};  
*fun* *prove-ball-tac* *ss* = *unfold-ball-tac* *ss* *THEN* *Quantifier1.prove-one-point-all-tac*;  
*val* *rearrange-ball* = *Quantifier1.rearrange-ball* *prove-ball-tac*;  
*in*

```

    val defBEX-regroup = Simplifier.simproc (the-context ())
      defined BEX [EX x:A. P x & Q x] rearrange-bex;
    val defBALL-regroup = Simplifier.simproc (the-context ())
      defined BALL [ALL x:A. P x --> Q x] rearrange-ball;
  end;

  Addsimprocs [defBALL-regroup, defBEX-regroup];
>>

```

### 5.4.3 Congruence rules

**lemma** *ball-cong*:

```

  A = B ==> (!x. x:B ==> P x = Q x) ==>
    (ALL x:A. P x) = (ALL x:B. Q x)
  by (simp add: Ball-def)

```

**lemma** *strong-ball-cong* [cong]:

```

  A = B ==> (!x. x:B =simp=> P x = Q x) ==>
    (ALL x:A. P x) = (ALL x:B. Q x)
  by (simp add: simp-implies-def Ball-def)

```

**lemma** *bex-cong*:

```

  A = B ==> (!x. x:B ==> P x = Q x) ==>
    (EX x:A. P x) = (EX x:B. Q x)
  by (simp add: Bex-def cong: conj-cong)

```

**lemma** *strong-bex-cong* [cong]:

```

  A = B ==> (!x. x:B =simp=> P x = Q x) ==>
    (EX x:A. P x) = (EX x:B. Q x)
  by (simp add: simp-implies-def Bex-def cong: conj-cong)

```

### 5.4.4 Subsets

**lemma** *subsetI* [atp,intro!]:  $(!x. x:A ==> x:B) ==> A \subseteq B$   
 by (auto simp add: mem-def intro: predicate1I)

Map the type *'a set ==> anything* to just *'a*; for overloading constants whose first argument has type *'a set*.

**lemma** *subsetD* [elim, intro?]:  $A \subseteq B ==> c \in A ==> c \in B$   
 — Rule in Modus Ponens style.  
 by (unfold mem-def) blast

**lemma** *rev-subsetD* [intro?]:  $c \in A ==> A \subseteq B ==> c \in B$   
 — The same, with reversed premises for use with *erule* – cf *rev-mp*.  
 by (rule subsetD)

Converts  $A \subseteq B$  to  $x \in A \implies x \in B$ .

**ML**  $\ll$

```

    fun impOfSubs th = th RSN (2, @{thm rev-subsetD})
  >>

```

```

lemma subsetCE [elim]:  $A \subseteq B \implies (c \notin A \implies P) \implies (c \in B \implies P)$ 
 $\implies P$ 
  — Classical elimination rule.
  by (unfold mem-def) blast

```

```

lemma subset-eq:  $A \subseteq B = (\forall x \in A. x \in B)$  by blast

```

Takes assumptions  $A \subseteq B$ ;  $c \in A$  and creates the assumption  $c \in B$ .

```

ML <<
  fun set-mp-tac i = etac @{thm subsetCE} i THEN mp-tac i
  >>

```

```

lemma contra-subsetD:  $A \subseteq B \implies c \notin B \implies c \notin A$ 
by blast

```

```

lemma subset-refl [simp,atp]:  $A \subseteq A$ 
by fast

```

```

lemma subset-trans:  $A \subseteq B \implies B \subseteq C \implies A \subseteq C$ 
by blast

```

#### 5.4.5 Equality

```

lemma set-ext: assumes prem:  $(!!x. (x:A) = (x:B))$  shows  $A = B$ 
  apply (rule prem [THEN ext, THEN arg-cong, THEN box-equals])
  apply (rule Collect-mem-eq)
  apply (rule Collect-mem-eq)
  done

```

```

lemma expand-set-eq:  $(A = B) = (ALL x. (x:A) = (x:B))$ 
by(auto intro:set-ext)

```

```

lemma subset-antisym [intro!]:  $A \subseteq B \implies B \subseteq A \implies A = B$ 
  — Anti-symmetry of the subset relation.
  by (iprover intro: set-ext subsetD)

```

Equality rules from ZF set theory – are they appropriate here?

```

lemma equalityD1:  $A = B \implies A \subseteq B$ 
by (simp add: subset-refl)

```

```

lemma equalityD2:  $A = B \implies B \subseteq A$ 
by (simp add: subset-refl)

```

Be careful when adding this to the claset as *subset-empty* is in the simpset:  
 $A = \{\}$  goes to  $\{\} \subseteq A$  and  $A \subseteq \{\}$  and then back to  $A = \{\}$ !

**lemma** *equalityE*:  $A = B \implies (A \subseteq B \implies B \subseteq A \implies P) \implies P$   
**by** (*simp add: subset-refl*)

**lemma** *equalityCE* [*elim*]:  
 $A = B \implies (c \in A \implies c \in B \implies P) \implies (c \notin A \implies c \notin B \implies P)$   
 $\implies P$   
**by** *blast*

**lemma** *eqset-imp-iff*:  $A = B \implies (x : A) = (x : B)$   
**by** *simp*

**lemma** *eqelem-imp-iff*:  $x = y \implies (x : A) = (y : A)$   
**by** *simp*

#### 5.4.6 The universal set – UNIV

**lemma** *UNIV-I* [*simp*]:  $x : UNIV$   
**by** (*simp add: UNIV-def*)

**declare** *UNIV-I* [*intro*] — unsafe makes it less likely to cause problems

**lemma** *UNIV-witness* [*intro?*]:  $EX x. x : UNIV$   
**by** *simp*

**lemma** *subset-UNIV* [*simp*]:  $A \subseteq UNIV$   
**by** (*rule subsetI*) (*rule UNIV-I*)

Eta-contracting these two rules (to remove  $P$ ) causes them to be ignored because of their interaction with congruence rules.

**lemma** *ball-UNIV* [*simp*]:  $Ball UNIV P = All P$   
**by** (*simp add: Ball-def*)

**lemma** *bex-UNIV* [*simp*]:  $Bex UNIV P = Ex P$   
**by** (*simp add: Bex-def*)

**lemma** *UNIV-eq-I*:  $(\bigwedge x. x \in A) \implies UNIV = A$   
**by** *auto*

#### 5.4.7 The empty set

**lemma** *empty-iff* [*simp*]:  $(c : \{\}) = False$   
**by** (*simp add: empty-def*)

**lemma** *emptyE* [*elim!*]:  $a : \{\} \implies P$   
**by** *simp*

**lemma** *empty-subsetI* [*iff*]:  $\{\} \subseteq A$   
 — One effect is to delete the ASSUMPTION  $\{\} \subseteq A$   
**by** *blast*

**lemma** *equals0I*: ( $\forall y. y \in A \implies \text{False}$ )  $\implies A = \{\}$   
**by** *blast*

**lemma** *equals0D*:  $A = \{\} \implies a \notin A$   
— Use for reasoning about disjointness:  $A \cap B = \{\}$   
**by** *blast*

**lemma** *ball-empty* [*simp*]:  $\text{Ball } \{\} P = \text{True}$   
**by** (*simp add: Ball-def*)

**lemma** *bex-empty* [*simp*]:  $\text{Bex } \{\} P = \text{False}$   
**by** (*simp add: Bex-def*)

**lemma** *UNIV-not-empty* [*iff*]:  $\text{UNIV} \sim = \{\}$   
**by** (*blast elim: equalityE*)

#### 5.4.8 The Powerset operator – Pow

**lemma** *Pow-iff* [*iff*]:  $(A \in \text{Pow } B) = (A \subseteq B)$   
**by** (*simp add: Pow-def*)

**lemma** *PowI*:  $A \subseteq B \implies A \in \text{Pow } B$   
**by** (*simp add: Pow-def*)

**lemma** *PowD*:  $A \in \text{Pow } B \implies A \subseteq B$   
**by** (*simp add: Pow-def*)

**lemma** *Pow-bottom*:  $\{\} \in \text{Pow } B$   
**by** *simp*

**lemma** *Pow-top*:  $A \in \text{Pow } A$   
**by** (*simp add: subset-refl*)

#### 5.4.9 Set complement

**lemma** *Compl-iff* [*simp*]:  $(c \in -A) = (c \notin A)$   
**by** (*simp add: mem-def fun-Compl-def bool-Compl-def*)

**lemma** *ComplI* [*intro!*]:  $(c \in A \implies \text{False}) \implies c \in -A$   
**by** (*unfold mem-def fun-Compl-def bool-Compl-def*) *blast*

This form, with negated conclusion, works well with the Classical prover. Negated assumptions behave like formulae on the right side of the notional turnstile ...

**lemma** *ComplD* [*dest!*]:  $c : -A \implies c \sim : A$   
**by** (*simp add: mem-def fun-Compl-def bool-Compl-def*)

**lemmas** *ComplE* = *ComplD* [*elim-format*]



**lemma** *Compl-eq*:  $- A = \{x. \sim x : A\}$  **by** *blast*

#### 5.4.10 Binary union – Un

**lemma** *Un-iff* [*simp*]:  $(c : A \text{ Un } B) = (c:A \mid c:B)$   
**by** (*unfold Un-def*) *blast*

**lemma** *UnI1* [*elim?*]:  $c:A \implies c : A \text{ Un } B$   
**by** *simp*

**lemma** *UnI2* [*elim?*]:  $c:B \implies c : A \text{ Un } B$   
**by** *simp*

Classical introduction rule: no commitment to  $A$  vs  $B$ .

**lemma** *UnCI* [*intro!*]:  $(c\sim:B \implies c:A) \implies c : A \text{ Un } B$   
**by** *auto*

**lemma** *UnE* [*elim!*]:  $c : A \text{ Un } B \implies (c:A \implies P) \implies (c:B \implies P) \implies P$   
**by** (*unfold Un-def*) *blast*

#### 5.4.11 Binary intersection – Int

**lemma** *Int-iff* [*simp*]:  $(c : A \text{ Int } B) = (c:A \ \& \ c:B)$   
**by** (*unfold Int-def*) *blast*

**lemma** *IntI* [*intro!*]:  $c:A \implies c:B \implies c : A \text{ Int } B$   
**by** *simp*

**lemma** *IntD1*:  $c : A \text{ Int } B \implies c:A$   
**by** *simp*

**lemma** *IntD2*:  $c : A \text{ Int } B \implies c:B$   
**by** *simp*

**lemma** *IntE* [*elim!*]:  $c : A \text{ Int } B \implies (c:A \implies c:B \implies P) \implies P$   
**by** *simp*

#### 5.4.12 Set difference

**lemma** *Diff-iff* [*simp*]:  $(c : A - B) = (c:A \ \& \ c\sim:B)$   
**by** (*simp add: mem-def fun-diff-def bool-diff-def*)

**lemma** *DiffI* [*intro!*]:  $c : A \implies c \sim : B \implies c : A - B$   
**by** *simp*

**lemma** *DiffD1*:  $c : A - B \implies c : A$   
**by** *simp*

**lemma** *DiffD2*:  $c : A - B \implies c : B \implies P$   
**by** *simp*

**lemma** *DiffE* [*elim!*]:  $c : A - B \implies (c:A \implies c^\sim:B \implies P) \implies P$   
**by** *simp*

**lemma** *set-diff-eq*:  $A - B = \{x. x : A \ \& \ \sim x : B\}$  **by** *blast*

**lemma** *Compl-eq-Diff-UNIV*:  $-A = (UNIV - A)$   
**by** *blast*

### 5.4.13 Augmenting a set – insert

**lemma** *insert-iff* [*simp*]:  $(a : \text{insert } b \ A) = (a = b \mid a:A)$   
**by** (*unfold insert-def*) *blast*

**lemma** *insertI1*:  $a : \text{insert } a \ B$   
**by** *simp*

**lemma** *insertI2*:  $a : B \implies a : \text{insert } b \ B$   
**by** *simp*

**lemma** *insertE* [*elim!*]:  $a : \text{insert } b \ A \implies (a = b \implies P) \implies (a:A \implies P) \implies P$   
**by** (*unfold insert-def*) *blast*

**lemma** *insertCI* [*intro!*]:  $(a^\sim:B \implies a = b) \implies a : \text{insert } b \ B$   
— Classical introduction rule.  
**by** *auto*

**lemma** *subset-insert-iff*:  $(A \subseteq \text{insert } x \ B) = (\text{if } x:A \text{ then } A - \{x\} \subseteq B \text{ else } A \subseteq B)$   
**by** *auto*

**lemma** *set-insert*:  
**assumes**  $x \in A$   
**obtains**  $B$  **where**  $A = \text{insert } x \ B$  **and**  $x \notin B$   
**proof**  
**from** *assms* **show**  $A = \text{insert } x \ (A - \{x\})$  **by** *blast*  
**next**  
**show**  $x \notin A - \{x\}$  **by** *blast*  
**qed**

**lemma** *insert-ident*:  $x \sim : A \implies x \sim : B \implies (\text{insert } x \ A = \text{insert } x \ B) = (A = B)$   
**by** *auto*

**5.4.14 Singletons, using insert**

**lemma** *singletonI* [*intro!*,*noatp*]:  $a : \{a\}$   
 — Redundant? But unlike *insertCI*, it proves the subgoal immediately!  
**by** (*rule insertI1*)

**lemma** *singletonD* [*dest!*,*noatp*]:  $b : \{a\} ==> b = a$   
**by** *blast*

**lemmas** *singletonE* = *singletonD* [*elim-format*]

**lemma** *singleton-iff*:  $(b : \{a\}) = (b = a)$   
**by** *blast*

**lemma** *singleton-inject* [*dest!*]:  $\{a\} = \{b\} ==> a = b$   
**by** *blast*

**lemma** *singleton-insert-inj-eq* [*iff*,*noatp*]:  
 $(\{b\} = \text{insert } a \ A) = (a = b \ \& \ A \subseteq \{b\})$   
**by** *blast*

**lemma** *singleton-insert-inj-eq'* [*iff*,*noatp*]:  
 $(\text{insert } a \ A = \{b\}) = (a = b \ \& \ A \subseteq \{b\})$   
**by** *blast*

**lemma** *subset-singletonD*:  $A \subseteq \{x\} ==> A = \{\} \mid A = \{x\}$   
**by** *fast*

**lemma** *singleton-conv* [*simp*]:  $\{x. x = a\} = \{a\}$   
**by** *blast*

**lemma** *singleton-conv2* [*simp*]:  $\{x. a = x\} = \{a\}$   
**by** *blast*

**lemma** *diff-single-insert*:  $A - \{x\} \subseteq B ==> x \in A ==> A \subseteq \text{insert } x \ B$   
**by** *blast*

**lemma** *doubleton-eq-iff*:  $(\{a,b\} = \{c,d\}) = (a=c \ \& \ b=d \mid a=d \ \& \ b=c)$   
**by** (*blast elim: equalityE*)

**5.4.15 Unions of families**

$\bigcup_{x:A. B \ x}$  is  $\bigcup B \ ' \ A$ .

**declare** *UNION-def* [*noatp*]

**lemma** *UN-iff* [*simp*]:  $(b : (\bigcup_{x:A. B \ x})) = (EX \ x:A. b : B \ x)$   
**by** (*unfold UNION-def*) *blast*

**lemma** *UN-I* [*intro*]:  $a:A ==> b : B \ a ==> b : (\bigcup_{x:A. B \ x})$

— The order of the premises presupposes that  $A$  is rigid;  $b$  may be flexible.  
**by** *auto*

**lemma** *UN-E* [*elim!*]:  $b : (UN\ x:A. B\ x) ==> (!!x. x:A ==> b: B\ x ==> R)$   
 $==> R$   
**by** (*unfold UNION-def*) *blast*

**lemma** *UN-cong* [*cong*]:  
 $A = B ==> (!!x. x:B ==> C\ x = D\ x) ==> (UN\ x:A. C\ x) = (UN\ x:B. D\ x)$   
**by** (*simp add: UNION-def*)

**lemma** *strong-UN-cong*:  
 $A = B ==> (!!x. x:B ==> C\ x = D\ x) ==> (UN\ x:A. C\ x) = (UN\ x:B. D\ x)$   
**by** (*simp add: UNION-def simp-implies-def*)

#### 5.4.16 Intersections of families

$INT\ x:A. B\ x$  is  $\bigcap B \text{ ‘ } A$ .

**lemma** *INT-iff* [*simp*]:  $(b: (INT\ x:A. B\ x)) = (ALL\ x:A. b: B\ x)$   
**by** (*unfold INTER-def*) *blast*

**lemma** *INT-I* [*intro!*]:  $(!!x. x:A ==> b: B\ x) ==> b : (INT\ x:A. B\ x)$   
**by** (*unfold INTER-def*) *blast*

**lemma** *INT-D* [*elim*]:  $b : (INT\ x:A. B\ x) ==> a:A ==> b: B\ a$   
**by** *auto*

**lemma** *INT-E* [*elim*]:  $b : (INT\ x:A. B\ x) ==> (b: B\ a ==> R) ==> (a \sim : A ==> R) ==> R$   
 — ”Classical” elimination – by the Excluded Middle on  $a \in A$ .  
**by** (*unfold INTER-def*) *blast*

**lemma** *INT-cong* [*cong*]:  
 $A = B ==> (!!x. x:B ==> C\ x = D\ x) ==> (INT\ x:A. C\ x) = (INT\ x:B. D\ x)$   
**by** (*simp add: INTER-def*)

#### 5.4.17 Union

**lemma** *Union-iff* [*simp, noatp*]:  $(A : Union\ C) = (EX\ X:C. A:X)$   
**by** (*unfold Union-def*) *blast*

**lemma** *UnionI* [*intro*]:  $X:C ==> A:X ==> A : Union\ C$   
 — The order of the premises presupposes that  $C$  is rigid;  $A$  may be flexible.  
**by** *auto*

**lemma** *UnionE* [*elim!*]:  $A : \text{Union } C \implies (!X. A:X \implies X:C \implies R) \implies R$   
**by** (*unfold Union-def*) *blast*

#### 5.4.18 Inter

**lemma** *Inter-iff* [*simp, noatp*]:  $(A : \text{Inter } C) = (\text{ALL } X:C. A:X)$   
**by** (*unfold Inter-def*) *blast*

**lemma** *InterI* [*intro!*]:  $(!X. X:C \implies A:X) \implies A : \text{Inter } C$   
**by** (*simp add: Inter-def*)

A “destruct” rule – every  $X$  in  $C$  contains  $A$  as an element, but  $A \in X$  can hold when  $X \in C$  does not! This rule is analogous to *spec*.

**lemma** *InterD* [*elim*]:  $A : \text{Inter } C \implies X:C \implies A:X$   
**by** *auto*

**lemma** *InterE* [*elim*]:  $A : \text{Inter } C \implies (X \sim C \implies R) \implies (A:X \implies R) \implies R$   
 — “Classical” elimination rule – does not require proving  $X \in C$ .  
**by** (*unfold Inter-def*) *blast*

Image of a set under a function. Frequently  $b$  does not have the syntactic form of  $f x$ .

**declare** *image-def* [*noatp*]

**lemma** *image-eqI* [*simp, intro*]:  $b = f x \implies x:A \implies b : f^*A$   
**by** (*unfold image-def*) *blast*

**lemma** *imageI*:  $x : A \implies f x : f^* A$   
**by** (*rule image-eqI*) (*rule refl*)

**lemma** *rev-image-eqI*:  $x:A \implies b = f x \implies b : f^*A$   
 — This version’s more effective when we already have the required  $x$ .  
**by** (*unfold image-def*) *blast*

**lemma** *imageE* [*elim!*]:  
 $b : (\%x. f x)^*A \implies (!x. b = f x \implies x:A \implies P) \implies P$   
 — The eta-expansion gives variable-name preservation.  
**by** (*unfold image-def*) *blast*

**lemma** *image-Un*:  $f^*(A \text{ Un } B) = f^*A \text{ Un } f^*B$   
**by** *blast*

**lemma** *image-eq-UN*:  $f^*A = (\text{UN } x:A. \{f x\})$   
**by** *blast*

**lemma** *image-iff*:  $(z : f^*A) = (\text{EX } x:A. z = f x)$

by *blast*

**lemma** *image-subset-iff*:  $(f^{\circ}A \subseteq B) = (\forall x \in A. f\ x \in B)$   
 — This rewrite rule would confuse users if made default.  
 by *blast*

**lemma** *subset-image-iff*:  $(B \subseteq f^{\circ}A) = (EX\ AA. AA \subseteq A \ \& \ B = f^{\circ}AA)$   
 apply *safe*  
 prefer 2 apply *fast*  
 apply (rule-tac  $x = \{a. a : A \ \& \ f\ a : B\}$  in *exI*, *fast*)  
 done

**lemma** *image-subsetI*:  $(!!x. x \in A ==> f\ x \in B) ==> f^{\circ}A \subseteq B$   
 — Replaces the three steps *subsetI*, *imageE*, *hypsubst*, but breaks too many existing proofs.  
 by *blast*

Range of a function – just a translation for image!

**lemma** *range-eqI*:  $b = f\ x ==> b \in \text{range } f$   
 by *simp*

**lemma** *rangeI*:  $f\ x \in \text{range } f$   
 by *simp*

**lemma** *rangeE* [*elim?*]:  $b \in \text{range } (\lambda x. f\ x) ==> (!!x. b = f\ x ==> P) ==> P$   
 by *blast*

#### 5.4.19 Set reasoning tools

Rewrite rules for boolean case-splitting: faster than *split-if* [*split*].

**lemma** *split-if-eq1*:  $((\text{if } Q \text{ then } x \text{ else } y) = b) = ((Q \longrightarrow x = b) \ \& \ (\sim Q \longrightarrow y = b))$   
 by (rule *split-if*)

**lemma** *split-if-eq2*:  $(a = (\text{if } Q \text{ then } x \text{ else } y)) = ((Q \longrightarrow a = x) \ \& \ (\sim Q \longrightarrow a = y))$   
 by (rule *split-if*)

Split ifs on either side of the membership relation. Not for [*simp*] – can cause goals to blow up!

**lemma** *split-if-mem1*:  $((\text{if } Q \text{ then } x \text{ else } y) : b) = ((Q \longrightarrow x : b) \ \& \ (\sim Q \longrightarrow y : b))$   
 by (rule *split-if*)

**lemma** *split-if-mem2*:  $(a : (\text{if } Q \text{ then } x \text{ else } y)) = ((Q \longrightarrow a : x) \ \& \ (\sim Q \longrightarrow a : y))$   
 by (rule *split-if* [where  $P = \%S. a : S$ ])

**lemmas** *split-ifs* = *if-bool-eq-conj split-if-eq1 split-if-eq2 split-if-mem1 split-if-mem2*

```
ML <<
  val mksimps-pairs = [(@{const-name Ball}, @{thms bspec})] @ mksimps-pairs;
>>
declaration << fn - ==>
  Simplifier.map-ss (fn ss ==> ss setmksimps (mksimps mksimps-pairs))
>>
```

#### 5.4.20 The “proper subset” relation

**lemma** *psubsetI* [*intro!*,*noatp*]:  $A \subseteq B \implies A \neq B \implies A \subset B$   
**by** (*unfold less-le*) *blast*

**lemma** *psubsetE* [*elim!*,*noatp*]:  
 $\llbracket A \subset B; \llbracket A \subseteq B; \sim (B \subseteq A) \rrbracket \implies R \rrbracket \implies R$   
**by** (*unfold less-le*) *blast*

**lemma** *psubset-insert-iff*:  
 $(A \subset \text{insert } x \ B) = (\text{if } x \in B \text{ then } A \subset B \text{ else if } x \in A \text{ then } A - \{x\} \subset B \text{ else } A \subseteq B)$   
**by** (*auto simp add: less-le subset-insert-iff*)

**lemma** *psubset-eq*:  $(A \subset B) = (A \subseteq B \ \& \ A \neq B)$   
**by** (*simp only: less-le*)

**lemma** *psubset-imp-subset*:  $A \subset B \implies A \subseteq B$   
**by** (*simp add: psubset-eq*)

**lemma** *psubset-trans*:  $\llbracket A \subset B; B \subset C \rrbracket \implies A \subset C$   
**apply** (*unfold less-le*)  
**apply** (*auto dest: subset-antisym*)  
**done**

**lemma** *psubsetD*:  $\llbracket A \subset B; c \in A \rrbracket \implies c \in B$   
**apply** (*unfold less-le*)  
**apply** (*auto dest: subsetD*)  
**done**

**lemma** *psubset-subset-trans*:  $A \subset B \implies B \subseteq C \implies A \subset C$   
**by** (*auto simp add: psubset-eq*)

**lemma** *subset-psubset-trans*:  $A \subseteq B \implies B \subset C \implies A \subset C$   
**by** (*auto simp add: psubset-eq*)

**lemma** *psubset-imp-ex-mem*:  $A \subset B \implies \exists b. b \in (B - A)$   
**by** (*unfold less-le*) *blast*

**lemma** *atomize-ball*:

( $!!x. x \in A \implies P x$ ) == *Trueprop* ( $\forall x \in A. P x$ )  
**by** (*simp only: Ball-def atomize-all atomize-imp*)

**lemmas** [*symmetric, rulify*] = *atomize-ball*  
**and** [*symmetric, defn*] = *atomize-ball*

## 5.5 Further set-theory lemmas

### 5.5.1 Derived rules involving subsets.

*insert.*

**lemma** *subset-insertI*:  $B \subseteq \text{insert } a \ B$   
**by** (*rule subsetI*) (*erule insertI2*)

**lemma** *subset-insertI2*:  $A \subseteq B \implies A \subseteq \text{insert } b \ B$   
**by** *blast*

**lemma** *subset-insert*:  $x \notin A \implies (A \subseteq \text{insert } x \ B) = (A \subseteq B)$   
**by** *blast*

Big Union – least upper bound of a set.

**lemma** *Union-upper*:  $B \in A \implies B \subseteq \text{Union } A$   
**by** (*iprover intro: subsetI UnionI*)

**lemma** *Union-least*: ( $!!X. X \in A \implies X \subseteq C$ )  $\implies \text{Union } A \subseteq C$   
**by** (*iprover intro: subsetI elim: UnionE dest: subsetD*)

General union.

**lemma** *UN-upper*:  $a \in A \implies B \ a \subseteq (\bigcup_{x \in A} B \ x)$   
**by** *blast*

**lemma** *UN-least*: ( $!!x. x \in A \implies B \ x \subseteq C$ )  $\implies (\bigcup_{x \in A} B \ x) \subseteq C$   
**by** (*iprover intro: subsetI elim: UN-E dest: subsetD*)

Big Intersection – greatest lower bound of a set.

**lemma** *Inter-lower*:  $B \in A \implies \text{Inter } A \subseteq B$   
**by** *blast*

**lemma** *Inter-subset*:

( $!!X. X \in A \implies X \subseteq B; A \sim \{\}$ )  $\implies \bigcap A \subseteq B$   
**by** *blast*

**lemma** *Inter-greatest*: ( $!!X. X \in A \implies C \subseteq X$ )  $\implies C \subseteq \text{Inter } A$   
**by** (*iprover intro: InterI subsetI dest: subsetD*)



**lemma** *INT-lower*:  $a \in A \implies (\bigcap_{x \in A} B x) \subseteq B a$   
**by** *blast*

**lemma** *INT-greatest*:  $(\forall x. x \in A \implies C \subseteq B x) \implies C \subseteq (\bigcap_{x \in A} B x)$   
**by** (*iprover intro: INT-I subsetI dest: subsetD*)

Finite Union – the least upper bound of two sets.

**lemma** *Un-upper1*:  $A \subseteq A \cup B$   
**by** *blast*

**lemma** *Un-upper2*:  $B \subseteq A \cup B$   
**by** *blast*

**lemma** *Un-least*:  $A \subseteq C \implies B \subseteq C \implies A \cup B \subseteq C$   
**by** *blast*

Finite Intersection – the greatest lower bound of two sets.

**lemma** *Int-lower1*:  $A \cap B \subseteq A$   
**by** *blast*

**lemma** *Int-lower2*:  $A \cap B \subseteq B$   
**by** *blast*

**lemma** *Int-greatest*:  $C \subseteq A \implies C \subseteq B \implies C \subseteq A \cap B$   
**by** *blast*

Set difference.

**lemma** *Diff-subset*:  $A - B \subseteq A$   
**by** *blast*

**lemma** *Diff-subset-conv*:  $(A - B \subseteq C) = (A \subseteq B \cup C)$   
**by** *blast*

### 5.5.2 Equalities involving union, intersection, inclusion, etc.

$\{\}$ .

**lemma** *Collect-const* [*simp*]:  $\{s. P\} = (\text{if } P \text{ then } \text{UNIV} \text{ else } \{\})$   
— supersedes *Collect-False-empty*  
**by** *auto*

**lemma** *subset-empty* [*simp*]:  $(A \subseteq \{\}) = (A = \{\})$   
**by** *blast*

**lemma** *not-psubset-empty* [*iff*]:  $\neg (A < \{\})$   
**by** (*unfold less-le*) *blast*

**lemma** *Collect-empty-eq* [*simp*]:  $(\text{Collect } P = \{\}) = (\forall x. \neg P x)$

by *blast*

**lemma** *empty-Collect-eq* [*simp*]:  $(\{\} = \text{Collect } P) = (\forall x. \neg P x)$   
by *blast*

**lemma** *Collect-neg-eq*:  $\{x. \neg P x\} = - \{x. P x\}$   
by *blast*

**lemma** *Collect-disj-eq*:  $\{x. P x \mid Q x\} = \{x. P x\} \cup \{x. Q x\}$   
by *blast*

**lemma** *Collect-imp-eq*:  $\{x. P x \longrightarrow Q x\} = -\{x. P x\} \cup \{x. Q x\}$   
by *blast*

**lemma** *Collect-conj-eq*:  $\{x. P x \ \& \ Q x\} = \{x. P x\} \cap \{x. Q x\}$   
by *blast*

**lemma** *Collect-all-eq*:  $\{x. \forall y. P x y\} = (\bigcap y. \{x. P x y\})$   
by *blast*

**lemma** *Collect-ball-eq*:  $\{x. \forall y \in A. P x y\} = (\bigcap y \in A. \{x. P x y\})$   
by *blast*

**lemma** *Collect-ex-eq* [*noatp*]:  $\{x. \exists y. P x y\} = (\bigcup y. \{x. P x y\})$   
by *blast*

**lemma** *Collect-bex-eq* [*noatp*]:  $\{x. \exists y \in A. P x y\} = (\bigcup y \in A. \{x. P x y\})$   
by *blast*

*insert.*

**lemma** *insert-is-Un*:  $\text{insert } a \ A = \{a\} \ \text{Un } A$   
— NOT SUITABLE FOR REWRITING since  $\{a\} == \text{insert } a \ \{\}$   
by *blast*

**lemma** *insert-not-empty* [*simp*]:  $\text{insert } a \ A \neq \{\}$   
by *blast*

**lemmas** *empty-not-insert* = *insert-not-empty* [*symmetric, standard*]  
**declare** *empty-not-insert* [*simp*]

**lemma** *insert-absorb*:  $a \in A ==> \text{insert } a \ A = A$   
— [*simp*] causes recursive calls when there are nested inserts  
— with *quadratic* running time  
by *blast*

**lemma** *insert-absorb2* [*simp*]:  $\text{insert } x \ (\text{insert } x \ A) = \text{insert } x \ A$   
by *blast*

**lemma** *insert-commute*:  $\text{insert } x \ (\text{insert } y \ A) = \text{insert } y \ (\text{insert } x \ A)$

**by** *blast*

**lemma** *insert-subset* [*simp*]:  $(\text{insert } x \ A \subseteq B) = (x \in B \ \& \ A \subseteq B)$   
**by** *blast*

**lemma** *mk-disjoint-insert*:  $a \in A \implies \exists B. A = \text{insert } a \ B \ \& \ a \notin B$   
 — use new  $B$  rather than  $A - \{a\}$  to avoid infinite unfolding  
**apply** (*rule-tac*  $x = A - \{a\}$  **in** *exI*, *blast*)  
**done**

**lemma** *insert-Collect*:  $\text{insert } a \ (\text{Collect } P) = \{u. u \neq a \longrightarrow P \ u\}$   
**by** *auto*

**lemma** *UN-insert-distrib*:  $u \in A \implies (\bigcup_{x \in A. \text{insert } a \ (B \ x)) = \text{insert } a \ (\bigcup_{x \in A. B \ x})$   
**by** *blast*

**lemma** *insert-inter-insert*[*simp*]:  $\text{insert } a \ A \cap \text{insert } a \ B = \text{insert } a \ (A \cap B)$   
**by** *blast*

**lemma** *insert-disjoint* [*simp*,*noatp*]:  
 $(\text{insert } a \ A \cap B = \{\}) = (a \notin B \wedge A \cap B = \{\})$   
 $(\{\} = \text{insert } a \ A \cap B) = (a \notin B \wedge \{\} = A \cap B)$   
**by** *auto*

**lemma** *disjoint-insert* [*simp*,*noatp*]:  
 $(B \cap \text{insert } a \ A = \{\}) = (a \notin B \wedge B \cap A = \{\})$   
 $(\{\} = A \cap \text{insert } b \ B) = (b \notin A \wedge \{\} = A \cap B)$   
**by** *auto*

*image.*

**lemma** *image-empty* [*simp*]:  $f' \{\} = \{\}$   
**by** *blast*

**lemma** *image-insert* [*simp*]:  $f' \ (\text{insert } a \ B) = \text{insert } (f \ a) \ (f' B)$   
**by** *blast*

**lemma** *image-constant*:  $x \in A \implies (\lambda x. c)' A = \{c\}$   
**by** *auto*

**lemma** *image-constant-conv*:  $(\%x. c)' A = (\text{if } A = \{\} \text{ then } \{\} \text{ else } \{c\})$   
**by** *auto*

**lemma** *image-image*:  $f' (g' A) = (\lambda x. f \ (g \ x))' A$   
**by** *blast*

**lemma** *insert-image* [*simp*]:  $x \in A \implies \text{insert } (f \ x) \ (f' A) = f' A$   
**by** *blast*

**lemma** *image-is-empty* [iff]:  $(f^{\circ}A = \{\}) = (A = \{\})$   
**by** *blast*

**lemma** *image-Collect* [noatp]:  $f^{\circ} \{x. P\ x\} = \{f\ x \mid x. P\ x\}$   
 — NOT suitable as a default simp rule: the RHS isn’t simpler than the LHS, with its implicit quantifier and conjunction. Also image enjoys better equational properties than does the RHS.  
**by** *blast*

**lemma** *if-image-distrib* [simp]:  
 $(\lambda x. \text{if } P\ x \text{ then } f\ x \text{ else } g\ x)^{\circ} S$   
 $= (f^{\circ} (S \cap \{x. P\ x\})) \cup (g^{\circ} (S \cap \{x. \neg P\ x\}))$   
**by** (*auto simp add: image-def*)

**lemma** *image-cong*:  $M = N \implies (!x. x \in N \implies f\ x = g\ x) \implies f^{\circ}M = g^{\circ}N$   
**by** (*simp add: image-def*)

*range.*

**lemma** *full-SetCompr-eq* [noatp]:  $\{u. \exists x. u = f\ x\} = \text{range } f$   
**by** *auto*

**lemma** *range-composition*:  $\text{range } (\lambda x. f\ (g\ x)) = f^{\circ} \text{range } g$   
**by** (*subst image-image, simp*)

*Int*

**lemma** *Int-absorb* [simp]:  $A \cap A = A$   
**by** *blast*

**lemma** *Int-left-absorb*:  $A \cap (A \cap B) = A \cap B$   
**by** *blast*

**lemma** *Int-commute*:  $A \cap B = B \cap A$   
**by** *blast*

**lemma** *Int-left-commute*:  $A \cap (B \cap C) = B \cap (A \cap C)$   
**by** *blast*

**lemma** *Int-assoc*:  $(A \cap B) \cap C = A \cap (B \cap C)$   
**by** *blast*

**lemmas** *Int-ac = Int-assoc Int-left-absorb Int-commute Int-left-commute*  
 — Intersection is an AC-operator

**lemma** *Int-absorb1*:  $B \subseteq A \implies A \cap B = B$   
**by** *blast*

**lemma** *Int-absorb2*:  $A \subseteq B \implies A \cap B = A$

by *blast*

**lemma** *Int-empty-left* [*simp*]:  $\{\} \cap B = \{\}$   
by *blast*

**lemma** *Int-empty-right* [*simp*]:  $A \cap \{\} = \{\}$   
by *blast*

**lemma** *disjoint-eq-subset-Compl*:  $(A \cap B = \{\}) = (A \subseteq -B)$   
by *blast*

**lemma** *disjoint-iff-not-equal*:  $(A \cap B = \{\}) = (\forall x \in A. \forall y \in B. x \neq y)$   
by *blast*

**lemma** *Int-UNIV-left* [*simp*]:  $UNIV \cap B = B$   
by *blast*

**lemma** *Int-UNIV-right* [*simp*]:  $A \cap UNIV = A$   
by *blast*

**lemma** *Int-eq-Inter*:  $A \cap B = \bigcap \{A, B\}$   
by *blast*

**lemma** *Int-Un-distrib*:  $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$   
by *blast*

**lemma** *Int-Un-distrib2*:  $(B \cup C) \cap A = (B \cap A) \cup (C \cap A)$   
by *blast*

**lemma** *Int-UNIV* [*simp, noatp*]:  $(A \cap B = UNIV) = (A = UNIV \ \& \ B = UNIV)$   
by *blast*

**lemma** *Int-subset-iff* [*simp*]:  $(C \subseteq A \cap B) = (C \subseteq A \ \& \ C \subseteq B)$   
by *blast*

**lemma** *Int-Collect*:  $(x \in A \cap \{x. P \ x\}) = (x \in A \ \& \ P \ x)$   
by *blast*

*Un.*

**lemma** *Un-absorb* [*simp*]:  $A \cup A = A$   
by *blast*

**lemma** *Un-left-absorb*:  $A \cup (A \cup B) = A \cup B$   
by *blast*

**lemma** *Un-commute*:  $A \cup B = B \cup A$   
by *blast*

**lemma** *Un-left-commute*:  $A \cup (B \cup C) = B \cup (A \cup C)$

by *blast*

**lemma** *Un-assoc*:  $(A \cup B) \cup C = A \cup (B \cup C)$   
by *blast*

**lemmas** *Un-ac = Un-assoc Un-left-absorb Un-commute Un-left-commute*  
— Union is an AC-operator

**lemma** *Un-absorb1*:  $A \subseteq B \implies A \cup B = B$   
by *blast*

**lemma** *Un-absorb2*:  $B \subseteq A \implies A \cup B = A$   
by *blast*

**lemma** *Un-empty-left [simp]*:  $\{\} \cup B = B$   
by *blast*

**lemma** *Un-empty-right [simp]*:  $A \cup \{\} = A$   
by *blast*

**lemma** *Un-UNIV-left [simp]*:  $UNIV \cup B = UNIV$   
by *blast*

**lemma** *Un-UNIV-right [simp]*:  $A \cup UNIV = UNIV$   
by *blast*

**lemma** *Un-eq-Union*:  $A \cup B = \bigcup \{A, B\}$   
by *blast*

**lemma** *Un-insert-left [simp]*:  $(\text{insert } a \ B) \cup C = \text{insert } a \ (B \cup C)$   
by *blast*

**lemma** *Un-insert-right [simp]*:  $A \cup (\text{insert } a \ B) = \text{insert } a \ (A \cup B)$   
by *blast*

**lemma** *Int-insert-left*:  
 $(\text{insert } a \ B) \cap C = (\text{if } a \in C \text{ then } \text{insert } a \ (B \cap C) \text{ else } B \cap C)$   
by *auto*

**lemma** *Int-insert-right*:  
 $A \cap (\text{insert } a \ B) = (\text{if } a \in A \text{ then } \text{insert } a \ (A \cap B) \text{ else } A \cap B)$   
by *auto*

**lemma** *Un-Int-distrib*:  $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$   
by *blast*

**lemma** *Un-Int-distrib2*:  $(B \cap C) \cup A = (B \cup A) \cap (C \cup A)$   
by *blast*

**lemma** *Un-Int-crazy*:

$$(A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A)$$

**by** *blast*

**lemma** *subset-Un-eq*:  $(A \subseteq B) = (A \cup B = B)$

**by** *blast*

**lemma** *Un-empty [iff]*:  $(A \cup B = \{\}) = (A = \{\} \ \& \ B = \{\})$

**by** *blast*

**lemma** *Un-subset-iff [simp]*:  $(A \cup B \subseteq C) = (A \subseteq C \ \& \ B \subseteq C)$

**by** *blast*

**lemma** *Un-Diff-Int*:  $(A - B) \cup (A \cap B) = A$

**by** *blast*

**lemma** *Diff-Int2*:  $A \cap C - B \cap C = A \cap C - B$

**by** *blast*

Set complement

**lemma** *Compl-disjoint [simp]*:  $A \cap -A = \{\}$

**by** *blast*

**lemma** *Compl-disjoint2 [simp]*:  $-A \cap A = \{\}$

**by** *blast*

**lemma** *Compl-partition*:  $A \cup -A = UNIV$

**by** *blast*

**lemma** *Compl-partition2*:  $-A \cup A = UNIV$

**by** *blast*

**lemma** *double-complement [simp]*:  $-(-A) = (A::'a \text{ set})$

**by** *blast*

**lemma** *Compl-Un [simp]*:  $-(A \cup B) = (-A) \cap (-B)$

**by** *blast*

**lemma** *Compl-Int [simp]*:  $-(A \cap B) = (-A) \cup (-B)$

**by** *blast*

**lemma** *Compl-UN [simp]*:  $-(\bigcup x \in A. B \ x) = (\bigcap x \in A. -B \ x)$

**by** *blast*

**lemma** *Compl-INT [simp]*:  $-(\bigcap x \in A. B \ x) = (\bigcup x \in A. -B \ x)$

**by** *blast*

**lemma** *subset-Compl-self-eq*:  $(A \subseteq -A) = (A = \{\})$

**by** *blast*

**lemma** *Un-Int-assoc-eq*:  $((A \cap B) \cup C = A \cap (B \cup C)) = (C \subseteq A)$   
 — Halmos, Naive Set Theory, page 16.  
**by** *blast*

**lemma** *Compl-UNIV-eq* [simp]:  $-UNIV = \{\}$   
**by** *blast*

**lemma** *Compl-empty-eq* [simp]:  $-\{\} = UNIV$   
**by** *blast*

**lemma** *Compl-subset-Compl-iff* [iff]:  $(-A \subseteq -B) = (B \subseteq A)$   
**by** *blast*

**lemma** *Compl-eq-Compl-iff* [iff]:  $(-A = -B) = (A = (B::'a \text{ set}))$   
**by** *blast*

*Union.*

**lemma** *Union-empty* [simp]:  $Union(\{\}) = \{\}$   
**by** *blast*

**lemma** *Union-UNIV* [simp]:  $Union UNIV = UNIV$   
**by** *blast*

**lemma** *Union-insert* [simp]:  $Union (\text{insert } a \ B) = a \cup \bigcup B$   
**by** *blast*

**lemma** *Union-Un-distrib* [simp]:  $\bigcup (A \ Un \ B) = \bigcup A \cup \bigcup B$   
**by** *blast*

**lemma** *Union-Int-subset*:  $\bigcup (A \cap B) \subseteq \bigcup A \cap \bigcup B$   
**by** *blast*

**lemma** *Union-empty-conv* [simp,noatp]:  $(\bigcup A = \{\}) = (\forall x \in A. x = \{\})$   
**by** *blast*

**lemma** *empty-Union-conv* [simp,noatp]:  $(\{\} = \bigcup A) = (\forall x \in A. x = \{\})$   
**by** *blast*

**lemma** *Union-disjoint*:  $(\bigcup C \cap A = \{\}) = (\forall B \in C. B \cap A = \{\})$   
**by** *blast*

*Inter.*

**lemma** *Inter-empty* [simp]:  $\bigcap \{\} = UNIV$   
**by** *blast*

**lemma** *Inter-UNIV* [simp]:  $\bigcap UNIV = \{\}$   
**by** *blast*



**lemma** *Inter-insert* [simp]:  $\bigcap (\text{insert } a \ B) = a \cap \bigcap B$   
**by** *blast*

**lemma** *Inter-Un-subset*:  $\bigcap A \cup \bigcap B \subseteq \bigcap (A \cap B)$   
**by** *blast*

**lemma** *Inter-Un-distrib*:  $\bigcap (A \cup B) = \bigcap A \cap \bigcap B$   
**by** *blast*

**lemma** *Inter-UNIV-conv* [simp, noatp]:  
 $(\bigcap A = \text{UNIV}) = (\forall x \in A. x = \text{UNIV})$   
 $(\text{UNIV} = \bigcap A) = (\forall x \in A. x = \text{UNIV})$   
**by** *blast+*

*UN* and *INT*.

Basic identities:

**lemma** *UN-empty* [simp, noatp]:  $(\bigcup x \in \{\}. B \ x) = \{\}$   
**by** *blast*

**lemma** *UN-empty2* [simp]:  $(\bigcup x \in A. \{\}) = \{\}$   
**by** *blast*

**lemma** *UN-singleton* [simp]:  $(\bigcup x \in A. \{x\}) = A$   
**by** *blast*

**lemma** *UN-absorb*:  $k \in I ==> A \ k \cup (\bigcup i \in I. A \ i) = (\bigcup i \in I. A \ i)$   
**by** *auto*

**lemma** *INT-empty* [simp]:  $(\bigcap x \in \{\}. B \ x) = \text{UNIV}$   
**by** *blast*

**lemma** *INT-absorb*:  $k \in I ==> A \ k \cap (\bigcap i \in I. A \ i) = (\bigcap i \in I. A \ i)$   
**by** *blast*

**lemma** *UN-insert* [simp]:  $(\bigcup x \in \text{insert } a \ A. B \ x) = B \ a \cup \text{UNION } A \ B$   
**by** *blast*

**lemma** *UN-Un* [simp]:  $(\bigcup i \in A \cup B. M \ i) = (\bigcup i \in A. M \ i) \cup (\bigcup i \in B. M \ i)$   
**by** *blast*

**lemma** *UN-UN-flatten*:  $(\bigcup x \in (\bigcup y \in A. B \ y). C \ x) = (\bigcup y \in A. \bigcup x \in B \ y. C \ x)$   
**by** *blast*

**lemma** *UN-subset-iff*:  $((\bigcup i \in I. A \ i) \subseteq B) = (\forall i \in I. A \ i \subseteq B)$   
**by** *blast*

**lemma** *INT-subset-iff*:  $(B \subseteq (\bigcap i \in I. A \ i)) = (\forall i \in I. B \subseteq A \ i)$   
**by** *blast*

**lemma** *INT-insert* [simp]:  $(\bigcap x \in \text{insert } a \ A. B \ x) = B \ a \cap \text{INTER } A \ B$   
**by** *blast*

**lemma** *INT-Un*:  $(\bigcap i \in A \cup B. M \ i) = (\bigcap i \in A. M \ i) \cap (\bigcap i \in B. M \ i)$   
**by** *blast*

**lemma** *INT-insert-distrib*:  
 $u \in A \implies (\bigcap x \in A. \text{insert } a \ (B \ x)) = \text{insert } a \ (\bigcap x \in A. B \ x)$   
**by** *blast*

**lemma** *Union-image-eq* [simp]:  $\bigcup (B' A) = (\bigcup x \in A. B \ x)$   
**by** *blast*

**lemma** *image-Union*:  $f' \bigcup S = (\bigcup x \in S. f' \ x)$   
**by** *blast*

**lemma** *Inter-image-eq* [simp]:  $\bigcap (B' A) = (\bigcap x \in A. B \ x)$   
**by** *blast*

**lemma** *UN-constant* [simp]:  $(\bigcup y \in A. c) = (\text{if } A = \{\} \text{ then } \{\} \text{ else } c)$   
**by** *auto*

**lemma** *INT-constant* [simp]:  $(\bigcap y \in A. c) = (\text{if } A = \{\} \text{ then } \text{UNIV} \text{ else } c)$   
**by** *auto*

**lemma** *UN-eq*:  $(\bigcup x \in A. B \ x) = \bigcup (\{Y. \exists x \in A. Y = B \ x\})$   
**by** *blast*

**lemma** *INT-eq*:  $(\bigcap x \in A. B \ x) = \bigcap (\{Y. \exists x \in A. Y = B \ x\})$   
— Look: it has an *existential* quantifier  
**by** *blast*

**lemma** *UNION-empty-conv*[simp]:  
 $(\{\} = (\text{UN } x:A. B \ x)) = (\forall x \in A. B \ x = \{\})$   
 $((\text{UN } x:A. B \ x) = \{\}) = (\forall x \in A. B \ x = \{\})$   
**by** *blast+*

**lemma** *INTER-UNIV-conv*[simp]:  
 $(\text{UNIV} = (\text{INT } x:A. B \ x)) = (\forall x \in A. B \ x = \text{UNIV})$   
 $((\text{INT } x:A. B \ x) = \text{UNIV}) = (\forall x \in A. B \ x = \text{UNIV})$   
**by** *blast+*

Distributive laws:

**lemma** *Int-Union*:  $A \cap \bigcup B = (\bigcup C \in B. A \cap C)$   
**by** *blast*

**lemma** *Int-Union2*:  $\bigcup B \cap A = (\bigcup C \in B. C \cap A)$   
**by** *blast*

**lemma** *Un-Union-image*:  $(\bigcup_{x \in C}. A \ x \cup B \ x) = \bigcup (A' C) \cup \bigcup (B' C)$   
 — Devlin, Fundamentals of Contemporary Set Theory, page 12, exercise 5:  
 — Union of a family of unions  
**by** *blast*

**lemma** *UN-Un-distrib*:  $(\bigcup_{i \in I}. A \ i \cup B \ i) = (\bigcup_{i \in I}. A \ i) \cup (\bigcup_{i \in I}. B \ i)$   
 — Equivalent version  
**by** *blast*

**lemma** *Un-Inter*:  $A \cup \bigcap B = (\bigcap_{C \in B}. A \cup C)$   
**by** *blast*

**lemma** *Int-Inter-image*:  $(\bigcap_{x \in C}. A \ x \cap B \ x) = \bigcap (A' C) \cap \bigcap (B' C)$   
**by** *blast*

**lemma** *INT-Int-distrib*:  $(\bigcap_{i \in I}. A \ i \cap B \ i) = (\bigcap_{i \in I}. A \ i) \cap (\bigcap_{i \in I}. B \ i)$   
 — Equivalent version  
**by** *blast*

**lemma** *Int-UN-distrib*:  $B \cap (\bigcup_{i \in I}. A \ i) = (\bigcup_{i \in I}. B \cap A \ i)$   
 — Halmos, Naive Set Theory, page 35.  
**by** *blast*

**lemma** *Un-INT-distrib*:  $B \cup (\bigcap_{i \in I}. A \ i) = (\bigcap_{i \in I}. B \cup A \ i)$   
**by** *blast*

**lemma** *Int-UN-distrib2*:  $(\bigcup_{i \in I}. A \ i) \cap (\bigcup_{j \in J}. B \ j) = (\bigcup_{i \in I}. \bigcup_{j \in J}. A \ i \cap B \ j)$   
**by** *blast*

**lemma** *Un-INT-distrib2*:  $(\bigcap_{i \in I}. A \ i) \cup (\bigcap_{j \in J}. B \ j) = (\bigcap_{i \in I}. \bigcap_{j \in J}. A \ i \cup B \ j)$   
**by** *blast*

Bounded quantifiers.

The following are not added to the default simpset because (a) they duplicate the body and (b) there are no similar rules for *Int*.

**lemma** *ball-Un*:  $(\forall x \in A \cup B. P \ x) = ((\forall x \in A. P \ x) \ \& \ (\forall x \in B. P \ x))$   
**by** *blast*

**lemma** *bex-Un*:  $(\exists x \in A \cup B. P \ x) = ((\exists x \in A. P \ x) \ | \ (\exists x \in B. P \ x))$   
**by** *blast*

**lemma** *ball-UN*:  $(\forall z \in \text{UNION } A \ B. P \ z) = (\forall x \in A. \forall z \in B \ x. P \ z)$   
**by** *blast*

**lemma** *bex-UN*:  $(\exists z \in \text{UNION } A \ B. P \ z) = (\exists x \in A. \exists z \in B \ x. P \ z)$

**by** *blast*

Set difference.

**lemma** *Diff-eq*:  $A - B = A \cap (-B)$   
**by** *blast*

**lemma** *Diff-eq-empty-iff* [*simp, noatp*]:  $(A - B = \{\}) = (A \subseteq B)$   
**by** *blast*

**lemma** *Diff-cancel* [*simp*]:  $A - A = \{\}$   
**by** *blast*

**lemma** *Diff-idemp* [*simp*]:  $(A - B) - B = A - (B::'a \text{ set})$   
**by** *blast*

**lemma** *Diff-triv*:  $A \cap B = \{\} ==> A - B = A$   
**by** (*blast elim: equalityE*)

**lemma** *empty-Diff* [*simp*]:  $\{\} - A = \{\}$   
**by** *blast*

**lemma** *Diff-empty* [*simp*]:  $A - \{\} = A$   
**by** *blast*

**lemma** *Diff-UNIV* [*simp*]:  $A - UNIV = \{\}$   
**by** *blast*

**lemma** *Diff-insert0* [*simp, noatp*]:  $x \notin A ==> A - \text{insert } x B = A - B$   
**by** *blast*

**lemma** *Diff-insert*:  $A - \text{insert } a B = A - B - \{a\}$   
 — NOT SUITABLE FOR REWRITING since  $\{a\} == \text{insert } a 0$   
**by** *blast*

**lemma** *Diff-insert2*:  $A - \text{insert } a B = A - \{a\} - B$   
 — NOT SUITABLE FOR REWRITING since  $\{a\} == \text{insert } a 0$   
**by** *blast*

**lemma** *insert-Diff-if*:  $\text{insert } x A - B = (\text{if } x \in B \text{ then } A - B \text{ else } \text{insert } x (A - B))$   
**by** *auto*

**lemma** *insert-Diff1* [*simp*]:  $x \in B ==> \text{insert } x A - B = A - B$   
**by** *blast*

**lemma** *insert-Diff-single* [*simp*]:  $\text{insert } a (A - \{a\}) = \text{insert } a A$   
**by** *blast*

**lemma** *insert-Diff*:  $a \in A ==> \text{insert } a (A - \{a\}) = A$

by *blast*

**lemma** *Diff-insert-absorb*:  $x \notin A \implies (\text{insert } x \ A) - \{x\} = A$   
by *auto*

**lemma** *Diff-disjoint [simp]*:  $A \cap (B - A) = \{\}$   
by *blast*

**lemma** *Diff-partition*:  $A \subseteq B \implies A \cup (B - A) = B$   
by *blast*

**lemma** *double-diff*:  $A \subseteq B \implies B \subseteq C \implies B - (C - A) = A$   
by *blast*

**lemma** *Un-Diff-cancel [simp]*:  $A \cup (B - A) = A \cup B$   
by *blast*

**lemma** *Un-Diff-cancel2 [simp]*:  $(B - A) \cup A = B \cup A$   
by *blast*

**lemma** *Diff-Un*:  $A - (B \cup C) = (A - B) \cap (A - C)$   
by *blast*

**lemma** *Diff-Int*:  $A - (B \cap C) = (A - B) \cup (A - C)$   
by *blast*

**lemma** *Un-Diff*:  $(A \cup B) - C = (A - C) \cup (B - C)$   
by *blast*

**lemma** *Int-Diff*:  $(A \cap B) - C = A \cap (B - C)$   
by *blast*

**lemma** *Diff-Int-distrib*:  $C \cap (A - B) = (C \cap A) - (C \cap B)$   
by *blast*

**lemma** *Diff-Int-distrib2*:  $(A - B) \cap C = (A \cap C) - (B \cap C)$   
by *blast*

**lemma** *Diff-Compl [simp]*:  $A - (- B) = A \cap B$   
by *auto*

**lemma** *Compl-Diff-eq [simp]*:  $- (A - B) = -A \cup B$   
by *blast*

Quantification over type *bool*.

**lemma** *bool-induct*:  $P \ \text{True} \implies P \ \text{False} \implies P \ x$   
by (*cases x*) *auto*

**lemma** *all-bool-eq*:  $(\forall b. P \ b) \longleftrightarrow P \ \text{True} \wedge P \ \text{False}$

**by** (*auto intro: bool-induct*)

**lemma** *bool-contrapos*:  $P\ x \implies \neg P\ False \implies P\ True$   
**by** (*cases x*) *auto*

**lemma** *ex-bool-eq*:  $(\exists b. P\ b) \longleftrightarrow P\ True \vee P\ False$   
**by** (*auto intro: bool-contrapos*)

**lemma** *Un-eq-UN*:  $A \cup B = (\bigcup b. \text{if } b \text{ then } A \text{ else } B)$   
**by** (*auto simp add: split-if-mem2*)

**lemma** *UN-bool-eq*:  $(\bigcup b::\text{bool}. A\ b) = (A\ True \cup A\ False)$   
**by** (*auto intro: bool-contrapos*)

**lemma** *INT-bool-eq*:  $(\bigcap b::\text{bool}. A\ b) = (A\ True \cap A\ False)$   
**by** (*auto intro: bool-induct*)

*Pow*

**lemma** *Pow-empty* [*simp*]:  $Pow\ \{\} = \{\{\}\}$   
**by** (*auto simp add: Pow-def*)

**lemma** *Pow-insert*:  $Pow\ (\text{insert } a\ A) = Pow\ A \cup (\text{insert } a\ 'Pow\ A)$   
**by** (*blast intro: image-eqI [where ?x = u - \{a\}, standard]*)

**lemma** *Pow-Compl*:  $Pow\ (\neg A) = \{\neg B \mid B. A \in Pow\ B\}$   
**by** (*blast intro: exI [where ?x = \neg u, standard]*)

**lemma** *Pow-UNIV* [*simp*]:  $Pow\ UNIV = UNIV$   
**by** *blast*

**lemma** *Un-Pow-subset*:  $Pow\ A \cup Pow\ B \subseteq Pow\ (A \cup B)$   
**by** *blast*

**lemma** *UN-Pow-subset*:  $(\bigcup x \in A. Pow\ (B\ x)) \subseteq Pow\ (\bigcup x \in A. B\ x)$   
**by** *blast*

**lemma** *subset-Pow-Union*:  $A \subseteq Pow\ (\bigcup A)$   
**by** *blast*

**lemma** *Union-Pow-eq* [*simp*]:  $\bigcup (Pow\ A) = A$   
**by** *blast*

**lemma** *Pow-Int-eq* [*simp*]:  $Pow\ (A \cap B) = Pow\ A \cap Pow\ B$   
**by** *blast*

**lemma** *Pow-INT-eq*:  $Pow\ (\bigcap x \in A. B\ x) = (\bigcap x \in A. Pow\ (B\ x))$   
**by** *blast*

Miscellany.

**lemma** *set-eq-subset*:  $(A = B) = (A \subseteq B \ \& \ B \subseteq A)$   
**by** *blast*

**lemma** *subset-iff*:  $(A \subseteq B) = (\forall t. t \in A \longrightarrow t \in B)$   
**by** *blast*

**lemma** *subset-iff-psubset-eq*:  $(A \subseteq B) = ((A \subset B) \mid (A = B))$   
**by** (*unfold less-le*) *blast*

**lemma** *all-not-in-conv* [*simp*]:  $(\forall x. x \notin A) = (A = \{\})$   
**by** *blast*

**lemma** *ex-in-conv*:  $(\exists x. x \in A) = (A \neq \{\})$   
**by** *blast*

**lemma** *distinct-lemma*:  $f \ x \neq f \ y \implies x \neq y$   
**by** *iprover*

Miniscoping: pushing in quantifiers and big Unions and Intersections.

**lemma** *UN-simps* [*simp*]:

!! $a \ B \ C. (UN \ x:C. insert \ a \ (B \ x)) = (if \ C=\{\} \ then \ \{\} \ else \ insert \ a \ (UN \ x:C. B \ x))$   
!! $A \ B \ C. (UN \ x:C. A \ x \ Un \ B) = ((if \ C=\{\} \ then \ \{\} \ else \ (UN \ x:C. A \ x) \ Un \ B))$   
!! $A \ B \ C. (UN \ x:C. A \ Un \ B \ x) = ((if \ C=\{\} \ then \ \{\} \ else \ A \ Un \ (UN \ x:C. B \ x)))$   
!! $A \ B \ C. (UN \ x:C. A \ x \ Int \ B) = ((UN \ x:C. A \ x) \ Int \ B)$   
!! $A \ B \ C. (UN \ x:C. A \ Int \ B \ x) = (A \ Int \ (UN \ x:C. B \ x))$   
!! $A \ B \ C. (UN \ x:C. A \ x - B) = ((UN \ x:C. A \ x) - B)$   
!! $A \ B \ C. (UN \ x:C. A - B \ x) = (A - (INT \ x:C. B \ x))$   
!! $A \ B. (UN \ x: Union \ A. B \ x) = (UN \ y:A. UN \ x:y. B \ x)$   
!! $A \ B \ C. (UN \ z: UNION \ A \ B. C \ z) = (UN \ x:A. UN \ z: B(x). C \ z)$   
!! $A \ B \ f. (UN \ x:f \ A. B \ x) = (UN \ a:A. B \ (f \ a))$   
**by** *auto*

**lemma** *INT-simps* [*simp*]:

!! $A \ B \ C. (INT \ x:C. A \ x \ Int \ B) = (if \ C=\{\} \ then \ UNIV \ else \ (INT \ x:C. A \ x) \ Int \ B)$   
!! $A \ B \ C. (INT \ x:C. A \ Int \ B \ x) = (if \ C=\{\} \ then \ UNIV \ else \ A \ Int \ (INT \ x:C. B \ x))$   
!! $A \ B \ C. (INT \ x:C. A \ x - B) = (if \ C=\{\} \ then \ UNIV \ else \ (INT \ x:C. A \ x) - B)$   
!! $A \ B \ C. (INT \ x:C. A - B \ x) = (if \ C=\{\} \ then \ UNIV \ else \ A - (UN \ x:C. B \ x))$   
!! $a \ B \ C. (INT \ x:C. insert \ a \ (B \ x)) = insert \ a \ (INT \ x:C. B \ x)$   
!! $A \ B \ C. (INT \ x:C. A \ x \ Un \ B) = ((INT \ x:C. A \ x) \ Un \ B)$   
!! $A \ B \ C. (INT \ x:C. A \ Un \ B \ x) = (A \ Un \ (INT \ x:C. B \ x))$   
!! $A \ B. (INT \ x: Union \ A. B \ x) = (INT \ y:A. INT \ x:y. B \ x)$   
!! $A \ B \ C. (INT \ z: UNION \ A \ B. C \ z) = (INT \ x:A. INT \ z: B(x). C \ z)$

$!!A B f. (INT x:f'A. B x) = (INT a:A. B (f a))$   
**by** *auto*

**lemma** *ball-simps* [*simp, noatp*]:

$!!A P Q. (ALL x:A. P x \mid Q) = ((ALL x:A. P x) \mid Q)$   
 $!!A P Q. (ALL x:A. P \mid Q x) = (P \mid (ALL x:A. Q x))$   
 $!!A P Q. (ALL x:A. P \dashv\vdash Q x) = (P \dashv\vdash (ALL x:A. Q x))$   
 $!!A P Q. (ALL x:A. P x \dashv\vdash Q) = ((EX x:A. P x) \dashv\vdash Q)$   
 $!!P. (ALL x:\{\}. P x) = True$   
 $!!P. (ALL x:UNIV. P x) = (ALL x. P x)$   
 $!!a B P. (ALL x:insert a B. P x) = (P a \& (ALL x:B. P x))$   
 $!!A P. (ALL x:Union A. P x) = (ALL y:A. ALL x:y. P x)$   
 $!!A B P. (ALL x: UNION A B. P x) = (ALL a:A. ALL x: B a. P x)$   
 $!!P Q. (ALL x:Collect Q. P x) = (ALL x. Q x \dashv\vdash P x)$   
 $!!A P f. (ALL x:f'A. P x) = (ALL x:A. P (f x))$   
 $!!A P. (\sim(ALL x:A. P x)) = (EX x:A. \sim P x)$   
**by** *auto*

**lemma** *bex-simps* [*simp, noatp*]:

$!!A P Q. (EX x:A. P x \& Q) = ((EX x:A. P x) \& Q)$   
 $!!A P Q. (EX x:A. P \& Q x) = (P \& (EX x:A. Q x))$   
 $!!P. (EX x:\{\}. P x) = False$   
 $!!P. (EX x:UNIV. P x) = (EX x. P x)$   
 $!!a B P. (EX x:insert a B. P x) = (P(a) \mid (EX x:B. P x))$   
 $!!A P. (EX x:Union A. P x) = (EX y:A. EX x:y. P x)$   
 $!!A B P. (EX x: UNION A B. P x) = (EX a:A. EX x:B a. P x)$   
 $!!P Q. (EX x:Collect Q. P x) = (EX x. Q x \& P x)$   
 $!!A P f. (EX x:f'A. P x) = (EX x:A. P (f x))$   
 $!!A P. (\sim(EX x:A. P x)) = (ALL x:A. \sim P x)$   
**by** *auto*

**lemma** *ball-conj-distrib*:

$(ALL x:A. P x \& Q x) = ((ALL x:A. P x) \& (ALL x:A. Q x))$   
**by** *blast*

**lemma** *bex-disj-distrib*:

$(EX x:A. P x \mid Q x) = ((EX x:A. P x) \mid (EX x:A. Q x))$   
**by** *blast*

Maxiscoping: pulling out big Unions and Intersections.

**lemma** *UN-extend-simps*:

$!!a B C. insert a (UN x:C. B x) = (if C=\{\} then \{a\} else (UN x:C. insert a (B x)))$   
 $!!A B C. (UN x:C. A x) Un B = (if C=\{\} then B else (UN x:C. A x Un B))$   
 $!!A B C. A Un (UN x:C. B x) = (if C=\{\} then A else (UN x:C. A Un B x))$   
 $!!A B C. ((UN x:C. A x) Int B) = (UN x:C. A x Int B)$   
 $!!A B C. (A Int (UN x:C. B x)) = (UN x:C. A Int B x)$   
 $!!A B C. ((UN x:C. A x) - B) = (UN x:C. A x - B)$   
 $!!A B C. (A - (INT x:C. B x)) = (UN x:C. A - B x)$



$!!A \ B. (UN \ y:A. UN \ x:y. B \ x) = (UN \ x: Union \ A. B \ x)$   
 $!!A \ B \ C. (UN \ x:A. UN \ z: B(x). C \ z) = (UN \ z: UNION \ A \ B. C \ z)$   
 $!!A \ B \ f. (UN \ a:A. B \ (f \ a)) = (UN \ x:f'A. B \ x)$   
**by** *auto*

**lemma** *INT-extend-simps*:

$!!A \ B \ C. (INT \ x:C. A \ x) \ Int \ B = (if \ C=\{\} \ then \ B \ else \ (INT \ x:C. A \ x \ Int \ B))$   
 $!!A \ B \ C. A \ Int \ (INT \ x:C. B \ x) = (if \ C=\{\} \ then \ A \ else \ (INT \ x:C. A \ Int \ B \ x))$   
 $!!A \ B \ C. (INT \ x:C. A \ x) - B = (if \ C=\{\} \ then \ UNIV - B \ else \ (INT \ x:C. A \ x - B))$   
 $!!A \ B \ C. A - (UN \ x:C. B \ x) = (if \ C=\{\} \ then \ A \ else \ (INT \ x:C. A - B \ x))$   
 $!!a \ B \ C. insert \ a \ (INT \ x:C. B \ x) = (INT \ x:C. insert \ a \ (B \ x))$   
 $!!A \ B \ C. ((INT \ x:C. A \ x) \ Un \ B) = (INT \ x:C. A \ x \ Un \ B)$   
 $!!A \ B \ C. A \ Un \ (INT \ x:C. B \ x) = (INT \ x:C. A \ Un \ B \ x)$   
 $!!A \ B. (INT \ y:A. INT \ x:y. B \ x) = (INT \ x: Union \ A. B \ x)$   
 $!!A \ B \ C. (INT \ x:A. INT \ z: B(x). C \ z) = (INT \ z: UNION \ A \ B. C \ z)$   
 $!!A \ B \ f. (INT \ a:A. B \ (f \ a)) = (INT \ x:f'A. B \ x)$   
**by** *auto*

### 5.5.3 Monotonicity of various operations

**lemma** *image-mono*:  $A \subseteq B \implies f'A \subseteq f'B$   
**by** *blast*

**lemma** *Pow-mono*:  $A \subseteq B \implies Pow \ A \subseteq Pow \ B$   
**by** *blast*

**lemma** *Union-mono*:  $A \subseteq B \implies \bigcup A \subseteq \bigcup B$   
**by** *blast*

**lemma** *Inter-anti-mono*:  $B \subseteq A \implies \bigcap A \subseteq \bigcap B$   
**by** *blast*

**lemma** *UN-mono*:  
 $A \subseteq B \implies (!!x. x \in A \implies f \ x \subseteq g \ x) \implies$   
 $(\bigcup_{x \in A}. f \ x) \subseteq (\bigcup_{x \in B}. g \ x)$   
**by** (*blast dest: subsetD*)

**lemma** *INT-anti-mono*:  
 $B \subseteq A \implies (!!x. x \in A \implies f \ x \subseteq g \ x) \implies$   
 $(\bigcap_{x \in A}. f \ x) \subseteq (\bigcap_{x \in A}. g \ x)$   
— The last inclusion is POSITIVE!  
**by** (*blast dest: subsetD*)

**lemma** *insert-mono*:  $C \subseteq D \implies insert \ a \ C \subseteq insert \ a \ D$   
**by** *blast*

**lemma** *Un-mono*:  $A \subseteq C \implies B \subseteq D \implies A \cup B \subseteq C \cup D$   
**by** *blast*

**lemma** *Int-mono*:  $A \subseteq C \implies B \subseteq D \implies A \cap B \subseteq C \cap D$   
**by** *blast*

**lemma** *Diff-mono*:  $A \subseteq C \implies D \subseteq B \implies A - B \subseteq C - D$   
**by** *blast*

**lemma** *Compl-anti-mono*:  $A \subseteq B \implies -B \subseteq -A$   
**by** *blast*

Monotonicity of implications.

**lemma** *in-mono*:  $A \subseteq B \implies x \in A \longrightarrow x \in B$   
**apply** (*rule impI*)  
**apply** (*erule subsetD, assumption*)  
**done**

**lemma** *conj-mono*:  $P1 \longrightarrow Q1 \implies P2 \longrightarrow Q2 \implies (P1 \ \& \ P2) \longrightarrow (Q1 \ \& \ Q2)$   
**by** *iprover*

**lemma** *disj-mono*:  $P1 \longrightarrow Q1 \implies P2 \longrightarrow Q2 \implies (P1 \ | \ P2) \longrightarrow (Q1 \ | \ Q2)$   
**by** *iprover*

**lemma** *imp-mono*:  $Q1 \longrightarrow P1 \implies P2 \longrightarrow Q2 \implies (P1 \longrightarrow P2) \longrightarrow (Q1 \longrightarrow Q2)$   
**by** *iprover*

**lemma** *imp-refl*:  $P \longrightarrow P \ ..$

**lemma** *ex-mono*:  $(!!x. P \ x \longrightarrow Q \ x) \implies (EX \ x. P \ x) \longrightarrow (EX \ x. Q \ x)$   
**by** *iprover*

**lemma** *all-mono*:  $(!!x. P \ x \longrightarrow Q \ x) \implies (ALL \ x. P \ x) \longrightarrow (ALL \ x. Q \ x)$   
**by** *iprover*

**lemma** *Collect-mono*:  $(!!x. P \ x \longrightarrow Q \ x) \implies Collect \ P \subseteq Collect \ Q$   
**by** *blast*

**lemma** *Int-Collect-mono*:  
 $A \subseteq B \implies (!!x. x \in A \implies P \ x \longrightarrow Q \ x) \implies A \cap Collect \ P \subseteq B \cap Collect \ Q$   
**by** *blast*

**lemmas** *basic-monos* =  
*subset-refl imp-refl disj-mono conj-mono*  
*ex-mono Collect-mono in-mono*

**lemma** *eq-to-mono*:  $a = b \implies c = d \implies b \longrightarrow d \implies a \longrightarrow c$

by *iprover*

**lemma** *eq-to-mono2*:  $a = b \implies c = d \implies \sim b \dashv\dashv \sim d \implies \sim a \dashv\dashv \sim c$   
by *iprover*

## 5.6 Inverse image of a function

**constdefs**

*vimage* ::  $('a \Rightarrow 'b) \Rightarrow 'b \text{ set} \Rightarrow 'a \text{ set}$     (**infixr**  $-' 90$ )  
[*code del*]:  $f -' B == \{x. f x : B\}$

### 5.6.1 Basic rules

**lemma** *vimage-eq* [*simp*]:  $(a : f -' B) = (f a : B)$   
by (*unfold vimage-def*) *blast*

**lemma** *vimage-singleton-eq*:  $(a : f -' \{b\}) = (f a = b)$   
by *simp*

**lemma** *vimageI* [*intro*]:  $f a = b \implies b : B \implies a : f -' B$   
by (*unfold vimage-def*) *blast*

**lemma** *vimageI2*:  $f a : A \implies a : f -' A$   
by (*unfold vimage-def*) *fast*

**lemma** *vimageE* [*elim!*]:  $a : f -' B \implies (!x. f a = x \implies x : B \implies P) \implies P$   
by (*unfold vimage-def*) *blast*

**lemma** *vimageD*:  $a : f -' A \implies f a : A$   
by (*unfold vimage-def*) *fast*

### 5.6.2 Equations

**lemma** *vimage-empty* [*simp*]:  $f -' \{\} = \{\}$   
by *blast*

**lemma** *vimage-Compl*:  $f -' (-A) = -(f -' A)$   
by *blast*

**lemma** *vimage-Un* [*simp*]:  $f -' (A \text{ Un } B) = (f -' A) \text{ Un } (f -' B)$   
by *blast*

**lemma** *vimage-Int* [*simp*]:  $f -' (A \text{ Int } B) = (f -' A) \text{ Int } (f -' B)$   
by *fast*

**lemma** *vimage-Union*:  $f -' (\text{Union } A) = (\text{UN } X:A. f -' X)$   
by *blast*

**lemma** *vimage-UN*:  $f -' (\text{UN } x:A. B x) = (\text{UN } x:A. f -' B x)$

by *blast*

**lemma** *vimage-INT*:  $f - ' (INT\ x:A. B\ x) = (INT\ x:A. f - ' B\ x)$   
by *blast*

**lemma** *vimage-Collect-eq* [*simp*]:  $f - ' Collect\ P = \{y. P\ (f\ y)\}$   
by *blast*

**lemma** *vimage-Collect*:  $(!!x. P\ (f\ x) = Q\ x) ==> f - ' (Collect\ P) = Collect\ Q$   
by *blast*

**lemma** *vimage-insert*:  $f - ' (insert\ a\ B) = (f - '\{a\})\ Un\ (f - ' B)$   
— NOT suitable for rewriting because of the recurrence of  $\{a\}$ .  
by *blast*

**lemma** *vimage-Diff*:  $f - ' (A - B) = (f - ' A) - (f - ' B)$   
by *blast*

**lemma** *vimage-UNIV* [*simp*]:  $f - ' UNIV = UNIV$   
by *blast*

**lemma** *vimage-eq-UN*:  $f - ' B = (UN\ y: B. f - '\{y\})$   
— NOT suitable for rewriting  
by *blast*

**lemma** *vimage-mono*:  $A \subseteq B ==> f - ' A \subseteq f - ' B$   
— monotonicity  
by *blast*

**lemma** *vimage-image-eq* [*noatp*]:  $f - ' (f - ' A) = \{y. EX\ x:A. f\ x = f\ y\}$   
by (*blast intro: sym*)

**lemma** *image-vimage-subset*:  $f - ' (f - ' A) <= A$   
by *blast*

**lemma** *image-vimage-eq* [*simp*]:  $f - ' (f - ' A) = A\ Int\ range\ f$   
by *blast*

**lemma** *image-Int-subset*:  $f - ' (A\ Int\ B) <= f - ' A\ Int\ f - ' B$   
by *blast*

**lemma** *image-diff-subset*:  $f - ' A - f - ' B <= f - ' (A - B)$   
by *blast*

**lemma** *image-UN*:  $f - ' (UNION\ A\ B) = (UN\ x:A. (f - ' (B\ x)))$   
by *blast*

## 5.7 Getting the Contents of a Singleton Set

**definition** *contents* :: 'a set  $\Rightarrow$  'a **where**  
`[code del]: contents X = (THE x. X = {x})`

**lemma** *contents-eq* `[simp]: contents {x} = x`  
**by** (*simp add: contents-def*)

## 5.8 Transitivity rules for calculational reasoning

**lemma** *set-rev-mp*:  $x:A \Rightarrow A \subseteq B \Rightarrow x:B$   
**by** (*rule subsetD*)

**lemma** *set-mp*:  $A \subseteq B \Rightarrow x:A \Rightarrow x:B$   
**by** (*rule subsetD*)

**lemmas** *basic-trans-rules* `[trans] =`  
*order-trans-rules set-rev-mp set-mp*

## 5.9 Least value operator

**lemma** *Least-mono*:  
 $\text{mono } (f::'a::\text{order} \Rightarrow 'b::\text{order}) \Rightarrow EX\ x:S. ALL\ y:S. x \leq y$   
 $\Rightarrow (LEAST\ y. y : f\ 'S) = f\ (LEAST\ x. x : S)$   
— Courtesy of Stephan Merz  
**apply** *clarify*  
**apply** (*erule-tac P = %x. x : S in LeastI2-order, fast*)  
**apply** (*rule LeastI2-order*)  
**apply** (*auto elim: monoD intro!: order-antisym*)  
**done**

## 5.10 Rudimentary code generation

**lemma** *empty-code* `[code]: {} x  $\longleftrightarrow$  False`  
**unfolding** *empty-def Collect-def* ..

**lemma** *UNIV-code* `[code]: UNIV x  $\longleftrightarrow$  True`  
**unfolding** *UNIV-def Collect-def* ..

**lemma** *insert-code* `[code]: insert y A x  $\longleftrightarrow$  y = x  $\vee$  A x`  
**unfolding** *insert-def Collect-def mem-def Un-def* **by** *auto*

**lemma** *inter-code* `[code]: (A  $\cap$  B) x  $\longleftrightarrow$  A x  $\wedge$  B x`  
**unfolding** *Int-def Collect-def mem-def* ..

**lemma** *union-code* `[code]: (A  $\cup$  B) x  $\longleftrightarrow$  A x  $\vee$  B x`  
**unfolding** *Un-def Collect-def mem-def* ..

**lemma** *image-code* `[code]: (f - 'A) x = A (f x)`  
**unfolding** *image-def Collect-def mem-def* ..

### 5.11 Complete lattices

**notation**

*less-eq* (**infix**  $\sqsubseteq$  50) and  
*less* (**infix**  $\sqsubset$  50) and  
*inf* (**infixl**  $\sqcap$  70) and  
*sup* (**infixl**  $\sqcup$  65)

**class** *complete-lattice* = *lattice* + *bot* + *top* +  
**fixes** *Inf* :: 'a set  $\Rightarrow$  'a ( $\sqcap$  - [900] 900)  
**and** *Sup* :: 'a set  $\Rightarrow$  'a ( $\sqcup$  - [900] 900)  
**assumes** *Inf-lower*:  $x \in A \Rightarrow \sqcap A \sqsubseteq x$   
**and** *Inf-greatest*:  $(\bigwedge x. x \in A \Rightarrow z \sqsubseteq x) \Rightarrow z \sqsubseteq \sqcap A$   
**assumes** *Sup-upper*:  $x \in A \Rightarrow x \sqsubseteq \sqcup A$   
**and** *Sup-least*:  $(\bigwedge x. x \in A \Rightarrow x \sqsubseteq z) \Rightarrow \sqcup A \sqsubseteq z$   
**begin**

**lemma** *Inf-Sup*:  $\sqcap A = \sqcup \{b. \forall a \in A. b \leq a\}$   
**by** (*auto intro: antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

**lemma** *Sup-Inf*:  $\sqcup A = \sqcap \{b. \forall a \in A. a \leq b\}$   
**by** (*auto intro: antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

**lemma** *Inf-Univ*:  $\sqcap UNIV = \sqcup \{\}$   
**unfolding** *Sup-Inf* **by** *auto*

**lemma** *Sup-Univ*:  $\sqcup UNIV = \sqcap \{\}$   
**unfolding** *Inf-Sup* **by** *auto*

**lemma** *Inf-insert*:  $\sqcap \text{insert } a \ A = a \sqcap \sqcap A$   
**by** (*auto intro: antisym Inf-greatest Inf-lower*)

**lemma** *Sup-insert*:  $\sqcup \text{insert } a \ A = a \sqcup \sqcup A$   
**by** (*auto intro: antisym Sup-least Sup-upper*)

**lemma** *Inf-singleton [simp]*:  
 $\sqcap \{a\} = a$   
**by** (*auto intro: antisym Inf-lower Inf-greatest*)

**lemma** *Sup-singleton [simp]*:  
 $\sqcup \{a\} = a$   
**by** (*auto intro: antisym Sup-upper Sup-least*)

**lemma** *Inf-insert-simp*:  
 $\sqcap \text{insert } a \ A = (\text{if } A = \{\} \text{ then } a \text{ else } a \sqcap \sqcap A)$   
**by** (*cases*  $A = \{\}$ ) (*simp-all, simp add: Inf-insert*)

**lemma** *Sup-insert-simp*:  
 $\sqcup \text{insert } a \ A = (\text{if } A = \{\} \text{ then } a \text{ else } a \sqcup \sqcup A)$   
**by** (*cases*  $A = \{\}$ ) (*simp-all, simp add: Sup-insert*)

**lemma** *Inf-binary*:

$\sqcap \{a, b\} = a \sqcap b$   
**by** (*simp add: Inf-insert-simp*)

**lemma** *Sup-binary*:

$\sqcup \{a, b\} = a \sqcup b$   
**by** (*simp add: Sup-insert-simp*)

**lemma** *bot-def*:

$\text{bot} = \sqcup \{\}$   
**by** (*auto intro: antisym Sup-least*)

**lemma** *top-def*:

$\text{top} = \sqcap \{\}$   
**by** (*auto intro: antisym Inf-greatest*)

**lemma** *sup-bot [simp]*:

$x \sqcup \text{bot} = x$   
**using** *bot-least [of x]* **by** (*simp add: le-iff-sup sup-commute*)

**lemma** *inf-top [simp]*:

$x \sqcap \text{top} = x$   
**using** *top-greatest [of x]* **by** (*simp add: le-iff-inf inf-commute*)

**definition** *SUPR* ::  $'b \text{ set} \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a$  **where**

$\text{SUPR } A \ f == \sqcup (f \text{ ` } A)$

**definition** *INFI* ::  $'b \text{ set} \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a$  **where**

$\text{INFI } A \ f == \sqcap (f \text{ ` } A)$

**end**

**syntax**

*-SUP1*    ::  $p\text{trns} \Rightarrow 'b \Rightarrow 'b$              $((\text{SUP} \text{ -./ -}) [0, 10] 10)$   
*-SUP*     ::  $p\text{trn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b$      $((\text{SUP} \text{ :-./ -}) [0, 10] 10)$   
*-INF1*    ::  $p\text{trns} \Rightarrow 'b \Rightarrow 'b$              $((\text{INF} \text{ -./ -}) [0, 10] 10)$   
*-INF*     ::  $p\text{trn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b$      $((\text{INF} \text{ :-./ -}) [0, 10] 10)$

**translations**

$\text{SUP } x \ y. B == \text{SUP } x. \text{SUP } y. B$   
 $\text{SUP } x. B == \text{CONST SUPR CONST UNIV } (\%x. B)$   
 $\text{SUP } x. B == \text{SUP } x:\text{CONST UNIV}. B$   
 $\text{SUP } x:A. B == \text{CONST SUPR } A (\%x. B)$   
 $\text{INF } x \ y. B == \text{INF } x. \text{INF } y. B$   
 $\text{INF } x. B == \text{CONST INFI CONST UNIV } (\%x. B)$   
 $\text{INF } x. B == \text{INF } x:\text{CONST UNIV}. B$   
 $\text{INF } x:A. B == \text{CONST INFI } A (\%x. B)$

```

print-translation ⟨⟨
  let
    fun btr' syn (A :: Abs abs :: ts) =
      let val (x,t) = atomic-abs-tr' abs
      in list-comb (Syntax.const syn $ x $ A $ t, ts) end
    val const-syntax-name = Sign.const-syntax-name @{theory} o fst o dest-Const
  in
    [(const-syntax-name @{term SUPR}, btr' -SUP), (const-syntax-name @{term INFI},
      btr' -INF)]
  end
  ⟩⟩

```

```

context complete-lattice
begin

```

```

lemma le-SUPI:  $i : A \implies M\ i \leq (SUP\ i:A.\ M\ i)$ 
by (auto simp add: SUPR-def intro: Sup-upper)

```

```

lemma SUP-leI:  $(\bigwedge i. i : A \implies M\ i \leq u) \implies (SUP\ i:A.\ M\ i) \leq u$ 
by (auto simp add: SUPR-def intro: Sup-least)

```

```

lemma INF-leI:  $i : A \implies (INF\ i:A.\ M\ i) \leq M\ i$ 
by (auto simp add: INFI-def intro: Inf-lower)

```

```

lemma le-INFI:  $(\bigwedge i. i : A \implies u \leq M\ i) \implies u \leq (INF\ i:A.\ M\ i)$ 
by (auto simp add: INFI-def intro: Inf-greatest)

```

```

lemma SUP-const[simp]:  $A \neq \{\} \implies (SUP\ i:A.\ M) = M$ 
by (auto intro: antisym SUP-leI le-SUPI)

```

```

lemma INF-const[simp]:  $A \neq \{\} \implies (INF\ i:A.\ M) = M$ 
by (auto intro: antisym INF-leI le-INFI)

```

```

end

```

## 5.12 Bool as complete lattice

```

instantiation bool :: complete-lattice
begin

```

```

definition
  Inf-bool-def:  $\bigcap A \longleftrightarrow (\forall x \in A. x)$ 

```

```

definition
  Sup-bool-def:  $\bigcup A \longleftrightarrow (\exists x \in A. x)$ 

```

```

instance
by intro-classes (auto simp add: Inf-bool-def Sup-bool-def le-bool-def)

```



**end**

**lemma** *Inf-empty-bool* [*simp*]:  
 $\sqcap \{\}$   
**unfolding** *Inf-bool-def* **by** *auto*

**lemma** *not-Sup-empty-bool* [*simp*]:  
 $\neg \sqcup \{\}$   
**unfolding** *Sup-bool-def* **by** *auto*

### 5.13 Fun as complete lattice

**instantiation** *fun* :: (*type*, *complete-lattice*) *complete-lattice*  
**begin**

**definition**  
*Inf-fun-def* [*code del*]:  $\sqcap A = (\lambda x. \sqcap \{y. \exists f \in A. y = f x\})$

**definition**  
*Sup-fun-def* [*code del*]:  $\sqcup A = (\lambda x. \sqcup \{y. \exists f \in A. y = f x\})$

**instance**  
**by** *intro-classes*  
(*auto simp add: Inf-fun-def Sup-fun-def le-fun-def*  
*intro: Inf-lower Sup-upper Inf-greatest Sup-least*)

**end**

**lemma** *Inf-empty-fun*:  
 $\sqcap \{\} = (\lambda -. \sqcap \{\})$   
**by** *rule (auto simp add: Inf-fun-def)*

**lemma** *Sup-empty-fun*:  
 $\sqcup \{\} = (\lambda -. \sqcup \{\})$   
**by** *rule (auto simp add: Sup-fun-def)*

### 5.14 Set as lattice

**lemma** *inf-set-eq*:  $A \sqcap B = A \cap B$   
**apply** (*rule subset-antisym*)  
**apply** (*rule Int-greatest*)  
**apply** (*rule inf-le1*)  
**apply** (*rule inf-le2*)  
**apply** (*rule inf-greatest*)  
**apply** (*rule Int-lower1*)  
**apply** (*rule Int-lower2*)  
**done**

**lemma** *sup-set-eq*:  $A \sqcup B = A \cup B$

```

apply (rule subset-antisym)
apply (rule sup-least)
apply (rule Un-upper1)
apply (rule Un-upper2)
apply (rule Un-least)
apply (rule sup-ge1)
apply (rule sup-ge2)
done

lemma mono-Int:  $\text{mono } f \implies f (A \cap B) \subseteq f A \cap f B$ 
apply (fold inf-set-eq sup-set-eq)
apply (erule mono-inf)
done

lemma mono-Un:  $\text{mono } f \implies f A \cup f B \subseteq f (A \cup B)$ 
apply (fold inf-set-eq sup-set-eq)
apply (erule mono-sup)
done

lemma Inf-set-eq:  $\bigcap S = \bigcap S$ 
apply (rule subset-antisym)
apply (rule Inter-greatest)
apply (erule Inf-lower)
apply (rule Inf-greatest)
apply (erule Inter-lower)
done

lemma Sup-set-eq:  $\bigcup S = \bigcup S$ 
apply (rule subset-antisym)
apply (rule Sup-least)
apply (erule Union-upper)
apply (rule Union-least)
apply (erule Sup-upper)
done

lemma top-set-eq:  $\text{top} = \text{UNIV}$ 
by (iprover intro!: subset-antisym subset-UNIV top-greatest)

lemma bot-set-eq:  $\text{bot} = \{\}$ 
by (iprover intro!: subset-antisym empty-subsetI bot-least)

no-notation
  less-eq (infix  $\sqsubseteq$  50) and
  less (infix  $\sqsubset$  50) and
  inf (infixl  $\sqcap$  70) and
  sup (infixl  $\sqcup$  65) and
  Inf (infix  $\bigcap$  - [900] 900) and
  Sup (infix  $\bigcup$  - [900] 900)

```

### 5.15 Misc theorem and ML bindings

**lemmas** *equalityI* = *subset-antisym*

**lemmas** *mem-simps* =

*insert-iff empty-iff Un-iff Int-iff Compl-iff Diff-iff*  
*mem-Collect-eq UN-iff Union-iff INT-iff Inter-iff*

— Each of these has ALREADY been added [*simp*] above.

**ML**  $\langle\langle$

*val Ball-def* = @{*thm Ball-def*}  
*val Bex-def* = @{*thm Bex-def*}  
*val CollectD* = @{*thm CollectD*}  
*val CollectE* = @{*thm CollectE*}  
*val CollectI* = @{*thm CollectI*}  
*val Collect-conj-eq* = @{*thm Collect-conj-eq*}  
*val Collect-mem-eq* = @{*thm Collect-mem-eq*}  
*val IntD1* = @{*thm IntD1*}  
*val IntD2* = @{*thm IntD2*}  
*val IntE* = @{*thm IntE*}  
*val IntI* = @{*thm IntI*}  
*val Int-Collect* = @{*thm Int-Collect*}  
*val UNIV-I* = @{*thm UNIV-I*}  
*val UNIV-witness* = @{*thm UNIV-witness*}  
*val UnE* = @{*thm UnE*}  
*val UnI1* = @{*thm UnI1*}  
*val UnI2* = @{*thm UnI2*}  
*val ballE* = @{*thm ballE*}  
*val ballI* = @{*thm ballI*}  
*val bexCI* = @{*thm bexCI*}  
*val bexE* = @{*thm bexE*}  
*val bexI* = @{*thm bexI*}  
*val bex-triv* = @{*thm bex-triv*}  
*val bspec* = @{*thm bspec*}  
*val contra-subsetD* = @{*thm contra-subsetD*}  
*val distinct-lemma* = @{*thm distinct-lemma*}  
*val eq-to-mono* = @{*thm eq-to-mono*}  
*val eq-to-mono2* = @{*thm eq-to-mono2*}  
*val equalityCE* = @{*thm equalityCE*}  
*val equalityD1* = @{*thm equalityD1*}  
*val equalityD2* = @{*thm equalityD2*}  
*val equalityE* = @{*thm equalityE*}  
*val equalityI* = @{*thm equalityI*}  
*val imageE* = @{*thm imageE*}  
*val imageI* = @{*thm imageI*}  
*val image-Un* = @{*thm image-Un*}  
*val image-insert* = @{*thm image-insert*}  
*val insert-commute* = @{*thm insert-commute*}  
*val insert-iff* = @{*thm insert-iff*}  
*val mem-Collect-eq* = @{*thm mem-Collect-eq*}  
*val rangeE* = @{*thm rangeE*}

```

val rangeI = @{thm rangeI}
val range-eqI = @{thm range-eqI}
val subsetCE = @{thm subsetCE}
val subsetD = @{thm subsetD}
val subsetI = @{thm subsetI}
val subset-refl = @{thm subset-refl}
val subset-trans = @{thm subset-trans}
val vimageD = @{thm vimageD}
val vimageE = @{thm vimageE}
val vimageI = @{thm vimageI}
val vimageI2 = @{thm vimageI2}
val vimage-Collect = @{thm vimage-Collect}
val vimage-Int = @{thm vimage-Int}
val vimage-Un = @{thm vimage-Un}
>>

end

```

## 6 Typedef: HOL type definitions

```

theory Typedef
imports Set
uses
  (Tools/typedef-package.ML)
  (Tools/typecopy-package.ML)
  (Tools/typedef-codegen.ML)
begin

ML <<
  structure HOL = struct val thy = theory HOL end;
  >> — belongs to theory HOL

locale type-definition =
  fixes Rep and Abs and A
  assumes Rep: Rep x ∈ A
    and Rep-inverse: Abs (Rep x) = x
    and Abs-inverse: y ∈ A ==> Rep (Abs y) = y
  — This will be axiomatized for each typedef!
begin

lemma Rep-inject:
  (Rep x = Rep y) = (x = y)
proof
  assume Rep x = Rep y
  then have Abs (Rep x) = Abs (Rep y) by (simp only:)
  moreover have Abs (Rep x) = x by (rule Rep-inverse)
  moreover have Abs (Rep y) = y by (rule Rep-inverse)
  ultimately show x = y by simp

```

```

next
  assume  $x = y$ 
  thus  $Rep\ x = Rep\ y$  by (simp only:)
qed

lemma Abs-inject:
  assumes  $x: x \in A$  and  $y: y \in A$ 
  shows  $(Abs\ x = Abs\ y) = (x = y)$ 
proof
  assume  $Abs\ x = Abs\ y$ 
  then have  $Rep\ (Abs\ x) = Rep\ (Abs\ y)$  by (simp only:)
  moreover from  $x$  have  $Rep\ (Abs\ x) = x$  by (rule Abs-inverse)
  moreover from  $y$  have  $Rep\ (Abs\ y) = y$  by (rule Abs-inverse)
  ultimately show  $x = y$  by simp
next
  assume  $x = y$ 
  thus  $Abs\ x = Abs\ y$  by (simp only:)
qed

lemma Rep-cases [cases set]:
  assumes  $y: y \in A$ 
  and hyp:  $!!x. y = Rep\ x ==> P$ 
  shows  $P$ 
proof (rule hyp)
  from  $y$  have  $Rep\ (Abs\ y) = y$  by (rule Abs-inverse)
  thus  $y = Rep\ (Abs\ y)$  ..
qed

lemma Abs-cases [cases type]:
  assumes  $r: !!y. x = Abs\ y ==> y \in A ==> P$ 
  shows  $P$ 
proof (rule r)
  have  $Abs\ (Rep\ x) = x$  by (rule Rep-inverse)
  thus  $x = Abs\ (Rep\ x)$  ..
  show  $Rep\ x \in A$  by (rule Rep)
qed

lemma Rep-induct [induct set]:
  assumes  $y: y \in A$ 
  and hyp:  $!!x. P\ (Rep\ x)$ 
  shows  $P\ y$ 
proof -
  have  $P\ (Rep\ (Abs\ y))$  by (rule hyp)
  moreover from  $y$  have  $Rep\ (Abs\ y) = y$  by (rule Abs-inverse)
  ultimately show  $P\ y$  by simp
qed

lemma Abs-induct [induct type]:
  assumes  $r: !!y. y \in A ==> P\ (Abs\ y)$ 

```

```

  shows  $P\ x$ 
proof -
  have  $Rep\ x \in A$  by (rule Rep)
  then have  $P\ (Abs\ (Rep\ x))$  by (rule r)
  moreover have  $Abs\ (Rep\ x) = x$  by (rule Rep-inverse)
  ultimately show  $P\ x$  by simp
qed

lemma Rep-range: range Rep = A
proof
  show range Rep  $\leq$  A using Rep by (auto simp add: image-def)
  show A  $\leq$  range Rep
  proof
    fix x assume  $x : A$ 
    hence  $x = Rep\ (Abs\ x)$  by (rule Abs-inverse [symmetric])
    thus  $x : range\ Rep$  by (rule range-eqI)
  qed
qed

lemma Abs-image: Abs ‘ A = UNIV
proof
  show Abs ‘ A  $\leq$  UNIV by (rule subset-UNIV)
next
  show UNIV  $\leq$  Abs ‘ A
  proof
    fix x
    have  $x = Abs\ (Rep\ x)$  by (rule Rep-inverse [symmetric])
    moreover have  $Rep\ x : A$  by (rule Rep)
    ultimately show  $x : Abs\ ‘\ A$  by (rule image-eqI)
  qed
qed

end

use Tools/typedef-package.ML setup TypedefPackage.setup
use Tools/typecopy-package.ML setup TypecopyPackage.setup
use Tools/typedef-codegen.ML setup TypedefCodegen.setup

end

```

## 7 Fun: Notions about functions

```

theory Fun
imports Set
begin

```

As a simplification rule, it replaces all function equalities by first-order equalities.

```

lemma expand-fun-eq:  $f = g \longleftrightarrow (\forall x. f\ x = g\ x)$ 
apply (rule iffI)
apply (simp (no-asm-simp))
apply (rule ext)
apply (simp (no-asm-simp))
done

```

```

lemma apply-inverse:
 $f\ x = u \implies (\bigwedge x. P\ x \implies g\ (f\ x) = x) \implies P\ x \implies x = g\ u$ 
by auto

```

## 7.1 The Identity Function $id$

```

definition
 $id :: 'a \Rightarrow 'a$ 
where
 $id = (\lambda x. x)$ 

```

```

lemma id-apply [simp]:  $id\ x = x$ 
by (simp add: id-def)

```

```

lemma image-ident [simp]:  $(\%x. x) \text{ ` } Y = Y$ 
by blast

```

```

lemma image-id [simp]:  $id \text{ ` } Y = Y$ 
by (simp add: id-def)

```

```

lemma vimage-ident [simp]:  $(\%x. x) \text{ -` } Y = Y$ 
by blast

```

```

lemma vimage-id [simp]:  $id \text{ -` } A = A$ 
by (simp add: id-def)

```

## 7.2 The Composition Operator $f \circ g$

```

definition
 $comp :: ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$  (infixl o 55)
where
 $f \circ g = (\lambda x. f\ (g\ x))$ 

```

```

notation (xsymbols)
 $comp$  (infixl o 55)

```

```

notation (HTML output)
 $comp$  (infixl o 55)

```

compatibility

**lemmas**  $o\text{-def} = comp\text{-def}$

**lemma** *o-apply* [simp]:  $(f \circ g) x = f (g x)$   
**by** (simp add: comp-def)

**lemma** *o-assoc*:  $f \circ (g \circ h) = f \circ g \circ h$   
**by** (simp add: comp-def)

**lemma** *id-o* [simp]:  $id \circ g = g$   
**by** (simp add: comp-def)

**lemma** *o-id* [simp]:  $f \circ id = f$   
**by** (simp add: comp-def)

**lemma** *image-compose*:  $(f \circ g)^{\circ} r = f^{\circ}(g^{\circ} r)$   
**by** (simp add: comp-def, blast)

**lemma** *UN-o*:  $UNION A (g \circ f) = UNION (f^{\circ} A) g$   
**by** (unfold comp-def, blast)

### 7.3 The Forward Composition Operator *fcomp*

**definition**

*fcomp* ::  $('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'c$  (infixl *o>* 60)  
**where**  
 $f \circ> g = (\lambda x. g (f x))$

**lemma** *fcomp-apply*:  $(f \circ> g) x = g (f x)$   
**by** (simp add: fcomp-def)

**lemma** *fcomp-assoc*:  $(f \circ> g) \circ> h = f \circ> (g \circ> h)$   
**by** (simp add: fcomp-def)

**lemma** *id-fcomp* [simp]:  $id \circ> g = g$   
**by** (simp add: fcomp-def)

**lemma** *fcomp-id* [simp]:  $f \circ> id = f$   
**by** (simp add: fcomp-def)

**no-notation** *fcomp* (infixl *o>* 60)

### 7.4 Injectivity and Surjectivity

**constdefs**

*inj-on* ::  $['a \Rightarrow 'b, 'a \text{ set}] \Rightarrow bool$  — injective  
*inj-on* *f* *A* == ! *x*:*A*. ! *y*:*A*. *f*(*x*)=*f*(*y*)  $\longrightarrow x=y$

A common special case: functions injective over the entire domain type.

**abbreviation**

*inj* *f* == *inj-on* *f* *UNIV*



**definition**

$\text{bij\_betw} :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow \text{bool}$  **where** — bijective  
 $[\text{code del}]: \text{bij\_betw } f \ A \ B \longleftrightarrow \text{inj\_on } f \ A \ \& \ f \, ' \ A = B$

**constdefs**

$\text{surj} :: ('a \Rightarrow 'b) \Rightarrow \text{bool}$   
 $\text{surj } f == ! y. ? x. y = f(x)$

$\text{bij} :: ('a \Rightarrow 'b) \Rightarrow \text{bool}$   
 $\text{bij } f == \text{inj } f \ \& \ \text{surj } f$

**lemma injI:**

**assumes**  $\bigwedge x \ y. f \ x = f \ y \implies x = y$   
**shows**  $\text{inj } f$   
**using** *assms* **unfolding** *inj-on-def* **by** *auto*

For Proofs in *Tools/datatype-rep-proofs*

**lemma datatype-injI:**

$(!! x. \text{ALL } y. f(x) = f(y) \longrightarrow x=y) \implies \text{inj}(f)$   
**by** (*simp add: inj-on-def*)

**theorem range-ex1-eq:**  $\text{inj } f \implies b : \text{range } f = (EX! x. b = f \ x)$ 

**by** (*unfold inj-on-def, blast*)

**lemma injD:**  $[\text{inj}(f); f(x) = f(y)] \implies x=y$ 

**by** (*simp add: inj-on-def*)

**lemma inj-eq:**  $\text{inj}(f) \implies (f(x) = f(y)) = (x=y)$ 

**by** (*force simp add: inj-on-def*)

**lemma inj-on-id[*simp*]:**  $\text{inj\_on } \text{id} \ A$ 

**by** (*simp add: inj-on-def*)

**lemma inj-on-id2[*simp*]:**  $\text{inj\_on } (\%x. x) \ A$ 

**by** (*simp add: inj-on-def*)

**lemma surj-id[*simp*]:**  $\text{surj } \text{id}$ 

**by** (*simp add: surj-def*)

**lemma bij-id[*simp*]:**  $\text{bij } \text{id}$ 

**by** (*simp add: bij-def inj-on-id surj-id*)

**lemma inj-onI:**

$(!! x \ y. [\text{inj\_on } f \ A; f(x) = f(y)] \implies x=y) \implies \text{inj\_on } f \ A$   
**by** (*simp add: inj-on-def*)

**lemma inj-on-inverseI:**  $(!!x. x:A \implies g(f(x)) = x) \implies \text{inj\_on } f \ A$ 

**by** (*auto dest: arg-cong [of concl: g] simp add: inj-on-def*)

**lemma** *inj-onD*:  $[[ \text{inj-on } f \ A; \ f(x)=f(y); \ x:A; \ y:A \ ]]$   $\implies x=y$   
**by** (*unfold inj-on-def, blast*)

**lemma** *inj-on-iff*:  $[[ \text{inj-on } f \ A; \ x:A; \ y:A \ ]]$   $\implies (f(x)=f(y)) = (x=y)$   
**by** (*blast dest!: inj-onD*)

**lemma** *comp-inj-on*:  
 $[[ \text{inj-on } f \ A; \ \text{inj-on } g \ (f'A) \ ]]$   $\implies \text{inj-on } (g \circ f) \ A$   
**by** (*simp add: comp-def inj-on-def*)

**lemma** *inj-on-imageI*:  $\text{inj-on } (g \circ f) \ A \implies \text{inj-on } g \ (f' A)$   
**apply**(*simp add: inj-on-def image-def*)  
**apply** *blast*  
**done**

**lemma** *inj-on-image-iff*:  $[[ \text{ALL } x:A. \ \text{ALL } y:A. \ (g(f \ x) = g(f \ y)) = (g \ x = g \ y); \ \text{inj-on } f \ A \ ]]$   $\implies \text{inj-on } g \ (f' A) = \text{inj-on } g \ A$   
**apply**(*unfold inj-on-def*)  
**apply** *blast*  
**done**

**lemma** *inj-on-contrad*:  $[[ \text{inj-on } f \ A; \ \sim x=y; \ x:A; \ y:A \ ]]$   $\implies \sim f(x)=f(y)$   
**by** (*unfold inj-on-def, blast*)

**lemma** *inj-singleton*:  $\text{inj } (\%s. \ \{s\})$   
**by** (*simp add: inj-on-def*)

**lemma** *inj-on-empty[iff]*:  $\text{inj-on } f \ \{\}$   
**by**(*simp add: inj-on-def*)

**lemma** *subset-inj-on*:  $[[ \text{inj-on } f \ B; \ A \leq B \ ]]$   $\implies \text{inj-on } f \ A$   
**by** (*unfold inj-on-def, blast*)

**lemma** *inj-on-Un*:  
 $\text{inj-on } f \ (A \cup B) =$   
 $(\text{inj-on } f \ A \ \& \ \text{inj-on } f \ B \ \& \ f'(A-B) \ \text{Int } f'(B-A) = \{\})$   
**apply**(*unfold inj-on-def*)  
**apply** (*blast intro:sym*)  
**done**

**lemma** *inj-on-insert[iff]*:  
 $\text{inj-on } f \ (\text{insert } a \ A) = (\text{inj-on } f \ A \ \& \ f \ a \ \sim: f'(A-\{a\}))$   
**apply**(*unfold inj-on-def*)  
**apply** (*blast intro:sym*)  
**done**

**lemma** *inj-on-diff*:  $\text{inj-on } f \ A \implies \text{inj-on } f \ (A-B)$   
**apply**(*unfold inj-on-def*)

**apply** (*blast*)  
**done**

**lemma** *surjI*: ( $\forall x. g(f\ x) = x$ )  $\implies$  *surj* *g*  
**apply** (*simp add: surj-def*)  
**apply** (*blast intro: sym*)  
**done**

**lemma** *surj-range*: *surj* *f*  $\implies$  *range* *f* = *UNIV*  
**by** (*auto simp add: surj-def*)

**lemma** *surjD*: *surj* *f*  $\implies$   $\exists x. y = f\ x$   
**by** (*simp add: surj-def*)

**lemma** *surjE*: *surj* *f*  $\implies$  ( $\forall x. y = f\ x \implies C$ )  $\implies$  *C*  
**by** (*simp add: surj-def, blast*)

**lemma** *comp-surj*: [*surj* *f*; *surj* *g*]  $\implies$  *surj* (*g* *o* *f*)  
**apply** (*simp add: comp-def surj-def, clarify*)  
**apply** (*drule-tac x = y in spec, clarify*)  
**apply** (*drule-tac x = x in spec, blast*)  
**done**

**lemma** *bijI*: [*inj* *f*; *surj* *f*]  $\implies$  *bij* *f*  
**by** (*simp add: bij-def*)

**lemma** *bij-is-inj*: *bij* *f*  $\implies$  *inj* *f*  
**by** (*simp add: bij-def*)

**lemma** *bij-is-surj*: *bij* *f*  $\implies$  *surj* *f*  
**by** (*simp add: bij-def*)

**lemma** *bij-betw-imp-inj-on*: *bij-betw* *f* *A* *B*  $\implies$  *inj-on* *f* *A*  
**by** (*simp add: bij-betw-def*)

**lemma** *bij-betw-inv*: **assumes** *bij-betw* *f* *A* *B* **shows**  $\exists g. \text{bij-betw } g\ B\ A$   
**proof** –

**have** *i*: *inj-on* *f* *A* **and** *s*: *f* ‘ *A* = *B*

**using** *assms* **by** (*auto simp: bij-betw-def*)

**let**  $?P = \%b\ a. a:A \wedge f\ a = b$  **let**  $?g = \%b. \text{The } (?P\ b)$

**{ fix** *a* *b* **assume** *P*:  $?P\ b\ a$

**hence** *ex1*:  $\exists a. ?P\ b\ a$  **using** *s* **unfolding** *image-def* **by** *blast*

**hence** *uex1*:  $\exists! a. ?P\ b\ a$  **by** (*blast dest: inj-onD[OF i]*)

**hence**  $?g\ b = a$  **using** *the1-equality*[*OF uex1, OF P*] *P* **by** *simp*

**}** **note** *g* = *this*

**have** *inj-on*  $?g\ B$

**proof**(*rule inj-onI*)

**fix** *x* *y* **assume** *x*:*B* *y*:*B*  $?g\ x = ?g\ y$

**from** *s*  $\langle x:B \rangle$  **obtain** *a1* **where** *a1*:  $?P\ x\ a1$  **unfolding** *image-def* **by** *blast*

```

    from  $s \langle y:B \rangle$  obtain  $a2$  where  $a2: ?P \ y \ a2$  unfolding image-def by blast
    from  $g[OF \ a1] \ a1 \ g[OF \ a2] \ a2 \ \langle ?g \ x = ?g \ y \rangle$  show  $x=y$  by simp
qed
moreover have  $?g \ ' \ B = A$ 
proof(auto simp: image-def)
  fix  $b$  assume  $b:B$ 
  with  $s$  obtain  $a$  where  $P: ?P \ b \ a$  unfolding image-def by blast
  thus  $?g \ b \in A$  using  $g[OF \ P]$  by auto
next
  fix  $a$  assume  $a:A$ 
  then obtain  $b$  where  $P: ?P \ b \ a$  using  $s$  unfolding image-def by blast
  then have  $b:B$  using  $s$  unfolding image-def by blast
  with  $g[OF \ P]$  show  $\exists b \in B. \ a = ?g \ b$  by blast
qed
ultimately show ?thesis by(auto simp: bij-betw-def)
qed

```

```

lemma surj-image-vimage-eq:  $\text{surj } f \implies f \ ' \ (f \ -' \ A) = A$ 
by (simp add: surj-range)

```

```

lemma inj-vimage-image-eq:  $\text{inj } f \implies f \ -' \ (f \ ' \ A) = A$ 
by (simp add: inj-on-def, blast)

```

```

lemma vimage-subsetD:  $\text{surj } f \implies f \ -' \ B \leq A \implies B \leq f \ ' \ A$ 
apply (unfold surj-def)
apply (blast intro: sym)
done

```

```

lemma vimage-subsetI:  $\text{inj } f \implies B \leq f \ ' \ A \implies f \ -' \ B \leq A$ 
by (unfold inj-on-def, blast)

```

```

lemma vimage-subset-eq:  $\text{bij } f \implies (f \ -' \ B \leq A) = (B \leq f \ ' \ A)$ 
apply (unfold bij-def)
apply (blast del: subsetI intro: vimage-subsetI vimage-subsetD)
done

```

```

lemma inj-on-image-Int:
   $[\text{inj-on } f \ C; \ A \leq C; \ B \leq C] \implies f \ '(A \ \text{Int } B) = f \ 'A \ \text{Int } f \ 'B$ 
apply (simp add: inj-on-def, blast)
done

```

```

lemma inj-on-image-set-diff:
   $[\text{inj-on } f \ C; \ A \leq C; \ B \leq C] \implies f \ '(A - B) = f \ 'A - f \ 'B$ 
apply (simp add: inj-on-def, blast)
done

```

```

lemma image-Int:  $\text{inj } f \implies f \ '(A \ \text{Int } B) = f \ 'A \ \text{Int } f \ 'B$ 
by (simp add: inj-on-def, blast)

```

**lemma** *image-set-diff*:  $\text{inj } f \implies f'(A-B) = f'A - f'B$   
**by** (*simp add: inj-on-def, blast*)

**lemma** *inj-image-mem-iff*:  $\text{inj } f \implies (f \ a : f'A) = (a : A)$   
**by** (*blast dest: injD*)

**lemma** *inj-image-subset-iff*:  $\text{inj } f \implies (f'A \leq f'B) = (A \leq B)$   
**by** (*simp add: inj-on-def, blast*)

**lemma** *inj-image-eq-iff*:  $\text{inj } f \implies (f'A = f'B) = (A = B)$   
**by** (*blast dest: injD*)

**lemma** *image-INT*:  
 $\llbracket \text{inj-on } f \ C; \ \text{ALL } x:A. B \ x \leq C; \ j:A \rrbracket$   
 $\implies f' \ (INTER \ A \ B) = (INT \ x:A. f' \ B \ x)$   
**apply** (*simp add: inj-on-def, blast*)  
**done**

**lemma** *bij-image-INT*:  $\text{bij } f \implies f' \ (INTER \ A \ B) = (INT \ x:A. f' \ B \ x)$   
**apply** (*simp add: bij-def*)  
**apply** (*simp add: inj-on-def surj-def, blast*)  
**done**

**lemma** *surj-Compl-image-subset*:  $\text{surj } f \implies \neg(f'A) \leq f'(\neg A)$   
**by** (*auto simp add: surj-def*)

**lemma** *inj-image-Compl-subset*:  $\text{inj } f \implies f'(\neg A) \leq \neg(f'A)$   
**by** (*auto simp add: inj-on-def*)

**lemma** *bij-image-Compl-eq*:  $\text{bij } f \implies f'(\neg A) = \neg(f'A)$   
**apply** (*simp add: bij-def*)  
**apply** (*rule equalityI*)  
**apply** (*simp-all (no-asm-simp) add: inj-image-Compl-subset surj-Compl-image-subset*)  
**done**

## 7.5 Function Updating

**constdefs**

*fun-upd* ::  $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b)$   
*fun-upd*  $f \ a \ b == \% x. \text{if } x=a \text{ then } b \text{ else } f \ x$

**nonterminals**

*updbinds upbind*

**syntax**

*-upbind* ::  $['a, 'a] \Rightarrow \text{upbind} \quad ((2- := / -))$   
 $\quad \quad \quad :: \text{upbind} \Rightarrow \text{updbinds} \quad (-)$   
*-updbinds*::  $[\text{upbind}, \text{updbinds}] \Rightarrow \text{updbinds} \ (-, / -)$

$-Update :: [a, updbinds] \Rightarrow a \quad (-/'((-)') [1000,0] 900)$

#### translations

$-Update f (-updbinds b bs) == -Update (-Update f b) bs$   
 $f(x:=y) == fun-upd f x y$

**lemma** *fun-upd-idem-iff*:  $(f(x:=y) = f) = (f x = y)$

**apply** (*simp add: fun-upd-def, safe*)

**apply** (*erule subst*)

**apply** (*rule-tac [2] ext, auto*)

**done**

**lemmas** *fun-upd-idem = fun-upd-idem-iff [THEN iffD2, standard]*

**lemmas** *fun-upd-triv = refl [THEN fun-upd-idem]*

**declare** *fun-upd-triv [iff]*

**lemma** *fun-upd-apply [simp]*:  $(f(x:=y))z = (if z=x then y else f z)$

**by** (*simp add: fun-upd-def*)

**lemma** *fun-upd-same*:  $(f(x:=y)) x = y$

**by** *simp*

**lemma** *fun-upd-other*:  $z \sim x \Rightarrow (f(x:=y)) z = f z$

**by** *simp*

**lemma** *fun-upd-upd [simp]*:  $f(x:=y, x:=z) = f(x:=z)$

**by** (*simp add: expand-fun-eq*)

**lemma** *fun-upd-twist*:  $a \sim c \Rightarrow (m(a:=b))(c:=d) = (m(c:=d))(a:=b)$

**by** (*rule ext, auto*)

**lemma** *inj-on-fun-updI*:  $\llbracket inj-on f A; y \notin f'A \rrbracket \Longrightarrow inj-on (f(x:=y)) A$

**by** (*fastsimp simp: inj-on-def image-def*)

**lemma** *fun-upd-image*:

$f(x:=y) \text{ ` } A = (if x \in A then insert y (f \text{ ` } (A - \{x\})) else f \text{ ` } A)$

**by** *auto*

## 7.6 override-on

### definition

*override-on* ::  $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow 'a \Rightarrow 'b$

**where**

$\text{override-on } f \ g \ A = (\lambda a. \text{ if } a \in A \text{ then } g \ a \text{ else } f \ a)$

**lemma** *override-on-emptyset*[simp]:  $\text{override-on } f \ g \ \{\} = f$   
**by** (simp add: override-on-def)

**lemma** *override-on-apply-notin*[simp]:  $a \sim: A \implies (\text{override-on } f \ g \ A) \ a = f \ a$   
**by** (simp add: override-on-def)

**lemma** *override-on-apply-in*[simp]:  $a : A \implies (\text{override-on } f \ g \ A) \ a = g \ a$   
**by** (simp add: override-on-def)

## 7.7 swap

**definition**

$\text{swap} :: 'a \Rightarrow 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$

**where**

$\text{swap } a \ b \ f = f \ (a := f \ b, b := f \ a)$

**lemma** *swap-self*:  $\text{swap } a \ a \ f = f$   
**by** (simp add: swap-def)

**lemma** *swap-commute*:  $\text{swap } a \ b \ f = \text{swap } b \ a \ f$   
**by** (rule ext, simp add: fun-upd-def swap-def)

**lemma** *swap-nilpotent* [simp]:  $\text{swap } a \ b \ (\text{swap } a \ b \ f) = f$   
**by** (rule ext, simp add: fun-upd-def swap-def)

**lemma** *inj-on-imp-inj-on-swap*:

$[[\text{inj-on } f \ A; a \in A; b \in A]] \implies \text{inj-on } (\text{swap } a \ b \ f) \ A$

**by** (simp add: inj-on-def swap-def, blast)

**lemma** *inj-on-swap-iff* [simp]:

**assumes**  $A: a \in A \ b \in A$  **shows**  $\text{inj-on } (\text{swap } a \ b \ f) \ A = \text{inj-on } f \ A$

**proof**

**assume**  $\text{inj-on } (\text{swap } a \ b \ f) \ A$

**with**  $A$  **have**  $\text{inj-on } (\text{swap } a \ b \ (\text{swap } a \ b \ f)) \ A$

**by** (iprover intro: inj-on-imp-inj-on-swap)

**thus**  $\text{inj-on } f \ A$  **by** simp

**next**

**assume**  $\text{inj-on } f \ A$

**with**  $A$  **show**  $\text{inj-on } (\text{swap } a \ b \ f) \ A$  **by** (iprover intro: inj-on-imp-inj-on-swap)

**qed**

**lemma** *surj-imp-surj-swap*:  $\text{surj } f \implies \text{surj } (\text{swap } a \ b \ f)$

**apply** (simp add: surj-def swap-def, clarify)

**apply** (case-tac  $y = f \ b$ , blast)

**apply** (case-tac  $y = f \ a$ , auto)

**done**

```

lemma surj-swap-iff [simp]: surj (swap a b f) = surj f
proof
  assume surj (swap a b f)
  hence surj (swap a b (swap a b f)) by (rule surj-imp-surj-swap)
  thus surj f by simp
next
  assume surj f
  thus surj (swap a b f) by (rule surj-imp-surj-swap)
qed

lemma bij-swap-iff: bij (swap a b f) = bij f
by (simp add: bij-def)

hide (open) const swap

```

## 7.8 Proof tool setup

simplifies terms of the form  $f(\dots, x := y, \dots, x := z, \dots)$  to  $f(\dots, x := z, \dots)$

```

simproc-setup fun-upd2 (f (v := w, x := y)) = << fn - =>
  let
    fun gen-fun-upd NONE T - - = NONE
    | gen-fun-upd (SOME f) T x y = SOME (Const (@{const-name fun-upd}, T)
    $ f $ x $ y)
    fun dest-fun-T1 (Type (-, T :: Ts)) = T
    fun find-double (t as Const (@{const-name fun-upd}, T) $ f $ x $ y) =
      let
        fun find (Const (@{const-name fun-upd}, T) $ g $ v $ w) =
          if v aconv x then SOME g else gen-fun-upd (find g) T v w
          | find t = NONE
        in (dest-fun-T1 T, gen-fun-upd (find f) T x y) end

    fun proc ss ct =
      let
        val ctxt = Simplifier.the-context ss
        val t = Thm.term-of ct
      in
        case find-double t of
          (T, NONE) => NONE
        | (T, SOME rhs) =>
          SOME (Goal.prove ctxt [] [] (Logic.mk-equals (t, rhs))
            (fn - =>
              rtac eq-reflection 1 THEN
              rtac ext 1 THEN
              simp-tac (Simplifier.inherit-context ss @{simpset} 1))
            )
      end
    in proc end
  >>

```



## 7.9 Code generator setup

### types-code

```

  fun ((- --> / -))
attach (term-of) ⟨⟨
  fun term-of-fun-type - aT - bT - = Free (<function>, aT --> bT);
  ⟩⟩
attach (test) ⟨⟨
  fun gen-fun-type aF aT bG bT i =
    let
      val tab = ref [];
      fun mk-upd (x, (-, y)) t = Const (Fun.fun-upd,
        (aT --> bT) --> aT --> bT --> aT --> bT) $ t $ aF x $ y ()
    in
      (fn x =>
        case AList.lookup op = (!tab) x of
          NONE =>
            let val p as (y, -) = bG i
            in (tab := (x, p) :: !tab; y) end
          | SOME (y, -) => y,
        fn () => Basics.fold mk-upd (!tab) (Const (HOL.undefined, aT --> bT)))
      end;
    ⟩⟩

```

```

code-const op ◦
  (SML infixl 5 o)
  (Haskell infixr 9 .)

```

```

code-const id
  (Haskell id)

```

```

end

```

## 8 Sum-Type: The Disjoint Sum of Two Types

```

theory Sum-Type
imports Typedef Fun
begin

```

The representations of the two injections

```

constdefs
  Inl-Rep :: ['a, 'a, 'b, bool] => bool
  Inl-Rep == (%a. %x y p. x=a & p)

  Inr-Rep :: ['b, 'a, 'b, bool] => bool
  Inr-Rep == (%b. %x y p. y=b & ~p)

```

**global**

```
typedef (Sum)
  ('a, 'b) +      (infixr + 10)
  = {f. (? a. f = Inl-Rep(a::'a)) | (? b. f = Inr-Rep(b::'b))}
by auto
```

**local**

abstract constants and syntax

**constdefs**

```
Inl :: 'a => 'a + 'b
Inl == (%a. Abs-Sum(Inl-Rep(a)))
```

```
Inr :: 'b => 'a + 'b
Inr == (%b. Abs-Sum(Inr-Rep(b)))
```

```
Plus :: ['a set, 'b set] => ('a + 'b) set      (infixr <+> 65)
A <+> B == (Inl'A) Un (Inr'B)
— disjoint sum for sets; the operator + is overloaded with wrong type!
```

```
Part :: ['a set, 'b => 'a] => 'a set
Part A h == A Int {x. ? z. x = h(z)}
— for selecting out the components of a mutually recursive definition
```

```
lemma Inl-RepI: Inl-Rep(a) : Sum
by (auto simp add: Sum-def)
```

```
lemma Inr-RepI: Inr-Rep(b) : Sum
by (auto simp add: Sum-def)
```

```
lemma inj-on-Abs-Sum: inj-on Abs-Sum Sum
apply (rule inj-on-inverseI)
apply (erule Abs-Sum-inverse)
done
```

## 8.1 Freeness Properties for *Inl* and *Inr*

Distinctness

```
lemma Inl-Rep-not-Inr-Rep: Inl-Rep(a) ~ = Inr-Rep(b)
by (auto simp add: Inl-Rep-def Inr-Rep-def expand-fun-eq)
```

```
lemma Inl-not-Inr [iff]: Inl(a) ~ = Inr(b)
apply (simp add: Inl-def Inr-def)
```

```

apply (rule inj-on-Abs-Sum [THEN inj-on-contrad])
apply (rule Inl-Rep-not-Inr-Rep)
apply (rule Inl-RepI)
apply (rule Inr-RepI)
done

```

```

lemmas Inr-not-Inl = Inl-not-Inr [THEN not-sym, standard]
declare Inr-not-Inl [iff]

```

```

lemmas Inl-neq-Inr = Inl-not-Inr [THEN notE, standard]
lemmas Inr-neq-Inl = sym [THEN Inl-neq-Inr, standard]

```

Injectiveness

```

lemma Inl-Rep-inject: Inl-Rep(a) = Inl-Rep(c) ==> a=c
by (auto simp add: Inl-Rep-def expand-fun-eq)

```

```

lemma Inr-Rep-inject: Inr-Rep(b) = Inr-Rep(d) ==> b=d
by (auto simp add: Inr-Rep-def expand-fun-eq)

```

```

lemma inj-Inl [simp]: inj-on Inl A
apply (simp add: Inl-def)
apply (rule inj-onI)
apply (erule inj-on-Abs-Sum [THEN inj-onD, THEN Inl-Rep-inject])
apply (rule Inl-RepI)
apply (rule Inl-RepI)
done

```

```

lemmas Inl-inject = inj-Inl [THEN injD, standard]

```

```

lemma inj-Inr [simp]: inj-on Inr A
apply (simp add: Inr-def)
apply (rule inj-onI)
apply (erule inj-on-Abs-Sum [THEN inj-onD, THEN Inr-Rep-inject])
apply (rule Inr-RepI)
apply (rule Inr-RepI)
done

```

```

lemmas Inr-inject = inj-Inr [THEN injD, standard]

```

```

lemma Inl-eq [iff]: (Inl(x)=Inl(y)) = (x=y)
by (blast dest!: Inl-inject)

```

```

lemma Inr-eq [iff]: (Inr(x)=Inr(y)) = (x=y)
by (blast dest!: Inr-inject)

```

## 8.2 The Disjoint Sum of Sets

```

lemma InlI [intro!]: a : A ==> Inl(a) : A <+> B
by (simp add: Plus-def)

```

**lemma** *InrI* [*intro!*]:  $b : B \implies \text{Inr}(b) : A <+> B$   
**by** (*simp add: Plus-def*)

**lemma** *PlusE* [*elim!*]:  

$$\begin{aligned} & [| u : A <+> B; \\ & \quad !!x. [| x:A; u=\text{Inl}(x) |] \implies P; \\ & \quad !!y. [| y:B; u=\text{Inr}(y) |] \implies P \\ & |] \implies P \end{aligned}$$
  
**by** (*auto simp add: Plus-def*)

Exhaustion rule for sums, a degenerate form of induction

**lemma** *sumE*:  

$$\begin{aligned} & [| !!x::'a. s = \text{Inl}(x) \implies P; !!y::'b. s = \text{Inr}(y) \implies P \\ & |] \implies P \end{aligned}$$
  
**apply** (*rule Abs-Sum-cases [of s]*)  
**apply** (*auto simp add: Sum-def Inl-def Inr-def*)  
**done**

**lemma** *UNIV-Plus-UNIV* [*simp*]:  $\text{UNIV} <+> \text{UNIV} = \text{UNIV}$   
**apply** (*rule set-ext*)  
**apply** (*rename-tac s*)  
**apply** (*rule-tac s=s in sumE*)  
**apply** *auto*  
**done**

### 8.3 The Part Primitive

**lemma** *Part-eqI* [*intro*]:  $[| a : A; a=h(b) |] \implies a : \text{Part } A \ h$   
**by** (*auto simp add: Part-def*)

**lemmas** *PartI* = *Part-eqI* [*OF - refl, standard*]

**lemma** *PartE* [*elim!*]:  $[| a : \text{Part } A \ h; !!z. [| a : A; a=h(z) |] \implies P |] \implies P$   
**by** (*auto simp add: Part-def*)

**lemma** *Part-subset*:  $\text{Part } A \ h \leq A$   
**by** (*auto simp add: Part-def*)

**lemma** *Part-mono*:  $A \leq B \implies \text{Part } A \ h \leq \text{Part } B \ h$   
**by** *blast*

**lemmas** *basic-monos* = *basic-monos Part-mono*

**lemma** *PartD1*:  $a : \text{Part } A \ h \implies a : A$

**by** (*simp add: Part-def*)

**lemma** *Part-id*:  $\text{Part } A \ (\%x. x) = A$   
**by** *blast*

**lemma** *Part-Int*:  $\text{Part } (A \text{ Int } B) \ h = (\text{Part } A \ h) \text{ Int } (\text{Part } B \ h)$   
**by** *blast*

**lemma** *Part-Collect*:  $\text{Part } (A \text{ Int } \{x. P \ x\}) \ h = (\text{Part } A \ h) \text{ Int } \{x. P \ x\}$   
**by** *blast*

**ML**

```

⟨⟨
  val Inl-RepI = thm Inl-RepI;
  val Inr-RepI = thm Inr-RepI;
  val inj-on-Abs-Sum = thm inj-on-Abs-Sum;
  val Inl-Rep-not-Inr-Rep = thm Inl-Rep-not-Inr-Rep;
  val Inl-not-Inr = thm Inl-not-Inr;
  val Inr-not-Inl = thm Inr-not-Inl;
  val Inl-neq-Inr = thm Inl-neq-Inr;
  val Inr-neq-Inl = thm Inr-neq-Inl;
  val Inl-Rep-inject = thm Inl-Rep-inject;
  val Inr-Rep-inject = thm Inr-Rep-inject;
  val inj-Inl = thm inj-Inl;
  val Inl-inject = thm Inl-inject;
  val inj-Inr = thm inj-Inr;
  val Inr-inject = thm Inr-inject;
  val Inl-eq = thm Inl-eq;
  val Inr-eq = thm Inr-eq;
  val InlI = thm InlI;
  val InrI = thm InrI;
  val PlusE = thm PlusE;
  val sumE = thm sumE;
  val Part-eqI = thm Part-eqI;
  val PartI = thm PartI;
  val PartE = thm PartE;
  val Part-subset = thm Part-subset;
  val Part-mono = thm Part-mono;
  val PartD1 = thm PartD1;
  val Part-id = thm Part-id;
  val Part-Int = thm Part-Int;
  val Part-Collect = thm Part-Collect;

  val basic-monos = thms basic-monos;
  ⟩⟩

```

**end**

## 9 Inductive: Knaster-Tarski Fixpoint Theorem and inductive definitions

```

theory Inductive
imports Lattices Sum-Type
uses
  (Tools/inductive-package.ML)
  Tools/dseq.ML
  (Tools/inductive-codegen.ML)
  (Tools/datatype-aux.ML)
  (Tools/datatype-prop.ML)
  (Tools/datatype-rep-proofs.ML)
  (Tools/datatype-abs-proofs.ML)
  (Tools/datatype-case.ML)
  (Tools/datatype-package.ML)
  (Tools/old-primrec-package.ML)
  (Tools/primrec-package.ML)
  (Tools/datatype-codegen.ML)
begin

```

### 9.1 Least and greatest fixed points

```

context complete-lattice
begin

```

```

definition
  lfp :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a where
  lfp f = Inf {u. f u  $\leq$  u} — least fixed point

```

```

definition
  gfp :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a where
  gfp f = Sup {u. u  $\leq$  f u} — greatest fixed point

```

### 9.2 Proof of Knaster-Tarski Theorem using *lfp*

*lfp f* is the least upper bound of the set  $\{u. f u \leq u\}$

```

lemma lfp-lowerbound: f A  $\leq$  A  $\implies$  lfp f  $\leq$  A
  by (auto simp add: lfp-def intro: Inf-lower)

```

```

lemma lfp-greatest: (!u. f u  $\leq$  u  $\implies$  A  $\leq$  u)  $\implies$  A  $\leq$  lfp f
  by (auto simp add: lfp-def intro: Inf-greatest)

```

**end**

```

lemma lfp-lemma2: mono f  $\implies$  f (lfp f)  $\leq$  lfp f
  by (iprover intro: lfp-greatest order-trans monoD lfp-lowerbound)

```

```

lemma lfp-lemma3: mono f  $\implies$  lfp f  $\leq$  f (lfp f)
  by (iprover intro: lfp-lemma2 monoD lfp-lowerbound)

```

**lemma** *lfp-unfold*:  $\text{mono } f \implies \text{lfp } f = f (\text{lfp } f)$   
**by** (*iprover* *intro*: *order-antisym* *lfp-lemma2* *lfp-lemma3*)

**lemma** *lfp-const*:  $\text{lfp } (\lambda x. t) = t$   
**by** (*rule* *lfp-unfold*) (*simp* *add:mono-def*)

### 9.3 General induction rules for least fixed points

**theorem** *lfp-induct*:  
**assumes** *mono*:  $\text{mono } f$  **and** *ind*:  $f (\inf (\text{lfp } f) P) \leq P$   
**shows**  $\text{lfp } f \leq P$   
**proof** –  
**have**  $\inf (\text{lfp } f) P \leq \text{lfp } f$  **by** (*rule* *inf-le1*)  
**with** *mono* **have**  $f (\inf (\text{lfp } f) P) \leq f (\text{lfp } f)$  **..**  
**also from** *mono* **have**  $f (\text{lfp } f) = \text{lfp } f$  **by** (*rule* *lfp-unfold* [*symmetric*])  
**finally have**  $f (\inf (\text{lfp } f) P) \leq \text{lfp } f$  **.**  
**from this and** *ind* **have**  $f (\inf (\text{lfp } f) P) \leq \inf (\text{lfp } f) P$  **by** (*rule* *le-infI*)  
**hence**  $\text{lfp } f \leq \inf (\text{lfp } f) P$  **by** (*rule* *lfp-lowerbound*)  
**also have**  $\inf (\text{lfp } f) P \leq P$  **by** (*rule* *inf-le2*)  
**finally show** *?thesis* **.**  
**qed**

**lemma** *lfp-induct-set*:  
**assumes** *lfp*:  $a: \text{lfp}(f)$   
**and** *mono*:  $\text{mono}(f)$   
**and** *indhyp*:  $!!x. [\mid x: f(\text{lfp}(f) \text{ Int } \{x. P(x)\}) \mid] \implies P(x)$   
**shows**  $P(a)$   
**by** (*rule* *lfp-induct* [*THEN* *subsetD*, *THEN* *CollectD*, *OF* *mono - lfp*])  
*(auto simp: inf-set-eq intro: indhyp)*

**lemma** *lfp-ordinal-induct*:  
**fixes**  $f :: 'a::\text{complete-lattice} \Rightarrow 'a$   
**assumes** *mono*:  $\text{mono } f$   
**and** *P-f*:  $\bigwedge S. P S \implies P (f S)$   
**and** *P-Union*:  $\bigwedge M. \forall S \in M. P S \implies P (\text{Sup } M)$   
**shows**  $P (\text{lfp } f)$

**proof** –  
**let**  $?M = \{S. S \leq \text{lfp } f \wedge P S\}$   
**have**  $P (\text{Sup } ?M)$  **using** *P-Union* **by** *simp*  
**also have**  $\text{Sup } ?M = \text{lfp } f$   
**proof** (*rule antisym*)  
**show**  $\text{Sup } ?M \leq \text{lfp } f$  **by** (*blast* *intro*: *Sup-least*)  
**hence**  $f (\text{Sup } ?M) \leq f (\text{lfp } f)$  **by** (*rule* *mono* [*THEN* *monoD*])  
**hence**  $f (\text{Sup } ?M) \leq \text{lfp } f$  **using** *mono* [*THEN* *lfp-unfold*] **by** *simp*  
**hence**  $f (\text{Sup } ?M) \in ?M$  **using** *P-f* *P-Union* **by** *simp*  
**hence**  $f (\text{Sup } ?M) \leq \text{Sup } ?M$  **by** (*rule* *Sup-upper*)  
**thus**  $\text{lfp } f \leq \text{Sup } ?M$  **by** (*rule* *lfp-lowerbound*)  
**qed**

**finally show** *?thesis* .  
**qed**

**lemma** *lfp-ordinal-induct-set*:  
**assumes** *mono*: *mono f*  
**and** *P-f*:  $!!S. P\ S ==> P(f\ S)$   
**and** *P-Union*:  $!!M. !S:M. P\ S ==> P(\text{Union } M)$   
**shows**  $P(\text{lfp } f)$   
**using** *assms* **unfolding** *Sup-set-eq* [*symmetric*]  
**by** (*rule lfp-ordinal-induct* [**where**  $P=P$ ])

Definition forms of *lfp-unfold* and *lfp-induct*, to control unfolding

**lemma** *def-lfp-unfold*:  $[[\ h==\text{lfp}(f); \text{mono}(f)\ ]] ==> h = f(h)$   
**by** (*auto intro!*: *lfp-unfold*)

**lemma** *def-lfp-induct*:  
 $[[\ A == \text{lfp}(f); \text{mono}(f);$   
 $\quad f(\inf A\ P) \leq P$   
 $]] ==> A \leq P$   
**by** (*blast intro*: *lfp-induct*)

**lemma** *def-lfp-induct-set*:  
 $[[\ A == \text{lfp}(f); \text{mono}(f); \ a:A;$   
 $\quad !!x. [[\ x: f(A\ \text{Int } \{x. P(x)\})\ ]] ==> P(x)$   
 $]] ==> P(a)$   
**by** (*blast intro*: *lfp-induct-set*)

**lemma** *lfp-mono*:  $(!!Z. f\ Z \leq g\ Z) ==> \text{lfp } f \leq \text{lfp } g$   
**by** (*rule lfp-lowerbound* [*THEN lfp-greatest*], *blast intro*: *order-trans*)

## 9.4 Proof of Knaster-Tarski Theorem using *gfp*

*gfp f* is the greatest lower bound of the set  $\{u. u \leq f\ u\}$

**lemma** *gfp-upperbound*:  $X \leq f\ X ==> X \leq \text{gfp } f$   
**by** (*auto simp add*: *gfp-def intro*: *Sup-upper*)

**lemma** *gfp-least*:  $(!!u. u \leq f\ u ==> u \leq X) ==> \text{gfp } f \leq X$   
**by** (*auto simp add*: *gfp-def intro*: *Sup-least*)

**lemma** *gfp-lemma2*:  $\text{mono } f ==> \text{gfp } f \leq f(\text{gfp } f)$   
**by** (*iprover intro*: *gfp-least order-trans monoD gfp-upperbound*)

**lemma** *gfp-lemma3*:  $\text{mono } f ==> f(\text{gfp } f) \leq \text{gfp } f$   
**by** (*iprover intro*: *gfp-lemma2 monoD gfp-upperbound*)

**lemma** *gfp-unfold*:  $\text{mono } f ==> \text{gfp } f = f(\text{gfp } f)$   
**by** (*iprover intro*: *order-antisym gfp-lemma2 gfp-lemma3*)



## 9.5 Coinduction rules for greatest fixed points

weak version

```
lemma weak-coinduct: [|  $a : X$ ;  $X \subseteq f(X)$  |] ==>  $a : \text{gfp}(f)$ 
by (rule gfp-upperbound [THEN subsetD], auto)
```

```
lemma weak-coinduct-image: !! $X$ . [|  $a : X$ ;  $g'X \subseteq f(g'X)$  |] ==>  $g a : \text{gfp } f$ 
apply (erule gfp-upperbound [THEN subsetD])
apply (erule imageI)
done
```

```
lemma coinduct-lemma:
  [|  $X \leq f(\text{sup } X (\text{gfp } f))$ ;  $\text{mono } f$  |] ==>  $\text{sup } X (\text{gfp } f) \leq f(\text{sup } X (\text{gfp } f))$ 
apply (frule gfp-lemma2)
apply (drule mono-sup)
apply (rule le-supI)
apply assumption
apply (rule order-trans)
apply (rule order-trans)
apply assumption
apply (rule sup-ge2)
apply assumption
done
```

strong version, thanks to Coen and Frost

```
lemma coinduct-set: [|  $\text{mono}(f)$ ;  $a : X$ ;  $X \subseteq f(X \text{ Un } \text{gfp}(f))$  |] ==>  $a : \text{gfp}(f)$ 
by (blast intro: weak-coinduct [OF coinduct-lemma, simplified sup-set-eq])
```

```
lemma coinduct: [|  $\text{mono}(f)$ ;  $X \leq f(\text{sup } X (\text{gfp } f))$  |] ==>  $X \leq \text{gfp}(f)$ 
apply (rule order-trans)
apply (rule sup-ge1)
apply (erule gfp-upperbound [OF coinduct-lemma])
apply assumption
done
```

```
lemma gfp-fun-UnI2: [|  $\text{mono}(f)$ ;  $a : \text{gfp}(f)$  |] ==>  $a : f(X \text{ Un } \text{gfp}(f))$ 
by (blast dest: gfp-lemma2 mono-Un)
```

## 9.6 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition  $X \subseteq f X$  to one expressed using both *lfp* and *gfp*

```
lemma coinduct3-mono-lemma:  $\text{mono}(f) ==> \text{mono}(\%x. f(x) \text{ Un } X \text{ Un } B)$ 
by (iprover intro: subset-refl monoI Un-mono monoD)
```

```
lemma coinduct3-lemma:
  [|  $X \subseteq f(\text{lfp}(\%x. f(x) \text{ Un } X \text{ Un } \text{gfp}(f)))$ ;  $\text{mono}(f)$  |]
  ==>  $\text{lfp}(\%x. f(x) \text{ Un } X \text{ Un } \text{gfp}(f)) \subseteq f(\text{lfp}(\%x. f(x) \text{ Un } X \text{ Un } \text{gfp}(f)))$ 
apply (rule subset-trans)
```

```

apply (erule coinduct3-mono-lemma [THEN lfp-lemma3])
apply (rule Un-least [THEN Un-least])
apply (rule subset-refl, assumption)
apply (rule gfp-unfold [THEN equalityD1, THEN subset-trans], assumption)
apply (rule monoD [where f=f], assumption)
apply (subst coinduct3-mono-lemma [THEN lfp-unfold], auto)
done

```

```

lemma coinduct3:
  [| mono(f); a:X; X  $\subseteq$  f(lfp(%x. f(x) Un X Un gfp(f))) |] ==> a : gfp(f)
apply (rule coinduct3-lemma [THEN [2] weak-coinduct])
apply (rule coinduct3-mono-lemma [THEN lfp-unfold, THEN ssubst], auto)
done

```

Definition forms of *gfp-unfold* and *coinduct*, to control unfolding

```

lemma def-gfp-unfold: [| A==gfp(f); mono(f) |] ==> A = f(A)
by (auto intro!: gfp-unfold)

```

```

lemma def-coinduct:
  [| A==gfp(f); mono(f); X  $\leq$  f(sup X A) |] ==> X  $\leq$  A
by (iprover intro!: coinduct)

```

```

lemma def-coinduct-set:
  [| A==gfp(f); mono(f); a:X; X  $\subseteq$  f(X Un A) |] ==> a: A
by (auto intro!: coinduct-set)

```

```

lemma def-Collect-coinduct:
  [| A == gfp(%w. Collect(P(w))); mono(%w. Collect(P(w)));
    a: X; !!z. z: X ==> P (X Un A) z |] ==>
    a : A
apply (erule def-coinduct-set, auto)
done

```

```

lemma def-coinduct3:
  [| A==gfp(f); mono(f); a:X; X  $\subseteq$  f(lfp(%x. f(x) Un X Un A)) |] ==> a: A
by (auto intro!: coinduct3)

```

Monotonicity of *gfp*!

```

lemma gfp-mono: (!Z. f Z  $\leq$  g Z) ==> gfp f  $\leq$  gfp g
by (rule gfp-upperbound [THEN gfp-least], blast intro: order-trans)

```

## 9.7 Inductive predicates and sets

Inversion of injective functions.

```

constdefs
  myinv :: ('a ==> 'b) ==> ('b ==> 'a)
  myinv (f :: 'a ==> 'b) ==  $\lambda y.$  THE x. f x = y

```

**lemma** *myinv-f-f*:  $\text{inj } f \implies \text{myinv } f (f x) = x$

**proof** –

**assume** *inj f*

**hence**  $(\text{THE } x'. f x' = f x) = (\text{THE } x'. x' = x)$

**by** (*simp only: inj-eq*)

**also have**  $\dots = x$  **by** (*rule the-eq-trivial*)

**finally show** *?thesis* **by** (*unfold myinv-def*)

**qed**

**lemma** *f-myinv-f*:  $\text{inj } f \implies y \in \text{range } f \implies f (\text{myinv } f y) = y$

**proof** (*unfold myinv-def*)

**assume** *inj: inj f*

**assume**  $y \in \text{range } f$

**then obtain** *x* **where**  $y = f x$  ..

**hence**  $x: f x = y$  ..

**thus**  $f (\text{THE } x. f x = y) = y$

**proof** (*rule theI*)

**fix** *x'* **assume**  $f x' = y$

**with** *x* **have**  $f x' = f x$  **by** *simp*

**with** *inj* **show**  $x' = x$  **by** (*rule injD*)

**qed**

**qed**

**hide** *const myinv*

Package setup.

**theorems** *basic-monos* =

*subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj*

*Collect-mono in-mono vimage-mono*

*imp-conv-disj not-not de-Morgan-disj de-Morgan-conj*

*not-all not-ex*

*Ball-def Bex-def*

*induct-rulify-fallback*

**ML**  $\langle\langle$

*val* *def-lfp-unfold* =  $\text{@}\{\text{thm } \text{def-lfp-unfold}\}$

*val* *def-gfp-unfold* =  $\text{@}\{\text{thm } \text{def-gfp-unfold}\}$

*val* *def-lfp-induct* =  $\text{@}\{\text{thm } \text{def-lfp-induct}\}$

*val* *def-coinduct* =  $\text{@}\{\text{thm } \text{def-coinduct}\}$

*val* *inf-bool-eq* =  $\text{@}\{\text{thm } \text{inf-bool-eq}\} \text{ RS } \text{@}\{\text{thm } \text{eq-reflection}\}$

*val* *inf-fun-eq* =  $\text{@}\{\text{thm } \text{inf-fun-eq}\} \text{ RS } \text{@}\{\text{thm } \text{eq-reflection}\}$

*val* *sup-bool-eq* =  $\text{@}\{\text{thm } \text{sup-bool-eq}\} \text{ RS } \text{@}\{\text{thm } \text{eq-reflection}\}$

*val* *sup-fun-eq* =  $\text{@}\{\text{thm } \text{sup-fun-eq}\} \text{ RS } \text{@}\{\text{thm } \text{eq-reflection}\}$

*val* *le-boolI* =  $\text{@}\{\text{thm } \text{le-boolI}\}$

*val* *le-boolI'* =  $\text{@}\{\text{thm } \text{le-boolI}'\}$

*val* *le-funI* =  $\text{@}\{\text{thm } \text{le-funI}\}$

*val* *le-boolE* =  $\text{@}\{\text{thm } \text{le-boolE}\}$

*val* *le-funE* =  $\text{@}\{\text{thm } \text{le-funE}\}$

```

val le-boolD = @{thm le-boolD}
val le-funD = @{thm le-funD}
val le-bool-def = @{thm le-bool-def} RS @{thm eq-reflection}
val le-fun-def = @{thm le-fun-def} RS @{thm eq-reflection}
>>

```

```

use Tools/inductive-package.ML
setup InductivePackage.setup

```

```

theorems [mono] =
  imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
  imp-conv-disj not-not de-Morgan-disj de-Morgan-conj
  not-all not-ex
  Ball-def Bex-def
  induct-rulify-fallback

```

## 9.8 Inductive datatypes and primitive recursion

Package setup.

```

use Tools/datatype-aux.ML
use Tools/datatype-prop.ML
use Tools/datatype-rep-proofs.ML
use Tools/datatype-abs-proofs.ML
use Tools/datatype-case.ML
use Tools/datatype-package.ML
setup DatatypePackage.setup
use Tools/old-primrec-package.ML
use Tools/primrec-package.ML

```

```

use Tools/datatype-codegen.ML
setup DatatypeCodegen.setup

```

```

use Tools/inductive-codegen.ML
setup InductiveCodegen.setup

```

Lambda-abstractions with pattern matching:

```

syntax
  -lam-pats-syntax :: cases-syn => 'a => 'b          ((%-) 10)
syntax (xsymbols)
  -lam-pats-syntax :: cases-syn => 'a => 'b          ((λ-) 10)

```

**parse-translation (advanced)** <<

```

let
  fun fun-tr ctxt [cs] =
    let
      val x = Free (Name.variant (Term.add-free-names cs []) x, dummyT);
      val ft = DatatypeCase.case-tr true DatatypePackage.datatype-of-constr
        ctxt [x, cs]
    in lambda x ft end

```

```

in [(-lam-pats-syntax, fun-tr)] end
>>

end

```

## 10 OrderedGroup: Ordered Groups

```

theory OrderedGroup
imports Lattices
uses ~~/src/Provers/Arith/abel-cancel.ML
begin

```

The theory of partially ordered groups is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

- <http://www.mathworld.com> by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer.

```

ML<<
structure AlgebraSimps =
  NamedThmsFun(val name = algebra-simps
                val description = algebra simplification rules);
>>

setup AlgebraSimps.setup

```

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

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

### 10.1 Semigroups and Monoids

```

class semigroup-add = plus +
  assumes add-assoc[algebra-simps]: (a + b) + c = a + (b + c)

class ab-semigroup-add = semigroup-add +

```

```

assumes add-commute[algebra-simps]:  $a + b = b + a$ 
begin

lemma add-left-commute[algebra-simps]:  $a + (b + c) = b + (a + c)$ 
by (rule mk-left-commute [of plus, OF add-assoc add-commute])

theorems add-ac = add-assoc add-commute add-left-commute

end

theorems add-ac = add-assoc add-commute add-left-commute

class semigroup-mult = times +
  assumes mult-assoc[algebra-simps]:  $(a * b) * c = a * (b * c)$ 

class ab-semigroup-mult = semigroup-mult +
  assumes mult-commute[algebra-simps]:  $a * b = b * a$ 
begin

lemma mult-left-commute[algebra-simps]:  $a * (b * c) = b * (a * c)$ 
by (rule mk-left-commute [of times, OF mult-assoc mult-commute])

theorems mult-ac = mult-assoc mult-commute mult-left-commute

end

theorems mult-ac = mult-assoc mult-commute mult-left-commute

class ab-semigroup-idem-mult = ab-semigroup-mult +
  assumes mult-idem[simp]:  $x * x = x$ 
begin

lemma mult-left-idem[simp]:  $x * (x * y) = x * y$ 
  unfolding mult-assoc [symmetric, of x] mult-idem ..

end

class monoid-add = zero + semigroup-add +
  assumes add-0-left [simp]:  $0 + a = a$ 
  and add-0-right [simp]:  $a + 0 = a$ 

lemma zero-reorient:  $0 = x \longleftrightarrow x = 0$ 
by (rule eq-commute)

class comm-monoid-add = zero + ab-semigroup-add +
  assumes add-0:  $0 + a = a$ 
begin

subclass monoid-add

```

```

proof qed (insert add-0, simp-all add: add-commute)

end

class monoid-mult = one + semigroup-mult +
  assumes mult-1-left [simp]:  $1 * a = a$ 
  assumes mult-1-right [simp]:  $a * 1 = a$ 

lemma one-reorient:  $1 = x \longleftrightarrow x = 1$ 
by (rule eq-commute)

class comm-monoid-mult = one + ab-semigroup-mult +
  assumes mult-1:  $1 * a = a$ 
begin

subclass monoid-mult
  proof qed (insert mult-1, simp-all add: mult-commute)

end

class cancel-semigroup-add = semigroup-add +
  assumes add-left-imp-eq:  $a + b = a + c \implies b = c$ 
  assumes add-right-imp-eq:  $b + a = c + a \implies b = c$ 
begin

lemma add-left-cancel [simp]:
   $a + b = a + c \longleftrightarrow b = c$ 
by (blast dest: add-left-imp-eq)

lemma add-right-cancel [simp]:
   $b + a = c + a \longleftrightarrow b = c$ 
by (blast dest: add-right-imp-eq)

end

class cancel-ab-semigroup-add = ab-semigroup-add +
  assumes add-imp-eq:  $a + b = a + c \implies b = c$ 
begin

subclass cancel-semigroup-add
proof
  fix a b c :: 'a
  assume a + b = a + c
  then show b = c by (rule add-imp-eq)
next
  fix a b c :: 'a
  assume b + a = c + a
  then have a + b = a + c by (simp only: add-commute)
  then show b = c by (rule add-imp-eq)

```

qed

end

class cancel-comm-monoid-add = cancel-ab-semigroup-add + comm-monoid-add

## 10.2 Groups

class group-add = minus + uminus + monoid-add +  
 assumes left-minus [simp]:  $- a + a = 0$   
 assumes diff-minus:  $a - b = a + (- b)$   
 begin

lemma minus-add-cancel:  $- a + (a + b) = b$   
 by (simp add: add-assoc[symmetric])

lemma minus-zero [simp]:  $- 0 = 0$   
 proof -  
 have  $- 0 = - 0 + (0 + 0)$  by (simp only: add-0-right)  
 also have  $\dots = 0$  by (rule minus-add-cancel)  
 finally show ?thesis .  
 qed

lemma minus-minus [simp]:  $- (- a) = a$   
 proof -  
 have  $- (- a) = - (- a) + (- a + a)$  by simp  
 also have  $\dots = a$  by (rule minus-add-cancel)  
 finally show ?thesis .  
 qed

lemma right-minus [simp]:  $a + - a = 0$   
 proof -  
 have  $a + - a = - (- a) + - a$  by simp  
 also have  $\dots = 0$  by (rule left-minus)  
 finally show ?thesis .  
 qed

lemma right-minus-eq:  $a - b = 0 \longleftrightarrow a = b$   
 proof  
 assume  $a - b = 0$   
 have  $a = (a - b) + b$  by (simp add: diff-minus add-assoc)  
 also have  $\dots = b$  using  $\langle a - b = 0 \rangle$  by simp  
 finally show  $a = b$  .  
 next  
 assume  $a = b$  thus  $a - b = 0$  by (simp add: diff-minus)  
 qed

lemma equals-zero-I:  
 assumes  $a + b = 0$  shows  $- a = b$



**proof** –  
 have  $- a = - a + (a + b)$  **using** *assms* **by** *simp*  
 also have  $\dots = b$  **by** (*simp add: add-associ[commutative]*)  
 finally **show** *?thesis* .  
**qed**

**lemma** *diff-self* [*simp*]:  $a - a = 0$   
**by** (*simp add: diff-minus*)

**lemma** *diff-0* [*simp*]:  $0 - a = - a$   
**by** (*simp add: diff-minus*)

**lemma** *diff-0-right* [*simp*]:  $a - 0 = a$   
**by** (*simp add: diff-minus*)

**lemma** *diff-minus-eq-add* [*simp*]:  $a - - b = a + b$   
**by** (*simp add: diff-minus*)

**lemma** *neg-equal-iff-equal* [*simp*]:  
 $- a = - b \longleftrightarrow a = b$

**proof**  
 assume  $- a = - b$   
 hence  $- (- a) = - (- b)$  **by** *simp*  
 thus  $a = b$  **by** *simp*  
**next**  
 assume  $a = b$   
 thus  $- a = - b$  **by** *simp*  
**qed**

**lemma** *neg-equal-0-iff-equal* [*simp*]:  
 $- a = 0 \longleftrightarrow a = 0$   
**by** (*subst neg-equal-iff-equal [symmetric], simp*)

**lemma** *neg-0-equal-iff-equal* [*simp*]:  
 $0 = - a \longleftrightarrow 0 = a$   
**by** (*subst neg-equal-iff-equal [symmetric], simp*)

The next two equations can make the simplifier loop!

**lemma** *equation-minus-iff*:  
 $a = - b \longleftrightarrow b = - a$   
**proof** –  
 have  $- (- a) = - b \longleftrightarrow - a = b$  **by** (*rule neg-equal-iff-equal*)  
 thus *?thesis* **by** (*simp add: eq-commute*)  
**qed**

**lemma** *minus-equation-iff*:  
 $- a = b \longleftrightarrow - b = a$   
**proof** –  
 have  $- a = - (- b) \longleftrightarrow a = - b$  **by** (*rule neg-equal-iff-equal*)

```

    thus ?thesis by (simp add: eq-commute)
qed

lemma diff-add-cancel:  $a - b + b = a$ 
by (simp add: diff-minus add-assoc)

lemma add-diff-cancel:  $a + b - b = a$ 
by (simp add: diff-minus add-assoc)

declare diff-minus[symmetric, algebra-simps]

lemma eq-neg-iff-add-eq-0:  $a = -b \longleftrightarrow a + b = 0$ 
proof
  assume  $a = -b$  then show  $a + b = 0$  by simp
next
  assume  $a + b = 0$ 
  moreover have  $a + (b + -b) = (a + b) + -b$ 
    by (simp only: add-assoc)
  ultimately show  $a = -b$  by simp
qed

end

class ab-group-add = minus + uminus + comm-monoid-add +
  assumes ab-left-minus:  $-a + a = 0$ 
  assumes ab-diff-minus:  $a - b = a + (-b)$ 
begin

subclass group-add
  proof qed (simp-all add: ab-left-minus ab-diff-minus)

subclass cancel-comm-monoid-add
proof
  fix  $a\ b\ c :: 'a$ 
  assume  $a + b = a + c$ 
  then have  $-a + a + b = -a + a + c$ 
    unfolding add-assoc by simp
  then show  $b = c$  by simp
qed

lemma uminus-add-conv-diff[algebra-simps]:
   $-a + b = b - a$ 
by (simp add: diff-minus add-commute)

lemma minus-add-distrib [simp]:
   $-(a + b) = -a + -b$ 
by (rule equals-zero-I) (simp add: add-ac)

lemma minus-diff-eq [simp]:

```

```

    - (a - b) = b - a
  by (simp add: diff-minus add-commute)

lemma add-diff-eq[algebra-simps]: a + (b - c) = (a + b) - c
by (simp add: diff-minus add-ac)

lemma diff-add-eq[algebra-simps]: (a - b) + c = (a + c) - b
by (simp add: diff-minus add-ac)

lemma diff-eq-eq[algebra-simps]: a - b = c  $\longleftrightarrow$  a = c + b
by (auto simp add: diff-minus add-assoc)

lemma eq-diff-eq[algebra-simps]: a = c - b  $\longleftrightarrow$  a + b = c
by (auto simp add: diff-minus add-assoc)

lemma diff-diff-eq[algebra-simps]: (a - b) - c = a - (b + c)
by (simp add: diff-minus add-ac)

lemma diff-diff-eq2[algebra-simps]: a - (b - c) = (a + c) - b
by (simp add: diff-minus add-ac)

lemma eq-iff-diff-eq-0: a = b  $\longleftrightarrow$  a - b = 0
by (simp add: algebra-simps)

lemma diff-eq-0-iff-eq [simp, noatp]: a - b = 0  $\longleftrightarrow$  a = b
by (simp add: algebra-simps)

end

```

### 10.3 (Partially) Ordered Groups

```

class pordered-ab-semigroup-add = order + ab-semigroup-add +
  assumes add-left-mono: a ≤ b  $\implies$  c + a ≤ c + b
begin

lemma add-right-mono:
  a ≤ b  $\implies$  a + c ≤ b + c
by (simp add: add-commute [of - c] add-left-mono)

non-strict, in both arguments

lemma add-mono:
  a ≤ b  $\implies$  c ≤ d  $\implies$  a + c ≤ b + d
  apply (erule add-right-mono [THEN order-trans])
  apply (simp add: add-commute add-left-mono)
  done

end

class pordered-cancel-ab-semigroup-add =

```

*pordered-ab-semigroup-add* + *cancel-ab-semigroup-add*  
**begin**

**lemma** *add-strict-left-mono*:  
 $a < b \implies c + a < c + b$   
**by** (*auto simp add: less-le add-left-mono*)

**lemma** *add-strict-right-mono*:  
 $a < b \implies a + c < b + c$   
**by** (*simp add: add-commute [of - c] add-strict-left-mono*)

Strict monotonicity in both arguments

**lemma** *add-strict-mono*:  
 $a < b \implies c < d \implies a + c < b + d$   
**apply** (*erule add-strict-right-mono [THEN less-trans]*)  
**apply** (*erule add-strict-left-mono*)  
**done**

**lemma** *add-less-le-mono*:  
 $a < b \implies c \leq d \implies a + c < b + d$   
**apply** (*erule add-strict-right-mono [THEN less-le-trans]*)  
**apply** (*erule add-left-mono*)  
**done**

**lemma** *add-le-less-mono*:  
 $a \leq b \implies c < d \implies a + c < b + d$   
**apply** (*erule add-right-mono [THEN le-less-trans]*)  
**apply** (*erule add-strict-left-mono*)  
**done**

**end**

**class** *pordered-ab-semigroup-add-imp-le* =  
*pordered-cancel-ab-semigroup-add* +  
**assumes** *add-le-imp-le-left*:  $c + a \leq c + b \implies a \leq b$   
**begin**

**lemma** *add-less-imp-less-left*:  
**assumes** *less*:  $c + a < c + b$  **shows**  $a < b$   
**proof** –  
**from** *less* **have** *le*:  $c + a \leq c + b$  **by** (*simp add: order-le-less*)  
**have**  $a \leq b$   
**apply** (*insert le*)  
**apply** (*drule add-le-imp-le-left*)  
**by** (*insert le, drule add-le-imp-le-left, assumption*)  
**moreover** **have**  $a \neq b$   
**proof** (*rule ccontr*)  
**assume**  $\sim(a \neq b)$   
**then** **have**  $a = b$  **by** *simp*

```

    then have  $c + a = c + b$  by simp
    with less show False by simp
  qed
  ultimately show  $a < b$  by (simp add: order-le-less)
qed

```

```

lemma add-less-imp-less-right:
   $a + c < b + c \implies a < b$ 
apply (rule add-less-imp-less-left [of c])
apply (simp add: add-commute)
done

```

```

lemma add-less-cancel-left [simp]:
   $c + a < c + b \iff a < b$ 
by (blast intro: add-less-imp-less-left add-strict-left-mono)

```

```

lemma add-less-cancel-right [simp]:
   $a + c < b + c \iff a < b$ 
by (blast intro: add-less-imp-less-right add-strict-right-mono)

```

```

lemma add-le-cancel-left [simp]:
   $c + a \leq c + b \iff a \leq b$ 
by (auto, drule add-le-imp-le-left, simp-all add: add-left-mono)

```

```

lemma add-le-cancel-right [simp]:
   $a + c \leq b + c \iff a \leq b$ 
by (simp add: add-commute [of a c] add-commute [of b c])

```

```

lemma add-le-imp-le-right:
   $a + c \leq b + c \implies a \leq b$ 
by simp

```

```

lemma max-add-distrib-left:
   $\max x y + z = \max (x + z) (y + z)$ 
unfolding max-def by auto

```

```

lemma min-add-distrib-left:
   $\min x y + z = \min (x + z) (y + z)$ 
unfolding min-def by auto

```

```

end

```

## 10.4 Support for reasoning about signs

```

class pordered-comm-monoid-add =
  pordered-cancel-ab-semigroup-add + comm-monoid-add
begin

```

```

lemma add-pos-nonneg:

```

```

    assumes  $0 < a$  and  $0 \leq b$  shows  $0 < a + b$ 
  proof -
    have  $0 + 0 < a + b$ 
      using assms by (rule add-less-le-mono)
    then show ?thesis by simp
  qed

```

```

lemma add-pos-pos:
  assumes  $0 < a$  and  $0 < b$  shows  $0 < a + b$ 
  by (rule add-pos-nonneg) (insert assms, auto)

```

```

lemma add-nonneg-pos:
  assumes  $0 \leq a$  and  $0 < b$  shows  $0 < a + b$ 
  proof -
    have  $0 + 0 < a + b$ 
      using assms by (rule add-le-less-mono)
    then show ?thesis by simp
  qed

```

```

lemma add-nonneg-nonneg:
  assumes  $0 \leq a$  and  $0 \leq b$  shows  $0 \leq a + b$ 
  proof -
    have  $0 + 0 \leq a + b$ 
      using assms by (rule add-mono)
    then show ?thesis by simp
  qed

```

```

lemma add-neg-nonpos:
  assumes  $a < 0$  and  $b \leq 0$  shows  $a + b < 0$ 
  proof -
    have  $a + b < 0 + 0$ 
      using assms by (rule add-less-le-mono)
    then show ?thesis by simp
  qed

```

```

lemma add-neg-neg:
  assumes  $a < 0$  and  $b < 0$  shows  $a + b < 0$ 
  by (rule add-neg-nonpos) (insert assms, auto)

```

```

lemma add-nonpos-neg:
  assumes  $a \leq 0$  and  $b < 0$  shows  $a + b < 0$ 
  proof -
    have  $a + b < 0 + 0$ 
      using assms by (rule add-le-less-mono)
    then show ?thesis by simp
  qed

```

```

lemma add-nonpos-nonpos:
  assumes  $a \leq 0$  and  $b \leq 0$  shows  $a + b \leq 0$ 

```

```

proof –
  have  $a + b \leq 0 + 0$ 
    using assms by (rule add-mono)
  then show ?thesis by simp
qed

lemmas add-sign-intros =
  add-pos-nonneg add-pos-pos add-nonneg-pos add-nonneg-nonneg
  add-neg-nonpos add-neg-neg add-nonpos-neg add-nonpos-nonpos

lemma add-nonneg-eq-0-iff:
  assumes  $x: 0 \leq x$  and  $y: 0 \leq y$ 
  shows  $x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$ 
proof (intro iffI conjI)
  have  $x = x + 0$  by simp
  also have  $x + 0 \leq x + y$  using  $y$  by (rule add-left-mono)
  also assume  $x + y = 0$ 
  also have  $0 \leq x$  using  $x$  .
  finally show  $x = 0$  .
next
  have  $y = 0 + y$  by simp
  also have  $0 + y \leq x + y$  using  $x$  by (rule add-right-mono)
  also assume  $x + y = 0$ 
  also have  $0 \leq y$  using  $y$  .
  finally show  $y = 0$  .
next
  assume  $x = 0 \wedge y = 0$ 
  then show  $x + y = 0$  by simp
qed

end

class pordered-ab-group-add =
  ab-group-add + pordered-ab-semigroup-add
begin

subclass pordered-cancel-ab-semigroup-add ..

subclass pordered-ab-semigroup-add-imp-le
proof
  fix  $a\ b\ c :: 'a$ 
  assume  $c + a \leq c + b$ 
  hence  $(-c) + (c + a) \leq (-c) + (c + b)$  by (rule add-left-mono)
  hence  $((-c) + c) + a \leq ((-c) + c) + b$  by (simp only: add-assoc)
  thus  $a \leq b$  by simp
qed

subclass pordered-comm-monoid-add ..

```

**lemma** *max-diff-distrib-left*:

**shows**  $\max x y - z = \max (x - z) (y - z)$

**by** (*simp add: diff-minus, rule max-add-distrib-left*)

**lemma** *min-diff-distrib-left*:

**shows**  $\min x y - z = \min (x - z) (y - z)$

**by** (*simp add: diff-minus, rule min-add-distrib-left*)

**lemma** *le-imp-neg-le*:

**assumes**  $a \leq b$  **shows**  $-b \leq -a$

**proof** –

**have**  $-a + a \leq -a + b$  **using**  $\langle a \leq b \rangle$  **by** (*rule add-left-mono*)

**hence**  $0 \leq -a + b$  **by** *simp*

**hence**  $0 + (-b) \leq (-a + b) + (-b)$  **by** (*rule add-right-mono*)

**thus** *?thesis* **by** (*simp add: add-assoc*)

**qed**

**lemma** *neg-le-iff-le* [*simp*]:  $-b \leq -a \longleftrightarrow a \leq b$

**proof**

**assume**  $-b \leq -a$

**hence**  $-(-a) \leq -(-b)$  **by** (*rule le-imp-neg-le*)

**thus**  $a \leq b$  **by** *simp*

**next**

**assume**  $a \leq b$

**thus**  $-b \leq -a$  **by** (*rule le-imp-neg-le*)

**qed**

**lemma** *neg-le-0-iff-le* [*simp*]:  $-a \leq 0 \longleftrightarrow 0 \leq a$

**by** (*subst neg-le-iff-le [symmetric], simp*)

**lemma** *neg-0-le-iff-le* [*simp*]:  $0 \leq -a \longleftrightarrow a \leq 0$

**by** (*subst neg-le-iff-le [symmetric], simp*)

**lemma** *neg-less-iff-less* [*simp*]:  $-b < -a \longleftrightarrow a < b$

**by** (*force simp add: less-le*)

**lemma** *neg-less-0-iff-less* [*simp*]:  $-a < 0 \longleftrightarrow 0 < a$

**by** (*subst neg-less-iff-less [symmetric], simp*)

**lemma** *neg-0-less-iff-less* [*simp*]:  $0 < -a \longleftrightarrow a < 0$

**by** (*subst neg-less-iff-less [symmetric], simp*)

The next several equations can make the simplifier loop!

**lemma** *less-minus-iff*:  $a < -b \longleftrightarrow b < -a$

**proof** –

**have**  $(-(-a) < -b) = (b < -a)$  **by** (*rule neg-less-iff-less*)

**thus** *?thesis* **by** *simp*

**qed**



**lemma** *minus-less-iff*:  $- a < b \longleftrightarrow - b < a$

**proof** –

have  $(- a < - (-b)) = (- b < a)$  **by** (*rule neg-less-iff-less*)

thus *?thesis* **by** *simp*

**qed**

**lemma** *le-minus-iff*:  $a \leq - b \longleftrightarrow b \leq - a$

**proof** –

have *mm*:  $!! a (b::'a). (-(-a)) < -b \implies -(-b) < -a$  **by** (*simp only: minus-less-iff*)

have  $(- (- a) \leq -b) = (b \leq - a)$

apply (*auto simp only: le-less*)

apply (*drule mm*)

apply (*simp-all*)

apply (*drule mm[simplified], assumption*)

done

then show *?thesis* **by** *simp*

**qed**

**lemma** *minus-le-iff*:  $- a \leq b \longleftrightarrow - b \leq a$

**by** (*auto simp add: le-less minus-less-iff*)

**lemma** *less-iff-diff-less-0*:  $a < b \longleftrightarrow a - b < 0$

**proof** –

have  $(a < b) = (a + (- b) < b + (-b))$

by (*simp only: add-less-cancel-right*)

also have  $\dots = (a - b < 0)$  **by** (*simp add: diff-minus*)

finally show *?thesis* .

**qed**

**lemma** *diff-less-eq[algebra-simps]*:  $a - b < c \longleftrightarrow a < c + b$

apply (*subst less-iff-diff-less-0 [of a]*)

apply (*rule less-iff-diff-less-0 [of - c, THEN ssubst]*)

apply (*simp add: diff-minus add-ac*)

done

**lemma** *less-diff-eq[algebra-simps]*:  $a < c - b \longleftrightarrow a + b < c$

apply (*subst less-iff-diff-less-0 [of plus a b]*)

apply (*subst less-iff-diff-less-0 [of a]*)

apply (*simp add: diff-minus add-ac*)

done

**lemma** *diff-le-eq[algebra-simps]*:  $a - b \leq c \longleftrightarrow a \leq c + b$

**by** (*auto simp add: le-less diff-less-eq diff-add-cancel add-diff-cancel*)

**lemma** *le-diff-eq[algebra-simps]*:  $a \leq c - b \longleftrightarrow a + b \leq c$

**by** (*auto simp add: le-less less-diff-eq diff-add-cancel add-diff-cancel*)

**lemma** *le-iff-diff-le-0*:  $a \leq b \longleftrightarrow a - b \leq 0$

**by** (*simp add: algebra-simps*)

Legacy - use *algebra-simps*

**lemmas** *group-simps*[*noatp*] = *algebra-simps*

**end**

Legacy - use *algebra-simps*

**lemmas** *group-simps*[*noatp*] = *algebra-simps*

**class** *ordered-ab-semigroup-add* =  
     *linorder* + *pordered-ab-semigroup-add*

**class** *ordered-cancel-ab-semigroup-add* =  
     *linorder* + *pordered-cancel-ab-semigroup-add*  
**begin**

**subclass** *ordered-ab-semigroup-add* ..

**subclass** *pordered-ab-semigroup-add-imp-le*  
**proof**

**fix** *a b c* :: '*a*

**assume** *le*:  $c + a \leq c + b$

**show**  $a \leq b$

**proof** (*rule ccontr*)

**assume** *w*:  $\sim a \leq b$

**hence**  $b \leq a$  **by** (*simp add: linorder-not-le*)

**hence** *le2*:  $c + b \leq c + a$  **by** (*rule add-left-mono*)

**have**  $a = b$

**apply** (*insert le*)

**apply** (*insert le2*)

**apply** (*drule antisym, simp-all*)

**done**

**with** *w* **show** *False*

**by** (*simp add: linorder-not-le [symmetric]*)

**qed**

**qed**

**end**

**class** *ordered-ab-group-add* =  
     *linorder* + *pordered-ab-group-add*  
**begin**

**subclass** *ordered-cancel-ab-semigroup-add* ..

**lemma** *neg-less-eq-nonneg*:

$-a \leq a \iff 0 \leq a$

**proof**

**assume** *A*:  $-a \leq a$  **show**  $0 \leq a$

**proof** (*rule classical*)

```

    assume  $\neg 0 \leq a$ 
    then have  $a < 0$  by auto
    with A have  $-a < 0$  by (rule le-less-trans)
    then show ?thesis by auto
  qed
next
  assume A:  $0 \leq a$  show  $-a \leq a$ 
  proof (rule order-trans)
    show  $-a \leq 0$  using A by (simp add: minus-le-iff)
  next
    show  $0 \leq a$  using A .
  qed
qed

```

```

lemma less-eq-neg-nonpos:
   $a \leq -a \longleftrightarrow a \leq 0$ 
proof
  assume A:  $a \leq -a$  show  $a \leq 0$ 
  proof (rule classical)
    assume  $\neg a \leq 0$ 
    then have  $0 < a$  by auto
    then have  $0 < -a$  using A by (rule less-le-trans)
    then show ?thesis by auto
  qed
next
  assume A:  $a \leq 0$  show  $a \leq -a$ 
  proof (rule order-trans)
    show  $0 \leq -a$  using A by (simp add: minus-le-iff)
  next
    show  $a \leq 0$  using A .
  qed
qed

```

```

lemma equal-neg-zero:
   $a = -a \longleftrightarrow a = 0$ 
proof
  assume  $a = 0$  then show  $a = -a$  by simp
next
  assume A:  $a = -a$  show  $a = 0$ 
  proof (cases  $0 \leq a$ )
    case True with A have  $0 \leq -a$  by auto
    with le-minus-iff have  $a \leq 0$  by simp
    with True show ?thesis by (auto intro: order-trans)
  next
    case False then have B:  $a \leq 0$  by auto
    with A have  $-a \leq 0$  by auto
    with B show ?thesis by (auto intro: order-trans)
  qed
qed

```

```

lemma neg-equal-zero:
  -  $a = a \longleftrightarrow a = 0$ 
  unfolding equal-neg-zero [symmetric] by auto

end

— FIXME localize the following

lemma add-increasing:
  fixes  $c :: 'a::\{pordered-ab-semigroup-add-imp-le, comm-monoid-add\}$ 
  shows  $[|0 \leq a; b \leq c|] \implies b \leq a + c$ 
by (insert add-mono [of 0 a b c], simp)

lemma add-increasing2:
  fixes  $c :: 'a::\{pordered-ab-semigroup-add-imp-le, comm-monoid-add\}$ 
  shows  $[|0 \leq c; b \leq a|] \implies b \leq a + c$ 
by (simp add:add-increasing add-commute[of a])

lemma add-strict-increasing:
  fixes  $c :: 'a::\{pordered-ab-semigroup-add-imp-le, comm-monoid-add\}$ 
  shows  $[|0 < a; b \leq c|] \implies b < a + c$ 
by (insert add-less-le-mono [of 0 a b c], simp)

lemma add-strict-increasing2:
  fixes  $c :: 'a::\{pordered-ab-semigroup-add-imp-le, comm-monoid-add\}$ 
  shows  $[|0 \leq a; b < c|] \implies b < a + c$ 
by (insert add-le-less-mono [of 0 a b c], simp)

class pordered-ab-group-add-abs = pordered-ab-group-add + abs +
  assumes abs-ge-zero [simp]:  $|a| \geq 0$ 
  and abs-ge-self:  $a \leq |a|$ 
  and abs-leI:  $a \leq b \implies -a \leq b \implies |a| \leq b$ 
  and abs-minus-cancel [simp]:  $|-a| = |a|$ 
  and abs-triangle-ineq:  $|a + b| \leq |a| + |b|$ 
begin

lemma abs-minus-le-zero:  $-|a| \leq 0$ 
  unfolding neg-le-0-iff-le by simp

lemma abs-of-nonneg [simp]:
  assumes nonneg:  $0 \leq a$  shows  $|a| = a$ 
proof (rule antisym)
  from nonneg le-imp-neg-le have  $-a \leq 0$  by simp
  from this nonneg have  $-a \leq a$  by (rule order-trans)
  then show  $|a| \leq a$  by (auto intro: abs-leI)
qed (rule abs-ge-self)

```

**lemma** *abs-idempotent* [*simp*]:  $||a|| = |a|$   
**by** (*rule antisym*)  
 (*auto intro!*: *abs-ge-self abs-leI order-trans [of uminus (abs a) zero abs a]*)

**lemma** *abs-eq-0* [*simp*]:  $|a| = 0 \longleftrightarrow a = 0$   
**proof** –  
**have**  $|a| = 0 \implies a = 0$   
**proof** (*rule antisym*)  
**assume** *zero*:  $|a| = 0$   
**with** *abs-ge-self* **show**  $a \leq 0$  **by** *auto*  
**from** *zero* **have**  $|-a| = 0$  **by** *simp*  
**with** *abs-ge-self [of uminus a]* **have**  $-a \leq 0$  **by** *auto*  
**with** *neg-le-0-iff-le* **show**  $0 \leq a$  **by** *auto*  
**qed**  
**then show** *?thesis* **by** *auto*  
**qed**

**lemma** *abs-zero* [*simp*]:  $|0| = 0$   
**by** *simp*

**lemma** *abs-0-eq* [*simp*, *noatp*]:  $0 = |a| \longleftrightarrow a = 0$   
**proof** –  
**have**  $0 = |a| \longleftrightarrow |a| = 0$  **by** (*simp only: eq-ac*)  
**thus** *?thesis* **by** *simp*  
**qed**

**lemma** *abs-le-zero-iff* [*simp*]:  $|a| \leq 0 \longleftrightarrow a = 0$   
**proof**  
**assume**  $|a| \leq 0$   
**then have**  $|a| = 0$  **by** (*rule antisym*) *simp*  
**thus**  $a = 0$  **by** *simp*  
**next**  
**assume**  $a = 0$   
**thus**  $|a| \leq 0$  **by** *simp*  
**qed**

**lemma** *zero-less-abs-iff* [*simp*]:  $0 < |a| \longleftrightarrow a \neq 0$   
**by** (*simp add: less-le*)

**lemma** *abs-not-less-zero* [*simp*]:  $\neg |a| < 0$   
**proof** –  
**have**  $a: \bigwedge x y. x \leq y \implies \neg y < x$  **by** *auto*  
**show** *?thesis* **by** (*simp add: a*)  
**qed**

**lemma** *abs-ge-minus-self*:  $-a \leq |a|$   
**proof** –  
**have**  $-a \leq |-a|$  **by** (*rule abs-ge-self*)  
**then show** *?thesis* **by** *simp*

qed

**lemma** *abs-minus-commute*:

$$|a - b| = |b - a|$$

**proof** -

**have**  $|a - b| = |-(a - b)|$  **by** (*simp only: abs-minus-cancel*)

**also have**  $\dots = |b - a|$  **by** *simp*

**finally show** *?thesis* .

qed

**lemma** *abs-of-pos*:  $0 < a \implies |a| = a$

**by** (*rule abs-of-nonneg, rule less-imp-le*)

**lemma** *abs-of-nonpos* [*simp*]:

**assumes**  $a \leq 0$  **shows**  $|a| = -a$

**proof** -

**let**  $?b = -a$

**have**  $-?b \leq 0 \implies |-?b| = -(-?b)$

**unfolding** *abs-minus-cancel* [*of ?b*]

**unfolding** *neg-le-0-iff-le* [*of ?b*]

**unfolding** *minus-minus* **by** (*erule abs-of-nonneg*)

**then show** *?thesis* **using** *assms* **by** *auto*

qed

**lemma** *abs-of-neg*:  $a < 0 \implies |a| = -a$

**by** (*rule abs-of-nonpos, rule less-imp-le*)

**lemma** *abs-le-D1*:  $|a| \leq b \implies a \leq b$

**by** (*insert abs-ge-self, blast intro: order-trans*)

**lemma** *abs-le-D2*:  $|a| \leq b \implies -a \leq b$

**by** (*insert abs-le-D1 [of uminus a], simp*)

**lemma** *abs-le-iff*:  $|a| \leq b \longleftrightarrow a \leq b \wedge -a \leq b$

**by** (*blast intro: abs-leI dest: abs-le-D1 abs-le-D2*)

**lemma** *abs-triangle-ineq2*:  $|a| - |b| \leq |a - b|$

**apply** (*simp add: algebra-simps*)

**apply** (*subgoal-tac abs a = abs (plus b (minus a b))*)

**apply** (*erule ssubst*)

**apply** (*rule abs-triangle-ineq*)

**apply** (*rule arg-cong[of - - abs]*)

**apply** (*simp add: algebra-simps*)

done

**lemma** *abs-triangle-ineq3*:  $||a| - |b|| \leq |a - b|$

**apply** (*subst abs-le-iff*)

**apply** *auto*

**apply** (*rule abs-triangle-ineq2*)

```

    apply (subst abs-minus-commute)
    apply (rule abs-triangle-ineq2)
done

```

```

lemma abs-triangle-ineq4:  $|a - b| \leq |a| + |b|$ 
proof -
  have  $\text{abs}(a - b) = \text{abs}(a + - b)$  by (subst diff-minus, rule refl)
  also have  $\dots \leq \text{abs } a + \text{abs } (- b)$  by (rule abs-triangle-ineq)
  finally show ?thesis by simp
qed

```

```

lemma abs-diff-triangle-ineq:  $|a + b - (c + d)| \leq |a - c| + |b - d|$ 
proof -
  have  $|a + b - (c + d)| = |(a - c) + (b - d)|$  by (simp add: diff-minus add-ac)
  also have  $\dots \leq |a - c| + |b - d|$  by (rule abs-triangle-ineq)
  finally show ?thesis .
qed

```

```

lemma abs-add-abs [simp]:
   $||a| + |b|| = |a| + |b|$  (is ?L = ?R)
proof (rule antisym)
  show  $?L \geq ?R$  by (rule abs-ge-self)
next
  have  $?L \leq ||a| + |b||$  by (rule abs-triangle-ineq)
  also have  $\dots = ?R$  by simp
  finally show  $?L \leq ?R$  .
qed

```

```
end
```

## 10.5 Lattice Ordered (Abelian) Groups

```

class lordered-ab-group-add-meet = pordered-ab-group-add + lower-semilattice
begin

```

```

lemma add-inf-distrib-left:
   $a + \inf b c = \inf (a + b) (a + c)$ 
apply (rule antisym)
apply (simp-all add: le-infI)
apply (rule add-le-imp-le-left [of uminus a])
apply (simp only: add-assoc [symmetric], simp)
apply rule
apply (rule add-le-imp-le-left [of a], simp only: add-assoc [symmetric], simp) +
done

```

```

lemma add-inf-distrib-right:
   $\inf a b + c = \inf (a + c) (b + c)$ 
proof -
  have  $c + \inf a b = \inf (c + a) (c + b)$  by (simp add: add-inf-distrib-left)

```

```

    thus ?thesis by (simp add: add-commute)
qed

end

class lordered-ab-group-add-join = pordered-ab-group-add + upper-semilattice
begin

lemma add-sup-distrib-left:
   $a + \sup b\ c = \sup (a + b)\ (a + c)$ 
  apply (rule antisym)
  apply (rule add-le-imp-le-left [of uminus a])
  apply (simp only: add-assoc[symmetric], simp)
  apply rule
  apply (rule add-le-imp-le-left [of a], simp only: add-assoc[symmetric], simp)+
  apply (rule le-supI)
  apply (simp-all)
done

lemma add-sup-distrib-right:
   $\sup a\ b + c = \sup (a+c)\ (b+c)$ 
proof -
  have  $c + \sup a\ b = \sup (c+a)\ (c+b)$  by (simp add: add-sup-distrib-left)
  thus ?thesis by (simp add: add-commute)
qed

end

class lordered-ab-group-add = pordered-ab-group-add + lattice
begin

subclass lordered-ab-group-add-meet ..
subclass lordered-ab-group-add-join ..

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 (rule inf-unique)
  fix a b :: 'a
  show  $-\sup (-a)\ (-b) \leq a$ 
    by (rule add-le-imp-le-right [of - sup (uminus a) (uminus b)])
      (simp, simp add: add-sup-distrib-left)
next
  fix a b :: 'a
  show  $-\sup (-a)\ (-b) \leq b$ 
    by (rule add-le-imp-le-right [of - sup (uminus a) (uminus b)])
      (simp, simp add: add-sup-distrib-left)
next

```



```

fix a b c :: 'a
assume a ≤ b a ≤ c
then show a ≤ - sup (-b) (-c) by (subst neg-le-iff-le [symmetric])
  (simp add: le-supI)
qed

```

```

lemma sup-eq-neg-inf: sup a b = - inf (-a) (-b)
proof (rule sup-unique)
  fix a b :: 'a
  show a ≤ - inf (-a) (-b)
    by (rule add-le-imp-le-right [of - inf (uminus a) (uminus b)])
      (simp, simp add: add-inf-distrib-left)
next
  fix a b :: 'a
  show b ≤ - inf (-a) (-b)
    by (rule add-le-imp-le-right [of - inf (uminus a) (uminus b)])
      (simp, simp add: add-inf-distrib-left)
next
  fix a b c :: 'a
  assume a ≤ c b ≤ c
  then show - inf (-a) (-b) ≤ c by (subst neg-le-iff-le [symmetric])
    (simp add: le-infI)
qed

```

```

lemma neg-inf-eq-sup: - inf a b = sup (-a) (-b)
by (simp add: inf-eq-neg-sup)

```

```

lemma neg-sup-eq-inf: - sup a b = inf (-a) (-b)
by (simp add: sup-eq-neg-inf)

```

```

lemma add-eq-inf-sup: a + b = sup a b + inf a b
proof -
  have 0 = - inf 0 (a-b) + inf (a-b) 0 by (simp add: inf-commute)
  hence 0 = sup 0 (b-a) + inf (a-b) 0 by (simp add: inf-eq-neg-sup)
  hence 0 = (-a + sup a b) + (inf a b + (-b))
    by (simp add: add-sup-distrib-left add-inf-distrib-right)
      (simp add: algebra-simps)
  thus ?thesis by (simp add: algebra-simps)
qed

```

## 10.6 Positive Part, Negative Part, Absolute Value

### definition

```

nprt :: 'a ⇒ 'a where
nprt x = inf x 0

```

### definition

```

pprt :: 'a ⇒ 'a where
pprt x = sup x 0

```

**lemma** *pprt-neg*:  $pprt (- x) = - nprt x$   
**proof** –  
 have  $sup (- x) 0 = sup (- x) (- 0)$  **unfolding** *minus-zero* ..  
 also have  $\dots = - inf x 0$  **unfolding** *neg-inf-eq-sup* ..  
 finally have  $sup (- x) 0 = - inf x 0$  .  
 then show *?thesis* **unfolding** *pprt-def nprt-def* .  
**qed**

**lemma** *nprt-neg*:  $nprt (- x) = - pprt x$   
**proof** –  
 from *pprt-neg* have  $pprt (- (- x)) = - nprt (- x)$  .  
 then have  $pprt x = - nprt (- x)$  **by** *simp*  
 then show *?thesis* **by** *simp*  
**qed**

**lemma** *prts*:  $a = pprt a + nprt a$   
**by** (*simp add: pprt-def nprt-def add-eq-inf-sup[symmetric]*)

**lemma** *zero-le-pprt*[*simp*]:  $0 \leq pprt a$   
**by** (*simp add: pprt-def*)

**lemma** *nprt-le-zero*[*simp*]:  $nprt a \leq 0$   
**by** (*simp add: nprt-def*)

**lemma** *le-eq-neg*:  $a \leq - b \longleftrightarrow a + b \leq 0$  (**is** *?l* = *?r*)  
**proof** –  
 have  $a: ?l \longrightarrow ?r$   
 apply (*auto*)  
 apply (*rule add-le-imp-le-right[of - uminus b -]*)  
 apply (*simp add: add-assoc*)  
 done  
 have  $b: ?r \longrightarrow ?l$   
 apply (*auto*)  
 apply (*rule add-le-imp-le-right[of - b -]*)  
 apply (*simp*)  
 done  
 from  $a b$  show *?thesis* **by** *blast*  
**qed**

**lemma** *pprt-0*[*simp*]:  $pprt 0 = 0$  **by** (*simp add: pprt-def*)  
**lemma** *nprt-0*[*simp*]:  $nprt 0 = 0$  **by** (*simp add: nprt-def*)

**lemma** *pprt-eq-id* [*simp, noatp*]:  $0 \leq x \implies pprt x = x$   
**by** (*simp add: pprt-def le-iff-sup sup-ACI*)

**lemma** *nprt-eq-id* [*simp, noatp*]:  $x \leq 0 \implies nprt x = x$   
**by** (*simp add: nprt-def le-iff-inf inf-ACI*)

**lemma** *pprt-eq-0* [*simp*, *noatp*]:  $x \leq 0 \implies \text{pprt } x = 0$   
**by** (*simp add: pprt-def le-iff-sup sup-ACI*)

**lemma** *nprrt-eq-0* [*simp*, *noatp*]:  $0 \leq x \implies \text{nprrt } x = 0$   
**by** (*simp add: nprrt-def le-iff-inf inf-ACI*)

**lemma** *sup-0-imp-0*:  $\text{sup } a \ (-a) = 0 \implies a = 0$

**proof** –

```
{
  fix a::'a
  assume hyp: sup a (-a) = 0
  hence sup a (-a) + a = a by (simp)
  hence sup (a+a) 0 = a by (simp add: add-sup-distrib-right)
  hence sup (a+a) 0 <= a by (simp)
  hence 0 <= a by (blast intro: order-trans inf-sup-ord)
}
note p = this
assume hyp: sup a (-a) = 0
hence hyp2: sup (-a) (-(-a)) = 0 by (simp add: sup-commute)
from p[OF hyp] p[OF hyp2] show a = 0 by simp
qed
```

**lemma** *inf-0-imp-0*:  $\text{inf } a \ (-a) = 0 \implies a = 0$

**apply** (*simp add: inf-eq-neg-sup*)  
**apply** (*simp add: sup-commute*)  
**apply** (*erule sup-0-imp-0*)  
**done**

**lemma** *inf-0-eq-0* [*simp*, *noatp*]:  $\text{inf } a \ (-a) = 0 \longleftrightarrow a = 0$   
**by** (*rule, erule inf-0-imp-0*) *simp*

**lemma** *sup-0-eq-0* [*simp*, *noatp*]:  $\text{sup } a \ (-a) = 0 \longleftrightarrow a = 0$   
**by** (*rule, erule sup-0-imp-0*) *simp*

**lemma** *zero-le-double-add-iff-zero-le-single-add* [*simp*]:

$$0 \leq a + a \longleftrightarrow 0 \leq a$$

**proof**

```
assume 0 <= a + a
hence a: inf (a+a) 0 = 0 by (simp add: le-iff-inf inf-commute)
have (inf a 0) + (inf a 0) = inf (inf (a+a) 0) a (is ?l=)
  by (simp add: add-sup-inf-distrib inf-ACI)
hence ?l = 0 + inf a 0 by (simp add: a, simp add: inf-commute)
hence inf a 0 = 0 by (simp only: add-right-cancel)
then show 0 <= a by (simp add: le-iff-inf inf-commute)
```

**next**

```
assume a: 0 <= a
show 0 <= a + a by (simp add: add-mono[OF a a, simplified])
qed
```

**lemma** *double-zero*:  $a + a = 0 \longleftrightarrow a = 0$

**proof**

**assume** *assm*:  $a + a = 0$

**then have**  $a + a + - a = - a$  **by** *simp*

**then have**  $a + (a + - a) = - a$  **by** (*simp only: add-assoc*)

**then have**  $a: - a = a$  **by** *simp*

**show**  $a = 0$  **apply** (*rule antisym*)

**apply** (*unfold neg-le-iff-le [symmetric, of a]*)

**unfolding** *a* **apply** *simp*

**unfolding** *zero-le-double-add-iff-zero-le-single-add [symmetric, of a]*

**unfolding** *assm* **unfolding** *le-less* **apply** *simp-all* **done**

**next**

**assume**  $a = 0$  **then show**  $a + a = 0$  **by** *simp*

**qed**

**lemma** *zero-less-double-add-iff-zero-less-single-add*:

$0 < a + a \longleftrightarrow 0 < a$

**proof** (*cases a = 0*)

**case** *True* **then show** *?thesis* **by** *auto*

**next**

**case** *False* **then show** *?thesis*

**unfolding** *less-le* **apply** *simp* **apply** *rule*

**apply** *clarify*

**apply** *rule*

**apply** *assumption*

**apply** (*rule notI*)

**unfolding** *double-zero* [*symmetric, of a*] **apply** *simp*

**done**

**qed**

**lemma** *double-add-le-zero-iff-single-add-le-zero* [*simp*]:

$a + a \leq 0 \longleftrightarrow a \leq 0$

**proof** –

**have**  $a + a \leq 0 \longleftrightarrow 0 \leq - (a + a)$  **by** (*subst le-minus-iff, simp*)

**moreover have**  $\dots \longleftrightarrow a \leq 0$  **by** (*simp add: zero-le-double-add-iff-zero-le-single-add*)

**ultimately show** *?thesis* **by** *blast*

**qed**

**lemma** *double-add-less-zero-iff-single-less-zero* [*simp*]:

$a + a < 0 \longleftrightarrow a < 0$

**proof** –

**have**  $a + a < 0 \longleftrightarrow 0 < - (a + a)$  **by** (*subst less-minus-iff, simp*)

**moreover have**  $\dots \longleftrightarrow a < 0$  **by** (*simp add: zero-less-double-add-iff-zero-less-single-add*)

**ultimately show** *?thesis* **by** *blast*

**qed**

**declare** *neg-inf-eq-sup* [*simp*] *neg-sup-eq-inf* [*simp*]

**lemma** *le-minus-self-iff*:  $a \leq - a \longleftrightarrow a \leq 0$

```

proof –
  from add-le-cancel-left [of uminus a plus a a zero]
  have ( $a \leq -a$ ) = ( $a+a \leq 0$ )
    by (simp add: add-assoc[symmetric])
  thus ?thesis by simp
qed

lemma minus-le-self-iff:  $- a \leq a \iff 0 \leq a$ 
proof –
  from add-le-cancel-left [of uminus a zero plus a a]
  have ( $-a \leq a$ ) = ( $0 \leq a+a$ )
    by (simp add: add-assoc[symmetric])
  thus ?thesis by simp
qed

lemma zero-le-iff-zero-nprt:  $0 \leq a \iff \text{nprt } a = 0$ 
by (simp add: le-iff-inf nprt-def inf-commute)

lemma le-zero-iff-zero-pprt:  $a \leq 0 \iff \text{pprt } a = 0$ 
by (simp add: le-iff-sup pprr-def sup-commute)

lemma le-zero-iff-pprr-id:  $0 \leq a \iff \text{pprr } a = a$ 
by (simp add: le-iff-sup pprr-def sup-commute)

lemma zero-le-iff-nprt-id:  $a \leq 0 \iff \text{nprt } a = a$ 
by (simp add: le-iff-inf nprt-def inf-commute)

lemma pprr-mono [simp, noatp]:  $a \leq b \implies \text{pprr } a \leq \text{pprr } b$ 
by (simp add: le-iff-sup pprr-def sup-ACI sup-assoc [symmetric, of a])

lemma nprt-mono [simp, noatp]:  $a \leq b \implies \text{nprt } a \leq \text{nprt } b$ 
by (simp add: le-iff-inf nprt-def inf-ACI inf-assoc [symmetric, of a])

end

lemmas add-sup-inf-distrib = add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right
add-sup-distrib-left

class lordered-ab-group-add-abs = lordered-ab-group-add + abs +
  assumes abs-lattice:  $|a| = \text{sup } a \ (-a)$ 
begin

lemma abs-prts:  $|a| = \text{pprr } a - \text{nprt } a$ 
proof –
  have  $0 \leq |a|$ 
  proof –
    have  $a \leq |a|$  and  $-a \leq |a|$  by (auto simp add: abs-lattice)
    show ?thesis by (rule add-mono [OF a b, simplified])
  qed

```

```

qed
then have  $0 \leq \sup a \ (-a)$  unfolding abs-lattice .
then have  $\sup (\sup a \ (-a)) \ 0 = \sup a \ (-a)$  by (rule sup-absorb1)
then show ?thesis
  by (simp add: add-sup-inf-distrib sup-ACI
      pprt-def nprrt-def diff-minus abs-lattice)
qed

subclass pordered-ab-group-add-abs
proof
  have abs-ge-zero [simp]:  $\bigwedge a. \ 0 \leq |a|$ 
  proof -
    fix a b
    have a:  $a \leq |a|$  and b:  $-a \leq |a|$  by (auto simp add: abs-lattice)
    show  $0 \leq |a|$  by (rule add-mono [OF a b, simplified])
  qed
  have abs-leI:  $\bigwedge a \ b. \ a \leq b \implies -a \leq b \implies |a| \leq b$ 
    by (simp add: abs-lattice le-supI)
  fix a b
  show  $0 \leq |a|$  by simp
  show  $a \leq |a|$ 
    by (auto simp add: abs-lattice)
  show  $|-a| = |a|$ 
    by (simp add: abs-lattice sup-commute)
  show  $a \leq b \implies -a \leq b \implies |a| \leq b$  by (fact abs-leI)
  show  $|a + b| \leq |a| + |b|$ 
  proof -
    have g:  $\text{abs } a + \text{abs } b = \sup (a+b) (\sup (-a-b) (\sup (-a+b) (a + (-b))))$ 
    (is  $\text{--sup } ?m \ ?n$ )
    by (simp add: abs-lattice add-sup-inf-distrib sup-ACI diff-minus)
    have a:  $a+b \leq \sup ?m \ ?n$  by (simp)
    have b:  $-a-b \leq ?n$  by (simp)
    have c:  $?n \leq \sup ?m \ ?n$  by (simp)
    from b c have d:  $-a-b \leq \sup ?m \ ?n$  by (rule order-trans)
    have e:  $-a-b = -(a+b)$  by (simp add: diff-minus)
    from a d e have abs(a+b):  $\leq \sup ?m \ ?n$ 
    by (drule-tac abs-leI, auto)
    with g[symmetric] show ?thesis by simp
  qed
qed

end

lemma sup-eq-if:
  fixes a :: 'a::{\iordered-ab-group-add, linorder}
  shows  $\sup a \ (-a) = (\text{if } a < 0 \text{ then } -a \text{ else } a)$ 
proof -
  note add-le-cancel-right [of a a - a, symmetric, simplified]
  moreover note add-le-cancel-right [of -a a a, symmetric, simplified]

```

**then show** *?thesis* **by** (auto simp: sup-max max-def)  
**qed**

**lemma** *abs-if-lattice*:  
**fixes**  $a :: 'a :: \{\text{ordered-ab-group-add-abs, linorder}\}$   
**shows**  $|a| = (\text{if } a < 0 \text{ then } -a \text{ else } a)$   
**by** auto

Needed for abelian cancellation simprocs:

**lemma** *add-cancel-21*:  $((x :: 'a :: \text{ab-group-add}) + (y + z) = y + u) = (x + z = u)$   
**apply** (subst add-left-commute)  
**apply** (subst add-left-cancel)  
**apply** simp  
**done**

**lemma** *add-cancel-end*:  $(x + (y + z) = y) = (x = - (z :: 'a :: \text{ab-group-add}))$   
**apply** (subst add-cancel-21[of - - 0, simplified])  
**apply** (simp add: add-right-cancel[symmetric, of  $x - z$  z, simplified])  
**done**

**lemma** *less-eqI*:  $(x :: 'a :: \text{pordered-ab-group-add}) - y = x' - y' \implies (x < y) = (x' < y')$   
**by** (simp add: less-iff-diff-less-0[of  $x$   $y$ ] less-iff-diff-less-0[of  $x'$   $y'$ ])

**lemma** *le-eqI*:  $(x :: 'a :: \text{pordered-ab-group-add}) - y = x' - y' \implies (y \leq x) = (y' \leq x')$   
**apply** (simp add: le-iff-diff-le-0[of  $y$   $x$ ] le-iff-diff-le-0[of  $y'$   $x'$ ])  
**apply** (simp add: neg-le-iff-le[symmetric, of  $y - x$  0] neg-le-iff-le[symmetric, of  $y' - x'$  0])  
**done**

**lemma** *eq-eqI*:  $(x :: 'a :: \text{ab-group-add}) - y = x' - y' \implies (x = y) = (x' = y')$   
**by** (simp only: eq-iff-diff-eq-0[of  $x$   $y$ ] eq-iff-diff-eq-0[of  $x'$   $y'$ ])

**lemma** *diff-def*:  $(x :: 'a :: \text{ab-group-add}) - y == x + (-y)$   
**by** (simp add: diff-minus)

**lemma** *add-minus-cancel*:  $(a :: 'a :: \text{ab-group-add}) + (-a + b) = b$   
**by** (simp add: add-assoc[symmetric])

**lemma** *le-add-right-mono*:  
**assumes**  
 $a \leq b + (c :: 'a :: \text{pordered-ab-group-add})$   
 $c \leq d$   
**shows**  $a \leq b + d$   
**apply** (rule-tac order-trans[where  $y = b + c$ ])  
**apply** (simp-all add: prems)  
**done**

**lemma** *estimate-by-abs*:

$a + b \leq (c :: 'a :: \text{ordered-ab-group-add-abs}) \implies a \leq c + \text{abs } b$

**proof** –

assume  $a + b \leq c$

hence 2:  $a \leq c + (-b)$  **by** (*simp add: algebra-simps*)

have 3:  $(-b) \leq \text{abs } b$  **by** (*rule abs-ge-minus-self*)

show ?thesis **by** (*rule le-add-right-mono[OF 2 3]*)

**qed**

## 10.7 Tools setup

**lemma** *add-mono-thms-ordered-semiring* [*noatp*]:

**fixes**  $i\ j\ k :: 'a :: \text{pordered-ab-semigroup-add}$

**shows**  $i \leq j \wedge k \leq l \implies i + k \leq j + l$

and  $i = j \wedge k \leq l \implies i + k \leq j + l$

and  $i \leq j \wedge k = l \implies i + k \leq j + l$

and  $i = j \wedge k = l \implies i + k = j + l$

**by** (*rule add-mono, clarify+*)

**lemma** *add-mono-thms-ordered-field* [*noatp*]:

**fixes**  $i\ j\ k :: 'a :: \text{pordered-cancel-ab-semigroup-add}$

**shows**  $i < j \wedge k = l \implies i + k < j + l$

and  $i = j \wedge k < l \implies i + k < j + l$

and  $i < j \wedge k \leq l \implies i + k < j + l$

and  $i \leq j \wedge k < l \implies i + k < j + l$

and  $i < j \wedge k < l \implies i + k < j + l$

**by** (*auto intro: add-strict-right-mono add-strict-left-mono*

*add-less-le-mono add-le-less-mono add-strict-mono*)

Simplification of  $x - y < (0 :: 'a)$ , etc.

**lemmas** *diff-less-0-iff-less* [*simp, noatp*] = *less-iff-diff-less-0* [*symmetric*]

**lemmas** *diff-le-0-iff-le* [*simp, noatp*] = *le-iff-diff-le-0* [*symmetric*]

**ML**  $\langle\langle$

*structure ab-group-add-cancel = Abel-Cancel*

*(*

*(\* term order for abelian groups \*)*

*fun agrp-ord (Const (a, -)) = find-index (fn a' => a = a')*

*[@{const-name HOL.zero}, @{const-name HOL.plus},*

*@{const-name HOL.uminus}, @{const-name HOL.minus}]*

*| agrp-ord - = ~1;*

*fun termless-agrp (a, b) = (TermOrd.term-lpo agrp-ord (a, b) = LESS);*

*local*

*val ac1 = mk-meta-eq @{thm add-assoc};*

*val ac2 = mk-meta-eq @{thm add-commute};*



```

val ac3 = mk-meta-eq @{thm add-left-commute};
fun solve-add-ac thy - (- $ (Const (@{const-name HOL.plus},-) $ - $ -) $ -) =
  SOME ac1
  | solve-add-ac thy - (- $ x $ (Const (@{const-name HOL.plus},-) $ y $ z)) =
    if termless-agrp (y, x) then SOME ac3 else NONE
  | solve-add-ac thy - (- $ x $ y) =
    if termless-agrp (y, x) then SOME ac2 else NONE
  | solve-add-ac thy - - = NONE
in
  val add-ac-proc = Simplifier.simproc (the-context ())
    add-ac-proc [x + y::'a::ab-semigroup-add] solve-add-ac;
end;

val eq-reflection = @{thm eq-reflection};

val T = @{typ 'a::ab-group-add};

val cancel-ss = HOL-basic-ss settermless termless-agrp
  addsimprocs [add-ac-proc] addsimps
  [@{thm add-0-left}, @{thm add-0-right}, @{thm diff-def},
   @{thm minus-add-distrib}, @{thm minus-minus}, @{thm minus-zero},
   @{thm right-minus}, @{thm left-minus}, @{thm add-minus-cancel},
   @{thm minus-add-cancel}];

val sum-pats = [@{cterm x + y::'a::ab-group-add}, @{cterm x - y::'a::ab-group-add}];

val eqI-rules = [@{thm less-eqI}, @{thm le-eqI}, @{thm eq-eqI}];

val dest-eqI =
  fst o HOLogic.dest-bin op = HOLogic.boolT o HOLogic.dest-Trueprop o concl-of;

);
>>

ML <<
  Addsimprocs [ab-group-add-cancel.sum-conv, ab-group-add-cancel.rel-conv];
>>

end

```

## 11 Ring-and-Field: (Ordered) Rings and Fields

```

theory Ring-and-Field
imports OrderedGroup
begin

```

The theory of partially ordered rings is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

- <http://www.mathworld.com> by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer.

```
class semiring = ab-semigroup-add + semigroup-mult +
  assumes left-distrib[algebra-simps]: (a + b) * c = a * c + b * c
  assumes right-distrib[algebra-simps]: a * (b + c) = a * b + a * c
begin
```

For the *combine-numerals* simproc

```
lemma combine-common-factor:
  a * e + (b * e + c) = (a + b) * e + c
by (simp add: left-distrib add-ac)
```

end

```
class mult-zero = times + zero +
  assumes mult-zero-left [simp]: 0 * a = 0
  assumes mult-zero-right [simp]: a * 0 = 0
```

```
class semiring-0 = semiring + comm-monoid-add + mult-zero
```

```
class semiring-0-cancel = semiring + cancel-comm-monoid-add
begin
```

```
subclass semiring-0
```

```
proof
```

```
  fix a :: 'a
```

```
  have 0 * a + 0 * a = 0 * a + 0 by (simp add: left-distrib [symmetric])
```

```
  thus 0 * a = 0 by (simp only: add-left-cancel)
```

```
next
```

```
  fix a :: 'a
```

```
  have a * 0 + a * 0 = a * 0 + 0 by (simp add: right-distrib [symmetric])
```

```
  thus a * 0 = 0 by (simp only: add-left-cancel)
```

```
qed
```

end

```
class comm-semiring = ab-semigroup-add + ab-semigroup-mult +
  assumes distrib: (a + b) * c = a * c + b * c
begin
```

```

subclass semiring
proof
  fix a b c :: 'a
  show  $(a + b) * c = a * c + b * c$  by (simp add: distrib)
  have  $a * (b + c) = (b + c) * a$  by (simp add: mult-ac)
  also have  $\dots = b * a + c * a$  by (simp only: distrib)
  also have  $\dots = a * b + a * c$  by (simp add: mult-ac)
  finally show  $a * (b + c) = a * b + a * c$  by blast
qed

end

class comm-semiring-0 = comm-semiring + comm-monoid-add + mult-zero
begin

subclass semiring-0 ..

end

class comm-semiring-0-cancel = comm-semiring + cancel-comm-monoid-add
begin

subclass semiring-0-cancel ..

subclass comm-semiring-0 ..

end

class zero-neq-one = zero + one +
  assumes zero-neq-one [simp]:  $0 \neq 1$ 
begin

lemma one-neq-zero [simp]:  $1 \neq 0$ 
by (rule not-sym) (rule zero-neq-one)

end

class semiring-1 = zero-neq-one + semiring-0 + monoid-mult

Abstract divisibility

class dvd = times
begin

definition dvd :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool (infixl dvd 50) where
  [code del]:  $b \text{ dvd } a \iff (\exists k. a = b * k)$ 

lemma dvdI [intro?]:  $a = b * k \implies b \text{ dvd } a$ 
  unfolding dvd-def ..

```

**lemma** *dvdE* [*elim?*]:  $b \text{ dvd } a \implies (\bigwedge k. a = b * k \implies P) \implies P$   
**unfolding** *dvd-def* **by** *blast*

**end**

**class** *comm-semiring-1* = *zero-neq-one* + *comm-semiring-0* + *comm-monoid-mult*  
+ *dvd*

**begin**

**subclass** *semiring-1* ..

**lemma** *dvd-refl*[*simp*]:  $a \text{ dvd } a$   
**proof**  
  **show**  $a = a * 1$  **by** *simp*  
**qed**

**lemma** *dvd-trans*:  
  **assumes**  $a \text{ dvd } b$  **and**  $b \text{ dvd } c$   
  **shows**  $a \text{ dvd } c$   
**proof** –  
  **from** *assms* **obtain**  $v$  **where**  $b = a * v$  **by** (*auto elim!*: *dvdE*)  
  **moreover from** *assms* **obtain**  $w$  **where**  $c = b * w$  **by** (*auto elim!*: *dvdE*)  
  **ultimately have**  $c = a * (v * w)$  **by** (*simp add: mult-assoc*)  
  **then show** *?thesis* ..  
**qed**

**lemma** *dvd-0-left-iff* [*noatp, simp*]:  $0 \text{ dvd } a \longleftrightarrow a = 0$   
**by** (*auto intro: dvd-refl elim!*: *dvdE*)

**lemma** *dvd-0-right* [*iff*]:  $a \text{ dvd } 0$   
**proof**  
  **show**  $0 = a * 0$  **by** *simp*  
**qed**

**lemma** *one-dvd* [*simp*]:  $1 \text{ dvd } a$   
**by** (*auto intro!*: *dvdI*)

**lemma** *dvd-mult*[*simp*]:  $a \text{ dvd } c \implies a \text{ dvd } (b * c)$   
**by** (*auto intro!*: *mult-left-commute dvdI elim!*: *dvdE*)

**lemma** *dvd-mult2*[*simp*]:  $a \text{ dvd } b \implies a \text{ dvd } (b * c)$   
  **apply** (*subst mult-commute*)  
  **apply** (*erule dvd-mult*)  
  **done**

**lemma** *dvd-triv-right* [*simp*]:  $a \text{ dvd } b * a$   
**by** (*rule dvd-mult*) (*rule dvd-refl*)

**lemma** *dvd-triv-left* [*simp*]:  $a \text{ dvd } a * b$   
**by** (*rule dvd-mult2*) (*rule dvd-refl*)

**lemma** *mult-dvd-mono*:  
**assumes**  $a \text{ dvd } b$   
**and**  $c \text{ dvd } d$   
**shows**  $a * c \text{ dvd } b * d$   
**proof** –  
**from**  $\langle a \text{ dvd } b \rangle$  **obtain**  $b'$  **where**  $b = a * b' ..$   
**moreover from**  $\langle c \text{ dvd } d \rangle$  **obtain**  $d'$  **where**  $d = c * d' ..$   
**ultimately have**  $b * d = (a * c) * (b' * d')$  **by** (*simp add: mult-ac*)  
**then show** *?thesis* ..  
**qed**

**lemma** *dvd-mult-left*:  $a * b \text{ dvd } c \implies a \text{ dvd } c$   
**by** (*simp add: dvd-def mult-assoc, blast*)

**lemma** *dvd-mult-right*:  $a * b \text{ dvd } c \implies b \text{ dvd } c$   
**unfolding** *mult-ac* [*of a*] **by** (*rule dvd-mult-left*)

**lemma** *dvd-0-left*:  $0 \text{ dvd } a \implies a = 0$   
**by** *simp*

**lemma** *dvd-add*[*simp*]:  
**assumes**  $a \text{ dvd } b$  **and**  $a \text{ dvd } c$  **shows**  $a \text{ dvd } (b + c)$   
**proof** –  
**from**  $\langle a \text{ dvd } b \rangle$  **obtain**  $b'$  **where**  $b = a * b' ..$   
**moreover from**  $\langle a \text{ dvd } c \rangle$  **obtain**  $c'$  **where**  $c = a * c' ..$   
**ultimately have**  $b + c = a * (b' + c')$  **by** (*simp add: right-distrib*)  
**then show** *?thesis* ..  
**qed**  
**end**

**class** *no-zero-divisors* = *zero* + *times* +  
**assumes** *no-zero-divisors*:  $a \neq 0 \implies b \neq 0 \implies a * b \neq 0$

**class** *semiring-1-cancel* = *semiring* + *cancel-comm-monoid-add*  
+ *zero-neq-one* + *monoid-mult*  
**begin**

**subclass** *semiring-0-cancel* ..

**subclass** *semiring-1* ..

**end**

**class** *comm-semiring-1-cancel* = *comm-semiring* + *cancel-comm-monoid-add*

+ *zero-neq-one* + *comm-monoid-mult*  
**begin**

**subclass** *semiring-1-cancel* ..  
**subclass** *comm-semiring-0-cancel* ..  
**subclass** *comm-semiring-1* ..

**end**

**class** *ring* = *semiring* + *ab-group-add*  
**begin**

**subclass** *semiring-0-cancel* ..

Distribution rules

**lemma** *minus-mult-left*:  $-(a * b) = -a * b$   
**by** (*rule equals-zero-I*) (*simp add: left-distrib [symmetric]*)

**lemma** *minus-mult-right*:  $-(a * b) = a * -b$   
**by** (*rule equals-zero-I*) (*simp add: right-distrib [symmetric]*)

Extract signs from products

**lemmas** *mult-minus-left* [*simp, noatp*] = *minus-mult-left* [*symmetric*]  
**lemmas** *mult-minus-right* [*simp, noatp*] = *minus-mult-right* [*symmetric*]

**lemma** *minus-mult-minus* [*simp*]:  $-a * -b = a * b$   
**by** *simp*

**lemma** *minus-mult-commute*:  $-a * b = a * -b$   
**by** *simp*

**lemma** *right-diff-distrib*[*algebra-simps*]:  $a * (b - c) = a * b - a * c$   
**by** (*simp add: right-distrib diff-minus*)

**lemma** *left-diff-distrib*[*algebra-simps*]:  $(a - b) * c = a * c - b * c$   
**by** (*simp add: left-distrib diff-minus*)

**lemmas** *ring-distrib*s[*noatp*] =  
*right-distrib left-distrib left-diff-distrib right-diff-distrib*

Legacy - use *algebra-simps*

**lemmas** *ring-simps*[*noatp*] = *algebra-simps*

**lemma** *eq-add-iff1*:  
 $a * e + c = b * e + d \iff (a - b) * e + c = d$   
**by** (*simp add: algebra-simps*)

**lemma** *eq-add-iff2*:  
 $a * e + c = b * e + d \iff c = (b - a) * e + d$

```

by (simp add: algebra-simps)

end

lemmas ring-distrib[noatp] =
  right-distrib left-distrib left-diff-distrib right-diff-distrib

class comm-ring = comm-semiring + ab-group-add
begin

subclass ring ..
subclass comm-semiring-0-cancel ..

end

class ring-1 = ring + zero-neq-one + monoid-mult
begin

subclass semiring-1-cancel ..

end

class comm-ring-1 = comm-ring + zero-neq-one + comm-monoid-mult

begin

subclass ring-1 ..
subclass comm-semiring-1-cancel ..

lemma dvd-minus-iff [simp]:  $x \text{ dvd } - y \longleftrightarrow x \text{ dvd } y$ 
proof
  assume  $x \text{ dvd } - y$ 
  then have  $x \text{ dvd } - 1 * - y$  by (rule dvd-mult)
  then show  $x \text{ dvd } y$  by simp
next
  assume  $x \text{ dvd } y$ 
  then have  $x \text{ dvd } - 1 * y$  by (rule dvd-mult)
  then show  $x \text{ dvd } - y$  by simp
qed

lemma minus-dvd-iff [simp]:  $- x \text{ dvd } y \longleftrightarrow x \text{ dvd } y$ 
proof
  assume  $- x \text{ dvd } y$ 
  then obtain  $k$  where  $y = - x * k$  ..
  then have  $y = x * - k$  by simp
  then show  $x \text{ dvd } y$  ..
next
  assume  $x \text{ dvd } y$ 
  then obtain  $k$  where  $y = x * k$  ..

```

```

    then have  $y = -x * -k$  by simp
    then show  $-x \text{ dvd } y$  ..
qed

```

```

lemma dvd-diff[simp]:  $x \text{ dvd } y \implies x \text{ dvd } z \implies x \text{ dvd } (y - z)$ 
by (simp add: diff-minus dvd-minus-iff)

```

```

end

```

```

class ring-no-zero-divisors = ring + no-zero-divisors
begin

```

```

lemma mult-eq-0-iff [simp]:
  shows  $a * b = 0 \iff (a = 0 \vee b = 0)$ 
proof (cases  $a = 0 \vee b = 0$ )
  case False then have  $a \neq 0$  and  $b \neq 0$  by auto
  then show ?thesis using no-zero-divisors by simp
next
  case True then show ?thesis by auto
qed

```

Cancellation of equalities with a common factor

```

lemma mult-cancel-right [simp, noatp]:
   $a * c = b * c \iff c = 0 \vee a = b$ 
proof -
  have  $(a * c = b * c) \iff ((a - b) * c = 0)$ 
  by (simp add: algebra-simps right-minus-eq)
  thus ?thesis by (simp add: disj-commute right-minus-eq)
qed

```

```

lemma mult-cancel-left [simp, noatp]:
   $c * a = c * b \iff c = 0 \vee a = b$ 
proof -
  have  $(c * a = c * b) \iff (c * (a - b) = 0)$ 
  by (simp add: algebra-simps right-minus-eq)
  thus ?thesis by (simp add: right-minus-eq)
qed

```

```

end

```

```

class ring-1-no-zero-divisors = ring-1 + ring-no-zero-divisors
begin

```

```

lemma mult-cancel-right1 [simp]:
   $c = b * c \iff c = 0 \vee b = 1$ 
by (insert mult-cancel-right [of 1 c b], force)

```

```

lemma mult-cancel-right2 [simp]:
   $a * c = c \iff c = 0 \vee a = 1$ 

```



**by** (*insert mult-cancel-right* [*of a c 1*], *simp*)

**lemma** *mult-cancel-left1* [*simp*]:

$$c = c * b \longleftrightarrow c = 0 \vee b = 1$$

**by** (*insert mult-cancel-left* [*of c 1 b*], *force*)

**lemma** *mult-cancel-left2* [*simp*]:

$$c * a = c \longleftrightarrow c = 0 \vee a = 1$$

**by** (*insert mult-cancel-left* [*of c a 1*], *simp*)

**end**

**class** *idom* = *comm-ring-1* + *no-zero-divisors*

**begin**

**subclass** *ring-1-no-zero-divisors* ..

**lemma** *square-eq-iff*:  $a * a = b * b \longleftrightarrow (a = b \vee a = -b)$

**proof**

**assume**  $a * a = b * b$

**then have**  $(a - b) * (a + b) = 0$

**by** (*simp add: algebra-simps*)

**then show**  $a = b \vee a = -b$

**by** (*simp add: right-minus-eq eq-neg-iff-add-eq-0*)

**next**

**assume**  $a = b \vee a = -b$

**then show**  $a * a = b * b$  **by** *auto*

**qed**

**lemma** *dvd-mult-cancel-right* [*simp*]:

$$a * c \text{ dvd } b * c \longleftrightarrow c = 0 \vee a \text{ dvd } b$$

**proof** –

**have**  $a * c \text{ dvd } b * c \longleftrightarrow (\exists k. b * c = (a * k) * c)$

**unfolding** *dvd-def* **by** (*simp add: mult-ac*)

**also have**  $(\exists k. b * c = (a * k) * c) \longleftrightarrow c = 0 \vee a \text{ dvd } b$

**unfolding** *dvd-def* **by** *simp*

**finally show** *?thesis* .

**qed**

**lemma** *dvd-mult-cancel-left* [*simp*]:

$$c * a \text{ dvd } c * b \longleftrightarrow c = 0 \vee a \text{ dvd } b$$

**proof** –

**have**  $c * a \text{ dvd } c * b \longleftrightarrow (\exists k. b * c = (a * k) * c)$

**unfolding** *dvd-def* **by** (*simp add: mult-ac*)

**also have**  $(\exists k. b * c = (a * k) * c) \longleftrightarrow c = 0 \vee a \text{ dvd } b$

**unfolding** *dvd-def* **by** *simp*

**finally show** *?thesis* .

**qed**

**end**

```
class division-ring = ring-1 + inverse +
  assumes left-inverse [simp]:  $a \neq 0 \implies \text{inverse } a * a = 1$ 
  assumes right-inverse [simp]:  $a \neq 0 \implies a * \text{inverse } a = 1$ 
begin
```

```
subclass ring-1-no-zero-divisors
```

```
proof
```

```
  fix a b :: 'a
```

```
  assume a:  $a \neq 0$  and b:  $b \neq 0$ 
```

```
  show  $a * b \neq 0$ 
```

```
  proof
```

```
    assume ab:  $a * b = 0$ 
```

```
    hence  $0 = \text{inverse } a * (a * b) * \text{inverse } b$  by simp
```

```
    also have  $\dots = (\text{inverse } a * a) * (b * \text{inverse } b)$ 
```

```
      by (simp only: mult-assoc)
```

```
    also have  $\dots = 1$  using a b by simp
```

```
    finally show False by simp
```

```
  qed
```

```
qed
```

```
lemma nonzero-imp-inverse-nonzero:
```

```
   $a \neq 0 \implies \text{inverse } a \neq 0$ 
```

```
proof
```

```
  assume ianz:  $\text{inverse } a = 0$ 
```

```
  assume  $a \neq 0$ 
```

```
  hence  $1 = a * \text{inverse } a$  by simp
```

```
  also have  $\dots = 0$  by (simp add: ianz)
```

```
  finally have  $1 = 0$  .
```

```
  thus False by (simp add: eq-commute)
```

```
qed
```

```
lemma inverse-zero-imp-zero:
```

```
   $\text{inverse } a = 0 \implies a = 0$ 
```

```
apply (rule classical)
```

```
apply (drule nonzero-imp-inverse-nonzero)
```

```
apply auto
```

```
done
```

```
lemma inverse-unique:
```

```
  assumes ab:  $a * b = 1$ 
```

```
  shows  $\text{inverse } a = b$ 
```

```
proof –
```

```
  have  $a \neq 0$  using ab by (cases a = 0) simp-all
```

```
  moreover have  $\text{inverse } a * (a * b) = \text{inverse } a$  by (simp add: ab)
```

```
  ultimately show ?thesis by (simp add: mult-assoc [symmetric])
```

```
qed
```

**lemma** *nonzero-inverse-minus-eq*:

$a \neq 0 \implies \text{inverse } (-a) = - \text{inverse } a$

**by** (rule *inverse-unique*) *simp*

**lemma** *nonzero-inverse-inverse-eq*:

$a \neq 0 \implies \text{inverse } (\text{inverse } a) = a$

**by** (rule *inverse-unique*) *simp*

**lemma** *nonzero-inverse-eq-imp-eq*:

**assumes**  $\text{inverse } a = \text{inverse } b$  **and**  $a \neq 0$  **and**  $b \neq 0$

**shows**  $a = b$

**proof** –

**from**  $\langle \text{inverse } a = \text{inverse } b \rangle$

**have**  $\text{inverse } (\text{inverse } a) = \text{inverse } (\text{inverse } b)$  **by** (rule *arg-cong*)

**with**  $\langle a \neq 0 \rangle$  **and**  $\langle b \neq 0 \rangle$  **show**  $a = b$

**by** (*simp add: nonzero-inverse-inverse-eq*)

**qed**

**lemma** *inverse-1* [*simp*]:  $\text{inverse } 1 = 1$

**by** (rule *inverse-unique*) *simp*

**lemma** *nonzero-inverse-mult-distrib*:

**assumes**  $a \neq 0$  **and**  $b \neq 0$

**shows**  $\text{inverse } (a * b) = \text{inverse } b * \text{inverse } a$

**proof** –

**have**  $a * (b * \text{inverse } b) * \text{inverse } a = 1$  **using** *assms* **by** *simp*

**hence**  $a * b * (\text{inverse } b * \text{inverse } a) = 1$  **by** (*simp only: mult-assoc*)

**thus** *?thesis* **by** (rule *inverse-unique*)

**qed**

**lemma** *division-ring-inverse-add*:

$a \neq 0 \implies b \neq 0 \implies \text{inverse } a + \text{inverse } b = \text{inverse } a * (a + b) * \text{inverse } b$

**by** (*simp add: algebra-simps*)

**lemma** *division-ring-inverse-diff*:

$a \neq 0 \implies b \neq 0 \implies \text{inverse } a - \text{inverse } b = \text{inverse } a * (b - a) * \text{inverse } b$

**by** (*simp add: algebra-simps*)

**end**

**class** *field* = *comm-ring-1* + *inverse* +

**assumes** *field-inverse*:  $a \neq 0 \implies \text{inverse } a * a = 1$

**assumes** *divide-inverse*:  $a / b = a * \text{inverse } b$

**begin**

**subclass** *division-ring*

**proof**

**fix**  $a :: 'a$

**assume**  $a \neq 0$

```

thus inverse a * a = 1 by (rule field-inverse)
thus a * inverse a = 1 by (simp only: mult-commute)
qed

subclass idom ..

lemma right-inverse-eq: b ≠ 0 ⇒ a / b = 1 ↔ a = b
proof
  assume neq: b ≠ 0
  {
    hence a = (a / b) * b by (simp add: divide-inverse mult-ac)
    also assume a / b = 1
    finally show a = b by simp
  }
next
  assume a = b
  with neq show a / b = 1 by (simp add: divide-inverse)
qed

```

```

lemma nonzero-inverse-eq-divide: a ≠ 0 ⇒ inverse a = 1 / a
by (simp add: divide-inverse)

```

```

lemma divide-self [simp]: a ≠ 0 ⇒ a / a = 1
by (simp add: divide-inverse)

```

```

lemma divide-zero-left [simp]: 0 / a = 0
by (simp add: divide-inverse)

```

```

lemma inverse-eq-divide: inverse a = 1 / a
by (simp add: divide-inverse)

```

```

lemma add-divide-distrib: (a+b) / c = a/c + b/c
by (simp add: divide-inverse algebra-simps)

```

There is no slick version using division by zero.

```

lemma inverse-add:
  [| a ≠ 0; b ≠ 0 |]
  ==> inverse a + inverse b = (a + b) * inverse a * inverse b
by (simp add: division-ring-inverse-add mult-ac)

```

```

lemma nonzero-mult-divide-mult-cancel-left [simp, noatp]:
assumes [simp]: b≠0 and [simp]: c≠0 shows (c*a)/(c*b) = a/b
proof –
  have (c*a)/(c*b) = c * a * (inverse b * inverse c)
  by (simp add: divide-inverse nonzero-inverse-mult-distrib)
  also have ... = a * inverse b * (inverse c * c)
  by (simp only: mult-ac)
  also have ... = a * inverse b by simp
  finally show ?thesis by (simp add: divide-inverse)

```

qed

**lemma** *nonzero-mult-divide-mult-cancel-right* [simp, noatp]:  
 $\llbracket b \neq 0; c \neq 0 \rrbracket \implies (a * c) / (b * c) = a / b$   
**by** (simp add: mult-commute [of - c])

**lemma** *divide-1* [simp]:  $a / 1 = a$   
**by** (simp add: divide-inverse)

**lemma** *times-divide-eq-right*:  $a * (b / c) = (a * b) / c$   
**by** (simp add: divide-inverse mult-assoc)

**lemma** *times-divide-eq-left*:  $(b / c) * a = (b * a) / c$   
**by** (simp add: divide-inverse mult-ac)

These are later declared as simp rules.

**lemmas** *times-divide-eq* [noatp] = *times-divide-eq-right times-divide-eq-left*

**lemma** *add-frac-eq*:  
**assumes**  $y \neq 0$  **and**  $z \neq 0$   
**shows**  $x / y + w / z = (x * z + w * y) / (y * z)$   
**proof** –  
**have**  $x / y + w / z = (x * z) / (y * z) + (y * w) / (y * z)$   
**using** *assms* **by** *simp*  
**also have**  $\dots = (x * z + y * w) / (y * z)$   
**by** (simp only: add-divide-distrib)  
**finally show** ?thesis  
**by** (simp only: mult-commute)  
 qed

Special Cancellation Simprules for Division

**lemma** *nonzero-mult-divide-cancel-right* [simp, noatp]:  
 $b \neq 0 \implies a * b / b = a$   
**using** *nonzero-mult-divide-mult-cancel-right* [of 1 b a] **by** *simp*

**lemma** *nonzero-mult-divide-cancel-left* [simp, noatp]:  
 $a \neq 0 \implies a * b / a = b$   
**using** *nonzero-mult-divide-mult-cancel-left* [of 1 a b] **by** *simp*

**lemma** *nonzero-divide-mult-cancel-right* [simp, noatp]:  
 $\llbracket a \neq 0; b \neq 0 \rrbracket \implies b / (a * b) = 1 / a$   
**using** *nonzero-mult-divide-mult-cancel-right* [of a b 1] **by** *simp*

**lemma** *nonzero-divide-mult-cancel-left* [simp, noatp]:  
 $\llbracket a \neq 0; b \neq 0 \rrbracket \implies a / (a * b) = 1 / b$   
**using** *nonzero-mult-divide-mult-cancel-left* [of b a 1] **by** *simp*

**lemma** *nonzero-mult-divide-mult-cancel-left2* [simp, noatp]:  
 $\llbracket b \neq 0; c \neq 0 \rrbracket \implies (c * a) / (b * c) = a / b$

**using** *nonzero-mult-divide-mult-cancel-left* [of  $b\ c\ a$ ] **by** (*simp add: mult-ac*)

**lemma** *nonzero-mult-divide-mult-cancel-right2* [*simp, noatp*]:

$\llbracket b \neq 0; c \neq 0 \rrbracket \implies (a * c) / (c * b) = a / b$

**using** *nonzero-mult-divide-mult-cancel-right* [of  $b\ c\ a$ ] **by** (*simp add: mult-ac*)

**lemma** *minus-divide-left*:  $-(a / b) = (-a) / b$

**by** (*simp add: divide-inverse*)

**lemma** *nonzero-minus-divide-right*:  $b \neq 0 \implies -(a / b) = a / (-b)$

**by** (*simp add: divide-inverse nonzero-inverse-minus-eq*)

**lemma** *nonzero-minus-divide-divide*:  $b \neq 0 \implies (-a) / (-b) = a / b$

**by** (*simp add: divide-inverse nonzero-inverse-minus-eq*)

**lemma** *divide-minus-left* [*simp, noatp*]:  $(-a) / b = -(a / b)$

**by** (*simp add: divide-inverse*)

**lemma** *diff-divide-distrib*:  $(a - b) / c = a / c - b / c$

**by** (*simp add: diff-minus add-divide-distrib*)

**lemma** *add-divide-eq-iff*:

$z \neq 0 \implies x + y / z = (z * x + y) / z$

**by** (*simp add: add-divide-distrib*)

**lemma** *divide-add-eq-iff*:

$z \neq 0 \implies x / z + y = (x + z * y) / z$

**by** (*simp add: add-divide-distrib*)

**lemma** *diff-divide-eq-iff*:

$z \neq 0 \implies x - y / z = (z * x - y) / z$

**by** (*simp add: diff-divide-distrib*)

**lemma** *divide-diff-eq-iff*:

$z \neq 0 \implies x / z - y = (x - z * y) / z$

**by** (*simp add: diff-divide-distrib*)

**lemma** *nonzero-eq-divide-eq*:  $c \neq 0 \implies a = b / c \longleftrightarrow a * c = b$

**proof** –

**assume** [*simp*]:  $c \neq 0$

**have**  $a = b / c \longleftrightarrow a * c = (b / c) * c$  **by** *simp*

**also have**  $\dots \longleftrightarrow a * c = b$  **by** (*simp add: divide-inverse mult-assoc*)

**finally show** *?thesis* .

**qed**

**lemma** *nonzero-divide-eq-eq*:  $c \neq 0 \implies b / c = a \longleftrightarrow b = a * c$

**proof** –

**assume** [*simp*]:  $c \neq 0$

**have**  $b / c = a \longleftrightarrow (b / c) * c = a * c$  **by** *simp*

also have ...  $\longleftrightarrow b = a * c$  **by** (*simp add: divide-inverse mult-assoc*)  
 finally show ?thesis .  
**qed**

**lemma** *divide-eq-imp*:  $c \neq 0 \implies b = a * c \implies b / c = a$   
**by** *simp*

**lemma** *eq-divide-imp*:  $c \neq 0 \implies a * c = b \implies a = b / c$   
**by** (*erule subst, simp*)

**lemmas** *field-eq-simps*[*noatp*] = *algebra-simps*

*add-divide-eq-iff divide-add-eq-iff*  
*diff-divide-eq-iff divide-diff-eq-iff*

*nonzero-eq-divide-eq nonzero-divide-eq-eq*

An example:

**lemma**  $\llbracket a \neq b; c \neq d; e \neq f \rrbracket \implies ((a-b)*(c-d)*(e-f))/((c-d)*(e-f)*(a-b)) = 1$   
**apply**(*subgoal-tac (c-d)\*(e-f)\*(a-b)  $\neq 0$* )  
**apply**(*simp add: field-eq-simps*)  
**apply**(*simp*)  
**done**

**lemma** *diff-frac-eq*:

$y \neq 0 \implies z \neq 0 \implies x / y - w / z = (x * z - w * y) / (y * z)$   
**by** (*simp add: field-eq-simps times-divide-eq*)

**lemma** *frac-eq-eq*:

$y \neq 0 \implies z \neq 0 \implies (x / y = w / z) = (x * z = w * y)$   
**by** (*simp add: field-eq-simps times-divide-eq*)

**end**

**class** *division-by-zero* = *zero* + *inverse* +  
**assumes** *inverse-zero* [*simp*]: *inverse 0 = 0*

**lemma** *divide-zero* [*simp*]:

$a / 0 = (0 :: 'a :: \{\text{field}, \text{division-by-zero}\})$   
**by** (*simp add: divide-inverse*)

**lemma** *divide-self-if* [*simp*]:

$a / (a :: 'a :: \{\text{field}, \text{division-by-zero}\}) = (\text{if } a=0 \text{ then } 0 \text{ else } 1)$   
**by** *simp*

**class** *mult-mono* = *times* + *zero* + *ord* +

**assumes** *mult-left-mono*:  $a \leq b \implies 0 \leq c \implies c * a \leq c * b$   
**assumes** *mult-right-mono*:  $a \leq b \implies 0 \leq c \implies a * c \leq b * c$

**class** *pordered-semiring* = *mult-mono* + *semiring-0* + *pordered-ab-semigroup-add*

**begin**

**lemma** *mult-mono*:

$$a \leq b \implies c \leq d \implies 0 \leq b \implies 0 \leq c \\ \implies a * c \leq b * d$$

**apply** (*erule mult-right-mono* [*THEN order-trans*], *assumption*)

**apply** (*erule mult-left-mono*, *assumption*)

**done**

**lemma** *mult-mono'*:

$$a \leq b \implies c \leq d \implies 0 \leq a \implies 0 \leq c \\ \implies a * c \leq b * d$$

**apply** (*rule mult-mono*)

**apply** (*fast intro: order-trans*) +

**done**

**end**

**class** *pordered-cancel-semiring* = *mult-mono* + *pordered-ab-semigroup-add*  
+ *semiring* + *cancel-comm-monoid-add*

**begin**

**subclass** *semiring-0-cancel* ..

**subclass** *pordered-semiring* ..

**lemma** *mult-nonneg-nonneg*:  $0 \leq a \implies 0 \leq b \implies 0 \leq a * b$

**using** *mult-left-mono* [*of zero b a*] **by** *simp*

**lemma** *mult-nonneg-nonpos*:  $0 \leq a \implies b \leq 0 \implies a * b \leq 0$

**using** *mult-left-mono* [*of b zero a*] **by** *simp*

**lemma** *mult-nonpos-nonneg*:  $a \leq 0 \implies 0 \leq b \implies a * b \leq 0$

**using** *mult-right-mono* [*of a zero b*] **by** *simp*

Legacy - use *mult-nonpos-nonneg*

**lemma** *mult-nonneg-nonpos2*:  $0 \leq a \implies b \leq 0 \implies b * a \leq 0$

**by** (*drule mult-right-mono* [*of b zero*], *auto*)

**lemma** *split-mult-neg-le*:  $(0 \leq a \ \& \ b \leq 0) \mid (a \leq 0 \ \& \ 0 \leq b) \implies a * b \leq 0$

**by** (*auto simp add: mult-nonneg-nonpos mult-nonneg-nonpos2*)

**end**

**class** *ordered-semiring* = *semiring* + *comm-monoid-add* + *ordered-cancel-ab-semigroup-add*  
+ *mult-mono*

**begin**



**subclass** *pordered-cancel-semiring* ..

**subclass** *pordered-comm-monoid-add* ..

**lemma** *mult-left-less-imp-less*:

$$c * a < c * b \implies 0 \leq c \implies a < b$$

**by** (*force simp add: mult-left-mono not-le [symmetric]*)

**lemma** *mult-right-less-imp-less*:

$$a * c < b * c \implies 0 \leq c \implies a < b$$

**by** (*force simp add: mult-right-mono not-le [symmetric]*)

**end**

**class** *ordered-semiring-strict* = *semiring* + *comm-monoid-add* + *ordered-cancel-ab-semigroup-add* +

**assumes** *mult-strict-left-mono*:  $a < b \implies 0 < c \implies c * a < c * b$

**assumes** *mult-strict-right-mono*:  $a < b \implies 0 < c \implies a * c < b * c$

**begin**

**subclass** *semiring-0-cancel* ..

**subclass** *ordered-semiring*

**proof**

**fix**  $a\ b\ c :: 'a$

**assume**  $A$ :  $a \leq b\ 0 \leq c$

**from**  $A$  **show**  $c * a \leq c * b$

**unfolding** *le-less*

**using** *mult-strict-left-mono* **by** (*cases c = 0*) *auto*

**from**  $A$  **show**  $a * c \leq b * c$

**unfolding** *le-less*

**using** *mult-strict-right-mono* **by** (*cases c = 0*) *auto*

**qed**

**lemma** *mult-left-le-imp-le*:

$$c * a \leq c * b \implies 0 < c \implies a \leq b$$

**by** (*force simp add: mult-strict-left-mono -not-less [symmetric]*)

**lemma** *mult-right-le-imp-le*:

$$a * c \leq b * c \implies 0 < c \implies a \leq b$$

**by** (*force simp add: mult-strict-right-mono not-less [symmetric]*)

**lemma** *mult-pos-pos*:  $0 < a \implies 0 < b \implies 0 < a * b$

**using** *mult-strict-left-mono* [*of zero b a*] **by** *simp*

**lemma** *mult-pos-neg*:  $0 < a \implies b < 0 \implies a * b < 0$

**using** *mult-strict-left-mono* [*of b zero a*] **by** *simp*

**lemma** *mult-neg-pos*:  $a < 0 \implies 0 < b \implies a * b < 0$

**using** *mult-strict-right-mono* [of *a zero b*] **by** *simp*

Legacy - use *mult-neg-pos*

**lemma** *mult-pos-neg2*:  $0 < a \implies b < 0 \implies b * a < 0$   
**by** (*drule mult-strict-right-mono* [of *b zero*], *auto*)

**lemma** *zero-less-mult-pos*:  
 $0 < a * b \implies 0 < a \implies 0 < b$   
**apply** (*cases b≤0*)  
**apply** (*auto simp add: le-less not-less*)  
**apply** (*drule-tac mult-pos-neg* [of *a b*])  
**apply** (*auto dest: less-not-sym*)  
**done**

**lemma** *zero-less-mult-pos2*:  
 $0 < b * a \implies 0 < a \implies 0 < b$   
**apply** (*cases b≤0*)  
**apply** (*auto simp add: le-less not-less*)  
**apply** (*drule-tac mult-pos-neg2* [of *a b*])  
**apply** (*auto dest: less-not-sym*)  
**done**

Strict monotonicity in both arguments

**lemma** *mult-strict-mono*:  
**assumes**  $a < b$  **and**  $c < d$  **and**  $0 < b$  **and**  $0 \leq c$   
**shows**  $a * c < b * d$   
**using** *assms* **apply** (*cases c=0*)  
**apply** (*simp add: mult-pos-pos*)  
**apply** (*erule mult-strict-right-mono* [THEN *less-trans*])  
**apply** (*force simp add: le-less*)  
**apply** (*erule mult-strict-left-mono, assumption*)  
**done**

This weaker variant has more natural premises

**lemma** *mult-strict-mono'*:  
**assumes**  $a < b$  **and**  $c < d$  **and**  $0 \leq a$  **and**  $0 \leq c$   
**shows**  $a * c < b * d$   
**by** (*rule mult-strict-mono*) (*insert assms, auto*)

**lemma** *mult-less-le-imp-less*:  
**assumes**  $a < b$  **and**  $c \leq d$  **and**  $0 \leq a$  **and**  $0 < c$   
**shows**  $a * c < b * d$   
**using** *assms* **apply** (*subgoal-tac a \* c < b \* c*)  
**apply** (*erule less-le-trans*)  
**apply** (*erule mult-left-mono*)  
**apply** *simp*  
**apply** (*erule mult-strict-right-mono*)  
**apply** *assumption*  
**done**

**lemma** *mult-le-less-imp-less*:

assumes  $a \leq b$  and  $c < d$  and  $0 < a$  and  $0 \leq c$   
 shows  $a * c < b * d$   
 using *assms* **apply** (*subgoal-tac*  $a * c \leq b * c$ )  
**apply** (*erule le-less-trans*)  
**apply** (*erule mult-strict-left-mono*)  
**apply** *simp*  
**apply** (*erule mult-right-mono*)  
**apply** *simp*  
**done**

**lemma** *mult-less-imp-less-left*:

assumes *less*:  $c * a < c * b$  and *nonneg*:  $0 \leq c$   
 shows  $a < b$   
**proof** (*rule ccontr*)  
 assume  $\neg a < b$   
 hence  $b \leq a$  **by** (*simp add: linorder-not-less*)  
 hence  $c * b \leq c * a$  **using** *nonneg* **by** (*rule mult-left-mono*)  
 with *this* and *less* **show** *False* **by** (*simp add: not-less [symmetric]*)  
**qed**

**lemma** *mult-less-imp-less-right*:

assumes *less*:  $a * c < b * c$  and *nonneg*:  $0 \leq c$   
 shows  $a < b$   
**proof** (*rule ccontr*)  
 assume  $\neg a < b$   
 hence  $b \leq a$  **by** (*simp add: linorder-not-less*)  
 hence  $b * c \leq a * c$  **using** *nonneg* **by** (*rule mult-right-mono*)  
 with *this* and *less* **show** *False* **by** (*simp add: not-less [symmetric]*)  
**qed**

**end**

**class** *mult-mono1* = *times* + *zero* + *ord* +  
 assumes *mult-mono1*:  $a \leq b \implies 0 \leq c \implies c * a \leq c * b$

**class** *pordered-comm-semiring* = *comm-semiring-0*  
 + *pordered-ab-semigroup-add* + *mult-mono1*  
**begin**

**subclass** *pordered-semiring*

**proof**  
 fix  $a b c :: 'a$   
 assume  $a \leq b$   $0 \leq c$   
 thus  $c * a \leq c * b$  **by** (*rule mult-mono1*)  
 thus  $a * c \leq b * c$  **by** (*simp only: mult-commute*)  
**qed**

**end**

**class** *pordered-cancel-comm-semiring* = *comm-semiring-0-cancel*  
 + *pordered-ab-semigroup-add* + *mult-mono1*  
**begin**

**subclass** *pordered-comm-semiring* ..  
**subclass** *pordered-cancel-semiring* ..

**end**

**class** *ordered-comm-semiring-strict* = *comm-semiring-0* + *ordered-cancel-ab-semigroup-add*  
 +  
**assumes** *mult-strict-left-mono-comm*:  $a < b \implies 0 < c \implies c * a < c * b$   
**begin**

**subclass** *ordered-semiring-strict*  
**proof**  
**fix**  $a\ b\ c :: 'a$   
**assume**  $a < b\ 0 < c$   
**thus**  $c * a < c * b$  **by** (*rule mult-strict-left-mono-comm*)  
**thus**  $a * c < b * c$  **by** (*simp only: mult-commute*)  
**qed**

**subclass** *pordered-cancel-comm-semiring*  
**proof**  
**fix**  $a\ b\ c :: 'a$   
**assume**  $a \leq b\ 0 \leq c$   
**thus**  $c * a \leq c * b$   
**unfolding** *le-less*  
**using** *mult-strict-left-mono* **by** (*cases c = 0*) *auto*  
**qed**

**end**

**class** *pordered-ring* = *ring* + *pordered-cancel-semiring*  
**begin**

**subclass** *pordered-ab-group-add* ..

Legacy - use *algebra-simps*

**lemmas** *ring-simps*[*noatp*] = *algebra-simps*

**lemma** *less-add-iff1*:  
 $a * e + c < b * e + d \longleftrightarrow (a - b) * e + c < d$   
**by** (*simp add: algebra-simps*)

**lemma** *less-add-iff2*:  
 $a * e + c < b * e + d \longleftrightarrow c < (b - a) * e + d$

**by** (*simp add: algebra-simps*)

**lemma** *le-add-iff1*:

$$a * e + c \leq b * e + d \longleftrightarrow (a - b) * e + c \leq d$$

**by** (*simp add: algebra-simps*)

**lemma** *le-add-iff2*:

$$a * e + c \leq b * e + d \longleftrightarrow c \leq (b - a) * e + d$$

**by** (*simp add: algebra-simps*)

**lemma** *mult-left-mono-neg*:

$$b \leq a \implies c \leq 0 \implies c * a \leq c * b$$

**apply** (*drule mult-left-mono [of - - uminus c]*)

**apply** (*simp-all add: minus-mult-left [symmetric]*)

**done**

**lemma** *mult-right-mono-neg*:

$$b \leq a \implies c \leq 0 \implies a * c \leq b * c$$

**apply** (*drule mult-right-mono [of - - uminus c]*)

**apply** (*simp-all add: minus-mult-right [symmetric]*)

**done**

**lemma** *mult-nonpos-nonpos*:  $a \leq 0 \implies b \leq 0 \implies 0 \leq a * b$

**using** *mult-right-mono-neg [of a zero b]* **by** *simp*

**lemma** *split-mult-pos-le*:

$$(0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0) \implies 0 \leq a * b$$

**by** (*auto simp add: mult-nonneg-nonneg mult-nonpos-nonpos*)

**end**

**class** *abs-if* = *minus* + *uminus* + *ord* + *zero* + *abs* +

**assumes** *abs-if*:  $|a| = (\text{if } a < 0 \text{ then } -a \text{ else } a)$

**class** *sgn-if* = *minus* + *uminus* + *zero* + *one* + *ord* + *sgn* +

**assumes** *sgn-if*:  $\text{sgn } x = (\text{if } x = 0 \text{ then } 0 \text{ else if } 0 < x \text{ then } 1 \text{ else } -1)$

**lemma** (*in sgn-if*) *sgn0*[*simp*]:  $\text{sgn } 0 = 0$

**by**(*simp add:sgn-if*)

**class** *ordered-ring* = *ring* + *ordered-semiring*

+ *ordered-ab-group-add* + *abs-if*

**begin**

**subclass** *pordered-ring* ..

**subclass** *pordered-ab-group-add-abs*

**proof**

**fix** *a b*

```

  show  $|a + b| \leq |a| + |b|$ 
by (auto simp add: abs-if not-less neg-less-eq-nonneg less-eq-neg-nonpos)
  (auto simp del: minus-add-distrib simp add: minus-add-distrib [symmetric]
    neg-less-eq-nonneg less-eq-neg-nonpos, auto intro: add-nonneg-nonneg,
    auto intro!: less-imp-le add-neg-neg)
qed (auto simp add: abs-if less-eq-neg-nonpos neg-equal-zero)

end

```

```

class ordered-ring-strict = ring + ordered-semiring-strict
  + ordered-ab-group-add + abs-if
begin

```

```

subclass ordered-ring ..

```

```

lemma mult-strict-left-mono-neg:  $b < a \implies c < 0 \implies c * a < c * b$ 
using mult-strict-left-mono [of  $b$   $a - c$ ] by simp

```

```

lemma mult-strict-right-mono-neg:  $b < a \implies c < 0 \implies a * c < b * c$ 
using mult-strict-right-mono [of  $b$   $a - c$ ] by simp

```

```

lemma mult-neg-neg:  $a < 0 \implies b < 0 \implies 0 < a * b$ 
using mult-strict-right-mono-neg [of  $a$  zero  $b$ ] by simp

```

```

subclass ring-no-zero-divisors

```

```

proof
  fix a b
  assume  $a \neq 0$  then have  $A: a < 0 \vee 0 < a$  by (simp add: neq-iff)
  assume  $b \neq 0$  then have  $B: b < 0 \vee 0 < b$  by (simp add: neq-iff)
  have  $a * b < 0 \vee 0 < a * b$ 
  proof (cases  $a < 0$ )
    case True note  $A' = this$ 
    show ?thesis proof (cases  $b < 0$ )
      case True with  $A'$ 
        show ?thesis by (auto dest: mult-neg-neg)
    next
      case False with  $B$  have  $0 < b$  by auto
      with  $A'$  show ?thesis by (auto dest: mult-strict-right-mono)
    qed
  next
    case False with  $A$  have  $A': 0 < a$  by auto
    show ?thesis proof (cases  $b < 0$ )
      case True with  $A'$ 
        show ?thesis by (auto dest: mult-strict-right-mono-neg)
    next
      case False with  $B$  have  $0 < b$  by auto
      with  $A'$  show ?thesis by (auto dest: mult-pos-pos)
    qed
  qed

```

```

qed
then show  $a * b \neq 0$  by (simp add: neq-iff)
qed

```

```

lemma zero-less-mult-iff:
 $0 < a * b \longleftrightarrow 0 < a \wedge 0 < b \vee a < 0 \wedge b < 0$ 
apply (auto simp add: mult-pos-pos mult-neg-neg)
apply (simp-all add: not-less le-less)
apply (erule disjE) apply assumption defer
apply (erule disjE) defer apply (drule sym) apply simp
apply (erule disjE) defer apply (drule sym) apply simp
apply (erule disjE) apply assumption apply (drule sym) apply simp
apply (drule sym) apply simp
apply (blast dest: zero-less-mult-pos)
apply (blast dest: zero-less-mult-pos2)
done

```

```

lemma zero-le-mult-iff:
 $0 \leq a * b \longleftrightarrow 0 \leq a \wedge 0 \leq b \vee a \leq 0 \wedge b \leq 0$ 
by (auto simp add: eq-commute [of 0] le-less not-less zero-less-mult-iff)

```

```

lemma mult-less-0-iff:
 $a * b < 0 \longleftrightarrow 0 < a \wedge b < 0 \vee a < 0 \wedge 0 < b$ 
apply (insert zero-less-mult-iff [of  $-a$   $b$ ])
apply (force simp add: minus-mult-left[symmetric])
done

```

```

lemma mult-le-0-iff:
 $a * b \leq 0 \longleftrightarrow 0 \leq a \wedge b \leq 0 \vee a \leq 0 \wedge 0 \leq b$ 
apply (insert zero-le-mult-iff [of  $-a$   $b$ ])
apply (force simp add: minus-mult-left[symmetric])
done

```

```

lemma zero-le-square [simp]:  $0 \leq a * a$ 
by (simp add: zero-le-mult-iff linear)

```

```

lemma not-square-less-zero [simp]:  $\neg (a * a < 0)$ 
by (simp add: not-less)

```

Cancellation laws for  $c * a < c * b$  and  $a * c < b * c$ , also with the relations  $\leq$  and equality.

These “disjunction” versions produce two cases when the comparison is an assumption, but effectively four when the comparison is a goal.

```

lemma mult-less-cancel-right-disj:
 $a * c < b * c \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$ 
apply (cases  $c = 0$ )
apply (auto simp add: neq-iff mult-strict-right-mono
mult-strict-right-mono-neg)

```

```

apply (auto simp add: not-less
           not-le [symmetric, of a*c]
           not-le [symmetric, of a])
apply (erule-tac [!] notE)
apply (auto simp add: less-imp-le mult-right-mono
           mult-right-mono-neg)
done

lemma mult-less-cancel-left-disj:
   $c * a < c * b \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$ 
apply (cases c = 0)
apply (auto simp add: neg-iff mult-strict-left-mono
           mult-strict-left-mono-neg)
apply (auto simp add: not-less
           not-le [symmetric, of c*a]
           not-le [symmetric, of a])
apply (erule-tac [!] notE)
apply (auto simp add: less-imp-le mult-left-mono
           mult-left-mono-neg)
done

```

The “conjunction of implication” lemmas produce two cases when the comparison is a goal, but give four when the comparison is an assumption.

```

lemma mult-less-cancel-right:
   $a * c < b * c \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$ 
using mult-less-cancel-right-disj [of a c b] by auto

```

```

lemma mult-less-cancel-left:
   $c * a < c * b \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$ 
using mult-less-cancel-left-disj [of c a b] by auto

```

```

lemma mult-le-cancel-right:
   $a * c \leq b * c \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$ 
by (simp add: not-less [symmetric] mult-less-cancel-right-disj)

```

```

lemma mult-le-cancel-left:
   $c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$ 
by (simp add: not-less [symmetric] mult-less-cancel-left-disj)

```

```

lemma mult-le-cancel-left-pos:
   $0 < c \implies c * a \leq c * b \longleftrightarrow a \leq b$ 
by (auto simp: mult-le-cancel-left)

```

```

lemma mult-le-cancel-left-neg:
   $c < 0 \implies c * a \leq c * b \longleftrightarrow b \leq a$ 
by (auto simp: mult-le-cancel-left)

```

```

lemma mult-less-cancel-left-pos:
   $0 < c \implies c * a < c * b \longleftrightarrow a < b$ 

```



**by** (*auto simp: mult-less-cancel-left*)

**lemma** *mult-less-cancel-left-neg*:

$c < 0 \implies c * a < c * b \longleftrightarrow b < a$

**by** (*auto simp: mult-less-cancel-left*)

**end**

Legacy - use *algebra-simps*

**lemmas** *ring-simps*[*noatp*] = *algebra-simps*

**lemmas** *mult-sign-intros* =

*mult-nonneg-nonneg mult-nonneg-nonpos*

*mult-nonpos-nonneg mult-nonpos-nonpos*

*mult-pos-pos mult-pos-neg*

*mult-neg-pos mult-neg-neg*

**class** *pordered-comm-ring* = *comm-ring* + *pordered-comm-semiring*

**begin**

**subclass** *pordered-ring* ..

**subclass** *pordered-cancel-comm-semiring* ..

**end**

**class** *ordered-semidom* = *comm-semiring-1-cancel* + *ordered-comm-semiring-strict*

+

**assumes** *zero-less-one* [*simp*]:  $0 < 1$

**begin**

**lemma** *pos-add-strict*:

**shows**  $0 < a \implies b < c \implies b < a + c$

**using** *add-strict-mono* [*of zero a b c*] **by** *simp*

**lemma** *zero-le-one* [*simp*]:  $0 \leq 1$

**by** (*rule zero-less-one* [*THEN less-imp-le*])

**lemma** *not-one-le-zero* [*simp*]:  $\neg 1 \leq 0$

**by** (*simp add: not-le*)

**lemma** *not-one-less-zero* [*simp*]:  $\neg 1 < 0$

**by** (*simp add: not-less*)

**lemma** *less-1-mult*:

**assumes**  $1 < m$  **and**  $1 < n$

**shows**  $1 < m * n$

**using** *assms mult-strict-mono* [*of 1 m 1 n*]

**by** (*simp add: less-trans* [*OF zero-less-one*])

**end**

**class** *ordered-idom* = *comm-ring-1* +  
*ordered-comm-semiring-strict* + *ordered-ab-group-add* +  
*abs-if* + *sgn-if*

**begin**

**subclass** *ordered-ring-strict* ..  
**subclass** *pordered-comm-ring* ..  
**subclass** *idom* ..

**subclass** *ordered-semidom*

**proof**

**have**  $0 \leq 1 * 1$  **by** (*rule zero-le-square*)

**thus**  $0 < 1$  **by** (*simp add: le-less*)

**qed**

**lemma** *linorder-neqE-ordered-idom*:

**assumes**  $x \neq y$  **obtains**  $x < y \mid y < x$

**using** *assms* **by** (*rule neqE*)

These cancellation simprules also produce two cases when the comparison is a goal.

**lemma** *mult-le-cancel-right1*:

$c \leq b * c \iff (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$

**by** (*insert mult-le-cancel-right [of 1 c b], simp*)

**lemma** *mult-le-cancel-right2*:

$a * c \leq c \iff (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$

**by** (*insert mult-le-cancel-right [of a c 1], simp*)

**lemma** *mult-le-cancel-left1*:

$c \leq c * b \iff (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$

**by** (*insert mult-le-cancel-left [of c 1 b], simp*)

**lemma** *mult-le-cancel-left2*:

$c * a \leq c \iff (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$

**by** (*insert mult-le-cancel-left [of c a 1], simp*)

**lemma** *mult-less-cancel-right1*:

$c < b * c \iff (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$

**by** (*insert mult-less-cancel-right [of 1 c b], simp*)

**lemma** *mult-less-cancel-right2*:

$a * c < c \iff (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$

**by** (*insert mult-less-cancel-right [of a c 1], simp*)

**lemma** *mult-less-cancel-left1*:

$$c < c * b \iff (0 \leq c \implies 1 < b) \wedge (c \leq 0 \implies b < 1)$$

**by** (*insert mult-less-cancel-left [of c 1 b], simp*)

**lemma** *mult-less-cancel-left2*:

$$c * a < c \iff (0 \leq c \implies a < 1) \wedge (c \leq 0 \implies 1 < a)$$

**by** (*insert mult-less-cancel-left [of c a 1], simp*)

**lemma** *sgn-sgn [simp]*:

$$\text{sgn} (\text{sgn } a) = \text{sgn } a$$

**unfolding** *sgn-if* **by** *simp*

**lemma** *sgn-0-0*:

$$\text{sgn } a = 0 \iff a = 0$$

**unfolding** *sgn-if* **by** *simp*

**lemma** *sgn-1-pos*:

$$\text{sgn } a = 1 \iff a > 0$$

**unfolding** *sgn-if* **by** (*simp add: neg-equal-zero*)

**lemma** *sgn-1-neg*:

$$\text{sgn } a = -1 \iff a < 0$$

**unfolding** *sgn-if* **by** (*auto simp add: equal-neg-zero*)

**lemma** *sgn-pos [simp]*:

$$0 < a \implies \text{sgn } a = 1$$

**unfolding** *sgn-1-pos* .

**lemma** *sgn-neg [simp]*:

$$a < 0 \implies \text{sgn } a = -1$$

**unfolding** *sgn-1-neg* .

**lemma** *sgn-times*:

$$\text{sgn } (a * b) = \text{sgn } a * \text{sgn } b$$

**by** (*auto simp add: sgn-if zero-less-mult-iff*)

**lemma** *abs-sgn*:  $\text{abs } k = k * \text{sgn } k$

**unfolding** *sgn-if abs-if* **by** *auto*

**lemma** *sgn-greater [simp]*:

$$0 < \text{sgn } a \iff 0 < a$$

**unfolding** *sgn-if* **by** *auto*

**lemma** *sgn-less [simp]*:

$$\text{sgn } a < 0 \iff a < 0$$

**unfolding** *sgn-if* **by** *auto*

**lemma** *abs-dvd-iff [simp]*:  $(\text{abs } m) \text{ dvd } k \iff m \text{ dvd } k$

**by** (*simp add: abs-if*)

**lemma** *dvd-abs-iff* [simp]:  $m \text{ dvd } (\text{abs } k) \longleftrightarrow m \text{ dvd } k$   
**by** (simp add: abs-if)

**end**

**class** *ordered-field* = *field* + *ordered-idom*

Simprules for comparisons where common factors can be cancelled.

**lemmas** *mult-compare-simps*[noatp] =  
*mult-le-cancel-right mult-le-cancel-left*  
*mult-le-cancel-right1 mult-le-cancel-right2*  
*mult-le-cancel-left1 mult-le-cancel-left2*  
*mult-less-cancel-right mult-less-cancel-left*  
*mult-less-cancel-right1 mult-less-cancel-right2*  
*mult-less-cancel-left1 mult-less-cancel-left2*  
*mult-cancel-right mult-cancel-left*  
*mult-cancel-right1 mult-cancel-right2*  
*mult-cancel-left1 mult-cancel-left2*

— FIXME continue localization here

**lemma** *inverse-nonzero-iff-nonzero* [simp]:  
 $(\text{inverse } a = 0) = (a = (0::'a::\{\text{division-ring}, \text{division-by-zero}\}))$   
**by** (force dest: *inverse-zero-imp-zero*)

**lemma** *inverse-minus-eq* [simp]:  
 $\text{inverse}(-a) = -\text{inverse}(a::'a::\{\text{division-ring}, \text{division-by-zero}\})$

**proof** cases

**assume**  $a=0$  **thus** ?thesis **by** (simp add: *inverse-zero*)

**next**

**assume**  $a \neq 0$

**thus** ?thesis **by** (simp add: *nonzero-inverse-minus-eq*)

**qed**

**lemma** *inverse-eq-imp-eq*:  
 $\text{inverse } a = \text{inverse } b \implies a = (b::'a::\{\text{division-ring}, \text{division-by-zero}\})$   
**apply** (cases  $a=0 \mid b=0$ )  
**apply** (force dest!: *inverse-zero-imp-zero*  
simp add: *eq-commute* [of  $0::'a$ ])  
**apply** (force dest!: *nonzero-inverse-eq-imp-eq*)  
**done**

**lemma** *inverse-eq-iff-eq* [simp]:  
 $(\text{inverse } a = \text{inverse } b) = (a = (b::'a::\{\text{division-ring}, \text{division-by-zero}\}))$   
**by** (force dest!: *inverse-eq-imp-eq*)

**lemma** *inverse-inverse-eq* [simp]:  
 $\text{inverse}(\text{inverse } (a::'a::\{\text{division-ring}, \text{division-by-zero}\})) = a$

```

proof cases
  assume  $a=0$  thus ?thesis by simp
next
  assume  $a \neq 0$ 
  thus ?thesis by (simp add: nonzero-inverse-inverse-eq)
qed

```

This version builds in division by zero while also re-orienting the right-hand side.

```

lemma inverse-mult-distrib [simp]:
   $\text{inverse}(a*b) = \text{inverse}(a) * \text{inverse}(b::'a::\{\text{field}, \text{division-by-zero}\})$ 
proof cases
  assume  $a \neq 0 \ \& \ b \neq 0$ 
  thus ?thesis by (simp add: nonzero-inverse-mult-distrib mult-commute)
next
  assume  $\sim (a \neq 0 \ \& \ b \neq 0)$ 
  thus ?thesis by force
qed

```

```

lemma inverse-divide [simp]:
   $\text{inverse}(a/b) = b / (a::'a::\{\text{field}, \text{division-by-zero}\})$ 
by (simp add: divide-inverse mult-commute)

```

### 11.1 Calculations with fractions

There is a whole bunch of simp-rules just for class *field* but none for class *field* and *nonzero-divides* because the latter are covered by a simproc.

```

lemma mult-divide-mult-cancel-left:
   $c \neq 0 \implies (c*a) / (c*b) = a / (b::'a::\{\text{field}, \text{division-by-zero}\})$ 
apply (cases b = 0)
apply (simp-all add: nonzero-mult-divide-mult-cancel-left)
done

```

```

lemma mult-divide-mult-cancel-right:
   $c \neq 0 \implies (a*c) / (b*c) = a / (b::'a::\{\text{field}, \text{division-by-zero}\})$ 
apply (cases b = 0)
apply (simp-all add: nonzero-mult-divide-mult-cancel-right)
done

```

```

lemma divide-divide-eq-right [simp, noatp]:
   $a / (b/c) = (a*c) / (b::'a::\{\text{field}, \text{division-by-zero}\})$ 
by (simp add: divide-inverse mult-ac)

```

```

lemma divide-divide-eq-left [simp, noatp]:
   $(a / b) / (c::'a::\{\text{field}, \text{division-by-zero}\}) = a / (b*c)$ 
by (simp add: divide-inverse mult-assoc)

```

### 11.1.1 Special Cancellation Simprules for Division

**lemma** *mult-divide-mult-cancel-left-if* [*simp*, *noatp*]:  
**fixes**  $c :: 'a :: \{\text{field}, \text{division-by-zero}\}$   
**shows**  $(c*a) / (c*b) = (\text{if } c=0 \text{ then } 0 \text{ else } a/b)$   
**by** (*simp add: mult-divide-mult-cancel-left*)

### 11.2 Division and Unary Minus

**lemma** *minus-divide-right*:  $-(a/b) = a / -(b::'a::\{\text{field}, \text{division-by-zero}\})$   
**by** (*simp add: divide-inverse*)

**lemma** *divide-minus-right* [*simp*, *noatp*]:  
 $a / -(b::'a::\{\text{field}, \text{division-by-zero}\}) = -(a / b)$   
**by** (*simp add: divide-inverse*)

**lemma** *minus-divide-divide*:  
 $(-a)/(-b) = a / (b::'a::\{\text{field}, \text{division-by-zero}\})$   
**apply** (*cases b=0, simp*)  
**apply** (*simp add: nonzero-minus-divide-divide*)  
**done**

**lemma** *eq-divide-eq*:  
 $((a::'a::\{\text{field}, \text{division-by-zero}\}) = b/c) = (\text{if } c \neq 0 \text{ then } a*c = b \text{ else } a=0)$   
**by** (*simp add: nonzero-eq-divide-eq*)

**lemma** *divide-eq-eq*:  
 $(b/c = (a::'a::\{\text{field}, \text{division-by-zero}\})) = (\text{if } c \neq 0 \text{ then } b = a*c \text{ else } a=0)$   
**by** (*force simp add: nonzero-divide-eq-eq*)

### 11.3 Ordered Fields

**lemma** *positive-imp-inverse-positive*:  
**assumes**  $a\text{-gt-}0: 0 < a$  **shows**  $0 < \text{inverse } a$  ( $a::'a::\text{ordered-field}$ )  
**proof** –  
**have**  $0 < a * \text{inverse } a$   
**by** (*simp add: a-gt-0 [THEN order-less-imp-not-eq2] zero-less-one*)  
**thus**  $0 < \text{inverse } a$   
**by** (*simp add: a-gt-0 [THEN order-less-not-sym] zero-less-mult-iff*)  
**qed**

**lemma** *negative-imp-inverse-negative*:  
 $a < 0 ==> \text{inverse } a < (0::'a::\text{ordered-field})$   
**by** (*insert positive-imp-inverse-positive [of  $-a$ ],*  
*simp add: nonzero-inverse-minus-eq order-less-imp-not-eq*)

**lemma** *inverse-le-imp-le*:  
**assumes** *invle*:  $\text{inverse } a \leq \text{inverse } b$  **and** *apos*:  $0 < a$   
**shows**  $b \leq (a::'a::\text{ordered-field})$   
**proof** (*rule classical*)

```

assume  $\sim b \leq a$ 
hence  $a < b$  by (simp add: linorder-not-le)
hence  $bpos: 0 < b$  by (blast intro: apos order-less-trans)
hence  $a * \text{inverse } a \leq a * \text{inverse } b$ 
by (simp add: apos invle order-less-imp-le mult-left-mono)
hence  $(a * \text{inverse } a) * b \leq (a * \text{inverse } b) * b$ 
by (simp add: bpos order-less-imp-le mult-right-mono)
thus  $b \leq a$  by (simp add: mult-assoc apos bpos order-less-imp-not-eq2)
qed

```

```

lemma inverse-positive-imp-positive:
assumes inv-gt-0:  $0 < \text{inverse } a$  and nz:  $a \neq 0$ 
shows  $0 < (a::'a::\text{ordered-field})$ 
proof –
  have  $0 < \text{inverse } (\text{inverse } a)$ 
  using inv-gt-0 by (rule positive-imp-inverse-positive)
  thus  $0 < a$ 
  using nz by (simp add: nonzero-inverse-inverse-eq)
qed

```

```

lemma inverse-positive-iff-positive [simp]:
   $(0 < \text{inverse } a) = (0 < (a::'a::\{\text{ordered-field}, \text{division-by-zero}\}))$ 
apply (cases  $a = 0$ , simp)
apply (blast intro: inverse-positive-imp-positive positive-imp-inverse-positive)
done

```

```

lemma inverse-negative-imp-negative:
assumes inv-less-0:  $\text{inverse } a < 0$  and nz:  $a \neq 0$ 
shows  $a < (0::'a::\text{ordered-field})$ 
proof –
  have  $\text{inverse } (\text{inverse } a) < 0$ 
  using inv-less-0 by (rule negative-imp-inverse-negative)
  thus  $a < 0$  using nz by (simp add: nonzero-inverse-inverse-eq)
qed

```

```

lemma inverse-negative-iff-negative [simp]:
   $(\text{inverse } a < 0) = (a < (0::'a::\{\text{ordered-field}, \text{division-by-zero}\}))$ 
apply (cases  $a = 0$ , simp)
apply (blast intro: inverse-negative-imp-negative negative-imp-inverse-negative)
done

```

```

lemma inverse-nonnegative-iff-nonnegative [simp]:
   $(0 \leq \text{inverse } a) = (0 \leq (a::'a::\{\text{ordered-field}, \text{division-by-zero}\}))$ 
by (simp add: linorder-not-less [symmetric])

```

```

lemma inverse-nonpositive-iff-nonpositive [simp]:
   $(\text{inverse } a \leq 0) = (a \leq (0::'a::\{\text{ordered-field}, \text{division-by-zero}\}))$ 
by (simp add: linorder-not-less [symmetric])

```

**lemma** *ordered-field-no-lb*:  $\forall x. \exists y. y < (x::'a::\text{ordered-field})$

**proof**

fix  $x::'a$   
 have  $m1: -(1::'a) < 0$  **by** *simp*  
 from *add-strict-right-mono*[*OF*  $m1$ , **where**  $c=x$ ]  
 have  $(-1) + x < x$  **by** *simp*  
 thus  $\exists y. y < x$  **by** *blast*

**qed**

**lemma** *ordered-field-no-ub*:  $\forall x. \exists y. y > (x::'a::\text{ordered-field})$

**proof**

fix  $x::'a$   
 have  $m1: (1::'a) > 0$  **by** *simp*  
 from *add-strict-right-mono*[*OF*  $m1$ , **where**  $c=x$ ]  
 have  $1 + x > x$  **by** *simp*  
 thus  $\exists y. y > x$  **by** *blast*

**qed**

## 11.4 Anti-Monotonicity of *inverse*

**lemma** *less-imp-inverse-less*:

**assumes** *less*:  $a < b$  **and** *apos*:  $0 < a$

**shows**  $\text{inverse } b < \text{inverse } (a::'a::\text{ordered-field})$

**proof** (*rule ccontr*)

assume  $\sim \text{inverse } b < \text{inverse } a$   
 hence  $\text{inverse } a \leq \text{inverse } b$  **by** (*simp add: linorder-not-less*)  
 hence  $\sim (a < b)$   
**by** (*simp add: linorder-not-less inverse-le-imp-le [OF - apos]*)  
 thus *False* **by** (*rule notE [OF - less]*)

**qed**

**lemma** *inverse-less-imp-less*:

$[\text{inverse } a < \text{inverse } b; 0 < a] \implies b < (a::'a::\text{ordered-field})$

**apply** (*simp add: order-less-le [of inverse a] order-less-le [of b]*)

**apply** (*force dest!: inverse-le-imp-le nonzero-inverse-eq-imp-eq*)

**done**

Both premises are essential. Consider -1 and 1.

**lemma** *inverse-less-iff-less* [*simp, noatp*]:

$[\text{inverse } a < \text{inverse } b] \implies (b < (a::'a::\text{ordered-field}))$

**by** (*blast intro: less-imp-inverse-less dest: inverse-less-imp-less*)

**lemma** *le-imp-inverse-le*:

$[a \leq b; 0 < a] \implies \text{inverse } b \leq \text{inverse } (a::'a::\text{ordered-field})$

**by** (*force simp add: order-le-less less-imp-inverse-less*)

**lemma** *inverse-le-iff-le* [*simp, noatp*]:

$[\text{inverse } a \leq \text{inverse } b] \implies (b \leq (a::'a::\text{ordered-field}))$

**by** (*blast intro: le-imp-inverse-le dest: inverse-le-imp-le*)



These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

**lemma** *inverse-le-imp-le-neg*:

```
[[inverse a ≤ inverse b; b < 0]] ==> b ≤ (a::'a::ordered-field)
apply (rule classical)
apply (subgoal-tac a < 0)
prefer 2 apply (force simp add: linorder-not-le intro: order-less-trans)
apply (insert inverse-le-imp-le [of -b -a])
apply (simp add: order-less-imp-not-eq nonzero-inverse-minus-eq)
done
```

**lemma** *less-imp-inverse-less-neg*:

```
[[a < b; b < 0]] ==> inverse b < inverse (a::'a::ordered-field)
apply (subgoal-tac a < 0)
prefer 2 apply (blast intro: order-less-trans)
apply (insert less-imp-inverse-less [of -b -a])
apply (simp add: order-less-imp-not-eq nonzero-inverse-minus-eq)
done
```

**lemma** *inverse-less-imp-less-neg*:

```
[[inverse a < inverse b; b < 0]] ==> b < (a::'a::ordered-field)
apply (rule classical)
apply (subgoal-tac a < 0)
prefer 2
apply (force simp add: linorder-not-less intro: order-le-less-trans)
apply (insert inverse-less-imp-less [of -b -a])
apply (simp add: order-less-imp-not-eq nonzero-inverse-minus-eq)
done
```

**lemma** *inverse-less-iff-less-neg* [simp,noatp]:

```
[[a < 0; b < 0]] ==> (inverse a < inverse b) = (b < (a::'a::ordered-field))
apply (insert inverse-less-iff-less [of -b -a])
apply (simp del: inverse-less-iff-less
      add: order-less-imp-not-eq nonzero-inverse-minus-eq)
done
```

**lemma** *le-imp-inverse-le-neg*:

```
[[a ≤ b; b < 0]] ==> inverse b ≤ inverse (a::'a::ordered-field)
by (force simp add: order-le-less less-imp-inverse-less-neg)
```

**lemma** *inverse-le-iff-le-neg* [simp,noatp]:

```
[[a < 0; b < 0]] ==> (inverse a ≤ inverse b) = (b ≤ (a::'a::ordered-field))
by (blast intro: le-imp-inverse-le-neg dest: inverse-le-imp-le-neg)
```

## 11.5 Inverses and the Number One

**lemma** *one-less-inverse-iff*:

```
(1 < inverse x) = (0 < x & x < (1::'a::{ordered-field,division-by-zero}))
proof cases
```

```

assume  $0 < x$ 
  with inverse-less-iff-less [OF zero-less-one, of  $x$ ]
    show ?thesis by simp
next
  assume notless:  $\sim (0 < x)$ 
  have  $\sim (1 < \text{inverse } x)$ 
  proof
    assume  $1 < \text{inverse } x$ 
    also with notless have  $\dots \leq 0$  by (simp add: linorder-not-less)
    also have  $\dots < 1$  by (rule zero-less-one)
    finally show False by auto
  qed
with notless show ?thesis by simp
qed

```

```

lemma inverse-eq-1-iff [simp]:
   $(\text{inverse } x = 1) = (x = (1::'a::\{\text{field}, \text{division-by-zero}\}))$ 
by (insert inverse-eq-iff-eq [of  $x$   $1$ ], simp)

```

```

lemma one-le-inverse-iff:
   $(1 \leq \text{inverse } x) = (0 < x \ \& \ x \leq (1::'a::\{\text{ordered-field}, \text{division-by-zero}\}))$ 
by (force simp add: order-le-less one-less-inverse-iff zero-less-one
      eq-commute [of  $1$ ])

```

```

lemma inverse-less-1-iff:
   $(\text{inverse } x < 1) = (x \leq 0 \mid 1 < (x::'a::\{\text{ordered-field}, \text{division-by-zero}\}))$ 
by (simp add: linorder-not-le [symmetric] one-le-inverse-iff)

```

```

lemma inverse-le-1-iff:
   $(\text{inverse } x \leq 1) = (x \leq 0 \mid 1 \leq (x::'a::\{\text{ordered-field}, \text{division-by-zero}\}))$ 
by (simp add: linorder-not-less [symmetric] one-less-inverse-iff)

```

## 11.6 Simplification of Inequalities Involving Literal Divisors

```

lemma pos-le-divide-eq:  $0 < (c::'a::\text{ordered-field}) \implies (a \leq b/c) = (a*c \leq b)$ 

```

```

proof –
  assume less:  $0 < c$ 
  hence  $(a \leq b/c) = (a*c \leq (b/c)*c)$ 
    by (simp add: mult-le-cancel-right order-less-not-sym [OF less])
  also have  $\dots = (a*c \leq b)$ 
    by (simp add: order-less-imp-not-eq2 [OF less] divide-inverse mult-assoc)
  finally show ?thesis .
qed

```

```

lemma neg-le-divide-eq:  $c < (0::'a::\text{ordered-field}) \implies (a \leq b/c) = (b \leq a*c)$ 

```

```

proof –
  assume less:  $c < 0$ 
  hence  $(a \leq b/c) = ((b/c)*c \leq a*c)$ 
    by (simp add: mult-le-cancel-right order-less-not-sym [OF less])

```

also have ... = ( $b \leq a*c$ )  
 by (simp add: order-less-imp-not-eq [OF less] divide-inverse mult-assoc)  
 finally show ?thesis .  
 qed

lemma le-divide-eq:

( $a \leq b/c$ ) =  
 (if  $0 < c$  then  $a*c \leq b$   
   else if  $c < 0$  then  $b \leq a*c$   
   else  $a \leq (0::'a::\{\text{ordered-field}, \text{division-by-zero}\})$ )

apply (cases c=0, simp)

apply (force simp add: pos-le-divide-eq neg-le-divide-eq linorder-neq-iff)

done

lemma pos-divide-le-eq:  $0 < (c::'a::\text{ordered-field}) \implies (b/c \leq a) = (b \leq a*c)$

proof -

assume less:  $0 < c$

hence ( $b/c \leq a$ ) = ( $(b/c)*c \leq a*c$ )

by (simp add: mult-le-cancel-right order-less-not-sym [OF less])

also have ... = ( $b \leq a*c$ )

by (simp add: order-less-imp-not-eq2 [OF less] divide-inverse mult-assoc)

finally show ?thesis .

qed

lemma neg-divide-le-eq:  $c < (0::'a::\text{ordered-field}) \implies (b/c \leq a) = (a*c \leq b)$

proof -

assume less:  $c < 0$

hence ( $b/c \leq a$ ) = ( $a*c \leq (b/c)*c$ )

by (simp add: mult-le-cancel-right order-less-not-sym [OF less])

also have ... = ( $a*c \leq b$ )

by (simp add: order-less-imp-not-eq [OF less] divide-inverse mult-assoc)

finally show ?thesis .

qed

lemma divide-le-eq:

( $b/c \leq a$ ) =  
 (if  $0 < c$  then  $b \leq a*c$   
   else if  $c < 0$  then  $a*c \leq b$   
   else  $0 \leq (a::'a::\{\text{ordered-field}, \text{division-by-zero}\})$ )

apply (cases c=0, simp)

apply (force simp add: pos-divide-le-eq neg-divide-le-eq linorder-neq-iff)

done

lemma pos-less-divide-eq:

$0 < (c::'a::\text{ordered-field}) \implies (a < b/c) = (a*c < b)$

proof -

assume less:  $0 < c$

hence ( $a < b/c$ ) = ( $a*c < (b/c)*c$ )

by (simp add: mult-less-cancel-right-disj order-less-not-sym [OF less])

also have ... = ( $a * c < b$ )  
 by (simp add: order-less-imp-not-eq2 [OF less] divide-inverse mult-assoc)  
 finally show ?thesis .  
 qed

lemma neg-less-divide-eq:  
 $c < (0::'a::ordered-field) ==> (a < b/c) = (b < a * c)$   
 proof -  
 assume less:  $c < 0$   
 hence  $(a < b/c) = ((b/c) * c < a * c)$   
 by (simp add: mult-less-cancel-right-disj order-less-not-sym [OF less])  
 also have ... = ( $b < a * c$ )  
 by (simp add: order-less-imp-not-eq [OF less] divide-inverse mult-assoc)  
 finally show ?thesis .  
 qed

lemma less-divide-eq:  
 $(a < b/c) =$   
 $(if\ 0 < c\ then\ a * c < b$   
 $\quad\quad\quad else\ if\ c < 0\ then\ b < a * c$   
 $\quad\quad\quad else\ a < (0::'a::\{ordered-field, division-by-zero\}))$   
 apply (cases  $c=0$ , simp)  
 apply (force simp add: pos-less-divide-eq neg-less-divide-eq linorder-neq-iff)  
 done

lemma pos-divide-less-eq:  
 $0 < (c::'a::ordered-field) ==> (b/c < a) = (b < a * c)$   
 proof -  
 assume less:  $0 < c$   
 hence  $(b/c < a) = ((b/c) * c < a * c)$   
 by (simp add: mult-less-cancel-right-disj order-less-not-sym [OF less])  
 also have ... = ( $b < a * c$ )  
 by (simp add: order-less-imp-not-eq2 [OF less] divide-inverse mult-assoc)  
 finally show ?thesis .  
 qed

lemma neg-divide-less-eq:  
 $c < (0::'a::ordered-field) ==> (b/c < a) = (a * c < b)$   
 proof -  
 assume less:  $c < 0$   
 hence  $(b/c < a) = (a * c < (b/c) * c)$   
 by (simp add: mult-less-cancel-right-disj order-less-not-sym [OF less])  
 also have ... = ( $a * c < b$ )  
 by (simp add: order-less-imp-not-eq [OF less] divide-inverse mult-assoc)  
 finally show ?thesis .  
 qed

lemma divide-less-eq:  
 $(b/c < a) =$

```

    (if 0 < c then b < a*c
      else if c < 0 then a*c < b
      else 0 < (a::'a::{ordered-field,division-by-zero}))
  apply (cases c=0, simp)
  apply (force simp add: pos-divide-less-eq neg-divide-less-eq linorder-neq-iff)
done

```

### 11.7 Field simplification

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

**lemmas** *field-simps*[noatp] = *field-eq-simps*

```

pos-divide-less-eq neg-divide-less-eq
pos-less-divide-eq neg-less-divide-eq
pos-divide-le-eq neg-divide-le-eq
pos-le-divide-eq neg-le-divide-eq

```

Lemmas *sign-simps* is a first attempt to automate proofs of positivity/negativity needed for *field-simps*. Have not added *sign-simps* to *field-simps* because the former can lead to case explosions.

**lemmas** *sign-simps*[noatp] = *group-simps*  
*zero-less-mult-iff mult-less-0-iff*

### 11.8 Division and Signs

**lemma** *zero-less-divide-iff*:

$((0::'a::{ordered-field,division-by-zero}) < a/b) = (0 < a \ \& \ 0 < b \mid a < 0 \ \& \ b < 0)$

**by** (*simp add: divide-inverse zero-less-mult-iff*)

**lemma** *divide-less-0-iff*:

$(a/b < (0::'a::{ordered-field,division-by-zero})) = (0 < a \ \& \ b < 0 \mid a < 0 \ \& \ 0 < b)$

**by** (*simp add: divide-inverse mult-less-0-iff*)

**lemma** *zero-le-divide-iff*:

$((0::'a::{ordered-field,division-by-zero}) \leq a/b) = (0 \leq a \ \& \ 0 \leq b \mid a \leq 0 \ \& \ b \leq 0)$

**by** (*simp add: divide-inverse zero-le-mult-iff*)

**lemma** *divide-le-0-iff*:

$(a/b \leq (0::'a::{ordered-field,division-by-zero})) = (0 \leq a \ \& \ b \leq 0 \mid a \leq 0 \ \& \ 0 \leq b)$

**by** (*simp add: divide-inverse mult-le-0-iff*)

**lemma** *divide-eq-0-iff* [simp,noatp]:  
 $(a/b = 0) = (a=0 \mid b=(0::'a::\{\text{field}, \text{division-by-zero}\}))$   
**by** (simp add: divide-inverse)

**lemma** *divide-pos-pos*:  
 $0 < (x::'a::\text{ordered-field}) \implies 0 < y \implies 0 < x / y$   
**by**(simp add:field-simps)

**lemma** *divide-nonneg-pos*:  
 $0 \leq (x::'a::\text{ordered-field}) \implies 0 < y \implies 0 \leq x / y$   
**by**(simp add:field-simps)

**lemma** *divide-neg-pos*:  
 $(x::'a::\text{ordered-field}) < 0 \implies 0 < y \implies x / y < 0$   
**by**(simp add:field-simps)

**lemma** *divide-nonpos-pos*:  
 $(x::'a::\text{ordered-field}) \leq 0 \implies 0 < y \implies x / y \leq 0$   
**by**(simp add:field-simps)

**lemma** *divide-pos-neg*:  
 $0 < (x::'a::\text{ordered-field}) \implies y < 0 \implies x / y < 0$   
**by**(simp add:field-simps)

**lemma** *divide-nonneg-neg*:  
 $0 \leq (x::'a::\text{ordered-field}) \implies y < 0 \implies x / y \leq 0$   
**by**(simp add:field-simps)

**lemma** *divide-neg-neg*:  
 $(x::'a::\text{ordered-field}) < 0 \implies y < 0 \implies 0 < x / y$   
**by**(simp add:field-simps)

**lemma** *divide-nonpos-neg*:  
 $(x::'a::\text{ordered-field}) \leq 0 \implies y < 0 \implies 0 \leq x / y$   
**by**(simp add:field-simps)

## 11.9 Cancellation Laws for Division

**lemma** *divide-cancel-right* [simp,noatp]:  
 $(a/c = b/c) = (c = 0 \mid a = (b::'a::\{\text{field}, \text{division-by-zero}\}))$   
**apply** (cases c=0, simp)  
**apply** (simp add: divide-inverse)  
**done**

**lemma** *divide-cancel-left* [simp,noatp]:  
 $(c/a = c/b) = (c = 0 \mid a = (b::'a::\{\text{field}, \text{division-by-zero}\}))$   
**apply** (cases c=0, simp)  
**apply** (simp add: divide-inverse)

done

### 11.10 Division and the Number One

Simplify expressions equated with 1

**lemma** *divide-eq-1-iff* [simp,noatp]:  
 $(a/b = 1) = (b \neq 0 \ \& \ a = (b::'a::\{\text{field}, \text{division-by-zero}\}))$   
**apply** (cases b=0, simp)  
**apply** (simp add: right-inverse-eq)  
**done**

**lemma** *one-eq-divide-iff* [simp,noatp]:  
 $(1 = a/b) = (b \neq 0 \ \& \ a = (b::'a::\{\text{field}, \text{division-by-zero}\}))$   
**by** (simp add: eq-commute [of 1])

**lemma** *zero-eq-1-divide-iff* [simp,noatp]:  
 $((0::'a::\{\text{ordered-field}, \text{division-by-zero}\}) = 1/a) = (a = 0)$   
**apply** (cases a=0, simp)  
**apply** (auto simp add: nonzero-eq-divide-eq)  
**done**

**lemma** *one-divide-eq-0-iff* [simp,noatp]:  
 $(1/a = (0::'a::\{\text{ordered-field}, \text{division-by-zero}\})) = (a = 0)$   
**apply** (cases a=0, simp)  
**apply** (insert zero-neq-one [THEN not-sym])  
**apply** (auto simp add: nonzero-divide-eq-eq)  
**done**

Simplify expressions such as  $0 < 1/x$  to  $0 < x$

**lemmas** *zero-less-divide-1-iff* = *zero-less-divide-iff* [of 1, simplified]  
**lemmas** *divide-less-0-1-iff* = *divide-less-0-iff* [of 1, simplified]  
**lemmas** *zero-le-divide-1-iff* = *zero-le-divide-iff* [of 1, simplified]  
**lemmas** *divide-le-0-1-iff* = *divide-le-0-iff* [of 1, simplified]

**declare** *zero-less-divide-1-iff* [simp,noatp]  
**declare** *divide-less-0-1-iff* [simp,noatp]  
**declare** *zero-le-divide-1-iff* [simp,noatp]  
**declare** *divide-le-0-1-iff* [simp,noatp]

### 11.11 Ordering Rules for Division

**lemma** *divide-strict-right-mono*:  
 $[|a < b; 0 < c|] ==> a / c < b / (c::'a::\text{ordered-field})$   
**by** (simp add: order-less-imp-not-eq2 divide-inverse mult-strict-right-mono  
positive-imp-inverse-positive)

**lemma** *divide-right-mono*:  
 $[|a \leq b; 0 \leq c|] ==> a / c \leq b / (c::'a::\{\text{ordered-field}, \text{division-by-zero}\})$   
**by** (force simp add: divide-strict-right-mono order-le-less)

```

lemma divide-right-mono-neg: (a::'a::{division-by-zero,ordered-field}) <= b
  ==> c <= 0 ==> b / c <= a / c
apply (drule divide-right-mono [of - - - c])
apply auto
done

```

```

lemma divide-strict-right-mono-neg:
  [|b < a; c < 0|] ==> a / c < b / (c::'a::ordered-field)
apply (drule divide-strict-right-mono [of - - - c], simp)
apply (simp add: order-less-imp-not-eq nonzero-minus-divide-right [symmetric])
done

```

The last premise ensures that  $a$  and  $b$  have the same sign

```

lemma divide-strict-left-mono:
  [|b < a; 0 < c; 0 < a*b|] ==> c / a < c / (b::'a::ordered-field)
by(auto simp: field-simps times-divide-eq zero-less-mult-iff mult-strict-right-mono)

```

```

lemma divide-left-mono:
  [|b ≤ a; 0 ≤ c; 0 < a*b|] ==> c / a ≤ c / (b::'a::ordered-field)
by(auto simp: field-simps times-divide-eq zero-less-mult-iff mult-right-mono)

```

```

lemma divide-left-mono-neg: (a::'a::{division-by-zero,ordered-field}) <= b
  ==> c <= 0 ==> 0 < a * b ==> c / a <= c / b
  apply (drule divide-left-mono [of - - - c])
  apply (auto simp add: mult-commute)
done

```

```

lemma divide-strict-left-mono-neg:
  [|a < b; c < 0; 0 < a*b|] ==> c / a < c / (b::'a::ordered-field)
by(auto simp: field-simps times-divide-eq zero-less-mult-iff mult-strict-right-mono-neg)

```

Simplify quotients that are compared with the value 1.

```

lemma le-divide-eq-1 [noatp]:
  fixes a :: 'a :: {ordered-field,division-by-zero}
  shows (1 ≤ b / a) = ((0 < a & a ≤ b) | (a < 0 & b ≤ a))
by (auto simp add: le-divide-eq)

```

```

lemma divide-le-eq-1 [noatp]:
  fixes a :: 'a :: {ordered-field,division-by-zero}
  shows (b / a ≤ 1) = ((0 < a & b ≤ a) | (a < 0 & a ≤ b) | a=0)
by (auto simp add: divide-le-eq)

```

```

lemma less-divide-eq-1 [noatp]:
  fixes a :: 'a :: {ordered-field,division-by-zero}
  shows (1 < b / a) = ((0 < a & a < b) | (a < 0 & b < a))
by (auto simp add: less-divide-eq)

```

```

lemma divide-less-eq-1 [noatp]:

```



**fixes**  $a :: 'a :: \{\text{ordered-field}, \text{division-by-zero}\}$   
**shows**  $(b / a < 1) = ((0 < a \ \& \ b < a) \mid (a < 0 \ \& \ a < b) \mid a=0)$   
**by**  $(\text{auto simp add: divide-less-eq})$

### 11.12 Conditional Simplification Rules: No Case Splits

**lemma** *le-divide-eq-1-pos* [*simp, noatp*]:  
**fixes**  $a :: 'a :: \{\text{ordered-field}, \text{division-by-zero}\}$   
**shows**  $0 < a \implies (1 \leq b/a) = (a \leq b)$   
**by**  $(\text{auto simp add: le-divide-eq})$

**lemma** *le-divide-eq-1-neg* [*simp, noatp*]:  
**fixes**  $a :: 'a :: \{\text{ordered-field}, \text{division-by-zero}\}$   
**shows**  $a < 0 \implies (1 \leq b/a) = (b \leq a)$   
**by**  $(\text{auto simp add: le-divide-eq})$

**lemma** *divide-le-eq-1-pos* [*simp, noatp*]:  
**fixes**  $a :: 'a :: \{\text{ordered-field}, \text{division-by-zero}\}$   
**shows**  $0 < a \implies (b/a \leq 1) = (b \leq a)$   
**by**  $(\text{auto simp add: divide-le-eq})$

**lemma** *divide-le-eq-1-neg* [*simp, noatp*]:  
**fixes**  $a :: 'a :: \{\text{ordered-field}, \text{division-by-zero}\}$   
**shows**  $a < 0 \implies (b/a \leq 1) = (a \leq b)$   
**by**  $(\text{auto simp add: divide-le-eq})$

**lemma** *less-divide-eq-1-pos* [*simp, noatp*]:  
**fixes**  $a :: 'a :: \{\text{ordered-field}, \text{division-by-zero}\}$   
**shows**  $0 < a \implies (1 < b/a) = (a < b)$   
**by**  $(\text{auto simp add: less-divide-eq})$

**lemma** *less-divide-eq-1-neg* [*simp, noatp*]:  
**fixes**  $a :: 'a :: \{\text{ordered-field}, \text{division-by-zero}\}$   
**shows**  $a < 0 \implies (1 < b/a) = (b < a)$   
**by**  $(\text{auto simp add: less-divide-eq})$

**lemma** *divide-less-eq-1-pos* [*simp, noatp*]:  
**fixes**  $a :: 'a :: \{\text{ordered-field}, \text{division-by-zero}\}$   
**shows**  $0 < a \implies (b/a < 1) = (b < a)$   
**by**  $(\text{auto simp add: divide-less-eq})$

**lemma** *divide-less-eq-1-neg* [*simp, noatp*]:  
**fixes**  $a :: 'a :: \{\text{ordered-field}, \text{division-by-zero}\}$   
**shows**  $a < 0 \implies b/a < 1 \iff a < b$   
**by**  $(\text{auto simp add: divide-less-eq})$

**lemma** *eq-divide-eq-1* [*simp, noatp*]:  
**fixes**  $a :: 'a :: \{\text{ordered-field}, \text{division-by-zero}\}$   
**shows**  $(1 = b/a) = ((a \neq 0 \ \& \ a = b))$

**by** (*auto simp add: eq-divide-eq*)

**lemma** *divide-eq-eq-1* [*simp, noatp*]:  
**fixes**  $a :: 'a :: \{\text{ordered-field}, \text{division-by-zero}\}$   
**shows**  $(b/a = 1) = ((a \neq 0 \ \& \ a = b))$   
**by** (*auto simp add: divide-eq-eq*)

### 11.13 Reasoning about inequalities with division

**lemma** *mult-right-le-one-le*:  $0 \leq (x :: 'a :: \text{ordered-idom}) \implies 0 \leq y \implies y \leq 1 \implies x * y \leq x$   
**by** (*auto simp add: mult-compare-simps*)

**lemma** *mult-left-le-one-le*:  $0 \leq (x :: 'a :: \text{ordered-idom}) \implies 0 \leq y \implies y \leq 1 \implies y * x \leq x$   
**by** (*auto simp add: mult-compare-simps*)

**lemma** *mult-imp-div-pos-le*:  $0 < (y :: 'a :: \text{ordered-field}) \implies x \leq z * y \implies x / y \leq z$   
**by** (*subst pos-divide-le-eq, assumption+*)

**lemma** *mult-imp-le-div-pos*:  $0 < (y :: 'a :: \text{ordered-field}) \implies z * y \leq x \implies z \leq x / y$   
**by** (*simp add: field-simps*)

**lemma** *mult-imp-div-pos-less*:  $0 < (y :: 'a :: \text{ordered-field}) \implies x < z * y \implies x / y < z$   
**by** (*simp add: field-simps*)

**lemma** *mult-imp-less-div-pos*:  $0 < (y :: 'a :: \text{ordered-field}) \implies z * y < x \implies z < x / y$   
**by** (*simp add: field-simps*)

**lemma** *frac-le*:  $(0 :: 'a :: \text{ordered-field}) \leq x \implies x \leq y \implies 0 < w \implies w \leq z \implies x / z \leq y / w$   
**apply** (*rule mult-imp-div-pos-le*)  
**apply** *simp*  
**apply** (*subst times-divide-eq-left*)  
**apply** (*rule mult-imp-le-div-pos, assumption*)  
**apply** (*rule mult-mono*)  
**apply** *simp-all*  
**done**

**lemma** *frac-less*:  $(0 :: 'a :: \text{ordered-field}) \leq x \implies x < y \implies 0 < w \implies w \leq z \implies x / z < y / w$   
**apply** (*rule mult-imp-div-pos-less*)  
**apply** *simp*

```

apply (subst times-divide-eq-left)
apply (rule mult-imp-less-div-pos, assumption)
apply (erule mult-less-le-imp-less)
apply simp-all
done

lemma frac-less2: (0::'a::ordered-field) < x ==>
  x <= y ==> 0 < w ==> w < z ==> x / z < y / w
apply (rule mult-imp-div-pos-less)
apply simp-all
apply (subst times-divide-eq-left)
apply (rule mult-imp-less-div-pos, assumption)
apply (erule mult-le-less-imp-less)
apply simp-all
done

```

It’s not obvious whether these should be simprules or not. Their effect is to gather terms into one big fraction, like  $a*b*c / x*y*z$ . The rationale for that is unclear, but many proofs seem to need them.

```
declare times-divide-eq [simp]
```

### 11.14 Ordered Fields are Dense

```
context ordered-semidom
begin
```

```

lemma less-add-one: a < a + 1
proof –
  have a + 0 < a + 1
    by (blast intro: zero-less-one add-strict-left-mono)
  thus ?thesis by simp
qed

```

```

lemma zero-less-two: 0 < 1 + 1
by (blast intro: less-trans zero-less-one less-add-one)

```

```
end
```

```

lemma less-half-sum: a < b ==> a < (a+b) / (1+1::'a::ordered-field)
by (simp add: field-simps zero-less-two)

```

```

lemma gt-half-sum: a < b ==> (a+b)/(1+1::'a::ordered-field) < b
by (simp add: field-simps zero-less-two)

```

```

instance ordered-field < dense-linear-order
proof
  fix x y :: 'a
  have x < x + 1 by simp
  then show  $\exists y. x < y$  ..

```

```

have  $x - 1 < x$  by simp
then show  $\exists y. y < x$  ..
show  $x < y \implies \exists z > x. z < y$  by (blast intro!: less-half-sum gt-half-sum)
qed

```

### 11.15 Absolute Value

```

context ordered-idom
begin

```

```

lemma mult-sgn-abs:  $\text{sgn } x * \text{abs } x = x$ 
  unfolding abs-if sgn-if by auto

```

```

end

```

```

lemma abs-one [simp]:  $\text{abs } 1 = (1 :: 'a :: \text{ordered-idom})$ 
by (simp add: abs-if zero-less-one [THEN order-less-not-sym])

```

```

class pordered-ring-abs = pordered-ring + pordered-ab-group-add-abs +
  assumes abs-eq-mult:
     $(0 \leq a \vee a \leq 0) \wedge (0 \leq b \vee b \leq 0) \implies |a * b| = |a| * |b|$ 

```

```

class lordered-ring = pordered-ring + lordered-ab-group-add-abs
begin

```

```

subclass lordered-ab-group-add-meet ..
subclass lordered-ab-group-add-join ..

```

```

end

```

```

lemma abs-le-mult:  $\text{abs } (a * b) \leq (\text{abs } a) * (\text{abs } (b :: 'a :: \text{lordered-ring}))$ 

```

```

proof -

```

```

  let  $?x = \text{pprt } a * \text{pprt } b - \text{pprt } a * \text{nprt } b - \text{nprt } a * \text{pprt } b + \text{nprt } a * \text{nprt } b$ 

```

```

  let  $?y = \text{pprt } a * \text{pprt } b + \text{pprt } a * \text{nprt } b + \text{nprt } a * \text{pprt } b + \text{nprt } a * \text{nprt } b$ 

```

```

  have  $a: (\text{abs } a) * (\text{abs } b) = ?x$ 

```

```

  by (simp only: abs-prts[of a] abs-prts[of b] algebra-simps)

```

```

  {

```

```

    fix  $u v :: 'a$ 

```

```

    have  $bh: \llbracket u = a; v = b \rrbracket \implies$ 

```

```

       $u * v = \text{pprt } a * \text{pprt } b + \text{pprt } a * \text{nprt } b +$ 
       $\text{nprt } a * \text{pprt } b + \text{nprt } a * \text{nprt } b$ 

```

```

    apply (subst prts[of u], subst prts[of v])

```

```

    apply (simp add: algebra-simps)

```

```

    done

```

```

  }

```

```

  note  $b = \text{this}[OF \text{ refl}[of a] \text{ refl}[of b]]$ 

```

```

  note  $\text{addm} = \text{add-mono}[of 0 :: 'a - 0 :: 'a, \text{simplified}]$ 

```

```

  note  $\text{addm2} = \text{add-mono}[of - 0 :: 'a - 0 :: 'a, \text{simplified}]$ 

```

```

have xy: - ?x <= ?y
  apply (simp)
  apply (rule-tac y=0::'a in order-trans)
  apply (rule addm2)
  apply (simp-all add: mult-nonneg-nonneg mult-nonpos-nonpos)
  apply (rule addm)
  apply (simp-all add: mult-nonneg-nonneg mult-nonpos-nonpos)
done
have yx: ?y <= ?x
  apply (simp add:diff-def)
  apply (rule-tac y=0 in order-trans)
  apply (rule addm2, (simp add: mult-nonneg-nonpos mult-nonneg-nonpos2)+)
  apply (rule addm, (simp add: mult-nonneg-nonpos mult-nonneg-nonpos2)+)
done
have i1: a*b <= abs a * abs b by (simp only: a b yx)
have i2: - (abs a * abs b) <= a*b by (simp only: a b xy)
show ?thesis
  apply (rule abs-leI)
  apply (simp add: i1)
  apply (simp add: i2[simplified minus-le-iff])
done
qed

instance lordered-ring ⊆ pordered-ring-abs
proof
  fix a b :: 'a:: lordered-ring
  assume (0 ≤ a ∨ a ≤ 0) ∧ (0 ≤ b ∨ b ≤ 0)
  show abs (a*b) = abs a * abs b
proof -
  have s: (0 <= a*b) | (a*b <= 0)
  apply (auto)
  apply (rule-tac split-mult-pos-le)
  apply (rule-tac contrapos-np[of a*b <= 0])
  apply (simp)
  apply (rule-tac split-mult-neg-le)
  apply (insert prems)
  apply (blast)
done
have mulprts: a * b = (pprt a + nprrt a) * (pprt b + nprrt b)
  by (simp add: prts[symmetric])
show ?thesis
proof cases
  assume 0 <= a * b
  then show ?thesis
    apply (simp-all add: mulprts abs-prts)
    apply (insert prems)
    apply (auto simp add:
      algebra-simps
      iffD1[OF zero-le-iff-zero-npr] iffD1[OF le-zero-iff-zero-pprt])

```

```

      iffD1[OF le-zero-iff-pprt-id] iffD1[OF zero-le-iff-nprt-id])
    apply (drule (1) mult-nonneg-nonpos[of a b], simp)
    apply (drule (1) mult-nonneg-nonpos2[of b a], simp)
  done
next
  assume  $\sim(0 \leq a * b)$ 
  with s have  $a * b \leq 0$  by simp
  then show ?thesis
    apply (simp-all add: mulprts abs-prts)
    apply (insert prems)
    apply (auto simp add: algebra-simps)
    apply (drule (1) mult-nonneg-nonneg[of a b], simp)
    apply (drule (1) mult-nonpos-nonpos[of a b], simp)
  done
qed
qed
qed

instance ordered-idom  $\subseteq$  pordered-ring-abs
by default (auto simp add: abs-if not-less
  equal-neg-zero neg-equal-zero mult-less-0-iff)

lemma abs-mult:  $\text{abs } (a * b) = \text{abs } a * \text{abs } (b::'a::\text{ordered-idom})$ 
by (simp add: abs-eq-mult linorder-linear)

lemma abs-mult-self:  $\text{abs } a * \text{abs } a = a * (a::'a::\text{ordered-idom})$ 
by (simp add: abs-if)

lemma nonzero-abs-inverse:
   $a \neq 0 \implies \text{abs } (\text{inverse } (a::'a::\text{ordered-field})) = \text{inverse } (\text{abs } a)$ 
apply (auto simp add: linorder-neq-iff abs-if nonzero-inverse-minus-eq
  negative-imp-inverse-negative)
apply (blast intro: positive-imp-inverse-positive elim: order-less-asm)
done

lemma abs-inverse [simp]:
   $\text{abs } (\text{inverse } (a::'a::\{\text{ordered-field}, \text{division-by-zero}\})) =$ 
   $\text{inverse } (\text{abs } a)$ 
apply (cases a=0, simp)
apply (simp add: nonzero-abs-inverse)
done

lemma nonzero-abs-divide:
   $b \neq 0 \implies \text{abs } (a / (b::'a::\text{ordered-field})) = \text{abs } a / \text{abs } b$ 
by (simp add: divide-inverse abs-mult nonzero-abs-inverse)

lemma abs-divide [simp]:
   $\text{abs } (a / (b::'a::\{\text{ordered-field}, \text{division-by-zero}\})) = \text{abs } a / \text{abs } b$ 
apply (cases b=0, simp)

```

**apply** (*simp add: nonzero-abs-divide*)  
**done**

**lemma** *abs-mult-less*:

$[[ \text{abs } a < c; \text{abs } b < d ]] \implies \text{abs } a * \text{abs } b < c * (d :: 'a :: \text{ordered-idom})$

**proof** –

**assume** *ac*:  $\text{abs } a < c$

**hence** *cpos*:  $0 < c$  **by** (*blast intro: order-le-less-trans abs-ge-zero*)

**assume**  $\text{abs } b < d$

**thus** *?thesis* **by** (*simp add: ac cpos mult-strict-mono*)

**qed**

**lemmas** *eq-minus-self-iff*[*noatp*] = *equal-neg-zero*

**lemma** *less-minus-self-iff*:  $(a < -a) = (a < (0 :: 'a :: \text{ordered-idom}))$

**unfolding** *order-less-le less-eq-neg-nonpos equal-neg-zero* ..

**lemma** *abs-less-iff*:  $(\text{abs } a < b) = (a < b \ \& \ -a < (b :: 'a :: \text{ordered-idom}))$

**apply** (*simp add: order-less-le abs-le-iff*)

**apply** (*auto simp add: abs-if neg-less-eq-nonneg less-eq-neg-nonpos*)

**done**

**lemma** *abs-mult-pos*:  $(0 :: 'a :: \text{ordered-idom}) \leq x \implies$

$(\text{abs } y) * x = \text{abs } (y * x)$

**apply** (*subst abs-mult*)

**apply** *simp*

**done**

**lemma** *abs-div-pos*:  $(0 :: 'a :: \{\text{division-by-zero}, \text{ordered-field}\}) < y \implies$

$\text{abs } x / y = \text{abs } (x / y)$

**apply** (*subst abs-divide*)

**apply** (*simp add: order-less-imp-le*)

**done**

## 11.16 Bounds of products via negative and positive Part

**lemma** *mult-le-prts*:

**assumes**

$a1 \leq (a :: 'a :: \text{ordered-ring})$

$a \leq a2$

$b1 \leq b$

$b \leq b2$

**shows**

$a * b \leq \text{pprt } a2 * \text{pprt } b2 + \text{pprt } a1 * \text{nprt } b2 + \text{nprt } a2 * \text{pprt } b1 + \text{nprt } a1$

$* \text{nprt } b1$

**proof** –

**have**  $a * b = (\text{pprt } a + \text{nprt } a) * (\text{pprt } b + \text{nprt } b)$

**apply** (*subst prts[symmetric]*) +

**apply** *simp*

```

done
then have  $a * b = \text{pprt } a * \text{pprt } b + \text{pprt } a * \text{nprrt } b + \text{nprrt } a * \text{pprt } b + \text{nprrt } a * \text{nprrt } b$ 
by (simp add: algebra-simps)
moreover have  $\text{pprt } a * \text{pprt } b \leq \text{pprt } a2 * \text{pprt } b2$ 
by (simp-all add: prems mult-mono)
moreover have  $\text{pprt } a * \text{nprrt } b \leq \text{pprt } a1 * \text{nprrt } b2$ 
proof -
  have  $\text{pprt } a * \text{nprrt } b \leq \text{pprt } a * \text{nprrt } b2$ 
  by (simp add: mult-left-mono prems)
  moreover have  $\text{pprt } a * \text{nprrt } b2 \leq \text{pprt } a1 * \text{nprrt } b2$ 
  by (simp add: mult-right-mono-neg prems)
  ultimately show ?thesis
  by simp
qed
moreover have  $\text{nprrt } a * \text{pprt } b \leq \text{nprrt } a2 * \text{pprt } b1$ 
proof -
  have  $\text{nprrt } a * \text{pprt } b \leq \text{nprrt } a2 * \text{pprt } b$ 
  by (simp add: mult-right-mono prems)
  moreover have  $\text{nprrt } a2 * \text{pprt } b \leq \text{nprrt } a2 * \text{pprt } b1$ 
  by (simp add: mult-left-mono-neg prems)
  ultimately show ?thesis
  by simp
qed
moreover have  $\text{nprrt } a * \text{nprrt } b \leq \text{nprrt } a1 * \text{nprrt } b1$ 
proof -
  have  $\text{nprrt } a * \text{nprrt } b \leq \text{nprrt } a * \text{nprrt } b1$ 
  by (simp add: mult-left-mono-neg prems)
  moreover have  $\text{nprrt } a * \text{nprrt } b1 \leq \text{nprrt } a1 * \text{nprrt } b1$ 
  by (simp add: mult-right-mono-neg prems)
  ultimately show ?thesis
  by simp
qed
ultimately show ?thesis
by - (rule add-mono | simp)+
qed

```

**lemma** *mult-ge-prts*:

```

assumes
   $a1 \leq (a::'a::\text{ordered-ring})$ 
   $a \leq a2$ 
   $b1 \leq b$ 
   $b \leq b2$ 
shows
   $a * b \geq \text{nprrt } a1 * \text{pprt } b2 + \text{nprrt } a2 * \text{nprrt } b2 + \text{pprt } a1 * \text{pprt } b1 + \text{pprt } a2 * \text{nprrt } b1$ 
proof -
  from prems have  $a1: -a2 \leq -a$  by auto
  from prems have  $a2: -a \leq -a1$  by auto

```



```

from mult-le-prts[of  $-a2 - a - a1\ b1\ b\ b2$ , OF  $a1\ a2$  prems(3) prems(4), simplified npert-neg pprt-neg]
have  $le: - (a * b) \leq - npert\ a1 * pprt\ b2 + - npert\ a2 * npert\ b2 + - pprt\ a1$ 
 $* pprt\ b1 + - pprt\ a2 * npert\ b1$  by simp
then have  $-(- npert\ a1 * pprt\ b2 + - npert\ a2 * npert\ b2 + - pprt\ a1 * pprt$ 
 $b1 + - pprt\ a2 * npert\ b1) \leq a * b$ 
by (simp only: minus-le-iff)
then show ?thesis by simp
qed

end

```

## 12 Nat: Natural numbers

```

theory Nat
imports Inductive Ring-and-Field
uses
  ~~/src/Tools/rat.ML
  ~~/src/Provers/Arith/cancel-sums.ML
  Tools/arith-data.ML
  (Tools/nat-arith.ML)
  ~~/src/Provers/Arith/fast-lin-arith.ML
  (Tools/lin-arith.ML)
begin

```

### 12.1 Type *ind*

```
typedecl ind
```

**axiomatization**

```

  Zero-Rep :: ind and
  Suc-Rep :: ind => ind

```

**where**

```

  — the axiom of infinity in 2 parts
  inj-Suc-Rep:      inj Suc-Rep and
  Suc-Rep-not-Zero-Rep: Suc-Rep  $x \neq$  Zero-Rep

```

### 12.2 Type *nat*

Type definition

```
inductive Nat :: ind => bool
```

**where**

```

  Zero-RepI: Nat Zero-Rep
  | Suc-RepI: Nat i => Nat (Suc-Rep i)

```

**global**

```

typedef (open Nat)
  nat = Nat
  by (rule exI, unfold mem-def, rule Nat.Zero-RepI)

constdefs
  Suc :: nat => nat
  Suc-def: Suc == (%n. Abs-Nat (Suc-Rep (Rep-Nat n)))

local

instantiation nat :: zero
begin

definition Zero-nat-def [code del]:
  0 = Abs-Nat Zero-Rep

instance ..

end

lemma Suc-not-Zero: Suc m ≠ 0
  by (simp add: Zero-nat-def Suc-def Abs-Nat-inject [unfolded mem-def]
    Rep-Nat [unfolded mem-def] Suc-RepI Zero-RepI Suc-Rep-not-Zero-Rep [unfolded
    mem-def])

lemma Zero-not-Suc: 0 ≠ Suc m
  by (rule not-sym, rule Suc-not-Zero not-sym)

rep-datatype 0 :: nat Suc
  apply (unfold Zero-nat-def Suc-def)
  apply (rule Rep-Nat-inverse [THEN subst]) — types force good instantiation
  apply (erule Rep-Nat [unfolded mem-def, THEN Nat.induct])
  apply (iprover elim: Abs-Nat-inverse [unfolded mem-def, THEN subst])
  apply (simp-all add: Abs-Nat-inject [unfolded mem-def] Rep-Nat [unfolded
    mem-def]
    Suc-RepI Zero-RepI Suc-Rep-not-Zero-Rep [unfolded mem-def]
    Suc-Rep-not-Zero-Rep [unfolded mem-def, symmetric]
    inj-Suc-Rep [THEN inj-eq] Rep-Nat-inject)
  done

lemma nat-induct [case-names 0 Suc, induct type: nat]:
  — for backward compatibility – names of variables differ
  fixes n
  assumes P 0
  and  $\bigwedge n. P\ n \implies P\ (Suc\ n)$ 
  shows P n
  using assms by (rule nat.induct)

declare nat.exhaust [case-names 0 Suc, cases type: nat]

```

```

lemmas nat-rec-0 = nat.recs(1)
and nat-rec-Suc = nat.recs(2)

```

```

lemmas nat-case-0 = nat.cases(1)
and nat-case-Suc = nat.cases(2)

```

Injectiveness and distinctness lemmas

```

lemma inj-Suc[simp]: inj-on Suc N
by (simp add: inj-on-def)

```

```

lemma Suc-neq-Zero: Suc m = 0 ==> R
by (rule notE, rule Suc-not-Zero)

```

```

lemma Zero-neq-Suc: 0 = Suc m ==> R
by (rule Suc-neq-Zero, erule sym)

```

```

lemma Suc-inject: Suc x = Suc y ==> x = y
by (rule inj-Suc [THEN injD])

```

```

lemma n-not-Suc-n: n ≠ Suc n
by (induct n) simp-all

```

```

lemma Suc-n-not-n: Suc n ≠ n
by (rule not-sym, rule n-not-Suc-n)

```

A special form of induction for reasoning about  $m < n$  and  $m - n$

```

lemma diff-induct: ( $\forall x. P\ x\ 0$ ) ==> ( $\forall y. P\ 0\ (Suc\ y)$ ) ==>
  ( $\forall x\ y. P\ x\ y$  ==>  $P\ (Suc\ x)\ (Suc\ y)$ ) ==>  $P\ m\ n$ 
apply (rule-tac x = m in spec)
apply (induct n)
prefer 2
apply (rule allI)
apply (induct-tac x, iprover+)
done

```

### 12.3 Arithmetic operators

```

instantiation nat :: {minus, comm-monoid-add}
begin

```

```

primrec plus-nat
where

```

```

  add-0:       $0 + n = (n::nat)$ 
| add-Suc:  $Suc\ m + n = Suc\ (m + n)$ 

```

```

lemma add-0-right [simp]:  $m + 0 = (m::nat)$ 
by (induct m) simp-all

```

**lemma** *add-Suc-right* [*simp*]:  $m + \text{Suc } n = \text{Suc } (m + n)$   
**by** (*induct m*) *simp-all*

**declare** *add-0* [*code*]

**lemma** *add-Suc-shift* [*code*]:  $\text{Suc } m + n = m + \text{Suc } n$   
**by** *simp*

**primrec** *minus-nat*

**where**

*diff-0*:  $m - 0 = (m::\text{nat})$   
| *diff-Suc*:  $m - \text{Suc } n = (\text{case } m - n \text{ of } 0 \Rightarrow 0 \mid \text{Suc } k \Rightarrow k)$

**declare** *diff-Suc* [*simp del*]

**declare** *diff-0* [*code*]

**lemma** *diff-0-eq-0* [*simp, code*]:  $0 - n = (0::\text{nat})$   
**by** (*induct n*) (*simp-all add: diff-Suc*)

**lemma** *diff-Suc-Suc* [*simp, code*]:  $\text{Suc } m - \text{Suc } n = m - n$   
**by** (*induct n*) (*simp-all add: diff-Suc*)

**instance** **proof**

**fix**  $n \ m \ q :: \text{nat}$

**show**  $(n + m) + q = n + (m + q)$  **by** (*induct n*) *simp-all*

**show**  $n + m = m + n$  **by** (*induct n*) *simp-all*

**show**  $0 + n = n$  **by** *simp*

**qed**

**end**

**instantiation** *nat* :: *comm-semiring-1-cancel*

**begin**

**definition**

*One-nat-def* [*simp*]:  $1 = \text{Suc } 0$

**primrec** *times-nat*

**where**

*mult-0*:  $0 * n = (0::\text{nat})$   
| *mult-Suc*:  $\text{Suc } m * n = n + (m * n)$

**lemma** *mult-0-right* [*simp*]:  $(m::\text{nat}) * 0 = 0$   
**by** (*induct m*) *simp-all*

**lemma** *mult-Suc-right* [*simp*]:  $m * \text{Suc } n = m + (m * n)$   
**by** (*induct m*) (*simp-all add: add-left-commute*)

**lemma** *add-mult-distrib*:  $(m + n) * k = (m * k) + ((n * k)::\text{nat})$

```

by (induct m) (simp-all add: add-assoc)

instance proof
  fix n m q :: nat
  show 0 ≠ (1::nat) unfolding One-nat-def by simp
  show 1 * n = n unfolding One-nat-def by simp
  show n * m = m * n by (induct n) simp-all
  show (n * m) * q = n * (m * q) by (induct n) (simp-all add: add-mult-distrib)
  show (n + m) * q = n * q + m * q by (rule add-mult-distrib)
  assume n + m = n + q thus m = q by (induct n) simp-all
qed

end

```

### 12.3.1 Addition

```

lemma nat-add-assoc: (m + n) + k = m + ((n + k)::nat)
  by (rule add-assoc)

```

```

lemma nat-add-commute: m + n = n + (m::nat)
  by (rule add-commute)

```

```

lemma nat-add-left-commute: x + (y + z) = y + ((x + z)::nat)
  by (rule add-left-commute)

```

```

lemma nat-add-left-cancel [simp]: (k + m = k + n) = (m = (n::nat))
  by (rule add-left-cancel)

```

```

lemma nat-add-right-cancel [simp]: (m + k = n + k) = (m = (n::nat))
  by (rule add-right-cancel)

```

Reasoning about  $m + 0 = 0$ , etc.

```

lemma add-is-0 [iff]:
  fixes m n :: nat
  shows (m + n = 0) = (m = 0 & n = 0)
  by (cases m) simp-all

```

```

lemma add-is-1:
  (m+n = Suc 0) = (m = Suc 0 & n=0 | m=0 & n = Suc 0)
  by (cases m) simp-all

```

```

lemma one-is-add:
  (Suc 0 = m + n) = (m = Suc 0 & n = 0 | m = 0 & n = Suc 0)
  by (rule trans, rule eq-commute, rule add-is-1)

```

```

lemma add-eq-self-zero:
  fixes m n :: nat
  shows m + n = m ⟹ n = 0
  by (induct m) simp-all

```

```

lemma inj-on-add-nat [simp]: inj-on ( $\%n::nat. n+k$ ) N
  apply (induct k)
  apply simp
  apply (drule comp-inj-on [OF - inj-Suc])
  apply (simp add:o-def)
done

```

### 12.3.2 Difference

```

lemma diff-self-eq-0 [simp]:  $(m::nat) - m = 0$ 
  by (induct m) simp-all

```

```

lemma diff-diff-left:  $(i::nat) - j - k = i - (j + k)$ 
  by (induct i j rule: diff-induct) simp-all

```

```

lemma Suc-diff-diff [simp]:  $(Suc\ m - n) - Suc\ k = m - n - k$ 
  by (simp add: diff-diff-left)

```

```

lemma diff-commute:  $(i::nat) - j - k = i - k - j$ 
  by (simp add: diff-diff-left add-commute)

```

```

lemma diff-add-inverse:  $(n + m) - n = (m::nat)$ 
  by (induct n) simp-all

```

```

lemma diff-add-inverse2:  $(m + n) - n = (m::nat)$ 
  by (simp add: diff-add-inverse add-commute [of m n])

```

```

lemma diff-cancel:  $(k + m) - (k + n) = m - (n::nat)$ 
  by (induct k) simp-all

```

```

lemma diff-cancel2:  $(m + k) - (n + k) = m - (n::nat)$ 
  by (simp add: diff-cancel add-commute)

```

```

lemma diff-add-0:  $n - (n + m) = (0::nat)$ 
  by (induct n) simp-all

```

```

lemma diff-Suc-1 [simp]:  $Suc\ n - 1 = n$ 
  unfolding One-nat-def by simp

```

Difference distributes over multiplication

```

lemma diff-mult-distrib:  $((m::nat) - n) * k = (m * k) - (n * k)$ 
  by (induct m n rule: diff-induct) (simp-all add: diff-cancel)

```

```

lemma diff-mult-distrib2:  $k * ((m::nat) - n) = (k * m) - (k * n)$ 
  by (simp add: diff-mult-distrib mult-commute [of k])

```

— NOT added as rewrites, since sometimes they are used from right-to-left

### 12.3.3 Multiplication

**lemma** *nat-mult-assoc*:  $(m * n) * k = m * ((n * k)::nat)$   
**by** (*rule mult-assoc*)

**lemma** *nat-mult-commute*:  $m * n = n * (m::nat)$   
**by** (*rule mult-commute*)

**lemma** *add-mult-distrib2*:  $k * (m + n) = (k * m) + ((k * n)::nat)$   
**by** (*rule right-distrib*)

**lemma** *mult-is-0* [*simp*]:  $((m::nat) * n = 0) = (m = 0 \mid n = 0)$   
**by** (*induct m*) *auto*

**lemmas** *nat-distrib* =  
*add-mult-distrib add-mult-distrib2 diff-mult-distrib diff-mult-distrib2*

**lemma** *mult-eq-1-iff* [*simp*]:  $(m * n = Suc\ 0) = (m = Suc\ 0 \ \&\ n = Suc\ 0)$   
**apply** (*induct m*)  
**apply** *simp*  
**apply** (*induct n*)  
**apply** *auto*  
**done**

**lemma** *one-eq-mult-iff* [*simp, noatp*]:  $(Suc\ 0 = m * n) = (m = Suc\ 0 \ \&\ n = Suc\ 0)$   
**apply** (*rule trans*)  
**apply** (*rule-tac* [2] *mult-eq-1-iff, fastsimp*)  
**done**

**lemma** *nat-mult-eq-1-iff* [*simp*]:  $m * n = (1::nat) \longleftrightarrow m = 1 \ \wedge\ n = 1$   
**unfolding** *One-nat-def* **by** (*rule mult-eq-1-iff*)

**lemma** *nat-1-eq-mult-iff* [*simp*]:  $(1::nat) = m * n \longleftrightarrow m = 1 \ \wedge\ n = 1$   
**unfolding** *One-nat-def* **by** (*rule one-eq-mult-iff*)

**lemma** *mult-cancel1* [*simp*]:  $(k * m = k * n) = (m = n \mid (k = (0::nat)))$   
**proof** –  
**have**  $k \neq 0 \implies k * m = k * n \implies m = n$   
**proof** (*induct n arbitrary: m*)  
**case** 0 **then show**  $m = 0$  **by** *simp*  
**next**  
**case** (*Suc n*) **then show**  $m = Suc\ n$   
**by** (*cases m*) (*simp-all add: eq-commute [of 0]*)  
**qed**  
**then show** *?thesis* **by** *auto*  
**qed**

**lemma** *mult-cancel2* [*simp*]:  $(m * k = n * k) = (m = n \mid (k = (0::nat)))$   
**by** (*simp add: mult-commute*)

**lemma** *Suc-mult-cancel1*:  $(\text{Suc } k * m = \text{Suc } k * n) = (m = n)$   
**by** (*subst mult-cancel1*) *simp*

## 12.4 Orders on *nat*

### 12.4.1 Operation definition

**instantiation** *nat* :: *linorder*  
**begin**

**primrec** *less-eq-nat* **where**  
 $(0::\text{nat}) \leq n \longleftrightarrow \text{True}$   
 $|\text{Suc } m \leq n \longleftrightarrow (\text{case } n \text{ of } 0 \Rightarrow \text{False} \mid \text{Suc } n \Rightarrow m \leq n)$

**declare** *less-eq-nat.simps* [*simp del*]

**lemma** [*code*]:  $(0::\text{nat}) \leq n \longleftrightarrow \text{True}$  **by** (*simp add: less-eq-nat.simps*)

**lemma** *le0* [*iff*]:  $0 \leq (n::\text{nat})$  **by** (*simp add: less-eq-nat.simps*)

**definition** *less-nat* **where**  
 $\text{less-eq-Suc-le}: n < m \longleftrightarrow \text{Suc } n \leq m$

**lemma** *Suc-le-mono* [*iff*]:  $\text{Suc } n \leq \text{Suc } m \longleftrightarrow n \leq m$   
**by** (*simp add: less-eq-nat.simps(2)*)

**lemma** *Suc-le-eq* [*code*]:  $\text{Suc } m \leq n \longleftrightarrow m < n$   
**unfolding** *less-eq-Suc-le* ..

**lemma** *le-0-eq* [*iff*]:  $(n::\text{nat}) \leq 0 \longleftrightarrow n = 0$   
**by** (*induct n*) (*simp-all add: less-eq-nat.simps(2)*)

**lemma** *not-less0* [*iff*]:  $\neg n < (0::\text{nat})$   
**by** (*simp add: less-eq-Suc-le*)

**lemma** *less-nat-zero-code* [*code*]:  $n < (0::\text{nat}) \longleftrightarrow \text{False}$   
**by** *simp*

**lemma** *Suc-less-eq* [*iff*]:  $\text{Suc } m < \text{Suc } n \longleftrightarrow m < n$   
**by** (*simp add: less-eq-Suc-le*)

**lemma** *less-Suc-eq-le* [*code*]:  $m < \text{Suc } n \longleftrightarrow m \leq n$   
**by** (*simp add: less-eq-Suc-le*)

**lemma** *le-SucI*:  $m \leq n \Longrightarrow m \leq \text{Suc } n$   
**by** (*induct m arbitrary: n*)  
*(simp-all add: less-eq-nat.simps(2) split: nat.splits)*

**lemma** *Suc-leD*:  $\text{Suc } m \leq n \Longrightarrow m \leq n$   
**by** (*cases n*) (*auto intro: le-SucI*)



```

lemma less-SucI:  $m < n \implies m < \text{Suc } n$ 
  by (simp add: less-eq-Suc-le) (erule Suc-leD)

lemma Suc-lessD:  $\text{Suc } m < n \implies m < n$ 
  by (simp add: less-eq-Suc-le) (erule Suc-leD)

instance
proof
  fix  $n\ m :: \text{nat}$ 
  show  $n < m \iff n \leq m \wedge \neg m \leq n$ 
  proof (induct n arbitrary: m)
    case 0 then show ?case by (cases m) (simp-all add: less-eq-Suc-le)
  next
    case (Suc n) then show ?case by (cases m) (simp-all add: less-eq-Suc-le)
  qed
next
  fix  $n :: \text{nat}$  show  $n \leq n$  by (induct n) simp-all
next
  fix  $n\ m :: \text{nat}$  assume  $n \leq m$  and  $m \leq n$ 
  then show  $n = m$ 
  by (induct n arbitrary: m)
    (simp-all add: less-eq-nat.simps(2) split: nat.splits)
next
  fix  $n\ m\ q :: \text{nat}$  assume  $n \leq m$  and  $m \leq q$ 
  then show  $n \leq q$ 
  proof (induct n arbitrary: m q)
    case 0 show ?case by simp
  next
    case (Suc n) then show ?case
    by (simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,
      simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,
      simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits)
  qed
next
  fix  $n\ m :: \text{nat}$  show  $n \leq m \vee m \leq n$ 
  by (induct n arbitrary: m)
    (simp-all add: less-eq-nat.simps(2) split: nat.splits)
qed

end

instantiation nat :: bot
begin

definition bot-nat :: nat where
  bot-nat = 0

instance proof
qed (simp add: bot-nat-def)

```

end

### 12.4.2 Introduction properties

**lemma** *lessI* [iff]:  $n < \text{Suc } n$   
 by (*simp add: less-Suc-eq-le*)

**lemma** *zero-less-Suc* [iff]:  $0 < \text{Suc } n$   
 by (*simp add: less-Suc-eq-le*)

### 12.4.3 Elimination properties

**lemma** *less-not-refl*:  $\sim n < (n::\text{nat})$   
 by (*rule order-less-irrefl*)

**lemma** *less-not-refl2*:  $n < m \implies m \neq (n::\text{nat})$   
 by (*rule not-sym*) (*rule less-imp-neq*)

**lemma** *less-not-refl3*:  $(s::\text{nat}) < t \implies s \neq t$   
 by (*rule less-imp-neq*)

**lemma** *less-irrefl-nat*:  $(n::\text{nat}) < n \implies R$   
 by (*rule notE*, *rule less-not-refl*)

**lemma** *less-zeroE*:  $(n::\text{nat}) < 0 \implies R$   
 by (*rule notE*) (*rule not-less0*)

**lemma** *less-Suc-eq*:  $(m < \text{Suc } n) = (m < n \mid m = n)$   
 unfolding *less-Suc-eq-le le-less* ..

**lemma** *less-Suc0* [iff]:  $(n < \text{Suc } 0) = (n = 0)$   
 by (*simp add: less-Suc-eq*)

**lemma** *less-one* [iff, noatp]:  $(n < (1::\text{nat})) = (n = 0)$   
 unfolding *One-nat-def* by (*rule less-Suc0*)

**lemma** *Suc-mono*:  $m < n \implies \text{Suc } m < \text{Suc } n$   
 by *simp*

”Less than” is antisymmetric, sort of

**lemma** *less-antisym*:  $\llbracket \neg n < m; n < \text{Suc } m \rrbracket \implies m = n$   
 unfolding *not-less less-Suc-eq-le* by (*rule antisym*)

**lemma** *nat-neq-iff*:  $((m::\text{nat}) \neq n) = (m < n \mid n < m)$   
 by (*rule linorder-neq-iff*)

**lemma** *nat-less-cases*: **assumes** *major*:  $(m::\text{nat}) < n \implies P \ n \ m$   
**and** *eqCase*:  $m = n \implies P \ n \ m$  **and** *lessCase*:  $n < m \implies P \ n \ m$   
**shows**  $P \ n \ m$

```

apply (rule less-linear [THEN disjE])
apply (erule-tac [2] disjE)
apply (erule lessCase)
apply (erule sym [THEN eqCase])
apply (erule major)
done

```

#### 12.4.4 Inductive (?) properties

**lemma** *Suc-lessI*:  $m < n \implies \text{Suc } m \neq n \implies \text{Suc } m < n$   
**unfolding** less-eq-Suc-le [of *m*] le-less **by** simp

**lemma** *lessE*:  
**assumes** major:  $i < k$   
**and** p1:  $k = \text{Suc } i \implies P$  **and** p2:  $\forall j. i < j \implies k = \text{Suc } j \implies P$   
**shows** *P*  
**proof** –  
**from** major **have**  $\exists j. i \leq j \wedge k = \text{Suc } j$   
**unfolding** less-eq-Suc-le **by** (induct *k*) simp-all  
**then have**  $(\exists j. i < j \wedge k = \text{Suc } j) \vee k = \text{Suc } i$   
**by** (clarsimp simp add: less-le)  
**with** p1 p2 **show** *P* **by** auto  
**qed**

**lemma** *less-SucE*: **assumes** major:  $m < \text{Suc } n$   
**and** less:  $m < n \implies P$  **and** eq:  $m = n \implies P$  **shows** *P*  
**apply** (rule major [THEN lessE])  
**apply** (rule eq, blast)  
**apply** (rule less, blast)  
**done**

**lemma** *Suc-lessE*: **assumes** major:  $\text{Suc } i < k$   
**and** minor:  $\forall j. i < j \implies k = \text{Suc } j \implies P$  **shows** *P*  
**apply** (rule major [THEN lessE])  
**apply** (erule lessI [THEN minor])  
**apply** (erule Suc-lessD [THEN minor], assumption)  
**done**

**lemma** *Suc-less-SucD*:  $\text{Suc } m < \text{Suc } n \implies m < n$   
**by** simp

**lemma** *less-trans-Suc*:  
**assumes** le:  $i < j$  **shows**  $j < k \implies \text{Suc } i < k$   
**apply** (induct *k*, simp-all)  
**apply** (insert le)  
**apply** (simp add: less-Suc-eq)  
**apply** (blast dest: Suc-lessD)  
**done**

Can be used with *less-Suc-eq* to get  $n = m \vee n < m$

**lemma** *not-less-eq*:  $\neg m < n \longleftrightarrow n < \text{Suc } m$   
**unfolding** *not-less less-Suc-eq-le* ..

**lemma** *not-less-eq-eq*:  $\neg m \leq n \longleftrightarrow \text{Suc } n \leq m$   
**unfolding** *not-le Suc-le-eq* ..

Properties of “less than or equal”

**lemma** *le-imp-less-Suc*:  $m \leq n \implies m < \text{Suc } n$   
**unfolding** *less-Suc-eq-le* .

**lemma** *Suc-n-not-le-n*:  $\sim \text{Suc } n \leq n$   
**unfolding** *not-le less-Suc-eq-le* ..

**lemma** *le-Suc-eq*:  $(m \leq \text{Suc } n) = (m \leq n \mid m = \text{Suc } n)$   
**by** (*simp add: less-Suc-eq-le [symmetric] less-Suc-eq*)

**lemma** *le-SucE*:  $m \leq \text{Suc } n \implies (m \leq n \implies R) \implies (m = \text{Suc } n \implies R)$   
 $\implies R$   
**by** (*drule le-Suc-eq [THEN iffD1], iprover+*)

**lemma** *Suc-leI*:  $m < n \implies \text{Suc}(m) \leq n$   
**unfolding** *Suc-le-eq* .

Stronger version of *Suc-leD*

**lemma** *Suc-le-lessD*:  $\text{Suc } m \leq n \implies m < n$   
**unfolding** *Suc-le-eq* .

**lemma** *less-imp-le-nat*:  $m < n \implies m \leq (n::\text{nat})$   
**unfolding** *less-eq-Suc-le* **by** (*rule Suc-leD*)

For instance,  $(\text{Suc } m < \text{Suc } n) = (\text{Suc } m \leq n) = (m < n)$

**lemmas** *le-simps* = *less-imp-le-nat less-Suc-eq-le Suc-le-eq*

Equivalence of  $m \leq n$  and  $m < n \vee m = n$

**lemma** *less-or-eq-imp-le*:  $m < n \mid m = n \implies m \leq (n::\text{nat})$   
**unfolding** *le-less* .

**lemma** *le-eq-less-or-eq*:  $(m \leq (n::\text{nat})) = (m < n \mid m = n)$   
**by** (*rule le-less*)

Useful with *blast*.

**lemma** *eq-imp-le*:  $(m::\text{nat}) = n \implies m \leq n$   
**by** *auto*

**lemma** *le-refl*:  $n \leq (n::\text{nat})$   
**by** *simp*

**lemma** *le-trans*:  $[\mid i \leq j; j \leq k \mid] \implies i \leq (k::\text{nat})$

**by** (*rule order-trans*)

**lemma** *le-anti-sym*:  $[[m \leq n; n \leq m]] \implies m = (n::nat)$   
**by** (*rule antisym*)

**lemma** *nat-less-le*:  $((m::nat) < n) = (m \leq n \ \& \ m \neq n)$   
**by** (*rule less-le*)

**lemma** *le-neq-implies-less*:  $(m::nat) \leq n \implies m \neq n \implies m < n$   
**unfolding** *less-le* ..

**lemma** *nat-le-linear*:  $(m::nat) \leq n \mid n \leq m$   
**by** (*rule linear*)

**lemmas** *linorder-neqE-nat* = *linorder-neqE* [**where** 'a = nat]

**lemma** *le-less-Suc-eq*:  $m \leq n \implies (n < Suc \ m) = (n = m)$   
**unfolding** *less-Suc-eq-le* **by** *auto*

**lemma** *not-less-less-Suc-eq*:  $\sim n < m \implies (n < Suc \ m) = (n = m)$   
**unfolding** *not-less* **by** (*rule le-less-Suc-eq*)

**lemmas** *not-less-simps* = *not-less-less-Suc-eq* *le-less-Suc-eq*

These two rules ease the use of primitive recursion. NOTE USE OF ==

**lemma** *def-nat-rec-0*:  $(!!n. f \ n == nat-rec \ c \ h \ n) \implies f \ 0 = c$   
**by** *simp*

**lemma** *def-nat-rec-Suc*:  $(!!n. f \ n == nat-rec \ c \ h \ n) \implies f \ (Suc \ n) = h \ n \ (f \ n)$   
**by** *simp*

**lemma** *not0-implies-Suc*:  $n \neq 0 \implies \exists m. n = Suc \ m$   
**by** (*cases n*) *simp-all*

**lemma** *gr0-implies-Suc*:  $n > 0 \implies \exists m. n = Suc \ m$   
**by** (*cases n*) *simp-all*

**lemma** *gr-implies-not0*: **fixes**  $n :: nat$  **shows**  $m < n \implies n \neq 0$   
**by** (*cases n*) *simp-all*

**lemma** *neq0-conv[iff]*: **fixes**  $n :: nat$  **shows**  $(n \neq 0) = (0 < n)$   
**by** (*cases n*) *simp-all*

This theorem is useful with *blast*

**lemma** *gr0I*:  $((n::nat) = 0 \implies False) \implies 0 < n$   
**by** (*rule neq0-conv[THEN iffD1]*, *iprover*)

**lemma** *gr0-conv-Suc*:  $(0 < n) = (\exists m. n = Suc \ m)$   
**by** (*fast intro: not0-implies-Suc*)

**lemma** *not-gr0* [*iff, noatp*]:  $!!n::nat. (\sim (0 < n)) = (n = 0)$   
**using** *neq0-conv* **by** *blast*

**lemma** *Suc-le-D*:  $(Suc\ n \leq m') ==> (?\ m. m' = Suc\ m)$   
**by** (*induct m'*) *simp-all*

Useful in certain inductive arguments

**lemma** *less-Suc-eq-0-disj*:  $(m < Suc\ n) = (m = 0 \mid (\exists j. m = Suc\ j \ \& \ j < n))$   
**by** (*cases m*) *simp-all*

#### 12.4.5 *min* and *max*

**lemma** *mono-Suc*: *mono Suc*  
**by** (*rule monoI*) *simp*

**lemma** *min-0L* [*simp*]:  $min\ 0\ n = (0::nat)$   
**by** (*rule min-leastL*) *simp*

**lemma** *min-0R* [*simp*]:  $min\ n\ 0 = (0::nat)$   
**by** (*rule min-leastR*) *simp*

**lemma** *min-Suc-Suc* [*simp*]:  $min\ (Suc\ m)\ (Suc\ n) = Suc\ (min\ m\ n)$   
**by** (*simp add: mono-Suc min-of-mono*)

**lemma** *min-Suc1*:  
 $min\ (Suc\ n)\ m = (case\ m\ of\ 0 => 0 \mid Suc\ m' => Suc\ (min\ n\ m'))$   
**by** (*simp split: nat.split*)

**lemma** *min-Suc2*:  
 $min\ m\ (Suc\ n) = (case\ m\ of\ 0 => 0 \mid Suc\ m' => Suc\ (min\ m'\ n))$   
**by** (*simp split: nat.split*)

**lemma** *max-0L* [*simp*]:  $max\ 0\ n = (n::nat)$   
**by** (*rule max-leastL*) *simp*

**lemma** *max-0R* [*simp*]:  $max\ n\ 0 = (n::nat)$   
**by** (*rule max-leastR*) *simp*

**lemma** *max-Suc-Suc* [*simp*]:  $max\ (Suc\ m)\ (Suc\ n) = Suc\ (max\ m\ n)$   
**by** (*simp add: mono-Suc max-of-mono*)

**lemma** *max-Suc1*:  
 $max\ (Suc\ n)\ m = (case\ m\ of\ 0 => Suc\ n \mid Suc\ m' => Suc\ (max\ n\ m'))$   
**by** (*simp split: nat.split*)

**lemma** *max-Suc2*:  
 $max\ m\ (Suc\ n) = (case\ m\ of\ 0 => Suc\ n \mid Suc\ m' => Suc\ (max\ m'\ n))$   
**by** (*simp split: nat.split*)

### 12.4.6 Monotonicity of Addition

**lemma** *Suc-pred* [simp]:  $n > 0 \implies \text{Suc } (n - \text{Suc } 0) = n$   
**by** (simp add: diff-Suc split: nat.split)

**lemma** *Suc-diff-1* [simp]:  $0 < n \implies \text{Suc } (n - 1) = n$   
**unfolding** *One-nat-def* **by** (rule *Suc-pred*)

**lemma** *nat-add-left-cancel-le* [simp]:  $(k + m \leq k + n) = (m \leq (n::\text{nat}))$   
**by** (induct k) simp-all

**lemma** *nat-add-left-cancel-less* [simp]:  $(k + m < k + n) = (m < (n::\text{nat}))$   
**by** (induct k) simp-all

**lemma** *add-gr-0* [iff]:  $!!m::\text{nat}. (m + n > 0) = (m > 0 \mid n > 0)$   
**by**(auto dest:gr0-implies-Suc)

strict, in 1st argument

**lemma** *add-less-mono1*:  $i < j \implies i + k < j + (k::\text{nat})$   
**by** (induct k) simp-all

strict, in both arguments

**lemma** *add-less-mono*:  $[[i < j; k < l]] \implies i + k < j + (l::\text{nat})$   
**apply** (rule *add-less-mono1* [THEN less-trans], assumption+)  
**apply** (induct j, simp-all)  
**done**

Deleted *less-natE*; use *less-imp-Suc-add RS exE*

**lemma** *less-imp-Suc-add*:  $m < n \implies (\exists k. n = \text{Suc } (m + k))$   
**apply** (induct n)  
**apply** (simp-all add: order-le-less)  
**apply** (blast elim!: less-SucE  
intro!: add-0-right [symmetric] add-Suc-right [symmetric])  
**done**

strict, in 1st argument; proof is by induction on  $k > 0$

**lemma** *mult-less-mono2*:  $(i::\text{nat}) < j \implies 0 < k \implies k * i < k * j$   
**apply**(auto simp: gr0-conv-Suc)  
**apply** (induct-tac m)  
**apply** (simp-all add: add-less-mono)  
**done**

The naturals form an ordered *comm-semiring-1-cancel*

**instance** *nat* :: *ordered-semidom*

**proof**

**fix**  $i\ j\ k :: \text{nat}$

**show**  $0 < (1::\text{nat})$  **by** simp

**show**  $i \leq j \implies k + i \leq k + j$  **by** simp

**show**  $i < j \implies 0 < k \implies k * i < k * j$  **by** (simp add: mult-less-mono2)

qed

instance nat :: no-zero-divisors

proof

fix a::nat and b::nat show  $a \sim 0 \implies b \sim 0 \implies a * b \sim 0$  by auto

qed

lemma nat-mult-1:  $(1::nat) * n = n$

by simp

lemma nat-mult-1-right:  $n * (1::nat) = n$

by simp

#### 12.4.7 Additional theorems about $op \leq$

Complete induction, aka course-of-values induction

instance nat :: wellorder proof

fix P and n :: nat

assume step:  $\bigwedge n::nat. (\bigwedge m. m < n \implies P m) \implies P n$

have  $\bigwedge q. q \leq n \implies P q$

proof (induct n)

case (0 n)

have  $P 0$  by (rule step) auto

thus ?case using 0 by auto

next

case (Suc m n)

then have  $n \leq m \vee n = \text{Suc } m$  by (simp add: le-Suc-eq)

thus ?case

proof

assume  $n \leq m$  thus  $P n$  by (rule Suc(1))

next

assume n:  $n = \text{Suc } m$

show  $P n$

by (rule step) (rule Suc(1), simp add: n le-simps)

qed

qed

then show  $P n$  by auto

qed

lemma Least-Suc:

$[| P n; \sim P 0 |] \implies (\text{LEAST } n. P n) = \text{Suc } (\text{LEAST } m. P (\text{Suc } m))$

apply (case-tac n, auto)

apply (frule LeastI)

apply (drule-tac  $P = \%x. P (\text{Suc } x)$  in LeastI)

apply (subgoal-tac  $(\text{LEAST } x. P x) \leq \text{Suc } (\text{LEAST } x. P (\text{Suc } x))$ )

apply (erule-tac [2] Least-le)

apply (case-tac  $\text{LEAST } x. P x$ , auto)

apply (drule-tac  $P = \%x. P (\text{Suc } x)$  in Least-le)

apply (blast intro: order-antisym)



done

**lemma** *Least-Suc2*:

$[|P\ n;\ Q\ m;\ \sim P\ 0;\ !k.\ P\ (Suc\ k) = Q\ k|] ==> Least\ P = Suc\ (Least\ Q)$   
**apply** (*erule* (1) *Least-Suc* [*THEN* *ssubst*])  
**apply** *simp*  
**done**

**lemma** *ex-least-nat-le*:  $\neg P(0) \implies P(n::nat) \implies \exists k \leq n. (\forall i < k. \neg P\ i) \ \&\ P(k)$

**apply** (*cases* *n*)  
**apply** *blast*  
**apply** (*rule-tac*  $x=LEAST\ k.\ P(k)$  **in** *exI*)  
**apply** (*blast* *intro*: *Least-le* *dest*: *not-less-Least* *intro*: *LeastI-ex*)  
**done**

**lemma** *ex-least-nat-less*:  $\neg P(0) \implies P(n::nat) \implies \exists k < n. (\forall i \leq k. \neg P\ i) \ \&\ P(k+1)$

**unfolding** *One-nat-def*  
**apply** (*cases* *n*)  
**apply** *blast*  
**apply** (*frule* (1) *ex-least-nat-le*)  
**apply** (*erule* *exE*)  
**apply** (*case-tac* *k*)  
**apply** *simp*  
**apply** (*rename-tac* *k1*)  
**apply** (*rule-tac*  $x=k1$  **in** *exI*)  
**apply** (*auto* *simp* *add*: *less-eq-Suc-le*)  
**done**

**lemma** *nat-less-induct*:

**assumes**  $!!n. \forall m::nat. m < n \longrightarrow P\ m ==> P\ n$  **shows**  $P\ n$   
**using** *assms* *less-induct* **by** *blast*

**lemma** *measure-induct-rule* [*case-names* *less*]:

**fixes**  $f :: 'a \Rightarrow nat$   
**assumes** *step*:  $\bigwedge x. (\bigwedge y. f\ y < f\ x \implies P\ y) \implies P\ x$   
**shows**  $P\ a$   
**by** (*induct*  $m \equiv f\ a$  *arbitrary*: *a rule*: *less-induct*) (*auto* *intro*: *step*)

old style induction rules:

**lemma** *measure-induct*:

**fixes**  $f :: 'a \Rightarrow nat$   
**shows**  $(\bigwedge x. \forall y. f\ y < f\ x \longrightarrow P\ y \implies P\ x) \implies P\ a$   
**by** (*rule* *measure-induct-rule* [*of*  $f\ P\ a$ ]) *iprover*

**lemma** *full-nat-induct*:

**assumes** *step*:  $(!!n. (ALL\ m. Suc\ m \leq n \longrightarrow P\ m) ==> P\ n)$   
**shows**  $P\ n$   
**by** (*rule* *less-induct*) (*auto* *intro*: *step* *simp*:*le-simps*)

An induction rule for establishing binary relations

```

lemma less-Suc-induct:
  assumes less:  $i < j$ 
    and step:  $!!i. P\ i\ (Suc\ i)$ 
    and trans:  $!!i\ j\ k. P\ i\ j \implies P\ j\ k \implies P\ i\ k$ 
  shows  $P\ i\ j$ 
proof –
  from less obtain  $k$  where  $j = Suc(i+k)$  by (auto dest: less-imp-Suc-add)
  have  $P\ i\ (Suc\ (i + k))$ 
  proof (induct k)
    case 0
    show ?case by (simp add: step)
  next
    case (Suc k)
    thus ?case by (auto intro: assms)
  qed
  thus  $P\ i\ j$  by (simp add: j)
qed

```

The method of infinite descent, frequently used in number theory. Provided by Roelof Oosterhuis.  $P(n)$  is true for all  $n \in \mathbb{N}$  if

- case “0”: given  $n = 0$  prove  $P(n)$ ,
- case “smaller”: given  $n > 0$  and  $\neg P(n)$  prove there exists a smaller integer  $m$  such that  $\neg P(m)$ .

A compact version without explicit base case:

```

lemma infinite-descent:
   $\llbracket !!n::nat. \neg P\ n \implies \exists m < n. \neg P\ m \rrbracket \implies P\ n$ 
by (induct n rule: less-induct, auto)

```

```

lemma infinite-descent0 [case-names 0 smaller]:
   $\llbracket P\ 0; !!n. n > 0 \implies \neg P\ n \implies (\exists m::nat. m < n \wedge \neg P\ m) \rrbracket \implies P\ n$ 
by (rule infinite-descent) (case-tac n>0, auto)

```

Infinite descent using a mapping to  $\mathbb{N}$ :  $P(x)$  is true for all  $x \in D$  if there exists a  $V : D \rightarrow \mathbb{N}$  and

- case “0”: given  $V(x) = 0$  prove  $P(x)$ ,
- case “smaller”: given  $V(x) > 0$  and  $\neg P(x)$  prove there exists a  $y \in D$  such that  $V(y) < V(x)$  and  $\neg P(y)$ .

NB: the proof also shows how to use the previous lemma.

```

corollary infinite-descent0-measure [case-names 0 smaller]:
  assumes A0:  $!!x. V\ x = (0::nat) \implies P\ x$ 
    and A1:  $!!x. V\ x > 0 \implies \neg P\ x \implies (\exists y. V\ y < V\ x \wedge \neg P\ y)$ 
  shows  $P\ x$ 

```

**proof** –  
**obtain**  $n$  **where**  $n = V\ x$  **by** *auto*  
**moreover have**  $\bigwedge x. V\ x = n \implies P\ x$   
**proof** (*induct n rule: infinite-descent0*)  
**case**  $0$  — i.e.  $V(x) = 0$   
**with**  $A0$  **show**  $P\ x$  **by** *auto*  
**next** — now  $n > 0$  and  $P(x)$  does not hold for some  $x$  with  $V(x) = n$   
**case** (*smaller n*)  
**then obtain**  $x$  **where**  $vxn: V\ x = n$  **and**  $V\ x > 0 \wedge \neg P\ x$  **by** *auto*  
**with**  $A1$  **obtain**  $y$  **where**  $V\ y < V\ x \wedge \neg P\ y$  **by** *auto*  
**with**  $vxn$  **obtain**  $m$  **where**  $m = V\ y \wedge m < n \wedge \neg P\ y$  **by** *auto*  
**then show** *?case* **by** *auto*  
**qed**  
**ultimately show**  $P\ x$  **by** *auto*  
**qed**

Again, without explicit base case:

**lemma** *infinite-descent-measure*:  
**assumes**  $!!x. \neg P\ x \implies \exists y. (V::'a \Rightarrow nat)\ y < V\ x \wedge \neg P\ y$  **shows**  $P\ x$   
**proof** –  
**from** *assms* **obtain**  $n$  **where**  $n = V\ x$  **by** *auto*  
**moreover have**  $!!x. V\ x = n \implies P\ x$   
**proof** (*induct n rule: infinite-descent, auto*)  
**fix**  $x$  **assume**  $\neg P\ x$   
**with** *assms* **show**  $\exists m < V\ x. \exists y. V\ y = m \wedge \neg P\ y$  **by** *auto*  
**qed**  
**ultimately show**  $P\ x$  **by** *auto*  
**qed**

A [clumsy] way of lifting  $<$  monotonicity to  $\leq$  monotonicity

**lemma** *less-mono-imp-le-mono*:  
 $\llbracket !!i\ j::nat. i < j \implies f\ i < f\ j; i \leq j \rrbracket \implies f\ i \leq ((f\ j)::nat)$   
**by** (*simp add: order-le-less*) (*blast*)

non-strict, in 1st argument

**lemma** *add-le-mono1*:  $i \leq j \implies i + k \leq j + (k::nat)$   
**by** (*rule add-right-mono*)

non-strict, in both arguments

**lemma** *add-le-mono*:  $\llbracket i \leq j; k \leq l \rrbracket \implies i + k \leq j + (l::nat)$   
**by** (*rule add-mono*)

**lemma** *le-add2*:  $n \leq ((m + n)::nat)$   
**by** (*insert add-right-mono [of 0 m n], simp*)

**lemma** *le-add1*:  $n \leq ((n + m)::nat)$   
**by** (*simp add: add-commute, rule le-add2*)

```

lemma less-add-Suc1:  $i < \text{Suc } (i + m)$ 
by (rule le-less-trans, rule le-add1, rule lessI)

lemma less-add-Suc2:  $i < \text{Suc } (m + i)$ 
by (rule le-less-trans, rule le-add2, rule lessI)

lemma less-iff-Suc-add:  $(m < n) = (\exists k. n = \text{Suc } (m + k))$ 
by (iprover intro!: less-add-Suc1 less-imp-Suc-add)

lemma trans-le-add1:  $(i::\text{nat}) \leq j \implies i \leq j + m$ 
by (rule le-trans, assumption, rule le-add1)

lemma trans-le-add2:  $(i::\text{nat}) \leq j \implies i \leq m + j$ 
by (rule le-trans, assumption, rule le-add2)

lemma trans-less-add1:  $(i::\text{nat}) < j \implies i < j + m$ 
by (rule less-le-trans, assumption, rule le-add1)

lemma trans-less-add2:  $(i::\text{nat}) < j \implies i < m + j$ 
by (rule less-le-trans, assumption, rule le-add2)

lemma add-lessD1:  $i + j < (k::\text{nat}) \implies i < k$ 
apply (rule le-less-trans [of - i+j])
apply (simp-all add: le-add1)
done

lemma not-add-less1 [iff]:  $\sim (i + j < (i::\text{nat}))$ 
apply (rule notI)
apply (drule add-lessD1)
apply (erule less-irrefl [THEN notE])
done

lemma not-add-less2 [iff]:  $\sim (j + i < (i::\text{nat}))$ 
by (simp add: add-commute)

lemma add-leD1:  $m + k \leq n \implies m \leq (n::\text{nat})$ 
apply (rule order-trans [of - m+k])
apply (simp-all add: le-add1)
done

lemma add-leD2:  $m + k \leq n \implies k \leq (n::\text{nat})$ 
apply (simp add: add-commute)
apply (erule add-leD1)
done

lemma add-leE:  $(m::\text{nat}) + k \leq n \implies (m \leq n \implies k \leq n \implies R) \implies R$ 
by (blast dest: add-leD1 add-leD2)

needs !!k for add-ac to work

```

**lemma** *less-add-eq-less*:  $!!k::nat. k < l ==> m + l = k + n ==> m < n$   
**by** (*force simp del: add-Suc-right*  
*simp add: less-iff-Suc-add add-Suc-right [symmetric] add-ac*)

#### 12.4.8 More results about difference

Addition is the inverse of subtraction: if  $n \leq m$  then  $n + (m - n) = m$ .

**lemma** *add-diff-inverse*:  $\sim m < n ==> n + (m - n) = (m::nat)$   
**by** (*induct m n rule: diff-induct simp-all*)

**lemma** *le-add-diff-inverse* [*simp*]:  $n \leq m ==> n + (m - n) = (m::nat)$   
**by** (*simp add: add-diff-inverse linorder-not-less*)

**lemma** *le-add-diff-inverse2* [*simp*]:  $n \leq m ==> (m - n) + n = (m::nat)$   
**by** (*simp add: add-commute*)

**lemma** *Suc-diff-le*:  $n \leq m ==> \text{Suc } m - n = \text{Suc } (m - n)$   
**by** (*induct m n rule: diff-induct simp-all*)

**lemma** *diff-less-Suc*:  $m - n < \text{Suc } m$   
**apply** (*induct m n rule: diff-induct*)  
**apply** (*erule-tac [3] less-SucE*)  
**apply** (*simp-all add: less-Suc-eq*)  
**done**

**lemma** *diff-le-self* [*simp*]:  $m - n \leq (m::nat)$   
**by** (*induct m n rule: diff-induct (simp-all add: le-SucI)*)

**lemma** *le-iff-add*:  $(m::nat) \leq n = (\exists k. n = m + k)$   
**by** (*auto simp: le-add1 dest!: le-add-diff-inverse sym [of - n]*)

**lemma** *less-imp-diff-less*:  $(j::nat) < k ==> j - n < k$   
**by** (*rule le-less-trans, rule diff-le-self*)

**lemma** *diff-Suc-less* [*simp*]:  $0 < n ==> n - \text{Suc } i < n$   
**by** (*cases n (auto simp add: le-simps)*)

**lemma** *diff-add-assoc*:  $k \leq (j::nat) ==> (i + j) - k = i + (j - k)$   
**by** (*induct j k rule: diff-induct simp-all*)

**lemma** *diff-add-assoc2*:  $k \leq (j::nat) ==> (j + i) - k = (j - k) + i$   
**by** (*simp add: add-commute diff-add-assoc*)

**lemma** *le-imp-diff-is-add*:  $i \leq (j::nat) ==> (j - i = k) = (j = k + i)$   
**by** (*auto simp add: diff-add-inverse2*)

**lemma** *diff-is-0-eq* [*simp*]:  $((m::nat) - n = 0) = (m \leq n)$   
**by** (*induct m n rule: diff-induct simp-all*)

**lemma** *diff-is-0-eq'* [simp]:  $m \leq n \implies (m::nat) - n = 0$   
**by** (rule iffD2, rule diff-is-0-eq)

**lemma** *zero-less-diff* [simp]:  $(0 < n - (m::nat)) = (m < n)$   
**by** (induct m n rule: diff-induct) simp-all

**lemma** *less-imp-add-positive*:  
**assumes**  $i < j$   
**shows**  $\exists k::nat. 0 < k \ \& \ i + k = j$   
**proof**  
**from** *assms* **show**  $0 < j - i \ \& \ i + (j - i) = j$   
**by** (simp add: order-less-imp-le)  
**qed**

a nice rewrite for bounded subtraction

**lemma** *nat-minus-add-max*:  
**fixes**  $n \ m :: nat$   
**shows**  $n - m + m = \max \ n \ m$   
**by** (simp add: max-def not-le order-less-imp-le)

**lemma** *nat-diff-split*:  
 $P(a - b::nat) = ((a < b \longrightarrow P \ 0) \ \& \ (ALL \ d. \ a = b + d \longrightarrow P \ d))$   
— elimination of  $-$  on *nat*  
**by** (cases  $a < b$ )  
(auto simp add: diff-is-0-eq [THEN iffD2] diff-add-inverse  
not-less le-less dest!: sym [of  $a$ ] sym [of  $b$ ] add-eq-self-zero)

**lemma** *nat-diff-split-asm*:  
 $P(a - b::nat) = (\sim (a < b \ \& \ \sim P \ 0 \mid (EX \ d. \ a = b + d \ \& \ \sim P \ d)))$   
— elimination of  $-$  on *nat* in assumptions  
**by** (auto split: nat-diff-split)

#### 12.4.9 Monotonicity of Multiplication

**lemma** *mult-le-mono1*:  $i \leq (j::nat) \implies i * k \leq j * k$   
**by** (simp add: mult-right-mono)

**lemma** *mult-le-mono2*:  $i \leq (j::nat) \implies k * i \leq k * j$   
**by** (simp add: mult-left-mono)

$\leq$  monotonicity, BOTH arguments

**lemma** *mult-le-mono*:  $i \leq (j::nat) \implies k \leq l \implies i * k \leq j * l$   
**by** (simp add: mult-mono)

**lemma** *mult-less-mono1*:  $(i::nat) < j \implies 0 < k \implies i * k < j * k$   
**by** (simp add: mult-strict-right-mono)

Differs from the standard *zero-less-mult-iff* in that there are no negative numbers.

```

lemma nat-0-less-mult-iff [simp]:  $(0 < (m::nat) * n) = (0 < m \ \& \ 0 < n)$ 
  apply (induct m)
  apply simp
  apply (case-tac n)
  apply simp-all
done

```

```

lemma one-le-mult-iff [simp]:  $(\text{Suc } 0 \leq m * n) = (\text{Suc } 0 \leq m \ \& \ \text{Suc } 0 \leq n)$ 
  apply (induct m)
  apply simp
  apply (case-tac n)
  apply simp-all
done

```

```

lemma mult-less-cancel2 [simp]:  $((m::nat) * k < n * k) = (0 < k \ \& \ m < n)$ 
  apply (safe intro!: mult-less-mono1)
  apply (case-tac k, auto)
  apply (simp del: le-0-eq add: linorder-not-le [symmetric])
  apply (blast intro: mult-le-mono1)
done

```

```

lemma mult-less-cancel1 [simp]:  $(k * (m::nat) < k * n) = (0 < k \ \& \ m < n)$ 
by (simp add: mult-commute [of k])

```

```

lemma mult-le-cancel1 [simp]:  $(k * (m::nat) \leq k * n) = (0 < k \ \longrightarrow \ m \leq n)$ 
by (simp add: linorder-not-less [symmetric], auto)

```

```

lemma mult-le-cancel2 [simp]:  $((m::nat) * k \leq n * k) = (0 < k \ \longrightarrow \ m \leq n)$ 
by (simp add: linorder-not-less [symmetric], auto)

```

```

lemma Suc-mult-less-cancel1:  $(\text{Suc } k * m < \text{Suc } k * n) = (m < n)$ 
by (subst mult-less-cancel1) simp

```

```

lemma Suc-mult-le-cancel1:  $(\text{Suc } k * m \leq \text{Suc } k * n) = (m \leq n)$ 
by (subst mult-le-cancel1) simp

```

```

lemma le-square:  $m \leq m * (m::nat)$ 
  by (cases m) (auto intro: le-add1)

```

```

lemma le-cube:  $(m::nat) \leq m * (m * m)$ 
  by (cases m) (auto intro: le-add1)

```

Lemma for *gcd*

```

lemma mult-eq-self-implies-10:  $(m::nat) = m * n \implies n = 1 \mid m = 0$ 
  apply (drule sym)
  apply (rule disjCI)
  apply (rule nat-less-cases, erule-tac [2] -)
  apply (drule-tac [2] mult-less-mono2)
  apply (auto)

```

```

done

the lattice order on nat
instantiation nat :: distrib-lattice
begin

definition
  (inf :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat) = min

definition
  (sup :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat) = max

instance by intro-classes
  (auto simp add: inf-nat-def sup-nat-def max-def not-le min-def
   intro: order-less-imp-le antisym elim!: order-trans order-less-trans)

end

```

## 12.5 Embedding of the Naturals into any *semiring-1*: *of-nat*

```

context semiring-1
begin

primrec
  of-nat :: nat  $\Rightarrow$  'a
where
  of-nat-0:    of-nat 0 = 0
  | of-nat-Suc: of-nat (Suc m) = 1 + of-nat m

lemma of-nat-1 [simp]: of-nat 1 = 1
  unfolding One-nat-def by simp

lemma of-nat-add [simp]: of-nat (m + n) = of-nat m + of-nat n
  by (induct m) (simp-all add: add-ac)

lemma of-nat-mult: of-nat (m * n) = of-nat m * of-nat n
  by (induct m) (simp-all add: add-ac left-distrib)

primrec of-nat-aux :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  nat  $\Rightarrow$  'a  $\Rightarrow$  'a where
  of-nat-aux inc 0 i = i
  | of-nat-aux inc (Suc n) i = of-nat-aux inc n (inc i) — tail recursive

lemma of-nat-code [code, code unfold, code inline del]:
  of-nat n = of-nat-aux ( $\lambda i. i + 1$ ) n 0
proof (induct n)
  case 0 then show ?case by simp
next
  case (Suc n)
  have  $\bigwedge i. \textit{of-nat-aux} (\lambda i. i + 1) \textit{n} (i + 1) = \textit{of-nat-aux} (\lambda i. i + 1) \textit{n} i + 1$ 

```



```

    by (induct n) simp-all
  from this [of 0] have of-nat-aux ( $\lambda i. i + 1$ ) n 1 = of-nat-aux ( $\lambda i. i + 1$ ) n 0
+ 1
    by simp
  with Suc show ?case by (simp add: add-commute)
qed

end

```

Class for unital semirings with characteristic zero. Includes non-ordered rings like the complex numbers.

```

class semiring-char-0 = semiring-1 +
  assumes of-nat-eq-iff [simp]: of-nat m = of-nat n  $\longleftrightarrow$  m = n
begin

```

Special cases where either operand is zero

```

lemma of-nat-0-eq-iff [simp, noatp]: 0 = of-nat n  $\longleftrightarrow$  0 = n
  by (rule of-nat-eq-iff [of 0, simplified])

```

```

lemma of-nat-eq-0-iff [simp, noatp]: of-nat m = 0  $\longleftrightarrow$  m = 0
  by (rule of-nat-eq-iff [of - 0, simplified])

```

```

lemma inj-of-nat: inj of-nat
  by (simp add: inj-on-def)

```

```

end

```

```

context ordered-semidom
begin

```

```

lemma zero-le-imp-of-nat: 0  $\leq$  of-nat m
  apply (induct m, simp-all)
  apply (erule order-trans)
  apply (rule ord-le-eq-trans [OF - add-commute])
  apply (rule less-add-one [THEN less-imp-le])
  done

```

```

lemma less-imp-of-nat-less: m < n  $\implies$  of-nat m < of-nat n
  apply (induct m n rule: diff-induct, simp-all)
  apply (insert add-less-le-mono [OF zero-less-one zero-le-imp-of-nat], force)
  done

```

```

lemma of-nat-less-imp-less: of-nat m < of-nat n  $\implies$  m < n
  apply (induct m n rule: diff-induct, simp-all)
  apply (insert zero-le-imp-of-nat)
  apply (force simp add: not-less [symmetric])
  done

```

```

lemma of-nat-less-iff [simp]: of-nat m < of-nat n  $\longleftrightarrow$  m < n

```

**by** (*blast intro: of-nat-less-imp-less less-imp-of-nat-less*)

**lemma** *of-nat-le-iff* [*simp*]:  $of\text{-}nat\ m \leq of\text{-}nat\ n \longleftrightarrow m \leq n$   
**by** (*simp add: not-less [symmetric] linorder-not-less [symmetric]*)

Every *ordered-semidom* has characteristic zero.

**subclass** *semiring-char-0*  
**proof qed** (*simp add: eq-iff order-eq-iff*)

Special cases where either operand is zero

**lemma** *of-nat-0-le-iff* [*simp*]:  $0 \leq of\text{-}nat\ n$   
**by** (*rule of-nat-le-iff [of 0, simplified]*)

**lemma** *of-nat-le-0-iff* [*simp, noatp*]:  $of\text{-}nat\ m \leq 0 \longleftrightarrow m = 0$   
**by** (*rule of-nat-le-iff [of - 0, simplified]*)

**lemma** *of-nat-0-less-iff* [*simp*]:  $0 < of\text{-}nat\ n \longleftrightarrow 0 < n$   
**by** (*rule of-nat-less-iff [of 0, simplified]*)

**lemma** *of-nat-less-0-iff* [*simp*]:  $\neg of\text{-}nat\ m < 0$   
**by** (*rule of-nat-less-iff [of - 0, simplified]*)

**end**

**context** *ring-1*  
**begin**

**lemma** *of-nat-diff*:  $n \leq m \implies of\text{-}nat\ (m - n) = of\text{-}nat\ m - of\text{-}nat\ n$   
**by** (*simp add: algebra-simps of-nat-add [symmetric]*)

**end**

**context** *ordered-idom*  
**begin**

**lemma** *abs-of-nat* [*simp*]:  $|of\text{-}nat\ n| = of\text{-}nat\ n$   
**unfolding** *abs-if* **by** *auto*

**end**

**lemma** *of-nat-id* [*simp*]:  $of\text{-}nat\ n = n$   
**by** (*induct n*) (*auto simp add: One-nat-def*)

**lemma** *of-nat-eq-id* [*simp*]:  $of\text{-}nat = id$   
**by** (*auto simp add: expand-fun-eq*)

## 12.6 The Set of Natural Numbers

**context** *semiring-1*

**begin**

**definition**

*Nats* :: 'a set **where**  
`[code del]: Nats = range of-nat`

**notation** (*xsymbols*)

*Nats* ( $\mathbb{N}$ )

**lemma** *of-nat-in-Nats* [*simp*]: *of-nat*  $n \in \mathbb{N}$   
**by** (*simp add: Nats-def*)

**lemma** *Nats-0* [*simp*]:  $0 \in \mathbb{N}$   
**apply** (*simp add: Nats-def*)  
**apply** (*rule range-eqI*)  
**apply** (*rule of-nat-0 [symmetric]*)  
**done**

**lemma** *Nats-1* [*simp*]:  $1 \in \mathbb{N}$   
**apply** (*simp add: Nats-def*)  
**apply** (*rule range-eqI*)  
**apply** (*rule of-nat-1 [symmetric]*)  
**done**

**lemma** *Nats-add* [*simp*]:  $a \in \mathbb{N} \implies b \in \mathbb{N} \implies a + b \in \mathbb{N}$   
**apply** (*auto simp add: Nats-def*)  
**apply** (*rule range-eqI*)  
**apply** (*rule of-nat-add [symmetric]*)  
**done**

**lemma** *Nats-mult* [*simp*]:  $a \in \mathbb{N} \implies b \in \mathbb{N} \implies a * b \in \mathbb{N}$   
**apply** (*auto simp add: Nats-def*)  
**apply** (*rule range-eqI*)  
**apply** (*rule of-nat-mult [symmetric]*)  
**done**

**end**

## 12.7 Further Arithmetic Facts Concerning the Natural Numbers

**lemma** *subst-equals*:

**assumes** *1*:  $t = s$  **and** *2*:  $u = t$   
**shows**  $u = s$   
**using** *2 1* **by** (*rule trans*)

**setup** *Arith-Data.setup*

**use** *Tools/nat-arith.ML*

```

declaration  $\langle\langle$  K Nat-Arith.setup  $\rangle\rangle$ 

use Tools/lin-arith.ML
declaration  $\langle\langle$  K Lin-Arith.setup  $\rangle\rangle$ 

lemmas [arith-split] = nat-diff-split split-min split-max

context order
begin

lemma lift-Suc-mono-le:
  assumes mono:  $!!n. f\ n \leq f(\text{Suc } n)$  and  $n \leq n'$ 
  shows  $f\ n \leq f\ n'$ 
proof (cases  $n < n'$ )
  case True
  thus ?thesis
    by (induct  $n\ n'$  rule: less-Suc-induct[consumes 1]) (auto intro: mono)
qed (insert  $\langle n \leq n' \rangle$ , auto) — trivial for  $n = n'$ 

lemma lift-Suc-mono-less:
  assumes mono:  $!!n. f\ n < f(\text{Suc } n)$  and  $n < n'$ 
  shows  $f\ n < f\ n'$ 
using  $\langle n < n' \rangle$ 
by (induct  $n\ n'$  rule: less-Suc-induct[consumes 1]) (auto intro: mono)

lemma lift-Suc-mono-less-iff:
   $(!!n. f\ n < f(\text{Suc } n)) \implies f(n) < f(m) \longleftrightarrow n < m$ 
by(blast intro: less-asym' lift-Suc-mono-less[of f]
    dest: linorder-not-less[THEN iffD1] le-eq-less-or-eq[THEN iffD1])

end

lemma mono-iff-le-Suc:  $\text{mono } f = (\forall n. f\ n \leq f(\text{Suc } n))$ 
unfolding mono-def
by (auto intro: lift-Suc-mono-le[of f])

lemma mono-nat-linear-lb:
   $(!!m\ n::\text{nat}. m < n \implies f\ m < f\ n) \implies f(m)+k \leq f(m+k)$ 
apply(induct-tac k)
  apply simp
  apply(erule-tac  $x=m+n$  in meta-allE)
  apply(erule-tac  $x=\text{Suc}(m+n)$  in meta-allE)
  apply simp
done

Subtraction laws, mostly by Clemens Ballarin

lemma diff-less-mono:  $[[\ a < (b::\text{nat});\ c \leq a\ ]\ ==>\ a-c < b-c$ 
by arith

```

**lemma** *less-diff-conv*:  $(i < j - k) = (i + k < (j :: nat))$   
**by** *arith*

**lemma** *le-diff-conv*:  $(j - k \leq (i :: nat)) = (j \leq i + k)$   
**by** *arith*

**lemma** *le-diff-conv2*:  $k \leq j \implies (i \leq j - k) = (i + k \leq (j :: nat))$   
**by** *arith*

**lemma** *diff-diff-cancel* [*simp*]:  $i \leq (n :: nat) \implies n - (n - i) = i$   
**by** *arith*

**lemma** *le-add-diff*:  $k \leq (n :: nat) \implies m \leq n + m - k$   
**by** *arith*

**lemma** *diff-less* [*simp*]:  $!!m :: nat. [0 < n; 0 < m] \implies m - n < m$   
**by** *arith*

Simplification of relational expressions involving subtraction

**lemma** *diff-diff-eq*:  $[k \leq m; k \leq (n :: nat)] \implies ((m - k) - (n - k)) = (m - n)$   
**by** (*simp split add: nat-diff-split*)

**lemma** *eq-diff-iff*:  $[k \leq m; k \leq (n :: nat)] \implies (m - k = n - k) = (m = n)$   
**by** (*auto split add: nat-diff-split*)

**lemma** *less-diff-iff*:  $[k \leq m; k \leq (n :: nat)] \implies (m - k < n - k) = (m < n)$   
**by** (*auto split add: nat-diff-split*)

**lemma** *le-diff-iff*:  $[k \leq m; k \leq (n :: nat)] \implies (m - k \leq n - k) = (m \leq n)$   
**by** (*auto split add: nat-diff-split*)

(Anti)Monotonicity of subtraction – by Stephan Merz

**lemma** *diff-le-mono*:  $m \leq (n :: nat) \implies (m - l) \leq (n - l)$   
**by** (*simp split add: nat-diff-split*)

**lemma** *diff-le-mono2*:  $m \leq (n :: nat) \implies (l - n) \leq (l - m)$   
**by** (*simp split add: nat-diff-split*)

**lemma** *diff-less-mono2*:  $[m < (n :: nat); m < l] \implies (l - n) < (l - m)$   
**by** (*simp split add: nat-diff-split*)

**lemma** *diffs0-imp-equal*:  $!!m :: nat. [m - n = 0; n - m = 0] \implies m = n$   
**by** (*simp split add: nat-diff-split*)

**lemma** *min-diff*:  $\min (m - (i :: nat)) (n - i) = \min m n - i$   
**unfolding** *min-def* **by** *auto*

**lemma** *inj-on-diff-nat*:

```

assumes k-le-n:  $\forall n \in N. k \leq (n::nat)$ 
shows inj-on  $(\lambda n. n - k) N$ 
proof (rule inj-onI)
  fix x y
  assume a:  $x \in N \ y \in N \ x - k = y - k$ 
  with k-le-n have  $x - k + k = y - k + k$  by auto
  with a k-le-n show  $x = y$  by auto
qed

```

Rewriting to pull differences out

```

lemma diff-diff-right [simp]:  $k \leq j \dashv\vdash i - (j - k) = i + (k::nat) - j$ 
by arith

```

```

lemma diff-Suc-diff-eq1 [simp]:  $k \leq j \implies m - Suc (j - k) = m + k - Suc j$ 
by arith

```

```

lemma diff-Suc-diff-eq2 [simp]:  $k \leq j \implies Suc (j - k) - m = Suc j - (k + m)$ 
by arith

```

Lemmas for ex/Factorization

```

lemma one-less-mult:  $[| Suc\ 0 < n; Suc\ 0 < m |] \implies Suc\ 0 < m * n$ 
by (cases m) auto

```

```

lemma n-less-m-mult-n:  $[| Suc\ 0 < n; Suc\ 0 < m |] \implies n < m * n$ 
by (cases m) auto

```

```

lemma n-less-n-mult-m:  $[| Suc\ 0 < n; Suc\ 0 < m |] \implies n < n * m$ 
by (cases m) auto

```

Specialized induction principles that work ”backwards”:

```

lemma inc-induct[consumes 1, case-names base step]:

```

```

  assumes less:  $i \leq j$ 
  assumes base:  $P\ j$ 
  assumes step:  $!!i. [| i < j; P\ (Suc\ i) |] \implies P\ i$ 
  shows  $P\ i$ 
  using less

```

```

proof (induct d == j - i arbitrary: i)

```

```

  case ( $0\ i$ )
  hence  $i = j$  by simp
  with base show ?case by simp

```

```

next

```

```

  case ( $Suc\ d\ i$ )
  hence  $i < j$   $P\ (Suc\ i)$ 
  by simp-all
  thus  $P\ i$  by (rule step)

```

```

qed

```

```

lemma strict-inc-induct[consumes 1, case-names base step]:

```

```

  assumes less:  $i < j$ 

```

```

assumes base: !!i. j = Suc i ==> P i
assumes step: !!i. [| i < j; P (Suc i) |] ==> P i
shows P i
using less
proof (induct d==j - i - 1 arbitrary: i)
  case (0 i)
    with <i < j> have j = Suc i by simp
    with base show ?case by simp
  next
    case (Suc d i)
    hence i < j P (Suc i)
      by simp-all
    thus P i by (rule step)
qed

lemma zero-induct-lemma: P k ==> (!!n. P (Suc n) ==> P n) ==> P (k - i)
  using inc-induct[of k - i k P, simplified] by blast

lemma zero-induct: P k ==> (!!n. P (Suc n) ==> P n) ==> P 0
  using inc-induct[of 0 k P] by blast

lemma nat-not-singleton: (∀ x. x = (0::nat)) = False
  by auto

```

```

lemmas add-diff-assoc = diff-add-assoc [symmetric]
lemmas add-diff-assoc2 = diff-add-assoc2[symmetric]
declare diff-diff-left [simp] add-diff-assoc [simp] add-diff-assoc2[simp]

```

At present we prove no analogue of *not-less-Least* or *Least-Suc*, since there appears to be no need.

## 12.8 size of a datatype value

```

class size =
  fixes size :: 'a ⇒ nat — see further theory Wellfounded
end

```

## 13 Product-Type: Cartesian products

```

theory Product-Type
imports Inductive
uses
  (Tools/split-rule.ML)
  (Tools/inductive-set-package.ML)
  (Tools/inductive-realizer.ML)
  (Tools/datatype-realizer.ML)

```

**begin**

### 13.1 *bool* is a datatype

**rep-datatype** *True False* **by** (*auto intro: bool-induct*)

**declare** *case-split* [*cases type: bool*]  
— prefer plain propositional version

**lemma**

**shows** [*code*]: *eq-class.eq False P*  $\longleftrightarrow \neg P$   
**and** [*code*]: *eq-class.eq True P*  $\longleftrightarrow P$   
**and** [*code*]: *eq-class.eq P False*  $\longleftrightarrow \neg P$   
**and** [*code*]: *eq-class.eq P True*  $\longleftrightarrow P$   
**and** [*code nbe*]: *eq-class.eq P P*  $\longleftrightarrow \text{True}$   
**by** (*simp-all add: eq*)

**code-const** *eq-class.eq* :: *bool*  $\Rightarrow$  *bool*  $\Rightarrow$  *bool*  
(*Haskell infixl 4 ==*)

**code-instance** *bool* :: *eq*  
(*Haskell -*)

### 13.2 Unit

**typedef** *unit* = { *True* }

**proof**

**show** *True* : ?*unit* ..

**qed**

**definition**

*Unity* :: *unit*    (*'()*)

**where**

*()* = *Abs-unit True*

**lemma** *unit-eq* [*noatp*]: *u* = *()*

**by** (*induct u*) (*simp add: unit-def Unity-def*)

Simplification procedure for *unit-eq*. Cannot use this rule directly — it loops!

**ML**  $\langle\langle$

*val unit-eq-proc* =

*let val unit-meta-eq* = *mk-meta-eq* @{*thm unit-eq*} *in*

*Simplifier.simproc* @{*theory*} *unit-eq* [*x::unit*]

(*fn* -  $\Rightarrow$  *fn* -  $\Rightarrow$  *fn t*  $\Rightarrow$  *if HOLogic.is-unit t then NONE else SOME*

*unit-meta-eq*)

*end*;

*Addsimprocs* [*unit-eq-proc*];



»

**rep-datatype** () *by simp*

**lemma** *unit-all-eq1*: (!*x*::*unit*. *PROP P x*) == *PROP P* ()  
*by simp*

**lemma** *unit-all-eq2*: (!*x*::*unit*. *PROP P*) == *PROP P*  
*by (rule triv-forall-equality)*

This rewrite counters the effect of *unit-eq-proc* on %*u*::*unit*. *f u*, replacing it by *f* rather than by %*u*. *f* ().

**lemma** *unit-abs-eta-conv* [*simp, noatp*]: (%*u*::*unit*. *f* ()) = *f*  
*by (rule ext) simp*

code generator setup

**instance** *unit* :: *eq* ..

**lemma** [*code*]:  
*eq-class.eq (u::unit) v*  $\longleftrightarrow$  *True* **unfolding** *eq unit-eq* [*of u*] *unit-eq* [*of v*] **by**  
*rule+*

**code-type** *unit*  
 (*SML unit*)  
 (*OCaml unit*)  
 (*Haskell* ())

**code-instance** *unit* :: *eq*  
 (*Haskell* −)

**code-const** *eq-class.eq* :: *unit*  $\Rightarrow$  *unit*  $\Rightarrow$  *bool*  
 (*Haskell infixl 4 ==*)

**code-const** *Unity*  
 (*SML* ())  
 (*OCaml* ())  
 (*Haskell* ())

**code-reserved** *SML*  
*unit*

**code-reserved** *OCaml*  
*unit*

### 13.3 Pairs

#### 13.3.1 Product type, basic operations and concrete syntax definition

```

Pair-Rep :: 'a ⇒ 'b ⇒ 'a ⇒ 'b ⇒ bool
where
  Pair-Rep a b = (λx y. x = a ∧ y = b)

global

typedef (Prod)
  ('a, 'b) * (infixr * 20)
  = {f. ∃ a b. f = Pair-Rep (a::'a) (b::'b)}
proof
  fix a b show Pair-Rep a b ∈ ?Prod
  by rule+
qed

```

```

syntax (xsymbols)
  *      :: [type, type] => type          ((- × / -) [21, 20] 20)
syntax (HTML output)
  *      :: [type, type] => type          ((- × / -) [21, 20] 20)

```

```

consts
  Pair      :: 'a ⇒ 'b ⇒ 'a × 'b
  fst       :: 'a × 'b ⇒ 'a
  snd       :: 'a × 'b ⇒ 'b
  split     :: ('a ⇒ 'b ⇒ 'c) ⇒ 'a × 'b ⇒ 'c
  curry     :: ('a × 'b ⇒ 'c) ⇒ 'a ⇒ 'b ⇒ 'c

```

**local**

```

defs
  Pair-def:   Pair a b == Abs-Prod (Pair-Rep a b)
  fst-def:    fst p == THE a. EX b. p = Pair a b
  snd-def:    snd p == THE b. EX a. p = Pair a b
  split-def:  split == (%c p. c (fst p) (snd p))
  curry-def:  curry == (%c x y. c (Pair x y))

```

Patterns – extends pre-defined type *pttrn* used in abstractions.

**nonterminals**

*tuple-args patterns*

```

syntax
  -tuple      :: 'a => tuple-args => 'a * 'b          ((1'(-, / -)))
  -tuple-arg  :: 'a => tuple-args                      (-)
  -tuple-args :: 'a => tuple-args => tuple-args        (-, / -)
  -pattern    :: [pttrn, patterns] => pttrn           ('(-, / -'))
               :: pttrn => patterns                     (-)
  -patterns   :: [pttrn, patterns] => patterns        (-, / -)

```

**translations**

$(x, y) == \text{Pair } x \ y$

```

-tuple x (-tuple-args y z) == -tuple x (-tuple-arg (-tuple y z))
%(x,y,zs).b == split(%x (y,zs).b)
%(x,y).b == split(%x y. b)
-abs (Pair x y) t => %(x,y).t

```

```

print-translation <<
let fun split-tr' [Abs (x,T,t as (Abs abs))] =
  (* split (%x y. t) => %(x,y) t *)
  let val (y,t') = atomic-abs-tr' abs;
      val (x',t'') = atomic-abs-tr' (x,T,t');

  in Syntax.const -abs $ (Syntax.const -pattern $x'$y) $ t'' end
| split-tr' [Abs (x,T,(s as Const (split,-)$t))] =
  (* split (%x. (split (%y z. t))) => %(x,y,z). t *)
  let val (Const (-abs,-)$ (Const (-pattern,-)$y$z)$t') = split-tr' [t];
      val (x',t'') = atomic-abs-tr' (x,T,t');
  in Syntax.const -abs$
    (Syntax.const -pattern$x'$(Syntax.const -patterns$y$z))$t'' end
| split-tr' [Const (split,-)$t] =
  (* split (split (%x y z. t)) => %((x,y),z). t *)
  split-tr' [(split-tr' [t])] (* inner split-tr' creates next pattern *)
| split-tr' [Const (-abs,-)$x-y$(Abs abs)] =
  (* split (%pttrn z. t) => %(pttrn,z). t *)
  let val (z,t) = atomic-abs-tr' abs;
  in Syntax.const -abs $ (Syntax.const -pattern $x-y$z) $ t end
| split-tr' - = raise Match;
in [(split, split-tr')]
end
>>

```

```

typed-print-translation <<
let
  fun split-guess-names-tr' - T [Abs (x,-,Abs -)] = raise Match
  | split-guess-names-tr' - T [Abs (x,xT,t)] =
    (case (head-of t) of
      Const (split,-) => raise Match
    | - => let
        val (:::yT::-) = binder-types (domain-type T) handle Bind => raise
Match;

        val (y,t') = atomic-abs-tr' (y,yT,(incr-boundvars 1 t)$Bound 0);
        val (x',t'') = atomic-abs-tr' (x,xT,t');
        in Syntax.const -abs $ (Syntax.const -pattern $x'$y) $ t'' end)
  | split-guess-names-tr' - T [t] =
    (case (head-of t) of
      Const (split,-) => raise Match

```

```

      | - => let
        val (xT::yT::-) = binder-types (domain-type T) handle Bind =>
raise Match;
        val (y,t') =
          atomic-abs-tr' (y,yT,(incr-boundvars 2 t)$Bound 1$Bound 0);
        val (x',t'') = atomic-abs-tr' (x,xT,t');
        in Syntax.const -abs $ (Syntax.const -pattern $x'$y) $ t'' end)
      | split-guess-names-tr' - - = raise Match;
in [(split, split-guess-names-tr')]
end
>>

```

Towards a datatype declaration

```

lemma surj-pair [simp]: EX x y. p = (x, y)
  apply (unfold Pair-def)
  apply (rule Rep-Prod [unfolded Prod-def, THEN CollectE])
  apply (erule exE, erule exE, rule exI, rule exI)
  apply (rule Rep-Prod-inverse [symmetric, THEN trans])
  apply (erule arg-cong)
done

```

```

lemma PairE [cases type: *]:
  obtains x y where p = (x, y)
  using surj-pair [of p] by blast

```

```

lemma ProdI: Pair-Rep a b ∈ Prod
  unfolding Prod-def by rule+

```

```

lemma Pair-Rep-inject: Pair-Rep a b = Pair-Rep a' b' ⟹ a = a' ∧ b = b'
  unfolding Pair-Rep-def by (drule fun-cong, drule fun-cong) blast

```

```

lemma inj-on-Abs-Prod: inj-on Abs-Prod Prod
  apply (rule inj-on-inverseI)
  apply (erule Abs-Prod-inverse)
done

```

```

lemma Pair-inject:
  assumes (a, b) = (a', b')
  and a = a' ⟹ b = b' ⟹ R
  shows R
  apply (insert prems [unfolded Pair-def])
  apply (rule inj-on-Abs-Prod [THEN inj-onD, THEN Pair-Rep-inject, THEN
conjE])
  apply (assumption | rule ProdI)+
done

```

```

rep-datatype (prod) Pair
proof -
  fix P p

```

```

  assume  $\bigwedge x y. P (x, y)$ 
  then show  $P p$  by (cases p) simp
qed (auto elim: Pair-inject)

```

```

lemmas Pair-eq = prod.inject

```

```

lemma fst-conv [simp, code]:  $\text{fst } (a, b) = a$ 
  unfolding fst-def by blast

```

```

lemma snd-conv [simp, code]:  $\text{snd } (a, b) = b$ 
  unfolding snd-def by blast

```

### 13.3.2 Basic rules and proof tools

```

lemma fst-eqD:  $\text{fst } (x, y) = a \implies x = a$ 
  by simp

```

```

lemma snd-eqD:  $\text{snd } (x, y) = a \implies y = a$ 
  by simp

```

```

lemma pair-collapse [simp]:  $(\text{fst } p, \text{snd } p) = p$ 
  by (cases p) simp

```

```

lemmas surjective-pairing = pair-collapse [symmetric]

```

```

lemma split-paired-all:  $(!!x. PROP P x) == (!!a b. PROP P (a, b))$ 
proof
  fix a b
  assume !!x. PROP P x
  then show PROP P (a, b) .
next
  fix x
  assume !!a b. PROP P (a, b)
  from  $\langle PROP P (\text{fst } x, \text{snd } x) \rangle$  show PROP P x by simp
qed

```

The rule *split-paired-all* does not work with the Simplifier because it also affects premises in congruence rules, where this can lead to premises of the form  $!!a b. \dots = ?P(a, b)$  which cannot be solved by reflexivity.

```

lemmas split-tupled-all = split-paired-all unit-all-eq2

```

```

ML <<
  (* replace parameters of product type by individual component parameters *)
  val safe-full-simp-tac = generic-simp-tac true (true, false, false);
  local (* filtering with exists-paired-all is an essential optimization *)
    fun exists-paired-all (Const (all, -) $ Abs (-, T, t)) =
      can HOLogic.dest-prodT T orelse exists-paired-all t
    | exists-paired-all (t $ u) = exists-paired-all t orelse exists-paired-all u
    | exists-paired-all (Abs (-, -, t)) = exists-paired-all t

```

```

    | exists-paired-all - = false;
    val ss = HOL-basic-ss
    addsimps [@{thm split-paired-all}, @{thm unit-all-eq2}, @{thm unit-abs-eta-conv}]
    addsimprocs [unit-eq-proc];
  in
    val split-all-tac = SUBGOAL (fn (t, i) =>
      if exists-paired-all t then safe-full-simp-tac ss i else no-tac);
    val unsafe-split-all-tac = SUBGOAL (fn (t, i) =>
      if exists-paired-all t then full-simp-tac ss i else no-tac);
    fun split-all th =
      if exists-paired-all (Thm.prop-of th) then full-simplify ss th else th;
    end;
  >>

```

```

declaration << fn - =>
  Classical.map-cs (fn cs => cs addSbefore (split-all-tac, split-all-tac))
>>

```

**lemma** *split-paired-All* [simp]:  $(ALL\ x.\ P\ x) = (ALL\ a\ b.\ P\ (a,\ b))$   
 — [iff] is not a good idea because it makes *blast* loop  
 by *fast*

**lemma** *split-paired-Ex* [simp]:  $(EX\ x.\ P\ x) = (EX\ a\ b.\ P\ (a,\ b))$   
 by *fast*

**lemma** *Pair-fst-snd-eq*:  $s = t \longleftrightarrow fst\ s = fst\ t \wedge snd\ s = snd\ t$   
 by (cases *s*, cases *t*) *simp*

**lemma** *prod-eqI* [intro?]:  $fst\ p = fst\ q \implies snd\ p = snd\ q \implies p = q$   
 by (simp add: *Pair-fst-snd-eq*)

### 13.3.3 split and curry

**lemma** *split-conv* [simp, code]:  $split\ f\ (a,\ b) = f\ a\ b$   
 by (simp add: *split-def*)

**lemma** *curry-conv* [simp, code]:  $curry\ f\ a\ b = f\ (a,\ b)$   
 by (simp add: *curry-def*)

**lemmas** *split* = *split-conv* — for backwards compatibility

**lemma** *splitI*:  $f\ a\ b \implies split\ f\ (a,\ b)$   
 by (rule *split-conv* [THEN *iffD2*])

**lemma** *splitD*:  $split\ f\ (a,\ b) \implies f\ a\ b$   
 by (rule *split-conv* [THEN *iffD1*])

**lemma** *curryI* [intro!]:  $f\ (a,\ b) \implies curry\ f\ a\ b$   
 by (simp add: *curry-def*)

```

lemma curryD [dest!]: curry f a b  $\implies$  f (a, b)
  by (simp add: curry-def)

lemma curryE: curry f a b  $\implies$  (f (a, b)  $\implies$  Q)  $\implies$  Q
  by (simp add: curry-def)

lemma curry-split [simp]: curry (split f) = f
  by (simp add: curry-def split-def)

lemma split-curry [simp]: split (curry f) = f
  by (simp add: curry-def split-def)

lemma split-Pair [simp]: ( $\lambda(x, y). (x, y)$ ) = id
  by (simp add: split-def id-def)

lemma split-eta: ( $\lambda(x, y). f (x, y)$ ) = f
  — Subsumes the old split-Pair when f is the identity function.
  by (rule ext) auto

lemma split-comp: split (f  $\circ$  g) x = f (g (fst x)) (snd x)
  by (cases x) simp

lemma split-twice: split f (split g p) = split ( $\lambda x y. \text{split } f (g x y)$ ) p
  unfolding split-def ..

lemma split-paired-The: (THE x. P x) = (THE (a, b). P (a, b))
  — Can’t be added to simpset: loops!
  by (simp add: split-eta)

lemma The-split: The (split P) = (THE xy. P (fst xy) (snd xy))
  by (simp add: split-def)

lemma split-weak-cong: p = q  $\implies$  split c p = split c q
  — Prevents simplification of c: much faster
  by (erule arg-cong)

lemma cond-split-eta: ( $!!x y. f x y = g (x, y)$ )  $\implies$  ( $\%(x, y). f x y$ ) = g
  by (simp add: split-eta)

```

Simplification procedure for *cond-split-eta*. Using *split-eta* as a rewrite rule is not general enough, and using *cond-split-eta* directly would render some existing proofs very inefficient; similarly for *split-beta*.

ML  $\ll$

*local*

```

  val cond-split-eta-ss = HOL-basic-ss addsimps [thm cond-split-eta]
  fun Pair-pat k 0 (Bound m) = (m = k)
  |   Pair-pat k i (Const (Pair, -) $ Bound m $ t) = i > 0 andalso

```

```

      m = k+i andalso Pair-pat k (i-1) t
|   Pair-pat - - = false;
fun no-args k i (Abs (-, -, t)) = no-args (k+1) i t
|   no-args k i (t $ u) = no-args k i t andalso no-args k i u
|   no-args k i (Bound m) = m < k orelse m > k+i
|   no-args - - = true;
fun split-pat tp i (Abs (-, -, t)) = if tp 0 i t then SOME (i,t) else NONE
|   split-pat tp i (Const (split, -) $ Abs (-, -, t)) = split-pat tp (i+1) t
|   split-pat tp i - = NONE;
fun metaeq ss lhs rhs = mk-meta-eq (Goal.prove (Simplifier.the-context ss) [] []
  (HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs)))
  (K (simp-tac (Simplifier.inherit-context ss cond-split-eta-ss) 1)));

fun beta-term-pat k i (Abs (-, -, t)) = beta-term-pat (k+1) i t
|   beta-term-pat k i (t $ u) = Pair-pat k i (t $ u) orelse
  (beta-term-pat k i t andalso beta-term-pat k i u)
|   beta-term-pat k i t = no-args k i t;
fun eta-term-pat k i (f $ arg) = no-args k i f andalso Pair-pat k i arg
|   eta-term-pat - - = false;
fun subst arg k i (Abs (x, T, t)) = Abs (x, T, subst arg (k+1) i t)
|   subst arg k i (t $ u) = if Pair-pat k i (t $ u) then incr-boundvars k arg
  else (subst arg k i t $ subst arg k i u)
|   subst arg k i t = t;
fun beta-proc ss (s as Const (split, -) $ Abs (-, -, t) $ arg) =
  (case split-pat beta-term-pat 1 t of
    SOME (i,f) => SOME (metaeq ss s (subst arg 0 i f))
  | NONE => NONE)
|   beta-proc - - = NONE;
fun eta-proc ss (s as Const (split, -) $ Abs (-, -, t)) =
  (case split-pat eta-term-pat 1 t of
    SOME (-,ft) => SOME (metaeq ss s (let val (f $ arg) = ft in f end))
  | NONE => NONE)
|   eta-proc - - = NONE;
in
  val split-beta-proc = Simplifier.simproc (the-context ()) split-beta [split f z] (K
beta-proc);
  val split-eta-proc = Simplifier.simproc (the-context ()) split-eta [split f] (K eta-proc);
end;

Addsimprocs [split-beta-proc, split-eta-proc];
>>

```

**lemma** *split-beta* [mono]:  $(\%(x, y). P\ x\ y)\ z = P\ (fst\ z)\ (snd\ z)$   
**by** (subst surjective-pairing, rule split-conv)

**lemma** *split-split* [noatp]:  $R(split\ c\ p) = (ALL\ x\ y. p = (x, y) \longrightarrow R(c\ x\ y))$   
— For use with *split* and the Simplifier.  
**by** (insert surj-pair [of p], clarify, simp)

*split-split* could be declared as [split] done after the Splitter has been speeded



up significantly; precompute the constants involved and don’t do anything unless the current goal contains one of those constants.

**lemma** *split-split-asm* [noatp]:  $R \text{ (split } c \text{ } p) = (\sim (EX \ x \ y. \ p = (x, y) \ \& \ (\sim R \ (c \ x \ y))))$   
**by** (*subst split-split, simp*)

*split* used as a logical connective or set former.

These rules are for use with *blast*; could instead call *simp* using *split* as rewrite.

**lemma** *splitI2*:  $!!p. [\![ \![a \ b. \ p = (a, b) \implies c \ a \ b] \!] \implies \text{split } c \ p]$   
**apply** (*simp only: split-tupled-all*)  
**apply** (*simp (no-asm-simp)*)  
**done**

**lemma** *splitI2'*:  $!!p. [\![ \![a \ b. \ (a, b) = p \implies c \ a \ b \ x] \!] \implies \text{split } c \ p \ x]$   
**apply** (*simp only: split-tupled-all*)  
**apply** (*simp (no-asm-simp)*)  
**done**

**lemma** *splitE*:  $\text{split } c \ p \implies (!x \ y. \ p = (x, y) \implies c \ x \ y \implies Q) \implies Q$   
**by** (*induct p*) (*auto simp add: split-def*)

**lemma** *splitE'*:  $\text{split } c \ p \ z \implies (!x \ y. \ p = (x, y) \implies c \ x \ y \ z \implies Q) \implies Q$   
**by** (*induct p*) (*auto simp add: split-def*)

**lemma** *splitE2*:  
 $[\![ \ Q \ (\text{split } P \ z); \ !x \ y. \ [z = (x, y); \ Q \ (P \ x \ y)] \implies R \ ] \implies R$   
**proof** –  
**assume** *q*:  $Q \ (\text{split } P \ z)$   
**assume** *r*:  $!x \ y. \ [z = (x, y); \ Q \ (P \ x \ y)] \implies R$   
**show** *R*  
**apply** (*rule r surjective-pairing*) +  
**apply** (*rule split-beta [THEN subst], rule q*)  
**done**  
**qed**

**lemma** *splitD'*:  $\text{split } R \ (a, b) \ c \implies R \ a \ b \ c$   
**by** *simp*

**lemma** *mem-splitI*:  $z: c \ a \ b \implies z: \text{split } c \ (a, b)$   
**by** *simp*

**lemma** *mem-splitI2*:  $!!p. [\![ \![a \ b. \ p = (a, b) \implies z: c \ a \ b] \!] \implies z: \text{split } c \ p]$   
**by** (*simp only: split-tupled-all, simp*)

**lemma** *mem-splitE*:  
**assumes** *major*:  $z: \text{split } c \ p$

```

and cases: !!x y. [| p = (x,y); z: c x y |] ==> Q
shows Q
by (rule major [unfolded split-def] cases surjective-pairing)+

declare mem-splitI2 [intro!] mem-splitI [intro!] splitI2' [intro!] splitI2 [intro!] splitI
[intro!]
declare mem-splitE [elim!] splitE' [elim!] splitE [elim!]

ML <<
  local (* filtering with exists-p-split is an essential optimization *)
    fun exists-p-split (Const (split,-) $ - $ (Const (Pair,-)$-$)) = true
      | exists-p-split (t $ u) = exists-p-split t orelse exists-p-split u
      | exists-p-split (Abs (-, -, t)) = exists-p-split t
      | exists-p-split - = false;
    val ss = HOL-basic-ss addsimps [thm split-conv];
  in
    val split-conv-tac = SUBGOAL (fn (t, i) =>
      if exists-p-split t then safe-full-simp-tac ss i else no-tac);
  end;
>>

declaration << fn - =>
  Classical.map-cs (fn cs => cs addSbefore (split-conv-tac, split-conv-tac))
>>

lemma split-eta-SetCompr [simp,noatp]: (%u. EX x y. u = (x, y) & P (x, y)) =
P
by (rule ext) fast

lemma split-eta-SetCompr2 [simp,noatp]: (%u. EX x y. u = (x, y) & P x y) =
split P
by (rule ext) fast

lemma split-part [simp]: (%(a,b). P & Q a b) = (%ab. P & split Q ab)
— Allows simplifications of nested splits in case of independent predicates.
by (rule ext) blast

lemma split-comp-eq:
  fixes f :: 'a => 'b => 'c and g :: 'd => 'a
  shows (%u. f (g (fst u)) (snd u)) = (split (%x. f (g x)))
  by (rule ext) auto

lemma pair-imageI [intro]: (a, b) : A ==> f a b : (%(a, b). f a b) ‘ A
apply (rule-tac x = (a, b) in image-eqI)
apply auto
done

```

**lemma** *The-split-eq* [*simp*]: (*THE* ( $x', y'$ ).  $x = x' \ \& \ y = y'$ ) = ( $x, y$ )  
**by** *blast*

Setup of internal *split-rule*.

**definition**

*internal-split* :: ( $'a \Rightarrow 'b \Rightarrow 'c$ )  $\Rightarrow 'a \times 'b \Rightarrow 'c$

**where**

*internal-split* == *split*

**lemma** *internal-split-conv*: *internal-split*  $c \ (a, b) = c \ a \ b$   
**by** (*simp only: internal-split-def split-conv*)

**hide** *const internal-split*

**use** *Tools/split-rule.ML*

**setup** *SplitRule.setup*

**lemmas** *prod-caseI* = *prod.cases* [*THEN iffD2, standard*]

**lemma** *prod-caseI2*:  $!!p. [\![\![a \ b. \ p = (a, b) \implies c \ a \ b]\!]\!] \implies \text{prod-case } c \ p$   
**by** *auto*

**lemma** *prod-caseI2'*:  $!!p. [\![\![a \ b. \ (a, b) = p \implies c \ a \ b \ x]\!]\!] \implies \text{prod-case } c \ p \ x$   
**by** (*auto simp: split-tupled-all*)

**lemma** *prod-caseE*:  $\text{prod-case } c \ p \implies (!!x \ y. \ p = (x, y) \implies c \ x \ y \implies Q) \implies Q$   
**by** (*induct p*) *auto*

**lemma** *prod-caseE'*:  $\text{prod-case } c \ p \ z \implies (!!x \ y. \ p = (x, y) \implies c \ x \ y \ z \implies Q) \implies Q$   
**by** (*induct p*) *auto*

**lemma** *prod-case-unfold*:  $\text{prod-case} = (\%c \ p. \ c \ (\text{fst } p) \ (\text{snd } p))$   
**by** (*simp add: expand-fun-eq*)

**declare** *prod-caseI2'* [*intro!*] *prod-caseI2* [*intro!*] *prod-caseI* [*intro!*]  
**declare** *prod-caseE'* [*elim!*] *prod-caseE* [*elim!*]

**lemma** *prod-case-split*:

*prod-case* = *split*

**by** (*auto simp add: expand-fun-eq*)

**lemma** *prod-case-beta*:

*prod-case*  $f \ p = f \ (\text{fst } p) \ (\text{snd } p)$

**unfolding** *prod-case-split split-beta* ..

### 13.4 Further cases/induct rules for tuples

**lemma** *prod-cases3* [*cases type*]:  
**obtains** (*fields*)  $a\ b\ c$  **where**  $y = (a, b, c)$   
**by** (*cases y, case-tac b*) *blast*

**lemma** *prod-induct3* [*case-names fields, induct type*]:  
 $(!!a\ b\ c. P\ (a, b, c)) \implies P\ x$   
**by** (*cases x*) *blast*

**lemma** *prod-cases4* [*cases type*]:  
**obtains** (*fields*)  $a\ b\ c\ d$  **where**  $y = (a, b, c, d)$   
**by** (*cases y, case-tac c*) *blast*

**lemma** *prod-induct4* [*case-names fields, induct type*]:  
 $(!!a\ b\ c\ d. P\ (a, b, c, d)) \implies P\ x$   
**by** (*cases x*) *blast*

**lemma** *prod-cases5* [*cases type*]:  
**obtains** (*fields*)  $a\ b\ c\ d\ e$  **where**  $y = (a, b, c, d, e)$   
**by** (*cases y, case-tac d*) *blast*

**lemma** *prod-induct5* [*case-names fields, induct type*]:  
 $(!!a\ b\ c\ d\ e. P\ (a, b, c, d, e)) \implies P\ x$   
**by** (*cases x*) *blast*

**lemma** *prod-cases6* [*cases type*]:  
**obtains** (*fields*)  $a\ b\ c\ d\ e\ f$  **where**  $y = (a, b, c, d, e, f)$   
**by** (*cases y, case-tac e*) *blast*

**lemma** *prod-induct6* [*case-names fields, induct type*]:  
 $(!!a\ b\ c\ d\ e\ f. P\ (a, b, c, d, e, f)) \implies P\ x$   
**by** (*cases x*) *blast*

**lemma** *prod-cases7* [*cases type*]:  
**obtains** (*fields*)  $a\ b\ c\ d\ e\ f\ g$  **where**  $y = (a, b, c, d, e, f, g)$   
**by** (*cases y, case-tac f*) *blast*

**lemma** *prod-induct7* [*case-names fields, induct type*]:  
 $(!!a\ b\ c\ d\ e\ f\ g. P\ (a, b, c, d, e, f, g)) \implies P\ x$   
**by** (*cases x*) *blast*

#### 13.4.1 Derived operations

The composition-uncurry combinator.

**notation** *fcomp* (**infixl**  $o > 60$ )

**definition**

$scomp :: ('a \Rightarrow 'b \times 'c) \Rightarrow ('b \Rightarrow 'c \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'd$  (**infixl**  $o \rightarrow 60$ )

where

$f \circ \rightarrow g = (\lambda x. \text{split } g \ (f \ x))$

**lemma** *scomp-apply*:  $(f \circ \rightarrow g) \ x = \text{split } g \ (f \ x)$   
**by** (*simp add: scomp-def*)

**lemma** *Pair-scomp*:  $\text{Pair } x \circ \rightarrow f = f \ x$   
**by** (*simp add: expand-fun-eq scomp-apply*)

**lemma** *scomp-Pair*:  $x \circ \rightarrow \text{Pair} = x$   
**by** (*simp add: expand-fun-eq scomp-apply*)

**lemma** *scomp-scomp*:  $(f \circ \rightarrow g) \circ \rightarrow h = f \circ \rightarrow (\lambda x. g \ x \circ \rightarrow h)$   
**by** (*simp add: expand-fun-eq split-twice scomp-def*)

**lemma** *scomp-fcomp*:  $(f \circ \rightarrow g) \circ > h = f \circ \rightarrow (\lambda x. g \ x \circ > h)$   
**by** (*simp add: expand-fun-eq scomp-apply fcomp-def split-def*)

**lemma** *fcomp-scomp*:  $(f \circ > g) \circ \rightarrow h = f \circ > (g \circ \rightarrow h)$   
**by** (*simp add: expand-fun-eq scomp-apply fcomp-apply*)

**no-notation** *fcomp* (**infixl**  $\circ >$  60)

**no-notation** *scomp* (**infixl**  $\circ \rightarrow$  60)

*prod-fun* — action of the product functor upon functions.

**definition** *prod-fun* ::  $('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'd) \Rightarrow 'a \times 'b \Rightarrow 'c \times 'd$  **where**  
 $[\text{code del}]: \text{prod-fun } f \ g = (\lambda(x, y). (f \ x, g \ y))$

**lemma** *prod-fun [simp, code]*:  $\text{prod-fun } f \ g \ (a, b) = (f \ a, g \ b)$   
**by** (*simp add: prod-fun-def*)

**lemma** *prod-fun-compose*:  $\text{prod-fun } (f1 \circ f2) \ (g1 \circ g2) = (\text{prod-fun } f1 \ g1 \circ \text{prod-fun } f2 \ g2)$   
**by** (*rule ext*) *auto*

**lemma** *prod-fun-ident [simp]*:  $\text{prod-fun } (\%x. x) \ (\%y. y) = (\%z. z)$   
**by** (*rule ext*) *auto*

**lemma** *prod-fun-imageI [intro]*:  $(a, b) : r \implies (f \ a, g \ b) : \text{prod-fun } f \ g \ 'r$   
**apply** (*rule image-eqI*)  
**apply** (*rule prod-fun [symmetric], assumption*)  
**done**

**lemma** *prod-fun-imageE [elim!]*:  
**assumes** *major*:  $c : (\text{prod-fun } f \ g) \ 'r$   
**and cases**:  $!!x \ y. [\ c=(f(x),g(y)); \ (x,y):r \ ] \implies P$   
**shows**  $P$   
**apply** (*rule major [THEN imageE]*)  
**apply** (*rule-tac p = x in PairE*)

```

apply (rule cases)
  apply (blast intro: prod-fun)
apply blast
done

```

**definition**

```

 $apfst :: ('a \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c \times 'b$ 
where
  [code del]:  $apfst\ f = prod\text{-}fun\ f\ id$ 

```

**definition**

```

 $apsnd :: ('b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'a \times 'c$ 
where
  [code del]:  $apsnd\ f = prod\text{-}fun\ id\ f$ 

```

**lemma** *apfst-conv* [*simp*, *code*]:

```

 $apfst\ f\ (x, y) = (f\ x, y)$ 
by (simp add: apfst-def)

```

**lemma** *upd-snd-conv* [*simp*, *code*]:

```

 $apsnd\ f\ (x, y) = (x, f\ y)$ 
by (simp add: apsnd-def)

```

Disjoint union of a family of sets – Sigma.

**definition** *Sigma* :: [*'a set*, *'a => 'b set*] => (*'a* × *'b*) *set* **where**  
*Sigma-def*:  $Sigma\ A\ B == \bigcup x:A. \bigcup y:B\ x. \{Pair\ x\ y\}$

**abbreviation**

```

 $Times :: ['a\ set, 'b\ set] \Rightarrow ('a * 'b)\ set$ 
  (infixr <*> 80) where
   $A\ <*>\ B == Sigma\ A\ (\%-. B)$ 

```

**notation** (*xsymbols*)

```

 $Times$  (infixr × 80)

```

**notation** (*HTML output*)

```

 $Times$  (infixr × 80)

```

**syntax**

```

@Sigma :: [pttrn, 'a set, 'b set] => ('a * 'b) set ((3SIGMA :-./ -) [0, 0, 10] 10)

```

**translations**

```

SIGMA  $x:A. B == Product\text{-}Type.Sigma\ A\ (\%x. B)$ 

```

**lemma** *SigmaI* [*intro!*]: [*a:A; b:B(a)*] ==> (*a,b*) : *Sigma A B*

```

by (unfold Sigma-def) blast

```

**lemma** *SigmaE* [*elim!*]:

```

[c: Sigma A B;

```

$$\begin{aligned} & !!x\ y. [\![\ x:A;\ y:B(x);\ c=(x,y)\ ]\!] ==> P \\ & [\!] ==> P \end{aligned}$$

— The general elimination rule.

**by** (*unfold Sigma-def*) *blast*

Elimination of  $(a, b) \in A \times B$  – introduces no eigenvariables.

**lemma** *SigmaD1*:  $(a, b) : \text{Sigma } A\ B ==> a : A$   
**by** *blast*

**lemma** *SigmaD2*:  $(a, b) : \text{Sigma } A\ B ==> b : B\ a$   
**by** *blast*

**lemma** *SigmaE2*:  

$$\begin{aligned} & [\! [\ (a, b) : \text{Sigma } A\ B; \\ & \quad [\! [\ a:A;\ b:B(a)\ ]\!] ==> P \\ & \quad ]\!] ==> P \end{aligned}$$
  
**by** *blast*

**lemma** *Sigma-cong*:  

$$\begin{aligned} & \llbracket A = B; !!x. x \in B \implies C\ x = D\ x \rrbracket \\ & \implies (\text{SIGMA } x: A. C\ x) = (\text{SIGMA } x: B. D\ x) \end{aligned}$$
  
**by** *auto*

**lemma** *Sigma-mono*:  $[\![\ A <= C; !!x. x:A ==> B\ x <= D\ x\ ]\!] ==> \text{Sigma } A\ B$   
 $<= \text{Sigma } C\ D$   
**by** *blast*

**lemma** *Sigma-empty1* [*simp*]:  $\text{Sigma } \{\} B = \{\}$   
**by** *blast*

**lemma** *Sigma-empty2* [*simp*]:  $A <*> \{\} = \{\}$   
**by** *blast*

**lemma** *UNIV-Times-UNIV* [*simp*]:  $\text{UNIV } <*> \text{UNIV} = \text{UNIV}$   
**by** *auto*

**lemma** *Compl-Times-UNIV1* [*simp*]:  $\neg (\text{UNIV } <*> A) = \text{UNIV } <*> (\neg A)$   
**by** *auto*

**lemma** *Compl-Times-UNIV2* [*simp*]:  $\neg (A <*> \text{UNIV}) = (\neg A) <*> \text{UNIV}$   
**by** *auto*

**lemma** *mem-Sigma-iff* [*iff*]:  $((a,b): \text{Sigma } A\ B) = (a:A \ \& \ b:B(a))$   
**by** *blast*

**lemma** *Times-subset-cancel2*:  $x:C ==> (A <*> C <= B <*> C) = (A <= B)$   
**by** *blast*

**lemma** *Times-eq-cancel2*:  $x:C ==> (A <*> C = B <*> C) = (A = B)$

**by** (*blast elim: equalityE*)

**lemma** *SetCompr-Sigma-eq*:

$\text{Collect } (\text{split } (\%x\ y.\ P\ x \ \&\ Q\ x\ y)) = (\text{SIGMA } x:\text{Collect } P.\ \text{Collect } (Q\ x))$

**by** *blast*

**lemma** *Collect-split [simp]*:  $\{(a,b).\ P\ a \ \&\ Q\ b\} = \text{Collect } P\ <*>\ \text{Collect } Q$

**by** *blast*

**lemma** *UN-Times-distrib*:

$(\text{UN } (a,b):(A\ <*>\ B).\ E\ a\ <*>\ F\ b) = (\text{UNION } A\ E)\ <*>\ (\text{UNION } B\ F)$

— Suggested by Pierre Chartier

**by** *blast*

**lemma** *split-paired-Ball-Sigma [simp,noatp]*:

$(\text{ALL } z:\text{Sigma } A\ B.\ P\ z) = (\text{ALL } x:A.\ \text{ALL } y:B\ x.\ P(x,y))$

**by** *blast*

**lemma** *split-paired-Bex-Sigma [simp,noatp]*:

$(\text{EX } z:\text{Sigma } A\ B.\ P\ z) = (\text{EX } x:A.\ \text{EX } y:B\ x.\ P(x,y))$

**by** *blast*

**lemma** *Sigma-Un-distrib1*:  $(\text{SIGMA } i:I\ \text{Un } J.\ C(i)) = (\text{SIGMA } i:I.\ C(i))\ \text{Un } (\text{SIGMA } j:J.\ C(j))$

**by** *blast*

**lemma** *Sigma-Un-distrib2*:  $(\text{SIGMA } i:I.\ A(i)\ \text{Un } B(i)) = (\text{SIGMA } i:I.\ A(i))\ \text{Un } (\text{SIGMA } i:I.\ B(i))$

**by** *blast*

**lemma** *Sigma-Int-distrib1*:  $(\text{SIGMA } i:I\ \text{Int } J.\ C(i)) = (\text{SIGMA } i:I.\ C(i))\ \text{Int } (\text{SIGMA } j:J.\ C(j))$

**by** *blast*

**lemma** *Sigma-Int-distrib2*:  $(\text{SIGMA } i:I.\ A(i)\ \text{Int } B(i)) = (\text{SIGMA } i:I.\ A(i))\ \text{Int } (\text{SIGMA } i:I.\ B(i))$

**by** *blast*

**lemma** *Sigma-Diff-distrib1*:  $(\text{SIGMA } i:I\ -\ J.\ C(i)) = (\text{SIGMA } i:I.\ C(i))\ -\ (\text{SIGMA } j:J.\ C(j))$

**by** *blast*

**lemma** *Sigma-Diff-distrib2*:  $(\text{SIGMA } i:I.\ A(i)\ -\ B(i)) = (\text{SIGMA } i:I.\ A(i))\ -\ (\text{SIGMA } i:I.\ B(i))$

**by** *blast*

**lemma** *Sigma-Union*:  $\text{Sigma } (\text{Union } X)\ B = (\text{UN } A:X.\ \text{Sigma } A\ B)$

**by** *blast*

Non-dependent versions are needed to avoid the need for higher-order match-



ing, especially when the rules are re-oriented.

**lemma** *Times-Un-distrib1*:  $(A \text{ Un } B) <*> C = (A <*> C) \text{ Un } (B <*> C)$   
**by** *blast*

**lemma** *Times-Int-distrib1*:  $(A \text{ Int } B) <*> C = (A <*> C) \text{ Int } (B <*> C)$   
**by** *blast*

**lemma** *Times-Diff-distrib1*:  $(A - B) <*> C = (A <*> C) - (B <*> C)$   
**by** *blast*

**lemma** *insert-times-insert[simp]*:  
 $\text{insert } a \ A \times \text{insert } b \ B =$   
 $\text{insert } (a,b) \ (A \times \text{insert } b \ B \cup \text{insert } a \ A \times B)$   
**by** *blast*

### 13.4.2 Code generator setup

**instance**  $*$  ::  $(eq, eq) \ eq \ ..$

**lemma** *[code]*:  
 $eq\text{-class}.eq \ (x1::'a::eq, y1::'b::eq) \ (x2, y2) \longleftrightarrow x1 = x2 \wedge y1 = y2$  **by** (*simp*  
*add: eq*)

**lemma** *split-case-cert*:  
**assumes**  $CASE \equiv \text{split } f$   
**shows**  $CASE \ (a, b) \equiv f \ a \ b$   
**using** *assms* **by** *simp*

**setup**  $\langle\langle$   
 $\text{Code.add-case } @\{thm \ \text{split-case-cert}\}$   
 $\rangle\rangle$

**code-type**  $*$   
 $(SML \ \text{infix } 2 \ *)$   
 $(OCaml \ \text{infix } 2 \ *)$   
 $(Haskell \ !((-), / \ (-)))$

**code-instance**  $*$  ::  $eq$   
 $(Haskell \ -)$

**code-const**  $eq\text{-class}.eq :: 'a::eq \times 'b::eq \Rightarrow 'a \times 'b \Rightarrow bool$   
 $(Haskell \ \text{infixl } 4 \ ==)$

**code-const** *Pair*  
 $(SML \ !((-), / \ (-)))$   
 $(OCaml \ !((-), / \ (-)))$   
 $(Haskell \ !((-), / \ (-)))$

**code-const** *fst* and *snd*

(*Haskell* *fst* **and** *snd*)

### types-code

```
*      ((- */ -))
attach (term-of) ⟨⟨
fun term-of-id-42 aF aT bF bT (x, y) = HOLogic.pair-const aT bT $ aF x $ bF
y;
⟩⟩
attach (test) ⟨⟨
fun gen-id-42 aG aT bG bT i =
  let
    val (x, t) = aG i;
    val (y, u) = bG i
    in ((x, y), fn () => HOLogic.pair-const aT bT $ t () $ u ()) end;
⟩⟩
```

### consts-code

*Pair* ((-, / -))

### setup ⟨⟨

*let*

```
fun strip-abs-split 0 t = ([], t)
| strip-abs-split i (Abs (s, T, t)) =
  let
    val s' = Codegen.new-name t s;
    val v = Free (s', T)
    in apfst (cons v) (strip-abs-split (i-1) (subst-bound (v, t))) end
| strip-abs-split i (u as Const (split, -) $ t) = (case strip-abs-split (i+1) t of
  (v :: v' :: vs, u) => (HOLogic.mk-prod (v, v') :: vs, u)
  | - => ([], u))
| strip-abs-split i t =
  strip-abs-split i (Abs (x, hd (binder-types (fastype-of t)), t $ Bound 0));
```

```
fun let-codegen thy defs dep thynome brack t gr = (case strip-comb t of
```

```
(t1 as Const (Let, -), t2 :: t3 :: ts) =>
```

```
let
```

```
fun dest-let (l as Const (Let, -) $ t $ u) =
```

```
(case strip-abs-split 1 u of
```

```
([p], u') => apfst (cons (p, t)) (dest-let u')
```

```
| - => ([], l))
```

```
| dest-let t = ([], t);
```

```
fun mk-code (l, r) gr =
```

```
let
```

```
val (pl, gr1) = Codegen.invoke-codegen thy defs dep thynome false l gr;
```

```
val (pr, gr2) = Codegen.invoke-codegen thy defs dep thynome false r gr1;
```

```
in ((pl, pr), gr2) end
```

```
in case dest-let (t1 $ t2 $ t3) of
```

```

    ([], -) => NONE
  | (ps, u) =>
    let
      val (qs, gr1) = fold-map mk-code ps gr;
      val (pu, gr2) = Codegen.invoke-codegen thy defs dep thyname false u gr1;
      val (pargs, gr3) = fold-map
        (Codegen.invoke-codegen thy defs dep thyname true) ts gr2
    in
      SOME (Codegen.mk-app brack
        (Pretty.blk (0, [Codegen.str let , Pretty.blk (0, List.concat
          (separate [Codegen.str ;, Pretty.brk 1] (map (fn (pl, pr) =>
            [Pretty.block [Codegen.str val , pl, Codegen.str =,
              Pretty.brk 1, pr]]) qs))),
          Pretty.brk 1, Codegen.str in , pu,
          Pretty.brk 1, Codegen.str end])) pargs, gr3)
    end
  end
  | - => NONE);

fun split-codegen thy defs dep thyname brack t gr = (case strip-comb t of
  (t1 as Const (split, -), t2 :: ts) =>
    let
      val ([p], u) = strip-abs-split 1 (t1 $ t2);
      val (q, gr1) = Codegen.invoke-codegen thy defs dep thyname false p gr;
      val (pu, gr2) = Codegen.invoke-codegen thy defs dep thyname false u gr1;
      val (pargs, gr3) = fold-map
        (Codegen.invoke-codegen thy defs dep thyname true) ts gr2
    in
      SOME (Codegen.mk-app brack
        (Pretty.block [Codegen.str (fn , q, Codegen.str =>,
          Pretty.brk 1, pu, Codegen.str )]) pargs, gr2)
    end
  | - => NONE);

in

  Codegen.add-codegen let-codegen let-codegen
  #> Codegen.add-codegen split-codegen split-codegen

end
>>

```

### 13.5 Legacy bindings

```

ML <<
  val Collect-split = thm Collect-split;
  val Compl-Times-UNIV1 = thm Compl-Times-UNIV1;
  val Compl-Times-UNIV2 = thm Compl-Times-UNIV2;
  val PairE = thm PairE;

```

```

val Pair-Rep-inject = thm Pair-Rep-inject;
val Pair-def = thm Pair-def;
val Pair-eq = @{thm prod.inject};
val Pair-fst-snd-eq = thm Pair-fst-snd-eq;
val ProdI = thm ProdI;
val SetCompr-Sigma-eq = thm SetCompr-Sigma-eq;
val SigmaD1 = thm SigmaD1;
val SigmaD2 = thm SigmaD2;
val SigmaE = thm SigmaE;
val SigmaE2 = thm SigmaE2;
val SigmaI = thm SigmaI;
val Sigma-Diff-distrib1 = thm Sigma-Diff-distrib1;
val Sigma-Diff-distrib2 = thm Sigma-Diff-distrib2;
val Sigma-Int-distrib1 = thm Sigma-Int-distrib1;
val Sigma-Int-distrib2 = thm Sigma-Int-distrib2;
val Sigma-Un-distrib1 = thm Sigma-Un-distrib1;
val Sigma-Un-distrib2 = thm Sigma-Un-distrib2;
val Sigma-Union = thm Sigma-Union;
val Sigma-def = thm Sigma-def;
val Sigma-empty1 = thm Sigma-empty1;
val Sigma-empty2 = thm Sigma-empty2;
val Sigma-mono = thm Sigma-mono;
val The-split = thm The-split;
val The-split-eq = thm The-split-eq;
val The-split-eq = thm The-split-eq;
val Times-Diff-distrib1 = thm Times-Diff-distrib1;
val Times-Int-distrib1 = thm Times-Int-distrib1;
val Times-Un-distrib1 = thm Times-Un-distrib1;
val Times-eq-cancel2 = thm Times-eq-cancel2;
val Times-subset-cancel2 = thm Times-subset-cancel2;
val UNIV-Times-UNIV = thm UNIV-Times-UNIV;
val UN-Times-distrib = thm UN-Times-distrib;
val Unity-def = thm Unity-def;
val cond-split-eta = thm cond-split-eta;
val fst-conv = thm fst-conv;
val fst-def = thm fst-def;
val fst-eqD = thm fst-eqD;
val inj-on-Abs-Prod = thm inj-on-Abs-Prod;
val mem-Sigma-iff = thm mem-Sigma-iff;
val mem-splitE = thm mem-splitE;
val mem-splitI = thm mem-splitI;
val mem-splitI2 = thm mem-splitI2;
val prod-eqI = thm prod-eqI;
val prod-fun = thm prod-fun;
val prod-fun-compose = thm prod-fun-compose;
val prod-fun-def = thm prod-fun-def;
val prod-fun-ident = thm prod-fun-ident;
val prod-fun-imageE = thm prod-fun-imageE;
val prod-fun-imageI = thm prod-fun-imageI;

```

```

val prod-induct = thm prod.induct;
val snd-conv = thm snd-conv;
val snd-def = thm snd-def;
val snd-eqD = thm snd-eqD;
val split = thm split;
val splitD = thm splitD;
val splitD' = thm splitD';
val splitE = thm splitE;
val splitE' = thm splitE';
val splitE2 = thm splitE2;
val splitI = thm splitI;
val splitI2 = thm splitI2;
val splitI2' = thm splitI2';
val split-beta = thm split-beta;
val split-conv = thm split-conv;
val split-def = thm split-def;
val split-eta = thm split-eta;
val split-eta-SetCompr = thm split-eta-SetCompr;
val split-eta-SetCompr2 = thm split-eta-SetCompr2;
val split-paired-All = thm split-paired-All;
val split-paired-Ball-Sigma = thm split-paired-Ball-Sigma;
val split-paired-Bex-Sigma = thm split-paired-Bex-Sigma;
val split-paired-Ex = thm split-paired-Ex;
val split-paired-The = thm split-paired-The;
val split-paired-all = thm split-paired-all;
val split-part = thm split-part;
val split-split = thm split-split;
val split-split-asm = thm split-split-asm;
val split-tupled-all = thm split-tupled-all;
val split-weak-cong = thm split-weak-cong;
val surj-pair = thm surj-pair;
val surjective-pairing = thm surjective-pairing;
val unit-abs-eta-conv = thm unit-abs-eta-conv;
val unit-all-eq1 = thm unit-all-eq1;
val unit-all-eq2 = thm unit-all-eq2;
val unit-eq = thm unit-eq;
>>

```

### 13.6 Further inductive packages

```

use Tools/inductive-realizer.ML
setup InductiveRealizer.setup

```

```

use Tools/inductive-set-package.ML
setup InductiveSetPackage.setup

```

```

use Tools/datatype-realizer.ML
setup DatatypeRealizer.setup

```

end

## 14 Datatype: Analogues of the Cartesian Product and Disjoint Sum for Datatypes

**theory** *Datatype*

**imports** *Nat Product-Type*

**begin**

**typedef** (*Node*)

$(\text{'a}, \text{'b}) \text{ node} = \{p. \text{EX } f \ x \ k. p = (f::\text{nat} \Rightarrow \text{'b} + \text{nat}, x::\text{'a} + \text{nat}) \ \& \ f \ k = \text{Inr } 0\}$   
 — it is a subtype of  $(\text{nat} \Rightarrow \text{'b} + \text{nat}) * (\text{'a} + \text{nat})$

**by** *auto*

Datatypes will be represented by sets of type *node*

**types** *'a item* =  $(\text{'a}, \text{unit}) \text{ node set}$

$(\text{'a}, \text{'b}) \text{ dtree} = (\text{'a}, \text{'b}) \text{ node set}$

**consts**

*Push* ::  $[(\text{'b} + \text{nat}), \text{nat} \Rightarrow (\text{'b} + \text{nat})] \Rightarrow (\text{nat} \Rightarrow (\text{'b} + \text{nat}))$

*Push-Node* ::  $[(\text{'b} + \text{nat}), (\text{'a}, \text{'b}) \text{ node}] \Rightarrow (\text{'a}, \text{'b}) \text{ node}$

*ndepth* ::  $(\text{'a}, \text{'b}) \text{ node} \Rightarrow \text{nat}$

*Atom* ::  $(\text{'a} + \text{nat}) \Rightarrow (\text{'a}, \text{'b}) \text{ dtree}$

*Leaf* ::  $\text{'a} \Rightarrow (\text{'a}, \text{'b}) \text{ dtree}$

*Numb* ::  $\text{nat} \Rightarrow (\text{'a}, \text{'b}) \text{ dtree}$

*Scons* ::  $[(\text{'a}, \text{'b}) \text{ dtree}, (\text{'a}, \text{'b}) \text{ dtree}] \Rightarrow (\text{'a}, \text{'b}) \text{ dtree}$

*In0* ::  $(\text{'a}, \text{'b}) \text{ dtree} \Rightarrow (\text{'a}, \text{'b}) \text{ dtree}$

*In1* ::  $(\text{'a}, \text{'b}) \text{ dtree} \Rightarrow (\text{'a}, \text{'b}) \text{ dtree}$

*Lim* ::  $(\text{'b} \Rightarrow (\text{'a}, \text{'b}) \text{ dtree}) \Rightarrow (\text{'a}, \text{'b}) \text{ dtree}$

*ntrunc* ::  $[\text{nat}, (\text{'a}, \text{'b}) \text{ dtree}] \Rightarrow (\text{'a}, \text{'b}) \text{ dtree}$

*uprod* ::  $[(\text{'a}, \text{'b}) \text{ dtree set}, (\text{'a}, \text{'b}) \text{ dtree set}] \Rightarrow (\text{'a}, \text{'b}) \text{ dtree set}$

*usum* ::  $[(\text{'a}, \text{'b}) \text{ dtree set}, (\text{'a}, \text{'b}) \text{ dtree set}] \Rightarrow (\text{'a}, \text{'b}) \text{ dtree set}$

*Split* ::  $[[(\text{'a}, \text{'b}) \text{ dtree}, (\text{'a}, \text{'b}) \text{ dtree}] \Rightarrow \text{'c}, (\text{'a}, \text{'b}) \text{ dtree}] \Rightarrow \text{'c}$

*Case* ::  $[[(\text{'a}, \text{'b}) \text{ dtree}] \Rightarrow \text{'c}, [(\text{'a}, \text{'b}) \text{ dtree}] \Rightarrow \text{'c}, (\text{'a}, \text{'b}) \text{ dtree}] \Rightarrow \text{'c}$

*dprod* ::  $[((\text{'a}, \text{'b}) \text{ dtree} * (\text{'a}, \text{'b}) \text{ dtree})\text{set}, ((\text{'a}, \text{'b}) \text{ dtree} * (\text{'a}, \text{'b}) \text{ dtree})\text{set}] \Rightarrow ((\text{'a}, \text{'b}) \text{ dtree} * (\text{'a}, \text{'b}) \text{ dtree})\text{set}$

*dsum* ::  $[((\text{'a}, \text{'b}) \text{ dtree} * (\text{'a}, \text{'b}) \text{ dtree})\text{set}, ((\text{'a}, \text{'b}) \text{ dtree} * (\text{'a}, \text{'b}) \text{ dtree})\text{set}] \Rightarrow ((\text{'a}, \text{'b}) \text{ dtree} * (\text{'a}, \text{'b}) \text{ dtree})\text{set}$

**defs**

*Push-Node-def:*  $Push\text{-}Node == (\%n\ x.\ Abs\text{-}Node\ (apfst\ (Push\ n)\ (Rep\text{-}Node\ x)))$

*Push-def:*  $Push == (\%b\ h.\ nat\text{-}case\ b\ h)$

*Atom-def:*  $Atom == (\%x.\ \{Abs\text{-}Node((\%k.\ Inr\ 0,\ x))\})$   
*Scons-def:*  $Scons\ M\ N == (Push\text{-}Node\ (Inr\ 1)\ 'M)\ Un\ (Push\text{-}Node\ (Inr\ (Suc\ 1))\ 'N)$

*Leaf-def:*  $Leaf == Atom\ o\ Inl$   
*Numb-def:*  $Numb == Atom\ o\ Inr$

*In0-def:*  $In0(M) == Scons\ (Numb\ 0)\ M$   
*In1-def:*  $In1(M) == Scons\ (Numb\ 1)\ M$

*Lim-def:*  $Lim\ f == Union\ \{z.\ ?\ x.\ z = Push\text{-}Node\ (Inl\ x)\ ' (f\ x)\}$

*ndepth-def:*  $ndepth(n) == (\%(f,x).\ LEAST\ k.\ f\ k = Inr\ 0)\ (Rep\text{-}Node\ n)$   
*ntrunc-def:*  $ntrunc\ k\ N == \{n.\ n:N\ \&\ ndepth(n) < k\}$

*uprod-def:*  $uprod\ A\ B == UN\ x:A.\ UN\ y:B.\ \{Scons\ x\ y\}$   
*usum-def:*  $usum\ A\ B == In0'A\ Un\ In1'B$

*Split-def:*  $Split\ c\ M == THE\ u.\ EX\ x\ y.\ M = Scons\ x\ y\ \&\ u = c\ x\ y$

*Case-def:*  $Case\ c\ d\ M == THE\ u.\ (EX\ x.\ M = In0(x)\ \&\ u = c(x))$   
 $\quad \mid (EX\ y.\ M = In1(y)\ \&\ u = d(y))$

*dprod-def:*  $dprod\ r\ s == UN\ (x,x'):r.\ UN\ (y,y'):s.\ \{(Scons\ x\ y,\ Scons\ x'\ y')\}$

*dsum-def:*  $dsum\ r\ s == (UN\ (x,x'):r.\ \{(In0(x),In0(x'))\})\ Un$   
 $\quad (UN\ (y,y'):s.\ \{(In1(y),In1(y'))\})$

**lemma** *apfst-convE*:

```

  [| q = apfst f p; !!x y. [| p = (x,y); q = (f(x),y) |] ==> R
  |] ==> R
by (force simp add: apfst-def)

```

**lemma** *Push-inject1*:  $\text{Push } i \ f = \text{Push } j \ g \implies i=j$

```

apply (simp add: Push-def expand-fun-eq)
apply (drule-tac x=0 in spec, simp)
done

```

**lemma** *Push-inject2*:  $\text{Push } i \ f = \text{Push } j \ g \implies f=g$

```

apply (auto simp add: Push-def expand-fun-eq)
apply (drule-tac x=Suc x in spec, simp)
done

```

**lemma** *Push-inject*:

```

  [| Push i f = Push j g; [| i=j; f=g |] ==> P |] ==> P
by (blast dest: Push-inject1 Push-inject2)

```

**lemma** *Push-neq-K0*:  $\text{Push } (\text{Inr } (\text{Suc } k)) \ f = (\%z. \text{Inr } 0) \implies P$

```

by (auto simp add: Push-def expand-fun-eq split: nat.split-asm)

```

**lemmas** *Abs-Node-inj* = *Abs-Node-inject* [THEN [2] rev-iffD1, standard]

**lemma** *Node-K0-I*:  $(\%k. \text{Inr } 0, a) : \text{Node}$

```

by (simp add: Node-def)

```

**lemma** *Node-Push-I*:  $p : \text{Node} \implies \text{apfst } (\text{Push } i) \ p : \text{Node}$

```

apply (simp add: Node-def Push-def)
apply (fast intro!: apfst-conv nat-case-Suc [THEN trans])
done

```

## 14.1 Freeness: Distinctness of Constructors

**lemma** *Scons-not-Atom* [iff]:  $\text{Scons } M \ N \neq \text{Atom}(a)$

```

apply (simp add: Atom-def Scons-def Push-Node-def One-nat-def)

```

```

apply (blast intro: Node-K0-I Rep-Node [THEN Node-Push-I]

```

```

  dest!: Abs-Node-inj

```

```

  elim!: apfst-convE sym [THEN Push-neq-K0])

```

```

done

```

**lemmas** *Atom-not-Scons* [iff] = *Scons-not-Atom* [THEN not-sym, standard]



```

lemma inj-Atom: inj(Atom)
apply (simp add: Atom-def)
apply (blast intro!: inj-onI Node-K0-I dest!: Abs-Node-inj)
done
lemmas Atom-inject = inj-Atom [THEN injD, standard]

```

```

lemma Atom-Atom-eq [iff]: (Atom(a)=Atom(b)) = (a=b)
by (blast dest!: Atom-inject)

```

```

lemma inj-Leaf: inj(Leaf)
apply (simp add: Leaf-def o-def)
apply (rule inj-onI)
apply (erule Atom-inject [THEN Inl-inject])
done

```

```

lemmas Leaf-inject [dest!] = inj-Leaf [THEN injD, standard]

```

```

lemma inj-Numb: inj(Numb)
apply (simp add: Numb-def o-def)
apply (rule inj-onI)
apply (erule Atom-inject [THEN Inr-inject])
done

```

```

lemmas Numb-inject [dest!] = inj-Numb [THEN injD, standard]

```

```

lemma Push-Node-inject:
  [| Push-Node i m =Push-Node j n; [| i=j; m=n |] ==> P
  |] ==> P
apply (simp add: Push-Node-def)
apply (erule Abs-Node-inj [THEN apfst-convE])
apply (rule Rep-Node [THEN Node-Push-I])+
apply (erule sym [THEN apfst-convE])
apply (blast intro: Rep-Node-inject [THEN iffD1] trans sym elim!: Push-inject)
done

```

```

lemma Scons-inject-lemma1: Scons M N <= Scons M' N' ==> M <= M'
apply (simp add: Scons-def One-nat-def)
apply (blast dest!: Push-Node-inject)
done

```

**lemma** *Scons-inject-lemma2*:  $Scons\ M\ N \leq Scons\ M'\ N' \implies N \leq N'$   
**apply** (*simp add: Scons-def One-nat-def*)  
**apply** (*blast dest!: Push-Node-inject*)  
**done**

**lemma** *Scons-inject1*:  $Scons\ M\ N = Scons\ M'\ N' \implies M = M'$   
**apply** (*erule equalityE*)  
**apply** (*iprover intro: equalityI Scons-inject-lemma1*)  
**done**

**lemma** *Scons-inject2*:  $Scons\ M\ N = Scons\ M'\ N' \implies N = N'$   
**apply** (*erule equalityE*)  
**apply** (*iprover intro: equalityI Scons-inject-lemma2*)  
**done**

**lemma** *Scons-inject*:  
 $[[\ Scons\ M\ N = Scons\ M'\ N';\ [\ M = M';\ N = N'\ ]\ ] \implies P\ ] \implies P$   
**by** (*iprover dest: Scons-inject1 Scons-inject2*)

**lemma** *Scons-Scons-eq [iff]*:  $(Scons\ M\ N = Scons\ M'\ N') = (M = M' \ \&\ N = N')$   
**by** (*blast elim!: Scons-inject*)

**lemma** *Scons-not-Leaf [iff]*:  $Scons\ M\ N \neq Leaf(a)$   
**by** (*simp add: Leaf-def o-def Scons-not-Atom*)

**lemmas** *Leaf-not-Scons [iff] = Scons-not-Leaf [THEN not-sym, standard]*

**lemma** *Scons-not-Numb [iff]*:  $Scons\ M\ N \neq Numb(k)$   
**by** (*simp add: Numb-def o-def Scons-not-Atom*)

**lemmas** *Numb-not-Scons [iff] = Scons-not-Numb [THEN not-sym, standard]*

**lemma** *Leaf-not-Numb [iff]*:  $Leaf(a) \neq Numb(k)$   
**by** (*simp add: Leaf-def Numb-def*)

**lemmas** *Numb-not-Leaf [iff] = Leaf-not-Numb [THEN not-sym, standard]*

**lemma** *ndepth-K0*:  $\text{ndepth } (\text{Abs-Node}(\%k. \text{Inr } 0, x)) = 0$   
**by** (*simp add: ndepth-def Node-K0-I [THEN Abs-Node-inverse] Least-equality*)

**lemma** *ndepth-Push-Node-aux*:  
 $\text{nat-case } (\text{Inr } (\text{Suc } i)) f k = \text{Inr } 0 \dashrightarrow \text{Suc}(\text{LEAST } x. f x = \text{Inr } 0) \leq k$   
**apply** (*induct-tac k, auto*)  
**apply** (*erule Least-le*)  
**done**

**lemma** *ndepth-Push-Node*:  
 $\text{ndepth } (\text{Push-Node } (\text{Inr } (\text{Suc } i)) n) = \text{Suc}(\text{ndepth}(n))$   
**apply** (*insert Rep-Node [of n, unfolded Node-def]*)  
**apply** (*auto simp add: ndepth-def Push-Node-def*  
 $\text{Rep-Node [THEN Node-Push-I, THEN Abs-Node-inverse]}$ )  
**apply** (*rule Least-equality*)  
**apply** (*auto simp add: Push-def ndepth-Push-Node-aux*)  
**apply** (*erule LeastI*)  
**done**

**lemma** *ntrunc-0* [*simp*]:  $\text{ntrunc } 0 M = \{\}$   
**by** (*simp add: ntrunc-def*)

**lemma** *ntrunc-Atom* [*simp*]:  $\text{ntrunc } (\text{Suc } k) (\text{Atom } a) = \text{Atom}(a)$   
**by** (*auto simp add: Atom-def ntrunc-def ndepth-K0*)

**lemma** *ntrunc-Leaf* [*simp*]:  $\text{ntrunc } (\text{Suc } k) (\text{Leaf } a) = \text{Leaf}(a)$   
**by** (*simp add: Leaf-def o-def ntrunc-Atom*)

**lemma** *ntrunc-Numb* [*simp*]:  $\text{ntrunc } (\text{Suc } k) (\text{Numb } i) = \text{Numb}(i)$   
**by** (*simp add: Numb-def o-def ntrunc-Atom*)

**lemma** *ntrunc-Scons* [*simp*]:  
 $\text{ntrunc } (\text{Suc } k) (\text{Scons } M N) = \text{Scons } (\text{ntrunc } k M) (\text{ntrunc } k N)$   
**by** (*auto simp add: Scons-def ntrunc-def One-nat-def ndepth-Push-Node*)

**lemma** *ntrunc-one-In0* [*simp*]:  $\text{ntrunc } (\text{Suc } 0) (\text{In0 } M) = \{\}$   
**apply** (*simp add: In0-def*)  
**apply** (*simp add: Scons-def*)  
**done**

**lemma** *ntrunc-In0* [*simp*]:  $\text{ntrunc } (\text{Suc}(\text{Suc } k)) (\text{In0 } M) = \text{In0 } (\text{ntrunc } (\text{Suc } k))$

$M$ )  
**by** (*simp add: In0-def*)

**lemma** *ntrunc-one-In1* [*simp*]:  $\text{ntrunc } (\text{Suc } 0) (\text{In1 } M) = \{\}$   
**apply** (*simp add: In1-def*)  
**apply** (*simp add: Scons-def*)  
**done**

**lemma** *ntrunc-In1* [*simp*]:  $\text{ntrunc } (\text{Suc}(\text{Suc } k)) (\text{In1 } M) = \text{In1 } (\text{ntrunc } (\text{Suc } k) M)$   
**by** (*simp add: In1-def*)

## 14.2 Set Constructions

**lemma** *uprodI* [*intro!*]:  $[\![ M:A; N:B ]\!] \implies \text{Scons } M N : \text{uprod } A B$   
**by** (*simp add: uprod-def*)

**lemma** *uprodE* [*elim!*]:  
 $[\![ c : \text{uprod } A B;$   
 $\quad !!x y. [\![ x:A; y:B; c = \text{Scons } x y ]\!] \implies P$   
 $\quad ]\!] \implies P$   
**by** (*auto simp add: uprod-def*)

**lemma** *uprodE2*:  $[\![ \text{Scons } M N : \text{uprod } A B; [\![ M:A; N:B ]\!] \implies P ]\!] \implies P$   
**by** (*auto simp add: uprod-def*)

**lemma** *usum-In0I* [*intro*]:  $M:A \implies \text{In0}(M) : \text{usum } A B$   
**by** (*simp add: usum-def*)

**lemma** *usum-In1I* [*intro*]:  $N:B \implies \text{In1}(N) : \text{usum } A B$   
**by** (*simp add: usum-def*)

**lemma** *usumE* [*elim!*]:  
 $[\![ u : \text{usum } A B;$   
 $\quad !!x. [\![ x:A; u=\text{In0}(x) ]\!] \implies P;$   
 $\quad !!y. [\![ y:B; u=\text{In1}(y) ]\!] \implies P$   
 $\quad ]\!] \implies P$   
**by** (*auto simp add: usum-def*)

**lemma** *In0-not-In1* [*iff*]:  $\text{In0}(M) \neq \text{In1}(N)$

**by** (*auto simp add: In0-def In1-def One-nat-def*)

**lemmas** *In1-not-In0* [iff] = *In0-not-In1* [THEN *not-sym*, *standard*]

**lemma** *In0-inject*:  $In0(M) = In0(N) \implies M=N$   
**by** (*simp add: In0-def*)

**lemma** *In1-inject*:  $In1(M) = In1(N) \implies M=N$   
**by** (*simp add: In1-def*)

**lemma** *In0-eq* [iff]:  $(In0\ M = In0\ N) = (M=N)$   
**by** (*blast dest!: In0-inject*)

**lemma** *In1-eq* [iff]:  $(In1\ M = In1\ N) = (M=N)$   
**by** (*blast dest!: In1-inject*)

**lemma** *inj-In0*: *inj In0*  
**by** (*blast intro!: inj-onI*)

**lemma** *inj-In1*: *inj In1*  
**by** (*blast intro!: inj-onI*)

**lemma** *Lim-inject*:  $Lim\ f = Lim\ g \implies f = g$   
**apply** (*simp add: Lim-def*)  
**apply** (*rule ext*)  
**apply** (*blast elim!: Push-Node-inject*)  
**done**

**lemma** *ntrunc-subsetI*:  $ntrunc\ k\ M \leq M$   
**by** (*auto simp add: ntrunc-def*)

**lemma** *ntrunc-subsetD*:  $(!!k. ntrunc\ k\ M \leq N) \implies M \leq N$   
**by** (*auto simp add: ntrunc-def*)

**lemma** *ntrunc-equality*:  $(!!k. ntrunc\ k\ M = ntrunc\ k\ N) \implies M=N$   
**apply** (*rule equalityI*)  
**apply** (*rule-tac* [!] *ntrunc-subsetD*)  
**apply** (*rule-tac* [!] *ntrunc-subsetI* [THEN [2] *subset-trans*], *auto*)  
**done**

**lemma** *ntrunc-o-equality*:  
 $[ [!!k. (ntrunc(k) \circ h1) = (ntrunc(k) \circ h2) ] ] \implies h1=h2$

```

apply (rule ntrunc-equality [THEN ext])
apply (simp add: expand-fun-eq)
done

```

```

lemma uprod-mono: [| A<=A'; B<=B' |] ==> uprod A B <= uprod A' B'
by (simp add: uprod-def, blast)

```

```

lemma usum-mono: [| A<=A'; B<=B' |] ==> usum A B <= usum A' B'
by (simp add: usum-def, blast)

```

```

lemma Scons-mono: [| M<=M'; N<=N' |] ==> Scons M N <= Scons M' N'
by (simp add: Scons-def, blast)

```

```

lemma In0-mono: M<=N ==> In0(M) <= In0(N)
by (simp add: In0-def subset-refl Scons-mono)

```

```

lemma In1-mono: M<=N ==> In1(M) <= In1(N)
by (simp add: In1-def subset-refl Scons-mono)

```

```

lemma Split [simp]: Split c (Scons M N) = c M N
by (simp add: Split-def)

```

```

lemma Case-In0 [simp]: Case c d (In0 M) = c(M)
by (simp add: Case-def)

```

```

lemma Case-In1 [simp]: Case c d (In1 N) = d(N)
by (simp add: Case-def)

```

```

lemma ntrunc-UN1: ntrunc k (UN x. f(x)) = (UN x. ntrunc k (f x))
by (simp add: ntrunc-def, blast)

```

```

lemma Scons-UN1-x: Scons (UN x. f x) M = (UN x. Scons (f x) M)
by (simp add: Scons-def, blast)

```

```

lemma Scons-UN1-y: Scons M (UN x. f x) = (UN x. Scons M (f x))
by (simp add: Scons-def, blast)

```

```

lemma In0-UN1: In0(UN x. f(x)) = (UN x. In0(f(x)))
by (simp add: In0-def Scons-UN1-y)

```

**lemma** *In1-UN1*:  $\text{In1}(\text{UN } x. f(x)) = (\text{UN } x. \text{In1}(f(x)))$   
**by** (*simp add: In1-def Scons-UN1-y*)

**lemma** *dprodI* [*intro!*]:  
 $\llbracket (M, M'):r; (N, N'):s \rrbracket \implies (\text{Scons } M \ N, \text{Scons } M' \ N') : \text{dprod } r \ s$   
**by** (*auto simp add: dprod-def*)

**lemma** *dprodE* [*elim!*]:  
 $\llbracket c : \text{dprod } r \ s; \text{!!}x \ y \ x' \ y'. \llbracket (x, x') : r; (y, y') : s; c = (\text{Scons } x \ y, \text{Scons } x' \ y') \rrbracket \implies P$   
 $\llbracket \rrbracket \implies P$   
**by** (*auto simp add: dprod-def*)

**lemma** *dsum-In0I* [*intro*]:  $(M, M'):r \implies (\text{In0}(M), \text{In0}(M')) : \text{dsum } r \ s$   
**by** (*auto simp add: dsum-def*)

**lemma** *dsum-In1I* [*intro*]:  $(N, N'):s \implies (\text{In1}(N), \text{In1}(N')) : \text{dsum } r \ s$   
**by** (*auto simp add: dsum-def*)

**lemma** *dsumE* [*elim!*]:  
 $\llbracket w : \text{dsum } r \ s; \text{!!}x \ x'. \llbracket (x, x') : r; w = (\text{In0}(x), \text{In0}(x')) \rrbracket \implies P; \text{!!}y \ y'. \llbracket (y, y') : s; w = (\text{In1}(y), \text{In1}(y')) \rrbracket \implies P$   
 $\llbracket \rrbracket \implies P$   
**by** (*auto simp add: dsum-def*)

**lemma** *dprod-mono*:  $\llbracket r \leq r'; s \leq s' \rrbracket \implies \text{dprod } r \ s \leq \text{dprod } r' \ s'$   
**by** *blast*

**lemma** *dsum-mono*:  $\llbracket r \leq r'; s \leq s' \rrbracket \implies \text{dsum } r \ s \leq \text{dsum } r' \ s'$   
**by** *blast*

**lemma** *dprod-Sigma*:  $(\text{dprod } (A \lt*> B) (C \lt*> D)) \leq (\text{uprod } A \ C) \lt*> (\text{uprod } B \ D)$

by *blast*

**lemmas** *dprod-subset-Sigma* = *subset-trans* [*OF dprod-mono dprod-Sigma, standard*]

**lemma** *dprod-subset-Sigma2*:  
 (*dprod* (*Sigma A B*) (*Sigma C D*)) <=  
*Sigma* (*uprod A C*) (*Split* (%*x y. uprod* (*B x*) (*D y*)))  
 by *auto*

**lemma** *dsum-Sigma*: (*dsum* (*A <\*> B*) (*C <\*> D*)) <= (*usum A C*) <\*> (*usum B D*)  
 by *blast*

**lemmas** *dsum-subset-Sigma* = *subset-trans* [*OF dsum-mono dsum-Sigma, standard*]

hides popular names

**hide** (**open**) *type node item*

**hide** (**open**) *const Push Node Atom Leaf Numb Lim Split Case*

## 15 Datatypes

### 15.1 Representing sums

**rep-datatype** (*sum*) *Inl Inr*

**proof** —

**fix** *P*  
**fix** *s :: 'a + 'b*  
**assume** *x:  $\bigwedge x::'a. P$  (*Inl x*) and *y:  $\bigwedge y::'b. P$  (*Inr y*)*  
**then show** *P s* **by** (*auto intro: sumE [of s]*)  
**qed** *simp-all**

**lemma** *sum-case-KK[*simp*]*: *sum-case* (%*x. a*) (%*x. a*) = (%*x. a*)  
**by** (*rule ext*) (*simp split: sum.split*)

**lemma** *surjective-sum*: *sum-case* (%*x::'a. f* (*Inl x*)) (%*y::'b. f* (*Inr y*)) *s* = *f(s)*  
**apply** (*rule-tac s = s in sumE*)  
**apply** (*erule ssubst*)  
**apply** (*rule sum.cases(1)*)  
**apply** (*erule ssubst*)  
**apply** (*rule sum.cases(2)*)  
**done**

**lemma** *sum-case-weak-cong*: *s = t ==> sum-case f g s = sum-case f g t*  
 — Prevents simplification of *f* and *g*: much faster.  
**by** *simp*



**lemma** *sum-case-inject*:

*sum-case f1 f2 = sum-case g1 g2 ==> (f1 = g1 ==> f2 = g2 ==> P) ==> P*

**proof** –

**assume** *a*: *sum-case f1 f2 = sum-case g1 g2*

**assume** *r*: *f1 = g1 ==> f2 = g2 ==> P*

**show** *P*

**apply** (*rule r*)

**apply** (*rule ext*)

**apply** (*cut-tac x = Inl x in a [THEN fun-cong], simp*)

**apply** (*rule ext*)

**apply** (*cut-tac x = Inr x in a [THEN fun-cong], simp*)

**done**

**qed**

**constdefs**

*Suml* :: (*'a* => *'c*) => *'a* + *'b* => *'c*

*Suml* == (%*f*. *sum-case f undefined*)

*Sumr* :: (*'b* => *'c*) => *'a* + *'b* => *'c*

*Sumr* == *sum-case undefined*

**lemma** *Suml-inject*: *Suml f = Suml g ==> f = g*

**by** (*unfold Suml-def*) (*erule sum-case-inject*)

**lemma** *Sumr-inject*: *Sumr f = Sumr g ==> f = g*

**by** (*unfold Sumr-def*) (*erule sum-case-inject*)

**primrec** *Projl* :: *'a* + *'b* => *'a*

**where** *Projl-Inl*: *Projl (Inl x) = x*

**primrec** *Projr* :: *'a* + *'b* => *'b*

**where** *Projr-Inr*: *Projr (Inr x) = x*

**hide (open)** *const Suml Sumr Projl Projr*

**end**

## 16 Power: Exponentiation

**theory** *Power*

**imports** *Nat*

**begin**

**class** *power* =

**fixes** *power* :: *'a* => *nat* => *'a* (infixr ^ 80)

## 16.1 Powers for Arbitrary Monoids

```

class recpower = monoid-mult + power +
  assumes power-0 [simp]:  $a ^ 0 = 1$ 
  assumes power-Suc [simp]:  $a ^ \text{Suc } n = a * (a ^ n)$ 

```

```

lemma power-0-Suc [simp]:  $(0::'a::\{\text{recpower}, \text{semiring-0}\}) ^ (\text{Suc } n) = 0$ 
  by simp

```

It looks plausible as a simprule, but its effect can be strange.

```

lemma power-0-left:  $0 ^ n = (\text{if } n=0 \text{ then } 1 \text{ else } (0::'a::\{\text{recpower}, \text{semiring-0}\}))$ 
  by (induct n) simp-all

```

```

lemma power-one [simp]:  $1 ^ n = (1::'a::\text{recpower})$ 
  by (induct n) simp-all

```

```

lemma power-one-right [simp]:  $(a::'a::\text{recpower}) ^ 1 = a$ 
  unfolding One-nat-def by simp

```

```

lemma power-commutes:  $(a::'a::\text{recpower}) ^ n * a = a * a ^ n$ 
  by (induct n) (simp-all add: mult-assoc)

```

```

lemma power-Suc2:  $(a::'a::\text{recpower}) ^ \text{Suc } n = a ^ n * a$ 
  by (simp add: power-commutes)

```

```

lemma power-add:  $(a::'a::\text{recpower}) ^ (m+n) = (a ^ m) * (a ^ n)$ 
  by (induct m) (simp-all add: mult-ac)

```

```

lemma power-mult:  $(a::'a::\text{recpower}) ^ (m*n) = (a ^ m) ^ n$ 
  by (induct n) (simp-all add: power-add)

```

```

lemma power-mult-distrib:  $((a::'a::\{\text{recpower}, \text{comm-monoid-mult}\}) * b) ^ n =$ 
 $(a ^ n) * (b ^ n)$ 
  by (induct n) (simp-all add: mult-ac)

```

```

lemma zero-less-power[simp]:
   $0 < (a::'a::\{\text{ordered-semidom}, \text{recpower}\}) ==> 0 < a ^ n$ 
  by (induct n) (simp-all add: mult-pos-pos)

```

```

lemma zero-le-power[simp]:
   $0 \leq (a::'a::\{\text{ordered-semidom}, \text{recpower}\}) ==> 0 \leq a ^ n$ 
  by (induct n) (simp-all add: mult-nonneg-nonneg)

```

```

lemma one-le-power[simp]:
   $1 \leq (a::'a::\{\text{ordered-semidom}, \text{recpower}\}) ==> 1 \leq a ^ n$ 
  apply (induct n)
  apply simp-all
  apply (rule order-trans [OF - mult-mono [of 1 - 1]])
  apply (simp-all add: order-trans [OF zero-le-one])
  done

```

```

lemma gt1-imp-ge0:  $1 < a \implies 0 \leq (a::'a::\text{ordered-semidom})$ 
  by (simp add: order-trans [OF zero-le-one order-less-imp-le])

lemma power-gt1-lemma:
  assumes gt1:  $1 < (a::'a::\{\text{ordered-semidom}, \text{recpower}\})$ 
  shows  $1 < a * a^n$ 
proof –
  have  $1 * 1 < a * 1$  using gt1 by simp
  also have  $\dots \leq a * a^n$  using gt1
    by (simp only: mult-mono gt1-imp-ge0 one-le-power order-less-imp-le
      zero-le-one order-refl)
  finally show ?thesis by simp
qed

lemma one-less-power[simp]:
   $\llbracket 1 < (a::'a::\{\text{ordered-semidom}, \text{recpower}\}); 0 < n \rrbracket \implies 1 < a^n$ 
by (cases n, simp-all add: power-gt1-lemma)

lemma power-gt1:
   $1 < (a::'a::\{\text{ordered-semidom}, \text{recpower}\}) \implies 1 < a^{(\text{Suc } n)}$ 
by (simp add: power-gt1-lemma)

lemma power-le-imp-le-exp:
  assumes gt1:  $(1::'a::\{\text{recpower}, \text{ordered-semidom}\}) < a$ 
  shows  $!!n. a^m \leq a^n \implies m \leq n$ 
proof (induct m)
  case 0
  show ?case by simp
next
  case (Suc m)
  show ?case
  proof (cases n)
  case 0
  from prems have  $a * a^m \leq 1$  by simp
  with gt1 show ?thesis
    by (force simp only: power-gt1-lemma
      linorder-not-less [symmetric])
  next
  case (Suc n)
  from prems show ?thesis
    by (force dest: mult-left-le-imp-le
      simp add: order-less-trans [OF zero-less-one gt1])
  qed
qed

```

Surely we can strengthen this? It holds for  $0 < a < 1$  too.

```

lemma power-inject-exp [simp]:
   $1 < (a::'a::\{\text{ordered-semidom}, \text{recpower}\}) \implies (a^m = a^n) = (m=n)$ 

```

**by** (*force simp add: order-antisym power-le-imp-le-exp*)

Can relax the first premise to  $(0::'a) < a$  in the case of the natural numbers.

**lemma** *power-less-imp-less-exp*:

$[[ (1::'a::\{\text{recpower, ordered-semidom}\}) < a; a^m < a^n ]] ==> m < n$

**by** (*simp add: order-less-le [of m n] order-less-le [of a^m a^n]*  
*power-le-imp-le-exp*)

**lemma** *power-mono*:

$[[ a \leq b; (0::'a::\{\text{recpower, ordered-semidom}\}) \leq a ]] ==> a^n \leq b^n$

**apply** (*induct n*)

**apply** *simp-all*

**apply** (*auto intro: mult-mono order-trans [of 0 a b]*)

**done**

**lemma** *power-strict-mono* [*rule-format*]:

$[[ a < b; (0::'a::\{\text{recpower, ordered-semidom}\}) \leq a ]] ==>$

$0 < n \longrightarrow a^n < b^n$

**apply** (*induct n*)

**apply** (*auto simp add: mult-strict-mono order-le-less-trans [of 0 a b]*)

**done**

**lemma** *power-eq-0-iff* [*simp*]:

$(a^n = 0) \longleftrightarrow$

$(a = (0::'a::\{\text{mult-zero, zero-neq-one, no-zero-divisors, recpower}\}) \ \& \ n \neq 0)$

**apply** (*induct n*)

**apply** (*auto simp add: no-zero-divisors*)

**done**

**lemma** *field-power-not-zero*:

$a \neq (0::'a::\{\text{ring-1-no-zero-divisors, recpower}\}) ==> a^n \neq 0$

**by** *force*

**lemma** *nonzero-power-inverse*:

**fixes**  $a :: 'a::\{\text{division-ring, recpower}\}$

**shows**  $a \neq 0 ==> \text{inverse } (a^n) = (\text{inverse } a)^n$

**apply** (*induct n*)

**apply** (*auto simp add: nonzero-inverse-mult-distrib power-commutes*)

**done**

Perhaps these should be simprules.

**lemma** *power-inverse*:

**fixes**  $a :: 'a::\{\text{division-ring, division-by-zero, recpower}\}$

**shows**  $\text{inverse } (a^n) = (\text{inverse } a)^n$

**apply** (*cases a = 0*)

**apply** (*simp add: power-0-left*)

**apply** (*simp add: nonzero-power-inverse*)

done

**lemma** *power-one-over*:  $1 / (a :: 'a :: \{\text{field}, \text{division-by-zero}, \text{recpower}\})^n =$   
 $(1 / a)^n$   
**apply** (*simp add: divide-inverse*)  
**apply** (*rule power-inverse*)  
**done**

**lemma** *nonzero-power-divide*:  
 $b \neq 0 \implies (a/b)^n = ((a :: 'a :: \{\text{field}, \text{recpower}\})^n) / (b^n)$   
**by** (*simp add: divide-inverse power-mult-distrib nonzero-power-inverse*)

**lemma** *power-divide*:  
 $(a/b)^n = ((a :: 'a :: \{\text{field}, \text{division-by-zero}, \text{recpower}\})^n) / b^n$   
**apply** (*case-tac b=0, simp add: power-0-left*)  
**apply** (*rule nonzero-power-divide*)  
**apply** *assumption*  
**done**

**lemma** *power-abs*:  $\text{abs}(a^n) = \text{abs}(a :: 'a :: \{\text{ordered-idom}, \text{recpower}\})^n$   
**apply** (*induct n*)  
**apply** (*auto simp add: abs-mult*)  
**done**

**lemma** *abs-power-minus* [*simp*]:  
**fixes**  $a :: 'a :: \{\text{ordered-idom}, \text{recpower}\}$  **shows**  $\text{abs}((-a)^n) = \text{abs}(a^n)$   
**by** (*simp add: abs-minus-cancel power-abs*)

**lemma** *zero-less-power-abs-iff* [*simp, noatp*]:  
 $(0 < \text{abs } a)^n = (a \neq (0 :: 'a :: \{\text{ordered-idom}, \text{recpower}\}) \mid n=0)$   
**proof** (*induct n*)  
**case** 0  
**show** ?case **by** *simp*  
**next**  
**case** (*Suc n*)  
**show** ?case **by** (*auto simp add: prems zero-less-mult-iff*)  
**qed**

**lemma** *zero-le-power-abs* [*simp*]:  
 $(0 :: 'a :: \{\text{ordered-idom}, \text{recpower}\}) \leq (\text{abs } a)^n$   
**by** (*rule zero-le-power [OF abs-ge-zero]*)

**lemma** *power-minus*:  $(-a)^n = (-1)^n * (a :: 'a :: \{\text{ring-1}, \text{recpower}\})^n$   
**proof** (*induct n*)  
**case** 0 **show** ?case **by** *simp*  
**next**  
**case** (*Suc n*) **then show** ?case  
**by** (*simp del: power-Suc add: power-Suc2 mult-assoc*)  
**qed**

Lemma for *power-strict-decreasing*

**lemma** *power-Suc-less*:

$[[ (0 :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}) < a; a < 1 ]]$   
 $\implies a * a^n < a^n$

**apply** (*induct n*)

**apply** (*auto simp add: mult-strict-left-mono*)

**done**

**lemma** *power-strict-decreasing*:

$[[ n < N; 0 < a; a < (1 :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}) ]]$   
 $\implies a^N < a^n$

**apply** (*erule rev-mp*)

**apply** (*induct N*)

**apply** (*auto simp add: power-Suc-less less-Suc-eq*)

**apply** (*rename-tac m*)

**apply** (*subgoal-tac a \* a^m < 1 \* a^n, simp*)

**apply** (*rule mult-strict-mono*)

**apply** (*auto simp add: order-less-imp-le*)

**done**

Proof resembles that of *power-strict-decreasing*

**lemma** *power-decreasing*:

$[[ n \leq N; 0 \leq a; a \leq (1 :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}) ]]$   
 $\implies a^N \leq a^n$

**apply** (*erule rev-mp*)

**apply** (*induct N*)

**apply** (*auto simp add: le-Suc-eq*)

**apply** (*rename-tac m*)

**apply** (*subgoal-tac a \* a^m \leq 1 \* a^n, simp*)

**apply** (*rule mult-mono*)

**apply** *auto*

**done**

**lemma** *power-Suc-less-one*:

$[[ 0 < a; a < (1 :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}) ]] \implies a^{\text{Suc } n} < 1$

**apply** (*insert power-strict-decreasing [of 0 Suc n a], simp*)

**done**

Proof again resembles that of *power-strict-decreasing*

**lemma** *power-increasing*:

$[[ n \leq N; (1 :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}) \leq a ]] \implies a^n \leq a^N$

**apply** (*erule rev-mp*)

**apply** (*induct N*)

**apply** (*auto simp add: le-Suc-eq*)

**apply** (*rename-tac m*)

**apply** (*subgoal-tac 1 \* a^n \leq a \* a^m, simp*)

**apply** (*rule mult-mono*)

**apply** (*auto simp add: order-trans [OF zero-le-one]*)

**done**

Lemma for *power-strict-increasing*

**lemma** *power-less-power-Suc*:

( $1 :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}$ )  $< a \implies a^n < a * a^n$   
**apply** (*induct*  $n$ )  
**apply** (*auto simp add: mult-strict-left-mono order-less-trans [OF zero-less-one]*)  
**done**

**lemma** *power-strict-increasing*:

( $[n < N; (1 :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}) < a]$ )  $\implies a^n < a^N$   
**apply** (*erule rev-mp*)  
**apply** (*induct*  $N$ )  
**apply** (*auto simp add: power-less-power-Suc less-Suc-eq*)  
**apply** (*rename-tac*  $m$ )  
**apply** (*subgoal-tac*  $1 * a^n < a * a^m$ , *simp*)  
**apply** (*rule mult-strict-mono*)  
**apply** (*auto simp add: order-less-trans [OF zero-less-one] order-less-imp-le*)  
**done**

**lemma** *power-increasing-iff [simp]*:

$1 < (b :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}) \implies (b^x \leq b^y) = (x \leq y)$   
**by** (*blast intro: power-le-imp-le-exp power-increasing order-less-imp-le*)

**lemma** *power-strict-increasing-iff [simp]*:

$1 < (b :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}) \implies (b^x < b^y) = (x < y)$   
**by** (*blast intro: power-less-imp-less-exp power-strict-increasing*)

**lemma** *power-le-imp-le-base*:

**assumes**  $le: a^{\text{Suc } n} \leq b^{\text{Suc } n}$   
**and**  $ynonneg: (0 :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}) \leq b$   
**shows**  $a \leq b$   
**proof** (*rule ccontr*)  
**assume**  $\sim a \leq b$   
**then have**  $b < a$  **by** (*simp only: linorder-not-le*)  
**then have**  $b^{\text{Suc } n} < a^{\text{Suc } n}$   
**by** (*simp only: prems power-strict-mono*)  
**from**  $le$  **and this** **show** *False*  
**by** (*simp add: linorder-not-less [symmetric]*)  
**qed**

**lemma** *power-less-imp-less-base*:

**fixes**  $a b :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}$   
**assumes**  $less: a^n < b^n$   
**assumes**  $nonneg: 0 \leq b$   
**shows**  $a < b$   
**proof** (*rule contrapos-pp [OF less]*)  
**assume**  $\sim a < b$   
**hence**  $b \leq a$  **by** (*simp only: linorder-not-less*)  
**hence**  $b^n \leq a^n$  **using**  $nonneg$  **by** (*rule power-mono*)  
**thus**  $\sim a^n < b^n$  **by** (*simp only: linorder-not-less*)

qed

**lemma** *power-inject-base*:

$\llbracket a \wedge \text{Suc } n = b \wedge \text{Suc } n; 0 \leq a; 0 \leq b \rrbracket$   
 $\implies a = (b :: 'a :: \{\text{ordered-semidom}, \text{recpower}\})$

**by** (*blast intro: power-le-imp-le-base order-antisym order-eq-refl sym*)

**lemma** *power-eq-imp-eq-base*:

**fixes**  $a \ b :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}$

**shows**  $\llbracket a \wedge n = b \wedge n; 0 \leq a; 0 \leq b; 0 < n \rrbracket \implies a = b$

**by** (*cases n, simp-all del: power-Suc, rule power-inject-base*)

The divides relation

**lemma** *le-imp-power-dvd*:

**fixes**  $a :: 'a :: \{\text{comm-semiring-1}, \text{recpower}\}$

**assumes**  $m \leq n$  **shows**  $a^m \text{ dvd } a^n$

**proof**

**have**  $a^n = a^{(m + (n - m))}$

**using**  $\langle m \leq n \rangle$  **by** *simp*

**also have**  $\dots = a^m * a^{(n - m)}$

**by** (*rule power-add*)

**finally show**  $a^n = a^m * a^{(n - m)}$  .

qed

**lemma** *power-le-dvd*:

**fixes**  $a \ b :: 'a :: \{\text{comm-semiring-1}, \text{recpower}\}$

**shows**  $a^n \text{ dvd } b \implies m \leq n \implies a^m \text{ dvd } b$

**by** (*rule dvd-trans [OF le-imp-power-dvd]*)

**lemma** *dvd-power-same*:

$(x :: 'a :: \{\text{comm-semiring-1}, \text{recpower}\}) \text{ dvd } y \implies x^n \text{ dvd } y^n$

**by** (*induct n (auto simp add: mult-dvd-mono)*)

**lemma** *dvd-power-le*:

$(x :: 'a :: \{\text{comm-semiring-1}, \text{recpower}\}) \text{ dvd } y \implies m \geq n \implies x^n \text{ dvd } y^m$

**by** (*rule power-le-dvd [OF dvd-power-same]*)

**lemma** *dvd-power [simp]*:

$n > 0 \mid (x :: 'a :: \{\text{comm-semiring-1}, \text{recpower}\}) = 1 \implies x \text{ dvd } x^n$

**apply** (*erule disjE*)

**apply** (*subgoal-tac  $x \wedge n = x^{(\text{Suc } (n - 1))}$* )

**apply** (*erule ssubst*)

**apply** (*subst power-Suc*)

**apply** *auto*

done



## 16.2 Exponentiation for the Natural Numbers

**instantiation** *nat* :: *recpower*

**begin**

**primrec** *power-nat* **where**

$p \wedge 0 = (1::nat)$

$| p \wedge (Suc\ n) = (p::nat) * (p \wedge n)$

**instance** **proof**

**fix** *z n* :: *nat*

**show**  $z \wedge 0 = 1$  **by** *simp*

**show**  $z \wedge (Suc\ n) = z * (z \wedge n)$  **by** *simp*

**qed**

**declare** *power-nat.simps* [*simp del*]

**end**

**lemma** *of-nat-power*:

$of\_nat\ (m \wedge n) = (of\_nat\ m::'a::\{semiring-1,recpower\}) \wedge n$

**by** (*induct n, simp-all add: of-nat-mult*)

**lemma** *nat-one-le-power* [*simp*]:  $Suc\ 0 \leq i \implies Suc\ 0 \leq i \wedge n$

**by** (*rule one-le-power [of i n, unfolded One-nat-def]*)

**lemma** *nat-zero-less-power-iff* [*simp*]:  $(x \wedge n > 0) = (x > (0::nat) \mid n=0)$

**by** (*induct n, auto*)

**lemma** *nat-power-eq-Suc-0-iff* [*simp*]:

$((x::nat) \wedge m = Suc\ 0) = (m = 0 \mid x = Suc\ 0)$

**by** (*induct-tac m, auto*)

**lemma** *power-Suc-0* [*simp*]:  $(Suc\ 0) \wedge n = Suc\ 0$

**by** *simp*

Valid for the naturals, but what if  $0 < i < 1$ ? Premises cannot be weakened: consider the case where  $i = (0::'a)$ ,  $m = (1::'a)$  and  $n = (0::'a)$ .

**lemma** *nat-power-less-imp-less*:

**assumes** *nonneg*:  $0 < (i::nat)$

**assumes** *less*:  $i \wedge m < i \wedge n$

**shows**  $m < n$

**proof** (*cases i = 1*)

**case** *True* **with** *less power-one* [**where**  $'a = nat$ ] **show** *?thesis* **by** *simp*

**next**

**case** *False* **with** *nonneg* **have**  $1 < i$  **by** *auto*

**from** *power-strict-increasing-iff* [*OF this*] *less* **show** *?thesis* ..

**qed**

**lemma** *power-diff*:

```

assumes nz:  $a \sim = 0$ 
shows  $n \leq m \implies (a :: 'a :: \{\text{recpower}, \text{field}\}) \wedge (m - n) = (a^m) / (a^n)$ 
by (induct m n rule: diff-induct)
      (simp-all add: nonzero-mult-divide-cancel-left nz)

end

```

## 17 Finite-Set: Finite sets

```

theory Finite-Set
imports Nat Product-Type Power
begin

```

### 17.1 Definition and basic properties

```

inductive finite :: 'a set => bool
```

**where**

```

  emptyI [simp, intro!]: finite {}
  | insertI [simp, intro!]: finite A ==> finite (insert a A)
```

**lemma** *ex-new-if-finite*: — does not depend on def of finite at all

```

assumes  $\neg \text{finite } (\text{UNIV} :: 'a \text{ set})$  and finite A
shows  $\exists a :: 'a. a \notin A$ 

```

**proof** —

```

from assms have  $A \neq \text{UNIV}$  by blast
thus ?thesis by blast

```

**qed**

**lemma** *finite-induct* [*case-names empty insert, induct set: finite*]:

```

  finite F ==>
     $P \{ \} \implies (!x F. \text{finite } F \implies x \notin F \implies P F \implies P (\text{insert } x F)) \implies$ 
     $P F$ 

```

— Discharging  $x \notin F$  entails extra work.

**proof** —

```

assume  $P \{ \}$  and
  insert:  $!x F. \text{finite } F \implies x \notin F \implies P F \implies P (\text{insert } x F)$ 
assume finite F
thus  $P F$ 
proof induct
  show  $P \{ \}$  by fact
  fix x F assume  $F: \text{finite } F$  and  $P: P F$ 
  show  $P (\text{insert } x F)$ 
  proof cases
    assume  $x \in F$ 
    hence  $\text{insert } x F = F$  by (rule insert-absorb)
    with  $P$  show ?thesis by (simp only.)
  next
    assume  $x \notin F$ 

```

```

    from  $F$  this  $P$  show ?thesis by (rule insert)
  qed
qed
qed

```

**lemma** *finite-ne-induct* [*case-names singleton insert, consumes 2*]:

**assumes** *fin*: *finite*  $F$  **shows**  $F \neq \{\}$   $\implies$

$\llbracket \bigwedge x. P\{x\};$   
 $\bigwedge x F. \llbracket \text{finite } F; F \neq \{\}; x \notin F; P F \rrbracket \implies P (\text{insert } x F) \rrbracket$   
 $\implies P F$

**using** *fin*

**proof** *induct*

**case** *empty* **thus** ?*case* **by** *simp*

**next**

**case** (*insert*  $x F$ )

**show** ?*case*

**proof** *cases*

**assume**  $F = \{\}$

**thus** ?*thesis* **using**  $\langle P \{x\} \rangle$  **by** *simp*

**next**

**assume**  $F \neq \{\}$

**thus** ?*thesis* **using** *insert* **by** *blast*

**qed**

**qed**

**lemma** *finite-subset-induct* [*consumes 2, case-names empty insert*]:

**assumes** *finite*  $F$  **and**  $F \subseteq A$

**and** *empty*:  $P \{\}$

**and** *insert*:  $\forall a F. \text{finite } F \implies a \in A \implies a \notin F \implies P F \implies P (\text{insert } a F)$

**shows**  $P F$

**proof**  $-$

**from**  $\langle \text{finite } F \rangle$  **and**  $\langle F \subseteq A \rangle$

**show** ?*thesis*

**proof** *induct*

**show**  $P \{\}$  **by** *fact*

**next**

**fix**  $x F$

**assume** *finite*  $F$  **and**  $x \notin F$  **and**

$P: F \subseteq A \implies P F$  **and**  $i: \text{insert } x F \subseteq A$

**show**  $P (\text{insert } x F)$

**proof** (*rule insert*)

**from**  $i$  **show**  $x \in A$  **by** *blast*

**from**  $i$  **have**  $F \subseteq A$  **by** *blast*

**with**  $P$  **show**  $P F$  .

**show** *finite*  $F$  **by** *fact*

**show**  $x \notin F$  **by** *fact*

**qed**

**qed**

qed

A finite choice principle. Does not need the SOME choice operator.

**lemma** *finite-set-choice*:

$finite\ A \implies ALL\ x:A. (EX\ y. P\ x\ y) \implies EX\ f. ALL\ x:A. P\ x\ (f\ x)$

**proof** (*induct set: finite*)

**case** *empty* **thus** ?case **by** *simp*

**next**

**case** (*insert a A*)

**then** **obtain** *f b* **where**  $f: ALL\ x:A. P\ x\ (f\ x)$  **and**  $ab: P\ a\ b$  **by** *auto*

**show** ?case (**is**  $EX\ f. ?P\ f$ )

**proof**

**show**  $?P(\%x. if\ x = a\ then\ b\ else\ f\ x)$  **using** *f ab* **by** *auto*

qed

qed

Finite sets are the images of initial segments of natural numbers:

**lemma** *finite-imp-nat-seg-image-inj-on*:

**assumes** *fin: finite A*

**shows**  $\exists\ (n::nat)\ f. A = f\ ' \{i. i < n\} \ \&\ inj-on\ f\ \{i. i < n\}$

**using** *fin*

**proof** *induct*

**case** *empty*

**show** ?case

**proof** **show**  $\exists f. \{\} = f\ ' \{i::nat. i < 0\} \ \&\ inj-on\ f\ \{i. i < 0\}$  **by** *simp*

qed

**next**

**case** (*insert a A*)

**have** *notinA: a  $\notin$  A* **by** *fact*

**from** *insert.hyps* **obtain** *n f*

**where**  $A = f\ ' \{i::nat. i < n\} \ inj-on\ f\ \{i. i < n\}$  **by** *blast*

**hence**  $insert\ a\ A = f(n:=a)\ ' \{i. i < Suc\ n\}$

$inj-on\ (f(n:=a))\ \{i. i < Suc\ n\}$  **using** *notinA*

**by** (*auto simp add: image-def Ball-def inj-on-def less-Suc-eq*)

**thus** ?case **by** *blast*

qed

**lemma** *nat-seg-image-imp-finite*:

$!!f\ A. A = f\ ' \{i::nat. i < n\} \implies finite\ A$

**proof** (*induct n*)

**case** 0 **thus** ?case **by** *simp*

**next**

**case** (*Suc n*)

**let**  $?B = f\ ' \{i. i < n\}$

**have** *finB: finite ?B* **by** (*rule Suc.hyps[OF refl]*)

**show** ?case

**proof** *cases*

**assume**  $\exists k < n. f\ n = f\ k$

**hence**  $A = ?B$  **using** *Suc.premis* **by** (*auto simp: less-Suc-eq*)

```

    thus ?thesis using finB by simp
  next
    assume  $\neg(\exists k < n. f\ n = f\ k)$ 
    hence  $A = \text{insert } (f\ n) \ ?B$  using Suc.prem by (auto simp: less-Suc-eq)
    thus ?thesis using finB by simp
  qed
qed

```

**lemma** *finite-conv-nat-seg-image*:  
 $\text{finite } A = (\exists (n::\text{nat})\ f. A = f\ ' \{i::\text{nat}. i < n\})$   
**by** (blast intro: nat-seg-image-imp-finite dest: finite-imp-nat-seg-image-inj-on)

**lemma** *finite-Collect-less-nat*[iff]:  $\text{finite}\{n::\text{nat}. n < k\}$   
**by** (fastsimp simp: finite-conv-nat-seg-image)

### 17.1.1 Finiteness and set theoretic constructions

**lemma** *finite-UnI*:  $\text{finite } F \implies \text{finite } G \implies \text{finite } (F \text{ Un } G)$   
**by** (induct set: finite) simp-all

**lemma** *finite-subset*:  $A \subseteq B \implies \text{finite } B \implies \text{finite } A$   
 — Every subset of a finite set is finite.

**proof** —  
 assume finite B  
 thus !!A.  $A \subseteq B \implies \text{finite } A$   
**proof** induct  
 case empty  
 thus ?case by simp  
 next  
 case (insert x F A)  
 have  $A \subseteq \text{insert } x\ F$  and  $r: A - \{x\} \subseteq F \implies \text{finite } (A - \{x\})$  by fact+  
 show finite A  
**proof** cases  
 assume  $x: x \in A$   
 with A have  $A - \{x\} \subseteq F$  by (simp add: subset-insert-iff)  
 with r have finite (A - {x}) .  
 hence finite (insert x (A - {x})) ..  
 also have  $\text{insert } x\ (A - \{x\}) = A$  using x by (rule insert-Diff)  
 finally show ?thesis .  
 next  
 show  $A \subseteq F \implies ?thesis$  by fact  
 assume  $x \notin A$   
 with A show  $A \subseteq F$  by (simp add: subset-insert-iff)  
 qed  
 qed  
 qed

**lemma** *finite-Un* [iff]:  $\text{finite } (F \text{ Un } G) = (\text{finite } F \ \& \ \text{finite } G)$   
**by** (blast intro: finite-subset [of - X Un Y, standard] finite-UnI)

**lemma** *finite-Collect-disjI* [*simp*]:

$\text{finite}\{x. P\ x \mid Q\ x\} = (\text{finite}\{x. P\ x\} \ \& \ \text{finite}\{x. Q\ x\})$

**by** (*simp add: Collect-disj-eq*)

**lemma** *finite-Int* [*simp, intro*]:  $\text{finite}\ F \mid \text{finite}\ G \implies \text{finite}\ (F \text{ Int } G)$

— The converse obviously fails.

**by** (*blast intro: finite-subset*)

**lemma** *finite-Collect-conjI* [*simp, intro*]:

$\text{finite}\{x. P\ x\} \mid \text{finite}\{x. Q\ x\} \implies \text{finite}\{x. P\ x \ \& \ Q\ x\}$

— The converse obviously fails.

**by** (*simp add: Collect-conj-eq*)

**lemma** *finite-Collect-le-nat* [*iff*]:  $\text{finite}\{n::\text{nat}. n \leq k\}$

**by** (*simp add: le-eq-less-or-eq*)

**lemma** *finite-insert* [*simp*]:  $\text{finite}\ (\text{insert}\ a\ A) = \text{finite}\ A$

**apply** (*subst insert-is-Un*)

**apply** (*simp only: finite-Un, blast*)

**done**

**lemma** *finite-Union* [*simp, intro*]:

$\llbracket \text{finite}\ A; \forall M. M \in A \implies \text{finite}\ M \rrbracket \implies \text{finite}(\bigcup A)$

**by** (*induct rule:finite-induct*) *simp-all*

**lemma** *finite-empty-induct*:

**assumes** *finite A*

**and** *P A*

**and**  $\forall a\ A. \text{finite}\ A \implies a:A \implies P\ A \implies P\ (A - \{a\})$

**shows**  $P\ \{\}$

**proof** —

**have**  $P\ (A - A)$

**proof** —

{

**fix**  $c\ b :: 'a\ \text{set}$

**assume**  $c: \text{finite}\ c$  **and**  $b: \text{finite}\ b$

**and**  $P1: P\ b$  **and**  $P2: \forall x\ y. \text{finite}\ y \implies x \in y \implies P\ y \implies P\ (y - \{x\})$

**have**  $c \subseteq b \implies P\ (b - c)$

**using**  $c$

**proof** *induct*

**case** *empty*

**from**  $P1$  **show** ?*case* **by** *simp*

**next**

**case** (*insert x F*)

**have**  $P\ (b - F - \{x\})$

**proof** (*rule P2*)

**from** -  $b$  **show**  $\text{finite}\ (b - F)$  **by** (*rule finite-subset*) *blast*

```

      from insert show  $x \in b - F$  by simp
      from insert show  $P (b - F)$  by simp
    qed
    also have  $b - F - \{x\} = b - \text{insert } x F$  by (rule Diff-insert [symmetric])
    finally show ?case .
  qed
}
then show ?thesis by this (simp-all add: assms)
qed
then show ?thesis by simp
qed

```

**lemma** *finite-Diff* [simp]:  $\text{finite } A \implies \text{finite } (A - B)$   
**by** (rule Diff-subset [THEN finite-subset])

**lemma** *finite-Diff2* [simp]:  
 assumes  $\text{finite } B$  shows  $\text{finite } (A - B) = \text{finite } A$   
**proof** –  
 have  $\text{finite } A \iff \text{finite}((A-B) \cup (A \cap B))$  by (simp add: Un-Diff-Int)  
 also have  $\dots \iff \text{finite}(A-B)$  using  $\langle \text{finite } B \rangle$  by (simp)  
 finally show ?thesis ..  
**qed**

**lemma** *finite-compl*[simp]:  
 $\text{finite}(A::'a \text{ set}) \implies \text{finite}(-A) = \text{finite}(\text{UNIV}::'a \text{ set})$   
**by**(simp add: Compl-eq-Diff-UNIV)

**lemma** *finite-Collect-not*[simp]:  
 $\text{finite}\{x::'a. P\ x\} \implies \text{finite}\{x. \sim P\ x\} = \text{finite}(\text{UNIV}::'a \text{ set})$   
**by**(simp add: Collect-neg-eq)

**lemma** *finite-Diff-insert* [iff]:  $\text{finite } (A - \text{insert } a B) = \text{finite } (A - B)$   
**apply** (subst Diff-insert)  
**apply** (case-tac  $a : A - B$ )  
**apply** (rule finite-insert [symmetric, THEN trans])  
**apply** (subst insert-Diff, simp-all)  
**done**

Image and Inverse Image over Finite Sets

**lemma** *finite-imageI*[simp]:  $\text{finite } F \implies \text{finite } (h \cdot F)$   
 — The image of a finite set is finite.  
**by** (induct set: finite) simp-all

**lemma** *finite-surj*:  $\text{finite } A \implies B \leq f \cdot A \implies \text{finite } B$   
**apply** (frule finite-imageI)  
**apply** (erule finite-subset, assumption)  
**done**

**lemma** *finite-range-imageI*:

```

    finite (range g) ==> finite (range (%x. f (g x)))
  apply (drule finite-imageI, simp add: range-composition)
done

```

**lemma** *finite-imageD*:  $\text{finite } (f^{\circ} A) \implies \text{inj-on } f \ A \implies \text{finite } A$

**proof** –

```

  have aux: !!A. finite (A - {}) = finite A by simp
  fix B :: 'a set
  assume finite B
  thus !!A. f^{\circ} A = B ==> inj-on f A ==> finite A
    apply induct
    apply simp
    apply (subgoal-tac EX y:A. f y = x & F = f^{\circ} (A - {y}))
    apply clarify
    apply (simp (no-asm-use) add: inj-on-def)
    apply (blast dest!: aux [THEN iffD1], atomize)
    apply (erule-tac V = ALL A. ?PP (A) in thin-rl)
    apply (frule subsetD [OF equalityD2 insertI1], clarify)
    apply (rule-tac x = xa in bexI)
    apply (simp-all add: inj-on-image-set-diff)
  done
qed (rule refl)

```

**lemma** *inj-vimage-singleton*:  $\text{inj } f \implies f^{-1}\{a\} \subseteq \{\text{THE } x. f \ x = a\}$

— The inverse image of a singleton under an injective function is included in a singleton.

```

  apply (auto simp add: inj-on-def)
  apply (blast intro: the-equality [symmetric])
done

```

**lemma** *finite-vimageI*:  $[[\text{finite } F; \text{inj } h]] \implies \text{finite } (h^{-1} F)$

— The inverse image of a finite set under an injective function is finite.

```

  apply (induct set: finite)
  apply simp-all
  apply (subst vimage-insert)
  apply (simp add: finite-Un finite-subset [OF inj-vimage-singleton])
done

```

The finite UNION of finite sets

**lemma** *finite-UN-I*:  $\text{finite } A \implies (!!a. a:A \implies \text{finite } (B \ a)) \implies \text{finite } (\text{UN } a:A. B \ a)$

**by** (induct set: finite) simp-all

Strengthen RHS to  $(\forall x \in A. \text{finite } (B \ x)) \wedge \text{finite } \{x \in A. B \ x \neq \{\}\}$ ?

We’d need to prove  $\text{finite } C \implies \forall A \ B. \text{UNION } A \ B \subseteq C \longrightarrow \text{finite } \{x \in A. B \ x \neq \{\}\}$  by induction.

**lemma** *finite-UN* [simp]:

$\text{finite } A \implies \text{finite } (\text{UNION } A \ B) = (\text{ALL } x:A. \text{finite } (B \ x))$



**by** (*blast intro: finite-UN-I finite-subset*)

**lemma** *finite-Collect-bex*[*simp*]: *finite A*  $\implies$   
 $\text{finite}\{x. \text{EX } y:A. Q \ x \ y\} = (\text{ALL } y:A. \text{finite}\{x. Q \ x \ y\})$   
**apply**(*subgoal-tac*  $\{x. \text{EX } y:A. Q \ x \ y\} = \text{UNION } A \ (\%y. \{x. Q \ x \ y\})$ )  
**apply** *auto*  
**done**

**lemma** *finite-Collect-bounded-ex*[*simp*]: *finite* $\{y. P \ y\} \implies$   
 $\text{finite}\{x. \text{EX } y. P \ y \ \& \ Q \ x \ y\} = (\text{ALL } y. P \ y \longrightarrow \text{finite}\{x. Q \ x \ y\})$   
**apply**(*subgoal-tac*  $\{x. \text{EX } y. P \ y \ \& \ Q \ x \ y\} = \text{UNION } \{y. P \ y\} \ (\%y. \{x. Q \ x \ y\})$ )  
**apply** *auto*  
**done**

**lemma** *finite-Plus*:  $[\text{finite } A; \text{finite } B] \implies \text{finite } (A <+> B)$   
**by** (*simp add: Plus-def*)

Sigma of finite sets

**lemma** *finite-SigmaI* [*simp*]:  
 $\text{finite } A \implies (!a. a:A \implies \text{finite } (B \ a)) \implies \text{finite } (\text{SIGMA } a:A. B \ a)$   
**by** (*unfold Sigma-def*) (*blast intro!: finite-UN-I*)

**lemma** *finite-cartesian-product*:  $[\text{finite } A; \text{finite } B] \implies$   
 $\text{finite } (A <*> B)$   
**by** (*rule finite-SigmaI*)

**lemma** *finite-Prod-UNIV*:  
 $\text{finite } (\text{UNIV}::'a \ \text{set}) \implies \text{finite } (\text{UNIV}::'b \ \text{set}) \implies \text{finite } (\text{UNIV}::('a * 'b) \ \text{set})$   
**apply** (*subgoal-tac*  $(\text{UNIV}::('a * 'b) \ \text{set}) = \text{Sigma } \text{UNIV } (\%x. \text{UNIV})$ )  
**apply** (*erule ssubst*)  
**apply** (*erule finite-SigmaI, auto*)  
**done**

**lemma** *finite-cartesian-productD1*:  
 $[\text{finite } (A <*> B); B \neq \{\}] \implies \text{finite } A$   
**apply** (*auto simp add: finite-conv-nat-seg-image*)  
**apply** (*drule-tac*  $x=n$  **in** *spec*)  
**apply** (*drule-tac*  $x=\text{fst } o \ f$  **in** *spec*)  
**apply** (*auto simp add: o-def*)  
**prefer** 2 **apply** (*force dest!: equalityD2*)  
**apply** (*drule equalityD1*)  
**apply** (*rename-tac*  $y \ x$ )  
**apply** (*subgoal-tac*  $\exists k. k < n \ \& \ f \ k = (x,y)$ )  
**prefer** 2 **apply** *force*  
**apply** *clarify*  
**apply** (*rule-tac*  $x=k$  **in** *image-eqI, auto*)  
**done**

```

lemma finite-cartesian-productD2:
  [| finite ( $A <*> B$ );  $A \neq \{\}$  |] ==> finite B
apply (auto simp add: finite-conv-nat-seg-image)
apply (drule-tac x=n in spec)
apply (drule-tac x=snd o f in spec)
apply (auto simp add: o-def)
prefer 2 apply (force dest!: equalityD2)
apply (drule equalityD1)
apply (rename-tac x y)
apply (subgoal-tac  $\exists k. k < n \ \& \ f \ k = (x,y)$ )
prefer 2 apply force
apply clarify
apply (rule-tac x=k in image-eqI, auto)
done

```

The powerset of a finite set

```

lemma finite-Pow-iff [iff]: finite (Pow A) = finite A
proof
  assume finite (Pow A)
  with - have finite ((%x. {x}) ‘ A) by (rule finite-subset blast)
  thus finite A by (rule finite-imageD [unfolded inj-on-def] simp)
next
  assume finite A
  thus finite (Pow A)
  by induct (simp-all add: finite-UnI finite-imageI Pow-insert)
qed

```

```

lemma finite-Collect-subsets[simp,intro]: finite A ==> finite {B. B  $\subseteq$  A}
by(simp add: Pow-def[symmetric])

```

```

lemma finite-UnionD: finite( $\bigcup A$ ) ==> finite A
by(blast intro: finite-subset[OF subset-Pow-Union])

```

## 17.2 Class *finite*

```

setup  $\ll$  Sign.add-path finite  $\gg$  — FIXME: name tweaking
class finite =
  assumes finite-UNIV: finite (UNIV :: 'a set)
setup  $\ll$  Sign.parent-path  $\gg$ 
hide const finite

```

```

context finite
begin

```

```

lemma finite [simp]: finite (A :: 'a set)
  by (rule subset-UNIV finite-UNIV finite-subset)+

```

**end**

**lemma** *UNIV-unit* [noatp]:  
 $UNIV = \{()\}$  **by** *auto*

**instance** *unit* :: *finite*  
**by** *default* (*simp add: UNIV-unit*)

**lemma** *UNIV-bool* [noatp]:  
 $UNIV = \{False, True\}$  **by** *auto*

**instance** *bool* :: *finite*  
**by** *default* (*simp add: UNIV-bool*)

**instance**  $*$  :: (*finite*, *finite*) *finite*  
**by** *default* (*simp only: UNIV-Times-UNIV [symmetric] finite-cartesian-product finite*)

**lemma** *inj-graph*: *inj* ( $\%f. \{(x, y). y = f\ x\}$ )  
**by** (*rule inj-onI, auto simp add: expand-set-eq expand-fun-eq*)

**instance** *fun* :: (*finite*, *finite*) *finite*  
**proof**  
  **show** *finite* (*UNIV* :: ('a => 'b) *set*)  
  **proof** (*rule finite-imageD*)  
    **let**  $?graph = \%f::'a \Rightarrow 'b. \{(x, y). y = f\ x\}$   
    **have**  $range\ ?graph \subseteq Pow\ UNIV$  **by** *simp*  
    **moreover have** *finite* ( $Pow\ (UNIV :: ('a * 'b)\ set)$ )  
      **by** (*simp only: finite-Pow-iff finite*)  
    **ultimately show** *finite* ( $range\ ?graph$ )  
      **by** (*rule finite-subset*)  
    **show** *inj*  $?graph$  **by** (*rule inj-graph*)  
  **qed**  
**qed**

**instance**  $+$  :: (*finite*, *finite*) *finite*  
**by** *default* (*simp only: UNIV-Plus-UNIV [symmetric] finite-Plus finite*)

### 17.3 A fold functional for finite sets

The intended behaviour is  $fold\ f\ z\ \{x_1, \dots, x_n\} = f\ x_1\ (\dots (f\ x_n\ z)\dots)$  if  $f$  is “left-commutative”:

**locale** *fun-left-comm* =  
  **fixes**  $f :: 'a \Rightarrow 'b \Rightarrow 'b$   
  **assumes** *fun-left-comm*:  $f\ x\ (f\ y\ z) = f\ y\ (f\ x\ z)$   
**begin**

On a functional level it looks much nicer:

**lemma** *fun-comp-comm*:  $f\ x \circ f\ y = f\ y \circ f\ x$

by (simp add: fun-left-comm expand-fun-eq)

end

**inductive** fold-graph :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  'b  $\Rightarrow$  'a set  $\Rightarrow$  'b  $\Rightarrow$  bool  
**for** f :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'b **and** z :: 'b **where**  
 emptyI [intro]: fold-graph f z {} z |  
 insertI [intro]:  $x \notin A \implies \text{fold-graph } f \ z \ A \ y$   
 $\implies \text{fold-graph } f \ z \ (\text{insert } x \ A) \ (f \ x \ y)$

**inductive-cases** empty-fold-graphE [elim!]: fold-graph f z {} x

**definition** fold :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  'b  $\Rightarrow$  'a set  $\Rightarrow$  'b **where**  
 [code del]: fold f z A = (THE y. fold-graph f z A y)

A tempting alternative for the definiens is *if finite A then THE y. fold-graph f z A y else e*. It allows the removal of finiteness assumptions from the theorems *fold-comm*, *fold-reindex* and *fold-distrib*. The proofs become ugly. It is not worth the effort. (???)

**lemma** Diff1-fold-graph:

fold-graph f z (A - {x}) y  $\implies x \in A \implies \text{fold-graph } f \ z \ A \ (f \ x \ y)$   
**by** (erule insert-Diff [THEN subst], rule fold-graph.intros, auto)

**lemma** fold-graph-imp-finite: fold-graph f z A x  $\implies$  finite A

**by** (induct set: fold-graph) auto

**lemma** finite-imp-fold-graph: finite A  $\implies \exists x. \text{fold-graph } f \ z \ A \ x$

**by** (induct set: finite) auto

### 17.3.1 From fold-graph to fold

**lemma** image-less-Suc:  $h \text{ ' } \{i. i < \text{Suc } m\} = \text{insert } (h \ m) \ (h \text{ ' } \{i. i < m\})$   
**by** (auto simp add: less-Suc-eq)

**lemma** insert-image-inj-on-eq:

$[[\text{insert } (h \ m) \ A = h \text{ ' } \{i. i < \text{Suc } m\}; h \ m \notin A;$   
 $\text{inj-on } h \ \{i. i < \text{Suc } m\}]]$   
 $\implies A = h \text{ ' } \{i. i < m\}$

**apply** (auto simp add: image-less-Suc inj-on-def)

**apply** (blast intro: less-trans)

**done**

**lemma** insert-inj-onE:

**assumes** aA:  $\text{insert } a \ A = h \text{ ' } \{i::\text{nat}. i < n\}$  **and** anot:  $a \notin A$

**and** inj-on:  $\text{inj-on } h \ \{i::\text{nat}. i < n\}$

**shows**  $\exists hm \ m. \text{inj-on } hm \ \{i::\text{nat}. i < m\} \ \& \ A = hm \text{ ' } \{i. i < m\} \ \& \ m < n$

**proof** (cases n)

**case 0** **thus** ?thesis **using** aA **by** auto

**next**

```

case (Suc m)
have nSuc: n = Suc m by fact
have mlessn: m < n by (simp add: nSuc)
from aA obtain k where hkeq: h k = a and klessn: k < n by (blast elim!:
equalityE)
let ?hm = Fun.swap k m h
have inj-hm: inj-on ?hm {i. i < n} using klessn mlessn
  by (simp add: inj-on-swap-iff inj-on)
show ?thesis
proof (intro exI conjI)
  show inj-on ?hm {i. i < m} using inj-hm
    by (auto simp add: nSuc less-Suc-eq intro: subset-inj-on)
  show m < n by (rule mlessn)
  show A = ?hm ‘ {i. i < m}
  proof (rule insert-image-inj-on-eq)
    show inj-on (Fun.swap k m h) {i. i < Suc m} using inj-hm nSuc by simp
    show ?hm m ∉ A by (simp add: swap-def hkeq anot)
    show insert (?hm m) A = ?hm ‘ {i. i < Suc m}
      using aA hkeq nSuc klessn
      by (auto simp add: swap-def image-less-Suc fun-upd-image
        less-Suc-eq inj-on-image-set-diff [OF inj-on])
  qed
qed
qed
qed

context fun-left-comm
begin

lemma fold-graph-determ-aux:
  A = h ‘ {i::nat. i < n} ⟹ inj-on h {i. i < n}
  ⟹ fold-graph f z A x ⟹ fold-graph f z A x'
  ⟹ x' = x
proof (induct n arbitrary: A x x' h rule: less-induct)
  case (less n)
  have IH: ⋀m h A x x'. m < n ⟹ A = h ‘ {i. i < m}
    ⟹ inj-on h {i. i < m} ⟹ fold-graph f z A x
    ⟹ fold-graph f z A x' ⟹ x' = x by fact
  have Afoldx: fold-graph f z A x and Afoldx': fold-graph f z A x'
    and A: A = h ‘ {i. i < n} and injh: inj-on h {i. i < n} by fact+
  show ?case
  proof (rule fold-graph.cases [OF Afoldx])
    assume A = {} and x = z
    with Afoldx' show x' = x by auto
  next
  fix B b u
  assume AbB: A = insert b B and x: x = f b u
    and notinB: b ∉ B and Bu: fold-graph f z B u
  show x' = x
  proof (rule fold-graph.cases [OF Afoldx'])

```

```

    assume  $A = \{\}$  and  $x' = z$ 
    with  $AbB$  show  $x' = x$  by blast
next
fix  $C\ c\ v$ 
assume  $AcC$ :  $A = \text{insert } c\ C$  and  $x'$ :  $x' = f\ c\ v$ 
  and  $\text{notin}C$ :  $c \notin C$  and  $Cv$ :  $\text{fold-graph } f\ z\ C\ v$ 
from  $A\ AbB$  have  $Beq$ :  $\text{insert } b\ B = h'\{i. i < n\}$  by simp
from  $\text{insert-inj-onE}$  [OF  $Beq\ \text{notin}B\ \text{inh}$ ]
obtain  $hB\ mB$  where  $\text{inj-on}B$ :  $\text{inj-on } hB\ \{i. i < mB\}$ 
  and  $Beq$ :  $B = hB\ '\{i. i < mB\}$  and  $\text{less}B$ :  $mB < n$  by auto
from  $A\ AcC$  have  $Ceq$ :  $\text{insert } c\ C = h'\{i. i < n\}$  by simp
from  $\text{insert-inj-onE}$  [OF  $Ceq\ \text{notin}C\ \text{inh}$ ]
obtain  $hC\ mC$  where  $\text{inj-on}C$ :  $\text{inj-on } hC\ \{i. i < mC\}$ 
  and  $Ceq$ :  $C = hC\ '\{i. i < mC\}$  and  $\text{less}C$ :  $mC < n$  by auto
show  $x' = x$ 
proof cases
  assume  $b = c$ 
  then moreover have  $B = C$  using  $AbB\ AcC\ \text{notin}B\ \text{notin}C$  by auto
  ultimately show ?thesis using  $Bu\ Cv\ x\ x'\ IH$  [OF  $\text{less}C\ Ceq\ \text{inj-on}C$ ]
    by auto
next
assume  $\text{diff}$ :  $b \neq c$ 
let  $?D = B - \{c\}$ 
have  $B$ :  $B = \text{insert } c\ ?D$  and  $C$ :  $C = \text{insert } b\ ?D$ 
  using  $AbB\ AcC\ \text{notin}B\ \text{notin}C\ \text{diff}$  by (blast elim!:equalityE)+
have  $\text{finite } A$  by (rule fold-graph-imp-finite [OF  $A\text{fold}x$ ])
with  $AbB$  have  $\text{finite } ?D$  by simp
then obtain  $d$  where  $D\text{foldd}$ :  $\text{fold-graph } f\ z\ ?D\ d$ 
  using  $\text{finite-imp-fold-graph}$  by iprover
moreover have  $\text{cin}B$ :  $c \in B$  using  $B$  by auto
ultimately have  $\text{fold-graph } f\ z\ B\ (f\ c\ d)$  by (rule Diff1-fold-graph)
hence  $f\ c\ d = u$  by (rule IH [OF  $\text{less}B\ Beq\ \text{inj-on}B\ Bu$ ])
moreover have  $f\ b\ d = v$ 
proof (rule IH [OF  $\text{less}C\ Ceq\ \text{inj-on}C\ Cv$ ])
  show  $\text{fold-graph } f\ z\ C\ (f\ b\ d)$  using  $C\ \text{notin}B\ D\text{foldd}$  by fastsimp
qed
ultimately show ?thesis
  using  $\text{fun-left-comm}$  [of  $c\ b$ ]  $x\ x'$  by (auto simp add: o-def)
qed
qed
qed
qed

lemma fold-graph-determ:
  fold-graph  $f\ z\ A\ x \implies \text{fold-graph } f\ z\ A\ y \implies y = x$ 
apply (frule fold-graph-imp-finite [THEN finite-imp-nat-seg-image-inj-on])
apply (blast intro: fold-graph-determ-aux [rule-format])
done

```

**lemma** *fold-equality*:

$fold-graph\ f\ z\ A\ y \implies fold\ f\ z\ A = y$   
**by** (*unfold fold-def*) (*blast intro: fold-graph-determ*)

The base case for *fold*:

**lemma** (*in -*) *fold-empty* [*simp*]:  $fold\ f\ z\ \{\} = z$   
**by** (*unfold fold-def*) *blast*

The various recursion equations for *fold*:

**lemma** *fold-insert-aux*:  $x \notin A$   
 $\implies fold-graph\ f\ z\ (insert\ x\ A)\ v \longleftrightarrow$   
 $(\exists y. fold-graph\ f\ z\ A\ y \wedge v = f\ x\ y)$   
**apply** *auto*  
**apply** (*rule-tac A1 = A and f1 = f in finite-imp-fold-graph [THEN exE]*)  
**apply** (*fastsimp dest: fold-graph-imp-finite*)  
**apply** (*blast intro: fold-graph-determ*)  
**done**

**lemma** *fold-insert* [*simp*]:  
 $finite\ A \implies x \notin A \implies fold\ f\ z\ (insert\ x\ A) = f\ x\ (fold\ f\ z\ A)$   
**apply** (*simp add: fold-def fold-insert-aux*)  
**apply** (*rule the-equality*)  
**apply** (*auto intro: finite-imp-fold-graph*  
*cong add: conj-cong simp add: fold-def[symmetric] fold-equality*)  
**done**

**lemma** *fold-fun-comm*:  
 $finite\ A \implies f\ x\ (fold\ f\ z\ A) = fold\ f\ (f\ x\ z)\ A$   
**proof** (*induct rule: finite-induct*)  
**case empty then show ?case by simp**  
**next**  
**case (insert y A) then show ?case**  
**by (simp add: fun-left-comm[of x])**  
**qed**

**lemma** *fold-insert2*:  
 $finite\ A \implies x \notin A \implies fold\ f\ z\ (insert\ x\ A) = fold\ f\ (f\ x\ z)\ A$   
**by** (*simp add: fold-insert fold-fun-comm*)

**lemma** *fold-rec*:  
**assumes** *finite A and x ∈ A*  
**shows**  $fold\ f\ z\ A = f\ x\ (fold\ f\ z\ (A - \{x\}))$   
**proof** -  
**have**  $A: A = insert\ x\ (A - \{x\})$  **using**  $\langle x \in A \rangle$  **by** *blast*  
**then have**  $fold\ f\ z\ A = fold\ f\ z\ (insert\ x\ (A - \{x\}))$  **by** *simp*  
**also have**  $\dots = f\ x\ (fold\ f\ z\ (A - \{x\}))$   
**by** (*rule fold-insert*) (*simp add: (finite A)+*)  
**finally show ?thesis .**  
**qed**

```

lemma fold-insert-remove:
  assumes finite A
  shows  $\text{fold } f \ z \ (\text{insert } x \ A) = f \ x \ (\text{fold } f \ z \ (A - \{x\}))$ 
proof –
  from  $\langle \text{finite } A \rangle$  have finite (insert x A) by auto
  moreover have  $x \in \text{insert } x \ A$  by auto
  ultimately have  $\text{fold } f \ z \ (\text{insert } x \ A) = f \ x \ (\text{fold } f \ z \ (\text{insert } x \ A - \{x\}))$ 
    by (rule fold-rec)
  then show ?thesis by simp
qed

end

```

A simplified version for idempotent functions:

```

locale fun-left-comm-idem = fun-left-comm +
  assumes fun-left-idem:  $f \ x \ (f \ x \ z) = f \ x \ z$ 
begin

```

The nice version:

```

lemma fun-comp-idem :  $f \ x \ o \ f \ x = f \ x$ 
by (simp add: fun-left-idem expand-fun-eq)

lemma fold-insert-idem:
  assumes fin: finite A
  shows  $\text{fold } f \ z \ (\text{insert } x \ A) = f \ x \ (\text{fold } f \ z \ A)$ 
proof cases
  assume  $x \in A$ 
  then obtain B where  $A = \text{insert } x \ B$  and  $x \notin B$  by (rule set-insert)
  then show ?thesis using assms by (simp add: fun-left-idem)
next
  assume  $x \notin A$  then show ?thesis using assms by simp
qed

declare fold-insert[simp del] fold-insert-idem[simp]

lemma fold-insert-idem2:
   $\text{finite } A \implies \text{fold } f \ z \ (\text{insert } x \ A) = \text{fold } f \ (f \ x \ z) \ A$ 
by(simp add: fold-fun-comm)

end

```

### 17.3.2 The derived combinator *fold-image*

```

definition fold-image ::  $('b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a \text{ set} \Rightarrow 'b$ 
where  $\text{fold-image } f \ g = \text{fold } (\%x \ y. f \ (g \ x) \ y)$ 

```

```

lemma fold-image-empty[simp]:  $\text{fold-image } f \ g \ z \ \{\} = z$ 
by(simp add: fold-image-def)

```



**context** *ab-semigroup-mult*  
**begin**

**lemma** *fold-image-insert[simp]*:  
**assumes** *finite A* **and**  $a \notin A$   
**shows** *fold-image times g z (insert a A) = g a \* (fold-image times g z A)*  
**proof** –  
   **interpret** *I: fun-left-comm*  $\%x\ y. (g\ x) * y$   
   **by** *unfold-locales (simp add: mult-ac)*  
   **show** *?thesis* **using** *assms* **by** *(simp add: fold-image-def I.fold-insert)*  
**qed**

**lemma** *fold-image-reindex*:  
**assumes** *fin: finite A*  
**shows** *inj-on h A  $\implies$  fold-image times g z (h `A) = fold-image times (g o h) z A*  
**using** *fin* **apply** *induct*  
   **apply** *simp*  
   **apply** *simp*  
**done**

**lemma** *fold-image-cong*:  
   *finite A  $\implies$*   
   *(!!x. x:A  $\implies$  g x = h x)  $\implies$  fold-image times g z A = fold-image times h z A*  
**apply** *(subgoal-tac ALL C. C <= A  $\longleftrightarrow$  (ALL x:C. g x = h x)  $\longleftrightarrow$  fold-image times g z C = fold-image times h z C)*  
   **apply** *simp*  
   **apply** *(erule finite-induct, simp)*  
   **apply** *(simp add: subset-insert-iff, clarify)*  
   **apply** *(subgoal-tac finite C)*  
   **prefer** 2 **apply** *(blast dest: finite-subset [COMP swap-prems-rl])*  
   **apply** *(subgoal-tac C = insert x (C - {x}))*  
   **prefer** 2 **apply** *blast*  
   **apply** *(erule ssubst)*  
   **apply** *(erule spec)*  
   **apply** *(erule (1) notE impE)*  
   **apply** *(simp add: Ball-def del: insert-Diff-single)*  
**done**

**end**

**context** *comm-monoid-mult*  
**begin**

**lemma** *fold-image-Un-Int*:

```

finite A ==> finite B ==>
  fold-image times g 1 A * fold-image times g 1 B =
  fold-image times g 1 (A Un B) * fold-image times g 1 (A Int B)
by (induct set: finite)
  (auto simp add: mult-ac insert-absorb Int-insert-left)

```

**corollary** *fold-Un-disjoint:*

```

finite A ==> finite B ==> A Int B = {} ==>
  fold-image times g 1 (A Un B) =
  fold-image times g 1 A * fold-image times g 1 B
by (simp add: fold-image-Un-Int)

```

**lemma** *fold-image-UN-disjoint:*

```

[[ finite I; ALL i:I. finite (A i);
  ALL i:I. ALL j:I. i ≠ j --> A i Int A j = {} ]]
==> fold-image times g 1 (UNION I A) =
  fold-image times (%i. fold-image times g 1 (A i)) 1 I
apply (induct set: finite, simp, atomize)
apply (subgoal-tac ALL i:F. x ≠ i)
prefer 2 apply blast
apply (subgoal-tac A x Int UNION F A = {})
prefer 2 apply blast
apply (simp add: fold-Un-disjoint)
done

```

**lemma** *fold-image-Sigma:*  $\text{finite } A \implies \text{ALL } x:A. \text{finite } (B x) \implies$

```

  fold-image times (%x. fold-image times (g x) 1 (B x)) 1 A =
  fold-image times (split g) 1 (SIGMA x:A. B x)
apply (subst Sigma-def)
apply (subst fold-image-UN-disjoint, assumption, simp)
  apply blast
apply (erule fold-image-cong)
apply (subst fold-image-UN-disjoint, simp, simp)
  apply blast
apply simp
done

```

**lemma** *fold-image-distrib:*  $\text{finite } A \implies$

```

  fold-image times (%x. g x * h x) 1 A =
  fold-image times g 1 A * fold-image times h 1 A
by (erule finite-induct) (simp-all add: mult-ac)

```

**lemma** *fold-image-related:*

```

assumes Re: R e e
and Rop: ∀ x1 y1 x2 y2. R x1 x2 ∧ R y1 y2 ⟶ R (x1 * y1) (x2 * y2)
and fS: finite S and Rfg: ∀ x∈S. R (h x) (g x)
shows R (fold-image (op *) h e S) (fold-image (op *) g e S)
using fS by (rule finite-subset-induct) (insert assms, auto)

```

**lemma** *fold-image-eq-general*:

**assumes** *fS*: *finite S*

**and** *h*:  $\forall y \in S'. \exists! x. x \in S \wedge h(x) = y$

**and** *f12*:  $\forall x \in S. h\ x \in S' \wedge f2(h\ x) = f1\ x$

**shows** *fold-image* (*op* \*) *f1* *e* *S* = *fold-image* (*op* \*) *f2* *e* *S'*

**proof** –

**from** *h f12* **have** *hS*:  $h\ ` S = S'$  **by** *auto*

**{fix** *x y* **assume** *H*:  $x \in S\ y \in S\ h\ x = h\ y$

**from** *f12 h H* **have**  $x = y$  **by** *auto* **}**

**hence** *hinj*: *inj-on* *h* *S* **unfolding** *inj-on-def Ex1-def* **by** *blast*

**from** *f12* **have** *th*:  $\bigwedge x. x \in S \implies (f2 \circ h)\ x = f1\ x$  **by** *auto*

**from** *hS* **have** *fold-image* (*op* \*) *f2* *e* *S'* = *fold-image* (*op* \*) *f2* *e* (*h* ` *S*) **by**

*simp*

**also** **have**  $\dots = \text{fold-image } (op\ *)\ (f2\ o\ h)\ e\ S$

**using** *fold-image-reindex*[*OF fS hinj*, *of f2 e*].

**also** **have**  $\dots = \text{fold-image } (op\ *)\ f1\ e\ S$  **using** *th fold-image-cong*[*OF fS*, *of f2 o h f1 e*]

**by** *blast*

**finally** **show** *?thesis* ..

**qed**

**lemma** *fold-image-eq-general-inverses*:

**assumes** *fS*: *finite S*

**and** *kh*:  $\bigwedge y. y \in T \implies k\ y \in S \wedge h\ (k\ y) = y$

**and** *hk*:  $\bigwedge x. x \in S \implies h\ x \in T \wedge k\ (h\ x) = x \wedge g\ (h\ x) = f\ x$

**shows** *fold-image* (*op* \*) *f* *e* *S* = *fold-image* (*op* \*) *g* *e* *T*

**apply** (*rule fold-image-eq-general*[*OF fS*, *of T h g f e*])

**apply** (*rule ballI*)

**apply** (*frule kh*)

**apply** (*rule ex1I*[])

**apply** *blast*

**apply** *clarsimp*

**apply** (*drule hk*) **apply** *simp*

**apply** (*rule sym*)

**apply** (*erule conjunct1*[*OF conjunct2*[*OF hk*]])

**apply** (*rule ballI*)

**apply** (*drule hk*)

**apply** *blast*

**done**

**end**

## 17.4 Generalized summation over a set

**interpretation** *comm-monoid-add*: *comm-monoid-mult* *0*::'*a*::*comm-monoid-add*  
*op* +

**proof** **qed** (*auto intro: add-assoc add-commute*)

**definition**  $setsum :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow 'b::comm-monoid-add$   
**where**  $setsum\ f\ A == \text{if finite } A \text{ then fold-image } (op\ +)\ f\ 0\ A \text{ else } 0$

**abbreviation**

$Setsum\ (\sum - [1000]\ 999) \text{ where}$   
 $\sum A == setsum\ (\%x. x)\ A$

Now: lot's of fancy syntax. First,  $setsum\ (\lambda x. e)\ A$  is written  $\sum x \in A. e$ .

**syntax**

$-setsum :: pttrn \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b::comm-monoid-add\ ((\mathcal{S}SUM\ -::-.)\ [0, 51, 10]\ 10)$

**syntax** (*xsymbols*)

$-setsum :: pttrn \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b::comm-monoid-add\ ((\mathcal{S}\sum\ -\in-.)\ [0, 51, 10]\ 10)$

**syntax** (*HTML output*)

$-setsum :: pttrn \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b::comm-monoid-add\ ((\mathcal{S}\sum\ -\in-.)\ [0, 51, 10]\ 10)$

**translations** — Beware of argument permutation!

$SUM\ i:A. b == CONST\ setsum\ (\%i. b)\ A$   
 $\sum i \in A. b == CONST\ setsum\ (\%i. b)\ A$

Instead of  $\sum x \in \{x. P\}. e$  we introduce the shorter  $\sum x | P. e$ .

**syntax**

$-qsetsum :: pttrn \Rightarrow bool \Rightarrow 'a \Rightarrow 'a\ ((\mathcal{S}SUM\ -\ |/\ -/)\ [0,0,10]\ 10)$

**syntax** (*xsymbols*)

$-qsetsum :: pttrn \Rightarrow bool \Rightarrow 'a \Rightarrow 'a\ ((\mathcal{S}\sum\ -\ |(-)/)\ [0,0,10]\ 10)$

**syntax** (*HTML output*)

$-qsetsum :: pttrn \Rightarrow bool \Rightarrow 'a \Rightarrow 'a\ ((\mathcal{S}\sum\ -\ |(-)/)\ [0,0,10]\ 10)$

**translations**

$SUM\ x|P. t \Rightarrow CONST\ setsum\ (\%x. t)\ \{x. P\}$   
 $\sum x | P. t \Rightarrow CONST\ setsum\ (\%x. t)\ \{x. P\}$

**print-translation**  $\ll$

*let*

$fun\ setsum-tr'\ [Abs(x, Tx, t), Const\ (Collect, -)\ \$\ Abs(y, Ty, P)] =$

$\text{if } x <> y \text{ then raise Match}$

$\text{else let val } x' = Syntax.mark-bound\ x$

$\text{val } t' = subst-bound(x', t)$

$\text{val } P' = subst-bound(x', P)$

$\text{in } Syntax.const\ -qsetsum\ \$\ Syntax.mark-bound\ x\ \$\ P'\ \$\ t' \text{ end}$

$\text{in } [(setsum, setsum-tr')]\ \text{end}$

$\gg$

**lemma**  $setsum-empty\ [simp]: setsum\ f\ \{\} = 0$

**by** ( $simp\ add: setsum-def$ )

**lemma** *setsum-insert* [simp]:

*finite F ==> a ∉ F ==> setsum f (insert a F) = f a + setsum f F*  
**by** (simp add: setsum-def)

**lemma** *setsum-infinitesimal* [simp]:  $\sim \text{finite } A \implies \text{setsum } f \ A = 0$

**by** (simp add: setsum-def)

**lemma** *setsum-reindex*:

*inj-on f B ==> setsum h (f ` B) = setsum (h ∘ f) B*  
**by**(auto simp add: setsum-def comm-monoid-add.fold-image-reindex dest!:finite-imageD)

**lemma** *setsum-reindex-id*:

*inj-on f B ==> setsum f B = setsum id (f ` B)*  
**by** (auto simp add: setsum-reindex)

**lemma** *setsum-reindex-nonzero*:

**assumes** *fS*: *finite S*  
**and** *nz*:  $\bigwedge x y. x \in S \implies y \in S \implies x \neq y \implies f x = f y \implies h (f x) = 0$   
**shows** *setsum h (f ` S) = setsum (h ∘ f) S*

**using** *nz*

**proof**(*induct rule: finite-induct[OF fS]*)

**case 1 thus** ?*case* **by** *simp*

**next**

**case** (2 *x F*)

{**assume** *fxF*: *f x ∈ f ` F* **hence**  $\exists y \in F. f y = f x$  **by** *auto*  
**then obtain** *y* **where** *y*: *y ∈ F f x = f y* **by** *auto*  
**from** 2.*hyps y* **have** *xy*: *x ≠ y* **by** *auto*

**from** 2.*prems*[*of x y*] 2.*hyps xy y* **have** *h0*: *h (f x) = 0* **by** *simp*  
**have** *setsum h (f ` insert x F) = setsum h (f ` F)* **using** *fxF* **by** *auto*  
**also have** ... = *setsum (h ∘ f) (insert x F)*  
**unfolding** *setsum-insert*[*OF*  $\langle \text{finite } F \rangle \langle x \notin F \rangle$ ]  
**using** *h0*  
**apply** *simp*  
**apply** (*rule* 2.*hyps*(3))  
**apply** (*rule-tac y=y in* 2.*prems*)  
**apply** *simp-all*  
**done**

**finally have** ?*case* .}

**moreover**

{**assume** *fxF*: *f x ∉ f ` F*  
**have** *setsum h (f ` insert x F) = h (f x) + setsum h (f ` F)*  
**using** *fxF* 2.*hyps* **by** *simp*  
**also have** ... = *setsum (h ∘ f) (insert x F)*  
**unfolding** *setsum-insert*[*OF*  $\langle \text{finite } F \rangle \langle x \notin F \rangle$ ]  
**apply** *simp*  
**apply** (*rule cong*[*OF refl*[*of op + (h (f x))*]]])  
**apply** (*rule* 2.*hyps*(3))  
**apply** (*rule-tac y=y in* 2.*prems*)

```

    apply simp-all
  done
  finally have ?case .}
  ultimately show ?case by blast
qed

```

**lemma** *setsum-cong*:

$A = B \implies (\!\! \lambda x. x:B \implies f\ x = g\ x) \implies \text{setsum } f\ A = \text{setsum } g\ B$   
**by** (*fastsimp simp: setsum-def intro: comm-monoid-add.fold-image-cong*)

**lemma** *strong-setsum-cong*[*cong*]:

$A = B \implies (\!\! \lambda x. x:B \implies f\ x = g\ x) \implies \text{setsum } (\!\! \lambda x. f\ x)\ A = \text{setsum } (\!\! \lambda x. g\ x)\ B$   
**by** (*fastsimp simp: simp-implies-def setsum-def intro: comm-monoid-add.fold-image-cong*)

**lemma** *setsum-cong2*:  $\llbracket \bigwedge x. x \in A \implies f\ x = g\ x \rrbracket \implies \text{setsum } f\ A = \text{setsum } g\ A$   
**by** (*rule setsum-cong[OF refl], auto*)

**lemma** *setsum-reindex-cong*:

$\llbracket \text{inj-on } f\ A; B = f\ ` A; \!\! \lambda a. a:A \implies g\ a = h\ (f\ a) \rrbracket \implies \text{setsum } h\ B = \text{setsum } g\ A$   
**by** (*simp add: setsum-reindex cong: setsum-cong*)

**lemma** *setsum-0*[*simp*]:  $\text{setsum } (\!\! \lambda i. 0)\ A = 0$   
**apply** (*clarsimp simp: setsum-def*)  
**apply** (*erule finite-induct, auto*)  
**done**

**lemma** *setsum-0'*:  $\text{ALL } a:A. f\ a = 0 \implies \text{setsum } f\ A = 0$   
**by** (*simp add: setsum-cong*)

**lemma** *setsum-Un-Int*:  $\text{finite } A \implies \text{finite } B \implies \text{setsum } g\ (A \cup B) + \text{setsum } g\ (A \cap B) = \text{setsum } g\ A + \text{setsum } g\ B$   
 — The reversed orientation looks more natural, but LOOPS as a simp rule!  
**by** (*simp add: setsum-def comm-monoid-add.fold-image-Un-Int [symmetric]*)

**lemma** *setsum-Un-disjoint*:  $\text{finite } A \implies \text{finite } B \implies A \cap B = \{\} \implies \text{setsum } g\ (A \cup B) = \text{setsum } g\ A + \text{setsum } g\ B$   
**by** (*subst setsum-Un-Int [symmetric], auto*)

**lemma** *setsum-mono-zero-left*:

**assumes** *fT*:  $\text{finite } T$  **and** *ST*:  $S \subseteq T$   
**and** *z*:  $\forall i \in T - S. f\ i = 0$   
**shows**  $\text{setsum } f\ S = \text{setsum } f\ T$   
**proof** —  
**have** *eq*:  $T = S \cup (T - S)$  **using** *ST* **by** *blast*  
**have** *d*:  $S \cap (T - S) = \{\}$  **using** *ST* **by** *blast*  
**from** *fT ST* **have** *f*:  $\text{finite } S \text{ finite } (T - S)$  **by** (*auto intro: finite-subset*)

**show** *?thesis*  
**by** (*simp add: setsum-Un-disjoint*[*OF f d, unfolded eq[symmetric]*] *setsum-0'*[*OF z*])  
**qed**

**lemma** *setsum-mono-zero-right*:

*finite T  $\implies S \subseteq T \implies \forall i \in T - S. f i = 0 \implies \text{setsum } f T = \text{setsum } f S$*   
**by**(*blast intro!: setsum-mono-zero-left[symmetric]*)

**lemma** *setsum-mono-zero-cong-left*:

**assumes** *fT: finite T and ST: S  $\subseteq$  T*  
**and** *z:  $\forall i \in T - S. g i = 0$*   
**and** *fg:  $\bigwedge x. x \in S \implies f x = g x$*   
**shows** *setsum f S = setsum g T*

**proof**–

**have** *eq: T = S  $\cup$  (T - S) using ST by blast*  
**have** *d: S  $\cap$  (T - S) = {} using ST by blast*  
**from** *fT ST have f: finite S finite (T - S) by (auto intro: finite-subset)*  
**show** *?thesis*  
**using** *fg by (simp add: setsum-Un-disjoint[OF f d, unfolded eq[symmetric]]*  
*setsum-0'[OF z])*  
**qed**

**lemma** *setsum-mono-zero-cong-right*:

**assumes** *fT: finite T and ST: S  $\subseteq$  T*  
**and** *z:  $\forall i \in T - S. f i = 0$*   
**and** *fg:  $\bigwedge x. x \in S \implies f x = g x$*   
**shows** *setsum f T = setsum g S*  
**using** *setsum-mono-zero-cong-left[OF fT ST z] fg[symmetric] by auto*

**lemma** *setsum-delta*:

**assumes** *fS: finite S*  
**shows** *setsum ( $\lambda k. \text{if } k=a \text{ then } b \text{ } k \text{ else } 0$ ) S = (if } a \in S \text{ then } b \text{ } a \text{ else } 0)*

**proof**–

**let** *?f = ( $\lambda k. \text{if } k=a \text{ then } b \text{ } k \text{ else } 0$ )*  
**{assume** *a: a  $\notin$  S*  
**hence**  $\forall k \in S. ?f k = 0$  **by** *simp*  
**hence** *?thesis using a by simp}*  
**moreover**  
**{assume** *a: a  $\in$  S*  
**let** *?A = S - {a}*  
**let** *?B = {a}*  
**have** *eq: S = ?A  $\cup$  ?B using a by blast*  
**have** *dj: ?A  $\cap$  ?B = {} by simp*  
**from** *fS have fAB: finite ?A finite ?B by auto*  
**have** *setsum ?f S = setsum ?f ?A + setsum ?f ?B*  
**using** *setsum-Un-disjoint[OF fAB dj, of ?f, unfolded eq[symmetric]]*  
**by** *simp*  
**then have** *?thesis using a by simp}*

**ultimately show** *?thesis* **by** *blast*

**qed**

**lemma** *setsum-delta'*:

**assumes** *fS*: *finite S* **shows**

*setsum* ( $\lambda k. \text{if } a = k \text{ then } b \ k \text{ else } 0$ ) *S* =

(*if* *a* ∈ *S* *then* *b a* *else* 0)

**using** *setsum-delta*[*OF fS*, *of a b*, *symmetric*]

**by** (*auto intro: setsum-cong*)

**lemma** *setsum-restrict-set*:

**assumes** *fA*: *finite A*

**shows** *setsum f* (*A* ∩ *B*) = *setsum* ( $\lambda x. \text{if } x \in B \text{ then } f \ x \text{ else } 0$ ) *A*

**proof**–

**from** *fA* **have** *fab*: *finite (A* ∩ *B)* **by** *auto*

**have** *aba*: *A* ∩ *B* ⊆ *A* **by** *blast*

**let** *?g* =  $\lambda x. \text{if } x \in A \cap B \text{ then } f \ x \text{ else } 0$

**from** *setsum-mono-zero-left*[*OF fA aba*, *of ?g*]

**show** *?thesis* **by** *simp*

**qed**

**lemma** *setsum-cases*:

**assumes** *fA*: *finite A*

**shows** *setsum* ( $\lambda x. \text{if } x \in B \text{ then } f \ x \text{ else } g \ x$ ) *A* =

*setsum f* (*A* ∩ *B*) + *setsum g* (*A* ∩ − *B*)

**proof**–

**have** *a*: *A* = *A* ∩ *B* ∪ *A* ∩ − *B* (*A* ∩ *B*) ∩ (*A* ∩ − *B*) = {}

**by** *blast+*

**from** *fA*

**have** *f*: *finite (A* ∩ *B)* *finite (A* ∩ − *B)* **by** *auto*

**let** *?g* =  $\lambda x. \text{if } x \in B \text{ then } f \ x \text{ else } g \ x$

**from** *setsum-Un-disjoint*[*OF f a(2)*, *of ?g*] *a(1)*

**show** *?thesis* **by** *simp*

**qed**

**lemma** *setsum-UN-disjoint*:

*finite I* ==> (*ALL i*:*I*. *finite (A i)*) ==>

(*ALL i*:*I*. *ALL j*:*I*. *i* ≠ *j* --> *A i* Int *A j* = {}) ==>

*setsum f* (*UNION I A*) = ( $\sum i \in I. \text{setsum } f \ (A \ i)$ )

**by**(*simp add: setsum-def comm-monoid-add.fold-image-UN-disjoint cong: setsum-cong*)

No need to assume that *C* is finite. If infinite, the rhs is directly 0, and  $\bigcup C$  is also infinite, hence the lhs is also 0.

**lemma** *setsum-Union-disjoint*:

[| (*ALL A*:*C*. *finite A*);

(*ALL A*:*C*. *ALL B*:*C*. *A* ≠ *B* --> *A* Int *B* = {}) |]

==> *setsum f* (*Union C*) = *setsum* (*setsum f*) *C*

**apply** (*cases finite C*)



```

prefer 2 apply (force dest: finite-UnionD simp add: setsum-def)
apply (erule setsum-UN-disjoint [of C id f])
apply (unfold Union-def id-def, assumption+)
done

```

```

lemma setsum-Sigma: finite A ==> ALL x:A. finite (B x) ==>
  (∑ x∈A. (∑ y∈B x. f x y)) = (∑ (x,y)∈(SIGMA x:A. B x). f x y)
by(simp add:setsum-def comm-monoid-add.fold-image-Sigma split-def cong:setsum-cong)

```

Here we can eliminate the finiteness assumptions, by cases.

```

lemma setsum-cartesian-product:
  (∑ x∈A. (∑ y∈B. f x y)) = (∑ (x,y) ∈ A <*> B. f x y)
apply (cases finite A)
apply (cases finite B)
apply (simp add: setsum-Sigma)
apply (cases A={}, simp)
apply (simp)
apply (auto simp add: setsum-def
  dest: finite-cartesian-productD1 finite-cartesian-productD2)
done

```

```

lemma setsum-addf: setsum (%x. f x + g x) A = (setsum f A + setsum g A)
by(simp add:setsum-def comm-monoid-add.fold-image-distrib)

```

#### 17.4.1 Properties in more restricted classes of structures

```

lemma setsum-SucD: setsum f A = Suc n ==> EX a:A. 0 < f a
apply (case-tac finite A)
prefer 2 apply (simp add: setsum-def)
apply (erule rev-mp)
apply (erule finite-induct, auto)
done

```

```

lemma setsum-eq-0-iff [simp]:
  finite F ==> (setsum f F = 0) = (ALL a:F. f a = (0::nat))
by (induct set: finite) auto

```

```

lemma setsum-eq-Suc0-iff: finite A ==>
  (setsum f A = Suc 0) = (EX a:A. f a = Suc 0 & (ALL b:A. a≠b → f b = 0))
apply(erule finite-induct)
apply (auto simp add:add-is-1)
done

```

```

lemmas setsum-eq-1-iff = setsum-eq-Suc0-iff[simplified One-nat-def[symmetric]]

```

```

lemma setsum-Un-nat: finite A ==> finite B ==>
  (setsum f (A Un B) :: nat) = setsum f A + setsum f B - setsum f (A Int B)
  — For the natural numbers, we have subtraction.

```

**by** (*subst setsum-Un-Int [symmetric], auto simp add: algebra-simps*)

**lemma** *setsum-Un: finite A ==> finite B ==>*

*(setsum f (A Un B) :: 'a :: ab-group-add) =*  
*setsum f A + setsum f B - setsum f (A Int B)*

**by** (*subst setsum-Un-Int [symmetric], auto simp add: algebra-simps*)

**lemma** (*in comm-monoid-mult*) *fold-image-1: finite S ==> ( $\forall x \in S. f x = 1$ ) ==>*  
*fold-image op \* f 1 S = 1*

**apply** (*induct set: finite*)

**apply** *simp by (auto simp add: fold-image-insert)*

**lemma** (*in comm-monoid-mult*) *fold-image-Un-one:*

**assumes** *fS: finite S and fT: finite T*

**and** *I0:  $\forall x \in S \cap T. f x = 1$*

**shows** *fold-image (op \*) f 1 (S  $\cup$  T) = fold-image (op \*) f 1 S \* fold-image (op \*) f 1 T*

**proof** –

**have** *fold-image op \* f 1 (S  $\cap$  T) = 1*

**apply** (*rule fold-image-1*)

**using** *fS fT I0 by auto*

**with** *fold-image-Un-Int[OF fS fT]* **show** *?thesis by simp*

**qed**

**lemma** *setsum-eq-general-reverses:*

**assumes** *fS: finite S and fT: finite T*

**and** *kh:  $\bigwedge y. y \in T \implies k y \in S \wedge h (k y) = y$*

**and** *hk:  $\bigwedge x. x \in S \implies h x \in T \wedge k (h x) = x \wedge g (h x) = f x$*

**shows** *setsum f S = setsum g T*

**apply** (*simp add: setsum-def fS fT*)

**apply** (*rule comm-monoid-add.fold-image-eq-general-inverses[OF fS]*)

**apply** (*erule kh*)

**apply** (*erule hk*)

**done**

**lemma** *setsum-Un-zero:*

**assumes** *fS: finite S and fT: finite T*

**and** *I0:  $\forall x \in S \cap T. f x = 0$*

**shows** *setsum f (S  $\cup$  T) = setsum f S + setsum f T*

**using** *fS fT*

**apply** (*simp add: setsum-def*)

**apply** (*rule comm-monoid-add.fold-image-Un-one*)

**using** *I0 by auto*

**lemma** *setsum-UNION-zero:*

**assumes** *fS: finite S and fSS:  $\forall T \in S. finite T$*

```

and f0:  $\bigwedge T1\ T2\ x.\ T1 \in S \implies T2 \in S \implies T1 \neq T2 \implies x \in T1 \implies x \in T2$ 
 $\implies f\ x = 0$ 
shows setsum f ( $\bigcup S$ ) = setsum ( $\lambda T.\ \text{setsum } f\ T$ ) S
using fSS f0
proof(induct rule: finite-induct[OF fS])
  case 1 thus ?case by simp
next
  case (2 T F)
  then have fTF: finite T  $\forall T \in F.$  finite T finite F and TF:  $T \notin F$ 
    and H: setsum f ( $\bigcup F$ ) = setsum (setsum f) F by (auto simp add: finite-insert)
  from fTF have fUF: finite ( $\bigcup F$ ) by (auto intro: finite-Union)
  from 2.prem1 TF fTF
  show ?case
    by (auto simp add: H[symmetric] intro: setsum-Un-zero[OF fTF(1) fUF, of f])
qed

```

```

lemma setsum-diff1-nat: (setsum f (A - {a}) :: nat) =
  (if a:A then setsum f A - f a else setsum f A)
apply (case-tac finite A)
prefer 2 apply (simp add: setsum-def)
apply (erule finite-induct)
apply (auto simp add: insert-Diff-if)
apply (drule-tac a = a in mk-disjoint-insert, auto)
done

```

```

lemma setsum-diff1: finite A  $\implies$ 
  (setsum f (A - {a}) :: ('a::ab-group-add)) =
  (if a:A then setsum f A - f a else setsum f A)
by (erule finite-induct) (auto simp add: insert-Diff-if)

```

```

lemma setsum-diff1 '[rule-format]:
  finite A  $\implies a \in A \longrightarrow (\sum x \in A. f\ x) = f\ a + (\sum x \in (A - \{a\}). f\ x)$ 
apply (erule finite-induct[where F=A and P=% A. ( $a \in A \longrightarrow (\sum x \in A. f\ x)$ 
  = f a + ( $\sum x \in (A - \{a\}). f\ x$ )])])
apply (auto simp add: insert-Diff-if add-ac)
done

```

```

lemma setsum-diff-nat:
assumes finite B and  $B \subseteq A$ 
shows (setsum f (A - B) :: nat) = (setsum f A) - (setsum f B)
using assms
proof induct
  show setsum f (A - {}) = (setsum f A) - (setsum f {}) by simp
next
  fix F x assume finF: finite F and xnotinF:  $x \notin F$ 

```

```

    and  $xFinA$ :  $insert\ x\ F \subseteq A$ 
    and  $IH$ :  $F \subseteq A \implies setsum\ f\ (A - F) = setsum\ f\ A - setsum\ f\ F$ 
    from  $xnotinF\ xFinA$  have  $xinAF$ :  $x \in (A - F)$  by simp
    from  $xinAF$  have  $A$ :  $setsum\ f\ ((A - F) - \{x\}) = setsum\ f\ (A - F) - f\ x$ 
      by (simp add: setsum-diff1-nat)
    from  $xFinA$  have  $F \subseteq A$  by simp
    with  $IH$  have  $setsum\ f\ (A - F) = setsum\ f\ A - setsum\ f\ F$  by simp
    with  $A$  have  $B$ :  $setsum\ f\ ((A - F) - \{x\}) = setsum\ f\ A - setsum\ f\ F - f\ x$ 
      by simp
    from  $xnotinF$  have  $A - insert\ x\ F = (A - F) - \{x\}$  by auto
    with  $B$  have  $C$ :  $setsum\ f\ (A - insert\ x\ F) = setsum\ f\ A - setsum\ f\ F - f\ x$ 
      by simp
    from  $finF\ xnotinF$  have  $setsum\ f\ (insert\ x\ F) = setsum\ f\ F + f\ x$  by simp
    with  $C$  have  $setsum\ f\ (A - insert\ x\ F) = setsum\ f\ A - setsum\ f\ (insert\ x\ F)$ 
      by simp
    thus  $setsum\ f\ (A - insert\ x\ F) = setsum\ f\ A - setsum\ f\ (insert\ x\ F)$  by simp
  qed

```

lemma *setsum-diff*:

```

    assumes  $le$ :  $finite\ A\ B \subseteq A$ 
    shows  $setsum\ f\ (A - B) = setsum\ f\ A - ((setsum\ f\ B)::('a::ab-group-add))$ 
  proof -
    from  $le$  have  $finiteB$ :  $finite\ B$  using finite-subset by auto
    show ?thesis using  $finiteB\ le$ 
    proof induct
      case empty
      thus ?case by auto
    next
      case ( $insert\ x\ F$ )
      thus ?case using  $le\ finiteB$ 
      by (simp add: Diff-insert[where  $a=x$  and  $B=F$ ]) setsum-diff1 insert-absorb
    qed
  qed

```

lemma *setsum-mono*:

```

    assumes  $le$ :  $\bigwedge i. i \in K \implies f\ (i::'a) \leq ((g\ i)::('b::\{comm-monoid-add, pordered-ab-semigroup-add\}))$ 
    shows  $(\sum i \in K. f\ i) \leq (\sum i \in K. g\ i)$ 
  proof (cases  $finite\ K$ )
    case True
    thus ?thesis using  $le$ 
    proof induct
      case empty
      thus ?case by simp
    next
      case insert
      thus ?case using add-mono by fastsimp
    qed
  next
    case False

```

```

thus ?thesis
  by (simp add: setsum-def)
qed

```

```

lemma setsum-strict-mono:
  fixes f :: 'a  $\Rightarrow$  'b::{pordered-cancel-ab-semigroup-add,comm-monoid-add}
  assumes finite A A  $\neq$  {}
  and !!x. x:A  $\implies$  f x < g x
  shows setsum f A < setsum g A
  using prems
proof (induct rule: finite-ne-induct)
  case singleton thus ?case by simp
next
  case insert thus ?case by (auto simp: add-strict-mono)
qed

```

```

lemma setsum-negf:
  setsum (%x. - (f x)::'a::ab-group-add) A = - setsum f A
proof (cases finite A)
  case True thus ?thesis by (induct set: finite) auto
next
  case False thus ?thesis by (simp add: setsum-def)
qed

```

```

lemma setsum-subtractf:
  setsum (%x. ((f x)::'a::ab-group-add) - g x) A =
    setsum f A - setsum g A
proof (cases finite A)
  case True thus ?thesis by (simp add: diff-minus setsum-addf setsum-negf)
next
  case False thus ?thesis by (simp add: setsum-def)
qed

```

```

lemma setsum-nonneg:
  assumes nn:  $\forall x \in A. (0::'a::\{pordered-ab-semigroup-add,comm-monoid-add\}) \leq$ 
    f x
  shows 0  $\leq$  setsum f A
proof (cases finite A)
  case True thus ?thesis using nn
  proof induct
    case empty then show ?case by simp
  next
    case (insert x F)
    then have 0 + 0  $\leq$  f x + setsum f F by (blast intro: add-mono)
    with insert show ?case by simp
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

```

```

lemma setsum-nonpos:
  assumes np:  $\forall x \in A. f\ x \leq (0 :: 'a :: \{pordered-ab-semigroup-add, comm-monoid-add\})$ 
  shows  $setsum\ f\ A \leq 0$ 
proof (cases finite A)
  case True thus ?thesis using np
proof induct
  case empty then show ?case by simp
next
  case (insert x F)
  then have  $f\ x + setsum\ f\ F \leq 0 + 0$  by (blast intro: add-mono)
  with insert show ?case by simp
qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-mono2:
  fixes f :: 'a  $\Rightarrow$  'b ::  $\{pordered-ab-semigroup-add-imp-le, comm-monoid-add\}$ 
  assumes fin: finite B and sub:  $A \subseteq B$  and nn:  $\bigwedge b. b \in B - A \implies 0 \leq f\ b$ 
  shows  $setsum\ f\ A \leq setsum\ f\ B$ 
proof -
  have  $setsum\ f\ A \leq setsum\ f\ A + setsum\ f\ (B - A)$ 
  by (simp add: add-increasing2[OF setsum-nonneg] nn Ball-def)
  also have  $\dots = setsum\ f\ (A \cup (B - A))$  using fin finite-subset[OF sub fin]
  by (simp add: setsum-Un-disjoint del: Un-Diff-cancel)
  also have  $A \cup (B - A) = B$  using sub by blast
  finally show ?thesis .
qed

lemma setsum-mono3: finite B  $\implies A \leq B \implies$ 
  ALL x: B - A.
   $0 \leq ((f\ x) :: 'a :: \{comm-monoid-add, pordered-ab-semigroup-add\}) \implies$ 
   $setsum\ f\ A \leq setsum\ f\ B$ 
apply (subgoal-tac setsum f B = setsum f A + setsum f (B - A))
apply (erule ssubst)
apply (subgoal-tac setsum f A + 0 <= setsum f A + setsum f (B - A))
apply simp
apply (rule add-left-mono)
apply (erule setsum-nonneg)
apply (subst setsum-Un-disjoint [THEN sym])
apply (erule finite-subset, assumption)
apply (rule finite-subset)
prefer 2
apply assumption
apply auto
apply (rule setsum-cong)
apply auto
done

```

```

lemma setsum-right-distrib:
  fixes  $f :: 'a \Rightarrow ('b::\text{semiring-0})$ 
  shows  $r * \text{setsum } f \ A = \text{setsum } (\%n. r * f \ n) \ A$ 
proof (cases finite A)
  case True
  thus ?thesis
  proof induct
    case empty thus ?case by simp
  next
    case (insert x A) thus ?case by (simp add: right-distrib)
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-left-distrib:
   $\text{setsum } f \ A * (r::'a::\text{semiring-0}) = (\sum n \in A. f \ n * r)$ 
proof (cases finite A)
  case True
  then show ?thesis
  proof induct
    case empty thus ?case by simp
  next
    case (insert x A) thus ?case by (simp add: left-distrib)
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-divide-distrib:
   $\text{setsum } f \ A / (r::'a::\text{field}) = (\sum n \in A. f \ n / r)$ 
proof (cases finite A)
  case True
  then show ?thesis
  proof induct
    case empty thus ?case by simp
  next
    case (insert x A) thus ?case by (simp add: add-divide-distrib)
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-abs[iff]:
  fixes  $f :: 'a \Rightarrow ('b::\text{pordered-ab-group-add-abs})$ 
  shows  $\text{abs } (\text{setsum } f \ A) \leq \text{setsum } (\%i. \text{abs}(f \ i)) \ A$ 
proof (cases finite A)
  case True

```

```

thus ?thesis
proof induct
  case empty thus ?case by simp
next
  case (insert x A)
  thus ?case by (auto intro: abs-triangle-ineq order-trans)
qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-abs-ge-zero[iff]:
  fixes f :: 'a => ('b::pordered-ab-group-add-abs)
  shows  $0 \leq \text{setsum } (\%i. \text{abs}(f\ i))\ A$ 
proof (cases finite A)
  case True
  thus ?thesis
  proof induct
    case empty thus ?case by simp
  next
    case (insert x A) thus ?case by (auto simp: add-nonneg-nonneg)
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma abs-setsum-abs[simp]:
  fixes f :: 'a => ('b::pordered-ab-group-add-abs)
  shows  $\text{abs } (\sum a \in A. \text{abs}(f\ a)) = (\sum a \in A. \text{abs}(f\ a))$ 
proof (cases finite A)
  case True
  thus ?thesis
  proof induct
    case empty thus ?case by simp
  next
    case (insert a A)
    hence  $|\sum a \in \text{insert } a\ A. |f\ a|| = ||f\ a| + (\sum a \in A. |f\ a|)|$  by simp
    also have  $\dots = ||f\ a| + |\sum a \in A. |f\ a||$  using insert by simp
    also have  $\dots = |f\ a| + |\sum a \in A. |f\ a||$ 
    by (simp del: abs-of-nonneg)
    also have  $\dots = (\sum a \in \text{insert } a\ A. |f\ a|)$  using insert by simp
    finally show ?case .
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

```

Commuting outer and inner summation

**lemma** *swap-inj-on*:



*inj-on* ( $\%(i, j). (j, i)$ ) ( $A \times B$ )  
**by** (*unfold inj-on-def*) *fast*

**lemma** *swap-product*:

( $\%(i, j). (j, i)$ )  $\text{'}$  ( $A \times B$ ) =  $B \times A$   
**by** (*simp add: split-def image-def*) *blast*

**lemma** *setsum-commute*:

( $\sum_{i \in A}. \sum_{j \in B}. f\ i\ j$ ) = ( $\sum_{j \in B}. \sum_{i \in A}. f\ i\ j$ )

**proof** (*simp add: setsum-cartesian-product*)

**have** ( $\sum (x, y) \in A <*> B. f\ x\ y$ ) =  
 ( $\sum (y, x) \in \%(i, j). (j, i) \text{'}$  ( $A \times B$ )).  $f\ x\ y$ )  
 (**is** ?s = -)

**apply** (*simp add: setsum-reindex* [**where**  $f = \%(i, j). (j, i)$ ] *swap-inj-on*)

**apply** (*simp add: split-def*)

**done**

**also have** ... = ( $\sum (y, x) \in B \times A. f\ x\ y$ )

(**is** - = ?t)

**apply** (*simp add: swap-product*)

**done**

**finally show** ?s = ?t .

**qed**

**lemma** *setsum-product*:

**fixes**  $f :: 'a \Rightarrow ('b::\text{semiring-0})$

**shows**  $\text{setsum } f\ A * \text{setsum } g\ B = (\sum_{i \in A}. \sum_{j \in B}. f\ i * g\ j)$

**by** (*simp add: setsum-right-distrib setsum-left-distrib*) (*rule setsum-commute*)

## 17.5 Generalized product over a set

**definition** *setprod* :: ( $'a \Rightarrow 'b$ )  $\Rightarrow 'a\ \text{set} \Rightarrow 'b::\text{comm-monoid-mult}$

**where** *setprod*  $f\ A == \text{if finite } A \text{ then fold-image } (op\ *)\ f\ 1\ A \text{ else } 1$

**abbreviation**

*Setprod* ( $\prod - [1000] 999$ ) **where**

$\prod A == \text{setprod } (\%x. x)\ A$

**syntax**

*-setprod* ::  $pttrn \Rightarrow 'a\ \text{set} \Rightarrow 'b \Rightarrow 'b::\text{comm-monoid-mult}$  ((*3PROD* -:: -) [0, 51, 10] 10)

**syntax** (*xsymbols*)

*-setprod* ::  $pttrn \Rightarrow 'a\ \text{set} \Rightarrow 'b \Rightarrow 'b::\text{comm-monoid-mult}$  ((*3* $\prod$  - $\in$ -. -) [0, 51, 10] 10)

**syntax** (*HTML output*)

*-setprod* ::  $pttrn \Rightarrow 'a\ \text{set} \Rightarrow 'b \Rightarrow 'b::\text{comm-monoid-mult}$  ((*3* $\prod$  - $\in$ -. -) [0, 51, 10] 10)

**translations** — Beware of argument permutation!

*PROD*  $i:A. b == \text{CONST } \text{setprod } (\%i. b)\ A$

$$\prod i \in A. b == \text{CONST setprod } (\%i. b) A$$

Instead of  $\prod x \in \{x. P\}. e$  we introduce the shorter  $\prod x | P. e$ .

**syntax**

$$\text{-qsetprod} :: \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow 'a \ ((\exists \text{PROD} - | / - / -) [0,0,10] 10)$$

**syntax** (*xsymbols*)

$$\text{-qsetprod} :: \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow 'a \ ((\exists \prod - | (-) / -) [0,0,10] 10)$$

**syntax** (*HTML output*)

$$\text{-qsetprod} :: \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow 'a \ ((\exists \prod - | (-) / -) [0,0,10] 10)$$

**translations**

$$\text{PROD } x | P. t \Rightarrow \text{CONST setprod } (\%x. t) \{x. P\}$$

$$\prod x | P. t \Rightarrow \text{CONST setprod } (\%x. t) \{x. P\}$$

**lemma** *setprod-empty* [*simp*]:  $\text{setprod } f \ \{\} = 1$

**by** (*auto simp add: setprod-def*)

**lemma** *setprod-insert* [*simp*]:  $[\mid \text{finite } A; a \notin A \mid] \Rightarrow$

$$\text{setprod } f \ (\text{insert } a \ A) = f \ a * \text{setprod } f \ A$$

**by** (*simp add: setprod-def*)

**lemma** *setprod-infinite* [*simp*]:  $\sim \text{finite } A \Rightarrow \text{setprod } f \ A = 1$

**by** (*simp add: setprod-def*)

**lemma** *setprod-reindex*:

$$\text{inj-on } f \ B \Rightarrow \text{setprod } h \ (f \ ' B) = \text{setprod } (h \circ f) \ B$$

**by**(*auto simp: setprod-def fold-image-reindex dest!: finite-imageD*)

**lemma** *setprod-reindex-id*:  $\text{inj-on } f \ B \Rightarrow \text{setprod } f \ B = \text{setprod id } (f \ ' B)$

**by** (*auto simp add: setprod-reindex*)

**lemma** *setprod-cong*:

$$A = B \Rightarrow (!x. x:B \Rightarrow f \ x = g \ x) \Rightarrow \text{setprod } f \ A = \text{setprod } g \ B$$

**by**(*fastsimp simp: setprod-def intro: fold-image-cong*)

**lemma** *strong-setprod-cong*[*cong*]:

$$A = B \Rightarrow (!x. x:B \Rightarrow \text{simp} \Rightarrow f \ x = g \ x) \Rightarrow \text{setprod } f \ A = \text{setprod } g \ B$$

**by**(*fastsimp simp: simp-implies-def setprod-def intro: fold-image-cong*)

**lemma** *setprod-reindex-cong*:  $\text{inj-on } f \ A \Rightarrow$

$$B = f \ ' A \Rightarrow g = h \circ f \Rightarrow \text{setprod } h \ B = \text{setprod } g \ A$$

**by** (*frule setprod-reindex, simp*)

**lemma** *strong-setprod-reindex-cong*: **assumes** *i*:  $\text{inj-on } f \ A$

**and** *B*:  $B = f \ ' A$  **and** *eq*:  $\bigwedge x. x \in A \Rightarrow g \ x = (h \circ f) \ x$

**shows**  $\text{setprod } h \ B = \text{setprod } g \ A$

**proof**–

**have**  $\text{setprod } h \ B = \text{setprod } (h \circ f) \ A$

```

    by (simp add: B setprod-reindex[OF i, of h])
  then show ?thesis apply simp
    apply (rule setprod-cong)
    apply simp
    by (simp add: eq)
qed

```

```

lemma setprod-Un-one:
  assumes fS: finite S and fT: finite T
  and I0:  $\forall x \in S \cap T. f x = 1$ 
  shows setprod f (S  $\cup$  T) = setprod f S * setprod f T
  using fS fT
  apply (simp add: setprod-def)
  apply (rule fold-image-Un-one)
  using I0 by auto

```

```

lemma setprod-1: setprod (%i. 1) A = 1
  apply (case-tac finite A)
  apply (erule finite-induct, auto simp add: mult-ac)
  done

```

```

lemma setprod-1': ALL a:F. f a = 1 ==> setprod f F = 1
  apply (subgoal-tac setprod f F = setprod (%x. 1) F)
  apply (erule ssubst, rule setprod-1)
  apply (rule setprod-cong, auto)
  done

```

```

lemma setprod-Un-Int: finite A ==> finite B
  ==> setprod g (A Un B) * setprod g (A Int B) = setprod g A * setprod g B
  by (simp add: setprod-def fold-image-Un-Int[symmetric])

```

```

lemma setprod-Un-disjoint: finite A ==> finite B
  ==> A Int B = {} ==> setprod g (A Un B) = setprod g A * setprod g B
  by (subst setprod-Un-Int [symmetric], auto)

```

```

lemma setprod-mono-one-left:
  assumes fT: finite T and ST: S  $\subseteq$  T
  and z:  $\forall i \in T - S. f i = 1$ 
  shows setprod f S = setprod f T
proof -
  have eq: T = S  $\cup$  (T - S) using ST by blast
  have d: S  $\cap$  (T - S) = {} using ST by blast
  from fT ST have f: finite S finite (T - S) by (auto intro: finite-subset)
  show ?thesis
    by (simp add: setprod-Un-disjoint[OF f d, unfolded eq[symmetric]] setprod-1'[OF z])
qed

```

**lemmas** *setprod-mono-one-right* = *setprod-mono-one-left* [THEN sym]

**lemma** *setprod-delta*:

**assumes** *fS*: *finite S*

**shows** *setprod* ( $\lambda k. \text{if } k=a \text{ then } b \ k \text{ else } 1$ ) *S* = (*if* *a*  $\in$  *S* *then* *b a* *else* 1)

**proof**–

**let** *?f* = ( $\lambda k. \text{if } k=a \text{ then } b \ k \text{ else } 1$ )

{**assume** *a*: *a*  $\notin$  *S*

**hence**  $\forall k \in S. ?f \ k = 1$  **by** *simp*

**hence** *?thesis* **using** *a* **by** (*simp add: setprod-1 cong add: setprod-cong*) }

**moreover**

{**assume** *a*: *a*  $\in$  *S*

**let** *?A* = *S* – {*a*}

**let** *?B* = {*a*}

**have** *eq*: *S* = *?A*  $\cup$  *?B* **using** *a* **by** *blast*

**have** *dj*: *?A*  $\cap$  *?B* = {} **by** *simp*

**from** *fS* **have** *fAB*: *finite ?A finite ?B* **by** *auto*

**have** *fA1*: *setprod ?f ?A* = 1 **apply** (*rule setprod-1'*) **by** *auto*

**have** *setprod ?f ?A \* setprod ?f ?B* = *setprod ?f S*

**using** *setprod-Un-disjoint*[*OF fAB dj, of ?f, unfolded eq[symmetric]*]

**by** *simp*

**then have** *?thesis* **using** *a* **by** (*simp add: fA1 cong add: setprod-cong cong*

*del: if-weak-cong*)}

**ultimately show** *?thesis* **by** *blast*

**qed**

**lemma** *setprod-delta'*:

**assumes** *fS*: *finite S* **shows**

*setprod* ( $\lambda k. \text{if } a = k \text{ then } b \ k \text{ else } 1$ ) *S* =

(*if* *a*  $\in$  *S* *then* *b a* *else* 1)

**using** *setprod-delta*[*OF fS, of a b, symmetric*]

**by** (*auto intro: setprod-cong*)

**lemma** *setprod-UN-disjoint*:

*finite I*  $\implies$  ( $\text{ALL } i:I. \text{finite } (A \ i)$ )  $\implies$

( $\text{ALL } i:I. \text{ALL } j:I. i \neq j \longrightarrow A \ i \text{ Int } A \ j = \{\}$ )  $\implies$

*setprod f* ( $\text{UNION } I \ A$ ) = *setprod* ( $\%i. \text{setprod } f \ (A \ i)$ ) *I*

**by**(*simp add: setprod-def fold-image-UN-disjoint cong: setprod-cong*)

**lemma** *setprod-Union-disjoint*:

[| ( $\text{ALL } A:C. \text{finite } A$ );

( $\text{ALL } A:C. \text{ALL } B:C. A \neq B \longrightarrow A \text{ Int } B = \{\}$ ) |]

$\implies \text{setprod } f \ (\text{Union } C) = \text{setprod } (\text{setprod } f) \ C$

**apply** (*cases finite C*)

**prefer** 2 **apply** (*force dest: finite-UnionD simp add: setprod-def*)

**apply** (*frule setprod-UN-disjoint [of C id f]*)

**apply** (*unfold Union-def id-def, assumption+*)

**done**

```

lemma setprod-Sigma: finite A ==> ALL x:A. finite (B x) ==>
  (∏ x∈A. (∏ y∈ B x. f x y)) =
  (∏ (x,y)∈(SIGMA x:A. B x). f x y)
by(simp add:setprod-def fold-image-Sigma split-def cong:setprod-cong)

```

Here we can eliminate the finiteness assumptions, by cases.

```

lemma setprod-cartesian-product:
  (∏ x∈A. (∏ y∈ B. f x y)) = (∏ (x,y)∈(A <*> B). f x y)
apply (cases finite A)
apply (cases finite B)
apply (simp add: setprod-Sigma)
apply (cases A={}, simp)
apply (simp add: setprod-1)
apply (auto simp add: setprod-def
  dest: finite-cartesian-productD1 finite-cartesian-productD2)
done

```

```

lemma setprod-timesf:
  setprod (%x. f x * g x) A = (setprod f A * setprod g A)
by(simp add:setprod-def fold-image-distrib)

```

### 17.5.1 Properties in more restricted classes of structures

```

lemma setprod-eq-1-iff [simp]:
  finite F ==> (setprod f F = 1) = (ALL a:F. f a = (1::nat))
by (induct set: finite) auto

```

```

lemma setprod-zero:
  finite A ==> EX x: A. f x = (0::'a::comm-semiring-1) ==> setprod f A = 0
apply (induct set: finite, force, clarsimp)
apply (erule disjE, auto)
done

```

```

lemma setprod-nonneg [rule-format]:
  (ALL x: A. (0::'a::ordered-semidom) ≤ f x) --> 0 ≤ setprod f A
by (cases finite A, induct set: finite, simp-all add: mult-nonneg-nonneg)

```

```

lemma setprod-pos [rule-format]: (ALL x: A. (0::'a::ordered-semidom) < f x)
  --> 0 < setprod f A
by (cases finite A, induct set: finite, simp-all add: mult-pos-pos)

```

```

lemma setprod-zero-iff[simp]: finite A ==>
  (setprod f A = (0::'a::{comm-semiring-1,no-zero-divisors})) =
  (EX x: A. f x = 0)
by (erule finite-induct, auto simp:no-zero-divisors)

```

```

lemma setprod-pos-nat:
  finite S ==> (ALL x : S. f x > (0::nat)) ==> setprod f S > 0

```

**using** *setprod-zero-iff* **by**(*simp del:neq0-conv add:neq0-conv[symmetric]*)

**lemma** *setprod-pos-nat-iff*[*simp*]:

*finite S ==> (setprod f S > 0) = (ALL x : S. f x > (0::nat))*

**using** *setprod-zero-iff* **by**(*simp del:neq0-conv add:neq0-conv[symmetric]*)

**lemma** *setprod-Un*: *finite A ==> finite B ==> (ALL x: A Int B. f x ≠ 0) ==>*

*(setprod f (A Un B) :: 'a :: {field})*

*= setprod f A \* setprod f B / setprod f (A Int B)*

**by** (*subst setprod-Un-Int [symmetric], auto*)

**lemma** *setprod-diff1*: *finite A ==> f a ≠ 0 ==>*

*(setprod f (A - {a}) :: 'a :: {field}) =*

*(if a:A then setprod f A / f a else setprod f A)*

**by** (*erule finite-induct*) (*auto simp add: insert-Diff-if*)

**lemma** *setprod-inversef*: *finite A ==>*

*ALL x: A. f x ≠ (0::'a::{field,division-by-zero}) ==>*

*setprod (inverse ∘ f) A = inverse (setprod f A)*

**by** (*erule finite-induct*) *auto*

**lemma** *setprod-dividef*:

*[finite A;*

*∀ x ∈ A. g x ≠ (0::'a::{field,division-by-zero})]*

*==> setprod (%x. f x / g x) A = setprod f A / setprod g A*

**apply** (*subgoal-tac*

*setprod (%x. f x / g x) A = setprod (%x. f x \* (inverse ∘ g) x) A*)

**apply** (*erule ssubst*)

**apply** (*subst divide-inverse*)

**apply** (*subst setprod-timesf*)

**apply** (*subst setprod-inversef, assumption+, rule refl*)

**apply** (*rule setprod-cong, rule refl*)

**apply** (*subst divide-inverse, auto*)

**done**

**lemma** *setprod-dvd-setprod* [*rule-format*]:

*(ALL x : A. f x dvd g x) ⟶ setprod f A dvd setprod g A*

**apply** (*cases finite A*)

**apply** (*induct set: finite*)

**apply** (*auto simp add: dvd-def*)

**apply** (*rule-tac x = k \* ka in exI*)

**apply** (*simp add: algebra-simps*)

**done**

**lemma** *setprod-dvd-setprod-subset*:

*finite B ⟹ A ≤ B ⟹ setprod f A dvd setprod f B*

**apply** (*subgoal-tac setprod f B = setprod f A \* setprod f (B - A)*)

**apply** (*unfold dvd-def, blast*)

**apply** (*subst setprod-Un-disjoint [symmetric]*)

```

apply (auto elim: finite-subset intro: setprod-cong)
done

```

```

lemma setprod-dvd-setprod-subset2:
  finite B  $\implies$  A  $\leq$  B  $\implies$  ALL x : A. (f x :: 'a :: comm-semiring-1) dvd g x  $\implies$ 
    setprod f A dvd setprod g B
apply (rule dvd-trans)
apply (rule setprod-dvd-setprod, erule (1) bspec)
apply (erule (1) setprod-dvd-setprod-subset)
done

```

```

lemma dvd-setprod: finite A  $\implies$  i:A  $\implies$ 
  (f i :: 'a :: comm-semiring-1) dvd setprod f A
by (induct set: finite) (auto intro: dvd-mult)

```

```

lemma dvd-setsum [rule-format]: (ALL i : A. d dvd f i)  $\longrightarrow$ 
  (d :: 'a :: comm-semiring-1) dvd (SUM x : A. f x)
apply (cases finite A)
apply (induct set: finite)
apply auto
done

```

## 17.6 Finite cardinality

This definition, although traditional, is ugly to work with:  $\text{card } A == \text{LEAST } n. \text{EX } f. A = \{f \ i \mid i. i < n\}$ . But now that we have *setsum* things are easy:

```

definition card :: 'a set  $\Rightarrow$  nat
where card A = setsum ( $\lambda x. 1$ ) A

```

```

lemma card-empty [simp]: card {} = 0
by (simp add: card-def)

```

```

lemma card-infinite [simp]:  $\sim$  finite A  $\implies$  card A = 0
by (simp add: card-def)

```

```

lemma card-eq-setsum: card A = setsum ( $\%x. 1$ ) A
by (simp add: card-def)

```

```

lemma card-insert-disjoint [simp]:
  finite A  $\implies$  x  $\notin$  A  $\implies$  card (insert x A) = Suc(card A)
by(simp add: card-def)

```

```

lemma card-insert-if:
  finite A  $\implies$  card (insert x A) = (if x:A then card A else Suc(card(A)))
by (simp add: insert-absorb)

```

```

lemma card-0-eq [simp,noatp]: finite A  $\implies$  (card A = 0) = (A = {})
apply auto

```

**apply** (*drule-tac*  $a = x$  **in** *mk-disjoint-insert*, *clarify*, *auto*)  
**done**

**lemma** *card-eq-0-iff*:  $(\text{card } A = 0) = (A = \{\} \mid \sim \text{finite } A)$   
**by** *auto*

**lemma** *card-Suc-Diff1*:  $\text{finite } A \implies x: A \implies \text{Suc } (\text{card } (A - \{x\})) = \text{card } A$   
**apply**(*rule-tac*  $t = A$  **in** *insert-Diff* [*THEN subst*], *assumption*)  
**apply**(*simp del:insert-Diff-single*)  
**done**

**lemma** *card-Diff-singleton*:  
 $\text{finite } A \implies x: A \implies \text{card } (A - \{x\}) = \text{card } A - 1$   
**by** (*simp add: card-Suc-Diff1 [symmetric]*)

**lemma** *card-Diff-singleton-if*:  
 $\text{finite } A \implies \text{card } (A - \{x\}) = (\text{if } x : A \text{ then } \text{card } A - 1 \text{ else } \text{card } A)$   
**by** (*simp add: card-Diff-singleton*)

**lemma** *card-Diff-insert*[*simp*]:  
**assumes** *finite*  $A$  **and**  $a:A$  **and**  $a \sim: B$   
**shows**  $\text{card}(A - \text{insert } a B) = \text{card}(A - B) - 1$   
**proof** –  
   **have**  $A - \text{insert } a B = (A - B) - \{a\}$  **using** *assms* **by** *blast*  
   **then show** *?thesis* **using** *assms* **by**(*simp add:card-Diff-singleton*)  
**qed**

**lemma** *card-insert*:  $\text{finite } A \implies \text{card } (\text{insert } x A) = \text{Suc } (\text{card } (A - \{x\}))$   
**by** (*simp add: card-insert-if card-Suc-Diff1 del:card-Diff-insert*)

**lemma** *card-insert-le*:  $\text{finite } A \implies \text{card } A \leq \text{card } (\text{insert } x A)$   
**by** (*simp add: card-insert-if*)

**lemma** *card-mono*:  $\llbracket \text{finite } B; A \subseteq B \rrbracket \implies \text{card } A \leq \text{card } B$   
**by** (*simp add: card-def setsum-mono2*)

**lemma** *card-seteq*:  $\text{finite } B \implies (!A. A \leq B \implies \text{card } B \leq \text{card } A \implies A = B)$   
**apply** (*induct set: finite, simp, clarify*)  
**apply** (*subgoal-tac*  $\text{finite } A \ \& \ A - \{x\} \leq F$ )  
   **prefer** 2 **apply** (*blast intro: finite-subset, atomize*)  
**apply** (*drule-tac*  $x = A - \{x\}$  **in** *spec*)  
**apply** (*simp add: card-Diff-singleton-if split add: split-if-asm*)  
**apply** (*case-tac*  $\text{card } A$ , *auto*)  
**done**

**lemma** *psubset-card-mono*:  $\text{finite } B \implies A < B \implies \text{card } A < \text{card } B$   
**apply** (*simp add: psubset-eq linorder-not-le [symmetric]*)



**apply** (*blast dest: card-seteq*)  
**done**

**lemma** *card-Un-Int: finite A ==> finite B*  
 $\implies \text{card } A + \text{card } B = \text{card } (A \text{ Un } B) + \text{card } (A \text{ Int } B)$   
**by**(*simp add:card-def setsum-Un-Int*)

**lemma** *card-Un-disjoint: finite A ==> finite B*  
 $\implies A \text{ Int } B = \{\} \implies \text{card } (A \text{ Un } B) = \text{card } A + \text{card } B$   
**by** (*simp add: card-Un-Int*)

**lemma** *card-Diff-subset:*  
 $\text{finite } B \implies B \leq A \implies \text{card } (A - B) = \text{card } A - \text{card } B$   
**by**(*simp add:card-def setsum-diff-nat*)

**lemma** *card-Diff1-less: finite A ==> x: A ==> card (A - {x}) < card A*  
**apply** (*rule Suc-less-SucD*)  
**apply** (*simp add: card-Suc-Diff1 del:card-Diff-insert*)  
**done**

**lemma** *card-Diff2-less:*  
 $\text{finite } A \implies x: A \implies y: A \implies \text{card } (A - \{x\} - \{y\}) < \text{card } A$   
**apply** (*case-tac x = y*)  
**apply** (*simp add: card-Diff1-less del:card-Diff-insert*)  
**apply** (*rule less-trans*)  
**prefer 2 apply** (*auto intro!: card-Diff1-less simp del:card-Diff-insert*)  
**done**

**lemma** *card-Diff1-le: finite A ==> card (A - {x}) <= card A*  
**apply** (*case-tac x : A*)  
**apply** (*simp-all add: card-Diff1-less less-imp-le*)  
**done**

**lemma** *card-psubset: finite B ==> A ⊆ B ==> card A < card B ==> A < B*  
**by** (*erule psubsetI, blast*)

**lemma** *insert-partition:*  

$$\llbracket x \notin F; \forall c1 \in \text{insert } x F. \forall c2 \in \text{insert } x F. c1 \neq c2 \longrightarrow c1 \cap c2 = \{\} \rrbracket$$

$$\implies x \cap \bigcup F = \{\}$$
  
**by** *auto*

main cardinality theorem

**lemma** *card-partition [rule-format]:*  
 $\text{finite } C \implies$   
 $\text{finite } (\bigcup C) \dashrightarrow$   
 $(\forall c \in C. \text{card } c = k) \dashrightarrow$   
 $(\forall c1 \in C. \forall c2 \in C. c1 \neq c2 \dashrightarrow c1 \cap c2 = \{\}) \dashrightarrow$   
 $k * \text{card}(C) = \text{card } (\bigcup C)$   
**apply** (*erule finite-induct, simp*)

```

apply (simp add: card-insert-disjoint card-Un-disjoint insert-partition
         finite-subset [of -  $\cup$  (insert  $x$   $F$ )])
done

```

The form of a finite set of given cardinality

```

lemma card-eq-SucD:
assumes card  $A = \text{Suc } k$ 
shows  $\exists b \in B. A = \text{insert } b \ B \ \& \ b \notin B \ \& \ \text{card } B = k \ \& \ (k=0 \longrightarrow B=\{\})$ 
proof -
  have fin: finite  $A$  using assms by (auto intro: ccontr)
  moreover have card  $A \neq 0$  using assms by auto
  ultimately obtain  $b$  where  $b \in A$  by auto
  show ?thesis
proof (intro exI conjI)
  show  $A = \text{insert } b \ (A - \{b\})$  using  $b$  by blast
  show  $b \notin A - \{b\}$  by blast
  show card  $(A - \{b\}) = k$  and  $k = 0 \longrightarrow A - \{b\} = \{\}$ 
    using assms  $b$  fin by (fastsimp dest: mk-disjoint-insert)+
qed
qed

```

```

lemma card-Suc-eq:
  (card  $A = \text{Suc } k =$ 
   ( $\exists b \in B. A = \text{insert } b \ B \ \& \ b \notin B \ \& \ \text{card } B = k \ \& \ (k=0 \longrightarrow B=\{\})$ ))
apply (rule iffI)
apply (erule card-eq-SucD)
apply (auto)
apply (subst card-insert)
apply (auto intro: ccontr)
done

```

```

lemma setsum-constant [simp]:  $(\sum x \in A. y) = \text{of-nat}(\text{card } A) * y$ 
apply (cases finite  $A$ )
apply (erule finite-induct)
apply (auto simp add: algebra-simps)
done

```

```

lemma setprod-constant: finite  $A \implies (\prod x \in A. (y :: 'a :: \{\text{recpower, comm-monoid-mult}\}))$ 
  =  $y^{\text{card } A}$ 
apply (erule finite-induct)
apply (auto simp add: power-Suc)
done

```

```

lemma setprod-gen-delta:
  assumes fS: finite  $S$ 
  shows setprod  $(\lambda k. \text{if } k=a \text{ then } b \ k \text{ else } c) \ S = (\text{if } a \in S \text{ then } (b \ a :: 'a :: \{\text{comm-monoid-mult,}$ 
     $\text{recpower}\}) * c^{\text{card } S - 1} \text{ else } c^{\text{card } S})$ 
proof -
  let ?f =  $(\lambda k. \text{if } k=a \text{ then } b \ k \text{ else } c)$ 

```

```

{assume a: a ∉ S
  hence ∀ k ∈ S. ?f k = c by simp
  hence ?thesis using a setprod-constant[OF fS, of c] by (simp add: setprod-1
congr add: setprod-cong) }
moreover
{assume a: a ∈ S
  let ?A = S - {a}
  let ?B = {a}
  have eq: S = ?A ∪ ?B using a by blast
  have dj: ?A ∩ ?B = {} by simp
  from fS have fAB: finite ?A finite ?B by auto
  have fA0: setprod ?f ?A = setprod (λi. c) ?A
    apply (rule setprod-cong) by auto
  have cA: card ?A = card S - 1 using fS a by auto
  have fA1: setprod ?f ?A = c ^ card ?A unfolding fA0 apply (rule setprod-constant)
using fS by auto
  have setprod ?f ?A * setprod ?f ?B = setprod ?f S
    using setprod-Un-disjoint[OF fAB dj, of ?f, unfolded eq[symmetric]]
    by simp
  then have ?thesis using a cA
    by (simp add: fA1 ring-simps congr add: setprod-cong congr del: if-weak-cong)}
ultimately show ?thesis by blast
qed

```

**lemma** *setsum-bounded*:

```

assumes le: ∧i. i ∈ A ⇒ f i ≤ (K::'a::{semiring-1, pordered-ab-semigroup-add})
shows setsum f A ≤ of-nat(card A) * K
proof (cases finite A)
  case True
    thus ?thesis using le setsum-mono[where K=A and g = %x. K] by simp
  next
    case False thus ?thesis by (simp add: setsum-def)
qed

```

### 17.6.1 Cardinality of unions

**lemma** *card-UN-disjoint*:

```

finite I ==> (ALL i:I. finite (A i)) ==>
  (ALL i:I. ALL j:I. i ≠ j --> A i Int A j = {})
==> card (UNION I A) = (∑ i ∈ I. card(A i))
apply (simp add: card-def del: setsum-constant)
apply (subgoal-tac
  setsum (%i. card (A i)) I = setsum (%i. (setsum (%x. 1) (A i))) I)
apply (simp add: setsum-UN-disjoint del: setsum-constant)
apply (simp cong: setsum-cong)
done

```

**lemma** *card-Union-disjoint*:

```

finite C ==> (ALL A:C. finite A) ==>
  (ALL A:C. ALL B:C. A ≠ B --> A Int B = {})
==> card (Union C) = setsum card C
apply (frule card-UN-disjoint [of C id])
apply (unfold Union-def id-def, assumption+)
done

```

### 17.6.2 Cardinality of image

The image of a finite set can be expressed using *fold-image*.

```

lemma image-eq-fold-image:
  finite A ==> f ‘ A = fold-image (op Un) (%x. {f x}) {} A
proof (induct rule: finite-induct)
  case empty then show ?case by simp
next
  interpret ab-semigroup-mult op Un
  proof qed auto
  case insert
  then show ?case by simp
qed

```

```

lemma card-image-le: finite A ==> card (f ‘ A) ≤ card A
apply (induct set: finite)
  apply simp
apply (simp add: le-SucI finite-imageI card-insert-if)
done

```

```

lemma card-image: inj-on f A ==> card (f ‘ A) = card A
by (simp add: card-def setsum-reindex o-def del: setsum-constant)

```

```

lemma endo-inj-surj: finite A ==> f ‘ A ⊆ A ==> inj-on f A ==> f ‘ A = A
by (simp add: card-seteq card-image)

```

```

lemma eq-card-imp-inj-on:
  [| finite A; card (f ‘ A) = card A |] ==> inj-on f A
apply (induct rule: finite-induct)
apply simp
apply (frule card-image-le [where f = f])
apply (simp add: card-insert-if split: if-splits)
done

```

```

lemma inj-on-iff-eq-card:
  finite A ==> inj-on f A = (card (f ‘ A) = card A)
by (blast intro: card-image eq-card-imp-inj-on)

```

```

lemma card-inj-on-le:
  [| inj-on f A; f ‘ A ⊆ B; finite B |] ==> card A ≤ card B
apply (subgoal-tac finite A)

```

```

apply (force intro: card-mono simp add: card-image [symmetric])
apply (blast intro: finite-imageD dest: finite-subset)
done

```

```

lemma card-bij-eq:
  [| inj-on f A; f ‘ A ⊆ B; inj-on g B; g ‘ B ⊆ A;
    finite A; finite B |] ==> card A = card B
by (auto intro: le-anti-sym card-inj-on-le)

```

### 17.6.3 Cardinality of products

```

lemma card-SigmaI [simp]:
  [| finite A; ALL a:A. finite (B a) |]
  ==> card (SIGMA x: A. B x) = (∑ a∈A. card (B a))
by (simp add: card-def setsum-Sigma del: setsum-constant)

```

```

lemma card-cartesian-product: card (A <*> B) = card(A) * card(B)
apply (cases finite A)
apply (cases finite B)
apply (auto simp add: card-eq-0-iff
  dest: finite-cartesian-productD1 finite-cartesian-productD2)
done

```

```

lemma card-cartesian-product-singleton: card({x} <*> A) = card(A)
by (simp add: card-cartesian-product)

```

### 17.6.4 Cardinality of sums

```

lemma card-Plus:
  assumes finite A and finite B
  shows card (A <+> B) = card A + card B
proof –
  have Inl‘A ∩ Inr‘B = {} by fast
  with assms show ?thesis
    unfolding Plus-def
    by (simp add: card-Un-disjoint card-image)
qed

```

### 17.6.5 Cardinality of the Powerset

```

lemma card-Pow: finite A ==> card (Pow A) = Suc (Suc 0) ^ card A
apply (induct set: finite)
apply (simp-all add: Pow-insert)
apply (subst card-Un-disjoint, blast)
apply (blast intro: finite-imageI, blast)
apply (subgoal-tac inj-on (insert x) (Pow F))
apply (simp add: card-image Pow-insert)
apply (unfold inj-on-def)
apply (blast elim!: equalityE)
done

```

Relates to equivalence classes. Based on a theorem of F. Kammüller.

```

lemma dvd-partition:
  finite (Union C) ==>
    ALL c : C. k dvd card c ==>
      (ALL c1: C. ALL c2: C. c1 ≠ c2 --> c1 Int c2 = {}) ==>
        k dvd card (Union C)
apply(frule finite-UnionD)
apply(rotate-tac -1)
apply (induct set: finite, simp-all, clarify)
apply (subst card-Un-disjoint)
  apply (auto simp add: dvd-add disjoint-eq-subset-Compl)
done

```

### 17.6.6 Relating injectivity and surjectivity

```

lemma finite-surj-inj: finite(A) ==> A <= f'A ==> inj-on f A
apply(rule eq-card-imp-inj-on, assumption)
apply(frule finite-imageI)
apply(drule (1) card-seteq)
  apply(erule card-image-le)
apply simp
done

```

```

lemma finite-UNIV-surj-inj: fixes f :: 'a => 'a
shows finite(UNIV:: 'a set) ==> surj f ==> inj f
by (blast intro: finite-surj-inj subset-UNIV dest:surj-range)

```

```

lemma finite-UNIV-inj-surj: fixes f :: 'a => 'a
shows finite(UNIV:: 'a set) ==> inj f ==> surj f
by(fastsimp simp:surj-def dest!: endo-inj-surj)

```

```

corollary infinite-UNIV-nat: ~finite(UNIV::nat set)
proof
  assume finite(UNIV::nat set)
  with finite-UNIV-inj-surj[of Suc]
  show False by simp (blast dest: Suc-neq-Zero surjD)
qed

```

```

lemma infinite-UNIV-char-0:
  ¬ finite (UNIV::'a::semiring-char-0 set)
proof
  assume finite (UNIV::'a set)
  with subset-UNIV have finite (range of-nat::'a set)
    by (rule finite-subset)
  moreover have inj (of-nat::nat => 'a)
    by (simp add: inj-on-def)
  ultimately have finite (UNIV::nat set)
    by (rule finite-imageD)
  then show False

```

by (*simp add: infinite-UNIV-nat*)  
qed

## 17.7 A fold functional for non-empty sets

Does not require start value.

**inductive**

*fold1Set* :: (*'a* => *'a* => *'a*) => *'a set* => *'a* => *bool*  
for *f* :: *'a* => *'a* => *'a*

**where**

*fold1Set-insertI* [*intro*]:  
[[ *fold-graph f a A x*; *a* ∉ *A* ]] ==> *fold1Set f (insert a A) x*

**constdefs**

*fold1* :: (*'a* => *'a* => *'a*) => *'a set* => *'a*  
*fold1 f A* == *THE x. fold1Set f A x*

**lemma** *fold1Set-nonempty*:

*fold1Set f A x* ==> *A* ≠ {}

by(*erule fold1Set.cases, simp-all*)

**inductive-cases** *empty-fold1SetE* [*elim!*]: *fold1Set f {} x*

**inductive-cases** *insert-fold1SetE* [*elim!*]: *fold1Set f (insert a X) x*

**lemma** *fold1Set-sing* [*iff*]: (*fold1Set f {a} b*) = (*a* = *b*)

by (*blast intro: fold-graph.intros elim: fold-graph.cases*)

**lemma** *fold1-singleton* [*simp*]: *fold1 f {a} = a*

by (*unfold fold1-def*) *blast*

**lemma** *finite-nonempty-imp-fold1Set*:

[[ *finite A*; *A* ≠ {} ]] ==> *EX x. fold1Set f A x*

**apply** (*induct A rule: finite-induct*)

**apply** (*auto dest: finite-imp-fold-graph [of - f]*)

**done**

First, some lemmas about *fold-graph*.

**context** *ab-semigroup-mult*

**begin**

**lemma** *fun-left-comm*: *fun-left-comm*(*op* \*)

by *unfold-locales (simp add: mult-ac)*

**lemma** *fold-graph-insert-swap*:

**assumes** *fold: fold-graph times (b::'a) A y* **and** *b* ∉ *A*

**shows** *fold-graph times z (insert b A) (z \* y)*

**proof** —

```

interpret fun-left-comm op *::'a ⇒ 'a ⇒ 'a by (rule fun-left-comm)
from assms show ?thesis
proof (induct rule: fold-graph.induct)
  case emptyI thus ?case by (force simp add: fold-insert-aux mult-commute)
next
  case (insertI x A y)
    have fold-graph times z (insert x (insert b A)) (x * (z * y))
      using insertI by force — how does id get unfolded?
    thus ?case by (simp add: insert-commute mult-ac)
qed
qed

```

```

lemma fold-graph-permute-diff:
  assumes fold: fold-graph times b A x
  shows !!a. [a ∈ A; b ∉ A] ⇒ fold-graph times a (insert b (A - {a})) x
  using fold
  proof (induct rule: fold-graph.induct)
    case emptyI thus ?case by simp
  next
    case (insertI x A y)
      have a = x ∨ a ∈ A using insertI by simp
      thus ?case
      proof
        assume a = x
        with insertI show ?thesis
          by (simp add: id-def [symmetric], blast intro: fold-graph-insert-swap)
      next
        assume ainA: a ∈ A
        hence fold-graph times a (insert x (insert b (A - {a}))) (x * y)
          using insertI by force
        moreover
        have insert x (insert b (A - {a})) = insert b (insert x A - {a})
          using ainA insertI by blast
        ultimately show ?thesis by simp
      qed
    qed
  qed

```

```

lemma fold1-eq-fold:
  assumes finite A a ∉ A shows fold1 times (insert a A) = fold times a A
  proof -
    interpret fun-left-comm op *::'a ⇒ 'a ⇒ 'a by (rule fun-left-comm)
    from assms show ?thesis
    apply (simp add: fold1-def fold-def)
    apply (rule the-equality)
    apply (best intro: fold-graph-determ theI dest: finite-imp-fold-graph [of - times])
    apply (rule sym, clarify)
    apply (case-tac Aa=A)
    apply (best intro: the-equality fold-graph-determ)
    apply (subgoal-tac fold-graph times a A x)

```



```

  apply (best intro: the-equality fold-graph-determ)
  apply (subgoal-tac insert aa ( $Aa - \{a\} = A$ ))
  prefer 2 apply (blast elim: equalityE)
  apply (auto dest: fold-graph-permute-diff [where a=a])
done
qed

```

```

lemma nonempty-iff: ( $A \neq \{\}$ ) = ( $\exists x B. A = \text{insert } x B \ \& \ x \notin B$ )
  apply safe
  apply simp
  apply (drule-tac x=x in spec)
  apply (drule-tac x=A-{x} in spec, auto)
done

```

```

lemma fold1-insert:
  assumes nonempty:  $A \neq \{\}$  and A: finite A  $x \notin A$ 
  shows fold1 times (insert x A) =  $x * \text{fold1 times } A$ 
  proof -
    interpret fun-left-comm op *::'a  $\Rightarrow$  'a  $\Rightarrow$  'a by (rule fun-left-comm)
    from nonempty obtain a A' where  $A = \text{insert } a A' \ \& \ a \sim: A'$ 
    by (auto simp add: nonempty-iff)
    with A show ?thesis
    by (simp add: insert-commute [of x] fold1-eq-fold eq-commute)
  qed
end

```

```

context ab-semigroup-idem-mult
begin

```

```

lemma fun-left-comm-idem: fun-left-comm-idem(op *)
  apply unfold-locales
  apply (simp add: mult-ac)
  apply (simp add: mult-idem mult-assoc[symmetric])
done

```

```

lemma fold1-insert-idem [simp]:
  assumes nonempty:  $A \neq \{\}$  and A: finite A
  shows fold1 times (insert x A) =  $x * \text{fold1 times } A$ 
  proof -
    interpret fun-left-comm-idem op *::'a  $\Rightarrow$  'a  $\Rightarrow$  'a
    by (rule fun-left-comm-idem)
    from nonempty obtain a A' where  $A' : A = \text{insert } a A' \ \& \ a \sim: A'$ 
    by (auto simp add: nonempty-iff)
    show ?thesis
  proof cases
    assume a = x
    thus ?thesis

```

```

proof cases
  assume  $A' = \{\}$ 
  with prems show ?thesis by (simp add: mult-idem)
next
  assume  $A' \neq \{\}$ 
  with prems show ?thesis
    by (simp add: fold1-insert mult-assoc [symmetric] mult-idem)
qed
next
  assume  $a \neq x$ 
  with prems show ?thesis
    by (simp add: insert-commute fold1-eq-fold fold-insert-idem)
qed
qed

lemma hom-fold1-commute:
assumes hom:  $\forall x y. h (x * y) = h x * h y$ 
and  $N$ : finite  $N$   $N \neq \{\}$  shows  $h (fold1 \text{ times } N) = fold1 \text{ times } (h \text{ ' } N)$ 
using  $N$  proof (induct rule: finite-ne-induct)
  case singleton thus ?case by simp
next
  case (insert  $n$   $N$ )
  then have  $h (fold1 \text{ times } (insert \ n \ N)) = h (n * fold1 \text{ times } N)$  by simp
  also have  $\dots = h \ n * h (fold1 \text{ times } N)$  by (rule hom)
  also have  $h (fold1 \text{ times } N) = fold1 \text{ times } (h \text{ ' } N)$  by (rule insert)
  also have  $times (h \ n) \dots = fold1 \text{ times } (insert (h \ n) (h \text{ ' } N))$ 
    using insert by (simp)
  also have  $insert (h \ n) (h \text{ ' } N) = h \text{ ' } insert \ n \ N$  by simp
  finally show ?case .
qed

end

```

Now the recursion rules for definitions:

```

lemma fold1-singleton-def:  $g = fold1 \ f \implies g \ \{a\} = a$ 
by (simp add: fold1-singleton)

```

```

lemma (in ab-semigroup-mult) fold1-insert-def:
   $\llbracket g = fold1 \text{ times}; \text{finite } A; x \notin A; A \neq \{\} \rrbracket \implies g (insert \ x \ A) = x * g \ A$ 
by (simp add: fold1-insert)

```

```

lemma (in ab-semigroup-idem-mult) fold1-insert-idem-def:
   $\llbracket g = fold1 \text{ times}; \text{finite } A; A \neq \{\} \rrbracket \implies g (insert \ x \ A) = x * g \ A$ 
by simp

```

### 17.7.1 Determinacy for fold1Set

```

declare
  empty-fold-graphE [rule del] fold-graph.intros [rule del]

```

*empty-fold1SetE* [rule *del*] *insert-fold1SetE* [rule *del*]  
 — No more proofs involve these relations.

### 17.7.2 Lemmas about *fold1*

**context** *ab-semigroup-mult*  
**begin**

**lemma** *fold1-Un*:  
**assumes** *A*: *finite A*  $A \neq \{\}$   
**shows**  $\text{finite } B \implies B \neq \{\} \implies A \text{ Int } B = \{\} \implies$   
 $\text{fold1 times } (A \text{ Un } B) = \text{fold1 times } A * \text{fold1 times } B$   
**using** *A* **by** (*induct rule: finite-ne-induct*)  
*(simp-all add: fold1-insert mult-assoc)*

**lemma** *fold1-in*:  
**assumes** *A*: *finite (A)*  $A \neq \{\}$  **and** *elem*:  $\bigwedge x y. x * y \in \{x, y\}$   
**shows**  $\text{fold1 times } A \in A$   
**using** *A*  
**proof** (*induct rule: finite-ne-induct*)  
**case** *singleton* **thus** ?*case* **by** *simp*  
**next**  
**case** *insert* **thus** ?*case* **using** *elem* **by** (*force simp add: fold1-insert*)  
**qed**  
**end**

**lemma** (*in ab-semigroup-idem-mult*) *fold1-Un2*:  
**assumes** *A*: *finite A*  $A \neq \{\}$   
**shows**  $\text{finite } B \implies B \neq \{\} \implies$   
 $\text{fold1 times } (A \text{ Un } B) = \text{fold1 times } A * \text{fold1 times } B$   
**using** *A*  
**proof** (*induct rule: finite-ne-induct*)  
**case** *singleton* **thus** ?*case* **by** *simp*  
**next**  
**case** *insert* **thus** ?*case* **by** (*simp add: mult-assoc*)  
**qed**

### 17.7.3 Fold1 in lattices with *inf* and *sup*

As an application of *fold1* we define infimum and supremum in (not necessarily complete!) lattices over (non-empty) sets by means of *fold1*.

**context** *lower-semilattice*  
**begin**

**lemma** *ab-semigroup-idem-mult-inf*:  
*ab-semigroup-idem-mult inf*  
**proof** **qed** (*rule inf-assoc inf-commute inf-idem*) +

```

lemma below-fold1-iff:
  assumes finite A  $A \neq \{\}$ 
  shows  $x \leq \text{fold1 inf } A \iff (\forall a \in A. x \leq a)$ 
proof –
  interpret ab-semigroup-idem-mult inf
  by (rule ab-semigroup-idem-mult-inf)
  show ?thesis using assms by (induct rule: finite-ne-induct) simp-all
qed

lemma fold1-belowI:
  assumes finite A
  and  $a \in A$ 
  shows  $\text{fold1 inf } A \leq a$ 
proof –
  from assms have  $A \neq \{\}$  by auto
  from  $\langle \text{finite } A \rangle \langle A \neq \{\} \rangle \langle a \in A \rangle$  show ?thesis
  proof (induct rule: finite-ne-induct)
    case singleton thus ?case by simp
  next
    interpret ab-semigroup-idem-mult inf
    by (rule ab-semigroup-idem-mult-inf)
    case (insert x F)
    from insert(5) have  $a = x \vee a \in F$  by simp
    thus ?case
  proof
    assume  $a = x$  thus ?thesis using insert
    by (simp add: mult-ac)
  next
    assume  $a \in F$ 
    hence  $\text{bel: fold1 inf } F \leq a$  by (rule insert)
    have  $\text{inf (fold1 inf (insert x F)) } a = \text{inf } x (\text{inf (fold1 inf } F) a)$ 
    using insert by (simp add: mult-ac)
    also have  $\text{inf (fold1 inf } F) a = \text{fold1 inf } F$ 
    using bel by (auto intro: antisym)
    also have  $\text{inf } x \dots = \text{fold1 inf (insert x F)}$ 
    using insert by (simp add: mult-ac)
    finally have aux:  $\text{inf (fold1 inf (insert x F)) } a = \text{fold1 inf (insert x F)}$  .
    moreover have  $\text{inf (fold1 inf (insert x F)) } a \leq a$  by simp
    ultimately show ?thesis by simp
  qed
qed
qed

end

lemma (in upper-semilattice) ab-semigroup-idem-mult-sup:
  ab-semigroup-idem-mult sup
  by (rule lower-semilattice.ab-semigroup-idem-mult-inf)
  (rule dual-lattice)

```

**context** *lattice*

**begin**

**definition**

*Inf-fin* :: 'a set  $\Rightarrow$  'a ( $\prod_{fin}$  [900] 900)

**where**

*Inf-fin* = *fold1 inf*

**definition**

*Sup-fin* :: 'a set  $\Rightarrow$  'a ( $\sqcup_{fin}$  [900] 900)

**where**

*Sup-fin* = *fold1 sup*

**lemma** *Inf-le-Sup* [*simp*]:  $\llbracket \text{finite } A; A \neq \{\} \rrbracket \Longrightarrow \prod_{fin} A \leq \sqcup_{fin} A$

**apply**(*unfold Sup-fin-def Inf-fin-def*)

**apply**(*subgoal-tac EX a. a:A*)

**prefer** 2 **apply** *blast*

**apply**(*erule exE*)

**apply**(*rule order-trans*)

**apply**(*erule (1) fold1-belowI*)

**apply**(*erule (1) lower-semilattice.fold1-belowI [OF dual-lattice]*)

**done**

**lemma** *sup-Inf-absorb* [*simp*]:

*finite A*  $\Longrightarrow a \in A \Longrightarrow \text{sup } a (\prod_{fin} A) = a$

**apply**(*subst sup-commute*)

**apply**(*simp add: Inf-fin-def sup-absorb2 fold1-belowI*)

**done**

**lemma** *inf-Sup-absorb* [*simp*]:

*finite A*  $\Longrightarrow a \in A \Longrightarrow \text{inf } a (\sqcup_{fin} A) = a$

**by** (*simp add: Sup-fin-def inf-absorb1*

*lower-semilattice.fold1-belowI [OF dual-lattice]*)

**end**

**context** *distrib-lattice*

**begin**

**lemma** *sup-Inf1-distrib*:

**assumes** *finite A*

**and** *A*  $\neq \{\}$

**shows** *sup x* ( $\prod_{fin} A$ ) =  $\prod_{fin} \{\text{sup } x \ a \mid a. a \in A\}$

**proof** –

**interpret** *ab-semigroup-idem-mult inf*

**by** (*rule ab-semigroup-idem-mult-inf*)

**from** *assms* **show** *?thesis*

**by** (*simp add: Inf-fin-def image-def*)

$hom-fold1-commute$  [where  $h=sup\ x$ ,  $OF\ sup-inf-distrib1$ ])  
 (rule  $arg-cong$  [where  $f=fold1\ inf$ ],  $blast$ )  
**qed**

**lemma**  $sup-Inf2-distrib$ :  
 assumes  $A: finite\ A\ A \neq \{\}$  and  $B: finite\ B\ B \neq \{\}$   
 shows  $sup\ (\bigcap_{fin} A)\ (\bigcap_{fin} B) = \bigcap_{fin} \{sup\ a\ b \mid a. a \in A \wedge b \in B\}$   
**using**  $A$  **proof** (induct rule:  $finite-ne-induct$ )  
 case  $singleton$  **thus** ?case  
 by (simp add:  $sup-Inf1-distrib$  [OF  $B$ ]  $fold1-singleton-def$  [OF  $Inf-fin-def$ ])  
**next**  
 interpret  $ab-semigroup-idem-mult\ inf$   
 by (rule  $ab-semigroup-idem-mult-inf$ )  
 case (insert  $x\ A$ )  
 have  $finB: finite\ \{sup\ x\ b \mid b. b \in B\}$   
 by (rule  $finite-surj$  [where  $f = sup\ x$ ,  $OF\ B(1)$ ],  $auto$ )  
 have  $finAB: finite\ \{sup\ a\ b \mid a. a \in A \wedge b \in B\}$   
**proof** –  
 have  $\{sup\ a\ b \mid a. a \in A \wedge b \in B\} = (UN\ a:A. UN\ b:B. \{sup\ a\ b\})$   
 by  $blast$   
 thus ?thesis by (simp add:  $insert(1)\ B(1)$ )  
**qed**  
 have  $ne: \{sup\ a\ b \mid a. a \in A \wedge b \in B\} \neq \{\}$  **using**  $insert\ B$  **by**  $blast$   
 have  $sup\ (\bigcap_{fin} (insert\ x\ A))\ (\bigcap_{fin} B) = sup\ (inf\ x\ (\bigcap_{fin} A))\ (\bigcap_{fin} B)$   
**using**  $insert$  **by** (simp add:  $fold1-insert-idem-def$  [OF  $Inf-fin-def$ ])  
 also have  $\dots = inf\ (sup\ x\ (\bigcap_{fin} B))\ (sup\ (\bigcap_{fin} A)\ (\bigcap_{fin} B))$  **by** (rule  $sup-inf-distrib2$ )  
 also have  $\dots = inf\ (\bigcap_{fin} \{sup\ x\ b \mid b. b \in B\})\ (\bigcap_{fin} \{sup\ a\ b \mid a. a \in A \wedge b \in B\})$   
**using**  $insert$  **by** (simp add:  $sup-Inf1-distrib$  [OF  $B$ ])  
 also have  $\dots = \bigcap_{fin} (\{sup\ x\ b \mid b. b \in B\} \cup \{sup\ a\ b \mid a. a \in A \wedge b \in B\})$   
 (is  $- = \bigcap_{fin} ?M$ )  
**using**  $B\ insert$   
 by (simp add:  $Inf-fin-def\ fold1-Un2$  [OF  $finB - finAB\ ne$ ])  
 also have  $?M = \{sup\ a\ b \mid a. a \in insert\ x\ A \wedge b \in B\}$   
 by  $blast$   
**finally show** ?case .  
**qed**

**lemma**  $inf-Sup1-distrib$ :  
 assumes  $finite\ A$  and  $A \neq \{\}$   
 shows  $inf\ x\ (\bigcup_{fin} A) = \bigcup_{fin} \{inf\ x\ a \mid a. a \in A\}$   
**proof** –  
 interpret  $ab-semigroup-idem-mult\ sup$   
 by (rule  $ab-semigroup-idem-mult-sup$ )  
**from**  $assms$  **show** ?thesis  
 by (simp add:  $Sup-fin-def\ image-def\ hom-fold1-commute$  [where  $h=inf\ x$ ,  $OF\ inf-sup-distrib1$ ])  
 (rule  $arg-cong$  [where  $f=fold1\ sup$ ],  $blast$ )  
**qed**

```

lemma inf-Sup2-distrib:
  assumes A: finite A A  $\neq \{\}$  and B: finite B B  $\neq \{\}$ 
  shows  $\inf (\bigsqcup_{fin} A) (\bigsqcup_{fin} B) = \bigsqcup_{fin} \{ \inf a \ b \mid a \ b. \ a \in A \wedge b \in B \}$ 
using A proof (induct rule: finite-ne-induct)
  case singleton thus ?case
    by (simp add: inf-Sup1-distrib [OF B] fold1-singleton-def [OF Sup-fin-def])
next
  case (insert x A)
  have finB: finite  $\{ \inf x \ b \mid b. \ b \in B \}$ 
    by (rule finite-surj [where f = %b. inf x b, OF B(1)], auto)
  have finAB: finite  $\{ \inf a \ b \mid a \ b. \ a \in A \wedge b \in B \}$ 
proof –
  have  $\{ \inf a \ b \mid a \ b. \ a \in A \wedge b \in B \} = (UN \ a:A. \ UN \ b:B. \ \{ \inf a \ b \})$ 
    by blast
  thus ?thesis by (simp add: insert(1) B(1))
qed
have ne:  $\{ \inf a \ b \mid a \ b. \ a \in A \wedge b \in B \} \neq \{\}$  using insert B by blast
interpret ab-semigroup-idem-mult sup
  by (rule ab-semigroup-idem-mult-sup)
have  $\inf (\bigsqcup_{fin} (\text{insert } x \ A)) (\bigsqcup_{fin} B) = \inf (\sup x (\bigsqcup_{fin} A)) (\bigsqcup_{fin} B)$ 
  using insert by (simp add: fold1-insert-idem-def [OF Sup-fin-def])
also have  $\dots = \sup (\inf x (\bigsqcup_{fin} B)) (\inf (\bigsqcup_{fin} A) (\bigsqcup_{fin} B))$  by (rule inf-sup-distrib2)
also have  $\dots = \sup (\bigsqcup_{fin} \{ \inf x \ b \mid b. \ b \in B \}) (\bigsqcup_{fin} \{ \inf a \ b \mid a \ b. \ a \in A \wedge b \in B \})$ 
using insert by (simp add: inf-Sup1-distrib [OF B])
also have  $\dots = \bigsqcup_{fin} (\{ \inf x \ b \mid b. \ b \in B \} \cup \{ \inf a \ b \mid a \ b. \ a \in A \wedge b \in B \})$ 
  (is - =  $\bigsqcup_{fin} ?M$ )
  using B insert
  by (simp add: Sup-fin-def fold1-Un2 [OF finB - finAB ne])
also have  $?M = \{ \inf a \ b \mid a \ b. \ a \in \text{insert } x \ A \wedge b \in B \}$ 
  by blast
finally show ?case .
qed

```

**end**

**context** *complete-lattice*

**begin**

Coincidence on finite sets in complete lattices:

```

lemma Inf-fin-Inf:
  assumes finite A and A  $\neq \{\}$ 
  shows  $\sqcap_{fin} A = \text{Inf } A$ 
proof –
  interpret ab-semigroup-idem-mult inf
  by (rule ab-semigroup-idem-mult-inf)
from assms show ?thesis
unfolding Inf-fin-def by (induct A set: finite)

```

```

      (simp-all add: Inf-insert-simp)
qed

lemma Sup-fin-Sup:
  assumes finite A and A ≠ {}
  shows  $\bigsqcup_{fin} A = Sup A$ 
proof -
  interpret ab-semigroup-idem-mult sup
  by (rule ab-semigroup-idem-mult-sup)
  from assms show ?thesis
  unfolding Sup-fin-def by (induct A set: finite)
    (simp-all add: Sup-insert-simp)
qed

end

```

#### 17.7.4 Fold1 in linear orders with *min* and *max*

As an application of *fold1* we define minimum and maximum in (not necessarily complete!) linear orders over (non-empty) sets by means of *fold1*.

```

context linorder
begin

```

```

lemma ab-semigroup-idem-mult-min:
  ab-semigroup-idem-mult min
  proof qed (auto simp add: min-def)

```

```

lemma ab-semigroup-idem-mult-max:
  ab-semigroup-idem-mult max
  proof qed (auto simp add: max-def)

```

```

lemma min-lattice:
  lower-semilattice (op ≤) (op <) min
  proof qed (auto simp add: min-def)

```

```

lemma max-lattice:
  lower-semilattice (op ≥) (op >) max
  proof qed (auto simp add: max-def)

```

```

lemma dual-max:
  ord.max (op ≥) = min
  by (auto simp add: ord.max-def-raw min-def-raw expand-fun-eq)

```

```

lemma dual-min:
  ord.min (op ≥) = max
  by (auto simp add: ord.min-def-raw max-def-raw expand-fun-eq)

```

```

lemma strict-below-fold1-iff:
  assumes finite A and A ≠ {}

```



shows  $x < \text{fold1 min } A \longleftrightarrow (\forall a \in A. x < a)$   
**proof** –  
 interpret *ab-semigroup-idem-mult min*  
 by (rule *ab-semigroup-idem-mult-min*)  
 from *assms* show ?thesis  
 by (induct rule: *finite-ne-induct*)  
 (simp-all add: *fold1-insert*)  
**qed**

**lemma** *fold1-below-iff*:  
 assumes *finite A* and  $A \neq \{\}$   
 shows  $\text{fold1 min } A \leq x \longleftrightarrow (\exists a \in A. a \leq x)$   
**proof** –  
 interpret *ab-semigroup-idem-mult min*  
 by (rule *ab-semigroup-idem-mult-min*)  
 from *assms* show ?thesis  
 by (induct rule: *finite-ne-induct*)  
 (simp-all add: *fold1-insert min-le-iff-disj*)  
**qed**

**lemma** *fold1-strict-below-iff*:  
 assumes *finite A* and  $A \neq \{\}$   
 shows  $\text{fold1 min } A < x \longleftrightarrow (\exists a \in A. a < x)$   
**proof** –  
 interpret *ab-semigroup-idem-mult min*  
 by (rule *ab-semigroup-idem-mult-min*)  
 from *assms* show ?thesis  
 by (induct rule: *finite-ne-induct*)  
 (simp-all add: *fold1-insert min-less-iff-disj*)  
**qed**

**lemma** *fold1-antimono*:  
 assumes  $A \neq \{\}$  and  $A \subseteq B$  and *finite B*  
 shows  $\text{fold1 min } B \leq \text{fold1 min } A$   
**proof** *cases*  
 assume  $A = B$  thus ?thesis by *simp*  
**next**  
 interpret *ab-semigroup-idem-mult min*  
 by (rule *ab-semigroup-idem-mult-min*)  
 assume  $A \neq B$   
 have  $B: B = A \cup (B - A)$  using  $\langle A \subseteq B \rangle$  by *blast*  
 have  $\text{fold1 min } B = \text{fold1 min } (A \cup (B - A))$  by (subst *B*) (rule *refl*)  
 also have  $\dots = \min (\text{fold1 min } A) (\text{fold1 min } (B - A))$   
**proof** –  
 have *finite A* by (rule *finite-subset* [OF  $\langle A \subseteq B \rangle$   $\langle \text{finite } B \rangle$ ])  
 moreover have *finite*  $(B - A)$  by (rule *finite-Diff* [OF  $\langle \text{finite } B \rangle$ ])  
 moreover have  $(B - A) \neq \{\}$  using *prems* by *blast*  
 moreover have  $A \text{ Int } (B - A) = \{\}$  using *prems* by *blast*  
 ultimately show ?thesis using  $\langle A \neq \{\} \rangle$  by (rule-tac *fold1-Un*)

qed  
 also have  $\dots \leq \text{fold1 min } A$  by (simp add: min-le-iff-disj)  
 finally show ?thesis .  
 qed

**definition**  
 $\text{Min} :: 'a \text{ set} \Rightarrow 'a$   
**where**  
 $\text{Min} = \text{fold1 min}$

**definition**  
 $\text{Max} :: 'a \text{ set} \Rightarrow 'a$   
**where**  
 $\text{Max} = \text{fold1 max}$

**lemmas**  $\text{Min-singleton [simp]} = \text{fold1-singleton-def [OF Min-def]}$   
**lemmas**  $\text{Max-singleton [simp]} = \text{fold1-singleton-def [OF Max-def]}$

**lemma**  $\text{Min-insert [simp]}$ :  
 assumes  $\text{finite } A$  and  $A \neq \{\}$   
 shows  $\text{Min (insert } x \text{ } A) = \text{min } x \text{ (Min } A)$   
**proof** –  
 interpret  $\text{ab-semigroup-idem-mult min}$   
 by (rule  $\text{ab-semigroup-idem-mult-min}$ )  
 from  $\text{assms}$  show ?thesis by (rule  $\text{fold1-insert-idem-def [OF Min-def]}$ )  
 qed

**lemma**  $\text{Max-insert [simp]}$ :  
 assumes  $\text{finite } A$  and  $A \neq \{\}$   
 shows  $\text{Max (insert } x \text{ } A) = \text{max } x \text{ (Max } A)$   
**proof** –  
 interpret  $\text{ab-semigroup-idem-mult max}$   
 by (rule  $\text{ab-semigroup-idem-mult-max}$ )  
 from  $\text{assms}$  show ?thesis by (rule  $\text{fold1-insert-idem-def [OF Max-def]}$ )  
 qed

**lemma**  $\text{Min-in [simp]}$ :  
 assumes  $\text{finite } A$  and  $A \neq \{\}$   
 shows  $\text{Min } A \in A$   
**proof** –  
 interpret  $\text{ab-semigroup-idem-mult min}$   
 by (rule  $\text{ab-semigroup-idem-mult-min}$ )  
 from  $\text{assms}$   $\text{fold1-in}$  show ?thesis by (fastsimp simp:  $\text{Min-def min-def}$ )  
 qed

**lemma**  $\text{Max-in [simp]}$ :  
 assumes  $\text{finite } A$  and  $A \neq \{\}$   
 shows  $\text{Max } A \in A$   
**proof** –

```

interpret ab-semigroup-idem-mult max
  by (rule ab-semigroup-idem-mult-max)
from assms fold1-in [of A] show ?thesis by (fastsimp simp: Max-def max-def)
qed

```

```

lemma Min-Un:
  assumes finite A and  $A \neq \{\}$  and finite B and  $B \neq \{\}$ 
  shows  $\text{Min } (A \cup B) = \min (\text{Min } A) (\text{Min } B)$ 
proof -
  interpret ab-semigroup-idem-mult min
    by (rule ab-semigroup-idem-mult-min)
  from assms show ?thesis
    by (simp add: Min-def fold1-Un2)
qed

```

```

lemma Max-Un:
  assumes finite A and  $A \neq \{\}$  and finite B and  $B \neq \{\}$ 
  shows  $\text{Max } (A \cup B) = \max (\text{Max } A) (\text{Max } B)$ 
proof -
  interpret ab-semigroup-idem-mult max
    by (rule ab-semigroup-idem-mult-max)
  from assms show ?thesis
    by (simp add: Max-def fold1-Un2)
qed

```

```

lemma hom-Min-commute:
  assumes  $\bigwedge x y. h (\min x y) = \min (h x) (h y)$ 
  and finite N and  $N \neq \{\}$ 
  shows  $h (\text{Min } N) = \text{Min } (h ` N)$ 
proof -
  interpret ab-semigroup-idem-mult min
    by (rule ab-semigroup-idem-mult-min)
  from assms show ?thesis
    by (simp add: Min-def hom-fold1-commute)
qed

```

```

lemma hom-Max-commute:
  assumes  $\bigwedge x y. h (\max x y) = \max (h x) (h y)$ 
  and finite N and  $N \neq \{\}$ 
  shows  $h (\text{Max } N) = \text{Max } (h ` N)$ 
proof -
  interpret ab-semigroup-idem-mult max
    by (rule ab-semigroup-idem-mult-max)
  from assms show ?thesis
    by (simp add: Max-def hom-fold1-commute [of h])
qed

```

```

lemma Min-le [simp]:
  assumes finite A and  $x \in A$ 

```

shows  $\text{Min } A \leq x$   
**proof** –  
 interpret *lower-semilattice*  $op \leq op < min$   
 by (rule *min-lattice*)  
 from *assms* **show** ?thesis **by** (simp add: *Min-def fold1-belowI*)  
**qed**

**lemma** *Max-ge [simp]*:  
 assumes *finite A* and  $x \in A$   
 shows  $x \leq \text{Max } A$   
**proof** –  
 interpret *lower-semilattice*  $op \geq op > max$   
 by (rule *max-lattice*)  
 from *assms* **show** ?thesis **by** (simp add: *Max-def fold1-belowI*)  
**qed**

**lemma** *Min-ge-iff [simp, noatp]*:  
 assumes *finite A* and  $A \neq \{\}$   
 shows  $x \leq \text{Min } A \longleftrightarrow (\forall a \in A. x \leq a)$   
**proof** –  
 interpret *lower-semilattice*  $op \leq op < min$   
 by (rule *min-lattice*)  
 from *assms* **show** ?thesis **by** (simp add: *Min-def below-fold1-iff*)  
**qed**

**lemma** *Max-le-iff [simp, noatp]*:  
 assumes *finite A* and  $A \neq \{\}$   
 shows  $\text{Max } A \leq x \longleftrightarrow (\forall a \in A. a \leq x)$   
**proof** –  
 interpret *lower-semilattice*  $op \geq op > max$   
 by (rule *max-lattice*)  
 from *assms* **show** ?thesis **by** (simp add: *Max-def below-fold1-iff*)  
**qed**

**lemma** *Min-gr-iff [simp, noatp]*:  
 assumes *finite A* and  $A \neq \{\}$   
 shows  $x < \text{Min } A \longleftrightarrow (\forall a \in A. x < a)$   
**proof** –  
 interpret *lower-semilattice*  $op \leq op < min$   
 by (rule *min-lattice*)  
 from *assms* **show** ?thesis **by** (simp add: *Min-def strict-below-fold1-iff*)  
**qed**

**lemma** *Max-less-iff [simp, noatp]*:  
 assumes *finite A* and  $A \neq \{\}$   
 shows  $\text{Max } A < x \longleftrightarrow (\forall a \in A. a < x)$   
**proof** –  
 note  $\text{Max} = \text{Max-def}$   
 interpret *linorder*  $op \geq op >$

```

    by (rule dual-linorder)
  from assms show ?thesis
    by (simp add: Max strict-below-fold1-iff [folded dual-max])
qed

```

```

lemma Min-le-iff [noatp]:
  assumes finite A and A ≠ {}
  shows Min A ≤ x ⟷ (∃ a ∈ A. a ≤ x)
proof -
  interpret lower-semilattice op ≤ op < min
    by (rule min-lattice)
  from assms show ?thesis
    by (simp add: Min-def fold1-below-iff)
qed

```

```

lemma Max-ge-iff [noatp]:
  assumes finite A and A ≠ {}
  shows x ≤ Max A ⟷ (∃ a ∈ A. x ≤ a)
proof -
  note Max = Max-def
  interpret linorder op ≥ op >
    by (rule dual-linorder)
  from assms show ?thesis
    by (simp add: Max fold1-below-iff [folded dual-max])
qed

```

```

lemma Min-less-iff [noatp]:
  assumes finite A and A ≠ {}
  shows Min A < x ⟷ (∃ a ∈ A. a < x)
proof -
  interpret lower-semilattice op ≤ op < min
    by (rule min-lattice)
  from assms show ?thesis
    by (simp add: Min-def fold1-strict-below-iff)
qed

```

```

lemma Max-gr-iff [noatp]:
  assumes finite A and A ≠ {}
  shows x < Max A ⟷ (∃ a ∈ A. x < a)
proof -
  note Max = Max-def
  interpret linorder op ≥ op >
    by (rule dual-linorder)
  from assms show ?thesis
    by (simp add: Max fold1-strict-below-iff [folded dual-max])
qed

```

```

lemma Min-eqI:
  assumes finite A

```

```

  assumes  $\bigwedge y. y \in A \implies y \geq x$ 
    and  $x \in A$ 
  shows  $\text{Min } A = x$ 
proof (rule antisym)
  from  $\langle x \in A \rangle$  have  $A \neq \{\}$  by auto
  with assms show  $\text{Min } A \geq x$  by simp
next
  from assms show  $x \geq \text{Min } A$  by simp
qed

```

```

lemma Max-eqI:
  assumes finite A
  assumes  $\bigwedge y. y \in A \implies y \leq x$ 
    and  $x \in A$ 
  shows  $\text{Max } A = x$ 
proof (rule antisym)
  from  $\langle x \in A \rangle$  have  $A \neq \{\}$  by auto
  with assms show  $\text{Max } A \leq x$  by simp
next
  from assms show  $x \leq \text{Max } A$  by simp
qed

```

```

lemma Min-antimono:
  assumes  $M \subseteq N$  and  $M \neq \{\}$  and finite N
  shows  $\text{Min } N \leq \text{Min } M$ 
proof -
  interpret distrib-lattice op  $\leq$  op  $<$  min max
    by (rule distrib-lattice-min-max)
  from assms show ?thesis by (simp add: Min-def fold1-antimono)
qed

```

```

lemma Max-mono:
  assumes  $M \subseteq N$  and  $M \neq \{\}$  and finite N
  shows  $\text{Max } M \leq \text{Max } N$ 
proof -
  note Max = Max-def
  interpret linorder op  $\geq$  op  $>$ 
    by (rule dual-linorder)
  from assms show ?thesis
    by (simp add: Max fold1-antimono [folded dual-max])
qed

```

```

lemma finite-linorder-induct[consumes 1, case-names empty insert]:
  finite A  $\implies P \{\} \implies$ 
    (!A b. finite A  $\implies \text{ALL } a:A. a < b \implies P A \implies P(\text{insert } b A)$ )
     $\implies P A$ 
proof (induct A rule: measure-induct-rule[where f=card])
  fix A :: 'a set
  assume IH: !! B. card B < card A  $\implies$  finite B  $\implies P \{\} \implies$ 

```

```

      (!!A b. finite A  $\implies$  ( $\forall a \in A. a < b$ )  $\implies$  P A  $\implies$  P (insert b A))
       $\implies$  P B
    and finite A and P {}
    and step: !!A b.  $\llbracket$ finite A;  $\forall a \in A. a < b$ ; P A $\rrbracket \implies$  P (insert b A)
    show P A
    proof (cases A = {})
      assume A = {} thus P A using <P {}> by simp
    next
      let ?B = A - {Max A} let ?A = insert (Max A) ?B
      assume A  $\neq$  {}
      with <finite A> have Max A : A by auto
      hence A: ?A = A using insert-Diff-single insert-absorb by auto
      note card-Diff1-less[OF <finite A> <Max A : A>]
      moreover have finite ?B using <finite A> by simp
      ultimately have P ?B using <P {}> step IH by blast
      moreover have  $\forall a \in ?B. a < \text{Max } A$ 
        using Max-ge [OF <finite A>] by fastsimp
      ultimately show P A
        using A insert-Diff-single step[OF <finite ?B>] by fastsimp
    qed
  qed
end

```

```

context ordered-ab-semigroup-add
begin

```

```

lemma add-Min-commute:
  fixes k
  assumes finite N and N  $\neq$  {}
  shows k + Min N = Min {k + m | m. m  $\in$  N}
proof -
  have  $\bigwedge x y. k + \min x y = \min (k + x) (k + y)$ 
    by (simp add: min-def not-le)
    (blast intro: antisym less-imp-le add-left-mono)
  with assms show ?thesis
    using hom-Min-commute [of plus k N]
    by simp (blast intro: arg-cong [where f = Min])
qed

```

```

lemma add-Max-commute:
  fixes k
  assumes finite N and N  $\neq$  {}
  shows k + Max N = Max {k + m | m. m  $\in$  N}
proof -
  have  $\bigwedge x y. k + \max x y = \max (k + x) (k + y)$ 
    by (simp add: max-def not-le)
    (blast intro: antisym less-imp-le add-left-mono)
  with assms show ?thesis

```

```

    using hom-Max-commute [of plus k N]
    by simp (blast intro: arg-cong [where f = Max])
qed

end

end

```

## 18 Relation: Relations

```

theory Relation
imports Datatype Finite-Set
begin

```

### 18.1 Definitions

#### definition

```

converse :: ('a * 'b) set => ('b * 'a) set
(( $\hat{\phantom{x}}$  - 1) [1000] 999) where
 $\hat{r} - 1 == \{(y, x). (x, y) : r\}$ 

```

#### notation (*xsymbols*)

```

converse (( $\hat{\phantom{x}}$  - 1) [1000] 999)

```

#### definition

```

rel-comp :: [('b * 'c) set, ('a * 'b) set] => ('a * 'c) set
(infixr 0 75) where
 $r \ O \ s == \{(x, z). \exists y. (x, y) : s \ \& \ (y, z) : r\}$ 

```

#### definition

```

Image :: [('a * 'b) set, 'a set] => 'b set
(infixl “ 90) where
 $r \ “ \ s == \{y. \exists x:s. (x, y):r\}$ 

```

#### definition

```

Id :: ('a * 'a) set where — the identity relation
Id == {p.  $\exists x. p = (x, x)$ }

```

#### definition

```

Id-on :: 'a set => ('a * 'a) set where — diagonal: identity over a set
Id-on A ==  $\bigcup_{x \in A}. \{(x, x)\}$ 

```

#### definition

```

Domain :: ('a * 'b) set => 'a set where
Domain r == {x.  $\exists y. (x, y):r$ }

```

#### definition

```

Range :: ('a * 'b) set => 'b set where

```



$\text{Range } r == \text{Domain}(r^{-1})$

**definition**

$\text{Field} :: ('a * 'a) \text{ set} \Rightarrow 'a \text{ set} \text{ where}$   
 $\text{Field } r == \text{Domain } r \cup \text{Range } r$

**definition**

$\text{refl-on} :: ['a \text{ set}, ('a * 'a) \text{ set}] \Rightarrow \text{bool} \text{ where}$  — reflexivity over a set  
 $\text{refl-on } A \ r == r \subseteq A \times A \ \& \ (\text{ALL } x: A. (x, x) : r)$

**abbreviation**

$\text{refl} :: ('a * 'a) \text{ set} \Rightarrow \text{bool} \text{ where}$  — reflexivity over a type  
 $\text{refl} == \text{refl-on UNIV}$

**definition**

$\text{sym} :: ('a * 'a) \text{ set} \Rightarrow \text{bool} \text{ where}$  — symmetry predicate  
 $\text{sym } r == \text{ALL } x \ y. (x, y) : r \longrightarrow (y, x) : r$

**definition**

$\text{antisym} :: ('a * 'a) \text{ set} \Rightarrow \text{bool} \text{ where}$  — antisymmetry predicate  
 $\text{antisym } r == \text{ALL } x \ y. (x, y) : r \longrightarrow (y, x) : r \longrightarrow x = y$

**definition**

$\text{trans} :: ('a * 'a) \text{ set} \Rightarrow \text{bool} \text{ where}$  — transitivity predicate  
 $\text{trans } r == (\text{ALL } x \ y \ z. (x, y) : r \longrightarrow (y, z) : r \longrightarrow (x, z) : r)$

**definition**

$\text{irrefl} :: ('a * 'a) \text{ set} \Rightarrow \text{bool} \text{ where}$   
 $\text{irrefl } r \equiv \forall x. (x, x) \notin r$

**definition**

$\text{total-on} :: 'a \text{ set} \Rightarrow ('a * 'a) \text{ set} \Rightarrow \text{bool} \text{ where}$   
 $\text{total-on } A \ r \equiv \forall x \in A. \forall y \in A. x \neq y \longrightarrow (x, y) \in r \vee (y, x) \in r$

**abbreviation**  $\text{total} \equiv \text{total-on UNIV}$

**definition**

$\text{single-valued} :: ('a * 'b) \text{ set} \Rightarrow \text{bool} \text{ where}$   
 $\text{single-valued } r == \text{ALL } x \ y. (x, y) : r \longrightarrow (\text{ALL } z. (x, z) : r \longrightarrow y = z)$

**definition**

$\text{inv-image} :: ('b * 'b) \text{ set} \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a * 'a) \text{ set} \text{ where}$   
 $\text{inv-image } r \ f == \{(x, y). (f \ x, f \ y) : r\}$

## 18.2 The identity relation

**lemma**  $\text{IdI}$  [intro]:  $(a, a) : \text{Id}$   
**by** (*simp add: Id-def*)

**lemma** *IdE* [*elim!*]:  $p : Id \implies (!!x. p = (x, x) \implies P) \implies P$   
**by** (*unfold Id-def*) (*iprover elim: CollectE*)

**lemma** *pair-in-Id-conv* [*iff*]:  $((a, b) : Id) = (a = b)$   
**by** (*unfold Id-def*) *blast*

**lemma** *refl-Id*: *refl Id*  
**by** (*simp add: refl-on-def*)

**lemma** *antisym-Id*: *antisym Id*  
 — A strange result, since *Id* is also symmetric.  
**by** (*simp add: antisym-def*)

**lemma** *sym-Id*: *sym Id*  
**by** (*simp add: sym-def*)

**lemma** *trans-Id*: *trans Id*  
**by** (*simp add: trans-def*)

### 18.3 Diagonal: identity over a set

**lemma** *Id-on-empty* [*simp*]:  $Id-on \{\} = \{\}$   
**by** (*simp add: Id-on-def*)

**lemma** *Id-on-eqI*:  $a = b \implies a : A \implies (a, b) : Id-on A$   
**by** (*simp add: Id-on-def*)

**lemma** *Id-onI* [*intro!, noatp*]:  $a : A \implies (a, a) : Id-on A$   
**by** (*rule Id-on-eqI*) (*rule refl*)

**lemma** *Id-onE* [*elim!*]:  
 $c : Id-on A \implies (!!x. x : A \implies c = (x, x) \implies P) \implies P$   
 — The general elimination rule.  
**by** (*unfold Id-on-def*) (*iprover elim!: UN-E singletonE*)

**lemma** *Id-on-iff*:  $((x, y) : Id-on A) = (x = y \ \& \ x : A)$   
**by** *blast*

**lemma** *Id-on-subset-Times*:  $Id-on A \subseteq A \times A$   
**by** *blast*

### 18.4 Composition of two relations

**lemma** *rel-compI* [*intro*]:  
 $(a, b) : s \implies (b, c) : r \implies (a, c) : r \ O \ s$   
**by** (*unfold rel-comp-def*) *blast*

**lemma** *rel-compE* [*elim!*]:  $xz : r \ O \ s \implies$   
 $(!!x \ y \ z. xz = (x, z) \implies (x, y) : s \implies (y, z) : r \implies P) \implies P$   
**by** (*unfold rel-comp-def*) (*iprover elim!: CollectE splitE exE conjE*)

**lemma** *rel-compEpair*:

$(a, c) : r \ O \ s \implies (!y. (a, y) : s \implies (y, c) : r \implies P) \implies P$   
**by** (*iprover elim: rel-compE Pair-inject ssubst*)

**lemma** *R-O-Id [simp]*:  $R \ O \ Id = R$   
**by** *fast*

**lemma** *Id-O-R [simp]*:  $Id \ O \ R = R$   
**by** *fast*

**lemma** *rel-comp-empty1 [simp]*:  $\{\} \ O \ R = \{\}$   
**by** *blast*

**lemma** *rel-comp-empty2 [simp]*:  $R \ O \ \{\} = \{\}$   
**by** *blast*

**lemma** *O-assoc*:  $(R \ O \ S) \ O \ T = R \ O \ (S \ O \ T)$   
**by** *blast*

**lemma** *trans-O-subset*:  $\text{trans } r \implies r \ O \ r \subseteq r$   
**by** (*unfold trans-def*) *blast*

**lemma** *rel-comp-mono*:  $r' \subseteq r \implies s' \subseteq s \implies (r' \ O \ s') \subseteq (r \ O \ s)$   
**by** *blast*

**lemma** *rel-comp-subset-Sigma*:

$s \subseteq A \times B \implies r \subseteq B \times C \implies (r \ O \ s) \subseteq A \times C$   
**by** *blast*

**lemma** *rel-comp-distrib [simp]*:  $R \ O \ (S \cup T) = (R \ O \ S) \cup (R \ O \ T)$   
**by** *auto*

**lemma** *rel-comp-distrib2 [simp]*:  $(S \cup T) \ O \ R = (S \ O \ R) \cup (T \ O \ R)$   
**by** *auto*

## 18.5 Reflexivity

**lemma** *refl-onI*:  $r \subseteq A \times A \implies (!x. x : A \implies (x, x) : r) \implies \text{refl-on } A \ r$   
**by** (*unfold refl-on-def*) (*iprover intro!: ballI*)

**lemma** *refl-onD*:  $\text{refl-on } A \ r \implies a : A \implies (a, a) : r$   
**by** (*unfold refl-on-def*) *blast*

**lemma** *refl-onD1*:  $\text{refl-on } A \ r \implies (x, y) : r \implies x : A$   
**by** (*unfold refl-on-def*) *blast*

**lemma** *refl-onD2*:  $\text{refl-on } A \ r \implies (x, y) : r \implies y : A$   
**by** (*unfold refl-on-def*) *blast*

**lemma** *refl-on-Int*:  $\text{refl-on } A \ r \implies \text{refl-on } B \ s \implies \text{refl-on } (A \cap B) \ (r \cap s)$   
**by** (*unfold refl-on-def*) *blast*

**lemma** *refl-on-Un*:  $\text{refl-on } A \ r \implies \text{refl-on } B \ s \implies \text{refl-on } (A \cup B) \ (r \cup s)$   
**by** (*unfold refl-on-def*) *blast*

**lemma** *refl-on-INTER*:  
 $\text{ALL } x:S. \text{refl-on } (A \ x) \ (r \ x) \implies \text{refl-on } (\text{INTER } S \ A) \ (\text{INTER } S \ r)$   
**by** (*unfold refl-on-def*) *fast*

**lemma** *refl-on-UNION*:  
 $\text{ALL } x:S. \text{refl-on } (A \ x) \ (r \ x) \implies \text{refl-on } (\text{UNION } S \ A) \ (\text{UNION } S \ r)$   
**by** (*unfold refl-on-def*) *blast*

**lemma** *refl-on-empty[simp]*:  $\text{refl-on } \{\} \ \{\}$   
**by**(*simp add:refl-on-def*)

**lemma** *refl-on-Id-on*:  $\text{refl-on } A \ (\text{Id-on } A)$   
**by** (*rule refl-onI [OF Id-on-subset-Times Id-onI]*)

## 18.6 Antisymmetry

**lemma** *antisymI*:  
 $(!!x \ y. (x, y) : r \implies (y, x) : r \implies x=y) \implies \text{antisym } r$   
**by** (*unfold antisym-def*) *iprover*

**lemma** *antisymD*:  $\text{antisym } r \implies (a, b) : r \implies (b, a) : r \implies a = b$   
**by** (*unfold antisym-def*) *iprover*

**lemma** *antisym-subset*:  $r \subseteq s \implies \text{antisym } s \implies \text{antisym } r$   
**by** (*unfold antisym-def*) *blast*

**lemma** *antisym-empty [simp]*:  $\text{antisym } \{\}$   
**by** (*unfold antisym-def*) *blast*

**lemma** *antisym-Id-on [simp]*:  $\text{antisym } (\text{Id-on } A)$   
**by** (*unfold antisym-def*) *blast*

## 18.7 Symmetry

**lemma** *symI*:  $(!!a \ b. (a, b) : r \implies (b, a) : r) \implies \text{sym } r$   
**by** (*unfold sym-def*) *iprover*

**lemma** *symD*:  $\text{sym } r \implies (a, b) : r \implies (b, a) : r$   
**by** (*unfold sym-def, blast*)

**lemma** *sym-Int*:  $\text{sym } r \implies \text{sym } s \implies \text{sym } (r \cap s)$   
**by** (*fast intro: symI dest: symD*)

**lemma** *sym-Un*:  $\text{sym } r \implies \text{sym } s \implies \text{sym } (r \cup s)$   
**by** (*fast intro: symI dest: symD*)

**lemma** *sym-INTER*:  $\text{ALL } x:S. \text{sym } (r \ x) \implies \text{sym } (\text{INTER } S \ r)$   
**by** (*fast intro: symI dest: symD*)

**lemma** *sym-UNION*:  $\text{ALL } x:S. \text{sym } (r \ x) \implies \text{sym } (\text{UNION } S \ r)$   
**by** (*fast intro: symI dest: symD*)

**lemma** *sym-Id-on* [*simp*]:  $\text{sym } (\text{Id-on } A)$   
**by** (*rule symI*) *clarify*

## 18.8 Transitivity

**lemma** *transI*:  
 $(!!x \ y \ z. (x, y) : r \implies (y, z) : r \implies (x, z) : r) \implies \text{trans } r$   
**by** (*unfold trans-def*) *iprover*

**lemma** *transD*:  $\text{trans } r \implies (a, b) : r \implies (b, c) : r \implies (a, c) : r$   
**by** (*unfold trans-def*) *iprover*

**lemma** *trans-Int*:  $\text{trans } r \implies \text{trans } s \implies \text{trans } (r \cap s)$   
**by** (*fast intro: transI elim: transD*)

**lemma** *trans-INTER*:  $\text{ALL } x:S. \text{trans } (r \ x) \implies \text{trans } (\text{INTER } S \ r)$   
**by** (*fast intro: transI elim: transD*)

**lemma** *trans-Id-on* [*simp*]:  $\text{trans } (\text{Id-on } A)$   
**by** (*fast intro: transI elim: transD*)

**lemma** *trans-diff-Id*:  $\text{trans } r \implies \text{antisym } r \implies \text{trans } (r - \text{Id})$   
**unfolding** *antisym-def trans-def* **by** *blast*

## 18.9 Irreflexivity

**lemma** *irrefl-diff-Id* [*simp*]:  $\text{irrefl}(r - \text{Id})$   
**by** (*simp add: irrefl-def*)

## 18.10 Totality

**lemma** *total-on-empty* [*simp*]:  $\text{total-on } \{\} \ r$   
**by** (*simp add: total-on-def*)

**lemma** *total-on-diff-Id* [*simp*]:  $\text{total-on } A \ (r - \text{Id}) = \text{total-on } A \ r$   
**by** (*simp add: total-on-def*)

## 18.11 Converse

**lemma** *converse-iff* [*iff*]:  $((a, b) : r^{-1}) = ((b, a) : r)$   
**by** (*simp add: converse-def*)

**lemma** *converseI[sym]*:  $(a, b) : r \implies (b, a) : r^{-1}$   
**by** (*simp add: converse-def*)

**lemma** *converseD[sym]*:  $(a, b) : r^{-1} \implies (b, a) : r$   
**by** (*simp add: converse-def*)

**lemma** *converseE [elim!]*:  
 $yx : r^{-1} \implies (!x y. yx = (y, x) \implies (x, y) : r \implies P) \implies P$   
 — More general than *converseD*, as it “splits” the member of the relation.  
**by** (*unfold converse-def*) (*iprover elim!: CollectE splitE bexE*)

**lemma** *converse-converse [simp]*:  $(r^{-1})^{-1} = r$   
**by** (*unfold converse-def*) *blast*

**lemma** *converse-rel-comp*:  $(r \circ s)^{-1} = s^{-1} \circ r^{-1}$   
**by** *blast*

**lemma** *converse-Int*:  $(r \cap s)^{-1} = r^{-1} \cap s^{-1}$   
**by** *blast*

**lemma** *converse-Un*:  $(r \cup s)^{-1} = r^{-1} \cup s^{-1}$   
**by** *blast*

**lemma** *converse-INTER*:  $(\text{INTER } S \ r)^{-1} = (\text{INT } x:S. (r \ x)^{-1})$   
**by** *fast*

**lemma** *converse-UNION*:  $(\text{UNION } S \ r)^{-1} = (\text{UN } x:S. (r \ x)^{-1})$   
**by** *blast*

**lemma** *converse-Id [simp]*:  $\text{Id}^{-1} = \text{Id}$   
**by** *blast*

**lemma** *converse-Id-on [simp]*:  $(\text{Id-on } A)^{-1} = \text{Id-on } A$   
**by** *blast*

**lemma** *refl-on-converse [simp]*:  $\text{refl-on } A \ (\text{converse } r) = \text{refl-on } A \ r$   
**by** (*unfold refl-on-def*) *auto*

**lemma** *sym-converse [simp]*:  $\text{sym} \ (\text{converse } r) = \text{sym } r$   
**by** (*unfold sym-def*) *blast*

**lemma** *antisym-converse [simp]*:  $\text{antisym} \ (\text{converse } r) = \text{antisym } r$   
**by** (*unfold antisym-def*) *blast*

**lemma** *trans-converse [simp]*:  $\text{trans} \ (\text{converse } r) = \text{trans } r$   
**by** (*unfold trans-def*) *blast*

**lemma** *sym-conv-converse-eq*:  $\text{sym } r = (r^{-1} = r)$

**by** (*unfold sym-def*) *fast*

**lemma** *sym-Un-converse*:  $\text{sym } (r \cup r^{-1})$   
**by** (*unfold sym-def*) *blast*

**lemma** *sym-Int-converse*:  $\text{sym } (r \cap r^{-1})$   
**by** (*unfold sym-def*) *blast*

**lemma** *total-on-converse*[*simp*]:  $\text{total-on } A \ (r^{-1}) = \text{total-on } A \ r$   
**by** (*auto simp: total-on-def*)

## 18.12 Domain

**declare** *Domain-def* [*noatp*]

**lemma** *Domain-iff*:  $(a : \text{Domain } r) = (\exists y. (a, y) : r)$   
**by** (*unfold Domain-def*) *blast*

**lemma** *DomainI* [*intro*]:  $(a, b) : r \implies a : \text{Domain } r$   
**by** (*iprover intro!: iffD2 [OF Domain-iff]*)

**lemma** *DomainE* [*elim!*]:  
 $a : \text{Domain } r \implies (!y. (a, y) : r \implies P) \implies P$   
**by** (*iprover dest!: iffD1 [OF Domain-iff]*)

**lemma** *Domain-empty* [*simp*]:  $\text{Domain } \{\} = \{\}$   
**by** *blast*

**lemma** *Domain-insert*:  $\text{Domain } (\text{insert } (a, b) \ r) = \text{insert } a \ (\text{Domain } r)$   
**by** *blast*

**lemma** *Domain-Id* [*simp*]:  $\text{Domain } \text{Id} = \text{UNIV}$   
**by** *blast*

**lemma** *Domain-Id-on* [*simp*]:  $\text{Domain } (\text{Id-on } A) = A$   
**by** *blast*

**lemma** *Domain-Un-eq*:  $\text{Domain}(A \cup B) = \text{Domain}(A) \cup \text{Domain}(B)$   
**by** *blast*

**lemma** *Domain-Int-subset*:  $\text{Domain}(A \cap B) \subseteq \text{Domain}(A) \cap \text{Domain}(B)$   
**by** *blast*

**lemma** *Domain-Diff-subset*:  $\text{Domain}(A) - \text{Domain}(B) \subseteq \text{Domain}(A - B)$   
**by** *blast*

**lemma** *Domain-Union*:  $\text{Domain } (\text{Union } S) = (\bigcup A \in S. \text{Domain } A)$   
**by** *blast*

**lemma** *Domain-converse* [simp]:  $\text{Domain}(r^{-1}) = \text{Range } r$   
**by** (auto simp: Range-def)

**lemma** *Domain-mono*:  $r \subseteq s \implies \text{Domain } r \subseteq \text{Domain } s$   
**by** blast

**lemma** *fst-eq-Domain*:  $\text{fst } R = \text{Domain } R$   
**by** (auto intro!: image-eqI)

**lemma** *Domain-dprod* [simp]:  $\text{Domain } (\text{dprod } r \ s) = \text{uprod } (\text{Domain } r) (\text{Domain } s)$   
**by** auto

**lemma** *Domain-dsum* [simp]:  $\text{Domain } (\text{dsum } r \ s) = \text{usum } (\text{Domain } r) (\text{Domain } s)$   
**by** auto

### 18.13 Range

**lemma** *Range-iff*:  $(a : \text{Range } r) = (\exists y. (y, a) : r)$   
**by** (simp add: Domain-def Range-def)

**lemma** *RangeI* [intro]:  $(a, b) : r \implies b : \text{Range } r$   
**by** (unfold Range-def) (iprover intro!: converseI DomainI)

**lemma** *RangeE* [elim!]:  $b : \text{Range } r \implies (!x. (x, b) : r \implies P) \implies P$   
**by** (unfold Range-def) (iprover elim!: DomainE dest!: converseD)

**lemma** *Range-empty* [simp]:  $\text{Range } \{\} = \{\}$   
**by** blast

**lemma** *Range-insert*:  $\text{Range } (\text{insert } (a, b) \ r) = \text{insert } b \ (\text{Range } r)$   
**by** blast

**lemma** *Range-Id* [simp]:  $\text{Range } \text{Id} = \text{UNIV}$   
**by** blast

**lemma** *Range-Id-on* [simp]:  $\text{Range } (\text{Id-on } A) = A$   
**by** auto

**lemma** *Range-Un-eq*:  $\text{Range}(A \cup B) = \text{Range}(A) \cup \text{Range}(B)$   
**by** blast

**lemma** *Range-Int-subset*:  $\text{Range}(A \cap B) \subseteq \text{Range}(A) \cap \text{Range}(B)$   
**by** blast

**lemma** *Range-Diff-subset*:  $\text{Range}(A) - \text{Range}(B) \subseteq \text{Range}(A - B)$   
**by** blast



**lemma** *Range-Union*:  $\text{Range } (\text{Union } S) = (\bigcup A \in S. \text{Range } A)$   
**by** *blast*

**lemma** *Range-converse[simp]*:  $\text{Range}(r^{-1}) = \text{Domain } r$   
**by** *blast*

**lemma** *snd-eq-Range*:  $\text{snd } \text{‘} R = \text{Range } R$   
**by** *(auto intro!: image-eqI)*

### 18.14 Field

**lemma** *mono-Field*:  $r \subseteq s \implies \text{Field } r \subseteq \text{Field } s$   
**by** *(auto simp: Field-def Domain-def Range-def)*

**lemma** *Field-empty[simp]*:  $\text{Field } \{\} = \{\}$   
**by** *(auto simp: Field-def)*

**lemma** *Field-insert[simp]*:  $\text{Field } (\text{insert } (a, b) r) = \{a, b\} \cup \text{Field } r$   
**by** *(auto simp: Field-def)*

**lemma** *Field-Un[simp]*:  $\text{Field } (r \cup s) = \text{Field } r \cup \text{Field } s$   
**by** *(auto simp: Field-def)*

**lemma** *Field-Union[simp]*:  $\text{Field } (\bigcup R) = \bigcup (\text{Field } \text{‘} R)$   
**by** *(auto simp: Field-def)*

**lemma** *Field-converse[simp]*:  $\text{Field}(r^{-1}) = \text{Field } r$   
**by** *(auto simp: Field-def)*

### 18.15 Image of a set under a relation

**declare** *Image-def* [*noatp*]

**lemma** *Image-iff*:  $(b : r \text{‘‘} A) = (\exists x : A. (x, b) : r)$   
**by** *(simp add: Image-def)*

**lemma** *Image-singleton*:  $r \text{‘‘} \{a\} = \{b. (a, b) : r\}$   
**by** *(simp add: Image-def)*

**lemma** *Image-singleton-iff [iff]*:  $(b : r \text{‘‘} \{a\}) = ((a, b) : r)$   
**by** *(rule Image-iff [THEN trans]) simp*

**lemma** *ImageI [intro, noatp]*:  $(a, b) : r \implies a : A \implies b : r \text{‘‘} A$   
**by** *(unfold Image-def) blast*

**lemma** *ImageE [elim!]*:  
 $b : r \text{‘‘} A \implies (!x. (x, b) : r \implies x : A \implies P) \implies P$   
**by** *(unfold Image-def) (iprover elim!: CollectE bexE)*

**lemma** *rev-ImageI*:  $a : A \implies (a, b) : r \implies b : r \text{‘‘} A$

— This version’s more effective when we already have the required  $a$   
**by** *blast*

**lemma** *Image-empty* [*simp*]:  $R “ \{\} = \{\}$   
**by** *blast*

**lemma** *Image-Id* [*simp*]:  $Id “ A = A$   
**by** *blast*

**lemma** *Image-Id-on* [*simp*]:  $Id-on A “ B = A \cap B$   
**by** *blast*

**lemma** *Image-Int-subset*:  $R “ (A \cap B) \subseteq R “ A \cap R “ B$   
**by** *blast*

**lemma** *Image-Int-eq*:  
 $single-valued (converse R) ==> R “ (A \cap B) = R “ A \cap R “ B$   
**by** (*simp add: single-valued-def, blast*)

**lemma** *Image-Un*:  $R “ (A \cup B) = R “ A \cup R “ B$   
**by** *blast*

**lemma** *Un-Image*:  $(R \cup S) “ A = R “ A \cup S “ A$   
**by** *blast*

**lemma** *Image-subset*:  $r \subseteq A \times B ==> r “ C \subseteq B$   
**by** (*iprover intro!: subsetI elim!: ImageE dest!: subsetD SigmaD2*)

**lemma** *Image-eq-UN*:  $r “ B = (\bigcup y \in B. r “ \{y\})$   
 — NOT suitable for rewriting  
**by** *blast*

**lemma** *Image-mono*:  $r' \subseteq r ==> A' \subseteq A ==> (r' “ A') \subseteq (r “ A)$   
**by** *blast*

**lemma** *Image-UN*:  $(r “ (UNION A B)) = (\bigcup x \in A. r “ (B x))$   
**by** *blast*

**lemma** *Image-INT-subset*:  $(r “ INTER A B) \subseteq (\bigcap x \in A. r “ (B x))$   
**by** *blast*

Converse inclusion requires some assumptions

**lemma** *Image-INT-eq*:  
 $[single-valued (r^{-1}); A \neq \{\}] ==> r “ INTER A B = (\bigcap x \in A. r “ B x)$   
**apply** (*rule equalityI*)  
**apply** (*rule Image-INT-subset*)  
**apply** (*simp add: single-valued-def, blast*)  
**done**

**lemma** *Image-subset-eq*:  $(r \text{“} A \subseteq B \text{”} = (A \subseteq - ((r \hat{-} 1) \text{“} (-B)))$   
**by** *blast*

### 18.16 Single valued relations

**lemma** *single-valuedI*:

$ALL\ x\ y.\ (x,y):r \dashrightarrow (ALL\ z.\ (x,z):r \dashrightarrow y=z) \implies \text{single-valued } r$   
**by** (*unfold single-valued-def*)

**lemma** *single-valuedD*:

$\text{single-valued } r \implies (x, y) : r \implies (x, z) : r \implies y = z$   
**by** (*simp add: single-valued-def*)

**lemma** *single-valued-rel-comp*:

$\text{single-valued } r \implies \text{single-valued } s \implies \text{single-valued } (r \circ s)$   
**by** (*unfold single-valued-def*) *blast*

**lemma** *single-valued-subset*:

$r \subseteq s \implies \text{single-valued } s \implies \text{single-valued } r$   
**by** (*unfold single-valued-def*) *blast*

**lemma** *single-valued-Id* [*simp*]: *single-valued Id*

**by** (*unfold single-valued-def*) *blast*

**lemma** *single-valued-Id-on* [*simp*]: *single-valued (Id-on A)*

**by** (*unfold single-valued-def*) *blast*

### 18.17 Graphs given by Collect

**lemma** *Domain-Collect-split* [*simp*]:  $\text{Domain}\{(x,y). P\ x\ y\} = \{x. EX\ y. P\ x\ y\}$

**by** *auto*

**lemma** *Range-Collect-split* [*simp*]:  $\text{Range}\{(x,y). P\ x\ y\} = \{y. EX\ x. P\ x\ y\}$

**by** *auto*

**lemma** *Image-Collect-split* [*simp*]:  $\{(x,y). P\ x\ y\} \text{“} A = \{y. EX\ x:A. P\ x\ y\}$

**by** *auto*

### 18.18 Inverse image

**lemma** *sym-inv-image*:  $\text{sym } r \implies \text{sym } (\text{inv-image } r\ f)$

**by** (*unfold sym-def inv-image-def*) *blast*

**lemma** *trans-inv-image*:  $\text{trans } r \implies \text{trans } (\text{inv-image } r\ f)$

**apply** (*unfold trans-def inv-image-def*)

**apply** (*simp (no-asm)*)

**apply** *blast*

**done**

### 18.19 Finiteness

```

lemma finite-converse [iff]: finite ( $r^{-1}$ ) = finite r
  apply (subgoal-tac  $r^{-1} = (\% (x,y). (y,x)) 'r$ )
  apply simp
  apply (rule iffI)
  apply (erule finite-imageD [unfolded inj-on-def])
  apply (simp split add: split-split)
  apply (erule finite-imageI)
  apply (simp add: converse-def image-def, auto)
  apply (rule beqI)
  prefer 2 apply assumption
  apply simp
done

```

**Finiteness of transitive closure** (Thanks to Sidi Ehmedy)

```

lemma finite-Field: finite r ==> finite (Field r)
  — A finite relation has a finite field (= domain  $\cup$  range).
  apply (induct set: finite)
  apply (auto simp add: Field-def Domain-insert Range-insert)
done

```

### 18.20 Version of *lfp-induct* for binary relations

```

lemmas lfp-induct2 =
  lfp-induct-set [of (a, b), split-format (complete)]

end

```

## 19 Predicate: Predicates as relations and enumerations

```

theory Predicate
imports Inductive Relation
begin

```

```

notation
  inf (infixl  $\sqcap$  70) and
  sup (infixl  $\sqcup$  65) and
  Inf ( $\sqcap$  - [900] 900) and
  Sup ( $\sqcup$  - [900] 900) and
  top ( $\top$ ) and
  bot ( $\perp$ )

```

## 19.1 Predicates as (complete) lattices

### 19.1.1 $op \sqcup$ on $bool$

**lemma** *sup-boolI1*:

$$P \implies P \sqcup Q$$

**by** (*simp add: sup-bool-eq*)

**lemma** *sup-boolI2*:

$$Q \implies P \sqcup Q$$

**by** (*simp add: sup-bool-eq*)

**lemma** *sup-boolE*:

$$P \sqcup Q \implies (P \implies R) \implies (Q \implies R) \implies R$$

**by** (*auto simp add: sup-bool-eq*)

### 19.1.2 Equality and Subsets

**lemma** *pred-equals-eq*:  $((\lambda x. x \in R) = (\lambda x. x \in S)) = (R = S)$

**by** (*simp add: mem-def*)

**lemma** *pred-equals-eq2* [*pred-set-conv*]:  $((\lambda x y. (x, y) \in R) = (\lambda x y. (x, y) \in S)) = (R = S)$

**by** (*simp add: expand-fun-eq mem-def*)

**lemma** *pred-subset-eq*:  $((\lambda x. x \in R) \leq (\lambda x. x \in S)) = (R \leq S)$

**by** (*simp add: mem-def*)

**lemma** *pred-subset-eq2* [*pred-set-conv*]:  $((\lambda x y. (x, y) \in R) \leq (\lambda x y. (x, y) \in S)) = (R \leq S)$

**by** *fast*

### 19.1.3 Top and bottom elements

**lemma** *top1I* [*intro!*]:  $top\ x$

**by** (*simp add: top-fun-eq top-bool-eq*)

**lemma** *top2I* [*intro!*]:  $top\ x\ y$

**by** (*simp add: top-fun-eq top-bool-eq*)

**lemma** *bot1E* [*elim!*]:  $bot\ x \implies P$

**by** (*simp add: bot-fun-eq bot-bool-eq*)

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

**by** (*simp add: bot-fun-eq bot-bool-eq*)

### 19.1.4 The empty set

**lemma** *bot-empty-eq*:  $bot = (\lambda x. x \in \{\})$

**by** (*auto simp add: expand-fun-eq*)

**lemma** *bot-empty-eq2*:  $\text{bot} = (\lambda x y. (x, y) \in \{\})$   
**by** (*auto simp add: expand-fun-eq*)

### 19.1.5 Binary union

**lemma** *sup1-iff* [*simp*]:  $\text{sup } A B x \longleftrightarrow A x \mid B x$   
**by** (*simp add: sup-fun-eq sup-bool-eq*)

**lemma** *sup2-iff* [*simp*]:  $\text{sup } A B x y \longleftrightarrow A x y \mid B x y$   
**by** (*simp add: sup-fun-eq sup-bool-eq*)

**lemma** *sup-Un-eq* [*pred-set-conv*]:  $\text{sup } (\lambda x. x \in R) (\lambda x. x \in S) = (\lambda x. x \in R \cup S)$   
**by** (*simp add: expand-fun-eq*)

**lemma** *sup-Un-eq2* [*pred-set-conv*]:  $\text{sup } (\lambda x y. (x, y) \in R) (\lambda x y. (x, y) \in S) = (\lambda x y. (x, y) \in R \cup S)$   
**by** (*simp add: expand-fun-eq*)

**lemma** *sup1I1* [*elim?*]:  $A x \Longrightarrow \text{sup } A B x$   
**by** *simp*

**lemma** *sup2I1* [*elim?*]:  $A x y \Longrightarrow \text{sup } A B x y$   
**by** *simp*

**lemma** *sup1I2* [*elim?*]:  $B x \Longrightarrow \text{sup } A B x$   
**by** *simp*

**lemma** *sup2I2* [*elim?*]:  $B x y \Longrightarrow \text{sup } A B x y$   
**by** *simp*

Classical introduction rule: no commitment to  $A$  vs  $B$ .

**lemma** *sup1CI* [*intro!*]:  $(\sim B x \Longrightarrow A x) \Longrightarrow \text{sup } A B x$   
**by** *auto*

**lemma** *sup2CI* [*intro!*]:  $(\sim B x y \Longrightarrow A x y) \Longrightarrow \text{sup } A B x y$   
**by** *auto*

**lemma** *sup1E* [*elim!*]:  $\text{sup } A B x \Longrightarrow (A x \Longrightarrow P) \Longrightarrow (B x \Longrightarrow P) \Longrightarrow P$   
**by** *simp iprover*

**lemma** *sup2E* [*elim!*]:  $\text{sup } A B x y \Longrightarrow (A x y \Longrightarrow P) \Longrightarrow (B x y \Longrightarrow P) \Longrightarrow P$   
**by** *simp iprover*

### 19.1.6 Binary intersection

**lemma** *inf1-iff* [*simp*]:  $\text{inf } A B x \longleftrightarrow A x \wedge B x$

**by** (*simp add: inf-fun-eq inf-bool-eq*)

**lemma** *inf2-iff* [*simp*]:  $\inf A B x y \longleftrightarrow A x y \wedge B x y$   
**by** (*simp add: inf-fun-eq inf-bool-eq*)

**lemma** *inf-Int-eq* [*pred-set-conv*]:  $\inf (\lambda x. x \in R) (\lambda x. x \in S) = (\lambda x. x \in R \cap S)$   
**by** (*simp add: expand-fun-eq*)

**lemma** *inf-Int-eq2* [*pred-set-conv*]:  $\inf (\lambda x y. (x, y) \in R) (\lambda x y. (x, y) \in S) =$   
 $(\lambda x y. (x, y) \in R \cap S)$   
**by** (*simp add: expand-fun-eq*)

**lemma** *inf1I* [*intro!*]:  $A x \implies B x \implies \inf A B x$   
**by** *simp*

**lemma** *inf2I* [*intro!*]:  $A x y \implies B x y \implies \inf A B x y$   
**by** *simp*

**lemma** *inf1D1*:  $\inf A B x \implies A x$   
**by** *simp*

**lemma** *inf2D1*:  $\inf A B x y \implies A x y$   
**by** *simp*

**lemma** *inf1D2*:  $\inf A B x \implies B x$   
**by** *simp*

**lemma** *inf2D2*:  $\inf A B x y \implies B x y$   
**by** *simp*

**lemma** *inf1E* [*elim!*]:  $\inf A B x \implies (A x \implies B x \implies P) \implies P$   
**by** *simp*

**lemma** *inf2E* [*elim!*]:  $\inf A B x y \implies (A x y \implies B x y \implies P) \implies P$   
**by** *simp*

### 19.1.7 Unions of families

**lemma** *SUP1-iff* [*simp*]:  $(\text{SUP } x:A. B x) b = (\text{EX } x:A. B x b)$   
**by** (*simp add: SUPR-def Sup-fun-def Sup-bool-def*) *blast*

**lemma** *SUP2-iff* [*simp*]:  $(\text{SUP } x:A. B x) b c = (\text{EX } x:A. B x b c)$   
**by** (*simp add: SUPR-def Sup-fun-def Sup-bool-def*) *blast*

**lemma** *SUP1-I* [*intro*]:  $a : A \implies B a b \implies (\text{SUP } x:A. B x) b$   
**by** *auto*

**lemma** *SUP2-I* [*intro*]:  $a : A \implies B a b c \implies (\text{SUP } x:A. B x) b c$   
**by** *auto*

**lemma** *SUP1-E* [*elim!*]:  $(\text{SUP } x:A. B \ x) \ b \implies (!x. x : A \implies B \ x \ b \implies R) \implies R$   
**by** *auto*

**lemma** *SUP2-E* [*elim!*]:  $(\text{SUP } x:A. B \ x) \ b \ c \implies (!x. x : A \implies B \ x \ b \ c \implies R) \implies R$   
**by** *auto*

**lemma** *SUP-UN-eq*:  $(\text{SUP } i. (\lambda x. x \in r \ i)) = (\lambda x. x \in (\text{UN } i. r \ i))$   
**by** (*simp add: expand-fun-eq*)

**lemma** *SUP-UN-eq2*:  $(\text{SUP } i. (\lambda x \ y. (x, y) \in r \ i)) = (\lambda x \ y. (x, y) \in (\text{UN } i. r \ i))$   
**by** (*simp add: expand-fun-eq*)

### 19.1.8 Intersections of families

**lemma** *INF1-iff* [*simp*]:  $(\text{INF } x:A. B \ x) \ b = (\text{ALL } x:A. B \ x \ b)$   
**by** (*simp add: INFI-def Inf-fun-def Inf-bool-def*) *blast*

**lemma** *INF2-iff* [*simp*]:  $(\text{INF } x:A. B \ x) \ b \ c = (\text{ALL } x:A. B \ x \ b \ c)$   
**by** (*simp add: INFI-def Inf-fun-def Inf-bool-def*) *blast*

**lemma** *INF1-I* [*intro!*]:  $(!x. x : A \implies B \ x \ b) \implies (\text{INF } x:A. B \ x) \ b$   
**by** *auto*

**lemma** *INF2-I* [*intro!*]:  $(!x. x : A \implies B \ x \ b \ c) \implies (\text{INF } x:A. B \ x) \ b \ c$   
**by** *auto*

**lemma** *INF1-D* [*elim*]:  $(\text{INF } x:A. B \ x) \ b \implies a : A \implies B \ a \ b$   
**by** *auto*

**lemma** *INF2-D* [*elim*]:  $(\text{INF } x:A. B \ x) \ b \ c \implies a : A \implies B \ a \ b \ c$   
**by** *auto*

**lemma** *INF1-E* [*elim*]:  $(\text{INF } x:A. B \ x) \ b \implies (B \ a \ b \implies R) \implies (a \sim: A \implies R) \implies R$   
**by** *auto*

**lemma** *INF2-E* [*elim*]:  $(\text{INF } x:A. B \ x) \ b \ c \implies (B \ a \ b \ c \implies R) \implies (a \sim: A \implies R) \implies R$   
**by** *auto*

**lemma** *INF-INT-eq*:  $(\text{INF } i. (\lambda x. x \in r \ i)) = (\lambda x. x \in (\text{INT } i. r \ i))$   
**by** (*simp add: expand-fun-eq*)

**lemma** *INF-INT-eq2*:  $(\text{INF } i. (\lambda x \ y. (x, y) \in r \ i)) = (\lambda x \ y. (x, y) \in (\text{INT } i. r \ i))$   
**by** (*simp add: expand-fun-eq*)



## 19.2 Predicates as relations

### 19.2.1 Composition

**inductive**

*pred-comp* :: [*'b* => *'c* => bool, *'a* => *'b* => bool] => *'a* => *'c* => bool  
(**infixr** *OO* 75)

**for** *r* :: *'b* => *'c* => bool **and** *s* :: *'a* => *'b* => bool

**where**

*pred-compI* [*intro*]: *s a b* ==> *r b c* ==> (*r OO s*) *a c*

**inductive-cases** *pred-compE* [*elim!*]: (*r OO s*) *a c*

**lemma** *pred-comp-rel-comp-eq* [*pred-set-conv*]:

(( $\lambda x y. (x, y) \in r$ ) *OO* ( $\lambda x y. (x, y) \in s$ )) = ( $\lambda x y. (x, y) \in r \ O \ s$ )

**by** (*auto simp add: expand-fun-eq elim: pred-compE*)

### 19.2.2 Converse

**inductive**

*conversep* :: (*'a* => *'b* => bool) => *'b* => *'a* => bool  
(( $\hat{\ }^{-1}$ ) [*1000*] *1000*)

**for** *r* :: *'a* => *'b* => bool

**where**

*conversepI*: *r a b* ==>  $r^{\hat{\ }^{-1}} b a$

**notation** (*xsymbols*)

*conversep* (( $\hat{\ }^{-1}$ ) [*1000*] *1000*)

**lemma** *conversepD*:

**assumes** *ab*:  $r^{\hat{\ }^{-1}} a b$

**shows** *r b a* **using** *ab*

**by** *cases simp*

**lemma** *conversep-iff* [*iff*]:  $r^{\hat{\ }^{-1}} a b = r b a$

**by** (*iprover intro: conversepI dest: conversepD*)

**lemma** *conversep-converse-eq* [*pred-set-conv*]:

( $\lambda x y. (x, y) \in r$ )  $\hat{\ }^{-1} = (\lambda x y. (x, y) \in r^{\hat{\ }^{-1}})$

**by** (*auto simp add: expand-fun-eq*)

**lemma** *conversep-conversep* [*simp*]:  $(r^{\hat{\ }^{-1}})^{\hat{\ }^{-1}} = r$

**by** (*iprover intro: order-antisym conversepI dest: conversepD*)

**lemma** *converse-pred-comp*:  $(r \ OO \ s)^{\hat{\ }^{-1}} = s^{\hat{\ }^{-1}} \ OO \ r^{\hat{\ }^{-1}}$

**by** (*iprover intro: order-antisym conversepI pred-compI*

*elim: pred-compE dest: conversepD*)

**lemma** *converse-meet*:  $(\inf r \ s)^{\hat{\ }^{-1}} = \inf r^{\hat{\ }^{-1}} \ s^{\hat{\ }^{-1}}$

**by** (*simp add: inf-fun-eq inf-bool-eq*)

(*iprover intro: conversepI ext dest: conversepD*)

**lemma** *converse-join*: ( $\sup r s$ )<sup>^--1</sup> =  $\sup r$ <sup>^--1</sup>  $s$ <sup>^--1</sup>  
**by** (*simp add: sup-fun-eq sup-bool-eq*)  
(*iprover intro: conversepI ext dest: conversepD*)

**lemma** *conversep-noteq* [*simp*]: ( $op \sim =$ )<sup>^--1</sup> =  $op \sim =$   
**by** (*auto simp add: expand-fun-eq*)

**lemma** *conversep-eq* [*simp*]: ( $op =$ )<sup>^--1</sup> =  $op =$   
**by** (*auto simp add: expand-fun-eq*)

### 19.2.3 Domain

**inductive**

*DomainP* :: ( $'a \Rightarrow 'b \Rightarrow bool$ )  $\Rightarrow 'a \Rightarrow bool$   
**for**  $r :: 'a \Rightarrow 'b \Rightarrow bool$

**where**

*DomainPI* [*intro*]:  $r a b \Rightarrow DomainP r a$

**inductive-cases** *DomainPE* [*elim!*]: *DomainP*  $r a$

**lemma** *DomainP-Domain-eq* [*pred-set-conv*]: *DomainP* ( $\lambda x y. (x, y) \in r$ ) = ( $\lambda x. x \in Domain r$ )  
**by** (*blast intro!: Orderings.order-antisym predicate1I*)

### 19.2.4 Range

**inductive**

*RangeP* :: ( $'a \Rightarrow 'b \Rightarrow bool$ )  $\Rightarrow 'b \Rightarrow bool$   
**for**  $r :: 'a \Rightarrow 'b \Rightarrow bool$

**where**

*RangePI* [*intro*]:  $r a b \Rightarrow RangeP r b$

**inductive-cases** *RangePE* [*elim!*]: *RangeP*  $r b$

**lemma** *RangeP-Range-eq* [*pred-set-conv*]: *RangeP* ( $\lambda x y. (x, y) \in r$ ) = ( $\lambda x. x \in Range r$ )  
**by** (*blast intro!: Orderings.order-antisym predicate1I*)

### 19.2.5 Inverse image

**definition**

*inv-imagep* :: ( $'b \Rightarrow 'b \Rightarrow bool$ )  $\Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$  **where**  
*inv-imagep*  $r f == \%x y. r (f x) (f y)$

**lemma** [*pred-set-conv*]: *inv-imagep* ( $\lambda x y. (x, y) \in r$ )  $f = (\lambda x y. (x, y) \in inv-image r f)$   
**by** (*simp add: inv-image-def inv-imagep-def*)

**lemma** *in-inv-imagep* [*simp*]: *inv-imagep* *r* *f* *x* *y* = *r* (*f* *x*) (*f* *y*)  
**by** (*simp* *add*: *inv-imagep-def*)

### 19.2.6 Powerset

**definition** *Powp* :: ('*a*  $\Rightarrow$  *bool*)  $\Rightarrow$  '*a* *set*  $\Rightarrow$  *bool* **where**  
*Powp* *A* ==  $\lambda B. \forall x \in B. A\ x$

**lemma** *Powp-Pow-eq* [*pred-set-conv*]: *Powp* ( $\lambda x. x \in A$ ) = ( $\lambda x. x \in Pow\ A$ )  
**by** (*auto* *simp* *add*: *Powp-def* *expand-fun-eq*)

**lemmas** *Powp-mono* [*mono*] = *Pow-mono* [*to-pred* *pred-subset-eq*]

### 19.2.7 Properties of relations

**abbreviation** *antisymP* :: ('*a*  $\Rightarrow$  '*a*  $\Rightarrow$  *bool*)  $\Rightarrow$  *bool* **where**  
*antisymP* *r* == *antisym* {(*x*, *y*). *r* *x* *y*}

**abbreviation** *transP* :: ('*a*  $\Rightarrow$  '*a*  $\Rightarrow$  *bool*)  $\Rightarrow$  *bool* **where**  
*transP* *r* == *trans* {(*x*, *y*). *r* *x* *y*}

**abbreviation** *single-valuedP* :: ('*a*  $\Rightarrow$  '*b*  $\Rightarrow$  *bool*)  $\Rightarrow$  *bool* **where**  
*single-valuedP* *r* == *single-valued* {(*x*, *y*). *r* *x* *y*}

## 19.3 Predicates as enumerations

### 19.3.1 The type of predicate enumerations (a monad)

**datatype** '*a* *pred* = *Pred* '*a*  $\Rightarrow$  *bool*

**primrec** *eval* :: '*a* *pred*  $\Rightarrow$  '*a*  $\Rightarrow$  *bool* **where**  
*eval-pred*: *eval* (*Pred* *f*) = *f*

**lemma** *Pred-eval* [*simp*]:  
*Pred* (*eval* *x*) = *x*  
**by** (*cases* *x*) *simp*

**lemma** *eval-inject*: *eval* *x* = *eval* *y*  $\longleftrightarrow$  *x* = *y*  
**by** (*cases* *x*) *auto*

**definition** *single* :: '*a*  $\Rightarrow$  '*a* *pred* **where**  
*single* *x* = *Pred* ((*op* =) *x*)

**definition** *bind* :: '*a* *pred*  $\Rightarrow$  ('*a*  $\Rightarrow$  '*b* *pred*)  $\Rightarrow$  '*b* *pred* (**infixl**  $\gg$  = 70) **where**  
*P*  $\gg$  = *f* = *Pred* ( $\lambda x. (\exists y. \text{eval } P\ y \wedge \text{eval } (f\ y)\ x)$ )

**instantiation** *pred* :: (*type*) *complete-lattice*  
**begin**

**definition**

$$P \leq Q \longleftrightarrow \text{eval } P \leq \text{eval } Q$$

**definition**

$$P < Q \longleftrightarrow \text{eval } P < \text{eval } Q$$

**definition**

$$\perp = \text{Pred } \perp$$

**definition**

$$\top = \text{Pred } \top$$

**definition**

$$P \sqcap Q = \text{Pred } (\text{eval } P \sqcap \text{eval } Q)$$

**definition**

$$P \sqcup Q = \text{Pred } (\text{eval } P \sqcup \text{eval } Q)$$

**definition**

$$\prod A = \text{Pred } (\text{INFI } A \text{ eval})$$

**definition**

$$\bigsqcup A = \text{Pred } (\text{SUPR } A \text{ eval})$$

**instance by default**

(*auto simp add: less-eq-pred-def less-pred-def*  
*inf-pred-def sup-pred-def bot-pred-def top-pred-def*  
*Inf-pred-def Sup-pred-def,*  
*auto simp add: le-fun-def less-fun-def le-bool-def less-bool-def*  
*eval-inject mem-def*)

**end**

**lemma bind-bind:**

( $P \gg= Q \gg= R = P \gg= (\lambda x. Q \ x \gg= R)$ )  
**by** (*auto simp add: bind-def expand-fun-eq*)

**lemma bind-single:**

$P \gg= \text{single} = P$   
**by** (*simp add: bind-def single-def*)

**lemma single-bind:**

$\text{single } x \gg= P = P \ x$   
**by** (*simp add: bind-def single-def*)

**lemma bottom-bind:**

$\perp \gg= P = \perp$   
**by** (*auto simp add: bot-pred-def bind-def expand-fun-eq*)

**lemma sup-bind:**

$(P \sqcup Q) \gg= R = P \gg= R \sqcup Q \gg= R$   
**by** (*auto simp add: bind-def sup-pred-def expand-fun-eq*)

**lemma** *Sup-bind*:  $(\sqcup A \gg= f) = \sqcup ((\lambda x. x \gg= f) \text{ ‘ } A)$   
**by** (*auto simp add: bind-def Sup-pred-def expand-fun-eq*)

**lemma** *pred-iffI*:  
**assumes**  $\bigwedge x. \text{eval } A \ x \Longrightarrow \text{eval } B \ x$   
**and**  $\bigwedge x. \text{eval } B \ x \Longrightarrow \text{eval } A \ x$   
**shows**  $A = B$   
**proof** –  
**from** *assms* **have**  $\bigwedge x. \text{eval } A \ x \longleftrightarrow \text{eval } B \ x$  **by** *blast*  
**then show** *?thesis* **by** (*cases A, cases B*) (*simp add: expand-fun-eq*)  
**qed**

**lemma** *singleI*:  $\text{eval } (\text{single } x) \ x$   
**unfolding** *single-def* **by** *simp*

**lemma** *singleI-unit*:  $\text{eval } (\text{single } ()) \ x$   
**by** *simp (rule singleI)*

**lemma** *singleE*:  $\text{eval } (\text{single } x) \ y \Longrightarrow (y = x \Longrightarrow P) \Longrightarrow P$   
**unfolding** *single-def* **by** *simp*

**lemma** *singleE'*:  $\text{eval } (\text{single } x) \ y \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow P$   
**by** (*erule singleE*) *simp*

**lemma** *bindI*:  $\text{eval } P \ x \Longrightarrow \text{eval } (Q \ x) \ y \Longrightarrow \text{eval } (P \gg= Q) \ y$   
**unfolding** *bind-def* **by** *auto*

**lemma** *bindE*:  $\text{eval } (R \gg= Q) \ y \Longrightarrow (\bigwedge x. \text{eval } R \ x \Longrightarrow \text{eval } (Q \ x) \ y \Longrightarrow P) \Longrightarrow P$   
**unfolding** *bind-def* **by** *auto*

**lemma** *botE*:  $\text{eval } \perp \ x \Longrightarrow P$   
**unfolding** *bot-pred-def* **by** *auto*

**lemma** *supI1*:  $\text{eval } A \ x \Longrightarrow \text{eval } (A \sqcup B) \ x$   
**unfolding** *sup-pred-def* **by** *simp*

**lemma** *supI2*:  $\text{eval } B \ x \Longrightarrow \text{eval } (A \sqcup B) \ x$   
**unfolding** *sup-pred-def* **by** *simp*

**lemma** *supE*:  $\text{eval } (A \sqcup B) \ x \Longrightarrow (\text{eval } A \ x \Longrightarrow P) \Longrightarrow (\text{eval } B \ x \Longrightarrow P) \Longrightarrow P$   
**unfolding** *sup-pred-def* **by** *auto*

### 19.3.2 Derived operations

**definition** *if-pred* :: *bool*  $\Rightarrow$  *unit* *pred* **where**

*if-pred-eq*:  $\text{if-pred } b = (\text{if } b \text{ then single } () \text{ else } \perp)$

**definition** *not-pred* ::  $\text{unit pred} \Rightarrow \text{unit pred}$  **where**  
*not-pred-eq*:  $\text{not-pred } P = (\text{if eval } P () \text{ then } \perp \text{ else single } ())$

**lemma** *if-predI*:  $P \Rightarrow \text{eval } (\text{if-pred } P) ()$   
**unfolding** *if-pred-eq* **by** (*auto intro: singleI*)

**lemma** *if-predE*:  $\text{eval } (\text{if-pred } b) x \Rightarrow (b \Rightarrow x = () \Rightarrow P) \Rightarrow P$   
**unfolding** *if-pred-eq* **by** (*cases b*) (*auto elim: botE*)

**lemma** *not-predI*:  $\neg P \Rightarrow \text{eval } (\text{not-pred } (\text{Pred } (\lambda u. P))) ()$   
**unfolding** *not-pred-eq eval-pred* **by** (*auto intro: singleI*)

**lemma** *not-predI'*:  $\neg \text{eval } P () \Rightarrow \text{eval } (\text{not-pred } P) ()$   
**unfolding** *not-pred-eq* **by** (*auto intro: singleI*)

**lemma** *not-predE*:  $\text{eval } (\text{not-pred } (\text{Pred } (\lambda u. P))) x \Rightarrow (\neg P \Rightarrow \text{thesis}) \Rightarrow \text{thesis}$   
**unfolding** *not-pred-eq*  
**by** (*auto split: split-if-asm elim: botE*)

**lemma** *not-predE'*:  $\text{eval } (\text{not-pred } P) x \Rightarrow (\neg \text{eval } P x \Rightarrow \text{thesis}) \Rightarrow \text{thesis}$   
**unfolding** *not-pred-eq*  
**by** (*auto split: split-if-asm elim: botE*)

### 19.3.3 Implementation

**datatype** 'a seq = *Empty* | *Insert* 'a 'a pred | *Join* 'a pred 'a seq

**primrec** *pred-of-seq* :: 'a seq  $\Rightarrow$  'a pred **where**  
*pred-of-seq Empty* =  $\perp$   
| *pred-of-seq (Insert x P)* =  $\text{single } x \sqcup P$   
| *pred-of-seq (Join P xq)* =  $P \sqcup \text{pred-of-seq } xq$

**definition** *Seq* ::  $(\text{unit} \Rightarrow \text{'a seq}) \Rightarrow \text{'a pred}$  **where**  
*Seq f* = *pred-of-seq* (*f* ())

**code-datatype** *Seq*

**primrec** *member* :: 'a seq  $\Rightarrow$  'a  $\Rightarrow$  bool **where**  
*member Empty x*  $\longleftrightarrow$  *False*  
| *member (Insert y P) x*  $\longleftrightarrow$   $x = y \vee \text{eval } P x$   
| *member (Join P xq) x*  $\longleftrightarrow$   $\text{eval } P x \vee \text{member } xq x$

**lemma** *eval-member*:  
*member xq* = *eval (pred-of-seq xq)*

**proof** (*induct xq*)  
**case** *Empty* **show** ?case

```

  by (auto simp add: expand-fun-eq elim: botE)
next
  case Insert show ?case
  by (auto simp add: expand-fun-eq elim: supE singleE intro: supI1 supI2 singleI)
next
  case Join then show ?case
  by (auto simp add: expand-fun-eq elim: supE intro: supI1 supI2)
qed

```

```

lemma eval-code [code]: eval (Seq f) = member (f ())
  unfolding Seq-def by (rule sym, rule eval-member)

```

```

lemma single-code [code]:
  single x = Seq ( $\lambda u. \text{Insert } x \perp$ )
  unfolding Seq-def by simp

```

```

primrec apply :: ('a  $\Rightarrow$  'b Predicate.pred)  $\Rightarrow$  'a seq  $\Rightarrow$  'b seq where
  apply f Empty = Empty
| apply f (Insert x P) = Join (f x) (Join (P  $\gg$  f) Empty)
| apply f (Join P xq) = Join (P  $\gg$  f) (apply f xq)

```

```

lemma apply-bind:
  pred-of-seq (apply f xq) = pred-of-seq xq  $\gg$  f
proof (induct xq)
  case Empty show ?case
  by (simp add: bottom-bind)
next
  case Insert show ?case
  by (simp add: single-bind sup-bind)
next
  case Join then show ?case
  by (simp add: sup-bind)
qed

```

```

lemma bind-code [code]:
  Seq g  $\gg$  f = Seq ( $\lambda u. \text{apply } f \text{ (g ())}$ )
  unfolding Seq-def by (rule sym, rule apply-bind)

```

```

lemma bot-set-code [code]:
   $\perp$  = Seq ( $\lambda u. \text{Empty}$ )
  unfolding Seq-def by simp

```

```

primrec adjunct :: 'a pred  $\Rightarrow$  'a seq  $\Rightarrow$  'a seq where
  adjunct P Empty = Join P Empty
| adjunct P (Insert x Q) = Insert x (Q  $\sqcup$  P)
| adjunct P (Join Q xq) = Join Q (adjunct P xq)

```

```

lemma adjunct-sup:
  pred-of-seq (adjunct P xq) = P  $\sqcup$  pred-of-seq xq

```

```

by (induct xq) (simp-all add: sup-assoc sup-commute sup-left-commute)

lemma sup-code [code]:
  Seq f  $\sqcup$  Seq g = Seq ( $\lambda u.$  case f ())
    of Empty  $\Rightarrow$  g ()
    | Insert x P  $\Rightarrow$  Insert x (P  $\sqcup$  Seq g)
    | Join P xq  $\Rightarrow$  adjunct (Seq g) (Join P xq)
proof (cases f ())
  case Empty
  thus ?thesis
    unfolding Seq-def by (simp add: sup-commute [of  $\perp$ ] sup-bot)
next
  case Insert
  thus ?thesis
    unfolding Seq-def by (simp add: sup-assoc)
next
  case Join
  thus ?thesis
    unfolding Seq-def
    by (simp add: adjunct-sup sup-assoc sup-commute sup-left-commute)
qed

```

```

primrec contained :: 'a seq  $\Rightarrow$  'a pred  $\Rightarrow$  bool where
  contained Empty Q  $\longleftrightarrow$  True
  | contained (Insert x P) Q  $\longleftrightarrow$  eval Q x  $\wedge$  P  $\leq$  Q
  | contained (Join P xq) Q  $\longleftrightarrow$  P  $\leq$  Q  $\wedge$  contained xq Q

```

```

lemma single-less-eq-eval:
  single x  $\leq$  P  $\longleftrightarrow$  eval P x
by (auto simp add: single-def less-eq-pred-def mem-def)

```

```

lemma contained-less-eq:
  contained xq Q  $\longleftrightarrow$  pred-of-seq xq  $\leq$  Q
by (induct xq) (simp-all add: single-less-eq-eval)

```

```

lemma less-eq-pred-code [code]:
  Seq f  $\leq$  Q = (case f ())
    of Empty  $\Rightarrow$  True
    | Insert x P  $\Rightarrow$  eval Q x  $\wedge$  P  $\leq$  Q
    | Join P xq  $\Rightarrow$  P  $\leq$  Q  $\wedge$  contained xq Q)
by (cases f ())
  (simp-all add: Seq-def single-less-eq-eval contained-less-eq)

```

```

lemma eq-pred-code [code]:
  fixes P Q :: 'a::eq pred
  shows eq-class.eq P Q  $\longleftrightarrow$  P  $\leq$  Q  $\wedge$  Q  $\leq$  P
  unfolding eq by auto

```

```

lemma [code]:

```



```

    pred-case  $f P = f \text{ (eval } P)$ 
    by (cases  $P$ ) simp

lemma [code]:
  pred-rec  $f P = f \text{ (eval } P)$ 
  by (cases  $P$ ) simp

no-notation
  inf (infixl  $\sqcap$  70) and
  sup (infixl  $\sqcup$  65) and
  Inf ( $\sqcap$  - [900] 900) and
  Sup ( $\sqcup$  - [900] 900) and
  top ( $\top$ ) and
  bot ( $\perp$ ) and
  bind (infixl  $\gg=$  70)

hide (open) type pred seq
hide (open) const Pred eval single bind if-pred not-pred
  Empty Insert Join Seq member pred-of-seq apply adjunct

end

```

## 20 Transitive-Closure: Reflexive and Transitive closure of a relation

```

theory Transitive-Closure
imports Predicate
uses ~~/src/Provers/trancl.ML
begin

```

*rtrancl* is reflexive/transitive closure, *trancl* is transitive closure, *reflcl* is reflexive closure.

These postfix operators have *maximum priority*, forcing their operands to be atomic.

**inductive-set**

```

  rtrancl :: ('a  $\times$  'a) set  $\Rightarrow$  ('a  $\times$  'a) set (( $\cdot^*$ ) [1000] 999)
  for r :: ('a  $\times$  'a) set

```

**where**

```

  rtrancl-refl [intro!, Pure.intro!, simp]:  $(a, a) : r^*$ 
  | rtrancl-into-rtrancl [Pure.intro]:  $(a, b) : r^* \Rightarrow (b, c) : r \Rightarrow (a, c) : r^*$ 

```

**inductive-set**

```

  trancl :: ('a  $\times$  'a) set  $\Rightarrow$  ('a  $\times$  'a) set (( $\cdot^+$ ) [1000] 999)
  for r :: ('a  $\times$  'a) set

```

**where**

```

  r-into-trancl [intro, Pure.intro]:  $(a, b) : r \Rightarrow (a, b) : r^+$ 
  | trancl-into-trancl [Pure.intro]:  $(a, b) : r^+ \Rightarrow (b, c) : r \Rightarrow (a, c) : r^+$ 

```

**notation**

$rtranc\!lp \ ((-\hat{**}) \ [1000] \ 1000)$  and  
 $tranc\!lp \ ((-\hat{++}) \ [1000] \ 1000)$

**abbreviation**

$reflclp :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool \ ((-\hat{==}) \ [1000] \ 1000)$

**where**

$r\hat{==} == \sup r \ op =$

**abbreviation**

$refcl :: ('a \times 'a) \ set \Rightarrow ('a \times 'a) \ set \ ((-\hat{=}) \ [1000] \ 999)$  **where**  
 $r\hat{=} == r \cup Id$

**notation** (*xsymbols*)

$rtranc\!lp \ ((-\hat{*}) \ [1000] \ 1000)$  and  
 $tranc\!lp \ ((-\hat{++}) \ [1000] \ 1000)$  and  
 $reflclp \ ((-\hat{==}) \ [1000] \ 1000)$  and  
 $rtranc\!l \ ((-\hat{*}) \ [1000] \ 999)$  and  
 $tranc\!l \ ((-\hat{+}) \ [1000] \ 999)$  and  
 $refcl \ ((-\hat{=}) \ [1000] \ 999)$

**notation** (*HTML output*)

$rtranc\!lp \ ((-\hat{*}) \ [1000] \ 1000)$  and  
 $tranc\!lp \ ((-\hat{++}) \ [1000] \ 1000)$  and  
 $reflclp \ ((-\hat{==}) \ [1000] \ 1000)$  and  
 $rtranc\!l \ ((-\hat{*}) \ [1000] \ 999)$  and  
 $tranc\!l \ ((-\hat{+}) \ [1000] \ 999)$  and  
 $refcl \ ((-\hat{=}) \ [1000] \ 999)$

**20.1 Reflexive closure**

**lemma**  $refl\text{-}refcl[simp]$ :  $refl(r\hat{=})$   
**by** ( $simp \ add:refl\text{-}on\text{-}def$ )

**lemma**  $antisym\text{-}refcl[simp]$ :  $antisym(r\hat{=}) = antisym \ r$   
**by** ( $simp \ add:antisym\text{-}def$ )

**lemma**  $trans\text{-}refclI[simp]$ :  $trans \ r \Longrightarrow trans(r\hat{=})$   
**unfolding**  $trans\text{-}def$  **by**  $blast$

**20.2 Reflexive-transitive closure**

**lemma**  $refcl\text{-}set\text{-}eq \ [pred\text{-}set\text{-}conv]$ :  $(\sup (\lambda x \ y. (x, y) \in r) \ op =) = (\lambda x \ y. (x, y) \in r \ Un \ Id)$   
**by** ( $simp \ add: expand\text{-}fun\text{-}eq$ )

**lemma**  $r\text{-}into\text{-}rtranc\!l \ [intro]$ :  $!!p. p \in r \Longrightarrow p \in r\hat{*}$   
 —  $rtranc\!l$  of  $r$  contains  $r$   
**apply** ( $simp \ only: split\text{-}tupled\text{-}all$ )

```

apply (erule rtrancl-refl [THEN rtrancl-into-rtrancl])
done

lemma r-into-rtranclp [intro]:  $r\ x\ y \implies r^{**}\ x\ y$ 
  — rtrancl of r contains r
  by (erule rtranclp.rtrancl-refl [THEN rtranclp.rtrancl-into-rtrancl])

lemma rtranclp-mono:  $r \leq s \implies r^{**} \leq s^{**}$ 
  — monotonicity of rtrancl
  apply (rule predicate2I)
  apply (erule rtranclp.induct)
  apply (rule-tac [2] rtranclp.rtrancl-into-rtrancl, blast+)
done

lemmas rtrancl-mono = rtranclp-mono [to-set]

theorem rtranclp-induct [consumes 1, case-names base step, induct set: rtranclp]:
  assumes  $a: r^{**}\ a\ b$ 
  and cases:  $P\ a\ !!y\ z. [\![\ r^{**}\ a\ y; r\ y\ z; P\ y\ ]\!] \implies P\ z$ 
  shows  $P\ b$ 
proof —
  from  $a$  have  $a = a \dashrightarrow P\ b$ 
  by (induct % $x\ y. x = a \dashrightarrow P\ y\ a\ b$ ) (iprover intro: cases)+
  then show ?thesis by iprover
qed

lemmas rtrancl-induct [induct set: rtrancl] = rtranclp-induct [to-set]

lemmas rtranclp-induct2 =
  rtranclp-induct[of - ( $ax, ay$ ) ( $bx, by$ ), split-rule,
    consumes 1, case-names refl step]

lemmas rtrancl-induct2 =
  rtrancl-induct[of ( $ax, ay$ ) ( $bx, by$ ), split-format (complete),
    consumes 1, case-names refl step]

lemma refl-rtrancl: refl ( $r^{**}$ )
by (unfold refl-on-def) fast

Transitivity of transitive closure.

lemma trans-rtrancl: trans ( $r^{**}$ )
proof (rule transI)
  fix  $x\ y\ z$ 
  assume  $(x, y) \in r^{**}$ 
  assume  $(y, z) \in r^{**}$ 
  then show  $(x, z) \in r^{**}$ 
  proof induct
    case base
    show  $(x, y) \in r^{**}$  by fact

```

```

next
  case (step u v)
  from  $\langle (x, u) \in r^* \rangle$  and  $\langle (u, v) \in r \rangle$ 
  show  $\langle (x, v) \in r^* \rangle$  ..
qed
qed

```

lemmas *rtrancl-trans* = *trans-rtrancl* [THEN *transD*, *standard*]

```

lemma rtranclp-trans:
  assumes xy:  $r^{**} x y$ 
  and yz:  $r^{**} y z$ 
  shows  $r^{**} x z$  using yz xy
  by induct iprover+

```

```

lemma rtranclE [cases set: rtrancl]:
  assumes major:  $(a::'a, b) : r^*$ 
  obtains
    (base)  $a = b$ 
  | (step)  $y$  where  $(a, y) : r^*$  and  $(y, b) : r$ 
  — elimination of rtrancl – by induction on a special formula
  apply (subgoal-tac  $(a::'a) = b \mid (EX y. (a, y) : r^* \ \& \ (y, b) : r)$ )
  apply (rule-tac [2] major [THEN rtrancl-induct])
  prefer 2 apply blast
  prefer 2 apply blast
  apply (erule asm-rl exE disjE conjE base step)+
  done

```

```

lemma rtrancl-Int-subset: [ $Id \subseteq s; r \ O \ (r^* \cap s) \subseteq s$ ] ==>  $r^* \subseteq s$ 
  apply (rule subsetI)
  apply (rule-tac  $p=x$  in PairE, clarify)
  apply (erule rtrancl-induct, auto)
  done

```

```

lemma converse-rtranclp-into-rtranclp:
   $r \ a \ b \implies r^{**} b \ c \implies r^{**} a \ c$ 
  by (rule rtranclp-trans) iprover+

```

lemmas *converse-rtrancl-into-rtrancl* = *converse-rtranclp-into-rtranclp* [to-set]

More  $r^*$  equations and inclusions.

```

lemma rtranclp-idemp [simp]:  $(r^{**})^{**} = r^{**}$ 
  apply (auto intro!: order-antisym)
  apply (erule rtranclp-induct)
  apply (rule rtranclp.rtrancl-refl)
  apply (blast intro: rtranclp-trans)
  done

```

lemmas *rtrancl-idemp* [simp] = *rtranclp-idemp* [to-set]

```

lemma rtrancl-idemp-self-comp [simp]:  $R^{\wedge*} \circ R^{\wedge*} = R^{\wedge*}$ 
  apply (rule set-ext)
  apply (simp only: split-tupled-all)
  apply (blast intro: rtrancl-trans)
  done

```

```

lemma rtrancl-subset-rtrancl:  $r \subseteq s^{\wedge*} \implies r^{\wedge*} \subseteq s^{\wedge*}$ 
  apply (drule rtrancl-mono)
  apply simp
  done

```

```

lemma rtranclp-subset:  $R \leq S \implies S \leq R^{\wedge**} \implies S^{\wedge**} = R^{\wedge**}$ 
  apply (drule rtranclp-mono)
  apply (drule rtranclp-mono)
  apply simp
  done

```

```

lemmas rtrancl-subset = rtranclp-subset [to-set]

```

```

lemma rtranclp-sup-rtranclp:  $(\sup (R^{\wedge**}) (S^{\wedge**}))^{\wedge**} = (\sup R S)^{\wedge**}$ 
  by (blast intro!: rtranclp-subset intro: rtranclp-mono [THEN predicate2D])

```

```

lemmas rtrancl-Un-rtrancl = rtranclp-sup-rtranclp [to-set]

```

```

lemma rtranclp-reflcl [simp]:  $(R^{\wedge==})^{\wedge**} = R^{\wedge**}$ 
  by (blast intro!: rtranclp-subset)

```

```

lemmas rtrancl-reflcl [simp] = rtranclp-reflcl [to-set]

```

```

lemma rtrancl-r-diff-Id:  $(r - Id)^{\wedge*} = r^{\wedge*}$ 
  apply (rule sym)
  apply (rule rtrancl-subset, blast, clarify)
  apply (rename-tac a b)
  apply (case-tac a = b)
  apply blast
  apply (blast intro!: r-into-rtrancl)
  done

```

```

lemma rtranclp-r-diff-Id:  $(\inf r \text{ op } \sim)^{\wedge**} = r^{\wedge**}$ 
  apply (rule sym)
  apply (rule rtranclp-subset)
  apply blast+
  done

```

```

theorem rtranclp-converseD:
  assumes  $r: (r^{\wedge--1})^{\wedge**} x y$ 
  shows  $r^{\wedge**} y x$ 
proof -

```

```

from  $r$  show ?thesis
  by induct (iprover intro: rtranclp-trans dest!: conversepD)+
qed

lemmas rtrancl-converseD = rtranclp-converseD [to-set]

theorem rtranclp-converseI:
  assumes  $r^{**} y x$ 
  shows  $(r^{--1})^{**} x y$ 
  using assms
  by induct (iprover intro: rtranclp-trans conversepI)+

lemmas rtrancl-converseI = rtranclp-converseI [to-set]

lemma rtrancl-converse:  $(r^{--1})^{**} = (r^{**})^{--1}$ 
  by (fast dest!: rtrancl-converseD intro!: rtrancl-converseI)

lemma sym-rtrancl:  $\text{sym } r \implies \text{sym } (r^{**})$ 
  by (simp only: sym-conv-converse-eq rtrancl-converse [symmetric])

theorem converse-rtranclp-induct[consumes 1]:
  assumes major:  $r^{**} a b$ 
  and cases:  $P b !!y z. [r y z; r^{**} z b; P z] \implies P y$ 
  shows  $P a$ 
  using rtranclp-converseI [OF major]
  by induct (iprover intro: cases dest!: conversepD rtranclp-converseD)+

lemmas converse-rtrancl-induct = converse-rtranclp-induct [to-set]

lemmas converse-rtranclp-induct2 =
  converse-rtranclp-induct [of - (ax,ay) (bx,by), split-rule,
    consumes 1, case-names refl step]

lemmas converse-rtrancl-induct2 =
  converse-rtrancl-induct [of (ax,ay) (bx,by), split-format (complete),
    consumes 1, case-names refl step]

lemma converse-rtranclpE:
  assumes major:  $r^{**} x z$ 
  and cases:  $x=z \implies P$ 
  !!y.  $[r x y; r^{**} y z] \implies P$ 
  shows  $P$ 
  apply (subgoal-tac x = z | (EX y. r x y & r^{**} y z))
  apply (rule-tac [2] major [THEN converse-rtranclp-induct])
  prefer 2 apply iprover
  prefer 2 apply iprover
  apply (erule asm-rl exE disjE conjE cases)+
done

```

**lemmas** *converse-rtranclE* = *converse-rtranclpE* [to-set]

**lemmas** *converse-rtranclpE2* = *converse-rtranclpE* [of - (xa,xb) (za,zb), split-rule]

**lemmas** *converse-rtranclE2* = *converse-rtranclE* [of (xa,xb) (za,zb), split-rule]

**lemma** *r-comp-rtrancl-eq*:  $r \circ r^* = r^* \circ r$   
**by** (blast elim: *rtranclE* *converse-rtranclE*  
intro: *rtrancl-into-rtrancl* *converse-rtrancl-into-rtrancl*)

**lemma** *rtrancl-unfold*:  $r^* = Id \cup r \circ r^*$   
**by** (auto intro: *rtrancl-into-rtrancl* elim: *rtranclE*)

### 20.3 Transitive closure

**lemma** *trancl-mono*:  $!!p. p \in r^+ \implies r \subseteq s \implies p \in s^+$   
**apply** (simp add: *split-tupled-all*)  
**apply** (erule *trancl.induct*)  
**apply** (iprover dest: *subsetD*)  
**done**

**lemma** *r-into-trancl'*:  $!!p. p : r \implies p : r^+$   
**by** (simp only: *split-tupled-all*) (erule *r-into-trancl*)

Conversions between *trancl* and *rtrancl*.

**lemma** *tranclp-into-rtranclp*:  $r^{++} a b \implies r^{**} a b$   
**by** (erule *tranclp.induct*) iprover+

**lemmas** *trancl-into-rtrancl* = *tranclp-into-rtranclp* [to-set]

**lemma** *rtranclp-into-tranclp1*: **assumes**  $r: r^{**} a b$   
**shows**  $!!c. r b c \implies r^{++} a c$  **using**  $r$   
**by** *induct* iprover+

**lemmas** *rtrancl-into-trancl1* = *rtranclp-into-tranclp1* [to-set]

**lemma** *rtranclp-into-tranclp2*:  $[| r a b; r^{**} b c |] \implies r^{++} a c$   
— intro rule from  $r$  and *rtrancl*  
**apply** (erule *rtranclp.cases*)  
**apply** iprover  
**apply** (rule *rtranclp-trans* [THEN *rtranclp-into-tranclp1*])  
**apply** (simp | rule *r-into-rtranclp*)  
**done**

**lemmas** *rtrancl-into-trancl2* = *rtranclp-into-tranclp2* [to-set]

Nice induction rule for *trancl*

**lemma** *tranclp-induct* [consumes 1, case-names base step, induct pred: *tranclp*]:  
**assumes**  $r^{++} a b$

```

and cases: !!y. r a y ==> P y
  !!y z. r^++ a y ==> r y z ==> P y ==> P z
shows P b
proof -
  from (r^++ a b) have a = a --> P b
  by (induct %x y. x = a --> P y a b) (iprover intro: cases)+
  then show ?thesis by iprover
qed

```

```

lemmas trancl-induct [induct set: trancl] = tranclp-induct [to-set]

```

```

lemmas tranclp-induct2 =
  tranclp-induct [of - (ax,ay) (bx,by), split-rule,
    consumes 1, case-names base step]

```

```

lemmas trancl-induct2 =
  trancl-induct [of (ax,ay) (bx,by), split-format (complete),
    consumes 1, case-names base step]

```

```

lemma tranclp-trans-induct:
  assumes major: r^++ x y
  and cases: !!x y. r x y ==> P x y
  !!x y z. [| r^++ x y; P x y; r^++ y z; P y z |] ==> P x z
shows P x y
  — Another induction rule for trancl, incorporating transitivity
  by (iprover intro: major [THEN tranclp-induct] cases)

```

```

lemmas trancl-trans-induct = tranclp-trans-induct [to-set]

```

```

lemma tranclE [cases set: trancl]:
  assumes (a, b) : r^+
  obtains
    (base) (a, b) : r
  | (step) c where (a, c) : r^+ and (c, b) : r
  using assms by cases simp-all

```

```

lemma trancl-Int-subset: [| r ⊆ s; r O (r^+ ∩ s) ⊆ s |] ==> r^+ ⊆ s
  apply (rule subsetI)
  apply (rule-tac p = x in PairE)
  apply clarify
  apply (erule trancl-induct)
  apply auto
done

```

```

lemma trancl-unfold: r^+ = r Un r O r^+
  by (auto intro: trancl-into-trancl elim: tranclE)

```

Transitivity of  $r^+$

```

lemma trans-trancl [simp]: trans (r^+)

```



```

proof (rule transI)
  fix x y z
  assume  $(x, y) \in r^+$ 
  assume  $(y, z) \in r^+$ 
  then show  $(x, z) \in r^+$ 
  proof induct
    case (base u)
    from  $\langle (x, y) \in r^+ \rangle$  and  $\langle (y, u) \in r \rangle$ 
    show  $(x, u) \in r^+ ..$ 
  next
    case (step u v)
    from  $\langle (x, u) \in r^+ \rangle$  and  $\langle (u, v) \in r \rangle$ 
    show  $(x, v) \in r^+ ..$ 
  qed
qed

```

**lemmas** trancl-trans = trans-trancl [THEN transD, standard]

```

lemma tranclp-trans:
  assumes xy:  $r^{++} x y$ 
  and yz:  $r^{++} y z$ 
  shows  $r^{++} x z$  using yz xy
  by induct iprover+

```

```

lemma trancl-id [simp]: trans r  $\implies r^+ = r$ 
  apply auto
  apply (erule trancl-induct)
  apply assumption
  apply (unfold trans-def)
  apply blast
  done

```

```

lemma rtranclp-tranclp-tranclp:
  assumes  $r^{**} x y$ 
  shows  $!!z. r^{++} y z \implies r^{++} x z$  using assms
  by induct (iprover intro: tranclp-trans)+

```

**lemmas** rtrancl-trancl-trancl = rtranclp-tranclp-tranclp [to-set]

```

lemma tranclp-into-tranclp2:  $r a b \implies r^{++} b c \implies r^{++} a c$ 
  by (erule tranclp-trans [OF tranclp.r-into-trancl])

```

**lemmas** trancl-into-trancl2 = tranclp-into-tranclp2 [to-set]

```

lemma trancl-insert:
   $(\text{insert } (y, x) r)^+ = r^+ \cup \{(a, b). (a, y) \in r^* \wedge (x, b) \in r^*\}$ 
  — primitive recursion for trancl over finite relations
  apply (rule equalityI)
  apply (rule subsetI)

```

```

apply (simp only: split-tupled-all)
apply (erule trancl-induct, blast)
apply (blast intro: rtrancl-into-trancl1 trancl-into-rtrancl r-into-trancl trancl-trans)
apply (rule subsetI)
apply (blast intro: trancl-mono rtrancl-mono
  [THEN [2] rev-subsetD] rtrancl-trancl-trancl rtrancl-into-trancl2)
done

lemma tranclp-converseI:  $(r^{++})^{--1} x y \implies (r^{--1})^{++} x y$ 
apply (erule conversepD)
apply (erule tranclp-induct)
apply (iprover intro: conversepI tranclp-trans)+
done

lemmas trancl-converseI = tranclp-converseI [to-set]

lemma tranclp-converseD:  $(r^{--1})^{++} x y \implies (r^{++})^{--1} x y$ 
apply (rule conversepI)
apply (erule tranclp-induct)
apply (iprover dest: conversepD intro: tranclp-trans)+
done

lemmas trancl-converseD = tranclp-converseD [to-set]

lemma tranclp-converse:  $(r^{--1})^{++} = (r^{++})^{--1}$ 
by (fastsimp simp add: expand-fun-eq
  intro!: tranclp-converseI dest!: tranclp-converseD)

lemmas trancl-converse = tranclp-converse [to-set]

lemma sym-trancl:  $\text{sym } r \implies \text{sym } (r^{+})$ 
by (simp only: sym-conv-converse-eq trancl-converse [symmetric])

lemma converse-trancl-induct:
assumes major:  $r^{++} a b$ 
and cases:  $!!y. r y b \implies P(y)$ 
!!y z. [ r y z; r^{++} z b; P(z) ] \implies P(y)
shows  $P a$ 
apply (rule tranclp-induct [OF tranclp-converseI, OF conversepI, OF major])
apply (rule cases)
apply (erule conversepD)
apply (blast intro: prems dest!: tranclp-converseD conversepD)
done

lemmas converse-trancl-induct = converse-tranclp-induct [to-set]

lemma tranclpD:  $R^{++} x y \implies \exists x z. R x z \wedge R^{**} z y$ 
apply (erule converse-tranclp-induct)
apply auto

```

```

apply (blast intro: rtranclp-trans)
done

lemmas tranclD = tranclpD [to-set]

lemma tranclD2:
   $(x, y) \in R^+ \implies \exists z. (x, z) \in R^* \wedge (z, y) \in R$ 
by (blast elim: tranclE intro: trancl-into-rtrancl)

lemma irrefl-tranclI:  $r^+-1 \cap r^* = \{\}$   $\implies (x, x) \notin r^+$ 
by (blast elim: tranclE dest: trancl-into-rtrancl)

lemma irrefl-trancl-rD:  $\forall x. (x, x) \notin r^+ \implies (x, y) \in r \implies x \neq y$ 
by (blast dest: r-into-trancl)

lemma trancl-subset-Sigma-aux:
   $(a, b) \in r^* \implies r \subseteq A \times A \implies a = b \vee a \in A$ 
by (induct rule: rtrancl-induct) auto

lemma trancl-subset-Sigma:  $r \subseteq A \times A \implies r^+ \subseteq A \times A$ 
apply (rule subsetI)
apply (simp only: split-tupled-all)
apply (erule tranclE)
apply (blast dest!: trancl-into-rtrancl trancl-subset-Sigma-aux)+
done

lemma reflcl-tranclp [simp]:  $(r^{++})^+ = r^{**}$ 
apply (safe intro!: order-antisym)
apply (erule tranclp-into-rtranclp)
apply (blast elim: rtranclp.cases dest: rtranclp-into-tranclp1)
done

lemmas reflcl-trancl [simp] = reflcl-tranclp [to-set]

lemma trancl-reflcl [simp]:  $(r^+)^+ = r^*$ 
apply safe
apply (drule trancl-into-rtrancl, simp)
apply (erule rtranclE, safe)
apply (rule r-into-trancl, simp)
apply (rule rtrancl-into-trancl1)
apply (erule rtrancl-reflcl [THEN equalityD2, THEN subsetD], fast)
done

lemma trancl-empty [simp]:  $\{\}^+ = \{\}$ 
by (auto elim: trancl-induct)

lemma rtrancl-empty [simp]:  $\{\}^* = Id$ 
by (rule subst [OF reflcl-trancl]) simp

```

**lemma** *rtranclpD*:  $R^{**} a b \implies a = b \vee a \neq b \wedge R^{++} a b$   
**by** (*force simp add: reflcl-tranclp [symmetric] simp del: reflcl-tranclp*)

**lemmas** *rtranclD* = *rtranclpD* [*to-set*]

**lemma** *rtrancl-eq-or-trancl*:  
 $(x,y) \in R^* = (x=y \vee x \neq y \wedge (x,y) \in R^+)$   
**by** (*fast elim: trancl-into-rtrancl dest: rtranclD*)

*Domain and Range*

**lemma** *Domain-rtrancl* [*simp*]:  $\text{Domain } (R^*) = \text{UNIV}$   
**by** *blast*

**lemma** *Range-rtrancl* [*simp*]:  $\text{Range } (R^*) = \text{UNIV}$   
**by** *blast*

**lemma** *rtrancl-Un-subset*:  $(R^* \cup S^*) \subseteq (R \cup S)^*$   
**by** (*rule rtrancl-Un-rtrancl [THEN subst]*) *fast*

**lemma** *in-rtrancl-UnI*:  $x \in R^* \vee x \in S^* \implies x \in (R \cup S)^*$   
**by** (*blast intro: subsetD [OF rtrancl-Un-subset]*)

**lemma** *trancl-domain* [*simp*]:  $\text{Domain } (r^+) = \text{Domain } r$   
**by** (*unfold Domain-def*) (*blast dest: tranclD*)

**lemma** *trancl-range* [*simp*]:  $\text{Range } (r^+) = \text{Range } r$   
**unfolding** *Range-def* **by** (*simp add: trancl-converse [symmetric]*)

**lemma** *Not-Domain-rtrancl*:  
 $x \sim: \text{Domain } R \implies ((x, y) : R^*) = (x = y)$   
**apply** *auto*  
**apply** (*erule rev-mp*)  
**apply** (*erule rtrancl-induct*)  
**apply** *auto*  
**done**

**lemma** *trancl-subset-Field2*:  $r^+ \leq \text{Field } r \times \text{Field } r$   
**apply** *clarify*  
**apply** (*erule trancl-induct*)  
**apply** (*auto simp add: Field-def*)  
**done**

**lemma** *finite-trancl*:  $\text{finite } (r^+) = \text{finite } r$   
**apply** *auto*  
**prefer** 2  
**apply** (*rule trancl-subset-Field2 [THEN finite-subset]*)  
**apply** (*rule finite-SigmaI*)  
**prefer** 3  
**apply** (*blast intro: r-into-trancl' finite-subset*)

```

apply (auto simp add: finite-Field)
done

```

More about converse *rtrancl* and *trancl*, should be merged with main body.

```

lemma single-valued-confluent:
  [| single-valued r; (x,y) ∈ r^*; (x,z) ∈ r^* |]
  ==> (y,z) ∈ r^* ∨ (z,y) ∈ r^*
apply (erule rtrancl-induct)
apply simp
apply (erule disjE)
apply (blast elim: converse-rtranclE dest: single-valuedD)
apply (blast intro: rtrancl-trans)
done

```

```

lemma r-r-into-trancl: (a, b) ∈ R ==> (b, c) ∈ R ==> (a, c) ∈ R^+
by (fast intro: trancl-trans)

```

```

lemma trancl-into-trancl [rule-format]:
  (a, b) ∈ r^+ ==> (b, c) ∈ r --> (a, c) ∈ r^+
apply (erule trancl-induct)
apply (fast intro: r-r-into-trancl)
apply (fast intro: r-r-into-trancl trancl-trans)
done

```

```

lemma tranclp-rtranclp-tranclp:
  r^{++} a b ==> r^{**} b c ==> r^{++} a c
apply (drule tranclpD)
apply (elim exE conjE)
apply (drule rtranclp-trans, assumption)
apply (drule rtranclp-into-tranclp2, assumption, assumption)
done

```

```

lemmas trancl-rtrancl-trancl = tranclp-rtranclp-tranclp [to-set]

```

```

lemmas transitive-closure-trans [trans] =
  r-r-into-trancl trancl-trans rtrancl-trans
  trancl.trancl-into-trancl trancl-into-trancl2
  rtrancl.rtrancl-into-rtrancl converse-rtrancl-into-rtrancl
  rtrancl-trancl-trancl trancl-rtrancl-trancl

```

```

lemmas transitive-closurep-trans' [trans] =
  tranclp-trans rtranclp-trans
  tranclp.trancl-into-trancl tranclp-into-tranclp2
  rtranclp.rtrancl-into-rtrancl converse-rtranclp-into-rtranclp
  rtranclp-tranclp-tranclp tranclp-rtranclp-tranclp

```

```

declare trancl-into-rtrancl [elim]

```

## 20.4 Setup of transitivity reasoner

ML  $\ll$

```

structure Tranc1-Tac = Tranc1-Tac-Fun (
  struct
    val r-into-tranc1 = @{thm tranc1.r-into-tranc1};
    val tranc1-trans  = @{thm tranc1-trans};
    val rtranc1-refl  = @{thm rtranc1.rtranc1-refl};
    val r-into-rtranc1 = @{thm r-into-rtranc1};
    val tranc1-into-rtranc1 = @{thm tranc1-into-rtranc1};
    val rtranc1-tranc1-tranc1 = @{thm rtranc1-tranc1-tranc1};
    val tranc1-rtranc1-tranc1 = @{thm tranc1-rtranc1-tranc1};
    val rtranc1-trans = @{thm rtranc1-trans};

    fun decomp (@{const Trueprop} $ t) =
      let fun dec (Const (op :, -) $ (Const (Pair, -) $ a $ b) $ rel) =
          let fun decr (Const (Transitive-Closure.rtranc1, -) $ r) = (r,r*)
              | decr (Const (Transitive-Closure.tranc1, -) $ r) = (r,r+)
              | decr r = (r,r);
          val (rel,r) = decr (Envir.beta-eta-contract rel);
          in SOME (a,b,rel,r) end
        | dec - = NONE
      in dec t end
      | decomp - = NONE;

  end);

structure Tranc1p-Tac = Tranc1-Tac-Fun (
  struct
    val r-into-tranc1 = @{thm tranc1p.r-into-tranc1};
    val tranc1-trans  = @{thm tranc1p-trans};
    val rtranc1-refl  = @{thm rtranc1p.rtranc1-refl};
    val r-into-rtranc1 = @{thm r-into-rtranc1p};
    val tranc1-into-rtranc1 = @{thm tranc1p-into-rtranc1p};
    val rtranc1-tranc1-tranc1 = @{thm rtranc1p-tranc1p-tranc1p};
    val tranc1-rtranc1-tranc1 = @{thm tranc1p-rtranc1p-tranc1p};
    val rtranc1-trans = @{thm rtranc1p-trans};

    fun decomp (@{const Trueprop} $ t) =
      let fun dec (rel $ a $ b) =
          let fun decr (Const (Transitive-Closure.rtranc1p, -) $ r) = (r,r*)
              | decr (Const (Transitive-Closure.tranc1p, -) $ r) = (r,r+)
              | decr r = (r,r);
          val (rel,r) = decr rel;
          in SOME (a, b, rel, r) end
        | dec - = NONE
      in dec t end
      | decomp - = NONE;

  end);

```

```

    end);
  >>

declaration << fn - =>
  Simplifier.map-ss (fn ss => ss
    addSolver (mk-solver Tranc1 (fn - => Tranc1-Tac.tranc1-tac))
    addSolver (mk-solver Rtranc1 (fn - => Tranc1-Tac.rtranc1-tac))
    addSolver (mk-solver Tranc1p (fn - => Tranc1p-Tac.tranc1-tac))
    addSolver (mk-solver Rtranc1p (fn - => Tranc1p-Tac.rtranc1-tac)))
  >>

method-setup tranc1 =
  << Scan.succeed (K (SIMPLE-METHOD' Tranc1-Tac.tranc1-tac)) >>
  << simple transitivity reasoner >>
method-setup rtranc1 =
  << Scan.succeed (K (SIMPLE-METHOD' Tranc1-Tac.rtranc1-tac)) >>
  << simple transitivity reasoner >>
method-setup tranc1p =
  << Scan.succeed (K (SIMPLE-METHOD' Tranc1p-Tac.tranc1-tac)) >>
  << simple transitivity reasoner (predicate version) >>
method-setup rtranc1p =
  << Scan.succeed (K (SIMPLE-METHOD' Tranc1p-Tac.rtranc1-tac)) >>
  << simple transitivity reasoner (predicate version) >>

end

```

## 21 Wellfounded: Well-founded Recursion

```

theory Wellfounded
imports Finite-Set Transitive-Closure Nat
uses (Tools/function-package/size.ML)
begin

```

### 21.1 Basic Definitions

```

inductive
  wfrec-rel :: ('a * 'a) set => (('a => 'b) => 'a => 'b) => 'a => 'b => bool
  for R :: ('a * 'a) set
  and F :: ('a => 'b) => 'a => 'b
  where
    wfrecI: ALL z. (z, x) : R --> wfrec-rel R F z (g z) ==>
      wfrec-rel R F x (F g x)

constdefs
  wf :: ('a * 'a) set => bool
  wf(r) == (!P. (!x. (!y. (y,x):r --> P(y)) --> P(x)) --> (!x. P(x)))

```

$wfP :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$   
 $wfP\ r == wf\ \{(x, y). r\ x\ y\}$

$acyclic :: ('a * 'a) set \Rightarrow bool$   
 $acyclic\ r == !x. (x, x) \sim: r^+ +$

$cut :: ('a \Rightarrow 'b) \Rightarrow ('a * 'a) set \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b$   
 $cut\ f\ r\ x == (\%y. \text{if } (y, x):r \text{ then } f\ y \text{ else undefined})$

$adm\text{-}wf :: ('a * 'a) set \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow bool$   
 $adm\text{-}wf\ R\ F == ALL\ f\ g\ x.$   
 $(ALL\ z. (z, x) : R \dashrightarrow f\ z = g\ z) \dashrightarrow F\ f\ x = F\ g\ x$

$wfrec :: ('a * 'a) set \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$   
 $[code\ del]: wfrec\ R\ F == \%x. THE\ y. wfrec\text{-}rel\ R\ (\%f\ x. F\ (cut\ f\ R\ x)\ x)\ x\ y$

**abbreviation**  $acyclicP :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$  **where**  
 $acyclicP\ r == acyclic\ \{(x, y). r\ x\ y\}$

**lemma**  $wfP\text{-}wf\text{-}eq$   $[pred\text{-}set\text{-}conv]: wfP\ (\lambda x\ y. (x, y) \in r) = wf\ r$   
**by**  $(simp\ add: wfP\text{-}def)$

**lemma**  $wfUNIVI$ :  
 $(!!P\ x. (ALL\ x. (ALL\ y. (y, x) : r \dashrightarrow P(y)) \dashrightarrow P(x)) \implies P(x)) \implies$   
 $wf(r)$   
**unfolding**  $wf\text{-}def$  **by**  $blast$

**lemmas**  $wfPUNIVI = wfUNIVI$   $[to\text{-}pred]$

Restriction to domain  $A$  and range  $B$ . If  $r$  is well-founded over their intersection, then  $wf\ r$

**lemma**  $wfI$ :  
 $[| r \subseteq A \lt * B;$   
 $!!x\ P. [| \forall y. (\forall y. (y, x) : r \dashrightarrow P(y)) \dashrightarrow P(x); x : A; x : B |] \implies P\ x |]$   
 $\implies wf\ r$   
**unfolding**  $wf\text{-}def$  **by**  $blast$

**lemma**  $wf\text{-}induct$ :  
 $[| wf(r);$   
 $!!x. [| ALL\ y. (y, x) : r \dashrightarrow P(y) |] \implies P(x)$   
 $|] \implies P(a)$   
**unfolding**  $wf\text{-}def$  **by**  $blast$

**lemmas**  $wfP\text{-}induct = wf\text{-}induct$   $[to\text{-}pred]$

**lemmas**  $wf\text{-}induct\text{-}rule = wf\text{-}induct$   $[rule\text{-}format, \text{consumes } 1, \text{case-names less,}$   
 $induct\ set: wf]$



**lemmas** *wfP-induct-rule* = *wf-induct-rule* [*to-pred*, *induct set*: *wfP*]

**lemma** *wf-not-sym*: *wf r ==> (a, x) : r ==> (x, a) ~: r*  
**by** (*induct a arbitrary: x set: wf*) *blast*

**lemmas** *wf-asym* = *wf-not-sym* [*elim-format*]

**lemma** *wf-not-refl* [*simp*]: *wf r ==> (a, a) ~: r*  
**by** (*blast elim: wf-asym*)

**lemmas** *wf-irrefl* = *wf-not-refl* [*elim-format*]

**lemma** *wf-wellorderI*:  
**assumes** *wf*: *wf {(x::'a::ord, y). x < y}*  
**assumes** *lin*: *OFCLASS('a::ord, linorder-class)*  
**shows** *OFCLASS('a::ord, wellorder-class)*  
**using** *lin* **by** (*rule wellorder-class.intro*)  
*(blast intro: wellorder-axioms.intro wf-induct-rule [OF wf])*

**lemma** (*in wellorder*) *wf*:  
*wf {(x, y). x < y}*  
**unfolding** *wf-def* **by** (*blast intro: less-induct*)

## 21.2 Basic Results

transitive closure of a well-founded relation is well-founded!

**lemma** *wf-trancl*:  
**assumes** *wf r*  
**shows** *wf (r<sup>+</sup>)*  
**proof** –  
 {  
**fix** *P* **and** *x*  
**assume** *induct-step*: *!!x. (!!y. (y, x) : r<sup>+</sup> ==> P y) ==> P x*  
**have** *P x*  
**proof** (*rule induct-step*)  
**fix** *y* **assume** *(y, x) : r<sup>+</sup>*  
**with** *(wf r)* **show** *P y*  
**proof** (*induct x arbitrary: y*)  
**case** (*less x*)  
**note** *hyp* = *( $\bigwedge x' y'. (x', x) : r ==> (y', x') : r<sup>+</sup> ==> P y'$ )*  
**from** *(y, x) : r<sup>+</sup>* **show** *P y*  
**proof** *cases*  
**case** *base*  
**show** *P y*  
**proof** (*rule induct-step*)  
**fix** *y'* **assume** *(y', y) : r<sup>+</sup>*  
**with** *(y, x) : r* **show** *P y'* **by** (*rule hyp [of y y']*)  
 }

```

      qed
    next
      case step
      then obtain  $x'$  where  $(x', x) : r$  and  $(y, x') : r^+ +$  by simp
      then show  $P y$  by (rule hyp [of  $x' y$ ])
    qed
  qed
  qed
} then show ?thesis unfolding wf-def by blast
qed

```

lemmas *wfP-trancl* = *wf-trancl* [to-pred]

```

lemma wf-converse-trancl:  $wf (r^-1) ==> wf ((r^+)^-1)$ 
  apply (subst trancl-converse [symmetric])
  apply (erule wf-trancl)
  done

```

Minimal-element characterization of well-foundedness

```

lemma wf-eq-minimal:  $wf r = (\forall Q x. x \in Q \longrightarrow (\exists z \in Q. \forall y. (y, z) \in r \longrightarrow y \notin Q))$ 

```

```

proof (intro iffI strip)
  fix  $Q :: 'a \text{ set}$  and  $x$ 
  assume wf  $r$  and  $x \in Q$ 
  then show  $\exists z \in Q. \forall y. (y, z) \in r \longrightarrow y \notin Q$ 
    unfolding wf-def
    by (blast dest: spec [of - % $x. x \in Q \longrightarrow (\exists z \in Q. \forall y. (y, z) \in r \longrightarrow y \notin Q)$ ])
next
  assume 1:  $\forall Q x. x \in Q \longrightarrow (\exists z \in Q. \forall y. (y, z) \in r \longrightarrow y \notin Q)$ 
  show wf  $r$ 
  proof (rule wfUNIVI)
    fix  $P :: 'a \Rightarrow \text{bool}$  and  $x$ 
    assume 2:  $\forall x. (\forall y. (y, x) \in r \longrightarrow P y) \longrightarrow P x$ 
    let ? $Q = \{x. \neg P x\}$ 
    have  $x \in ?Q \longrightarrow (\exists z \in ?Q. \forall y. (y, z) \in r \longrightarrow y \notin ?Q)$ 
      by (rule 1 [THEN spec, THEN spec])
    then have  $\neg P x \longrightarrow (\exists z. \neg P z \wedge (\forall y. (y, z) \in r \longrightarrow P y))$  by simp
    with 2 have  $\neg P x \longrightarrow (\exists z. \neg P z \wedge P z)$  by fast
    then show  $P x$  by simp
  qed
qed

```

lemma *wfE-min*:

```

  assumes wf  $R$   $x \in Q$ 
  obtains  $z$  where  $z \in Q \wedge y. (y, z) \in R \implies y \notin Q$ 
  using assms unfolding wf-eq-minimal by blast

```

lemma *wfI-min*:

```

  ( $\wedge x Q. x \in Q \implies \exists z \in Q. \forall y. (y, z) \in R \longrightarrow y \notin Q$ )

```

$\Rightarrow wf\ R$   
**unfolding** *wf-eq-minimal* **by** *blast*

**lemmas** *wfP-eq-minimal* = *wf-eq-minimal* [*to-pred*]

Well-foundedness of subsets

**lemma** *wf-subset*: [| *wf*(*r*); *p* <=*r* |] ==> *wf*(*p*)  
**apply** (*simp* (*no-asm-use*) *add*: *wf-eq-minimal*)  
**apply** *fast*  
**done**

**lemmas** *wfP-subset* = *wf-subset* [*to-pred*]

Well-foundedness of the empty relation

**lemma** *wf-empty* [*iff*]: *wf*({})  
**by** (*simp* *add*: *wf-def*)

**lemmas** *wfP-empty* [*iff*] =  
*wf-empty* [*to-pred* *bot-empty-eq2*, *simplified* *bot-fun-eq* *bot-bool-eq*]

**lemma** *wf-Int1*: *wf* *r* ==> *wf* (*r* *Int* *r'*)  
**apply** (*erule* *wf-subset*)  
**apply** (*rule* *Int-lower1*)  
**done**

**lemma** *wf-Int2*: *wf* *r* ==> *wf* (*r'* *Int* *r*)  
**apply** (*erule* *wf-subset*)  
**apply** (*rule* *Int-lower2*)  
**done**

Well-foundedness of insert

**lemma** *wf-insert* [*iff*]: *wf*(*insert* (*y*,*x*) *r*) = (*wf*(*r*) & (*x*,*y*)  $\sim$ : *r*<sup>^\*</sup>)  
**apply** (*rule* *iffI*)  
**apply** (*blast* *elim*: *wf-trancl* [*THEN* *wf-irrefl*]  
*intro*: *rtrancl-into-trancl1* *wf-subset*  
*rtrancl-mono* [*THEN* [*2*] *rev-subsetD*])  
**apply** (*simp* *add*: *wf-eq-minimal*, *safe*)  
**apply** (*rule* *allE*, *assumption*, *erule* *impE*, *blast*)  
**apply** (*erule* *bexE*)  
**apply** (*rename-tac* *a*, *case-tac* *a* = *x*)  
**prefer** 2  
**apply** *blast*  
**apply** (*case-tac* *y*:*Q*)  
**prefer** 2 **apply** *blast*  
**apply** (*rule-tac* *x* = {*z*. *z*:*Q* & (*z*,*y*) : *r*<sup>^\*</sup>} **in** *allE*)  
**apply** *assumption*  
**apply** (*erule-tac* *V* = *ALL* *Q*. (*EX* *x*. *x* : *Q*)  $\longrightarrow$  ?*P* *Q* **in** *thin-rl*)  
— essential for speed

Blast with new substOccur fails

```
apply (fast intro: converse-rtrancl-into-rtrancl)
done
```

Well-foundedness of image

```
lemma wf-prod-fun-image: [| wf r; inj f |] ==> wf(prod-fun f f ‘ r)
apply (simp only: wf-eq-minimal, clarify)
apply (case-tac EX p. f p : Q)
apply (erule-tac x = {p. f p : Q} in allE)
apply (fast dest: inj-onD, blast)
done
```

### 21.3 Well-Foundedness Results for Unions

```
lemma wf-union-compatible:
  assumes wf R wf S
  assumes S O R ⊆ R
  shows wf (R ∪ S)
proof (rule wfI-min)
  fix x :: 'a and Q
  let ?Q' = {x ∈ Q. ∀ y. (y, x) ∈ R ⟶ y ∉ Q}
  assume x ∈ Q
  obtain a where a ∈ ?Q'
    by (rule wfE-min [OF ⟨wf R⟩ ⟨x ∈ Q⟩]) blast
  with ⟨wf S⟩
  obtain z where z ∈ ?Q' and zmin: ∧ y. (y, z) ∈ S ⟹ y ∉ ?Q' by (erule
wfE-min)
  {
    fix y assume (y, z) ∈ S
    then have y ∉ ?Q' by (rule zmin)

    have y ∉ Q
    proof
      assume y ∈ Q
      with ⟨y ∉ ?Q'⟩
      obtain w where (w, y) ∈ R and w ∈ Q by auto
      from ⟨(w, y) ∈ R⟩ ⟨(y, z) ∈ S⟩ have (w, z) ∈ S O R by (rule rel-compI)
      with ⟨S O R ⊆ R⟩ have (w, z) ∈ R ..
      with ⟨z ∈ ?Q'⟩ have w ∉ Q by blast
      with ⟨w ∈ Q⟩ show False by contradiction
    qed
  }
  with ⟨z ∈ ?Q'⟩ show  $\exists z \in Q. \forall y. (y, z) \in R \cup S \longrightarrow y \notin Q$  by blast
qed
```

Well-foundedness of indexed union with disjoint domains and ranges

```
lemma wf-UN: [| ALL i:I. wf(r i);
  ALL i:I. ALL j:I. r i ~ = r j ⟶ Domain(r i) Int Range(r j) = {}
|] ==> wf(UN i:I. r i)
```

```

apply (simp only: wf-eq-minimal, clarify)
apply (rename-tac A a, case-tac EX i:I. EX a:A. EX b:A. (b,a) : r i)
prefer 2
apply force
apply clarify
apply (drule bspec, assumption)
apply (erule-tac x={a. a:A & (EX b:A. (b,a) : r i) } in allE)
apply (blast elim!: allE)
done

lemmas wfP-SUP = wf-UN [where I=UNIV and r=λi. {(x, y). r i x y},
to-pred SUP-UN-eq2 bot-empty-eq pred-equals-eq, simplified, standard]

lemma wf-Union:
  [| ALL r:R. wf r;
    ALL r:R. ALL s:R. r ~ = s --> Domain r Int Range s = {}
  |] ==> wf(Union R)
apply (simp add: Union-def)
apply (blast intro: wf-UN)
done

lemma wf-Un:
  [| wf r; wf s; Domain r Int Range s = {} |] ==> wf(r Un s)
using wf-union-compatible[of s r]
by (auto simp: Un-ac)

lemma wf-union-merge:
  wf (R ∪ S) = wf (R O R ∪ R O S ∪ S) (is wf ?A = wf ?B)
proof
  assume wf ?A
  with wf-trancl have wfT: wf (?A ^+).
  moreover have ?B ⊆ ?A ^+
    by (subst trancl-unfold, subst trancl-unfold) blast
  ultimately show wf ?B by (rule wf-subset)
next
  assume wf ?B

  show wf ?A
  proof (rule wfI-min)
    fix Q :: 'a set and x
    assume x ∈ Q

    with ⟨wf ?B⟩
    obtain z where z ∈ Q and ∧y. (y, z) ∈ ?B ⟹ y ∉ Q
      by (erule wfE-min)
    then have A1: ∧y. (y, z) ∈ R O R ⟹ y ∉ Q
      and A2: ∧y. (y, z) ∈ R O S ⟹ y ∉ Q
      and A3: ∧y. (y, z) ∈ S ⟹ y ∉ Q

```

```

by auto

show  $\exists z \in Q. \forall y. (y, z) \in ?A \longrightarrow y \notin Q$ 
proof (cases  $\forall y. (y, z) \in R \longrightarrow y \notin Q$ )
  case True
    with  $\langle z \in Q \rangle A3$  show ?thesis by blast
  next
    case False
    then obtain  $z'$  where  $z' \in Q \ (z', z) \in R$  by blast

    have  $\forall y. (y, z') \in ?A \longrightarrow y \notin Q$ 
    proof (intro allI impI)
      fix y assume  $(y, z') \in ?A$ 
      then show  $y \notin Q$ 
      proof
        assume  $(y, z') \in R$ 
        then have  $(y, z) \in R \ O \ R$  using  $\langle (z', z) \in R \rangle ..$ 
        with A1 show  $y \notin Q$  .
      next
        assume  $(y, z') \in S$ 
        then have  $(y, z) \in R \ O \ S$  using  $\langle (z', z) \in R \rangle ..$ 
        with A2 show  $y \notin Q$  .
      qed
    qed
  with  $\langle z' \in Q \rangle$  show ?thesis ..
qed
qed
qed
qed

```

**lemma** *wf-comp-self*:  $wf\ R = wf\ (R \ O \ R)$  — special case  
 by (rule *wf-union-merge* [where  $S = \{\}$ , simplified])

### 21.3.1 acyclic

**lemma** *acyclicI*:  $ALL\ x. (x, x) \sim: r^+ \implies acyclic\ r$   
 by (simp add: *acyclic-def*)

**lemma** *wf-acyclic*:  $wf\ r \implies acyclic\ r$   
**apply** (simp add: *acyclic-def*)  
**apply** (blast elim: *wf-trancl* [THEN *wf-irrefl*])  
**done**

**lemmas** *wfP-acyclicP* = *wf-acyclic* [to-pred]

**lemma** *acyclic-insert* [iff]:  
 $acyclic(insert\ (y,x)\ r) = (acyclic\ r \ \& \ (x,y) \sim: r^+)$   
**apply** (simp add: *acyclic-def trancl-insert*)  
**apply** (blast intro: *rtrancl-trans*)  
**done**

**lemma** *acyclic-converse* [iff]:  $acyclic(r^{-1}) = acyclic\ r$   
**by** (*simp add: acyclic-def trancl-converse*)

**lemmas** *acyclicP-converse* [iff] = *acyclic-converse* [to-pred]

**lemma** *acyclic-impl-antisym-rtrancl*:  $acyclic\ r \impl antisym(r^*)$   
**apply** (*simp add: acyclic-def antisym-def*)  
**apply** (*blast elim: rtranclE intro: rtrancl-into-trancl1 rtrancl-trancl-trancl*)  
**done**

**lemma** *acyclic-subset*:  $[| acyclic\ s; r \leq s |] \impl acyclic\ r$   
**apply** (*simp add: acyclic-def*)  
**apply** (*blast intro: trancl-mono*)  
**done**

Wellfoundedness of finite acyclic relations

**lemma** *finite-acyclic-wf* [rule-format]:  $finite\ r \impl acyclic\ r \dashv\vdash wf\ r$   
**apply** (*erule finite-induct, blast*)  
**apply** (*simp (no-asm-simp) only: split-tupled-all*)  
**apply** *simp*  
**done**

**lemma** *finite-acyclic-wf-converse*:  $[| finite\ r; acyclic\ r |] \impl wf\ (r^{-1})$   
**apply** (*erule finite-converse [THEN iffD2, THEN finite-acyclic-wf]*)  
**apply** (*erule acyclic-converse [THEN iffD2]*)  
**done**

**lemma** *wf-iff-acyclic-if-finite*:  $finite\ r \impl wf\ r = acyclic\ r$   
**by** (*blast intro: finite-acyclic-wf wf-acyclic*)

## 21.4 Well-Founded Recursion

cut

**lemma** *cuts-eq*:  $(cut\ f\ r\ x = cut\ g\ r\ x) = (ALL\ y. (y,x):r \dashv\vdash f(y)=g(y))$   
**by** (*simp add: expand-fun-eq cut-def*)

**lemma** *cut-apply*:  $(x,a):r \impl (cut\ f\ r\ a)(x) = f(x)$   
**by** (*simp add: cut-def*)

Inductive characterization of wfrec combinator; for details see: John Harrison, “Inductive definitions: automation and application”

**lemma** *wfrec-unique*:  $[| adm-wf\ R\ F; wf\ R |] \impl EX! y. wfrec-rel\ R\ F\ x\ y$   
**apply** (*simp add: adm-wf-def*)  
**apply** (*erule-tac a=x in wf-induct*)  
**apply** (*rule ex1I*)

```

apply (rule-tac g = %x. THE y. wfrec-rel R F x y in wfrec-rel.wfrecI)
apply (fast dest!: theI')
apply (erule wfrec-rel.cases, simp)
apply (erule allE, erule allE, erule allE, erule mp)
apply (fast intro: the-equality [symmetric])
done

```

```

lemma adm-lemma: adm-wf R (%f x. F (cut f R x) x)
apply (simp add: adm-wf-def)
apply (intro strip)
apply (rule cuts-eq [THEN iffD2, THEN subst], assumption)
apply (rule refl)
done

```

```

lemma wfrec: wf(r) ==> wfrec r H a = H (cut (wfrec r H) r a) a
apply (simp add: wfrec-def)
apply (rule adm-lemma [THEN wfrec-unique, THEN the1-equality], assumption)
apply (rule wfrec-rel.wfrecI)
apply (intro strip)
apply (erule adm-lemma [THEN wfrec-unique, THEN theI'])
done

```

## 21.5 Code generator setup

```

consts-code
  wfrec ((<module>wfrec?)
attach <<
  fun wfrec f x = f (wfrec f) x;
  >>

```

## 21.6 nat is well-founded

```

lemma less-nat-rel: op < = (λm n. n = Suc m) ^++
proof (rule ext, rule ext, rule iffI)
  fix n m :: nat
  assume m < n
  then show (λm n. n = Suc m) ^++ m n
  proof (induct n)
    case 0 then show ?case by auto
  next
    case (Suc n) then show ?case
      by (auto simp add: less-Suc-eq-le le-less intro: tranclp.trancl-into-trancl)
  qed
next
  fix n m :: nat
  assume (λm n. n = Suc m) ^++ m n
  then show m < n
    by (induct n)
      (simp-all add: less-Suc-eq-le reflexive le-less)
  qed

```



**definition**

$pred\text{-}nat :: (nat * nat) \text{ set}$  **where**  
 $pred\text{-}nat = \{(m, n). n = Suc\ m\}$

**definition**

$less\text{-}than :: (nat * nat) \text{ set}$  **where**  
 $less\text{-}than = pred\text{-}nat^+ +$

**lemma**  $less\text{-}eq: (m, n) \in pred\text{-}nat^+ \longleftrightarrow m < n$   
**unfolding**  $less\text{-}nat\text{-}rel\ pred\text{-}nat\text{-}def\ trancl\text{-}def$  **by** *simp*

**lemma**  $pred\text{-}nat\text{-}trancl\text{-}eq\text{-}le:$   
 $(m, n) \in pred\text{-}nat^* \longleftrightarrow m \leq n$   
**unfolding**  $less\text{-}eq\ rtrancl\text{-}eq\text{-}or\text{-}trancl$  **by** *auto*

**lemma**  $wf\text{-}pred\text{-}nat: wf\ pred\text{-}nat$   
**apply** (*unfold wf-def pred-nat-def, clarify*)  
**apply** (*induct-tac x, blast+*)  
**done**

**lemma**  $wf\text{-}less\text{-}than\ [iff]: wf\ less\text{-}than$   
**by** (*simp add: less-than-def wf-pred-nat [THEN wf-trancl]*)

**lemma**  $trans\text{-}less\text{-}than\ [iff]: trans\ less\text{-}than$   
**by** (*simp add: less-than-def trans-trancl*)

**lemma**  $less\text{-}than\text{-}iff\ [iff]: ((x, y): less\text{-}than) = (x < y)$   
**by** (*simp add: less-than-def less-eq*)

**lemma**  $wf\text{-}less: wf\ \{(x, y::nat). x < y\}$   
**using**  $wf\text{-}less\text{-}than$  **by** (*simp add: less-than-def less-eq [symmetric]*)

## 21.7 Accessible Part

Inductive definition of the accessible part  $acc\ r$  of a relation; see also [?].

**inductive-set**

$acc :: ('a * 'a) \text{ set} \Rightarrow 'a \text{ set}$   
**for**  $r :: ('a * 'a) \text{ set}$   
**where**  
 $accI: (!y. (y, x) : r \Rightarrow y : acc\ r) \Rightarrow x : acc\ r$

**abbreviation**

$termip :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$  **where**  
 $termip\ r == accp\ (r^{-1-1})$

**abbreviation**

$termi :: ('a * 'a) \text{ set} \Rightarrow 'a \text{ set}$  **where**  
 $termi\ r == acc\ (r^{-1})$

**lemmas** *accpI* = *accp.accI*

Induction rules

**theorem** *accp-induct*:

**assumes** *major*: *accp r a*  
  **assumes** *hyp*:  $\forall x. \text{accp } r \ x \implies \forall y. r \ y \ x \implies P \ y \implies P \ x$   
  **shows** *P a*  
  **apply** (*rule major [THEN accp.induct]*)  
  **apply** (*rule hyp*)  
  **apply** (*rule accp.accI*)  
  **apply** *fast*  
  **apply** *fast*  
  **done**

**theorems** *accp-induct-rule* = *accp-induct [rule-format, induct set: accp]*

**theorem** *accp-downward*:  $\text{accp } r \ b \implies r \ a \ b \implies \text{accp } r \ a$

**apply** (*erule accp.cases*)  
  **apply** *fast*  
  **done**

**lemma** *not-accp-down*:

**assumes** *na*:  $\neg \text{accp } R \ x$   
  **obtains** *z* **where** *R z x* **and**  $\neg \text{accp } R \ z$   
**proof** –  
  **assume** *a*:  $\bigwedge z. \llbracket R \ z \ x; \neg \text{accp } R \ z \rrbracket \implies \text{thesis}$

**show** *thesis*

**proof** (*cases*  $\forall z. R \ z \ x \longrightarrow \text{accp } R \ z$ )  
  **case** *True*  
  **hence**  $\bigwedge z. R \ z \ x \implies \text{accp } R \ z$  **by** *auto*  
  **hence** *accp R x*  
  **by** (*rule accp.accI*)  
  **with** *na* **show** *thesis ..*

**next**

**case** *False* **then obtain** *z* **where** *R z x* **and**  $\neg \text{accp } R \ z$   
    **by** *auto*  
    **with** *a* **show** *thesis .*

**qed**

**qed**

**lemma** *accp-downwards-aux*:  $r^{**} \ b \ a \implies \text{accp } r \ a \implies \text{accp } r \ b$

**apply** (*erule rtranclp-induct*)  
  **apply** *blast*  
  **apply** (*blast dest: accp-downward*)  
  **done**

**theorem** *accp-downwards*:  $\text{accp } r \ a \implies r^{**} \ b \ a \implies \text{accp } r \ b$

```

apply (blast dest: accp-downwards-aux)
done

theorem accp-wfPI:  $\forall x. \text{accp } r \ x ==> \text{wfP } r$ 
  apply (rule wfPUNIVI)
  apply (induct-tac P x rule: accp-induct)
  apply blast
  apply blast
  done

theorem accp-wfPD:  $\text{wfP } r ==> \text{accp } r \ x$ 
  apply (erule wfP-induct-rule)
  apply (rule accp.accI)
  apply blast
  done

theorem wfP-accp-iff:  $\text{wfP } r = (\forall x. \text{accp } r \ x)$ 
  apply (blast intro: accp-wfPI dest: accp-wfPD)
  done

```

Smaller relations have bigger accessible parts:

```

lemma accp-subset:
  assumes sub:  $R1 \leq R2$ 
  shows  $\text{accp } R2 \leq \text{accp } R1$ 
proof (rule predicate1I)
  fix x assume  $\text{accp } R2 \ x$ 
  then show  $\text{accp } R1 \ x$ 
  proof (induct x)
    fix x
    assume ih:  $\bigwedge y. R2 \ y \ x \implies \text{accp } R1 \ y$ 
    with sub show  $\text{accp } R1 \ x$ 
    by (blast intro: accp.accI)
  qed
qed

```

This is a generalized induction theorem that works on subsets of the accessible part.

```

lemma accp-subset-induct:
  assumes subset:  $D \leq \text{accp } R$ 
  and dcl:  $\bigwedge x \ z. \llbracket D \ x; R \ z \ x \rrbracket \implies D \ z$ 
  and Dx:  $D \ x$ 
  and istep:  $\bigwedge x. \llbracket D \ x; (\bigwedge z. R \ z \ x \implies P \ z) \rrbracket \implies P \ x$ 
  shows  $P \ x$ 
proof –
  from subset and  $\langle D \ x \rangle$ 
  have  $\text{accp } R \ x \ ..$ 
  then show  $P \ x$  using  $\langle D \ x \rangle$ 
  proof (induct x)
    fix x

```

```

    assume  $D\ x$ 
    and  $\bigwedge y. R\ y\ x \implies D\ y \implies P\ y$ 
    with dcl and istep show  $P\ x$  by blast
qed
qed

```

Set versions of the above theorems

```
lemmas acc-induct = accp-induct [to-set]
```

```
lemmas acc-induct-rule = acc-induct [rule-format, induct set: acc]
```

```
lemmas acc-downward = accp-downward [to-set]
```

```
lemmas not-acc-down = not-accp-down [to-set]
```

```
lemmas acc-downwards-aux = accp-downwards-aux [to-set]
```

```
lemmas acc-downwards = accp-downwards [to-set]
```

```
lemmas acc-wfI = accp-wfPI [to-set]
```

```
lemmas acc-wfD = accp-wfPD [to-set]
```

```
lemmas wf-acc-iff = wfP-accp-iff [to-set]
```

```
lemmas acc-subset = accp-subset [to-set pred-subset-eq]
```

```
lemmas acc-subset-induct = accp-subset-induct [to-set pred-subset-eq]
```

## 21.8 Tools for building wellfounded relations

Inverse Image

```

lemma wf-inv-image [simp,intro!]:  $wf(r) \implies wf(inv\_image\ r\ (f::'a \implies 'b))$ 
apply (simp (no-asm-use) add: inv-image-def wf-eq-minimal)
apply clarify
apply (subgoal-tac  $EX\ (w::'b) . w : \{w. EX\ (x::'a) . x : Q \ \&\ (f\ x = w)\}$ )
prefer 2 apply (blast del: allE)
apply (erule allE)
apply (erule (1) notE impE)
apply blast
done

```

```

lemma in-inv-image[simp]:  $((x,y) : inv\_image\ r\ f) = ((f\ x, f\ y) : r)$ 
  by (auto simp: inv-image-def)

```

Measure functions into *nat*

```

definition measure ::  $('a \implies nat) \implies ('a * 'a)_{set}$ 
where measure == inv-image less-than

```

**lemma** *in-measure*[*simp*]:  $((x,y) : \text{measure } f) = (f\ x < f\ y)$   
**by** (*simp add:measure-def*)

**lemma** *wf-measure* [*iff*]: *wf* (*measure* *f*)  
**apply** (*unfold measure-def*)  
**apply** (*rule wf-less-than [THEN wf-inv-image]*)  
**done**

Lexicographic combinations

**definition**  
*lex-prod* ::  $[(a*a')\text{set}, (b*b')\text{set}] \Rightarrow ((a*a')*(b*b'))\text{set}$   
 (**infixr** *<\*lex\*>* 80)

**where**

$ra\ <*lex*>\ rb == \{((a,b),(a',b')).\ (a,a') : ra \mid a=a' \ \&\ (b,b') : rb\}$

**lemma** *wf-lex-prod* [*intro!*]:  $[\text{wf}(ra); \text{wf}(rb)] \implies \text{wf}(ra\ <*lex*>\ rb)$   
**apply** (*unfold wf-def lex-prod-def*)  
**apply** (*rule allI, rule impI*)  
**apply** (*simp (no-asm-use) only: split-paired-All*)  
**apply** (*drule spec, erule mp*)  
**apply** (*rule allI, rule impI*)  
**apply** (*drule spec, erule mp, blast*)  
**done**

**lemma** *in-lex-prod*[*simp*]:  
 $((a,b),(a',b')) : r\ <*lex*>\ s \iff ((a,a') : r \vee (a = a' \wedge (b, b') : s))$   
**by** (*auto simp:lex-prod-def*)

*op* *<\*lex\*>* preserves transitivity

**lemma** *trans-lex-prod* [*intro!*]:  
 $[\text{trans } R1; \text{trans } R2] \implies \text{trans } (R1\ <*lex*>\ R2)$   
**by** (*unfold trans-def lex-prod-def, blast*)

lexicographic combinations with measure functions

**definition**  
*mlex-prod* ::  $(a \Rightarrow \text{nat}) \Rightarrow (a \times a)\ \text{set} \Rightarrow (a \times a)\ \text{set}$  (**infixr** *<\*mlex\*>* 80)

**where**

$f\ <*mlex*>\ R = \text{inv-image } (\text{less-than } <*lex*>\ R) (\%x.\ (f\ x, x))$

**lemma** *wf-mlex*:  $\text{wf } R \implies \text{wf } (f\ <*mlex*>\ R)$   
**unfolding** *mlex-prod-def*  
**by** *auto*

**lemma** *mlex-less*:  $f\ x < f\ y \implies (x, y) \in f\ <*mlex*>\ R$   
**unfolding** *mlex-prod-def* **by** *simp*

**lemma** *mlex-leq*:  $f\ x \leq f\ y \implies (x, y) \in R \implies (x, y) \in f\ <*mlex*>\ R$   
**unfolding** *mlex-prod-def* **by** *auto*

proper subset relation on finite sets

**definition** *finite-psubset* :: ('a set \* 'a set) set  
**where** *finite-psubset* == {(A,B). A < B & finite B}

**lemma** *wf-finite-psubset[simp]*: wf(*finite-psubset*)  
**apply** (*unfold finite-psubset-def*)  
**apply** (*rule wf-measure [THEN wf-subset]*)  
**apply** (*simp add: measure-def inv-image-def less-than-def less-eq*)  
**apply** (*fast elim!: psubset-card-mono*)  
**done**

**lemma** *trans-finite-psubset*: *trans finite-psubset*  
**by** (*simp add: finite-psubset-def less-le trans-def, blast*)

**lemma** *in-finite-psubset[simp]*: (A, B) ∈ *finite-psubset* = (A < B & finite B)  
**unfolding** *finite-psubset-def* **by** *auto*

max- and min-extension of order to finite sets

**inductive-set** *max-ext* :: ('a × 'a) set ⇒ ('a set × 'a set) set  
**for** *R* :: ('a × 'a) set  
**where**

*max-extI[intro]*: *finite X* ⇒ *finite Y* ⇒ *Y* ≠ {} ⇒ (∧*x*. *x* ∈ *X* ⇒ ∃*y* ∈ *Y*.  
(*x*, *y*) ∈ *R*) ⇒ (*X*, *Y*) ∈ *max-ext R*

**lemma** *max-ext-wf*:

**assumes** *wf*: wf *r*

**shows** wf (*max-ext r*)

**proof** (*rule acc-wfI, intro allI*)

**fix** *M* **show** *M* ∈ acc (*max-ext r*) (**is** - ∈ ?*W*)

**proof** *cases*

**assume** *finite M*

**thus** ?thesis

**proof** (*induct M*)

**show** {} ∈ ?*W*

**by** (*rule accI*) (*auto elim: max-ext.cases*)

**next**

**fix** *M a* **assume** *M* ∈ ?*W* *finite M*

**with** *wf* **show** *insert a M* ∈ ?*W*

**proof** (*induct arbitrary: M*)

**fix** *M a*

**assume** *M* ∈ ?*W* **and** [*intro*]: *finite M*

**assume** *hyp*: ∧*b M*. (*b*, *a*) ∈ *r* ⇒ *M* ∈ ?*W* ⇒ *finite M* ⇒ *insert b M*

∈ ?*W*

{

**fix** *N M* :: 'a set

**assume** *finite N* *finite M*

**then**

**have** [ *M* ∈ ?*W* ; (∧*y*. *y* ∈ *N* ⇒ (*y*, *a*) ∈ *r*) ] ⇒ *N* ∪ *M* ∈ ?*W*

**by** (*induct N arbitrary: M*) (*auto simp: hyp*)

```

}
note add-less = this

show insert a M ∈ ?W
proof (rule accI)
  fix N assume Nless: (N, insert a M) ∈ max-ext r
  hence asm1:  $\bigwedge x. x \in N \implies (x, a) \in r \vee (\exists y \in M. (x, y) \in r)$ 
    by (auto elim!: max-ext.cases)

  let ?N1 = { n ∈ N. (n, a) ∈ r }
  let ?N2 = { n ∈ N. (n, a) ∉ r }
  have N: ?N1 ∪ ?N2 = N by (rule set-ext) auto
  from Nless have finite N by (auto elim: max-ext.cases)
  then have finites: finite ?N1 finite ?N2 by auto

  have ?N2 ∈ ?W
  proof cases
    assume [simp]: M = {}
    have Mw: {} ∈ ?W by (rule accI) (auto elim: max-ext.cases)

    from asm1 have ?N2 = {} by auto
    with Mw show ?N2 ∈ ?W by (simp only:)
  next
    assume M ≠ {}
    have N2: (?N2, M) ∈ max-ext r
      by (rule max-extI[OF - - ⟨M ≠ {}⟩]) (insert asm1, auto intro: finites)

    with ⟨M ∈ ?W⟩ show ?N2 ∈ ?W by (rule acc-downward)
  qed
  with finites have ?N1 ∪ ?N2 ∈ ?W
    by (rule add-less) simp
  then show N ∈ ?W by (simp only: N)
qed
qed
qed
next
  assume [simp]: ¬ finite M
  show ?thesis
    by (rule accI) (auto elim: max-ext.cases)
qed
qed

lemma max-ext-additive:
  (A, B) ∈ max-ext R  $\implies$  (C, D) ∈ max-ext R  $\implies$ 
  (A ∪ C, B ∪ D) ∈ max-ext R
  by (force elim!: max-ext.cases)

definition

```

$min-ext :: ('a \times 'a) set \Rightarrow ('a set \times 'a set) set$   
**where**  
 $[code del]: min-ext\ r = \{(X, Y) \mid X\ Y.\ X \neq \{\}\} \wedge (\forall y \in Y. (\exists x \in X. (x, y) \in r))\}$

**lemma** *min-ext-wf*:  
**assumes** *wf r*  
**shows** *wf (min-ext r)*  
**proof** (*rule wfI-min*)  
**fix** *Q* :: *'a set set*  
**fix** *x*  
**assume** *nonempty*:  $x \in Q$   
**show**  $\exists m \in Q. (\forall n. (n, m) \in min-ext\ r \longrightarrow n \notin Q)$   
**proof** *cases*  
**assume**  $Q = \{\{\}\}$  **thus** *?thesis* **by** (*simp add: min-ext-def*)  
**next**  
**assume**  $Q \neq \{\{\}\}$   
**with** *nonempty*  
**obtain** *e x* **where**  $x \in Q\ e \in x$  **by** *force*  
**then have**  $eU: e \in \bigcup Q$  **by** *auto*  
**with**  $\langle wf\ r \rangle$   
**obtain** *z* **where**  $z: z \in \bigcup Q \wedge y. (y, z) \in r \Longrightarrow y \notin \bigcup Q$   
**by** (*erule wfE-min*)  
**from** *z* **obtain** *m* **where**  $m \in Q\ z \in m$  **by** *auto*  
**from**  $\langle m \in Q \rangle$   
**show** *?thesis*  
**proof** (*rule, intro bexI allI impI*)  
**fix** *n*  
**assume** *smaller*:  $(n, m) \in min-ext\ r$   
**with**  $\langle z \in m \rangle$  **obtain** *y* **where**  $y: y \in n\ (y, z) \in r$  **by** (*auto simp: min-ext-def*)  
**then show**  $n \notin Q$  **using**  $z(2)$  **by** *auto*  
**qed**  
**qed**  
**qed**

Wellfoundedness of *same-fst*

**definition**

$same-fst :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow ('b * 'b) set) \Rightarrow (('a * 'b) * ('a * 'b)) set$   
**where**  
 $same-fst\ P\ R == \{((x', y'), (x, y)) \mid x' = x \ \&\ P\ x \ \&\ (y', y) : R\ x\}$   
— For *rec-def* declarations where the first *n* parameters stay unchanged in the recursive call.

**lemma** *same-fstI* [*intro!*]:  
 $[| P\ x; (y', y) : R\ x |] \Longrightarrow ((x, y'), (x, y)) : same-fst\ P\ R$   
**by** (*simp add: same-fst-def*)

**lemma** *wf-same-fst*:  
**assumes** *prem*:  $(!!x. P\ x \Longrightarrow wf(R\ x))$



```

  shows wf(same-fst P R)
apply (simp cong del: imp-cong add: wf-def same-fst-def)
apply (intro strip)
apply (rename-tac a b)
apply (case-tac wf (R a))
  apply (erule-tac a = b in wf-induct, blast)
apply (blast intro: prem)
done

```

## 21.9 Weakly decreasing sequences (w.r.t. some well-founded order) stabilize.

This material does not appear to be used any longer.

**lemma** *sequence-trans*:  $[\![ \text{ALL } i. (f (\text{Suc } i), f i) : r^* ]\!] \implies (f (i+k), f i) : r^*$   
**by** (*induct k*) (*auto intro: rtrancl-trans*)

**lemma** *wf-weak-decr-stable*:

```

  assumes as: ALL i. (f (Suc i), f i) : r^* wf (r^+)
  shows EX i. ALL k. f (i+k) = f i
proof -
  have lem: !!x. [ [ ALL i. (f (Suc i), f i) : r^*; wf (r^+) ] ]
    ==> ALL m. f m = x --> (EX i. ALL k. f (m+i+k) = f (m+i))
  apply (erule wf-induct, clarify)
  apply (case-tac EX j. (f (m+j), f m) : r^+)
  apply clarify
  apply (subgoal-tac EX i. ALL k. f ((m+j) + i+k) = f ((m+j) + i))
  apply clarify
  apply (rule-tac x = j+i in exI)
  apply (simp add: add-ac, blast)
  apply (rule-tac x = 0 in exI, clarsimp)
  apply (drule-tac i = m and k = k in sequence-trans)
  apply (blast elim: rtranclE dest: rtrancl-into-trancl1)
done

```

```

  from lem[OF as, THEN spec, of 0, simplified]
  show ?thesis by auto
qed

```

**lemma** *weak-decr-stable*:

```

  ALL i. f (Suc i) <= ((f i)::nat) ==> EX i. ALL k. f (i+k) = f i
apply (rule-tac r = pred-nat in wf-weak-decr-stable)
apply (simp add: pred-nat-trancl-eq-le)
apply (intro wf-trancl wf-pred-nat)
done

```

## 21.10 size of a datatype value

**use** *Tools/function-package/size.ML*

```

setup Size.setup

lemma size-bool [code]:
  size (b::bool) = 0 by (cases b) auto

lemma nat-size [simp, code]: size (n::nat) = n
  by (induct n) simp-all

declare prod.size [noatp]

lemma [code]:
  size (P :: 'a Predicate.pred) = 0 by (cases P) simp

lemma [code]:
  pred-size f P = 0 by (cases P) simp

end

```

## 22 FunDef: Function Definitions and Termination Proofs

```

theory FunDef
imports Wellfounded
uses
  Tools/prop-logic.ML
  Tools/sat-solver.ML
  (Tools/function-package/fundef-lib.ML)
  (Tools/function-package/fundef-common.ML)
  (Tools/function-package/inductive-wrap.ML)
  (Tools/function-package/context-tree.ML)
  (Tools/function-package/fundef-core.ML)
  (Tools/function-package/sum-tree.ML)
  (Tools/function-package/mutual.ML)
  (Tools/function-package/pattern-split.ML)
  (Tools/function-package/fundef-package.ML)
  (Tools/function-package/auto-term.ML)
  (Tools/function-package/measure-functions.ML)
  (Tools/function-package/lexicographic-order.ML)
  (Tools/function-package/fundef-datatype.ML)
  (Tools/function-package/induction-scheme.ML)
  (Tools/function-package/termination.ML)
  (Tools/function-package/decompose.ML)
  (Tools/function-package/descent.ML)
  (Tools/function-package/scnp-solve.ML)
  (Tools/function-package/scnp-reconstruct.ML)
begin

```

## 22.1 Definitions with default value.

**definition**

$THE\text{-}default :: 'a \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a$  **where**  
 $THE\text{-}default\ d\ P = (if\ (\exists!x. P\ x)\ then\ (THE\ x. P\ x)\ else\ d)$

**lemma**  $THE\text{-}defaultI'$ :  $\exists!x. P\ x \Longrightarrow P\ (THE\text{-}default\ d\ P)$   
**by** (*simp add: theI' THE-default-def*)

**lemma**  $THE\text{-}default1\text{-equality}$ :

$\llbracket \exists!x. P\ x; P\ a \rrbracket \Longrightarrow THE\text{-}default\ d\ P = a$   
**by** (*simp add: the1-equality THE-default-def*)

**lemma**  $THE\text{-}default\text{-none}$ :

$\neg(\exists!x. P\ x) \Longrightarrow THE\text{-}default\ d\ P = d$   
**by** (*simp add: THE-default-def*)

**lemma**  $fundef\text{-}ex1\text{-existence}$ :

**assumes**  $f\text{-def}$ :  $f == (\lambda x::'a. THE\text{-}default\ (d\ x)\ (\lambda y. G\ x\ y))$   
**assumes**  $ex1$ :  $\exists!y. G\ x\ y$   
**shows**  $G\ x\ (f\ x)$   
**apply** (*simp only: f-def*)  
**apply** (*rule THE-defaultI'*)  
**apply** (*rule ex1*)  
**done**

**lemma**  $fundef\text{-}ex1\text{-uniqueness}$ :

**assumes**  $f\text{-def}$ :  $f == (\lambda x::'a. THE\text{-}default\ (d\ x)\ (\lambda y. G\ x\ y))$   
**assumes**  $ex1$ :  $\exists!y. G\ x\ y$   
**assumes**  $elm$ :  $G\ x\ (h\ x)$   
**shows**  $h\ x = f\ x$   
**apply** (*simp only: f-def*)  
**apply** (*rule THE-default1-equality [symmetric]*)  
**apply** (*rule ex1*)  
**apply** (*rule elm*)  
**done**

**lemma**  $fundef\text{-}ex1\text{-iff}$ :

**assumes**  $f\text{-def}$ :  $f == (\lambda x::'a. THE\text{-}default\ (d\ x)\ (\lambda y. G\ x\ y))$   
**assumes**  $ex1$ :  $\exists!y. G\ x\ y$   
**shows**  $(G\ x\ y) = (f\ x = y)$   
**apply** (*auto simp: ex1 f-def THE-default1-equality*)  
**apply** (*rule THE-defaultI'*)  
**apply** (*rule ex1*)  
**done**

**lemma**  $fundef\text{-default-value}$ :

**assumes**  $f\text{-def}$ :  $f == (\lambda x::'a. THE\text{-}default\ (d\ x)\ (\lambda y. G\ x\ y))$   
**assumes**  $graph$ :  $\bigwedge x\ y. G\ x\ y \Longrightarrow D\ x$

```

assumes  $\neg D\ x$ 
shows  $f\ x = d\ x$ 
proof -
  have  $\neg(\exists y. G\ x\ y)$ 
  proof
    assume  $\exists y. G\ x\ y$ 
    hence  $D\ x$  using graph ..
    with  $\neg D\ x$  show False ..
  qed
hence  $\neg(\exists!y. G\ x\ y)$  by blast

thus ?thesis
  unfolding f-def
  by (rule THE-default-none)
qed

```

```

definition in-rel-def[simp]:
  in-rel  $R\ x\ y == (x, y) \in R$ 

```

```

lemma wf-in-rel:
  wf  $R \implies wfP\ (in-rel\ R)$ 
by (simp add: wfP-def)

```

```

use Tools/function-package/fundef-lib.ML
use Tools/function-package/fundef-common.ML
use Tools/function-package/inductive-wrap.ML
use Tools/function-package/context-tree.ML
use Tools/function-package/fundef-core.ML
use Tools/function-package/sum-tree.ML
use Tools/function-package/mutual.ML
use Tools/function-package/pattern-split.ML
use Tools/function-package/auto-term.ML
use Tools/function-package/fundef-package.ML
use Tools/function-package/fundef-datatype.ML
use Tools/function-package/induction-scheme.ML

```

```

setup ⟨⟨
  FundefPackage.setup
  #> FundefDatatype.setup
  #> InductionScheme.setup
  ⟩⟩

```

## 22.2 Measure Functions

```

inductive is-measure ::  $('a \Rightarrow nat) \Rightarrow bool$ 
where is-measure-trivial: is-measure  $f$ 

```

```

use Tools/function-package/measure-functions.ML
setup MeasureFunctions.setup

```

**lemma** *measure-size*[*measure-function*]: *is-measure size*  
**by** (*rule is-measure-trivial*)

**lemma** *measure-fst*[*measure-function*]: *is-measure f*  $\implies$  *is-measure* ( $\lambda p. f (fst p)$ )  
**by** (*rule is-measure-trivial*)

**lemma** *measure-snd*[*measure-function*]: *is-measure f*  $\implies$  *is-measure* ( $\lambda p. f (snd p)$ )  
**by** (*rule is-measure-trivial*)

**use** *Tools/function-package/lexicographic-order.ML*  
**setup** *LexicographicOrder.setup*

### 22.3 Congruence Rules

**lemma** *let-cong* [*fundef-cong*]:  
 $M = N \implies (\bigwedge x. x = N \implies f x = g x) \implies Let\ M\ f = Let\ N\ g$   
**unfolding** *Let-def* **by** *blast*

**lemmas** [*fundef-cong*] =  
*if-cong image-cong INT-cong UN-cong*  
*bex-cong ball-cong imp-cong*

**lemma** *split-cong* [*fundef-cong*]:  
 $(\bigwedge x\ y. (x, y) = q \implies f\ x\ y = g\ x\ y) \implies p = q$   
 $\implies split\ f\ p = split\ g\ q$   
**by** (*auto simp: split-def*)

**lemma** *comp-cong* [*fundef-cong*]:  
 $f\ (g\ x) = f'\ (g'\ x') \implies (f\ o\ g)\ x = (f'\ o\ g')\ x'$   
**unfolding** *o-apply* .

### 22.4 Simp rules for termination proofs

**lemma** *termination-basic-simps*[*termination-simp*]:  
 $x < (y::nat) \implies x < y + z$   
 $x < z \implies x < y + z$   
 $x \leq y \implies x \leq y + (z::nat)$   
 $x \leq z \implies x \leq y + (z::nat)$   
 $x < y \implies x \leq (y::nat)$   
**by** *arith+*

**declare** *le-imp-less-Suc*[*termination-simp*]

**lemma** *prod-size-simp*[*termination-simp*]:  
 $prod\text{-}size\ f\ g\ p = f\ (fst\ p) + g\ (snd\ p) + Suc\ 0$   
**by** (*induct p*) *auto*

## 22.5 Decomposition

**lemma** *less-by-empty*:

$$A = \{\} \implies A \subseteq B$$

**and** *union-comp-emptyL*:

$$\llbracket A \ O \ C = \{\}; B \ O \ C = \{\} \rrbracket \implies (A \cup B) \ O \ C = \{\}$$

**and** *union-comp-emptyR*:

$$\llbracket A \ O \ B = \{\}; A \ O \ C = \{\} \rrbracket \implies A \ O \ (B \cup C) = \{\}$$

**and** *wf-no-loop*:

$$R \ O \ R = \{\} \implies wf \ R$$

**by** (*auto simp add: wf-comp-self*[of *R*])

## 22.6 Reduction Pairs

**definition**

$$reduction\text{-}pair \ P = (wf \ (fst \ P) \wedge snd \ P \ O \ fst \ P \subseteq fst \ P)$$

**lemma** *reduction-pairI*[*intro*]:  $wf \ R \implies S \ O \ R \subseteq R \implies reduction\text{-}pair \ (R, S)$

**unfolding** *reduction-pair-def* **by** *auto*

**lemma** *reduction-pair-lemma*:

**assumes** *rp*: *reduction-pair* *P*

**assumes**  $R \subseteq fst \ P$

**assumes**  $S \subseteq snd \ P$

**assumes** *wf* *S*

**shows** *wf*  $(R \cup S)$

**proof** –

**from** *rp*  $\langle S \subseteq snd \ P \rangle$  **have**  $wf \ (fst \ P) \ S \ O \ fst \ P \subseteq fst \ P$

**unfolding** *reduction-pair-def* **by** *auto*

**with**  $\langle wf \ S \rangle$  **have**  $wf \ (fst \ P \cup S)$

**by** (*auto intro: wf-union-compatible*)

**moreover from**  $\langle R \subseteq fst \ P \rangle$  **have**  $R \cup S \subseteq fst \ P \cup S$  **by** *auto*

**ultimately show** *?thesis* **by** (*rule wf-subset*)

**qed**

**definition**

$$rp\text{-}inv\text{-}image = (\lambda(R,S) \ f. (inv\text{-}image \ R \ f, inv\text{-}image \ S \ f))$$

**lemma** *rp-inv-image-rp*:

$$reduction\text{-}pair \ P \implies reduction\text{-}pair \ (rp\text{-}inv\text{-}image \ P \ f)$$

**unfolding** *reduction-pair-def* *rp-inv-image-def* *split-def*

**by** *force*

## 22.7 Concrete orders for SCNP termination proofs

**definition** *pair-less* = *less-than*  $\langle *lex* \rangle$  *less-than*

**definition** [*code del*]: *pair-leq* = *pair-less*  $\hat{=}$

**definition** *max-strict* = *max-ext* *pair-less*

**definition** [*code del*]: *max-weak* = *max-ext* *pair-leq*  $\cup \{(\{\}, \{\})\}$

**definition** [*code del*]: *min-strict* = *min-ext* *pair-less*

**definition** `[code del]: min-weak = min-ext pair-leq  $\cup \{(\{\}, \{\})\}$`

**lemma** `wf-pair-less[simp]: wf pair-less`  
**by** `(auto simp: pair-less-def)`

Introduction rules for *pair-less/pair-leq*

**lemma** `pair-leqI1:  $a < b \implies ((a, s), (b, t)) \in \text{pair-leq}$`   
**and** `pair-leqI2:  $a \leq b \implies s \leq t \implies ((a, s), (b, t)) \in \text{pair-leq}$`   
**and** `pair-lessI1:  $a < b \implies ((a, s), (b, t)) \in \text{pair-less}$`   
**and** `pair-lessI2:  $a \leq b \implies s < t \implies ((a, s), (b, t)) \in \text{pair-less}$`   
**unfolding** `pair-leq-def pair-less-def` **by** `auto`

Introduction rules for *max*

**lemma** `smax-emptyI:`  
`finite Y  $\implies Y \neq \{\} \implies (\{\}, Y) \in \text{max-strict}$`   
**and** `smax-insertI:`  
 `$\llbracket y \in Y; (x, y) \in \text{pair-less}; (X, Y) \in \text{max-strict} \rrbracket \implies (\text{insert } x \ X, Y) \in \text{max-strict}$`   
**and** `wmax-emptyI:`  
`finite X  $\implies (\{\}, X) \in \text{max-weak}$`   
**and** `wmax-insertI:`  
 `$\llbracket y \in YS; (x, y) \in \text{pair-leq}; (XS, YS) \in \text{max-weak} \rrbracket \implies (\text{insert } x \ XS, YS) \in \text{max-weak}$`   
**unfolding** `max-strict-def max-weak-def` **by** `(auto elim!: max-ext.cases)`

Introduction rules for *min*

**lemma** `smin-emptyI:`  
 `$X \neq \{\} \implies (X, \{\}) \in \text{min-strict}$`   
**and** `smin-insertI:`  
 `$\llbracket x \in XS; (x, y) \in \text{pair-less}; (XS, YS) \in \text{min-strict} \rrbracket \implies (XS, \text{insert } y \ YS) \in \text{min-strict}$`   
**and** `wmin-emptyI:`  
 `$(X, \{\}) \in \text{min-weak}$`   
**and** `wmin-insertI:`  
 `$\llbracket x \in XS; (x, y) \in \text{pair-leq}; (XS, YS) \in \text{min-weak} \rrbracket \implies (XS, \text{insert } y \ YS) \in \text{min-weak}$`   
**by** `(auto simp: min-strict-def min-weak-def min-ext-def)`

Reduction Pairs

**lemma** `max-ext-compat:`  
**assumes** `S O R  $\subseteq$  R`  
**shows**  `$(\text{max-ext } S \cup \{(\{\}, \{\})\}) O \text{max-ext } R \subseteq \text{max-ext } R$`   
**using** `assms`  
**apply** `auto`  
**apply** `(elim max-ext.cases)`  
**apply** `rule`  
**apply** `auto[3]`  
**apply** `(drule-tac x=xa in meta-spec)`  
**apply** `simp`

```

apply (erule bexE)
apply (drule-tac x=xb in meta-spec)
by auto

lemma max-rpair-set: reduction-pair (max-strict, max-weak)
  unfolding max-strict-def max-weak-def
apply (intro reduction-pairI max-ext-wf)
apply simp
apply (rule max-ext-compat)
by (auto simp: pair-less-def pair-leq-def)

lemma min-ext-compat:
  assumes S O R  $\subseteq R$ 
  shows (min-ext S  $\cup \{(\{\}, \{\})\}$ ) O min-ext R  $\subseteq$  min-ext R
using assms
apply (auto simp: min-ext-def)
apply (drule-tac x=ya in bspec, assumption)
apply (erule bexE)
apply (drule-tac x=xc in bspec)
apply assumption
by auto

lemma min-rpair-set: reduction-pair (min-strict, min-weak)
  unfolding min-strict-def min-weak-def
apply (intro reduction-pairI min-ext-wf)
apply simp
apply (rule min-ext-compat)
by (auto simp: pair-less-def pair-leq-def)

```

## 22.8 Tool setup

```

use Tools/function-package/termination.ML
use Tools/function-package/decompose.ML
use Tools/function-package/descent.ML
use Tools/function-package/scnp-solve.ML
use Tools/function-package/scnp-reconstruct.ML

setup  $\ll$  ScnpReconstruct.setup  $\gg$ 

ML-val — setup inactive
 $\ll$ 
  Context.theory-map (FundefCommon.set-termination-prover (ScnpReconstruct.decomp-scnp
    [ScnpSolve.MAX, ScnpSolve.MIN, ScnpSolve.MS]))
 $\gg$ 

end

```



## 23 Record: Extensible records with structural subtyping

```

theory Record
imports Product-Type
uses (Tools/record-package.ML)
begin

```

```

lemma prop-subst:  $s = t \implies PROP\ P\ t \implies PROP\ P\ s$ 
  by simp

```

```

lemma rec-UNIV-I:  $\bigwedge x. x \in UNIV \equiv True$ 
  by simp

```

```

lemma rec-True-simp:  $(True \implies PROP\ P) \equiv PROP\ P$ 
  by simp

```

```

lemma K-record-comp:  $(\lambda x. c) \circ f = (\lambda x. c)$ 
  by (simp add: comp-def)

```

### 23.1 Concrete record syntax

**nonterminals**

*ident field-type field-types field fields update updates*

**syntax**

```

-constify      ::  $id \Rightarrow ident$                 (-)
-constify      ::  $longid \Rightarrow ident$              (-)

-field-type     ::  $[ident, type] \Rightarrow field-type$     ((2- ::/ -))
               ::  $field-type \Rightarrow field-types$       (-)
-field-types    ::  $[field-type, field-types] \Rightarrow field-types$   (-,/ -)
-record-type    ::  $field-types \Rightarrow type$            ((3'(| - |')))
-record-type-scheme ::  $[field-types, type] \Rightarrow type$   ((3'(| -,/ (2... ::/ -) |')))

-field          ::  $[ident, 'a] \Rightarrow field$            ((2- =/ -))
               ::  $field \Rightarrow fields$                 (-)
-fields         ::  $[field, fields] \Rightarrow fields$       (-,/ -)
-record         ::  $fields \Rightarrow 'a$                    ((3'(| - |')))
-record-scheme  ::  $[fields, 'a] \Rightarrow 'a$              ((3'(| -,/ (2... =/ -) |')))

-update-name    ::  $idt$ 
-update         ::  $[ident, 'a] \Rightarrow update$           ((2- :=/ -))
               ::  $update \Rightarrow updates$               (-)
-updates        ::  $[update, updates] \Rightarrow updates$   (-,/ -)
-record-update  ::  $['a, updates] \Rightarrow 'b$            (-/(3'(| - |') [900,0] 900))

```

**syntax** (*xsymbols*)

```

-record-type    ::  $field-types \Rightarrow type$            ((3(|-)))
-record-type-scheme ::  $[field-types, type] \Rightarrow type$   ((3(|-,/ (2... ::/ -)|)))

```

```

-record          :: fields => 'a
-record-scheme   :: [fields, 'a] => 'a
-record-update    :: ['a, updates] => 'b

((3|-|))
((3|-/ (2... =/ -)|))
(-/(3|-|) [900,0] 900)

use Tools/record-package.ML
setup RecordPackage.setup

end

```

## 24 Option: Datatype option

```

theory Option
imports Datatype Finite-Set
begin

```

```

datatype 'a option = None | Some 'a

```

```

lemma not-None-eq [iff]: (x ~ = None) = (EX y. x = Some y)
  by (induct x) auto

```

```

lemma not-Some-eq [iff]: (ALL y. x ~ = Some y) = (x = None)
  by (induct x) auto

```

Although it may appear that both of these equalities are helpful only when applied to assumptions, in practice it seems better to give them the uniform iff attribute.

```

lemma option-caseE:
  assumes c: (case x of None => P | Some y => Q y)
  obtains
    (None) x = None and P
  | (Some) y where x = Some y and Q y
  using c by (cases x) simp-all

```

```

lemma insert-None-conv-UNIV: insert None (range Some) = UNIV
  by (rule set-ext, case-tac x) auto

```

```

instance option :: (finite) finite proof
qed (simp add: insert-None-conv-UNIV [symmetric])

```

```

lemma inj-Some [simp]: inj-on Some A
  by (rule inj-onI) simp

```

### 24.0.1 Operations

```

primrec the :: 'a option => 'a where
the (Some x) = x

```

```

primrec set :: 'a option => 'a set where

```

```

set None = {} |
set (Some x) = {x}

```

```

lemma ospec [dest]: (ALL x:set A. P x) ==> A = Some x ==> P x
by simp

```

```

declaration << fn - ==>
  Classical.map-cs (fn cs ==> cs addSD2 (ospec, thm ospec))
>>

```

```

lemma elem-set [iff]: (x : set xo) = (xo = Some x)
by (cases xo) auto

```

```

lemma set-empty-eq [simp]: (set xo = {}) = (xo = None)
by (cases xo) auto

```

```

definition
  map :: ('a => 'b) => 'a option => 'b option
where
  [code del]: map = (%f y. case y of None ==> None | Some x ==> Some (f x))

```

```

lemma option-map-None [simp, code]: map f None = None
by (simp add: map-def)

```

```

lemma option-map-Some [simp, code]: map f (Some x) = Some (f x)
by (simp add: map-def)

```

```

lemma option-map-is-None [iff]:
  (map f opt = None) = (opt = None)
by (simp add: map-def split add: option.split)

```

```

lemma option-map-eq-Some [iff]:
  (map f xo = Some y) = (EX z. xo = Some z & f z = y)
by (simp add: map-def split add: option.split)

```

```

lemma option-map-comp:
  map f (map g opt) = map (f o g) opt
by (simp add: map-def split add: option.split)

```

```

lemma option-map-o-sum-case [simp]:
  map f o sum-case g h = sum-case (map f o g) (map f o h)
by (rule ext) (simp split: sum.split)

```

```

hide (open) const set map

```

## 24.0.2 Code generator setup

```

definition

```

```

is-none :: 'a option ⇒ bool where
is-none-none [code post, symmetric, code inline]: is-none x ⟷ x = None

lemma is-none-code [code]:
  shows is-none None ⟷ True
  and is-none (Some x) ⟷ False
  unfolding is-none-none [symmetric] by simp-all

hide (open) const is-none

code-type option
  (SML - option)
  (OCaml - option)
  (Haskell Maybe -)

code-const None and Some
  (SML NONE and SOME)
  (OCaml None and Some -)
  (Haskell Nothing and Just)

code-instance option :: eq
  (Haskell -)

code-const eq-class.eq :: 'a::eq option ⇒ 'a option ⇒ bool
  (Haskell infixl 4 ==)

code-reserved SML
  option NONE SOME

code-reserved OCaml
  option None Some

end

```

## 25 Extraction: Program extraction for HOL

```

theory Extraction
imports Option
uses Tools/rewrite-hol-proof.ML
begin

```

### 25.1 Setup

```

setup ⟨⟨
  let
  fun realizes-set-proc (Const (realizes, Type (fun, [Type (Null, []), -])) $ r $
    (Const (op :, -) $ x $ S) = (case strip-comb S of
      (Var (ixn, U), ts) => SOME (list-comb (Var (ixn, U), ts @ [x]))

```

```

| (Free (s, U), ts) => SOME (list-comb (Free (s, U), ts @ [x]))
| - => NONE)
| realizes-set-proc (Const (realizes, Type (fun, [T, -])) $ r $
  (Const (op :, -) $ x $ S)) = (case strip-comb S of
    (Var (ixn, U), ts) => SOME (list-comb (Var (ixn, T --> U), r :: ts @
[x]))
| (Free (s, U), ts) => SOME (list-comb (Free (s, T --> U), r :: ts @ [x]))
| - => NONE)
| realizes-set-proc - = NONE;

```

in

```

Extraction.add-types
  [(bool, ([], NONE))] #>
Extraction.set-preprocessor (fn thy =>
  Proofterm.rewrite-proof-notypes
    ([], RewriteHOLProof.elim-cong :: ProofRewriteRules.rprocs true) o
  Proofterm.rewrite-proof thy
    (RewriteHOLProof.rews, ProofRewriteRules.rprocs true) o
  ProofRewriteRules.elim-vars (curry Const @{const-name default}))
end
>>

```

**lemmas** [extraction-expand] =  
 meta-spec atomize-eq atomize-all atomize-imp atomize-conj  
 allE rev-mp conjE Eq-TrueI Eq-FalseI eqTrueI eqTrueE eq-cong2  
 notE' impE' impE iffE imp-cong simp-thms eq-True eq-False  
 induct-forall-eq induct-implies-eq induct-equal-eq induct-conj-eq  
 induct-forall-def induct-implies-def induct-equal-def induct-conj-def  
 induct-atomize induct-rulify induct-rulify-fallback  
 True-implies-equals TrueE

**datatype** *sumbool* = *Left* | *Right*

## 25.2 Type of extracted program

### extract-type

*typeof* (Trueprop *P*)  $\equiv$  *typeof* *P*

*typeof* *P*  $\equiv$  *Type* (*TYPE*(*Null*))  $\implies$  *typeof* *Q*  $\equiv$  *Type* (*TYPE*('Q))  $\implies$   
*typeof* (*P*  $\longrightarrow$  *Q*)  $\equiv$  *Type* (*TYPE*('Q))

*typeof* *Q*  $\equiv$  *Type* (*TYPE*(*Null*))  $\implies$  *typeof* (*P*  $\longrightarrow$  *Q*)  $\equiv$  *Type* (*TYPE*(*Null*))

*typeof* *P*  $\equiv$  *Type* (*TYPE*('P))  $\implies$  *typeof* *Q*  $\equiv$  *Type* (*TYPE*('Q))  $\implies$   
*typeof* (*P*  $\longrightarrow$  *Q*)  $\equiv$  *Type* (*TYPE*('P  $\Rightarrow$  'Q))

( $\lambda x.$  *typeof* (*P* *x*))  $\equiv$  ( $\lambda x.$  *Type* (*TYPE*(*Null*)))  $\implies$   
*typeof* ( $\forall x.$  *P* *x*)  $\equiv$  *Type* (*TYPE*(*Null*))

$$(\lambda x. \text{typeof } (P \ x)) \equiv (\lambda x. \text{Type } (\text{TYPE}('P))) \implies \\ \text{typeof } (\forall x::'a. P \ x) \equiv \text{Type } (\text{TYPE}('a \Rightarrow 'P))$$

$$(\lambda x. \text{typeof } (P \ x)) \equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\ \text{typeof } (\exists x::'a. P \ x) \equiv \text{Type } (\text{TYPE}('a))$$

$$(\lambda x. \text{typeof } (P \ x)) \equiv (\lambda x. \text{Type } (\text{TYPE}('P))) \implies \\ \text{typeof } (\exists x::'a. P \ x) \equiv \text{Type } (\text{TYPE}('a \times 'P))$$

$$\text{typeof } P \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\ \text{typeof } (P \vee Q) \equiv \text{Type } (\text{TYPE}(\text{sumbool}))$$

$$\text{typeof } P \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\ \text{typeof } (P \vee Q) \equiv \text{Type } (\text{TYPE}('Q \text{ option}))$$

$$\text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\ \text{typeof } (P \vee Q) \equiv \text{Type } (\text{TYPE}('P \text{ option}))$$

$$\text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\ \text{typeof } (P \vee Q) \equiv \text{Type } (\text{TYPE}('P + 'Q))$$

$$\text{typeof } P \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\ \text{typeof } (P \wedge Q) \equiv \text{Type } (\text{TYPE}('Q))$$

$$\text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\ \text{typeof } (P \wedge Q) \equiv \text{Type } (\text{TYPE}('P))$$

$$\text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\ \text{typeof } (P \wedge Q) \equiv \text{Type } (\text{TYPE}('P \times 'Q))$$

$$\text{typeof } (P = Q) \equiv \text{typeof } ((P \longrightarrow Q) \wedge (Q \longrightarrow P))$$

$$\text{typeof } (x \in P) \equiv \text{typeof } P$$

### 25.3 Realizability

#### realizability

$$(\text{realizes } t \ (\text{Trueprop } P)) \equiv (\text{Trueprop } (\text{realizes } t \ P))$$

$$(\text{typeof } P) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\ (\text{realizes } t \ (P \longrightarrow Q)) \equiv (\text{realizes } \text{Null } P \longrightarrow \text{realizes } t \ Q)$$

$$(\text{typeof } P) \equiv (\text{Type } (\text{TYPE}('P))) \implies \\ (\text{typeof } Q) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\ (\text{realizes } t \ (P \longrightarrow Q)) \equiv (\forall x::'P. \text{realizes } x \ P \longrightarrow \text{realizes } \text{Null } Q)$$

$$(\text{realizes } t \ (P \longrightarrow Q)) \equiv (\forall x. \text{realizes } x \ P \longrightarrow \text{realizes } (t \ x) \ Q)$$

$$(\lambda x. \text{typeof } (P \ x)) \equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies$$

$$\begin{aligned}
& (\text{realizes } t \ (\forall x. P \ x)) \equiv (\forall x. \text{realizes } \text{Null} \ (P \ x)) \\
& (\text{realizes } t \ (\forall x. P \ x)) \equiv (\forall x. \text{realizes } (t \ x) \ (P \ x)) \\
& (\lambda x. \text{typeof} \ (P \ x)) \equiv (\lambda x. \text{Type} \ (\text{TYPE}(\text{Null}))) \implies \\
& \quad (\text{realizes } t \ (\exists x. P \ x)) \equiv (\text{realizes } \text{Null} \ (P \ t)) \\
& (\text{realizes } t \ (\exists x. P \ x)) \equiv (\text{realizes } (\text{snd } t) \ (P \ (\text{fst } t))) \\
& (\text{typeof } P) \equiv (\text{Type} \ (\text{TYPE}(\text{Null}))) \implies \\
& (\text{typeof } Q) \equiv (\text{Type} \ (\text{TYPE}(\text{Null}))) \implies \\
& \quad (\text{realizes } t \ (P \vee Q)) \equiv \\
& \quad (\text{case } t \text{ of } \text{Left} \Rightarrow \text{realizes } \text{Null } P \mid \text{Right} \Rightarrow \text{realizes } \text{Null } Q) \\
& (\text{typeof } P) \equiv (\text{Type} \ (\text{TYPE}(\text{Null}))) \implies \\
& \quad (\text{realizes } t \ (P \vee Q)) \equiv \\
& \quad (\text{case } t \text{ of } \text{None} \Rightarrow \text{realizes } \text{Null } P \mid \text{Some } q \Rightarrow \text{realizes } q \ Q) \\
& (\text{typeof } Q) \equiv (\text{Type} \ (\text{TYPE}(\text{Null}))) \implies \\
& \quad (\text{realizes } t \ (P \vee Q)) \equiv \\
& \quad (\text{case } t \text{ of } \text{None} \Rightarrow \text{realizes } \text{Null } Q \mid \text{Some } p \Rightarrow \text{realizes } p \ P) \\
& (\text{realizes } t \ (P \vee Q)) \equiv \\
& \quad (\text{case } t \text{ of } \text{Inl } p \Rightarrow \text{realizes } p \ P \mid \text{Inr } q \Rightarrow \text{realizes } q \ Q) \\
& (\text{typeof } P) \equiv (\text{Type} \ (\text{TYPE}(\text{Null}))) \implies \\
& \quad (\text{realizes } t \ (P \wedge Q)) \equiv (\text{realizes } \text{Null } P \wedge \text{realizes } t \ Q) \\
& (\text{typeof } Q) \equiv (\text{Type} \ (\text{TYPE}(\text{Null}))) \implies \\
& \quad (\text{realizes } t \ (P \wedge Q)) \equiv (\text{realizes } t \ P \wedge \text{realizes } \text{Null } Q) \\
& (\text{realizes } t \ (P \wedge Q)) \equiv (\text{realizes } (\text{fst } t) \ P \wedge \text{realizes } (\text{snd } t) \ Q) \\
& \text{typeof } P \equiv \text{Type} \ (\text{TYPE}(\text{Null})) \implies \\
& \quad \text{realizes } t \ (\neg P) \equiv \neg \text{realizes } \text{Null } P \\
& \text{typeof } P \equiv \text{Type} \ (\text{TYPE}('P)) \implies \\
& \quad \text{realizes } t \ (\neg P) \equiv (\forall x::'P. \neg \text{realizes } x \ P) \\
& \text{typeof } (P::\text{bool}) \equiv \text{Type} \ (\text{TYPE}(\text{Null})) \implies \\
& \text{typeof } Q \equiv \text{Type} \ (\text{TYPE}(\text{Null})) \implies \\
& \quad \text{realizes } t \ (P = Q) \equiv \text{realizes } \text{Null } P = \text{realizes } \text{Null } Q \\
& (\text{realizes } t \ (P = Q)) \equiv (\text{realizes } t \ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))
\end{aligned}$$

## 25.4 Computational content of basic inference rules

**theorem** *disjE-realizer*:

**assumes**  $r$ :  $\text{case } x \text{ of } \text{Inl } p \Rightarrow P \ p \mid \text{Inr } q \Rightarrow Q \ q$

**and**  $r1: \bigwedge p. P\ p \implies R\ (f\ p)$  **and**  $r2: \bigwedge q. Q\ q \implies R\ (g\ q)$   
**shows**  $R\ (\text{case } x \text{ of } Inl\ p \Rightarrow f\ p \mid Inr\ q \Rightarrow g\ q)$   
**proof** (cases  $x$ )  
   **case**  $Inl$   
     **with**  $r$  **show** ?thesis **by** simp (rule  $r1$ )  
**next**  
   **case**  $Inr$   
     **with**  $r$  **show** ?thesis **by** simp (rule  $r2$ )  
**qed**

**theorem** *disjE-realizer2*:  
   **assumes**  $r: \text{case } x \text{ of } None \Rightarrow P \mid Some\ q \Rightarrow Q\ q$   
   **and**  $r1: P \implies R\ f$  **and**  $r2: \bigwedge q. Q\ q \implies R\ (g\ q)$   
   **shows**  $R\ (\text{case } x \text{ of } None \Rightarrow f \mid Some\ q \Rightarrow g\ q)$   
**proof** (cases  $x$ )  
   **case**  $None$   
     **with**  $r$  **show** ?thesis **by** simp (rule  $r1$ )  
**next**  
   **case**  $Some$   
     **with**  $r$  **show** ?thesis **by** simp (rule  $r2$ )  
**qed**

**theorem** *disjE-realizer3*:  
   **assumes**  $r: \text{case } x \text{ of } Left \Rightarrow P \mid Right \Rightarrow Q$   
   **and**  $r1: P \implies R\ f$  **and**  $r2: Q \implies R\ g$   
   **shows**  $R\ (\text{case } x \text{ of } Left \Rightarrow f \mid Right \Rightarrow g)$   
**proof** (cases  $x$ )  
   **case**  $Left$   
     **with**  $r$  **show** ?thesis **by** simp (rule  $r1$ )  
**next**  
   **case**  $Right$   
     **with**  $r$  **show** ?thesis **by** simp (rule  $r2$ )  
**qed**

**theorem** *conjI-realizer*:  
    $P\ p \implies Q\ q \implies P\ (fst\ (p, q)) \wedge Q\ (snd\ (p, q))$   
   **by** simp

**theorem** *exI-realizer*:  
    $P\ y\ x \implies P\ (snd\ (x, y))\ (fst\ (x, y))$  **by** simp

**theorem** *exE-realizer*:  $P\ (snd\ p)\ (fst\ p) \implies$   
    $(\bigwedge x\ y. P\ y\ x \implies Q\ (f\ x\ y)) \implies Q\ (\text{let } (x, y) = p \text{ in } f\ x\ y)$   
   **by** (cases  $p$ ) (simp add: Let-def)

**theorem** *exE-realizer'*:  $P\ (snd\ p)\ (fst\ p) \implies$   
    $(\bigwedge x\ y. P\ y\ x \implies Q) \implies Q$  **by** (cases  $p$ ) simp

**setup**  $\ll$



*Sign.add-const-constraint* (@{const-name default}, SOME @{typ 'a::type})  
 >>

**realizers**

*impI* (*P*, *Q*):  $\lambda pq. pq$   
 $\Lambda P Q pq (h: -). allI \cdot - \cdot (\Lambda x. impI \cdot - \cdot - \cdot (h \cdot x))$

*impI* (*P*): *Null*  
 $\Lambda P Q (h: -). allI \cdot - \cdot (\Lambda x. impI \cdot - \cdot - \cdot (h \cdot x))$

*impI* (*Q*):  $\lambda q. q \Lambda P Q q. impI \cdot - \cdot -$

*impI*: *Null impI*

*mp* (*P*, *Q*):  $\lambda pq. pq$   
 $\Lambda P Q pq (h: -) p. mp \cdot - \cdot - \cdot (spec \cdot - \cdot p \cdot h)$

*mp* (*P*): *Null*  
 $\Lambda P Q (h: -) p. mp \cdot - \cdot - \cdot (spec \cdot - \cdot p \cdot h)$

*mp* (*Q*):  $\lambda q. q \Lambda P Q q. mp \cdot - \cdot -$

*mp*: *Null mp*

*allI* (*P*):  $\lambda p. p \Lambda P p. allI \cdot -$

*allI*: *Null allI*

*spec* (*P*):  $\lambda x p. p x \Lambda P x p. spec \cdot - \cdot x$

*spec*: *Null spec*

*exI* (*P*):  $\lambda x p. (x, p) \Lambda P x p. exI\text{-realizer} \cdot P \cdot p \cdot x$

*exI*:  $\lambda x. x \Lambda P x (h: -). h$

*exE* (*P*, *Q*):  $\lambda p pq. let (x, y) = p in pq x y$   
 $\Lambda P Q p (h: -) pq. exE\text{-realizer} \cdot P \cdot p \cdot Q \cdot pq \cdot h$

*exE* (*P*): *Null*  
 $\Lambda P Q p. exE\text{-realizer}' \cdot - \cdot - \cdot -$

*exE* (*Q*):  $\lambda x pq. pq x$   
 $\Lambda P Q x (h1: -) pq (h2: -). h2 \cdot x \cdot h1$

*exE*: *Null*  
 $\Lambda P Q x (h1: -) (h2: -). h2 \cdot x \cdot h1$

*conjI* (*P*, *Q*): *Pair*

$$\Lambda P Q p (h: -) q. \text{conjI-realizer} \cdot P \cdot p \cdot Q \cdot q \cdot h$$

$$\begin{aligned} \text{conjI} (P): \lambda p. p \\ \Lambda P Q p. \text{conjI} \cdot \cdot \cdot \cdot \end{aligned}$$

$$\begin{aligned} \text{conjI} (Q): \lambda q. q \\ \Lambda P Q (h: -) q. \text{conjI} \cdot \cdot \cdot \cdot \cdot h \end{aligned}$$

$$\text{conjI}: \text{Null conjI}$$

$$\begin{aligned} \text{conjunct1} (P, Q): \text{fst} \\ \Lambda P Q pq. \text{conjunct1} \cdot \cdot \cdot \cdot \end{aligned}$$

$$\begin{aligned} \text{conjunct1} (P): \lambda p. p \\ \Lambda P Q p. \text{conjunct1} \cdot \cdot \cdot \cdot \end{aligned}$$

$$\begin{aligned} \text{conjunct1} (Q): \text{Null} \\ \Lambda P Q q. \text{conjunct1} \cdot \cdot \cdot \cdot \end{aligned}$$

$$\text{conjunct1}: \text{Null conjunct1}$$

$$\begin{aligned} \text{conjunct2} (P, Q): \text{snd} \\ \Lambda P Q pq. \text{conjunct2} \cdot \cdot \cdot \cdot \end{aligned}$$

$$\begin{aligned} \text{conjunct2} (P): \text{Null} \\ \Lambda P Q p. \text{conjunct2} \cdot \cdot \cdot \cdot \end{aligned}$$

$$\begin{aligned} \text{conjunct2} (Q): \lambda p. p \\ \Lambda P Q p. \text{conjunct2} \cdot \cdot \cdot \cdot \end{aligned}$$

$$\text{conjunct2}: \text{Null conjunct2}$$

$$\begin{aligned} \text{disjI1} (P, Q): \text{Inl} \\ \Lambda P Q p. \text{iffD2} \cdot \cdot \cdot \cdot \cdot (\text{sum.cases-1} \cdot P \cdot \cdot \cdot p) \end{aligned}$$

$$\begin{aligned} \text{disjI1} (P): \text{Some} \\ \Lambda P Q p. \text{iffD2} \cdot \cdot \cdot \cdot \cdot (\text{option.cases-2} \cdot \cdot \cdot P \cdot p) \end{aligned}$$

$$\begin{aligned} \text{disjI1} (Q): \text{None} \\ \Lambda P Q. \text{iffD2} \cdot \cdot \cdot \cdot \cdot (\text{option.cases-1} \cdot \cdot \cdot \cdot) \end{aligned}$$

$$\begin{aligned} \text{disjI1}: \text{Left} \\ \Lambda P Q. \text{iffD2} \cdot \cdot \cdot \cdot \cdot (\text{sumbool.cases-1} \cdot \cdot \cdot \cdot) \end{aligned}$$

$$\begin{aligned} \text{disjI2} (P, Q): \text{Inr} \\ \Lambda Q P q. \text{iffD2} \cdot \cdot \cdot \cdot \cdot (\text{sum.cases-2} \cdot \cdot \cdot Q \cdot q) \end{aligned}$$

$$\begin{aligned} \text{disjI2} (P): \text{None} \\ \Lambda Q P. \text{iffD2} \cdot \cdot \cdot \cdot \cdot (\text{option.cases-1} \cdot \cdot \cdot \cdot) \end{aligned}$$

$disjI2 \ (Q): \text{Some}$   
 $\Lambda \ Q \ P \ q. \text{iffD2} \cdot \cdot \cdot \cdot (option.cases-2 \cdot \cdot \cdot \cdot Q \cdot q)$

$disjI2: \text{Right}$   
 $\Lambda \ Q \ P. \text{iffD2} \cdot \cdot \cdot \cdot (sumbool.cases-2 \cdot \cdot \cdot \cdot)$

$disjE \ (P, Q, R): \lambda pq \ pr \ qr.$   
 $(case \ pq \ of \ Inl \ p \Rightarrow pr \ p \mid Inr \ q \Rightarrow qr \ q)$   
 $\Lambda \ P \ Q \ R \ pq \ (h1: -) \ pr \ (h2: -) \ qr.$   
 $disjE\text{-realizer} \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \cdot h1 \cdot h2$

$disjE \ (Q, R): \lambda pq \ pr \ qr.$   
 $(case \ pq \ of \ None \Rightarrow pr \mid Some \ q \Rightarrow qr \ q)$   
 $\Lambda \ P \ Q \ R \ pq \ (h1: -) \ pr \ (h2: -) \ qr.$   
 $disjE\text{-realizer2} \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \cdot h1 \cdot h2$

$disjE \ (P, R): \lambda pq \ pr \ qr.$   
 $(case \ pq \ of \ None \Rightarrow qr \mid Some \ p \Rightarrow pr \ p)$   
 $\Lambda \ P \ Q \ R \ pq \ (h1: -) \ pr \ (h2: -) \ qr \ (h3: -).$   
 $disjE\text{-realizer2} \cdot \cdot \cdot \cdot pq \cdot R \cdot qr \cdot pr \cdot h1 \cdot h3 \cdot h2$

$disjE \ (R): \lambda pq \ pr \ qr.$   
 $(case \ pq \ of \ Left \Rightarrow pr \mid Right \Rightarrow qr)$   
 $\Lambda \ P \ Q \ R \ pq \ (h1: -) \ pr \ (h2: -) \ qr.$   
 $disjE\text{-realizer3} \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \cdot h1 \cdot h2$

$disjE \ (P, Q): \text{Null}$   
 $\Lambda \ P \ Q \ R \ pq. \text{disjE-realizer} \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot$

$disjE \ (Q): \text{Null}$   
 $\Lambda \ P \ Q \ R \ pq. \text{disjE-realizer2} \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot$

$disjE \ (P): \text{Null}$   
 $\Lambda \ P \ Q \ R \ pq \ (h1: -) \ (h2: -) \ (h3: -).$   
 $disjE\text{-realizer2} \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot h1 \cdot h3 \cdot h2$

$disjE: \text{Null}$   
 $\Lambda \ P \ Q \ R \ pq. \text{disjE-realizer3} \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot$

$FalseE \ (P): \text{default}$   
 $\Lambda \ P. \text{FalseE} \cdot \cdot$

$FalseE: \text{Null FalseE}$

$notI \ (P): \text{Null}$   
 $\Lambda \ P \ (h: -). \text{allI} \cdot \cdot \cdot (\Lambda \ x. \text{notI} \cdot \cdot \cdot (h \cdot x))$

$notI: \text{Null notI}$

$notE (P, R): \lambda p. default$   
 $\Lambda P R (h: -) p. notE \cdot - \cdot - \cdot (spec \cdot - \cdot p \cdot h)$

$notE (P): Null$   
 $\Lambda P R (h: -) p. notE \cdot - \cdot - \cdot (spec \cdot - \cdot p \cdot h)$

$notE (R): default$   
 $\Lambda P R. notE \cdot - \cdot -$

$notE: Null notE$

$subst (P): \lambda s t ps. ps$   
 $\Lambda s t P (h: -) ps. subst \cdot s \cdot t \cdot P ps \cdot h$

$subst: Null subst$

$iffD1 (P, Q): fst$   
 $\Lambda Q P pq (h: -) p.$   
 $mp \cdot - \cdot - \cdot (spec \cdot - \cdot p \cdot (conjunct1 \cdot - \cdot - \cdot h))$

$iffD1 (P): \lambda p. p$   
 $\Lambda Q P p (h: -). mp \cdot - \cdot - \cdot (conjunct1 \cdot - \cdot - \cdot h)$

$iffD1 (Q): Null$   
 $\Lambda Q P q1 (h: -) q2.$   
 $mp \cdot - \cdot - \cdot (spec \cdot - \cdot q2 \cdot (conjunct1 \cdot - \cdot - \cdot h))$

$iffD1: Null iffD1$

$iffD2 (P, Q): snd$   
 $\Lambda P Q pq (h: -) q.$   
 $mp \cdot - \cdot - \cdot (spec \cdot - \cdot q \cdot (conjunct2 \cdot - \cdot - \cdot h))$

$iffD2 (P): \lambda p. p$   
 $\Lambda P Q p (h: -). mp \cdot - \cdot - \cdot (conjunct2 \cdot - \cdot - \cdot h)$

$iffD2 (Q): Null$   
 $\Lambda P Q q1 (h: -) q2.$   
 $mp \cdot - \cdot - \cdot (spec \cdot - \cdot q2 \cdot (conjunct2 \cdot - \cdot - \cdot h))$

$iffD2: Null iffD2$

$iffI (P, Q): Pair$   
 $\Lambda P Q pq (h1 : -) qp (h2 : -). conjI-realizer \cdot$   
 $(\lambda pq. \forall x. P x \longrightarrow Q (pq x)) \cdot pq \cdot$   
 $(\lambda qp. \forall x. Q x \longrightarrow P (qp x)) \cdot qp \cdot$   
 $(allI \cdot - \cdot (\Lambda x. impI \cdot - \cdot - \cdot (h1 \cdot x))) \cdot$   
 $(allI \cdot - \cdot (\Lambda x. impI \cdot - \cdot - \cdot (h2 \cdot x)))$

```

iffI (P): λp. p
  Λ P Q (h1 : -) p (h2 : -). conjI . . . . .
    (allI . . . (Λ x. impI . . . . . (h1 · x))) .
    (impI . . . . . h2)

iffI (Q): λq. q
  Λ P Q q (h1 : -) (h2 : -). conjI . . . . .
    (impI . . . . . h1) .
    (allI . . . (Λ x. impI . . . . . (h2 · x)))

iffI: Null iffI

setup ⟨⟨
  Sign.add-const-constraint (@{const-name default}, SOME @{typ 'a::default})
⟩⟩

end

```

## 26 Divides: The division operators div and mod

```

theory Divides
imports Nat Power Product-Type
uses ~~/src/Provers/Arith/cancel-div-mod.ML
begin

```

### 26.1 Syntactic division operations

```

class div = dvd +
  fixes div :: 'a ⇒ 'a ⇒ 'a (infixl div 70)
  and mod :: 'a ⇒ 'a ⇒ 'a (infixl mod 70)

```

### 26.2 Abstract division in commutative semirings.

```

class semiring-div = comm-semiring-1-cancel + div +
  assumes mod-div-equality: a div b * b + a mod b = a
  and div-by-0 [simp]: a div 0 = 0
  and div-0 [simp]: 0 div a = 0
  and div-mult-self1 [simp]: b ≠ 0 ⇒ (a + c * b) div b = c + a div b
begin

```

*op div and op mod*

```

lemma mod-div-equality2: b * (a div b) + a mod b = a
  unfolding mult-commute [of b]
  by (rule mod-div-equality)

```

**lemma** *mod-div-equality'*:  $a \bmod b + a \operatorname{div} b * b = a$   
**using** *mod-div-equality* [of  $a \ b$ ]  
**by** (*simp only: add-ac*)

**lemma** *div-mod-equality*:  $((a \operatorname{div} b) * b + a \bmod b) + c = a + c$   
**by** (*simp add: mod-div-equality*)

**lemma** *div-mod-equality2*:  $(b * (a \operatorname{div} b) + a \bmod b) + c = a + c$   
**by** (*simp add: mod-div-equality2*)

**lemma** *mod-by-0* [*simp*]:  $a \bmod 0 = a$   
**using** *mod-div-equality* [of  $a \ 0$ ] **by** *simp*

**lemma** *mod-0* [*simp*]:  $0 \bmod a = 0$   
**using** *mod-div-equality* [of  $0 \ a$ ] *div-0* **by** *simp*

**lemma** *div-mult-self2* [*simp*]:  
**assumes**  $b \neq 0$   
**shows**  $(a + b * c) \operatorname{div} b = c + a \operatorname{div} b$   
**using** *assms div-mult-self1* [of  $b \ a \ c$ ] **by** (*simp add: mult-commute*)

**lemma** *mod-mult-self1* [*simp*]:  $(a + c * b) \bmod b = a \bmod b$   
**proof** (*cases b = 0*)  
**case** *True* **then show** *?thesis* **by** *simp*  
**next**  
**case** *False*  
**have**  $a + c * b = (a + c * b) \operatorname{div} b * b + (a + c * b) \bmod b$   
**by** (*simp add: mod-div-equality*)  
**also from** *False div-mult-self1* [of  $b \ a \ c$ ] **have**  
 $\dots = (c + a \operatorname{div} b) * b + (a + c * b) \bmod b$   
**by** (*simp add: algebra-simps*)  
**finally have**  $a = a \operatorname{div} b * b + (a + c * b) \bmod b$   
**by** (*simp add: add-commute* [of  $a$ ] *add-assoc left-distrib*)  
**then have**  $a \operatorname{div} b * b + (a + c * b) \bmod b = a \operatorname{div} b * b + a \bmod b$   
**by** (*simp add: mod-div-equality*)  
**then show** *?thesis* **by** *simp*  
**qed**

**lemma** *mod-mult-self2* [*simp*]:  $(a + b * c) \bmod b = a \bmod b$   
**by** (*simp add: mult-commute* [of  $b$ ])

**lemma** *div-mult-self1-is-id* [*simp*]:  $b \neq 0 \implies b * a \operatorname{div} b = a$   
**using** *div-mult-self2* [of  $b \ 0 \ a$ ] **by** *simp*

**lemma** *div-mult-self2-is-id* [*simp*]:  $b \neq 0 \implies a * b \operatorname{div} b = a$   
**using** *div-mult-self1* [of  $b \ 0 \ a$ ] **by** *simp*

**lemma** *mod-mult-self1-is-0* [*simp*]:  $b * a \bmod b = 0$   
**using** *mod-mult-self2* [of  $0 \ b \ a$ ] **by** *simp*

**lemma** *mod-mult-self2-is-0* [*simp*]:  $a * b \bmod b = 0$   
**using** *mod-mult-self1* [*of 0 a b*] **by** *simp*

**lemma** *div-by-1* [*simp*]:  $a \operatorname{div} 1 = a$   
**using** *div-mult-self2-is-id* [*of 1 a*] *zero-neq-one* **by** *simp*

**lemma** *mod-by-1* [*simp*]:  $a \bmod 1 = 0$   
**proof** –  
**from** *mod-div-equality* [*of a one*] *div-by-1* **have**  $a + a \bmod 1 = a$  **by** *simp*  
**then have**  $a + a \bmod 1 = a + 0$  **by** *simp*  
**then show** *?thesis* **by** (*rule add-left-imp-eq*)  
**qed**

**lemma** *mod-self* [*simp*]:  $a \bmod a = 0$   
**using** *mod-mult-self2-is-0* [*of 1*] **by** *simp*

**lemma** *div-self* [*simp*]:  $a \neq 0 \implies a \operatorname{div} a = 1$   
**using** *div-mult-self2-is-id* [*of - 1*] **by** *simp*

**lemma** *div-add-self1* [*simp*]:  
**assumes**  $b \neq 0$   
**shows**  $(b + a) \operatorname{div} b = a \operatorname{div} b + 1$   
**using** *assms div-mult-self1* [*of b a 1*] **by** (*simp add: add-commute*)

**lemma** *div-add-self2* [*simp*]:  
**assumes**  $b \neq 0$   
**shows**  $(a + b) \operatorname{div} b = a \operatorname{div} b + 1$   
**using** *assms div-add-self1* [*of b a*] **by** (*simp add: add-commute*)

**lemma** *mod-add-self1* [*simp*]:  
 $(b + a) \bmod b = a \bmod b$   
**using** *mod-mult-self1* [*of a 1 b*] **by** (*simp add: add-commute*)

**lemma** *mod-add-self2* [*simp*]:  
 $(a + b) \bmod b = a \bmod b$   
**using** *mod-mult-self1* [*of a 1 b*] **by** *simp*

**lemma** *mod-div-decomp*:  
**fixes**  $a b$   
**obtains**  $q r$  **where**  $q = a \operatorname{div} b$  **and**  $r = a \bmod b$   
**and**  $a = q * b + r$   
**proof** –  
**from** *mod-div-equality* **have**  $a = a \operatorname{div} b * b + a \bmod b$  **by** *simp*  
**moreover have**  $a \operatorname{div} b = a \operatorname{div} b$  **..**  
**moreover have**  $a \bmod b = a \bmod b$  **..**  
**note that ultimately show** *thesis* **by** *blast*  
**qed**

**lemma** *dvd-eq-mod-eq-0* [code unfold]:  $a \text{ dvd } b \iff b \text{ mod } a = 0$

**proof**

**assume**  $b \text{ mod } a = 0$

**with** *mod-div-equality* [of  $b \ a$ ] **have**  $b \text{ div } a * a = b$  **by** *simp*

**then have**  $b = a * (b \text{ div } a)$  **unfolding** *mult-commute* ..

**then have**  $\exists c. b = a * c$  ..

**then show**  $a \text{ dvd } b$  **unfolding** *dvd-def* .

**next**

**assume**  $a \text{ dvd } b$

**then have**  $\exists c. b = a * c$  **unfolding** *dvd-def* .

**then obtain**  $c$  **where**  $b = a * c$  ..

**then have**  $b \text{ mod } a = a * c \text{ mod } a$  **by** *simp*

**then have**  $b \text{ mod } a = c * a \text{ mod } a$  **by** (*simp add: mult-commute*)

**then show**  $b \text{ mod } a = 0$  **by** *simp*

**qed**

**lemma** *mod-div-trivial* [*simp*]:  $a \text{ mod } b \text{ div } b = 0$

**proof** (*cases*  $b = 0$ )

**assume**  $b = 0$

**thus** *?thesis* **by** *simp*

**next**

**assume**  $b \neq 0$

**hence**  $a \text{ div } b + a \text{ mod } b \text{ div } b = (a \text{ mod } b + a \text{ div } b * b) \text{ div } b$

**by** (*rule div-mult-self1* [*symmetric*])

**also have**  $\dots = a \text{ div } b$

**by** (*simp only: mod-div-equality'*)

**also have**  $\dots = a \text{ div } b + 0$

**by** *simp*

**finally show** *?thesis*

**by** (*rule add-left-imp-eq*)

**qed**

**lemma** *mod-mod-trivial* [*simp*]:  $a \text{ mod } b \text{ mod } b = a \text{ mod } b$

**proof** –

**have**  $a \text{ mod } b \text{ mod } b = (a \text{ mod } b + a \text{ div } b * b) \text{ mod } b$

**by** (*simp only: mod-mult-self1*)

**also have**  $\dots = a \text{ mod } b$

**by** (*simp only: mod-div-equality'*)

**finally show** *?thesis* .

**qed**

**lemma** *dvd-imp-mod-0*:  $a \text{ dvd } b \implies b \text{ mod } a = 0$

**by** (*rule dvd-eq-mod-eq-0* [*THEN iffD1*])

**lemma** *dvd-div-mult-self*:  $a \text{ dvd } b \implies (b \text{ div } a) * a = b$

**by** (*subst* (2) *mod-div-equality* [of  $b \ a$ , *symmetric*]) (*simp add: dvd-imp-mod-0*)

**lemma** *dvd-div-mult*:  $a \text{ dvd } b \implies (b \text{ div } a) * c = b * c \text{ div } a$

**apply** (*cases*  $a = 0$ )



```

apply simp
apply (auto simp: dvd-def mult-assoc)
done

```

```

lemma div-dvd-div[simp]:
   $a \text{ dvd } b \implies a \text{ dvd } c \implies (b \text{ div } a \text{ dvd } c \text{ div } a) = (b \text{ dvd } c)$ 
apply (cases a = 0)
apply simp
apply (unfold dvd-def)
apply auto
apply (blast intro:mult-assoc[symmetric])
apply (fastsimp simp add: mult-assoc)
done

```

```

lemma dvd-mod-imp-dvd:  $[[k \text{ dvd } m \text{ mod } n; k \text{ dvd } n]] \implies k \text{ dvd } m$ 
apply (subgoal-tac k dvd (m div n) * n + m mod n)
apply (simp add: mod-div-equality)
apply (simp only: dvd-add dvd-mult)
done

```

Addition respects modular equivalence.

```

lemma mod-add-left-eq:  $(a + b) \text{ mod } c = (a \text{ mod } c + b) \text{ mod } c$ 
proof –
  have  $(a + b) \text{ mod } c = (a \text{ div } c * c + a \text{ mod } c + b) \text{ mod } c$ 
    by (simp only: mod-div-equality)
  also have  $\dots = (a \text{ mod } c + b + a \text{ div } c * c) \text{ mod } c$ 
    by (simp only: add-ac)
  also have  $\dots = (a \text{ mod } c + b) \text{ mod } c$ 
    by (rule mod-mult-self1)
  finally show ?thesis .
qed

```

```

lemma mod-add-right-eq:  $(a + b) \text{ mod } c = (a + b \text{ mod } c) \text{ mod } c$ 
proof –
  have  $(a + b) \text{ mod } c = (a + (b \text{ div } c * c + b \text{ mod } c)) \text{ mod } c$ 
    by (simp only: mod-div-equality)
  also have  $\dots = (a + b \text{ mod } c + b \text{ div } c * c) \text{ mod } c$ 
    by (simp only: add-ac)
  also have  $\dots = (a + b \text{ mod } c) \text{ mod } c$ 
    by (rule mod-mult-self1)
  finally show ?thesis .
qed

```

```

lemma mod-add-eq:  $(a + b) \text{ mod } c = (a \text{ mod } c + b \text{ mod } c) \text{ mod } c$ 
by (rule trans [OF mod-add-left-eq mod-add-right-eq])

```

```

lemma mod-add-cong:
  assumes  $a \text{ mod } c = a' \text{ mod } c$ 
  assumes  $b \text{ mod } c = b' \text{ mod } c$ 

```

```

shows (a + b) mod c = (a' + b') mod c
proof -
  have (a mod c + b mod c) mod c = (a' mod c + b' mod c) mod c
    unfolding assms ..
  thus ?thesis
    by (simp only: mod-add-eq [symmetric])
qed

```

```

lemma div-add[simp]: z dvd x  $\implies$  z dvd y
 $\implies$  (x + y) div z = x div z + y div z
by(cases z=0, simp, unfold dvd-def, auto simp add: algebra-simps)

```

Multiplication respects modular equivalence.

```

lemma mod-mult-left-eq: (a * b) mod c = ((a mod c) * b) mod c
proof -
  have (a * b) mod c = ((a div c * c + a mod c) * b) mod c
    by (simp only: mod-div-equality)
  also have ... = (a mod c * b + a div c * b * c) mod c
    by (simp only: algebra-simps)
  also have ... = (a mod c * b) mod c
    by (rule mod-mult-self1)
  finally show ?thesis .
qed

```

```

lemma mod-mult-right-eq: (a * b) mod c = (a * (b mod c)) mod c
proof -
  have (a * b) mod c = (a * (b div c * c + b mod c)) mod c
    by (simp only: mod-div-equality)
  also have ... = (a * (b mod c) + a * (b div c) * c) mod c
    by (simp only: algebra-simps)
  also have ... = (a * (b mod c)) mod c
    by (rule mod-mult-self1)
  finally show ?thesis .
qed

```

```

lemma mod-mult-eq: (a * b) mod c = ((a mod c) * (b mod c)) mod c
by (rule trans [OF mod-mult-left-eq mod-mult-right-eq])

```

```

lemma mod-mult-cong:
  assumes a mod c = a' mod c
  assumes b mod c = b' mod c
  shows (a * b) mod c = (a' * b') mod c
proof -
  have (a mod c * (b mod c)) mod c = (a' mod c * (b' mod c)) mod c
    unfolding assms ..
  thus ?thesis
    by (simp only: mod-mult-eq [symmetric])
qed

```

```

lemma mod-mod-cancel:
  assumes  $c \text{ dvd } b$ 
  shows  $a \text{ mod } b \text{ mod } c = a \text{ mod } c$ 
proof –
  from  $\langle c \text{ dvd } b \rangle$  obtain  $k$  where  $b = c * k$ 
  by (rule dvdE)
  have  $a \text{ mod } b \text{ mod } c = a \text{ mod } (c * k) \text{ mod } c$ 
  by (simp only:  $\langle b = c * k \rangle$ )
  also have  $\dots = (a \text{ mod } (c * k) + a \text{ div } (c * k) * k * c) \text{ mod } c$ 
  by (simp only: mod-mult-self1)
  also have  $\dots = (a \text{ div } (c * k) * (c * k) + a \text{ mod } (c * k)) \text{ mod } c$ 
  by (simp only: add-ac mult-ac)
  also have  $\dots = a \text{ mod } c$ 
  by (simp only: mod-div-equality)
  finally show ?thesis .
qed

end

lemma div-mult-div-if-dvd:  $(y::'a::\{\text{semiring-div}, \text{no-zero-divisors}\}) \text{ dvd } x \implies$ 
 $z \text{ dvd } w \implies (x \text{ div } y) * (w \text{ div } z) = (x * w) \text{ div } (y * z)$ 
unfolding dvd-def
apply clarify
apply (case-tac  $y = 0$ )
apply simp
apply (case-tac  $z = 0$ )
apply simp
apply (simp add: algebra-simps)
apply (subst mult-assoc [symmetric])
apply (simp add: no-zero-divisors)
done

lemma div-power:  $(y::'a::\{\text{semiring-div}, \text{no-zero-divisors}, \text{recpower}\}) \text{ dvd } x \implies$ 
 $(x \text{ div } y)^n = x^n \text{ div } y^n$ 
apply (induct n)
apply simp
apply (simp add: div-mult-div-if-dvd dvd-power-same)
done

class ring-div = semiring-div + comm-ring-1
begin

```

Negation respects modular equivalence.

```

lemma mod-minus-eq:  $(- a) \text{ mod } b = (- (a \text{ mod } b)) \text{ mod } b$ 
proof –
  have  $(- a) \text{ mod } b = (- (a \text{ div } b * b + a \text{ mod } b)) \text{ mod } b$ 
  by (simp only: mod-div-equality)
  also have  $\dots = (- (a \text{ mod } b) + - (a \text{ div } b) * b) \text{ mod } b$ 

```

```

    by (simp only: minus-add-distrib minus-mult-left add-ac)
  also have  $\dots = (- (a \bmod b)) \bmod b$ 
    by (rule mod-mult-self1)
  finally show ?thesis .
qed

```

```

lemma mod-minus-cong:
  assumes  $a \bmod b = a' \bmod b$ 
  shows  $(- a) \bmod b = (- a') \bmod b$ 
proof -
  have  $(- (a \bmod b)) \bmod b = (- (a' \bmod b)) \bmod b$ 
    unfolding assms ..
  thus ?thesis
    by (simp only: mod-minus-eq [symmetric])
qed

```

Subtraction respects modular equivalence.

```

lemma mod-diff-left-eq:  $(a - b) \bmod c = (a \bmod c - b) \bmod c$ 
  unfolding diff-minus
  by (intro mod-add-cong mod-minus-cong) simp-all

```

```

lemma mod-diff-right-eq:  $(a - b) \bmod c = (a - b \bmod c) \bmod c$ 
  unfolding diff-minus
  by (intro mod-add-cong mod-minus-cong) simp-all

```

```

lemma mod-diff-eq:  $(a - b) \bmod c = (a \bmod c - b \bmod c) \bmod c$ 
  unfolding diff-minus
  by (intro mod-add-cong mod-minus-cong) simp-all

```

```

lemma mod-diff-cong:
  assumes  $a \bmod c = a' \bmod c$ 
  assumes  $b \bmod c = b' \bmod c$ 
  shows  $(a - b) \bmod c = (a' - b') \bmod c$ 
  unfolding diff-minus using assms
  by (intro mod-add-cong mod-minus-cong)

```

```

lemma dvd-neg-div:  $y \text{ dvd } x \implies -x \text{ div } y = - (x \text{ div } y)$ 
apply (case-tac  $y = 0$ ) apply simp
apply (auto simp add: dvd-def)
apply (subgoal-tac  $-(y * k) = y * -k$ )
  apply (erule ssubst)
  apply (erule div-mult-self1-is-id)
apply simp
done

```

```

lemma dvd-div-neg:  $y \text{ dvd } x \implies x \text{ div } -y = - (x \text{ div } y)$ 
apply (case-tac  $y = 0$ ) apply simp
apply (auto simp add: dvd-def)
apply (subgoal-tac  $y * k = -y * -k$ )

```

```

apply (erule ssubst)
apply (rule div-mult-self1-is-id)
apply simp
apply simp
done

end

```

### 26.3 Division on *nat*

We define *op div* and *op mod* on *nat* by means of a characteristic relation with two input arguments *m*, *n* and two output arguments *q*(uotient) and *r*(emainder).

**definition** *divmod-rel* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *bool* **where**  
*divmod-rel m n q r*  $\longleftrightarrow m = q * n + r \wedge (\text{if } n > 0 \text{ then } 0 \leq r \wedge r < n \text{ else } q = 0)$

*divmod-rel* is total:

**lemma** *divmod-rel-ex*:

```

obtains q r where divmod-rel m n q r
proof (cases n = 0)
  case True with that show thesis
    by (auto simp add: divmod-rel-def)
next
  case False
  have  $\exists q r. m = q * n + r \wedge r < n$ 
  proof (induct m)
    case 0 with  $\langle n \neq 0 \rangle$ 
    have  $(0::nat) = 0 * n + 0 \wedge 0 < n$  by simp
    then show ?case by blast
  next
    case (Suc m) then obtain q' r'
      where m:  $m = q' * n + r'$  and n:  $r' < n$  by auto
    then show ?case proof (cases Suc r' < n)
      case True
      from m n have Suc m =  $q' * n + \text{Suc } r'$  by simp
      with True show ?thesis by blast
    next
      case False then have  $n \leq \text{Suc } r'$  by auto
      moreover from n have  $\text{Suc } r' \leq n$  by auto
      ultimately have  $n = \text{Suc } r'$  by auto
      with m have Suc m =  $\text{Suc } q' * n + 0$  by simp
      with  $\langle n \neq 0 \rangle$  show ?thesis by blast
    qed
  qed
with that show thesis
  using  $\langle n \neq 0 \rangle$  by (auto simp add: divmod-rel-def)
qed

```

*divmod-rel* is injective:

```

lemma divmod-rel-unique-div:
  assumes divmod-rel m n q r
    and divmod-rel m n q' r'
  shows  $q = q'$ 
proof (cases  $n = 0$ )
  case True with assms show ?thesis
    by (simp add: divmod-rel-def)
next
  case False
  have aux:  $\bigwedge q\ r\ q'\ r'.\ q' * n + r' = q * n + r \implies r < n \implies q' \leq (q::nat)$ 
  apply (rule leI)
  apply (subst less-iff-Suc-add)
  apply (auto simp add: add-mult-distrib)
  done
  from  $\langle n \neq 0 \rangle$  assms show ?thesis
    by (auto simp add: divmod-rel-def
      intro: order-antisym dest: aux sym)
qed

```

```

lemma divmod-rel-unique-mod:
  assumes divmod-rel m n q r
    and divmod-rel m n q' r'
  shows  $r = r'$ 
proof –
  from assms have  $q = q'$  by (rule divmod-rel-unique-div)
  with assms show ?thesis by (simp add: divmod-rel-def)
qed

```

We instantiate divisibility on the natural numbers by means of *divmod-rel*:

```

instantiation nat :: semiring-div
begin

```

```

definition divmod :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\times$  nat where
  [code del]: divmod m n = (THE (q, r). divmod-rel m n q r)

```

```

definition div-nat where
  m div n = fst (divmod m n)

```

```

definition mod-nat where
  m mod n = snd (divmod m n)

```

```

lemma divmod-div-mod:
  divmod m n = (m div n, m mod n)
  unfolding div-nat-def mod-nat-def by simp

```

```

lemma divmod-eq:
  assumes divmod-rel m n q r
  shows divmod m n = (q, r)

```

**using** *assms* **by** (*auto simp add: divmod-def*  
*dest: divmod-rel-unique-div divmod-rel-unique-mod*)

**lemma** *div-eq*:  
**assumes** *divmod-rel m n q r*  
**shows**  $m \text{ div } n = q$   
**using** *assms* **by** (*auto dest: divmod-eq simp add: div-nat-def*)

**lemma** *mod-eq*:  
**assumes** *divmod-rel m n q r*  
**shows**  $m \text{ mod } n = r$   
**using** *assms* **by** (*auto dest: divmod-eq simp add: mod-nat-def*)

**lemma** *divmod-rel: divmod-rel m n (m div n) (m mod n)*  
**proof** –  
**from** *divmod-rel-ex*  
**obtain**  $q\ r$  **where** *rel: divmod-rel m n q r* .  
**moreover with** *div-eq mod-eq* **have**  $m \text{ div } n = q$  **and**  $m \text{ mod } n = r$   
**by** *simp-all*  
**ultimately show** *?thesis* **by** *simp*  
**qed**

**lemma** *divmod-zero*:  
 $\text{divmod } m\ 0 = (0, m)$   
**proof** –  
**from** *divmod-rel [of m 0]* **show** *?thesis*  
**unfolding** *divmod-div-mod divmod-rel-def* **by** *simp*  
**qed**

**lemma** *divmod-base*:  
**assumes**  $m < n$   
**shows**  $\text{divmod } m\ n = (0, m)$   
**proof** –  
**from** *divmod-rel [of m n]* **show** *?thesis*  
**unfolding** *divmod-div-mod divmod-rel-def*  
**using** *assms* **by** (*cases m div n = 0*)  
*(auto simp add: gr0-conv-Suc [of m div n])*  
**qed**

**lemma** *divmod-step*:  
**assumes**  $0 < n$  **and**  $n \leq m$   
**shows**  $\text{divmod } m\ n = (\text{Suc } ((m - n) \text{ div } n), (m - n) \text{ mod } n)$   
**proof** –  
**from** *divmod-rel* **have** *divmod-m-n: divmod-rel m n (m div n) (m mod n)* .  
**with** *assms* **have** *m-div-n: m div n  $\geq$  1*  
**by** (*cases m div n*) (*auto simp add: divmod-rel-def*)  
**from** *assms divmod-m-n* **have** *divmod-rel (m - n) n (m div n - Suc 0) (m mod n)*  
**by** (*cases m div n*) (*auto simp add: divmod-rel-def*)

```

with divmod-eq have  $\text{divmod } (m - n) \ n = (m \text{ div } n - \text{Suc } 0, m \bmod n)$  by
simp
moreover from divmod-div-mod have  $\text{divmod } (m - n) \ n = ((m - n) \text{ div } n,$ 
 $(m - n) \bmod n)$  .
ultimately have  $m \text{ div } n = \text{Suc } ((m - n) \text{ div } n)$ 
and  $m \bmod n = (m - n) \bmod n$  using m-div-n by simp-all
then show ?thesis using divmod-div-mod by simp
qed

```

The “recursion” equations for *op div* and *op mod*

```

lemma div-less [simp]:
  fixes  $m \ n :: \text{nat}$ 
  assumes  $m < n$ 
  shows  $m \text{ div } n = 0$ 
  using assms divmod-base divmod-div-mod by simp

```

```

lemma le-div-geq:
  fixes  $m \ n :: \text{nat}$ 
  assumes  $0 < n$  and  $n \leq m$ 
  shows  $m \text{ div } n = \text{Suc } ((m - n) \text{ div } n)$ 
  using assms divmod-step divmod-div-mod by simp

```

```

lemma mod-less [simp]:
  fixes  $m \ n :: \text{nat}$ 
  assumes  $m < n$ 
  shows  $m \bmod n = m$ 
  using assms divmod-base divmod-div-mod by simp

```

```

lemma le-mod-geq:
  fixes  $m \ n :: \text{nat}$ 
  assumes  $n \leq m$ 
  shows  $m \bmod n = (m - n) \bmod n$ 
  using assms divmod-step divmod-div-mod by (cases  $n = 0$ ) simp-all

```

**instance proof**

```

  fix  $m \ n :: \text{nat}$  show  $m \text{ div } n * n + m \bmod n = m$ 
    using divmod-rel [of  $m \ n$ ] by (simp add: divmod-rel-def)
next
  fix  $n :: \text{nat}$  show  $n \text{ div } 0 = 0$ 
    using divmod-zero divmod-div-mod [of  $n \ 0$ ] by simp
next
  fix  $n :: \text{nat}$  show  $0 \text{ div } n = 0$ 
    using divmod-rel [of  $0 \ n$ ] by (cases  $n$ ) (simp-all add: divmod-rel-def)
next
  fix  $m \ n \ q :: \text{nat}$  assume  $n \neq 0$  then show  $(q + m * n) \text{ div } n = m + q \text{ div } n$ 
    by (induct  $m$ ) (simp-all add: le-div-geq)
qed

```

**end**



Simproc for cancelling *op div* and *op mod*

```

ML <<
  structure CancelDivModData =
  struct

    val div-name = @{const-name div};
    val mod-name = @{const-name mod};
    val mk-binop = HOLogic.mk-binop;
    val mk-sum = Nat-Arith.mk-sum;
    val dest-sum = Nat-Arith.dest-sum;

    (*logic*)

    val div-mod-egs = map mk-meta-eq [@{thm div-mod-equality}, @{thm div-mod-equality2}]

    val trans = trans

    val prove-eq-sums =
      let val_simps = @{thm add-0} :: @{thm add-0-right} :: @{thms add-ac}
      in Arith-Data.prove-conv2 all-tac (Arith-Data.simp-all-tac_simps) end;

  end;

  structure CancelDivMod = CancelDivModFun(CancelDivModData);

  val cancel-div-mod-proc = Simplifier.simproc (the-context ())
    cancel-div-mod [(m::nat) + n] (K CancelDivMod.proc);

  Addsimprocs[cancel-div-mod-proc];
>>

```

code generator setup

```

lemma divmod-if [code]: divmod m n = (if n = 0 ∨ m < n then (0, m) else
  let (q, r) = divmod (m - n) n in (Suc q, r))
by (simp add: divmod-zero divmod-base divmod-step)
    (simp add: divmod-div-mod)

```

**code-modulename** *SML*

*Divides Nat*

**code-modulename** *OCaml*

*Divides Nat*

**code-modulename** *Haskell*

*Divides Nat*

### 26.3.1 Quotient

**lemma** div-geq:  $0 < n \implies \neg m < n \implies m \text{ div } n = \text{Suc } ((m - n) \text{ div } n)$

**by** (*simp add: le-div-geq linorder-not-less*)

**lemma** *div-if*:  $0 < n \implies m \text{ div } n = (\text{if } m < n \text{ then } 0 \text{ else } \text{Suc } ((m - n) \text{ div } n))$   
**by** (*simp add: div-geq*)

**lemma** *div-mult-self-is-m* [*simp*]:  $0 < n \implies (m * n) \text{ div } n = (m :: \text{nat})$   
**by** *simp*

**lemma** *div-mult-self1-is-m* [*simp*]:  $0 < n \implies (n * m) \text{ div } n = (m :: \text{nat})$   
**by** *simp*

### 26.3.2 Remainder

**lemma** *mod-less-divisor* [*simp*]:  
 fixes  $m \ n :: \text{nat}$   
 assumes  $n > 0$   
 shows  $m \text{ mod } n < (n :: \text{nat})$   
 using *assms divmod-rel unfolding divmod-rel-def* **by** *auto*

**lemma** *mod-less-eq-dividend* [*simp*]:  
 fixes  $m \ n :: \text{nat}$   
 shows  $m \text{ mod } n \leq m$   
**proof** (*rule add-leD2*)  
 from *mod-div-equality* **have**  $m \text{ div } n * n + m \text{ mod } n = m$  .  
 then **show**  $m \text{ div } n * n + m \text{ mod } n \leq m$  **by** *auto*  
**qed**

**lemma** *mod-geq*:  $\neg m < (n :: \text{nat}) \implies m \text{ mod } n = (m - n) \text{ mod } n$   
**by** (*simp add: le-mod-geq linorder-not-less*)

**lemma** *mod-if*:  $m \text{ mod } (n :: \text{nat}) = (\text{if } m < n \text{ then } m \text{ else } (m - n) \text{ mod } n)$   
**by** (*simp add: le-mod-geq*)

**lemma** *mod-1* [*simp*]:  $m \text{ mod } \text{Suc } 0 = 0$   
**by** (*induct m*) (*simp-all add: mod-geq*)

**lemma** *mod-mult-distrib*:  $(m \text{ mod } n) * (k :: \text{nat}) = (m * k) \text{ mod } (n * k)$   
**apply** (*cases n = 0, simp*)  
**apply** (*cases k = 0, simp*)  
**apply** (*induct m rule: nat-less-induct*)  
**apply** (*subst mod-if, simp*)  
**apply** (*simp add: mod-geq diff-mult-distrib*)  
**done**

**lemma** *mod-mult-distrib2*:  $(k :: \text{nat}) * (m \text{ mod } n) = (k * m) \text{ mod } (k * n)$   
**by** (*simp add: mult-commute [of k] mod-mult-distrib*)

**lemma** *mult-div-cancel*:  $(n :: \text{nat}) * (m \text{ div } n) = m - (m \text{ mod } n)$

by (cut-tac  $a = m$  and  $b = n$  in mod-div-equality2, arith)

lemma mod-le-divisor[simp]:  $0 < n \implies m \bmod n \leq (n::nat)$   
 apply (drule mod-less-divisor [where  $m = m$ ])  
 apply simp  
 done

### 26.3.3 Quotient and Remainder

lemma divmod-rel-mult1-eq:  
 [| divmod-rel  $b \ c \ q \ r$ ;  $c > 0$  |]  
 $\implies$  divmod-rel  $(a*b) \ c \ (a*q + a*r \text{ div } c) \ (a*r \bmod c)$   
 by (auto simp add: split-ifs divmod-rel-def algebra-simps)

lemma div-mult1-eq:  $(a*b) \text{ div } c = a*(b \text{ div } c) + a*(b \bmod c) \text{ div } (c::nat)$   
 apply (cases  $c = 0$ , simp)  
 apply (blast intro: divmod-rel [THEN divmod-rel-mult1-eq, THEN div-eq])  
 done

lemma divmod-rel-add1-eq:  
 [| divmod-rel  $a \ c \ aq \ ar$ ; divmod-rel  $b \ c \ bq \ br$ ;  $c > 0$  |]  
 $\implies$  divmod-rel  $(a + b) \ c \ (aq + bq + (ar+br) \text{ div } c) \ ((ar + br) \bmod c)$   
 by (auto simp add: split-ifs divmod-rel-def algebra-simps)

lemma div-add1-eq:  
 $(a+b) \text{ div } (c::nat) = a \text{ div } c + b \text{ div } c + ((a \bmod c + b \bmod c) \text{ div } c)$   
 apply (cases  $c = 0$ , simp)  
 apply (blast intro: divmod-rel-add1-eq [THEN div-eq] divmod-rel)  
 done

lemma mod-lemma: [|  $(0::nat) < c$ ;  $r < b$  |]  $\implies b * (q \bmod c) + r < b * c$   
 apply (cut-tac  $m = q$  and  $n = c$  in mod-less-divisor)  
 apply (drule-tac [2]  $m = q \bmod c$  in less-imp-Suc-add, auto)  
 apply (erule-tac  $P = \%x. ?lhs < ?rhs \ x$  in ssubst)  
 apply (simp add: add-mult-distrib2)  
 done

lemma divmod-rel-mult2-eq: [| divmod-rel  $a \ b \ q \ r$ ;  $0 < b$ ;  $0 < c$  |]  
 $\implies$  divmod-rel  $a \ (b*c) \ (q \text{ div } c) \ (b*(q \bmod c) + r)$   
 by (auto simp add: mult-ac divmod-rel-def add-mult-distrib2 [symmetric] mod-lemma)

lemma div-mult2-eq:  $a \text{ div } (b*c) = (a \text{ div } b) \text{ div } (c::nat)$   
 apply (cases  $b = 0$ , simp)  
 apply (cases  $c = 0$ , simp)  
 apply (force simp add: divmod-rel [THEN divmod-rel-mult2-eq, THEN div-eq])  
 done

lemma mod-mult2-eq:  $a \bmod (b*c) = b*(a \text{ div } b \bmod c) + a \bmod (b::nat)$

```

apply (cases b = 0, simp)
apply (cases c = 0, simp)
apply (auto simp add: mult-commute divmod-rel [THEN divmod-rel-mult2-eq,
THEN mod-eq])
done

```

### 26.3.4 Cancellation of Common Factors in Division

**lemma** *div-mult-mult-lemma*:

```

  [| (0::nat) < b; 0 < c |] ==> (c*a) div (c*b) = a div b
by (auto simp add: div-mult2-eq)

```

**lemma** *div-mult-mult1* [simp]:  $(0::nat) < c \implies (c*a) \text{ div } (c*b) = a \text{ div } b$

```

apply (cases b = 0)
apply (auto simp add: linorder-neq-iff [of b] div-mult-mult-lemma)
done

```

**lemma** *div-mult-mult2* [simp]:  $(0::nat) < c \implies (a*c) \text{ div } (b*c) = a \text{ div } b$

```

apply (drule div-mult-mult1)
apply (auto simp add: mult-commute)
done

```

### 26.3.5 Further Facts about Quotient and Remainder

**lemma** *div-1* [simp]:  $m \text{ div } \text{Suc } 0 = m$

**by** (induct m) (simp-all add: div-geq)

**lemma** *div-le-mono* [rule-format]:

$\forall m::nat. m \leq n \longrightarrow (m \text{ div } k) \leq (n \text{ div } k)$

```

apply (case-tac k=0, simp)
apply (induct n rule: nat-less-induct, clarify)
apply (case-tac n<k)

```

**apply** *simp*

**apply** (case-tac m<k)

**apply** *simp*

```

apply (simp add: div-geq diff-le-mono)
done

```

**lemma** *div-le-mono2*:  $!!m::nat. [| 0 < m; m \leq n |] \implies (k \text{ div } n) \leq (k \text{ div } m)$

```

apply (subgoal-tac 0<n)
prefer 2 apply simp
apply (induct-tac k rule: nat-less-induct)
apply (rename-tac k)

```

```

apply (case-tac  $k < n$ , simp)
apply (subgoal-tac  $\sim (k < m)$  )
  prefer 2 apply simp
apply (simp add: div-geq)
apply (subgoal-tac  $(k - n) \text{ div } n \leq (k - m) \text{ div } n$ )
  prefer 2
  apply (blast intro: div-le-mono diff-le-mono2)
apply (rule le-trans, simp)
apply (simp)
done

```

```

lemma div-le-dividend [simp]:  $m \text{ div } n \leq (m::\text{nat})$ 
apply (case-tac  $n=0$ , simp)
apply (subgoal-tac  $m \text{ div } n \leq m \text{ div } 1$ , simp)
apply (rule div-le-mono2)
apply (simp-all (no-asm-simp))
done

```

```

lemma div-less-dividend [rule-format]:
  !! $n::\text{nat}$ .  $1 < n \implies 0 < m \dashv\vdash m \text{ div } n < m$ 
apply (induct-tac  $m$  rule: nat-less-induct)
apply (rename-tac  $m$ )
apply (case-tac  $m < n$ , simp)
apply (subgoal-tac  $0 < n$ )
  prefer 2 apply simp
apply (simp add: div-geq)
apply (case-tac  $n < m$ )
  apply (subgoal-tac  $(m - n) \text{ div } n < (m - n)$  )
    apply (rule impI less-trans-Suc)+
apply assumption
  apply (simp-all)
done

```

```

lemma nat-div-eq-0 [simp]:  $(n::\text{nat}) > 0 \implies ((m \text{ div } n) = 0) = (m < n)$ 
by(auto, subst mod-div-equality [of  $m$   $n$ , symmetric], auto)

```

```

lemma nat-div-gt-0 [simp]:  $(n::\text{nat}) > 0 \implies ((m \text{ div } n) > 0) = (m \geq n)$ 
by (subst neq0-conv [symmetric], auto)

```

```

declare div-less-dividend [simp]

```

A fact for the mutilated chess board

```

lemma mod-Suc:  $\text{Suc}(m) \text{ mod } n = (\text{if } \text{Suc}(m \text{ mod } n) = n \text{ then } 0 \text{ else } \text{Suc}(m \text{ mod } n))$ 
apply (case-tac  $n=0$ , simp)
apply (induct  $m$  rule: nat-less-induct)
apply (case-tac  $\text{Suc } (na) < n$ )

```

**apply** (*frule lessI* [THEN *less-trans*], *simp add: less-not-refl3*)

**apply** (*simp add: linorder-not-less le-Suc-eq mod-geq*)

**apply** (*auto simp add: Suc-diff-le le-mod-geq*)

**done**

### 26.3.6 The Divides Relation

**lemma** *dvd-1-left* [*iff*]: *Suc 0 dvd k*

**unfolding** *dvd-def* **by** *simp*

**lemma** *dvd-1-iff-1* [*simp*]:  $(m \text{ dvd } \text{Suc } 0) = (m = \text{Suc } 0)$

**by** (*simp add: dvd-def*)

**lemma** *nat-dvd-1-iff-1* [*simp*]:  $m \text{ dvd } (1::\text{nat}) \longleftrightarrow m = 1$

**by** (*simp add: dvd-def*)

**lemma** *dvd-anti-sym*:  $[[m \text{ dvd } n; n \text{ dvd } m]] \implies m = (n::\text{nat})$

**unfolding** *dvd-def*

**by** (*force dest: mult-eq-self-implies-10 simp add: mult-assoc mult-eq-1-iff*)

*op dvd* is a partial order

**interpretation** *dvd*: *order op dvd  $\lambda n m :: \text{nat}. n \text{ dvd } m \wedge \neg m \text{ dvd } n$*

**proof** **qed** (*auto intro: dvd-refl dvd-trans dvd-anti-sym*)

**lemma** *nat-dvd-diff* [*simp*]:  $[[k \text{ dvd } m; k \text{ dvd } n]] \implies k \text{ dvd } (m - n :: \text{nat})$

**unfolding** *dvd-def*

**by** (*blast intro: diff-mult-distrib2 [symmetric]*)

**lemma** *dvd-diffD*:  $[[k \text{ dvd } m - n; k \text{ dvd } n; n \leq m]] \implies k \text{ dvd } (m::\text{nat})$

**apply** (*erule linorder-not-less [THEN iffD2, THEN add-diff-inverse, THEN subst]*)

**apply** (*blast intro: dvd-add*)

**done**

**lemma** *dvd-diffD1*:  $[[k \text{ dvd } m - n; k \text{ dvd } m; n \leq m]] \implies k \text{ dvd } (n::\text{nat})$

**by** (*drule-tac m = m in nat-dvd-diff, auto*)

**lemma** *dvd-reduce*:  $(k \text{ dvd } n + k) = (k \text{ dvd } (n::\text{nat}))$

**apply** (*rule iffI*)

**apply** (*erule-tac [2] dvd-add*)

**apply** (*rule-tac [2] dvd-refl*)

**apply** (*subgoal-tac n = (n+k) - k*)

**prefer 2 apply simp**

**apply** (*erule ssubst*)

**apply** (*erule nat-dvd-diff*)

**apply** (*rule dvd-refl*)

**done**

```

lemma dvd-mod: !!n::nat. [| f dvd m; f dvd n |] ==> f dvd m mod n
  unfolding dvd-def
  apply (case-tac n = 0, auto)
  apply (blast intro: mod-mult-distrib2 [symmetric])
  done

```

```

lemma dvd-mod-iff: k dvd n ==> ((k::nat) dvd m mod n) = (k dvd m)
by (blast intro: dvd-mod-imp-dvd dvd-mod)

```

```

lemma dvd-mult-cancel: !!k::nat. [| k*m dvd k*n; 0<k |] ==> m dvd n
  unfolding dvd-def
  apply (erule exE)
  apply (simp add: mult-ac)
  done

```

```

lemma dvd-mult-cancel1: 0<m ==> (m*n dvd m) = (n = (1::nat))
  apply auto
  apply (subgoal-tac m*n dvd m*1)
  apply (drule dvd-mult-cancel, auto)
  done

```

```

lemma dvd-mult-cancel2: 0<m ==> (n*m dvd m) = (n = (1::nat))
  apply (subst mult-commute)
  apply (erule dvd-mult-cancel1)
  done

```

```

lemma dvd-imp-le: [| k dvd n; 0 < n |] ==> k ≤ (n::nat)
by (auto elim!: dvdE) (auto simp add: gr0-conv-Suc)

```

```

lemma nat-dvd-not-less: (0::nat) < m ==> m < n ==> ¬ n dvd m
by (auto elim!: dvdE) (auto simp add: gr0-conv-Suc)

```

```

lemma dvd-mult-div-cancel: n dvd m ==> n * (m div n) = (m::nat)
  apply (subgoal-tac m mod n = 0)
  apply (simp add: mult-div-cancel)
  apply (simp only: dvd-eq-mod-eq-0)
  done

```

```

lemma nat-zero-less-power-iff [simp]: (x^n > 0) = (x > (0::nat) | n=0)
  by (induct n) auto

```

```

lemma power-dvd-imp-le: [| i^m dvd i^n; (1::nat) < i |] ==> m ≤ n
  apply (rule power-le-imp-le-exp, assumption)
  apply (erule dvd-imp-le, simp)
  done

```

```

lemma mod-eq-0-iff: (m mod d = 0) = (∃ q::nat. m = d*q)
by (auto simp add: dvd-eq-mod-eq-0 [symmetric] dvd-def)

```

**lemmas** *mod-eq-0D* [*dest!*] = *mod-eq-0-iff* [*THEN iffD1*]

**lemma** *mod-eqD*:  $(m \bmod d = r) ==> \exists q::nat. m = r + q*d$   
**apply** (*cut-tac a = m in mod-div-equality*)  
**apply** (*simp only: add-ac*)  
**apply** (*blast intro: sym*)  
**done**

**lemma** *split-div*:

$P(n \text{ div } k :: nat) =$   
 $((k = 0 \longrightarrow P\ 0) \wedge (k \neq 0 \longrightarrow (!i. !j < k. n = k*i + j \longrightarrow P\ i)))$   
 $(\text{is } ?P = ?Q \text{ is } - = (- \wedge (- \longrightarrow ?R)))$

**proof**

**assume**  $P: ?P$

**show**  $?Q$

**proof** (*cases*)

**assume**  $k = 0$

**with**  $P$  **show**  $?Q$  **by** *simp*

**next**

**assume**  $\text{not } 0: k \neq 0$

**thus**  $?Q$

**proof** (*simp, intro allI impI*)

**fix**  $i\ j$

**assume**  $n: n = k*i + j$  **and**  $j: j < k$

**show**  $P\ i$

**proof** (*cases*)

**assume**  $i = 0$

**with**  $n\ j\ P$  **show**  $P\ i$  **by** *simp*

**next**

**assume**  $i \neq 0$

**with**  $\text{not } 0\ n\ j\ P$  **show**  $P\ i$  **by** (*simp add: add-ac*)

**qed**

**qed**

**qed**

**next**

**assume**  $Q: ?Q$

**show**  $?P$

**proof** (*cases*)

**assume**  $k = 0$

**with**  $Q$  **show**  $?P$  **by** *simp*

**next**

**assume**  $\text{not } 0: k \neq 0$

**with**  $Q$  **have**  $R: ?R$  **by** *simp*

**from**  $\text{not } 0\ R$  [*THEN spec, of n div k, THEN spec, of n mod k*]

**show**  $?P$  **by** *simp*

**qed**

**qed**



**lemma** *split-div-lemma*:

**assumes**  $0 < n$

**shows**  $n * q \leq m \wedge m < n * \text{Suc } q \longleftrightarrow q = ((m::\text{nat}) \text{ div } n) \text{ (is } ?lhs \longleftrightarrow ?rhs)$

**proof**

**assume**  $?rhs$

**with** *mult-div-cancel* **have**  $nq: n * q = m - (m \bmod n)$  **by** *simp*

**then have**  $A: n * q \leq m$  **by** *simp*

**have**  $n - (m \bmod n) > 0$  **using** *mod-less-divisor assms* **by** *auto*

**then have**  $m < m + (n - (m \bmod n))$  **by** *simp*

**then have**  $m < n + (m - (m \bmod n))$  **by** *simp*

**with**  $nq$  **have**  $m < n + n * q$  **by** *simp*

**then have**  $B: m < n * \text{Suc } q$  **by** *simp*

**from**  $A B$  **show**  $?lhs$  **..**

**next**

**assume**  $P: ?lhs$

**then have** *divmod-rel*  $m n q (m - n * q)$

**unfolding** *divmod-rel-def* **by** (*auto simp add: mult-ac*)

**then show**  $?rhs$  **using** *divmod-rel* **by** (*rule divmod-rel-unique-div*)

**qed**

**theorem** *split-div'*:

$P ((m::\text{nat}) \text{ div } n) = ((n = 0 \wedge P 0) \vee$

$(\exists q. (n * q \leq m \wedge m < n * (\text{Suc } q)) \wedge P q))$

**apply** (*case-tac*  $0 < n$ )

**apply** (*simp only: add: split-div-lemma*)

**apply** *simp-all*

**done**

**lemma** *split-mod*:

$P(n \bmod k :: \text{nat}) =$

$((k = 0 \longrightarrow P n) \wedge (k \neq 0 \longrightarrow (!i. !j < k. n = k*i + j \longrightarrow P j)))$

$(\text{is } ?P = ?Q \text{ is } - = (- \wedge (- \longrightarrow ?R)))$

**proof**

**assume**  $P: ?P$

**show**  $?Q$

**proof** (*cases*)

**assume**  $k = 0$

**with**  $P$  **show**  $?Q$  **by** *simp*

**next**

**assume** *not0*:  $k \neq 0$

**thus**  $?Q$

**proof** (*simp, intro allI impI*)

**fix**  $i j$

**assume**  $n = k*i + j \ j < k$

**thus**  $P j$  **using** *not0 P* **by** (*simp add: add-ac mult-ac*)

**qed**

**qed**

**next**

**assume**  $Q: ?Q$

```

show ?P
proof (cases)
  assume  $k = 0$ 
  with  $Q$  show ?P by simp
next
  assume  $not0: k \neq 0$ 
  with  $Q$  have  $R: ?R$  by simp
  from  $not0$   $R$  [THEN spec, of  $n \text{ div } k$ , THEN spec, of  $n \text{ mod } k$ ]
  show ?P by simp
qed
qed

```

```

theorem mod-div-equality':  $(m::nat) \text{ mod } n = m - (m \text{ div } n) * n$ 
  apply (rule-tac  $P \Rightarrow \%x. m \text{ mod } n = x - (m \text{ div } n) * n$ ) in
    subst [OF mod-div-equality [of -  $n$ ]]
  apply arith
done

```

```

lemma div-mod-equality':
  fixes  $m n :: nat$ 
  shows  $m \text{ div } n * n = m - m \text{ mod } n$ 
proof -
  have  $m \text{ mod } n \leq m \text{ mod } n ..$ 
  from div-mod-equality have
     $m \text{ div } n * n + m \text{ mod } n - m \text{ mod } n = m - m \text{ mod } n$  by simp
  with diff-add-assoc [OF  $\langle m \text{ mod } n \leq m \text{ mod } n \rangle$ , of  $m \text{ div } n * n$ ] have
     $m \text{ div } n * n + (m \text{ mod } n - m \text{ mod } n) = m - m \text{ mod } n$ 
  by simp
  then show ?thesis by simp
qed

```

### 26.3.7 An “induction” law for modulus arithmetic.

```

lemma mod-induct-0:
  assumes step:  $\forall i < p. P i \longrightarrow P ((\text{Suc } i) \text{ mod } p)$ 
  and base:  $P i$  and  $i: i < p$ 
  shows  $P 0$ 
proof (rule ccontr)
  assume contra:  $\neg(P 0)$ 
  from  $i$  have  $p: 0 < p$  by simp
  have  $\forall k. 0 < k \longrightarrow \neg P (p-k)$  (is  $\forall k. ?A k$ )
  proof
    fix  $k$ 
    show ?A  $k$ 
    proof (induct  $k$ )
      show ?A  $0$  by simp — by contradiction
    next
      fix  $n$ 
      assume ih: ?A  $n$ 

```

```

show ?A (Suc n)
proof (clarsimp)
  assume y: P (p - Suc n)
  have n: Suc n < p
  proof (rule ccontr)
    assume ¬(Suc n < p)
    hence p - Suc n = 0
    by simp
  with y contra show False
  by simp
qed
hence n2: Suc (p - Suc n) = p - n by arith
from p have p - Suc n < p by arith
with y step have z: P ((Suc (p - Suc n)) mod p)
  by blast
show False
proof (cases n=0)
  case True
    with z n2 contra show ?thesis by simp
  next
    case False
    with p have p - n < p by arith
    with z n2 False ih show ?thesis by simp
  qed
qed
qed
qed
moreover
from i obtain k where 0 < k ∧ i + k = p
  by (blast dest: less-imp-add-positive)
hence 0 < k ∧ i = p - k by auto
moreover
note base
ultimately
show False by blast
qed

lemma mod-induct:
  assumes step: ∀ i < p. P i ⟶ P ((Suc i) mod p)
  and base: P i and i: i < p and j: j < p
  shows P j
proof -
  have ∀ j < p. P j
  proof
    fix j
    show j < p ⟶ P j (is ?A j)
    proof (induct j)
      from step base i show ?A 0
      by (auto elim: mod-induct-0)
    qed
  qed

```

```

next
  fix k
  assume ih: ?A k
  show ?A (Suc k)
  proof
    assume suc: Suc k < p
    hence k: k < p by simp
    with ih have P k ..
    with step k have P (Suc k mod p)
      by blast
    moreover
    from suc have Suc k mod p = Suc k
      by simp
    ultimately
    show P (Suc k) by simp
  qed
qed
qed
with j show ?thesis by blast
qed

end

```

## 27 Plain: Plain HOL

```

theory Plain
imports Datatype FunDef Record Extraction Divides
begin

```

Plain bootstrap of fundamental HOL tools and packages; does not include *Hilbert-Choice*.

```
ML << path-add ~/src/HOL/Library >>
```

```
end
```

## 28 Relation-Power: Powers of Relations and Functions

```

theory Relation-Power
imports Power Transitive-Closure Plain
begin

```

```

instance
  fun :: (type, type) power ..
  — only type 'a ⇒ 'a should be in class power!

```

**overloading**

$relpow \equiv power :: ('a \times 'a) \text{ set} \Rightarrow nat \Rightarrow ('a \times 'a) \text{ set}$  (**unchecked**)

**begin**

$R \wedge n = R \circ \dots \circ R$ , the  $n$ -fold composition of  $R$

**primrec**  $relpow$  **where**

$(R :: ('a \times 'a) \text{ set}) \wedge 0 = Id$

$| (R :: ('a \times 'a) \text{ set}) \wedge Suc\ n = R \circ (R \wedge n)$

**end**

**overloading**

$funpow \equiv power :: ('a \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow 'a$  (**unchecked**)

**begin**

$f \wedge n = f \circ \dots \circ f$ , the  $n$ -fold composition of  $f$

**primrec**  $funpow$  **where**

$(f :: 'a \Rightarrow 'a) \wedge 0 = id$

$| (f :: 'a \Rightarrow 'a) \wedge Suc\ n = f \circ (f \wedge n)$

**end**

WARNING: due to the limits of Isabelle’s type classes, exponentiation on functions and relations has too general a domain, namely  $('a \times 'b) \text{ set}$  and  $'a \Rightarrow 'b$ . Explicit type constraints may therefore be necessary. For example,  $range\ (f \wedge n) = A$  and  $Range\ (R \wedge n) = B$  need constraints.

Circumvent this problem for code generation:

**primrec**

$fun\text{-}pow :: nat \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$

**where**

$fun\text{-}pow\ 0\ f = id$

$| fun\text{-}pow\ (Suc\ n)\ f = f \circ fun\text{-}pow\ n\ f$

**lemma**  $funpow\text{-}fun\text{-}pow$  [code unfold]:  $f \wedge n = fun\text{-}pow\ n\ f$

**unfolding**  $funpow\text{-}def\ fun\text{-}pow\text{-}def$  ..

**lemma**  $funpow\text{-}add$ :  $f \wedge (m+n) = f \wedge m \circ f \wedge n$

**by** (induct  $m$ )  $simp\text{-}all$

**lemma**  $funpow\text{-}swap1$ :  $f((f \wedge n)\ x) = (f \wedge n)(f\ x)$

**proof** –

**have**  $f((f \wedge n)\ x) = (f \wedge (n+1))\ x$  **unfolding**  $One\text{-}nat\text{-}def$  **by**  $simp$

**also have**  $\dots = (f \wedge n \circ f \wedge 1)\ x$  **by** ( $simp\ only$ :  $funpow\text{-}add$ )

**also have**  $\dots = (f \wedge n)(f\ x)$  **unfolding**  $One\text{-}nat\text{-}def$  **by**  $simp$

**finally show**  $?thesis$  .

**qed**

```

lemma rel-pow-1 [simp]:
  fixes  $R :: ('a * 'a) \text{set}$ 
  shows  $R^1 = R$ 
  unfolding One-nat-def by simp

lemma rel-pow-0-I:  $(x, x) : R^0$ 
by simp

lemma rel-pow-Suc-I:  $\llbracket (x, y) : R^n; (y, z) : R \rrbracket \implies (x, z) : R^{(Suc\ n)}$ 
by auto

lemma rel-pow-Suc-I2:
   $(x, y) : R \implies (y, z) : R^n \implies (x, z) : R^{(Suc\ n)}$ 
apply (induct n arbitrary: z)
apply simp
apply fastsimp
done

lemma rel-pow-0-E:  $\llbracket (x, y) : R^0; x=y \implies P \rrbracket \implies P$ 
by simp

lemma rel-pow-Suc-E:
   $\llbracket (x, z) : R^{(Suc\ n)}; !!y. \llbracket (x, y) : R^n; (y, z) : R \rrbracket \implies P \rrbracket \implies P$ 
by auto

lemma rel-pow-E:
   $\llbracket (x, z) : R^n; \llbracket n=0; x = z \rrbracket \implies P; !!y\ m. \llbracket n = Suc\ m; (x, y) : R^m; (y, z) : R \rrbracket \implies P \rrbracket \implies P$ 
by (cases n) auto

lemma rel-pow-Suc-D2:
   $(x, z) : R^{(Suc\ n)} \implies (\exists y. (x, y) : R \ \& \ (y, z) : R^n)$ 
apply (induct n arbitrary: x z)
apply (blast intro: rel-pow-0-I elim: rel-pow-0-E rel-pow-Suc-E)
apply (blast intro: rel-pow-Suc-I elim: rel-pow-0-E rel-pow-Suc-E)
done

lemma rel-pow-Suc-D2':
   $\forall x\ y\ z. (x, y) : R^n \ \& \ (y, z) : R \dashrightarrow (\exists w. (x, w) : R \ \& \ (w, z) : R^n)$ 
by (induct n) (simp-all, blast)

lemma rel-pow-E2:
   $\llbracket (x, z) : R^n; \llbracket n=0; x = z \rrbracket \implies P; !!y\ m. \llbracket n = Suc\ m; (x, y) : R; (y, z) : R^m \rrbracket \implies P \rrbracket \implies P$ 
apply (case-tac n, simp)
apply (cut-tac n=nat and R=R in rel-pow-Suc-D2', simp, blast)
done

```

```

lemma rtrancI-imp-UN-rel-pow: !!p. p:R* ==> p : (UN n. Rn)
  apply (simp only: split-tupled-all)
  apply (erule rtrancI-induct)
  apply (blast intro: rel-pow-0-I rel-pow-Suc-I)+
done

```

```

lemma rel-pow-imp-rtrancI: !!p. p:Rn ==> p:R*
  apply (simp only: split-tupled-all)
  apply (induct n)
  apply (blast intro: rtrancI-refl elim: rel-pow-0-E)
  apply (blast elim: rel-pow-Suc-E intro: rtrancI-into-rtrancI)
done

```

```

lemma rtrancI-is-UN-rel-pow: R* = (UN n. Rn)
  by (blast intro: rtrancI-imp-UN-rel-pow rel-pow-imp-rtrancI)

```

```

lemma trancI-power:
  x ∈ r+ = (∃ n > 0. x ∈ rn)
  apply (cases x)
  apply simp
  apply (rule iffI)
  apply (drule trancID2)
  apply (clarsimp simp: rtrancI-is-UN-rel-pow)
  apply (rule-tac x=Suc x in exI)
  apply (clarsimp simp: rel-comp-def)
  apply fastsimp
  apply clarsimp
  apply (case-tac n, simp)
  apply clarsimp
  apply (drule rel-pow-imp-rtrancI)
  apply fastsimp
done

```

```

lemma single-valued-rel-pow:
  !!r::('a * 'a) set. single-valued r ==> single-valued (rn)
  apply (rule single-valuedI)
  apply (induct n)
  apply simp
  apply (fast dest: single-valuedD elim: rel-pow-Suc-E)
done

```

**ML**

```

⟨⟨
  val funpow-add = thm funpow-add;
  val rel-pow-1 = thm rel-pow-1;
  val rel-pow-0-I = thm rel-pow-0-I;
  val rel-pow-Suc-I = thm rel-pow-Suc-I;
  val rel-pow-Suc-I2 = thm rel-pow-Suc-I2;

```

```

val rel-pow-0-E = thm rel-pow-0-E;
val rel-pow-Suc-E = thm rel-pow-Suc-E;
val rel-pow-E = thm rel-pow-E;
val rel-pow-Suc-D2 = thm rel-pow-Suc-D2;
val rel-pow-Suc-D2 = thm rel-pow-Suc-D2;
val rel-pow-E2 = thm rel-pow-E2;
val rtrancl-imp-UN-rel-pow = thm rtrancl-imp-UN-rel-pow;
val rel-pow-imp-rtrancl = thm rel-pow-imp-rtrancl;
val rtrancl-is-UN-rel-pow = thm rtrancl-is-UN-rel-pow;
val single-valued-rel-pow = thm single-valued-rel-pow;
>>

end

```

## 29 Equiv-Relations: Equivalence Relations in Higher-Order Set Theory

```

theory Equiv-Relations
imports Finite-Set Relation Plain
begin

```

### 29.1 Equivalence relations

```

locale equiv =
  fixes A and r
  assumes refl-on: refl-on A r
  and sym: sym r
  and trans: trans r

```

Suppes, Theorem 70:  $r$  is an equiv relation iff  $r^{-1} \circ r = r$ .

First half:  $\text{equiv } A \ r \implies r^{-1} \circ r = r$ .

```

lemma sym-trans-comp-subset:
  sym r ==> trans r ==> r^{-1} \circ r \subseteq r
by (unfold trans-def sym-def converse-def) blast

```

```

lemma refl-on-comp-subset: refl-on A r ==> r \subseteq r^{-1} \circ r
by (unfold refl-on-def) blast

```

```

lemma equiv-comp-eq: equiv A r ==> r^{-1} \circ r = r
  apply (unfold equiv-def)
  apply clarify
  apply (rule equalityI)
  apply (iprover intro: sym-trans-comp-subset refl-on-comp-subset)+
done

```

Second half.

```

lemma comp-equivI:

```



```

 $r^{-1} \circ r = r \implies \text{Domain } r = A \implies \text{equiv } A \ r$ 
apply (unfold equiv-def refl-on-def sym-def trans-def)
apply (erule equalityE)
apply (subgoal-tac  $\forall x y. (x, y) \in r \longrightarrow (y, x) \in r$ )
apply fast
apply fast
done

```

## 29.2 Equivalence classes

**lemma** *equiv-class-subset*:

```

 $\text{equiv } A \ r \implies (a, b) \in r \implies r^{\{\{a\}\}} \subseteq r^{\{\{b\}\}}$ 
— lemma for the next result
by (unfold equiv-def trans-def sym-def) blast

```

**theorem** *equiv-class-eq*:  $\text{equiv } A \ r \implies (a, b) \in r \implies r^{\{\{a\}\}} = r^{\{\{b\}\}}$

```

apply (assumption | rule equalityI equiv-class-subset)+
apply (unfold equiv-def sym-def)
apply blast
done

```

**lemma** *equiv-class-self*:  $\text{equiv } A \ r \implies a \in A \implies a \in r^{\{\{a\}\}}$

```

by (unfold equiv-def refl-on-def) blast

```

**lemma** *subset-equiv-class*:

```

 $\text{equiv } A \ r \implies r^{\{\{b\}\}} \subseteq r^{\{\{a\}\}} \implies b \in A \implies (a, b) \in r$ 
— lemma for the next result
by (unfold equiv-def refl-on-def) blast

```

**lemma** *eq-equiv-class*:

```

 $r^{\{\{a\}\}} = r^{\{\{b\}\}} \implies \text{equiv } A \ r \implies b \in A \implies (a, b) \in r$ 
by (iprover intro: equalityD2 subset-equiv-class)

```

**lemma** *equiv-class-nondisjoint*:

```

 $\text{equiv } A \ r \implies x \in (r^{\{\{a\}\}} \cap r^{\{\{b\}\}}) \implies (a, b) \in r$ 
by (unfold equiv-def trans-def sym-def) blast

```

**lemma** *equiv-type*:  $\text{equiv } A \ r \implies r \subseteq A \times A$

```

by (unfold equiv-def refl-on-def) blast

```

**theorem** *equiv-class-eq-iff*:

```

 $\text{equiv } A \ r \implies ((x, y) \in r) = (r^{\{\{x\}\}} = r^{\{\{y\}\}} \ \& \ x \in A \ \& \ y \in A)$ 
by (blast intro!: equiv-class-eq dest: eq-equiv-class equiv-type)

```

**theorem** *eq-equiv-class-iff*:

```

 $\text{equiv } A \ r \implies x \in A \implies y \in A \implies (r^{\{\{x\}\}} = r^{\{\{y\}\}}) = ((x, y) \in r)$ 
by (blast intro!: equiv-class-eq dest: eq-equiv-class equiv-type)

```

### 29.3 Quotients

**definition** *quotient* :: 'a set  $\Rightarrow$  ('a  $\times$  'a) set  $\Rightarrow$  'a set set (infixl '/' 90) where  
 [code del]:  $A//r = (\bigcup x \in A. \{r''\{x\}\})$  — set of equiv classes

**lemma** *quotientI*:  $x \in A \implies r''\{x\} \in A//r$   
 by (unfold quotient-def) blast

**lemma** *quotientE*:  
 $X \in A//r \implies (!x. X = r''\{x\} \implies x \in A \implies P) \implies P$   
 by (unfold quotient-def) blast

**lemma** *Union-quotient*:  $\text{equiv } A \ r \implies \text{Union } (A//r) = A$   
 by (unfold equiv-def refl-on-def quotient-def) blast

**lemma** *quotient-disj*:  
 $\text{equiv } A \ r \implies X \in A//r \implies Y \in A//r \implies X = Y \mid (X \cap Y = \{\})$   
 apply (unfold quotient-def)  
 apply clarify  
 apply (rule equiv-class-eq)  
 apply assumption  
 apply (unfold equiv-def trans-def sym-def)  
 apply blast  
 done

**lemma** *quotient-eqI*:  
 $[[\text{equiv } A \ r; X \in A//r; Y \in A//r; x \in X; y \in Y; (x,y) \in r]] \implies X = Y$   
 apply (clarify elim!: quotientE)  
 apply (rule equiv-class-eq, assumption)  
 apply (unfold equiv-def sym-def trans-def, blast)  
 done

**lemma** *quotient-eq-iff*:  
 $[[\text{equiv } A \ r; X \in A//r; Y \in A//r; x \in X; y \in Y]] \implies (X = Y) = ((x,y) \in r)$   
 apply (rule iffI)  
 prefer 2 apply (blast del: equalityI intro: quotient-eqI)  
 apply (clarify elim!: quotientE)  
 apply (unfold equiv-def sym-def trans-def, blast)  
 done

**lemma** *eq-equiv-class-iff2*:  
 $[[\text{equiv } A \ r; x \in A; y \in A]] \implies (\{x\}/r = \{y\}/r) = ((x,y) : r)$   
 by (simp add: quotient-def eq-equiv-class-iff)

**lemma** *quotient-empty* [simp]:  $\{\}/r = \{\}$   
 by (simp add: quotient-def)

**lemma** *quotient-is-empty* [iff]:  $(A//r = \{\}) = (A = \{\})$

**by**(*simp add: quotient-def*)

**lemma** *quotient-is-empty2* [*iff*]:  $(\{\} = A//r) = (A = \{\})$   
**by**(*simp add: quotient-def*)

**lemma** *singleton-quotient*:  $\{x\} // r = \{r \text{ `` } \{x\}\}$   
**by**(*simp add: quotient-def*)

**lemma** *quotient-diff1*:  
 $\llbracket \text{inj-on } (\%a. \{a\} // r) \ A; \ a \in A \rrbracket \implies (A - \{a\}) // r = A // r - \{a\} // r$   
**apply**(*simp add: quotient-def inj-on-def*)  
**apply** *blast*  
**done**

## 29.4 Defining unary operations upon equivalence classes

A congruence-preserving function

**locale** *congruent* =  
**fixes** *r* **and** *f*  
**assumes** *congruent*:  $(y, z) \in r \implies f\ y = f\ z$

**abbreviation**  
 $RESPECTS :: ('a \implies 'b) \implies ('a * 'a) \text{ set} \implies \text{bool}$   
 $(\text{infixr } \text{respects } 80) \text{ where}$   
 $f \text{ respects } r == \text{congruent } r\ f$

**lemma** *UN-constant-eq*:  $a \in A \implies \forall y \in A. f\ y = c \implies (\bigcup y \in A. f(y)) = c$   
— lemma required to prove *UN-equiv-class*  
**by** *auto*

**lemma** *UN-equiv-class*:  
 $\text{equiv } A\ r \implies f \text{ respects } r \implies a \in A$   
 $\implies (\bigcup x \in r \text{ `` } \{a\}. f\ x) = f\ a$   
— Conversion rule  
**apply** (*rule equiv-class-self [THEN UN-constant-eq], assumption+*)  
**apply** (*unfold equiv-def congruent-def sym-def*)  
**apply** (*blast del: equalityI*)  
**done**

**lemma** *UN-equiv-class-type*:  
 $\text{equiv } A\ r \implies f \text{ respects } r \implies X \in A // r \implies$   
 $(!!x. x \in A \implies f\ x \in B) \implies (\bigcup x \in X. f\ x) \in B$   
**apply** (*unfold quotient-def*)  
**apply** *clarify*  
**apply** (*subst UN-equiv-class*)  
**apply** *auto*  
**done**

Sufficient conditions for injectiveness. Could weaken premises! major premise could be an inclusion; bcong could be  $!!y. y \in A \implies f y \in B$ .

**lemma** *UN-equiv-class-inject*:

```
equiv A r ==> f respects r ==>
  (⋃ x ∈ X. f x) = (⋃ y ∈ Y. f y) ==> X ∈ A//r ==> Y ∈ A//r
  ==> (!x y. x ∈ A ==> y ∈ A ==> f x = f y ==> (x, y) ∈ r)
  ==> X = Y
```

**apply** (*unfold quotient-def*)

**apply** *clarify*

**apply** (*rule equiv-class-eq*)

**apply** *assumption*

**apply** (*subgoal-tac f x = f xa*)

**apply** *blast*

**apply** (*erule box-equals*)

**apply** (*assumption | rule UN-equiv-class*)**+**

**done**

## 29.5 Defining binary operations upon equivalence classes

A congruence-preserving function of two arguments

**locale** *congruent2* =

**fixes** *r1 and r2 and f*

**assumes** *congruent2*:

$(y1, z1) \in r1 \implies (y2, z2) \in r2 \implies f y1 y2 = f z1 z2$

Abbreviation for the common case where the relations are identical

**abbreviation**

*RESPECTS2*:: [*'a* => *'a* => *'b*, (*'a* \* *'a*) *set*] => *bool*

(**infixr** *respects2* 80) **where**

*f respects2 r* == *congruent2 r r f*

**lemma** *congruent2-implies-congruent*:

*equiv A r1 ==> congruent2 r1 r2 f ==> a ∈ A ==> congruent r2 (f a)*

**by** (*unfold congruent-def congruent2-def equiv-def refl-on-def*) *blast*

**lemma** *congruent2-implies-congruent-UN*:

*equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f ==> a ∈ A2 ==>*

*congruent r1 (λx1. ⋃ x2 ∈ r2“{a}. f x1 x2)*

**apply** (*unfold congruent-def*)

**apply** *clarify*

**apply** (*rule equiv-type [THEN subsetD, THEN SigmaE2], assumption+*)

**apply** (*simp add: UN-equiv-class congruent2-implies-congruent*)

**apply** (*unfold congruent2-def equiv-def refl-on-def*)

**apply** (*blast del: equalityI*)

**done**

**lemma** *UN-equiv-class2*:

```

equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f ==> a1 ∈ A1 ==> a2
∈ A2
==> (⋃ x1 ∈ r1“{a1}. ⋃ x2 ∈ r2“{a2}. f x1 x2) = f a1 a2
by (simp add: UN-equiv-class congruent2-implies-congruent
congruent2-implies-congruent-UN)

```

**lemma** *UN-equiv-class-type2*:

```

equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f
==> X1 ∈ A1//r1 ==> X2 ∈ A2//r2
==> (!!x1 x2. x1 ∈ A1 ==> x2 ∈ A2 ==> f x1 x2 ∈ B)
==> (⋃ x1 ∈ X1. ⋃ x2 ∈ X2. f x1 x2) ∈ B
apply (unfold quotient-def)
apply clarify
apply (blast intro: UN-equiv-class-type congruent2-implies-congruent-UN
congruent2-implies-congruent quotientI)
done

```

**lemma** *UN-UN-split-split-eq*:

```

(⋃ (x1, x2) ∈ X. ⋃ (y1, y2) ∈ Y. A x1 x2 y1 y2) =
(⋃ x ∈ X. ⋃ y ∈ Y. (λ(x1, x2). (λ(y1, y2). A x1 x2 y1 y2) y) x)
— Allows a natural expression of binary operators,
— without explicit calls to split
by auto

```

**lemma** *congruent2I*:

```

equiv A1 r1 ==> equiv A2 r2
==> (!!y z w. w ∈ A2 ==> (y,z) ∈ r1 ==> f y w = f z w)
==> (!!y z w. w ∈ A1 ==> (y,z) ∈ r2 ==> f w y = f w z)
==> congruent2 r1 r2 f
— Suggested by John Harrison – the two subproofs may be
— much simpler than the direct proof.
apply (unfold congruent2-def equiv-def refl-on-def)
apply clarify
apply (blast intro: trans)
done

```

**lemma** *congruent2-commuteI*:

```

assumes equivA: equiv A r
and commute: !!y z. y ∈ A ==> z ∈ A ==> f y z = f z y
and cong: !!y z w. w ∈ A ==> (y,z) ∈ r ==> f w y = f w z
shows f respects2 r
apply (rule congruent2I [OF equivA equivA])
apply (rule commute [THEN trans])
apply (rule-tac [3] commute [THEN trans, symmetric])
apply (rule-tac [5] sym)
apply (rule cong | assumption |
erule equivA [THEN equiv-type, THEN subsetD, THEN SigmaE2])
done

```

## 29.6 Quotients and finiteness

Suggested by Florian Kammüller

```

lemma finite-quotient: finite  $A \implies r \subseteq A \times A \implies \text{finite } (A/r)$ 
  — recall equiv  $?A \ ?r \implies ?r \subseteq ?A \times ?A$ 
  apply (rule finite-subset)
    apply (erule-tac [2] finite-Pow-iff [THEN iffD2])
  apply (unfold quotient-def)
  apply blast
done

lemma finite-equiv-class:
  finite  $A \implies r \subseteq A \times A \implies X \in A/r \implies \text{finite } X$ 
  apply (unfold quotient-def)
  apply (rule finite-subset)
  prefer 2 apply assumption
  apply blast
done

lemma equiv-imp-dvd-card:
  finite  $A \implies \text{equiv } A \ r \implies \forall X \in A/r. k \text{ dvd card } X$ 
   $\implies k \text{ dvd card } A$ 
  apply (rule Union-quotient [THEN subst [where  $P = \lambda A. k \text{ dvd card } A$ ]])
  apply assumption
  apply (rule dvd-partition)
  prefer 3 apply (blast dest: quotient-disj)
  apply (simp-all add: Union-quotient equiv-type)
done

lemma card-quotient-disjoint:
   $\llbracket \text{finite } A; \text{inj-on } (\lambda x. \{x\} // r) \ A \rrbracket \implies \text{card}(A/r) = \text{card } A$ 
  apply(simp add:quotient-def)
  apply(subst card-UN-disjoint)
  apply assumption
  apply simp
  apply(fastsimp simp add:inj-on-def)
  apply (simp add:setsum-constant)
done

end

```

## 30 Int: The Integers as Equivalence Classes over Pairs of Natural Numbers

```

theory Int
imports Equiv-Relations Nat Wellfounded
uses

```

```

(Tools/numeral.ML)
(Tools/numeral-syntax.ML)
~~/src/Provers/Arith/assoc-fold.ML
~~/src/Provers/Arith/cancel-numerals.ML
~~/src/Provers/Arith/combine-numerals.ML
(Tools/int-arith.ML)
begin

```

### 30.1 The equivalence relation underlying the integers

**definition** *intrel* ::  $((nat \times nat) \times (nat \times nat))$  set **where**  
 $[code\ del]: intrel = \{(x, y), (u, v) \mid x\ y\ u\ v.\ x + v = u + y\}$

**typedef** (*Integ*)  
*int* = *UNIV* // *intrel*  
**by** (*auto simp add: quotient-def*)

**instantiation** *int* :: {*zero, one, plus, minus, uminus, times, ord, abs, sgn*}  
**begin**

**definition**  
*Zero-int-def*  $[code\ del]: 0 = Abs-Integ\ (intrel\ \{(0, 0)\})$

**definition**  
*One-int-def*  $[code\ del]: 1 = Abs-Integ\ (intrel\ \{(1, 0)\})$

**definition**  
*add-int-def*  $[code\ del]: z + w = Abs-Integ\ (\bigcup (x, y) \in Rep-Integ\ z.\ \bigcup (u, v) \in Rep-Integ\ w.\ intrel\ \{(x + u, y + v)\})$

**definition**  
*minus-int-def*  $[code\ del]:$   
 $-z = Abs-Integ\ (\bigcup (x, y) \in Rep-Integ\ z.\ intrel\ \{(y, x)\})$

**definition**  
*diff-int-def*  $[code\ del]: z - w = z + (-w :: int)$

**definition**  
*mult-int-def*  $[code\ del]: z * w = Abs-Integ\ (\bigcup (x, y) \in Rep-Integ\ z.\ \bigcup (u, v) \in Rep-Integ\ w.\ intrel\ \{(x*u + y*v, x*v + y*u)\})$

**definition**  
*le-int-def*  $[code\ del]:$   
 $z \leq w \iff (\exists x\ y\ u\ v.\ x + v \leq u + y \wedge (x, y) \in Rep-Integ\ z \wedge (u, v) \in Rep-Integ\ w)$

**definition**

*less-int-def* [code del]:  $(z::int) < w \iff z \leq w \wedge z \neq w$

**definition**

*zabs-def*:  $|i::int| = (if\ i < 0\ then\ -\ i\ else\ i)$

**definition**

*zsgn-def*:  $sgn\ (i::int) = (if\ i=0\ then\ 0\ else\ if\ 0<i\ then\ 1\ else\ -\ 1)$

**instance ..**

**end**

### 30.2 Construction of the Integers

**lemma** *intrel-iff* [simp]:  $((x,y),(u,v)) \in intrel = (x+v = u+y)$   
**by** (simp add: intrel-def)

**lemma** *equiv-intrel*: *equiv UNIV intrel*

**by** (simp add: intrel-def equiv-def refl-on-def sym-def trans-def)

Reduces equality of equivalence classes to the *intrel* relation:  $(intrel\ \{\!x\} = intrel\ \{\!y\}) = ((x, y) \in intrel)$

**lemmas** *equiv-intrel-iff* [simp] = *eq-equiv-class-iff* [OF *equiv-intrel UNIV-I UNIV-I*]

All equivalence classes belong to set of representatives

**lemma** [simp]:  $intrel\ \{(x,y)\} \in Integ$

**by** (auto simp add: Integ-def intrel-def quotient-def)

Reduces equality on abstractions to equality on representatives:  $\llbracket x \in Integ; y \in Integ \rrbracket \implies (Abs-Integ\ x = Abs-Integ\ y) = (x = y)$

**declare** *Abs-Integ-inject* [simp,noatp] *Abs-Integ-inverse* [simp,noatp]

Case analysis on the representation of an integer as an equivalence class of pairs of naturals.

**lemma** *eq-Abs-Integ* [case-names *Abs-Integ*, cases type: *int*]:

$(!!x\ y.\ z = Abs-Integ(intrel\ \{(x,y)\}) \implies P) \implies P$

**apply** (rule *Abs-Integ-cases* [of *z*])

**apply** (auto simp add: Integ-def quotient-def)

**done**

### 30.3 Arithmetic Operations

**lemma** *minus*:  $- Abs-Integ(intrel\ \{(x,y)\}) = Abs-Integ(intrel\ \{(y,x)\})$

**proof** –

**have**  $(\lambda(x,y). intrel\ \{(y,x)\})$  respects *intrel*

**by** (simp add: congruent-def)

**thus** ?thesis

**by** (simp add: minus-int-def UN-equiv-class [OF *equiv-intrel*])



qed

lemma add:

$Abs-Integ \text{ (intrel “}\{(x,y)\}\text{)} + Abs-Integ \text{ (intrel “}\{(u,v)\}\text{)} =$   
 $Abs-Integ \text{ (intrel “}\{(x+u, y+v)\}\text{)}$

proof –

have  $(\lambda z w. (\lambda(x,y). (\lambda(u,v). \text{intrel “}\{(x+u, y+v)\}\text{)} w) z)$   
 $\text{respects2 intrel}$

by (simp add: congruent2-def)

thus ?thesis

by (simp add: add-int-def UN-UN-split-split-eq  
 $UN-equiv-class2 [OF equiv-intrel equiv-intrel]$ )

qed

Congruence property for multiplication

lemma mult-congruent2:

$(\%p1\ p2. (\%(x,y). (\%(u,v). \text{intrel “}\{(x*u + y*v, x*v + y*u)\}\text{)} p2) p1)$   
 $\text{respects2 intrel}$

apply (rule equiv-intrel [THEN congruent2-commuteI])

apply (force simp add: mult-ac, clarify)

apply (simp add: congruent-def mult-ac)

apply (rename-tac u v w x y z)

apply (subgoal-tac  $u*y + x*y = w*y + v*y \ \& \ u*z + x*z = w*z + v*z$ )

apply (simp add: mult-ac)

apply (simp add: add-mult-distrib [symmetric])

done

lemma mult:

$Abs-Integ((\text{intrel “}\{(x,y)\}\text{)}) * Abs-Integ((\text{intrel “}\{(u,v)\}\text{)}) =$   
 $Abs-Integ(\text{intrel “}\{(x*u + y*v, x*v + y*u)\}\text{)})$

by (simp add: mult-int-def UN-UN-split-split-eq mult-congruent2  
 $UN-equiv-class2 [OF equiv-intrel equiv-intrel]$ )

The integers form a *comm-ring-1*

instance int :: comm-ring-1

proof

fix i j k :: int

show  $(i + j) + k = i + (j + k)$

by (cases i, cases j, cases k) (simp add: add add-assoc)

show  $i + j = j + i$

by (cases i, cases j) (simp add: add-ac add)

show  $0 + i = i$

by (cases i) (simp add: Zero-int-def add)

show  $-i + i = 0$

by (cases i) (simp add: Zero-int-def minus add)

show  $i - j = i + -j$

by (simp add: diff-int-def)

show  $(i * j) * k = i * (j * k)$

by (cases i, cases j, cases k) (simp add: mult algebra-simps)

```

show  $i * j = j * i$ 
  by (cases i, cases j) (simp add: mult algebra-simps)
show  $1 * i = i$ 
  by (cases i) (simp add: One-int-def mult)
show  $(i + j) * k = i * k + j * k$ 
  by (cases i, cases j, cases k) (simp add: add mult algebra-simps)
show  $0 \neq (1::int)$ 
  by (simp add: Zero-int-def One-int-def)
qed

```

```

lemma int-def:  $of\_nat\ m = Abs\_Integ\ (intrel\ \{(m, 0)\})$ 
by (induct m, simp-all add: Zero-int-def One-int-def add)

```

### 30.4 The $\leq$ Ordering

```

lemma le:
   $(Abs\_Integ(intrel\ \{(x,y)\}) \leq Abs\_Integ(intrel\ \{(u,v)\})) = (x+v \leq u+y)$ 
by (force simp add: le-int-def)

```

```

lemma less:
   $(Abs\_Integ(intrel\ \{(x,y)\}) < Abs\_Integ(intrel\ \{(u,v)\})) = (x+v < u+y)$ 
by (simp add: less-int-def le order-less-le)

```

```

instance int :: linorder

```

```

proof
  fix  $i\ j\ k :: int$ 
  show antisym:  $i \leq j \implies j \leq i \implies i = j$ 
    by (cases i, cases j) (simp add: le)
  show  $(i < j) = (i \leq j \wedge \neg j \leq i)$ 
    by (auto simp add: less-int-def dest: antisym)
  show  $i \leq i$ 
    by (cases i) (simp add: le)
  show  $i \leq j \implies j \leq k \implies i \leq k$ 
    by (cases i, cases j, cases k) (simp add: le)
  show  $i \leq j \vee j \leq i$ 
    by (cases i, cases j) (simp add: le linorder-linear)
qed

```

```

instantiation int :: distrib-lattice

```

```

begin

```

```

definition

```

```

   $(inf :: int \Rightarrow int \Rightarrow int) = min$ 

```

```

definition

```

```

   $(sup :: int \Rightarrow int \Rightarrow int) = max$ 

```

```

instance

```

```

  by intro-classes

```

```

(auto simp add: inf-int-def sup-int-def min-max.sup-inf-distrib1)

end

instance int :: pordered-cancel-ab-semigroup-add
proof
  fix i j k :: int
  show  $i \leq j \implies k + i \leq k + j$ 
  by (cases i, cases j, cases k) (simp add: le add)
qed

```

Strict Monotonicity of Multiplication

strict, in 1st argument; proof is by induction on  $k \geq 0$

```

lemma zmult-zless-mono2-lemma:
   $(i::int) < j \implies 0 < k \implies \text{of\_nat } k * i < \text{of\_nat } k * j$ 
apply (induct k, simp)
apply (simp add: left-distrib)
apply (case-tac k=0)
apply (simp-all add: add-strict-mono)
done

```

```

lemma zero-le-imp-eq-int:  $(0::int) \leq k \implies \exists n. k = \text{of\_nat } n$ 
apply (cases k)
apply (auto simp add: le add int-def Zero-int-def)
apply (rule-tac  $x=x-y$  in exI, simp)
done

```

```

lemma zero-less-imp-eq-int:  $(0::int) < k \implies \exists n>0. k = \text{of\_nat } n$ 
apply (cases k)
apply (simp add: less int-def Zero-int-def)
apply (rule-tac  $x=x-y$  in exI, simp)
done

```

```

lemma zmult-zless-mono2:  $[i < j; (0::int) < k] \implies k*i < k*j$ 
apply (drule zero-less-imp-eq-int)
apply (auto simp add: zmult-zless-mono2-lemma)
done

```

The integers form an ordered integral domain

```

instance int :: ordered-idom
proof
  fix i j k :: int
  show  $i < j \implies 0 < k \implies k * i < k * j$ 
  by (rule zmult-zless-mono2)
  show  $|i| = (\text{if } i < 0 \text{ then } -i \text{ else } i)$ 
  by (simp only: zabs-def)
  show  $\text{sgn } (i::int) = (\text{if } i=0 \text{ then } 0 \text{ else if } 0 < i \text{ then } 1 \text{ else } -1)$ 
  by (simp only: zsgn-def)

```

qed

instance int :: lordered-ring

proof

fix k :: int

show abs k = sup k (- k)

by (auto simp add: sup-int-def zabs-def max-def less-minus-self-iff [symmetric])

qed

lemma zless-imp-add1-zle:  $w < z \implies w + (1::int) \leq z$

apply (cases w, cases z)

apply (simp add: less le add One-int-def)

done

lemma zless-iff-Suc-zadd:

$(w :: int) < z \iff (\exists n. z = w + \text{of-nat} (\text{Suc } n))$

apply (cases z, cases w)

apply (auto simp add: less add int-def)

apply (rename-tac a b c d)

apply (rule-tac  $x=a+d - \text{Suc}(c+b)$  in exI)

apply arith

done

lemmas int-distrib =

left-distrib [of z1::int z2 w, standard]

right-distrib [of w::int z1 z2, standard]

left-diff-distrib [of z1::int z2 w, standard]

right-diff-distrib [of w::int z1 z2, standard]

### 30.5 Embedding of the Integers into any ring-1: of-int

context ring-1

begin

definition

of-int :: int  $\Rightarrow$  'a

where

[code del]: of-int z = contents ( $\bigcup (i, j) \in \text{Rep-Integ } z. \{ \text{of-nat } i - \text{of-nat } j \}$ )

lemma of-int: of-int (Abs-Integ (intrel “  $\{(i, j)\}$  ”)) = of-nat i - of-nat j

proof -

have  $(\lambda(i, j). \{ \text{of-nat } i - (\text{of-nat } j :: 'a) \})$  respects intrel

by (simp add: congruent-def algebra-simps of-nat-add [symmetric])

del: of-nat-add)

thus ?thesis

by (simp add: of-int-def UN-equiv-class [OF equiv-intrel])

qed

lemma of-int-0 [simp]: of-int 0 = 0

**by** (*simp add: of-int Zero-int-def*)

**lemma** *of-int-1* [*simp*]: *of-int 1 = 1*  
**by** (*simp add: of-int One-int-def*)

**lemma** *of-int-add* [*simp*]: *of-int (w+z) = of-int w + of-int z*  
**by** (*cases w, cases z, simp add: algebra-simps of-int add*)

**lemma** *of-int-minus* [*simp*]: *of-int (-z) = - (of-int z)*  
**by** (*cases z, simp add: algebra-simps of-int minus*)

**lemma** *of-int-diff* [*simp*]: *of-int (w - z) = of-int w - of-int z*  
**by** (*simp add: OrderedGroup.diff-minus diff-minus*)

**lemma** *of-int-mult* [*simp*]: *of-int (w\*z) = of-int w \* of-int z*  
**apply** (*cases w, cases z*)  
**apply** (*simp add: algebra-simps of-int mult of-nat-mult*)  
**done**

Collapse nested embeddings

**lemma** *of-int-of-nat-eq* [*simp*]: *of-int (of-nat n) = of-nat n*  
**by** (*induct n*) *auto*

**end**

**context** *ordered-idom*  
**begin**

**lemma** *of-int-le-iff* [*simp*]:  
*of-int w ≤ of-int z ⟷ w ≤ z*  
**by** (*cases w, cases z, simp add: of-int le minus algebra-simps of-nat-add [symmetric]*  
*del: of-nat-add*)

Special cases where either operand is zero

**lemmas** *of-int-0-le-iff* [*simp*] = *of-int-le-iff [of 0, simplified]*  
**lemmas** *of-int-le-0-iff* [*simp*] = *of-int-le-iff [of - 0, simplified]*

**lemma** *of-int-less-iff* [*simp*]:  
*of-int w < of-int z ⟷ w < z*  
**by** (*simp add: not-le [symmetric] linorder-not-le [symmetric]*)

Special cases where either operand is zero

**lemmas** *of-int-0-less-iff* [*simp*] = *of-int-less-iff [of 0, simplified]*  
**lemmas** *of-int-less-0-iff* [*simp*] = *of-int-less-iff [of - 0, simplified]*

**end**

Class for unital rings with characteristic zero. Includes non-ordered rings like the complex numbers.

```

class ring-char-0 = ring-1 + semiring-char-0
begin

lemma of-int-eq-iff [simp]:
  of-int w = of-int z  $\longleftrightarrow$  w = z
apply (cases w, cases z, simp add: of-int)
apply (simp only: diff-eq-eq diff-add-eq eq-diff-eq)
apply (simp only: of-nat-add [symmetric] of-nat-eq-iff)
done

```

Special cases where either operand is zero

```

lemmas of-int-0-eq-iff [simp] = of-int-eq-iff [of 0, simplified]
lemmas of-int-eq-0-iff [simp] = of-int-eq-iff [of - 0, simplified]

```

**end**

Every *ordered-idom* has characteristic zero.

```

subclass (in ordered-idom) ring-char-0 by intro-locales

```

```

lemma of-int-eq-id [simp]: of-int = id
proof
  fix z show of-int z = id z
  by (cases z) (simp add: of-int add minus int-def diff-minus)
qed

```

### 30.6 Magnitude of an Integer, as a Natural Number: *nat*

**definition**

```

  nat :: int  $\Rightarrow$  nat

```

**where**

```

  [code del]: nat z = contents ( $\bigcup (x, y) \in \text{Rep-Integ } z. \{x-y\}$ )

```

```

lemma nat: nat (Abs-Integ (intrel“{(x,y)})) = x-y

```

**proof** –

```

  have ( $\lambda(x,y). \{x-y\}$ ) respects intrel

```

```

  by (simp add: congruent-def) arith

```

```

  thus ?thesis

```

```

  by (simp add: nat-def UN-equiv-class [OF equiv-intrel])

```

**qed**

```

lemma nat-int [simp]: nat (of-nat n) = n

```

```

by (simp add: nat int-def)

```

```

lemma nat-zero [simp]: nat 0 = 0

```

```

by (simp add: Zero-int-def nat)

```

```

lemma int-nat-eq [simp]: of-nat (nat z) = (if 0  $\leq$  z then z else 0)

```

```

by (cases z, simp add: nat le int-def Zero-int-def)

```

**corollary** *nat-0-le*:  $0 \leq z \implies \text{of-nat } (\text{nat } z) = z$   
**by** *simp*

**lemma** *nat-le-0* [*simp*]:  $z \leq 0 \implies \text{nat } z = 0$   
**by** (*cases z, simp add: nat le Zero-int-def*)

**lemma** *nat-le-eq-zle*:  $0 < w \mid 0 \leq z \implies (\text{nat } w \leq \text{nat } z) = (w \leq z)$   
**apply** (*cases w, cases z*)  
**apply** (*simp add: nat le linorder-not-le [symmetric] Zero-int-def, arith*)  
**done**

An alternative condition is  $(0::'a) \leq w$

**corollary** *nat-mono-iff*:  $0 < z \implies (\text{nat } w < \text{nat } z) = (w < z)$   
**by** (*simp add: nat-le-eq-zle linorder-not-le [symmetric]*)

**corollary** *nat-less-eq-zless*:  $0 \leq w \implies (\text{nat } w < \text{nat } z) = (w < z)$   
**by** (*simp add: nat-le-eq-zle linorder-not-le [symmetric]*)

**lemma** *zless-nat-conj* [*simp*]:  $(\text{nat } w < \text{nat } z) = (0 < z \ \& \ w < z)$   
**apply** (*cases w, cases z*)  
**apply** (*simp add: nat le Zero-int-def linorder-not-le [symmetric], arith*)  
**done**

**lemma** *nonneg-eq-int*:  
**fixes**  $z :: \text{int}$   
**assumes**  $0 \leq z$  **and**  $\bigwedge m. z = \text{of-nat } m \implies P$   
**shows**  $P$   
**using** *assms* **by** (*blast dest: nat-0-le sym*)

**lemma** *nat-eq-iff*:  $(\text{nat } w = m) = (\text{if } 0 \leq w \text{ then } w = \text{of-nat } m \text{ else } m=0)$   
**by** (*cases w, simp add: nat le int-def Zero-int-def, arith*)

**corollary** *nat-eq-iff2*:  $(m = \text{nat } w) = (\text{if } 0 \leq w \text{ then } w = \text{of-nat } m \text{ else } m=0)$   
**by** (*simp only: eq-commute [of m] nat-eq-iff*)

**lemma** *nat-less-iff*:  $0 \leq w \implies (\text{nat } w < m) = (w < \text{of-nat } m)$   
**apply** (*cases w*)  
**apply** (*simp add: nat le int-def Zero-int-def linorder-not-le[symmetric], arith*)  
**done**

**lemma** *nat-0-iff* [*simp*]:  $\text{nat}(i::\text{int}) = 0 \iff i \leq 0$   
**by** (*simp add: nat-eq-iff*) *arith*

**lemma** *int-eq-iff*:  $(\text{of-nat } m = z) = (m = \text{nat } z \ \& \ 0 \leq z)$   
**by** (*auto simp add: nat-eq-iff2*)

**lemma** *zero-less-nat-eq* [*simp*]:  $(0 < \text{nat } z) = (0 < z)$   
**by** (*insert zless-nat-conj [of 0], auto*)

**lemma** *nat-add-distrib*:

$[[ (0 :: \text{int}) \leq z; \ 0 \leq z' ]] \implies \text{nat } (z+z') = \text{nat } z + \text{nat } z'$   
**by** (*cases*  $z$ , *cases*  $z'$ , *simp add: nat add le Zero-int-def*)

**lemma** *nat-diff-distrib*:

$[[ (0 :: \text{int}) \leq z'; \ z' \leq z ]] \implies \text{nat } (z-z') = \text{nat } z - \text{nat } z'$   
**by** (*cases*  $z$ , *cases*  $z'$ ,  
*simp add: nat add minus diff-minus le Zero-int-def*)

**lemma** *nat-zminus-int* [*simp*]:  $\text{nat } (- (\text{of-nat } n)) = 0$

**by** (*simp add: int-def minus nat Zero-int-def*)

**lemma** *zless-nat-eq-int-zless*:  $(m < \text{nat } z) = (\text{of-nat } m < z)$

**by** (*cases*  $z$ , *simp add: nat less int-def, arith*)

**context** *ring-1*

**begin**

**lemma** *of-nat-nat*:  $0 \leq z \implies \text{of-nat } (\text{nat } z) = \text{of-int } z$

**by** (*cases*  $z$  *rule: eq-Abs-Integ*)  
*(simp add: nat le of-int Zero-int-def of-nat-diff)*

**end**

For termination proofs:

**lemma** *measure-function-int*[*measure-function*]: *is-measure* (*nat o abs*) ..

### 30.7 Lemmas about the Function *of-nat* and Orderings

**lemma** *negative-zless-0*:  $-(\text{of-nat } (\text{Suc } n)) < (0 :: \text{int})$

**by** (*simp add: order-less-le del: of-nat-Suc*)

**lemma** *negative-zless* [*iff*]:  $-(\text{of-nat } (\text{Suc } n)) < (\text{of-nat } m :: \text{int})$

**by** (*rule negative-zless-0* [*THEN* *order-less-le-trans*], *simp*)

**lemma** *negative-zle-0*:  $-\text{of-nat } n \leq (0 :: \text{int})$

**by** (*simp add: minus-le-iff*)

**lemma** *negative-zle* [*iff*]:  $-\text{of-nat } n \leq (\text{of-nat } m :: \text{int})$

**by** (*rule order-trans* [*OF* *negative-zle-0 of-nat-0-le-iff*])

**lemma** *not-zle-0-negative* [*simp*]:  $\sim (0 \leq -(\text{of-nat } (\text{Suc } n)) :: \text{int})$

**by** (*subst le-minus-iff, simp del: of-nat-Suc*)

**lemma** *int-zle-neg*:  $((\text{of-nat } n :: \text{int}) \leq -\text{of-nat } m) = (n = 0 \ \& \ m = 0)$

**by** (*simp add: int-def le minus Zero-int-def*)

**lemma** *not-int-zless-negative* [*simp*]:  $\sim ((\text{of-nat } n :: \text{int}) < -\text{of-nat } m)$

**by** (*simp add: linorder-not-less*)



**lemma** *negative-eq-positive* [*simp*]:  $((- \text{ of-nat } n :: \text{int}) = \text{ of-nat } m) = (n = 0 \ \& \ m = 0)$

**by** (*force simp add: order-eq-iff [of - of-nat n] int-zle-neg*)

**lemma** *zle-iff-zadd*:  $(w :: \text{int}) \leq z \iff (\exists n. z = w + \text{ of-nat } n)$

**proof** –

**have**  $(w \leq z) = (0 \leq z - w)$

**by** (*simp only: le-diff-eq add-0-left*)

**also have**  $\dots = (\exists n. z - w = \text{ of-nat } n)$

**by** (*auto elim: zero-le-imp-eq-int*)

**also have**  $\dots = (\exists n. z = w + \text{ of-nat } n)$

**by** (*simp only: algebra-simps*)

**finally show** *?thesis* .

**qed**

**lemma** *zadd-int-left*:  $\text{ of-nat } m + (\text{ of-nat } n + z) = \text{ of-nat } (m + n) + (z :: \text{int})$

**by** *simp*

**lemma** *int-Suc0-eq-1*:  $\text{ of-nat } (\text{Suc } 0) = (1 :: \text{int})$

**by** *simp*

This version is proved for all ordered rings, not just integers! It is proved here because attribute *arith-split* is not available in theory *Ring-and-Field*. But is it really better than just rewriting with *abs-if*?

**lemma** *abs-split* [*arith-split, noatp*]:

$P(\text{abs}(a :: 'a :: \text{ordered-idom})) = ((0 \leq a \implies P \ a) \ \& \ (a < 0 \implies P \ (-a)))$

**by** (*force dest: order-less-le-trans simp add: abs-if linorder-not-less*)

**lemma** *negD*:  $(x :: \text{int}) < 0 \implies \exists n. x = - (\text{ of-nat } (\text{Suc } n))$

**apply** (*cases x*)

**apply** (*auto simp add: le-minus Zero-int-def int-def order-less-le*)

**apply** (*rule-tac x=y - Suc x in exI, arith*)

**done**

### 30.8 Cases and induction

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.

**theorem** *int-cases* [*cases type: int, case-names nonneg neg*]:

$[!! \ n. (z :: \text{int}) = \text{ of-nat } n \implies P; \ !! \ n. z = - (\text{ of-nat } (\text{Suc } n)) \implies P \ ] \implies P$

**apply** (*cases z < 0, blast dest!: negD*)

**apply** (*simp add: linorder-not-less del: of-nat-Suc*)

**apply** *auto*

**apply** (*blast dest: nat-0-le [THEN sym]*)

**done**

```

theorem int-induct [induct type: int, case-names nonneg neg]:
  [|! n. P (of-nat n :: int); !n. P (– (of-nat (Suc n))) |] ==> P z
  by (cases z rule: int-cases) auto

```

Contributed by Brian Huffman

```

theorem int-diff-cases:
  obtains (diff) m n where (z::int) = of-nat m – of-nat n
apply (cases z rule: eq-Abs-Integ)
apply (rule-tac m=x and n=y in diff)
apply (simp add: int-def diff-def minus add)
done

```

### 30.9 Binary representation

This formalization defines binary arithmetic in terms of the integers rather than using a datatype. This avoids multiple representations (leading zeroes, etc.) See *ZF/Tools/twos-compl.ML*, function *int-of-binary*, for the numerical interpretation.

The representation expects that  $(m \bmod 2)$  is 0 or 1, even if m is negative; For instance,  $-5 \div 2 = -3$  and  $-5 \bmod 2 = 1$ ; thus  $-5 = (-3)*2 + 1$ . This two’s complement binary representation derives from the paper “An Efficient Representation of Arithmetic for Term Rewriting” by Dave Cohen and Phil Watson, *Rewriting Techniques and Applications*, Springer LNCS 488 (240-251), 1991.

#### 30.9.1 The constructors *Bit0*, *Bit1*, *Pls* and *Min*

```

definition
  Pls :: int where
    [code del]: Pls = 0

```

```

definition
  Min :: int where
    [code del]: Min = – 1

```

```

definition
  Bit0 :: int  $\Rightarrow$  int where
    [code del]: Bit0 k = k + k

```

```

definition
  Bit1 :: int  $\Rightarrow$  int where
    [code del]: Bit1 k = 1 + k + k

```

```

class number = — for numeric types: nat, int, real, ...
  fixes number-of :: int  $\Rightarrow$  'a

```

```

use Tools/numeral.ML

```

**syntax**

*-Numeral* :: *num-const*  $\Rightarrow$  'a    (-)

**use** *Tools/numeral-syntax.ML*

**setup** *NumeralSyntax.setup*

**abbreviation**

*Numeral0*  $\equiv$  *number-of Pls*

**abbreviation**

*Numeral1*  $\equiv$  *number-of (Bit1 Pls)*

**lemma** *Let-number-of [simp]*: *Let (number-of v) f = f (number-of v)*

— Unfold all *lets* involving constants

**unfolding** *Let-def* ..

**definition**

*succ* :: *int*  $\Rightarrow$  *int* **where**

[*code del*]: *succ k = k + 1*

**definition**

*pred* :: *int*  $\Rightarrow$  *int* **where**

[*code del*]: *pred k = k - 1*

**lemmas**

*max-number-of [simp]* = *max-def*

[*of number-of u number-of v, standard, simp*]

**and**

*min-number-of [simp]* = *min-def*

[*of number-of u number-of v, standard, simp*]

— unfolding *minx* and *max* on numerals

**lemmas** *numeral-simps* =

*succ-def pred-def Pls-def Min-def Bit0-def Bit1-def*

Removal of leading zeroes

**lemma** *Bit0-Pls [simp, code post]*:

*Bit0 Pls = Pls*

**unfolding** *numeral-simps* **by** *simp*

**lemma** *Bit1-Min [simp, code post]*:

*Bit1 Min = Min*

**unfolding** *numeral-simps* **by** *simp*

**lemmas** *normalize-bin-simps* =

*Bit0-Pls Bit1-Min*

### 30.9.2 Successor and predecessor functions

Successor

**lemma** *succ-Pls*:

$\text{succ } Pls = Bit1 \ Pls$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *succ-Min*:

$\text{succ } Min = Pls$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *succ-Bit0*:

$\text{succ } (Bit0 \ k) = Bit1 \ k$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *succ-Bit1*:

$\text{succ } (Bit1 \ k) = Bit0 \ (\text{succ } k)$

**unfolding** *numeral-simps* **by** *simp*

**lemmas** *succ-bin-simps* [*simp*] =

*succ-Pls succ-Min succ-Bit0 succ-Bit1*

Predecessor

**lemma** *pred-Pls*:

$\text{pred } Pls = Min$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *pred-Min*:

$\text{pred } Min = Bit0 \ Min$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *pred-Bit0*:

$\text{pred } (Bit0 \ k) = Bit1 \ (\text{pred } k)$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *pred-Bit1*:

$\text{pred } (Bit1 \ k) = Bit0 \ k$

**unfolding** *numeral-simps* **by** *simp*

**lemmas** *pred-bin-simps* [*simp*] =

*pred-Pls pred-Min pred-Bit0 pred-Bit1*

### 30.9.3 Binary arithmetic

Addition

**lemma** *add-Pls*:

$Pls + k = k$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *add-Min*:

$$\text{Min} + k = \text{pred } k$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *add-Bit0-Bit0*:

$$(\text{Bit0 } k) + (\text{Bit0 } l) = \text{Bit0 } (k + l)$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *add-Bit0-Bit1*:

$$(\text{Bit0 } k) + (\text{Bit1 } l) = \text{Bit1 } (k + l)$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *add-Bit1-Bit0*:

$$(\text{Bit1 } k) + (\text{Bit0 } l) = \text{Bit1 } (k + l)$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *add-Bit1-Bit1*:

$$(\text{Bit1 } k) + (\text{Bit1 } l) = \text{Bit0 } (k + \text{succ } l)$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *add-Pls-right*:

$$k + \text{Pls} = k$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *add-Min-right*:

$$k + \text{Min} = \text{pred } k$$

**unfolding** *numeral-simps* **by** *simp*

**lemmas** *add-bin-simps* [*simp*] =

*add-Pls add-Min add-Pls-right add-Min-right*

*add-Bit0-Bit0 add-Bit0-Bit1 add-Bit1-Bit0 add-Bit1-Bit1*

Negation

**lemma** *minus-Pls*:

$$- \text{Pls} = \text{Pls}$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *minus-Min*:

$$- \text{Min} = \text{Bit1 } \text{Pls}$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *minus-Bit0*:

$$- (\text{Bit0 } k) = \text{Bit0 } (- k)$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *minus-Bit1*:

$$- (\text{Bit1 } k) = \text{Bit1 } (\text{pred } (- k))$$

**unfolding** *numeral-simps* **by** *simp*

**lemmas** *minus-bin-simps* [*simp*] =  
*minus-Pls minus-Min minus-Bit0 minus-Bit1*

Subtraction

**lemma** *diff-bin-simps* [*simp*]:  
 $k - Pls = k$   
 $k - Min = succ\ k$   
 $Pls - (Bit0\ l) = Bit0\ (Pls - l)$   
 $Pls - (Bit1\ l) = Bit1\ (Min - l)$   
 $Min - (Bit0\ l) = Bit1\ (Min - l)$   
 $Min - (Bit1\ l) = Bit0\ (Min - l)$   
 $(Bit0\ k) - (Bit0\ l) = Bit0\ (k - l)$   
 $(Bit0\ k) - (Bit1\ l) = Bit1\ (pred\ k - l)$   
 $(Bit1\ k) - (Bit0\ l) = Bit1\ (k - l)$   
 $(Bit1\ k) - (Bit1\ l) = Bit0\ (k - l)$   
**unfolding** *numeral-simps* **by** *simp-all*

Multiplication

**lemma** *mult-Pls*:  
 $Pls * w = Pls$   
**unfolding** *numeral-simps* **by** *simp*

**lemma** *mult-Min*:  
 $Min * k = -\ k$   
**unfolding** *numeral-simps* **by** *simp*

**lemma** *mult-Bit0*:  
 $(Bit0\ k) * l = Bit0\ (k * l)$   
**unfolding** *numeral-simps int-distrib* **by** *simp*

**lemma** *mult-Bit1*:  
 $(Bit1\ k) * l = (Bit0\ (k * l)) + l$   
**unfolding** *numeral-simps int-distrib* **by** *simp*

**lemmas** *mult-bin-simps* [*simp*] =  
*mult-Pls mult-Min mult-Bit0 mult-Bit1*

### 30.9.4 Binary comparisons

Preliminaries

**lemma** *even-less-0-iff*:  
 $a + a < 0 \iff a < (0::'a::ordered-idom)$   
**proof** –  
**have**  $a + a < 0 \iff (1+1)*a < 0$  **by** (*simp add: left-distrib*)  
**also have**  $(1+1)*a < 0 \iff a < 0$   
**by** (*simp add: mult-less-0-iff zero-less-two*  
*order-less-not-sym [OF zero-less-two]*)

finally show ?thesis .  
qed

lemma le-imp-0-less:  
assumes le:  $0 \leq z$   
shows  $(0::int) < 1 + z$   
proof -  
have  $0 \leq z$  by fact  
also have  $\dots < z + 1$  by (rule less-add-one)  
also have  $\dots = 1 + z$  by (simp add: add-ac)  
finally show  $0 < 1 + z$  .  
qed

lemma odd-less-0-iff:  
 $(1 + z + z < 0) = (z < (0::int))$   
proof (cases z rule: int-cases)  
case (nonneg n)  
thus ?thesis by (simp add: linorder-not-less add-assoc add-increasing  
le-imp-0-less [THEN order-less-imp-le])  
next  
case (neg n)  
thus ?thesis by (simp del: of-nat-Suc of-nat-add of-nat-1  
add: algebra-simps of-nat-1 [where 'a=int, symmetric] of-nat-add [symmetric])  
qed

lemma bin-less-0-simps:  
 $Pls < 0 \longleftrightarrow False$   
 $Min < 0 \longleftrightarrow True$   
 $Bit0\ w < 0 \longleftrightarrow w < 0$   
 $Bit1\ w < 0 \longleftrightarrow w < 0$   
unfolding numeral-simps  
by (simp-all add: even-less-0-iff odd-less-0-iff)

lemma less-bin-lemma:  $k < l \longleftrightarrow k - l < (0::int)$   
by simp

lemma le-iff-pred-less:  $k \leq l \longleftrightarrow \text{pred } k < l$   
unfolding numeral-simps  
proof  
have  $k - 1 < k$  by simp  
also assume  $k \leq l$   
finally show  $k - 1 < l$  .  
next  
assume  $k - 1 < l$   
hence  $(k - 1) + 1 \leq l$  by (rule zless-imp-add1-zle)  
thus  $k \leq l$  by simp  
qed

lemma succ-pred:  $\text{succ } (\text{pred } x) = x$

**unfolding** *numeral-simps* by *simp*

Less-than

**lemma** *less-bin-simps* [*simp*]:

$Pls < Pls \longleftrightarrow False$   
 $Pls < Min \longleftrightarrow False$   
 $Pls < Bit0\ k \longleftrightarrow Pls < k$   
 $Pls < Bit1\ k \longleftrightarrow Pls \leq k$   
 $Min < Pls \longleftrightarrow True$   
 $Min < Min \longleftrightarrow False$   
 $Min < Bit0\ k \longleftrightarrow Min < k$   
 $Min < Bit1\ k \longleftrightarrow Min < k$   
 $Bit0\ k < Pls \longleftrightarrow k < Pls$   
 $Bit0\ k < Min \longleftrightarrow k \leq Min$   
 $Bit1\ k < Pls \longleftrightarrow k < Pls$   
 $Bit1\ k < Min \longleftrightarrow k < Min$   
 $Bit0\ k < Bit0\ l \longleftrightarrow k < l$   
 $Bit0\ k < Bit1\ l \longleftrightarrow k \leq l$   
 $Bit1\ k < Bit0\ l \longleftrightarrow k < l$   
 $Bit1\ k < Bit1\ l \longleftrightarrow k < l$

**unfolding** *le-iff-pred-less*

*less-bin-lemma* [of *Pls*]  
*less-bin-lemma* [of *Min*]  
*less-bin-lemma* [of *k*]  
*less-bin-lemma* [of *Bit0 k*]  
*less-bin-lemma* [of *Bit1 k*]  
*less-bin-lemma* [of *pred Pls*]  
*less-bin-lemma* [of *pred k*]

**by** (*simp-all add: bin-less-0-simps succ-pred*)

Less-than-or-equal

**lemma** *le-bin-simps* [*simp*]:

$Pls \leq Pls \longleftrightarrow True$   
 $Pls \leq Min \longleftrightarrow False$   
 $Pls \leq Bit0\ k \longleftrightarrow Pls \leq k$   
 $Pls \leq Bit1\ k \longleftrightarrow Pls \leq k$   
 $Min \leq Pls \longleftrightarrow True$   
 $Min \leq Min \longleftrightarrow True$   
 $Min \leq Bit0\ k \longleftrightarrow Min < k$   
 $Min \leq Bit1\ k \longleftrightarrow Min \leq k$   
 $Bit0\ k \leq Pls \longleftrightarrow k \leq Pls$   
 $Bit0\ k \leq Min \longleftrightarrow k \leq Min$   
 $Bit1\ k \leq Pls \longleftrightarrow k < Pls$   
 $Bit1\ k \leq Min \longleftrightarrow k \leq Min$   
 $Bit0\ k \leq Bit0\ l \longleftrightarrow k \leq l$   
 $Bit0\ k \leq Bit1\ l \longleftrightarrow k \leq l$   
 $Bit1\ k \leq Bit0\ l \longleftrightarrow k < l$   
 $Bit1\ k \leq Bit1\ l \longleftrightarrow k \leq l$

**unfolding** *not-less* [*symmetric*]



**by** (*simp-all add: not-le*)

Equality

**lemma** *eq-bin-simps* [*simp*]:

$Pls = Pls \longleftrightarrow True$

$Pls = Min \longleftrightarrow False$

$Pls = Bit0\ l \longleftrightarrow Pls = l$

$Pls = Bit1\ l \longleftrightarrow False$

$Min = Pls \longleftrightarrow False$

$Min = Min \longleftrightarrow True$

$Min = Bit0\ l \longleftrightarrow False$

$Min = Bit1\ l \longleftrightarrow Min = l$

$Bit0\ k = Pls \longleftrightarrow k = Pls$

$Bit0\ k = Min \longleftrightarrow False$

$Bit1\ k = Pls \longleftrightarrow False$

$Bit1\ k = Min \longleftrightarrow k = Min$

$Bit0\ k = Bit0\ l \longleftrightarrow k = l$

$Bit0\ k = Bit1\ l \longleftrightarrow False$

$Bit1\ k = Bit0\ l \longleftrightarrow False$

$Bit1\ k = Bit1\ l \longleftrightarrow k = l$

**unfolding** *order-eq-iff* [**where** '*a=int*]

**by** (*simp-all add: not-less*)

### 30.10 Converting Numerals to Rings: *number-of*

**class** *number-ring* = *number* + *comm-ring-1* +

**assumes** *number-of-eq*: *number-of* *k* = *of-int* *k*

self-embedding of the integers

**instantiation** *int* :: *number-ring*

**begin**

**definition** *int-number-of-def* [*code del*]:

*number-of* *w* = (*of-int* *w* :: *int*)

**instance proof**

**qed** (*simp only: int-number-of-def*)

**end**

**lemma** *number-of-is-id*:

*number-of* (*k*::*int*) = *k*

**unfolding** *int-number-of-def* **by** *simp*

**lemma** *number-of-succ*:

*number-of* (*succ* *k*) = (*1* + *number-of* *k* :: '*a*::*number-ring*)

**unfolding** *number-of-eq numeral-simps* **by** *simp*

**lemma** *number-of-pred*:

$number-of\ (pred\ w) = (-\ 1 + number-of\ w :: 'a::number-ring)$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

**lemma** *number-of-minus*:  
 $number-of\ (uminus\ w) = (-\ (number-of\ w) :: 'a::number-ring)$   
**unfolding** *number-of-eq* **by** (*rule of-int-minus*)

**lemma** *number-of-add*:  
 $number-of\ (v + w) = (number-of\ v + number-of\ w :: 'a::number-ring)$   
**unfolding** *number-of-eq* **by** (*rule of-int-add*)

**lemma** *number-of-diff*:  
 $number-of\ (v - w) = (number-of\ v - number-of\ w :: 'a::number-ring)$   
**unfolding** *number-of-eq* **by** (*rule of-int-diff*)

**lemma** *number-of-mult*:  
 $number-of\ (v * w) = (number-of\ v * number-of\ w :: 'a::number-ring)$   
**unfolding** *number-of-eq* **by** (*rule of-int-mult*)

The correctness of shifting. But it doesn't seem to give a measurable speed-up.

**lemma** *double-number-of-Bit0*:  
 $(1 + 1) * number-of\ w = (number-of\ (Bit0\ w) :: 'a::number-ring)$   
**unfolding** *number-of-eq numeral-simps left-distrib* **by** *simp*

Converting numerals 0 and 1 to their abstract versions.

**lemma** *numeral-0-eq-0* [*simp*]:  
 $Numeral0 = (0 :: 'a::number-ring)$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

**lemma** *numeral-1-eq-1* [*simp*]:  
 $Numeral1 = (1 :: 'a::number-ring)$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

Special-case simplification for small constants.

Unary minus for the abstract constant 1. Cannot be inserted as a simp rule until later: it is *number-of-Min* re-oriented!

**lemma** *numeral-m1-eq-minus-1*:  
 $(-1 :: 'a::number-ring) = -\ 1$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

**lemma** *mult-minus1* [*simp*]:  
 $-1 * z = -(z :: 'a::number-ring)$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

**lemma** *mult-minus1-right* [*simp*]:  
 $z * -1 = -(z :: 'a::number-ring)$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

**lemma** *minus-number-of-mult* [simp]:  
 $-(\text{number-of } w) * z = \text{number-of } (\text{uminus } w) * (z :: 'a :: \text{number-ring})$   
**unfolding** *number-of-eq* **by** *simp*

Subtraction

**lemma** *diff-number-of-eq*:  
 $\text{number-of } v - \text{number-of } w =$   
 $(\text{number-of } (v + \text{uminus } w) :: 'a :: \text{number-ring})$   
**unfolding** *number-of-eq* **by** *simp*

**lemma** *number-of-Pls*:  
 $\text{number-of } \text{Pls} = (0 :: 'a :: \text{number-ring})$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

**lemma** *number-of-Min*:  
 $\text{number-of } \text{Min} = (- 1 :: 'a :: \text{number-ring})$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

**lemma** *number-of-Bit0*:  
 $\text{number-of } (\text{Bit0 } w) = (0 :: 'a :: \text{number-ring}) + (\text{number-of } w) + (\text{number-of } w)$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

**lemma** *number-of-Bit1*:  
 $\text{number-of } (\text{Bit1 } w) = (1 :: 'a :: \text{number-ring}) + (\text{number-of } w) + (\text{number-of } w)$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

### 30.10.1 Equality of Binary Numbers

First version by Norbert Voelker

**definition**  
 $\text{iszero} :: 'a :: \text{semiring-1} \Rightarrow \text{bool}$   
**where**  
 $\text{iszero } z \longleftrightarrow z = 0$

**lemma** *iszero-0*:  $\text{iszero } 0$   
**by** (*simp add: iszero-def*)

**lemma** *not-iszero-1*:  $\sim \text{iszero } 1$   
**by** (*simp add: iszero-def eq-commute*)

**lemma** *eq-number-of-eq*:  
 $((\text{number-of } x :: 'a :: \text{number-ring}) = \text{number-of } y) =$   
 $\text{iszero } (\text{number-of } (x + \text{uminus } y) :: 'a)$   
**unfolding** *iszero-def number-of-add number-of-minus*  
**by** (*simp add: algebra-simps*)

**lemma** *iszero-number-of-Pls*:

*iszero* ((*number-of Pls*)::'*a*::*number-ring*)  
**unfolding** *iszero-def numeral-0-eq-0* ..

**lemma** *nonzero-number-of-Min*:  
 $\sim$  *iszero* ((*number-of Min*)::'*a*::*number-ring*)  
**unfolding** *iszero-def numeral-m1-eq-minus-1* **by** *simp*

### 30.10.2 Comparisons, for Ordered Rings

**lemmas** *double-eq-0-iff* = *double-zero*

**lemma** *odd-nonzero*:  
 $1 + z + z \neq (0::\text{int})$   
**proof** (*cases z rule: int-cases*)  
  **case** (*nonneg n*)  
    **have**  $le: 0 \leq z+z$  **by** (*simp add: nonneg add-increasing*)  
    **thus** *?thesis* **using** *le-imp-0-less [OF le]*  
    **by** (*auto simp add: add-assoc*)  
  **next**  
  **case** (*neg n*)  
  **show** *?thesis*  
  **proof**  
    **assume** *eq: 1 + z + z = 0*  
    **have**  $(0::\text{int}) < 1 + (\text{of-nat } n + \text{of-nat } n)$   
    **by** (*simp add: le-imp-0-less add-increasing*)  
    **also have**  $\dots = -(1 + z + z)$   
    **by** (*simp add: neg add-assoc [symmetric]*)  
    **also have**  $\dots = 0$  **by** (*simp add: eq*)  
    **finally have**  $0 < 0$  ..  
    **thus** *False* **by** *blast*  
  **qed**  
**qed**

**lemma** *iszero-number-of-Bit0*:  
*iszero* (*number-of (Bit0 w)*::'*a*) =  
*iszero* (*number-of w*::'*a*::{*ring-char-0*,*number-ring*})  
**proof** –  
  **have** (*of-int w + of-int w = (0::'a)*)  $\implies$  (*w = 0*)  
  **proof** –  
    **assume** *eq: of-int w + of-int w = (0::'a)*  
    **then have** *of-int (w + w) = (of-int 0 :: 'a)* **by** *simp*  
    **then have**  $w + w = 0$  **by** (*simp only: of-int-eq-iff*)  
    **then show**  $w = 0$  **by** (*simp only: double-eq-0-iff*)  
  **qed**  
  **thus** *?thesis*  
  **by** (*auto simp add: iszero-def number-of-eq numeral-simps*)  
**qed**

**lemma** *iszero-number-of-Bit1*:

```

~ iszero (number-of (Bit1 w)::'a::{ring-char-0,number-ring})
proof -
  have 1 + of-int w + of-int w ≠ (0::'a)
  proof
    assume eq: 1 + of-int w + of-int w = (0::'a)
    hence of-int (1 + w + w) = (of-int 0 :: 'a) by simp
    hence 1 + w + w = 0 by (simp only: of-int-eq-iff)
    with odd-nonzero show False by blast
  qed
thus ?thesis
  by (auto simp add: iszero-def number-of-eq numeral-simps)
qed

```

```

lemmas iszero-simps =
  iszero-0 not-iszero-1
  iszero-number-of-Pls nonzero-number-of-Min
  iszero-number-of-Bit0 iszero-number-of-Bit1

```

### 30.10.3 The Less-Than Relation

```

lemma double-less-0-iff:
  (a + a < 0) = (a < (0::'a::ordered-idom))
proof -
  have (a + a < 0) = ((1+1)*a < 0) by (simp add: left-distrib)
  also have ... = (a < 0)
    by (simp add: mult-less-0-iff zero-less-two
              order-less-not-sym [OF zero-less-two])
  finally show ?thesis .
qed

```

```

lemma odd-less-0:
  (1 + z + z < 0) = (z < (0::int))
proof (cases z rule: int-cases)
  case (nonneg n)
    thus ?thesis by (simp add: linorder-not-less add-assoc add-increasing
                          le-imp-0-less [THEN order-less-imp-le])
  next
    case (neg n)
    thus ?thesis by (simp del: of-nat-Suc of-nat-add of-nat-1
                          add: algebra-simps of-nat-1 [where 'a=int, symmetric] of-nat-add [symmetric])
qed

```

Less-Than or Equals

Reduces  $a \leq b$  to  $\neg b < a$  for ALL numerals.

```

lemmas le-number-of-eq-not-less =
  linorder-not-less [of number-of w number-of v, symmetric,
    standard]

```

Absolute value (*abs*)

**lemma** *abs-number-of*:  
 $\text{abs}(\text{number-of } x :: 'a :: \{\text{ordered-idom}, \text{number-ring}\}) =$   
 $(\text{if } \text{number-of } x < (0 :: 'a) \text{ then } -\text{number-of } x \text{ else } \text{number-of } x)$   
**by** (*simp add: abs-if*)

Re-orientation of the equation  $\text{nnn} = x$

**lemma** *number-of-reorient*:  
 $(\text{number-of } w = x) = (x = \text{number-of } w)$   
**by** *auto*

#### 30.10.4 Simplification of arithmetic operations on integer constants.

**lemmas** *arith-extra-simps* [*standard*, *simp*] =  
 $\text{number-of-add}$  [*symmetric*]  
 $\text{number-of-minus}$  [*symmetric*]  
 $\text{numeral-m1-eq-minus-1}$  [*symmetric*]  
 $\text{number-of-mult}$  [*symmetric*]  
 $\text{diff-number-of-eq}$  *abs-number-of*

For making a minimal simpset, one must include these default simprules.  
 Also include *simp-thms*.

**lemmas** *arith-simps* =  
 $\text{normalize-bin-simps}$   $\text{pred-bin-simps}$   $\text{succ-bin-simps}$   
 $\text{add-bin-simps}$   $\text{minus-bin-simps}$   $\text{mult-bin-simps}$   
 $\text{abs-zero}$   $\text{abs-one}$  *arith-extra-simps*

Simplification of relational operations

**lemma** *less-number-of* [*simp*]:  
 $(\text{number-of } x :: 'a :: \{\text{ordered-idom}, \text{number-ring}\}) < \text{number-of } y \longleftrightarrow x < y$   
**unfolding**  $\text{number-of-eq}$  **by** (*rule of-int-less-iff*)

**lemma** *le-number-of* [*simp*]:  
 $(\text{number-of } x :: 'a :: \{\text{ordered-idom}, \text{number-ring}\}) \leq \text{number-of } y \longleftrightarrow x \leq y$   
**unfolding**  $\text{number-of-eq}$  **by** (*rule of-int-le-iff*)

**lemma** *eq-number-of* [*simp*]:  
 $(\text{number-of } x :: 'a :: \{\text{ring-char-0}, \text{number-ring}\}) = \text{number-of } y \longleftrightarrow x = y$   
**unfolding**  $\text{number-of-eq}$  **by** (*rule of-int-eq-iff*)

**lemmas** *rel-simps* [*simp*] =  
 $\text{less-number-of}$   $\text{less-bin-simps}$   
 $\text{le-number-of}$   $\text{le-bin-simps}$   
 $\text{eq-number-of-eq}$   $\text{eq-bin-simps}$   
 $\text{iszero-simps}$

#### 30.10.5 Simplification of arithmetic when nested to the right.

**lemma** *add-number-of-left* [*simp*]:

```

number-of v + (number-of w + z) =
  (number-of (v + w) + z::'a::number-ring)
by (simp add: add-assoc [symmetric])

```

```

lemma mult-number-of-left [simp]:
  number-of v * (number-of w * z) =
    (number-of (v * w) * z::'a::number-ring)
by (simp add: mult-assoc [symmetric])

```

```

lemma add-number-of-diff1:
  number-of v + (number-of w - c) =
    number-of (v + w) - (c::'a::number-ring)
by (simp add: diff-minus add-number-of-left)

```

```

lemma add-number-of-diff2 [simp]:
  number-of v + (c - number-of w) =
    number-of (v + uminus w) + (c::'a::number-ring)
by (simp add: algebra-simps diff-number-of-eq [symmetric])

```

### 30.11 The Set of Integers

```

context ring-1
begin

```

```

definition Ints :: 'a set where
  [code del]: Ints = range of-int

```

```

end

```

```

notation (xsymbols)
  Ints ( $\mathbb{Z}$ )

```

```

context ring-1
begin

```

```

lemma Ints-0 [simp]: 0 ∈  $\mathbb{Z}$ 
apply (simp add: Ints-def)
apply (rule range-eqI)
apply (rule of-int-0 [symmetric])
done

```

```

lemma Ints-1 [simp]: 1 ∈  $\mathbb{Z}$ 
apply (simp add: Ints-def)
apply (rule range-eqI)
apply (rule of-int-1 [symmetric])
done

```

```

lemma Ints-add [simp]: a ∈  $\mathbb{Z}$  ⇒ b ∈  $\mathbb{Z}$  ⇒ a + b ∈  $\mathbb{Z}$ 
apply (auto simp add: Ints-def)

```

```

apply (rule range-eqI)
apply (rule of-int-add [symmetric])
done

```

```

lemma Ints-minus [simp]:  $a \in \mathbb{Z} \implies -a \in \mathbb{Z}$ 
apply (auto simp add: Ints-def)
apply (rule range-eqI)
apply (rule of-int-minus [symmetric])
done

```

```

lemma Ints-mult [simp]:  $a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a * b \in \mathbb{Z}$ 
apply (auto simp add: Ints-def)
apply (rule range-eqI)
apply (rule of-int-mult [symmetric])
done

```

```

lemma Ints-cases [cases set: Ints]:
  assumes  $q \in \mathbb{Z}$ 
  obtains (of-int)  $z$  where  $q = \text{of-int } z$ 
  unfolding Ints-def
proof –
  from  $\langle q \in \mathbb{Z} \rangle$  have  $q \in \text{range of-int}$  unfolding Ints-def .
  then obtain  $z$  where  $q = \text{of-int } z$  ..
  then show thesis ..
qed

```

```

lemma Ints-induct [case-names of-int, induct set: Ints]:
   $q \in \mathbb{Z} \implies (\bigwedge z. P (\text{of-int } z)) \implies P q$ 
  by (rule Ints-cases) auto

```

**end**

```

lemma Ints-diff [simp]:  $a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a - b \in \mathbb{Z}$ 
apply (auto simp add: Ints-def)
apply (rule range-eqI)
apply (rule of-int-diff [symmetric])
done

```

The premise involving  $\mathbb{Z}$  prevents  $a = (1::'a) / (2::'a)$ .

```

lemma Ints-double-eq-0-iff:
  assumes in-Ints:  $a \in \text{Ints}$ 
  shows  $(a + a = 0) = (a = (0::'a::\text{ring-char-0}))$ 
proof –
  from in-Ints have  $a \in \text{range of-int}$  unfolding Ints-def [symmetric] .
  then obtain  $z$  where  $a = \text{of-int } z$  ..
  show ?thesis
proof
  assume  $a = 0$ 
  thus  $a + a = 0$  by simp

```



```

next
  assume eq: a + a = 0
  hence of-int (z + z) = (of-int 0 :: 'a) by (simp add: a)
  hence z + z = 0 by (simp only: of-int-eq-iff)
  hence z = 0 by (simp only: double-eq-0-iff)
  thus a = 0 by (simp add: a)
qed
qed

lemma Ints-odd-nonzero:
  assumes in-Ints: a ∈ Ints
  shows 1 + a + a ≠ (0 :: 'a :: ring-char-0)
proof -
  from in-Ints have a ∈ range of-int unfolding Ints-def [symmetric] .
  then obtain z where a: a = of-int z ..
  show ?thesis
  proof
    assume eq: 1 + a + a = 0
    hence of-int (1 + z + z) = (of-int 0 :: 'a) by (simp add: a)
    hence 1 + z + z = 0 by (simp only: of-int-eq-iff)
    with odd-nonzero show False by blast
  qed
qed

```

```

lemma Ints-number-of:
  (number-of w :: 'a :: number-ring) ∈ Ints
  unfolding number-of-eq Ints-def by simp

```

```

lemma Ints-odd-less-0:
  assumes in-Ints: a ∈ Ints
  shows (1 + a + a < 0) = (a < (0 :: 'a :: ordered-idom))
proof -
  from in-Ints have a ∈ range of-int unfolding Ints-def [symmetric] .
  then obtain z where a: a = of-int z ..
  hence ((1 :: 'a) + a + a < 0) = (of-int (1 + z + z) < (of-int 0 :: 'a))
    by (simp add: a)
  also have ... = (z < 0) by (simp only: of-int-less-iff odd-less-0)
  also have ... = (a < 0) by (simp add: a)
  finally show ?thesis .
qed

```

### 30.12 setsum and setprod

```

lemma of-nat-setsum: of-nat (setsum f A) = (∑ x∈A. of-nat(f x))
  apply (cases finite A)
  apply (erule finite-induct, auto)
  done

```

```

lemma of-int-setsum: of-int (setsum f A) = (∑ x∈A. of-int(f x))

```

```

apply (cases finite A)
apply (erule finite-induct, auto)
done

```

```

lemma of-nat-setprod: of-nat (setprod f A) = ( $\prod x \in A. \text{of-nat}(f\ x)$ )
apply (cases finite A)
apply (erule finite-induct, auto simp add: of-nat-mult)
done

```

```

lemma of-int-setprod: of-int (setprod f A) = ( $\prod x \in A. \text{of-int}(f\ x)$ )
apply (cases finite A)
apply (erule finite-induct, auto)
done

```

```

lemmas int-setsum = of-nat-setsum [where 'a=int]
lemmas int-setprod = of-nat-setprod [where 'a=int]

```

### 30.13 Inequality Reasoning for the Arithmetic Simproc

```

lemma add-numeral-0: Numeral0 + a = (a::'a::number-ring)
by simp

```

```

lemma add-numeral-0-right: a + Numeral0 = (a::'a::number-ring)
by simp

```

```

lemma mult-numeral-1: Numeral1 * a = (a::'a::number-ring)
by simp

```

```

lemma mult-numeral-1-right: a * Numeral1 = (a::'a::number-ring)
by simp

```

```

lemma divide-numeral-1: a / Numeral1 = (a::'a::{number-ring,field})
by simp

```

```

lemma inverse-numeral-1:
  inverse Numeral1 = (Numeral1::'a::{number-ring,field})
by simp

```

Theorem lists for the cancellation simprocs. The use of binary numerals for 0 and 1 reduces the number of special cases.

```

lemmas add-0s = add-numeral-0 add-numeral-0-right
lemmas mult-1s = mult-numeral-1 mult-numeral-1-right
               mult-minus1 mult-minus1-right

```

### 30.14 Special Arithmetic Rules for Abstract 0 and 1

Arithmetic computations are defined for binary literals, which leaves 0 and 1 as special cases. Addition already has rules for 0, but not 1. Multiplication and unary minus already have rules for both 0 and 1.

**lemma** *binop-eq*:  $[[f\ x\ y = g\ x\ y; x = x'; y = y']] ==> f\ x'\ y' = g\ x'\ y'$   
**by** *simp*

**lemmas** *add-number-of-eq = number-of-add* [*symmetric*]

Allow 1 on either or both sides

**lemma** *one-add-one-is-two*:  $1 + 1 = (2::'a::\text{number-ring})$   
**by** (*simp del: numeral-1-eq-1 add: numeral-1-eq-1* [*symmetric*] *add-number-of-eq*)

**lemmas** *add-special* =  
*one-add-one-is-two*  
*binop-eq* [*of op +, OF add-number-of-eq numeral-1-eq-1 refl, standard*]  
*binop-eq* [*of op +, OF add-number-of-eq refl numeral-1-eq-1, standard*]

Allow 1 on either or both sides (1-1 already simplifies to 0)

**lemmas** *diff-special* =  
*binop-eq* [*of op -, OF diff-number-of-eq numeral-1-eq-1 refl, standard*]  
*binop-eq* [*of op -, OF diff-number-of-eq refl numeral-1-eq-1, standard*]

Allow 0 or 1 on either side with a binary numeral on the other

**lemmas** *eq-special* =  
*binop-eq* [*of op =, OF eq-number-of-eq numeral-0-eq-0 refl, standard*]  
*binop-eq* [*of op =, OF eq-number-of-eq numeral-1-eq-1 refl, standard*]  
*binop-eq* [*of op =, OF eq-number-of-eq refl numeral-0-eq-0, standard*]  
*binop-eq* [*of op =, OF eq-number-of-eq refl numeral-1-eq-1, standard*]

Allow 0 or 1 on either side with a binary numeral on the other

**lemmas** *less-special* =  
*binop-eq* [*of op <, OF less-number-of numeral-0-eq-0 refl, standard*]  
*binop-eq* [*of op <, OF less-number-of numeral-1-eq-1 refl, standard*]  
*binop-eq* [*of op <, OF less-number-of refl numeral-0-eq-0, standard*]  
*binop-eq* [*of op <, OF less-number-of refl numeral-1-eq-1, standard*]

Allow 0 or 1 on either side with a binary numeral on the other

**lemmas** *le-special* =  
*binop-eq* [*of op ≤, OF le-number-of numeral-0-eq-0 refl, standard*]  
*binop-eq* [*of op ≤, OF le-number-of numeral-1-eq-1 refl, standard*]  
*binop-eq* [*of op ≤, OF le-number-of refl numeral-0-eq-0, standard*]  
*binop-eq* [*of op ≤, OF le-number-of refl numeral-1-eq-1, standard*]

**lemmas** *arith-special*[*simp*] =  
*add-special diff-special eq-special less-special le-special*

**lemma** *min-max-01*:  $\min\ (0::\text{int})\ 1 = 0 \ \&\ \min\ (1::\text{int})\ 0 = 0 \ \&\$   
 $\max\ (0::\text{int})\ 1 = 1 \ \&\ \max\ (1::\text{int})\ 0 = 1$   
**by**(*simp add:min-def max-def*)

```

lemmas min-max-special[simp] =
  min-max-01
  max-def[of 0::int number-of v, standard, simp]
  min-def[of 0::int number-of v, standard, simp]
  max-def[of number-of u 0::int, standard, simp]
  min-def[of number-of u 0::int, standard, simp]
  max-def[of 1::int number-of v, standard, simp]
  min-def[of 1::int number-of v, standard, simp]
  max-def[of number-of u 1::int, standard, simp]
  min-def[of number-of u 1::int, standard, simp]

```

Legacy theorems

```

lemmas zle-int = of-nat-le-iff [where 'a=int]
lemmas int-int-eq = of-nat-eq-iff [where 'a=int]

```

### 30.15 Setting up simplification procedures

```

lemmas int-arith-rules =
  neg-le-iff-le numeral-0-eq-0 numeral-1-eq-1
  minus-zero diff-minus left-minus right-minus
  mult-zero-left mult-zero-right mult-Bit1 mult-1-right
  mult-minus-left mult-minus-right
  minus-add-distrib minus-minus mult-assoc
  of-nat-0 of-nat-1 of-nat-Suc of-nat-add of-nat-mult
  of-int-0 of-int-1 of-int-add of-int-mult

```

```

use Tools/int-arith.ML
declaration << K Int-Arith.setup >>

```

### 30.16 Lemmas About Small Numerals

```

lemma of-int-m1 [simp]: of-int -1 = (-1 :: 'a :: number-ring)
proof -
  have (of-int -1 :: 'a) = of-int (- 1) by simp
  also have ... = - of-int 1 by (simp only: of-int-minus)
  also have ... = -1 by simp
  finally show ?thesis .
qed

```

```

lemma abs-minus-one [simp]: abs (-1) = (1::'a::{ordered-idom,number-ring})
by (simp add: abs-if)

```

```

lemma abs-power-minus-one [simp]:
  abs(-1 ^ n) = (1::'a::{ordered-idom,number-ring,recpower})
by (simp add: power-abs)

```

```

lemma of-int-number-of-eq [simp]:
  of-int (number-of v) = (number-of v :: 'a :: number-ring)

```

by (simp add: number-of-eq)

Lemmas for specialist use, NOT as default simprules

**lemma** mult-2:  $2 * z = (z + z :: 'a :: \text{number-ring})$

**proof** –

**have**  $2 * z = (1 + 1) * z$  **by** simp

**also have**  $\dots = z + z$  **by** (simp add: left-distrib)

**finally show** ?thesis .

**qed**

**lemma** mult-2-right:  $z * 2 = (z + z :: 'a :: \text{number-ring})$

**by** (subst mult-commute, rule mult-2)

### 30.17 More Inequality Reasoning

**lemma** zless-add1-eq:  $(w < z + (1 :: \text{int})) = (w < z \mid w = z)$

**by** arith

**lemma** add1-zle-eq:  $(w + (1 :: \text{int}) \leq z) = (w < z)$

**by** arith

**lemma** zle-diff1-eq [simp]:  $(w \leq z - (1 :: \text{int})) = (w < z)$

**by** arith

**lemma** zle-add1-eq-le [simp]:  $(w < z + (1 :: \text{int})) = (w \leq z)$

**by** arith

**lemma** int-one-le-iff-zero-less:  $((1 :: \text{int}) \leq z) = (0 < z)$

**by** arith

### 30.18 The functions nat and int

Simplify the terms  $\text{int } (0 :: 'a)$ ,  $\text{int } (\text{Suc } 0)$  and  $w + - z$

**declare** Zero-int-def [symmetric, simp]

**declare** One-int-def [symmetric, simp]

**lemmas** diff-int-def-symmetric = diff-int-def [symmetric, simp]

**lemma** nat-0:  $\text{nat } 0 = 0$

**by** (simp add: nat-eq-iff)

**lemma** nat-1:  $\text{nat } 1 = \text{Suc } 0$

**by** (subst nat-eq-iff, simp)

**lemma** nat-2:  $\text{nat } 2 = \text{Suc } (\text{Suc } 0)$

**by** (subst nat-eq-iff, simp)

**lemma** one-less-nat-eq [simp]:  $(\text{Suc } 0 < \text{nat } z) = (1 < z)$

**apply** (insert zless-nat-conj [of 1 z])

```

apply (auto simp add: nat-1)
done

```

This simplifies expressions of the form  $\text{int } n = z$  where  $z$  is an integer literal.

```

lemmas int-eq-iff-number-of [simp] = int-eq-iff [of - number-of v, standard]

```

```

lemma split-nat [arith-split]:
   $P(\text{nat}(i::\text{int})) = ((\forall n. i = \text{of-nat } n \longrightarrow P\ n) \ \&\ (i < 0 \longrightarrow P\ 0))$ 
  (is  $?P = (?L \ \&\ ?R)$ )
proof (cases  $i < 0$ )
  case True thus  $?thesis$  by auto
next
  case False
  have  $?P = ?L$ 
  proof
    assume  $?P$  thus  $?L$  using False by clarsimp
  next
    assume  $?L$  thus  $?P$  using False by simp
  qed
  with False show  $?thesis$  by simp
qed

```

```

context ring-1
begin

```

```

lemma of-int-of-nat [nitpick-const-simp]:
   $\text{of-int } k = (\text{if } k < 0 \text{ then } - \text{of-nat } (\text{nat } (- k)) \text{ else } \text{of-nat } (\text{nat } k))$ 
proof (cases  $k < 0$ )
  case True then have  $0 \leq -k$  by simp
  then have  $\text{of-nat } (\text{nat } (- k)) = \text{of-int } (- k)$  by (rule of-nat-nat)
  with True show  $?thesis$  by simp
next
  case False then show  $?thesis$  by (simp add: not-less of-nat-nat)
qed

end

```

```

lemma nat-mult-distrib:
  fixes  $z\ z' :: \text{int}$ 
  assumes  $0 \leq z$ 
  shows  $\text{nat } (z * z') = \text{nat } z * \text{nat } z'$ 
proof (cases  $0 \leq z'$ )
  case False with assms have  $z * z' \leq 0$ 
    by (simp add: not-le mult-le-0-iff)
  then have  $\text{nat } (z * z') = 0$  by simp
  moreover from False have  $\text{nat } z' = 0$  by simp
  ultimately show  $?thesis$  by simp
next
  case True with assms have  $z * z' \geq 0$  by (simp add: zero-le-mult-iff)

```

```

show ?thesis
  by (rule injD [of of-nat :: nat  $\Rightarrow$  int, OF inj-of-nat])
    (simp only: of-nat-mult of-nat-nat [OF True]
      of-nat-nat [OF assms] of-nat-nat [OF ge-0], simp)
qed

lemma nat-mult-distrib-neg:  $z \leq (0::int) \Rightarrow nat(z * z') = nat(-z) * nat(-z')$ 
apply (rule trans)
apply (rule-tac [2] nat-mult-distrib, auto)
done

lemma nat-abs-mult-distrib:  $nat(abs(w * z)) = nat(abs w) * nat(abs z)$ 
apply (cases z=0 | w=0)
apply (auto simp add: abs-if nat-mult-distrib [symmetric]
  nat-mult-distrib-neg [symmetric] mult-less-0-iff)
done

```

### 30.19 Induction principles for int

Well-founded segments of the integers

#### definition

*int-ge-less-than* ::  $int \Rightarrow (int * int) \text{ set}$

#### where

*int-ge-less-than*  $d = \{(z', z). d \leq z' \ \& \ z' < z\}$

**theorem** wf-int-ge-less-than: wf (*int-ge-less-than*  $d$ )

#### proof –

**have** *int-ge-less-than*  $d \subseteq \text{measure } (\%z. nat(z - d))$

**by** (auto simp add: *int-ge-less-than-def*)

**thus** ?thesis

**by** (rule wf-subset [OF wf-measure])

**qed**

This variant looks odd, but is typical of the relations suggested by Rank-Finder.

#### definition

*int-ge-less-than2* ::  $int \Rightarrow (int * int) \text{ set}$

#### where

*int-ge-less-than2*  $d = \{(z', z). d \leq z \ \& \ z' < z\}$

**theorem** wf-int-ge-less-than2: wf (*int-ge-less-than2*  $d$ )

#### proof –

**have** *int-ge-less-than2*  $d \subseteq \text{measure } (\%z. nat(1 + z - d))$

**by** (auto simp add: *int-ge-less-than2-def*)

**thus** ?thesis

**by** (rule wf-subset [OF wf-measure])

**qed**

**abbreviation**

$$int :: nat \Rightarrow int$$
**where**

$$int \equiv of\_nat$$

**theorem** *int-ge-induct* [*case-names base step, induct set: int*]:

**fixes**  $i :: int$

**assumes**  $ge: k \leq i$  **and**

*base*:  $P\ k$  **and**

*step*:  $\bigwedge i. k \leq i \implies P\ i \implies P\ (i + 1)$

**shows**  $P\ i$

**proof** –

{ **fix**  $n$  **have**  $\bigwedge i::int. n = nat(i-k) \implies k \leq i \implies P\ i$

**proof** (*induct n*)

**case** 0

**hence**  $i = k$  **by** *arith*

**thus**  $P\ i$  **using** *base* **by** *simp*

**next**

**case** (*Suc n*)

**then have**  $n = nat((i - 1) - k)$  **by** *arith*

**moreover**

**have**  $ki1: k \leq i - 1$  **using** *Suc.prem*s **by** *arith*

**ultimately**

**have**  $P(i - 1)$  **by** (*rule Suc.hyps*)

**from** *step* [*OF ki1 this*] **show** ?*case* **by** *simp*

**qed**

}

**with** *ge* **show** ?*thesis* **by** *fast*

**qed**

**theorem** *int-gr-induct* [*case-names base step, induct set: int*]:

**assumes**  $gr: k < (i::int)$  **and**

*base*:  $P(k+1)$  **and**

*step*:  $\bigwedge i. \llbracket k < i; P\ i \rrbracket \implies P(i+1)$

**shows**  $P\ i$

**apply** (*rule int-ge-induct* [*of k + 1*])

**using** *gr* **apply** *arith*

**apply** (*rule base*)

**apply** (*rule step, simp+*)

**done**

**theorem** *int-le-induct* [*consumes 1, case-names base step*]:

**assumes**  $le: i \leq (k::int)$  **and**

*base*:  $P(k)$  **and**

*step*:  $\bigwedge i. \llbracket i \leq k; P\ i \rrbracket \implies P(i - 1)$

**shows**  $P\ i$

**proof** –



```

{ fix n have  $\bigwedge i::int. n = \text{nat}(k-i) \implies i \leq k \implies P\ i$ 
  proof (induct n)
    case 0
    hence  $i = k$  by arith
    thus  $P\ i$  using base by simp
  next
    case (Suc n)
    hence  $n = \text{nat}(k - (i+1))$  by arith
    moreover
    have  $ki1: i + 1 \leq k$  using Suc.prem by arith
    ultimately
    have  $P(i+1)$  by (rule Suc.hyps)
    from step[OF ki1 this] show ?case by simp
  qed
}
with le show ?thesis by fast
qed

```

```

theorem int-less-induct [consumes 1, case-names base step]:
  assumes less:  $(i::int) < k$  and
    base:  $P(k - 1)$  and
    step:  $\bigwedge i. \llbracket i < k; P\ i \rrbracket \implies P(i - 1)$ 
  shows  $P\ i$ 
  apply (rule int-le-induct[of  $k - 1$ ])
  using less apply arith
  apply (rule base)
  apply (rule step, simp+)
  done

```

### 30.20 Intermediate value theorems

```

lemma int-val-lemma:
   $(\forall i < n::nat. \text{abs}(f(i+1) - f\ i) \leq 1) \longrightarrow$ 
   $f\ 0 \leq k \longrightarrow k \leq f\ n \longrightarrow (\exists i \leq n. f\ i = (k::int))$ 
  unfolding One-nat-def
  apply (induct n, simp)
  apply (intro strip)
  apply (erule impE, simp)
  apply (erule tac  $x = n$  in allE, simp)
  apply (case-tac  $k = f\ (Suc\ n)$ )
  apply force
  apply (erule impE)
  apply (simp add: abs-if split add: split-if-asm)
  apply (blast intro: le-SucI)
  done

```

```

lemmas nat0-intermed-int-val = int-val-lemma [rule-format (no-asm)]

```

```

lemma nat-intermed-int-val:

```

```

    [|  $\forall i. m \leq i \ \& \ i < n \longrightarrow \text{abs}(f(i + 1::\text{nat}) - f\ i) \leq 1; m < n;$ 
       $f\ m \leq k; k \leq f\ n$  |] ==> ?  $i. m \leq i \ \& \ i \leq n \ \& \ f\ i = (k::\text{int})$ 
  apply (cut-tac  $n = n - m$  and  $f = \%i. f\ (i + m)$  and  $k = k$ 
    in int-val-lemma)
  unfolding One-nat-def
  apply simp
  apply (erule exE)
  apply (rule-tac  $x = i + m$  in exI, arith)
  done

```

### 30.21 Products and 1, by T. M. Rasmussen

```

lemma zabs-less-one-iff [simp]:  $(|z| < 1) = (z = (0::\text{int}))$ 
by arith

```

```

lemma abs-zmult-eq-1:  $(|m * n| = 1) ==> |m| = (1::\text{int})$ 
apply (cases  $|n|=1$ )
apply (simp add: abs-mult)
apply (rule ccontr)
apply (auto simp add: linorder-neq-iff abs-mult)
apply (subgoal-tac  $2 \leq |m| \ \& \ 2 \leq |n|$ )
  prefer 2 apply arith
apply (subgoal-tac  $2 * 2 \leq |m| * |n|$ , simp)
apply (rule mult-mono, auto)
done

```

```

lemma pos-zmult-eq-1-iff-lemma:  $(m * n = 1) ==> m = (1::\text{int}) \mid m = -1$ 
by (insert abs-zmult-eq-1 [of m n], arith)

```

```

lemma pos-zmult-eq-1-iff:  $0 < (m::\text{int}) ==> (m * n = 1) = (m = 1 \ \& \ n = 1)$ 
apply (auto dest: pos-zmult-eq-1-iff-lemma)
apply (simp add: mult-commute [of m])
apply (frule pos-zmult-eq-1-iff-lemma, auto)
done

```

```

lemma zmult-eq-1-iff:  $(m * n = (1::\text{int})) = ((m = 1 \ \& \ n = 1) \mid (m = -1 \ \& \ n = -1))$ 
apply (rule iffI)
  apply (frule pos-zmult-eq-1-iff-lemma)
  apply (simp add: mult-commute [of m])
  apply (frule pos-zmult-eq-1-iff-lemma, auto)
done

```

```

lemma infinite-UNIV-int:  $\sim \text{finite}(\text{UNIV}::\text{int set})$ 
proof
  assume finite(UNIV::int set)
  moreover have  $\sim (EX\ i::\text{int}. 2 * i = 1)$ 
  by (auto simp: pos-zmult-eq-1-iff)

```

```

ultimately show False using finite-UNIV-inj-surj[of %n::int. n+n]
  by (simp add:inj-on-def surj-def) (blast intro:sym)
qed

```

### 30.22 Integer Powers

```

instantiation int :: recpower
begin

```

```

primrec power-int where
  p ^ 0 = (1::int)
| p ^ (Suc n) = (p::int) * (p ^ n)

```

```

instance proof
  fix z :: int
  fix n :: nat
  show z ^ 0 = 1 by simp
  show z ^ Suc n = z * (z ^ n) by simp
qed

```

```

declare power-int.simps [simp del]

```

```

end

```

```

lemma zpower-zadd-distrib: x ^ (y + z) = ((x ^ y) * (x ^ z)::int)
  by (rule Power.power-add)

```

```

lemma zpower-zpower: (x ^ y) ^ z = (x ^ (y * z)::int)
  by (rule Power.power-mult [symmetric])

```

```

lemma zero-less-zpower-abs-iff [simp]:
  (0 < abs x ^ n) ⟷ (x ≠ (0::int) | n = 0)
  by (induct n) (auto simp add: zero-less-mult-iff)

```

```

lemma zero-le-zpower-abs [simp]: (0::int) ≤ abs x ^ n
  by (induct n) (auto simp add: zero-le-mult-iff)

```

```

lemma of-int-power:
  of-int (z ^ n) = (of-int z ^ n :: 'a::{recpower, ring-1})
  by (induct n) simp-all

```

```

lemma int-power: int (m ^ n) = (int m) ^ n
  by (rule of-nat-power)

```

```

lemmas zpower-int = int-power [symmetric]

```

### 30.23 Further theorems on numerals

#### 30.23.1 Special Simplification for Constants

These distributive laws move literals inside sums and differences.

**lemmas** *left-distrib-number-of [simp] =  
left-distrib [of - - number-of v, standard]*

**lemmas** *right-distrib-number-of [simp] =  
right-distrib [of number-of v, standard]*

**lemmas** *left-diff-distrib-number-of [simp] =  
left-diff-distrib [of - - number-of v, standard]*

**lemmas** *right-diff-distrib-number-of [simp] =  
right-diff-distrib [of number-of v, standard]*

These are actually for fields, like real: but where else to put them?

**lemmas** *zero-less-divide-iff-number-of [simp, noatp] =  
zero-less-divide-iff [of number-of w, standard]*

**lemmas** *divide-less-0-iff-number-of [simp, noatp] =  
divide-less-0-iff [of number-of w, standard]*

**lemmas** *zero-le-divide-iff-number-of [simp, noatp] =  
zero-le-divide-iff [of number-of w, standard]*

**lemmas** *divide-le-0-iff-number-of [simp, noatp] =  
divide-le-0-iff [of number-of w, standard]*

Replaces *inverse #nn* by  $1/\#nn$ . It looks strange, but then other simprocs simplify the quotient.

**lemmas** *inverse-eq-divide-number-of [simp] =  
inverse-eq-divide [of number-of w, standard]*

These laws simplify inequalities, moving unary minus from a term into the literal.

**lemmas** *less-minus-iff-number-of [simp, noatp] =  
less-minus-iff [of number-of v, standard]*

**lemmas** *le-minus-iff-number-of [simp, noatp] =  
le-minus-iff [of number-of v, standard]*

**lemmas** *equation-minus-iff-number-of [simp, noatp] =  
equation-minus-iff [of number-of v, standard]*

**lemmas** *minus-less-iff-number-of [simp, noatp] =  
minus-less-iff [of - number-of v, standard]*

**lemmas** *minus-le-iff-number-of* [*simp*, *noatp*] =  
*minus-le-iff* [*of* - *number-of* *v*, *standard*]

**lemmas** *minus-equation-iff-number-of* [*simp*, *noatp*] =  
*minus-equation-iff* [*of* - *number-of* *v*, *standard*]

To Simplify Inequalities Where One Side is the Constant 1

**lemma** *less-minus-iff-1* [*simp*,*noatp*]:  
**fixes** *b::'b::*{*ordered-idom*,*number-ring*}  
**shows**  $(1 < - b) = (b < -1)$   
**by** *auto*

**lemma** *le-minus-iff-1* [*simp*,*noatp*]:  
**fixes** *b::'b::*{*ordered-idom*,*number-ring*}  
**shows**  $(1 \leq - b) = (b \leq -1)$   
**by** *auto*

**lemma** *equation-minus-iff-1* [*simp*,*noatp*]:  
**fixes** *b::'b::number-ring*  
**shows**  $(1 = - b) = (b = -1)$   
**by** (*subst equation-minus-iff*, *auto*)

**lemma** *minus-less-iff-1* [*simp*,*noatp*]:  
**fixes** *a::'a::*{*ordered-idom*,*number-ring*}  
**shows**  $(- a < 1) = (-1 < a)$   
**by** *auto*

**lemma** *minus-le-iff-1* [*simp*,*noatp*]:  
**fixes** *a::'a::*{*ordered-idom*,*number-ring*}  
**shows**  $(- a \leq 1) = (-1 \leq a)$   
**by** *auto*

**lemma** *minus-equation-iff-1* [*simp*,*noatp*]:  
**fixes** *a::'a::number-ring*  
**shows**  $(- a = 1) = (a = -1)$   
**by** (*subst minus-equation-iff*, *auto*)

Cancellation of constant factors in comparisons ( $<$  and  $\leq$ )

**lemmas** *mult-less-cancel-left-number-of* [*simp*, *noatp*] =  
*mult-less-cancel-left* [*of* *number-of* *v*, *standard*]

**lemmas** *mult-less-cancel-right-number-of* [*simp*, *noatp*] =  
*mult-less-cancel-right* [*of* - *number-of* *v*, *standard*]

**lemmas** *mult-le-cancel-left-number-of* [*simp*, *noatp*] =  
*mult-le-cancel-left* [*of* *number-of* *v*, *standard*]

**lemmas** *mult-le-cancel-right-number-of* [*simp*, *noatp*] =

*mult-le-cancel-right* [of - number-of  $v$ , standard]

Multiplying out constant divisors in comparisons ( $<$ ,  $\leq$  and  $=$ )

**lemmas** *le-divide-eq-number-of1* [simp] = *le-divide-eq* [of - - number-of  $w$ , standard]

**lemmas** *divide-le-eq-number-of1* [simp] = *divide-le-eq* [of - number-of  $w$ , standard]

**lemmas** *less-divide-eq-number-of1* [simp] = *less-divide-eq* [of - - number-of  $w$ , standard]

**lemmas** *divide-less-eq-number-of1* [simp] = *divide-less-eq* [of - number-of  $w$ , standard]

**lemmas** *eq-divide-eq-number-of1* [simp] = *eq-divide-eq* [of - - number-of  $w$ , standard]

**lemmas** *divide-eq-eq-number-of1* [simp] = *divide-eq-eq* [of - number-of  $w$ , standard]

### 30.23.2 Optional Simplification Rules Involving Constants

Simplify quotients that are compared with a literal constant.

**lemmas** *le-divide-eq-number-of* = *le-divide-eq* [of number-of  $w$ , standard]

**lemmas** *divide-le-eq-number-of* = *divide-le-eq* [of - - number-of  $w$ , standard]

**lemmas** *less-divide-eq-number-of* = *less-divide-eq* [of number-of  $w$ , standard]

**lemmas** *divide-less-eq-number-of* = *divide-less-eq* [of - - number-of  $w$ , standard]

**lemmas** *eq-divide-eq-number-of* = *eq-divide-eq* [of number-of  $w$ , standard]

**lemmas** *divide-eq-eq-number-of* = *divide-eq-eq* [of - - number-of  $w$ , standard]

Not good as automatic simprules because they cause case splits.

**lemmas** *divide-const-simps* =

*le-divide-eq-number-of divide-le-eq-number-of less-divide-eq-number-of*  
*divide-less-eq-number-of eq-divide-eq-number-of divide-eq-eq-number-of*  
*le-divide-eq-1 divide-le-eq-1 less-divide-eq-1 divide-less-eq-1*

Division By  $-1$

**lemma** *divide-minus1* [simp]:

$x / -1 = -(x :: 'a :: \{\text{field}, \text{division-by-zero}, \text{number-ring}\})$

**by** *simp*

**lemma** *minus1-divide* [simp]:

$-1 / (x :: 'a :: \{\text{field}, \text{division-by-zero}, \text{number-ring}\}) = -(1/x)$

**by** (*simp add: divide-inverse inverse-minus-eq*)

**lemma** *half-gt-zero-iff*:

$(0 < r/2) = (0 < (r :: 'a :: \{\text{ordered-field}, \text{division-by-zero}, \text{number-ring}\}))$

**by** *auto*

**lemmas** *half-gt-zero* [simp] = *half-gt-zero-iff* [THEN *iffD2*, standard]

### 30.24 Configuration of the code generator

**code-datatype** *Pls Min Bit0 Bit1 number-of* :: *int*  $\Rightarrow$  *int*

```

lemmas pred-succ-numeral-code [code] =
  pred-bin-simps succ-bin-simps

lemmas plus-numeral-code [code] =
  add-bin-simps
  arith-extra-simps(1) [where 'a = int]

lemmas minus-numeral-code [code] =
  minus-bin-simps
  arith-extra-simps(2) [where 'a = int]
  arith-extra-simps(5) [where 'a = int]

lemmas times-numeral-code [code] =
  mult-bin-simps
  arith-extra-simps(4) [where 'a = int]

instantiation int :: eq
begin

definition [code del]: eq-class.eq k l  $\longleftrightarrow$  k - l = (0::int)

instance by default (simp add: eq-int-def)

end

lemma eq-number-of-int-code [code]:
  eq-class.eq (number-of k :: int) (number-of l)  $\longleftrightarrow$  eq-class.eq k l
  unfolding eq-int-def number-of-is-id ..

lemma eq-int-code [code]:
  eq-class.eq Int.Pls Int.Pls  $\longleftrightarrow$  True
  eq-class.eq Int.Pls Int.Min  $\longleftrightarrow$  False
  eq-class.eq Int.Pls (Int.Bit0 k2)  $\longleftrightarrow$  eq-class.eq Int.Pls k2
  eq-class.eq Int.Pls (Int.Bit1 k2)  $\longleftrightarrow$  False
  eq-class.eq Int.Min Int.Pls  $\longleftrightarrow$  False
  eq-class.eq Int.Min Int.Min  $\longleftrightarrow$  True
  eq-class.eq Int.Min (Int.Bit0 k2)  $\longleftrightarrow$  False
  eq-class.eq Int.Min (Int.Bit1 k2)  $\longleftrightarrow$  eq-class.eq Int.Min k2
  eq-class.eq (Int.Bit0 k1) Int.Pls  $\longleftrightarrow$  eq-class.eq k1 Int.Pls
  eq-class.eq (Int.Bit1 k1) Int.Pls  $\longleftrightarrow$  False
  eq-class.eq (Int.Bit0 k1) Int.Min  $\longleftrightarrow$  False
  eq-class.eq (Int.Bit1 k1) Int.Min  $\longleftrightarrow$  eq-class.eq k1 Int.Min
  eq-class.eq (Int.Bit0 k1) (Int.Bit0 k2)  $\longleftrightarrow$  eq-class.eq k1 k2
  eq-class.eq (Int.Bit0 k1) (Int.Bit1 k2)  $\longleftrightarrow$  False
  eq-class.eq (Int.Bit1 k1) (Int.Bit0 k2)  $\longleftrightarrow$  False
  eq-class.eq (Int.Bit1 k1) (Int.Bit1 k2)  $\longleftrightarrow$  eq-class.eq k1 k2
  unfolding eq-equals by simp-all

```

**lemma** *eq-int-refl* [code nbe]:  
 $eq\_class.eq\ (k :: int)\ k \longleftrightarrow True$   
**by** (rule *HOL.eq-refl*)

**lemma** *less-eq-number-of-int-code* [code]:  
 $(number\_of\ k :: int) \leq number\_of\ l \longleftrightarrow k \leq l$   
**unfolding** *number-of-is-id* ..

**lemma** *less-eq-int-code* [code]:  
 $Int.Pl\ s \leq Int.Pl\ s \longleftrightarrow True$   
 $Int.Pl\ s \leq Int.Min \longleftrightarrow False$   
 $Int.Pl\ s \leq Int.Bit0\ k \longleftrightarrow Int.Pl\ s \leq k$   
 $Int.Pl\ s \leq Int.Bit1\ k \longleftrightarrow Int.Pl\ s \leq k$   
 $Int.Min \leq Int.Pl\ s \longleftrightarrow True$   
 $Int.Min \leq Int.Min \longleftrightarrow True$   
 $Int.Min \leq Int.Bit0\ k \longleftrightarrow Int.Min < k$   
 $Int.Min \leq Int.Bit1\ k \longleftrightarrow Int.Min \leq k$   
 $Int.Bit0\ k \leq Int.Pl\ s \longleftrightarrow k \leq Int.Pl\ s$   
 $Int.Bit1\ k \leq Int.Pl\ s \longleftrightarrow k < Int.Pl\ s$   
 $Int.Bit0\ k \leq Int.Min \longleftrightarrow k \leq Int.Min$   
 $Int.Bit1\ k \leq Int.Min \longleftrightarrow k \leq Int.Min$   
 $Int.Bit0\ k1 \leq Int.Bit0\ k2 \longleftrightarrow k1 \leq k2$   
 $Int.Bit0\ k1 \leq Int.Bit1\ k2 \longleftrightarrow k1 \leq k2$   
 $Int.Bit1\ k1 \leq Int.Bit0\ k2 \longleftrightarrow k1 < k2$   
 $Int.Bit1\ k1 \leq Int.Bit1\ k2 \longleftrightarrow k1 \leq k2$   
**by** *simp-all*

**lemma** *less-number-of-int-code* [code]:  
 $(number\_of\ k :: int) < number\_of\ l \longleftrightarrow k < l$   
**unfolding** *number-of-is-id* ..

**lemma** *less-int-code* [code]:  
 $Int.Pl\ s < Int.Pl\ s \longleftrightarrow False$   
 $Int.Pl\ s < Int.Min \longleftrightarrow False$   
 $Int.Pl\ s < Int.Bit0\ k \longleftrightarrow Int.Pl\ s < k$   
 $Int.Pl\ s < Int.Bit1\ k \longleftrightarrow Int.Pl\ s \leq k$   
 $Int.Min < Int.Pl\ s \longleftrightarrow True$   
 $Int.Min < Int.Min \longleftrightarrow False$   
 $Int.Min < Int.Bit0\ k \longleftrightarrow Int.Min < k$   
 $Int.Min < Int.Bit1\ k \longleftrightarrow Int.Min < k$   
 $Int.Bit0\ k < Int.Pl\ s \longleftrightarrow k < Int.Pl\ s$   
 $Int.Bit1\ k < Int.Pl\ s \longleftrightarrow k < Int.Pl\ s$   
 $Int.Bit0\ k < Int.Min \longleftrightarrow k \leq Int.Min$   
 $Int.Bit1\ k < Int.Min \longleftrightarrow k < Int.Min$   
 $Int.Bit0\ k1 < Int.Bit0\ k2 \longleftrightarrow k1 < k2$   
 $Int.Bit0\ k1 < Int.Bit1\ k2 \longleftrightarrow k1 \leq k2$   
 $Int.Bit1\ k1 < Int.Bit0\ k2 \longleftrightarrow k1 < k2$   
 $Int.Bit1\ k1 < Int.Bit1\ k2 \longleftrightarrow k1 < k2$   
**by** *simp-all*



**definition**

*nat-aux* :: *int*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat* **where**  
*nat-aux* *i* *n* = *nat* *i* + *n*

**lemma** [*code*]:

*nat-aux* *i* *n* = (if *i*  $\leq$  0 then *n* else *nat-aux* (*i* - 1) (*Suc* *n*)) — tail recursive  
**by** (*auto simp add: nat-aux-def nat-eq-iff linorder-not-le order-less-imp-le*  
*dest: zless-imp-add1-zle*)

**lemma** [*code*]: *nat* *i* = *nat-aux* *i* 0

**by** (*simp add: nat-aux-def*)

**hide** (**open**) *const nat-aux***lemma** *zero-is-num-zero* [*code*, *code inline*, *symmetric*, *code post*]:

(0::*int*) = *Numeral0*  
**by** *simp*

**lemma** *one-is-num-one* [*code*, *code inline*, *symmetric*, *code post*]:

(1::*int*) = *Numeral1*  
**by** *simp*

**code-modulename** *SML*

*Int Integer*

**code-modulename** *OCaml*

*Int Integer*

**code-modulename** *Haskell*

*Int Integer*

**types-code**

*int* (*int*)

**attach** (*term-of*)  $\langle\langle$ 

*val term-of-int* = *HOLogic.mk-number HOLogic.intT*;

$\rangle\rangle$

**attach** (*test*)  $\langle\langle$ 

*fun gen-int* *i* =

*let val j* = *one-of* [ $\sim$ 1, 1] \* *random-range* 0 *i*

*in* (*j*, *fn* () => *term-of-int* *j*) *end*;

$\rangle\rangle$

**setup**  $\langle\langle$ 

*let*

*fun strip-number-of* (@{*term Int.number-of* :: *int* => *int*} \$ *t*) = *t*  
| *strip-number-of* *t* = *t*;

```

fun numeral-codegen thy defs dep module b t gr =
  let val i = HOLogic.dest-numeral (strip-number-of t)
  in
    SOME (Codegen.str (string-of-int i),
      snd (Codegen.invoke-tycodegen thy defs dep module false HOLogic.intT gr))
  end handle TERM - => NONE;

in

Codegen.add-codegen numeral-codegen numeral-codegen

end
>>

consts-code
  number-of :: int ⇒ int    ((-))
  0 :: int          (0)
  1 :: int          (1)
  uminus :: int ⇒ int      (~)
  op + :: int ⇒ int ⇒ int  ((- +/ -))
  op * :: int ⇒ int ⇒ int  ((- */ -))
  op ≤ :: int ⇒ int ⇒ bool ((- <= / -))
  op < :: int ⇒ int ⇒ bool ((- < / -))

quickcheck-params [default-type = int]

hide (open) const Pls Min Bit0 Bit1 succ pred

```

### 30.25 Legacy theorems

```

lemmas zminus-zminus = minus-minus [of z::int, standard]
lemmas zminus-0 = minus-zero [where 'a=int]
lemmas zminus-zadd-distrib = minus-add-distrib [of z::int w, standard]
lemmas zadd-commute = add-commute [of z::int w, standard]
lemmas zadd-assoc = add-assoc [of z1::int z2 z3, standard]
lemmas zadd-left-commute = add-left-commute [of x::int y z, standard]
lemmas zadd-ac = zadd-assoc zadd-commute zadd-left-commute
lemmas zmult-ac = OrderedGroup.mult-ac
lemmas zadd-0 = OrderedGroup.add-0-left [of z::int, standard]
lemmas zadd-0-right = OrderedGroup.add-0-left [of z::int, standard]
lemmas zadd-zminus-inverse2 = left-minus [of z::int, standard]
lemmas zmult-zminus = mult-minus-left [of z::int w, standard]
lemmas zmult-commute = mult-commute [of z::int w, standard]
lemmas zmult-assoc = mult-assoc [of z1::int z2 z3, standard]
lemmas zadd-zmult-distrib = left-distrib [of z1::int z2 w, standard]
lemmas zadd-zmult-distrib2 = right-distrib [of w::int z1 z2, standard]
lemmas zdiff-zmult-distrib = left-diff-distrib [of z1::int z2 w, standard]
lemmas zdiff-zmult-distrib2 = right-diff-distrib [of w::int z1 z2, standard]

```

```

lemmas zmult-1 = mult-1-left [of z::int, standard]
lemmas zmult-1-right = mult-1-right [of z::int, standard]

lemmas zle-refl = order-refl [of w::int, standard]
lemmas zle-trans = order-trans [where 'a=int and x=i and y=j and z=k,
standard]
lemmas zle-anti-sym = order-antisym [of z::int w, standard]
lemmas zle-linear = linorder-linear [of z::int w, standard]
lemmas zless-linear = linorder-less-linear [where 'a = int]

lemmas zadd-left-mono = add-left-mono [of i::int j k, standard]
lemmas zadd-strict-right-mono = add-strict-right-mono [of i::int j k, standard]
lemmas zadd-zless-mono = add-less-le-mono [of w'::int w z' z, standard]

lemmas int-0-less-1 = zero-less-one [where 'a=int]
lemmas int-0-neq-1 = zero-neq-one [where 'a=int]

lemmas inj-int = inj-of-nat [where 'a=int]
lemmas zadd-int = of-nat-add [where 'a=int, symmetric]
lemmas int-mult = of-nat-mult [where 'a=int]
lemmas zmult-int = of-nat-mult [where 'a=int, symmetric]
lemmas int-eq-0-conv = of-nat-eq-0-iff [where 'a=int and m=n, standard]
lemmas zless-int = of-nat-less-iff [where 'a=int]
lemmas int-less-0-conv = of-nat-less-0-iff [where 'a=int and m=k, standard]
lemmas zero-less-int-conv = of-nat-0-less-iff [where 'a=int]
lemmas zero-zle-int = of-nat-0-le-iff [where 'a=int]
lemmas int-le-0-conv = of-nat-le-0-iff [where 'a=int and m=n, standard]
lemmas int-0 = of-nat-0 [where 'a=int]
lemmas int-1 = of-nat-1 [where 'a=int]
lemmas int-Suc = of-nat-Suc [where 'a=int]
lemmas abs-int-eq = abs-of-nat [where 'a=int and n=m, standard]
lemmas of-int-int-eq = of-int-of-nat-eq [where 'a=int]
lemmas zdiff-int = of-nat-diff [where 'a=int, symmetric]
lemmas zless-le = less-int-def
lemmas int-eq-of-nat = TrueI

end

```

### 31 IntDiv: The Division Operators div and mod

```

theory IntDiv
imports Int Divides FunDef
uses
  ~~/src/Provers/Arith/cancel-numeral-factor.ML
  ~~/src/Provers/Arith/extract-common-term.ML
  (Tools/int-factor-simprocs.ML)
begin

```

$$[code]: \text{divmod-rel } a \ b = (\lambda(q, r). \ a = b * q + r \wedge \\ (if \ 0 < b \text{ then } 0 \leq r \wedge r < b \text{ else } b < r \wedge r \leq 0))$$

```
[code]: adjust  $b = (\lambda(q, r). \text{ if } 0 \leq r - b \text{ then } (2 * q + 1, r - b)$ 
      else  $(2 * q, r))$ 
```

instance ..

**end**

**lemma** *divmod-mod-div*:

*divmod*  $p \ q = (p \ \text{div} \ q, p \ \text{mod} \ q)$   
**by** (*auto simp add: div-def mod-def*)

Here is the division algorithm in ML:

```

fun posDivAlg (a,b) =
  if a<b then (0,a)
  else let val (q,r) = posDivAlg(a, 2*b)
        in if 0<=r-b then (2*q+1, r-b) else (2*q, r)
      end

fun negDivAlg (a,b) =
  if 0<=a+b then (~1,a+b)
  else let val (q,r) = negDivAlg(a, 2*b)
        in if 0<=r-b then (2*q+1, r-b) else (2*q, r)
      end;

fun negateSnd (q,r:int) = (q,~r);

fun divmod (a,b) = if 0<=a then
  if b>0 then posDivAlg (a,b)
  else if a=0 then (0,0)
else negateSnd (negDivAlg (~a,~b))
  else
  if 0<b then negDivAlg (a,b)
  else negateSnd (posDivAlg (~a,~b));

```

### 31.1 Uniqueness and Monotonicity of Quotients and Remainders

**lemma** *unique-quotient-lemma*:

$[| \ b * q' + r' \leq b * q + r; \ 0 \leq r'; \ r' < b; \ r < b \ |]$   
 $\implies q' \leq (q :: \text{int})$   
**apply** (*subgoal-tac*  $r' + b * (q' - q) \leq r$ )  
**prefer** 2 **apply** (*simp add: right-diff-distrib*)  
**apply** (*subgoal-tac*  $0 < b * (1 + q - q')$ )  
**apply** (*erule-tac* [2] *order-le-less-trans*)  
**prefer** 2 **apply** (*simp add: right-diff-distrib right-distrib*)  
**apply** (*subgoal-tac*  $b * q' < b * (1 + q)$ )  
**prefer** 2 **apply** (*simp add: right-diff-distrib right-distrib*)  
**apply** (*simp add: mult-less-cancel-left*)  
**done**

**lemma** *unique-quotient-lemma-neg*:

$[[\ b * q' + r' \leq b * q + r; \ r \leq 0; \ b < r; \ b < r' \ ]]$   
 $\implies q \leq (q'::int)$

**by** (*rule-tac*  $b = -b$  **and**  $r = -r'$  **and**  $r' = -r$  **in** *unique-quotient-lemma*,  
*auto*)

**lemma** *unique-quotient*:

$[[\ \text{divmod-rel } a \ b \ (q, r); \ \text{divmod-rel } a \ b \ (q', r'); \ b \neq 0 \ ]]$   
 $\implies q = q'$

**apply** (*simp add: divmod-rel-def linorder-neg-iff split: split-if-asm*)

**apply** (*blast intro: order-antisym*

*dest: order-eq-refl [THEN unique-quotient-lemma]*

*order-eq-refl [THEN unique-quotient-lemma-neg] sym*) +

**done**

**lemma** *unique-remainder*:

$[[\ \text{divmod-rel } a \ b \ (q, r); \ \text{divmod-rel } a \ b \ (q', r'); \ b \neq 0 \ ]]$   
 $\implies r = r'$

**apply** (*subgoal-tac*  $q = q'$ )

**apply** (*simp add: divmod-rel-def*)

**apply** (*blast intro: unique-quotient*)

**done**

### 31.2 Correctness of *posDivAlg*, the Algorithm for Non-Negative Dividends

And positive divisors

**lemma** *adjust-eq [simp]*:

*adjust*  $b \ (q, r) =$   
 $(\text{let } \text{diff} = r - b \text{ in}$   
 $\text{if } 0 \leq \text{diff} \text{ then } (2 * q + 1, \text{diff})$   
 $\text{else } (2 * q, r))$

**by** (*simp add: Let-def adjust-def*)

**declare** *posDivAlg.simps* [*simp del*]

use with a *simproc* to avoid repeatedly proving the premise

**lemma** *posDivAlg-eqn*:

$0 < b \implies$   
 $\text{posDivAlg } a \ b = (\text{if } a < b \text{ then } (0, a) \text{ else } \text{adjust } b \ (\text{posDivAlg } a \ (2 * b)))$

**by** (*rule posDivAlg.simps [THEN trans], simp*)

Correctness of *posDivAlg*: it computes quotients correctly

**theorem** *posDivAlg-correct*:

**assumes**  $0 \leq a$  **and**  $0 < b$

**shows**  $\text{divmod-rel } a \ b \ (\text{posDivAlg } a \ b)$

```

using prems apply (induct a b rule: posDivAlg.induct)
apply auto
apply (simp add: divmod-rel-def)
apply (subst posDivAlg-eqn, simp add: right-distrib)
apply (case-tac a < b)
apply simp-all
apply (erule splitE)
apply (auto simp add: right-distrib Let-def)
done

```

### 31.3 Correctness of *negDivAlg*, the Algorithm for Negative Dividends

And positive divisors

```
declare negDivAlg.simps [simp del]
```

use with a simproc to avoid repeatedly proving the premise

```

lemma negDivAlg-eqn:
  0 < b ==>
    negDivAlg a b =
      (if 0 ≤ a+b then (-1, a+b) else adjust b (negDivAlg a (2*b)))
by (rule negDivAlg.simps [THEN trans], simp)

```

```

lemma negDivAlg-correct:
  assumes a < 0 and b > 0
  shows divmod-rel a b (negDivAlg a b)
using prems apply (induct a b rule: negDivAlg.induct)
apply (auto simp add: linorder-not-le)
apply (simp add: divmod-rel-def)
apply (subst negDivAlg-eqn, assumption)
apply (case-tac a + b < (0::int))
apply simp-all
apply (erule splitE)
apply (auto simp add: right-distrib Let-def)
done

```

### 31.4 Existence Shown by Proving the Division Algorithm to be Correct

```

lemma divmod-rel-0: b ≠ 0 ==> divmod-rel 0 b (0, 0)
by (auto simp add: divmod-rel-def linorder-neq-iff)

```

```

lemma posDivAlg-0 [simp]: posDivAlg 0 b = (0, 0)
by (subst posDivAlg.simps, auto)

```

```

lemma negDivAlg-minus1 [simp]: negDivAlg -1 b = (-1, b - 1)
by (subst negDivAlg.simps, auto)

```

**lemma** *negateSnd-eq* [simp]:  $\text{negateSnd}(q, r) = (q, -r)$   
**by** (simp add: *negateSnd-def*)

**lemma** *divmod-rel-neg*:  $\text{divmod-rel } (-a) (-b) \text{ } qr \implies \text{divmod-rel } a \ b \ (\text{negateSnd } qr)$   
**by** (auto simp add: *split-ifs divmod-rel-def*)

**lemma** *divmod-correct*:  $b \neq 0 \implies \text{divmod-rel } a \ b \ (\text{divmod } a \ b)$   
**by** (force simp add: *linorder-neg-iff divmod-rel-0 divmod-def divmod-rel-neg posDivAlg-correct negDivAlg-correct*)

Arbitrary definitions for division by zero. Useful to simplify certain equations.

**lemma** *DIVISION-BY-ZERO* [simp]:  $a \text{ div } (0::\text{int}) = 0 \ \& \ a \text{ mod } (0::\text{int}) = a$   
**by** (simp add: *div-def mod-def divmod-def posDivAlg.simps*)

Basic laws about division and remainder

**lemma** *zmod-zdiv-equality*:  $(a::\text{int}) = b * (a \text{ div } b) + (a \text{ mod } b)$   
**apply** (case-tac  $b = 0$ , simp)  
**apply** (cut-tac  $a = a$  **and**  $b = b$  **in** *divmod-correct*)  
**apply** (auto simp add: *divmod-rel-def div-def mod-def*)  
**done**

**lemma** *zdiv-zmod-equality*:  $(b * (a \text{ div } b) + (a \text{ mod } b)) + k = (a::\text{int}) + k$   
**by** (simp add: *zmod-zdiv-equality[symmetric]*)

**lemma** *zdiv-zmod-equality2*:  $((a \text{ div } b) * b + (a \text{ mod } b)) + k = (a::\text{int}) + k$   
**by** (simp add: *mult-commute zmod-zdiv-equality[symmetric]*)

Tool setup

**ML**  $\ll$   
*local*

```
structure CancelDivMod = CancelDivModFun(
  struct
    val div-name = @{const-name Divides.div};
    val mod-name = @{const-name Divides.mod};
    val mk-binop = HOLogic.mk-binop;
    val mk-sum = Int-Numeral-Simprocs.mk-sum HOLogic.intT;
    val dest-sum = Int-Numeral-Simprocs.dest-sum;
    val div-mod-eqs =
      map mk-meta-eq [ @{thm zdiv-zmod-equality},
        @{thm zdiv-zmod-equality2} ];
    val trans = trans;
    val prove-eq-sums =
      let
        val_simps = @{thm diff-int-def} :: Int-Numeral-Simprocs.add-0s @ @{thms
          zadd-ac}
```



```

    in Arith-Data.prove-conv2 all-tac (Arith-Data.simp-all-tac_simps) end;
end)

```

```

in

```

```

val cancel-zdiv-zmod-proc = Simplifier.simproc (the-context ())
  cancel-zdiv-zmod [(m::int) + n] (K CancelDivMod.proc)

```

```

end;

```

```

Addsimprocs [cancel-zdiv-zmod-proc]
>>

```

```

lemma pos-mod-conj : (0::int) < b ==> 0 ≤ a mod b & a mod b < b
apply (cut-tac a = a and b = b in divmod-correct)
apply (auto simp add: divmod-rel-def mod-def)
done

```

```

lemmas pos-mod-sign [simp] = pos-mod-conj [THEN conjunct1, standard]
and pos-mod-bound [simp] = pos-mod-conj [THEN conjunct2, standard]

```

```

lemma neg-mod-conj : b < (0::int) ==> a mod b ≤ 0 & b < a mod b
apply (cut-tac a = a and b = b in divmod-correct)
apply (auto simp add: divmod-rel-def div-def mod-def)
done

```

```

lemmas neg-mod-sign [simp] = neg-mod-conj [THEN conjunct1, standard]
and neg-mod-bound [simp] = neg-mod-conj [THEN conjunct2, standard]

```

### 31.5 General Properties of div and mod

```

lemma divmod-rel-div-mod: b ≠ 0 ==> divmod-rel a b (a div b, a mod b)
apply (cut-tac a = a and b = b in zmod-zdiv-equality)
apply (force simp add: divmod-rel-def linorder-neq-iff)
done

```

```

lemma divmod-rel-div: [| divmod-rel a b (q, r); b ≠ 0 |] ==> a div b = q
by (simp add: divmod-rel-div-mod [THEN unique-quotient])

```

```

lemma divmod-rel-mod: [| divmod-rel a b (q, r); b ≠ 0 |] ==> a mod b = r
by (simp add: divmod-rel-div-mod [THEN unique-remainder])

```

```

lemma div-pos-pos-trivial: [| (0::int) ≤ a; a < b |] ==> a div b = 0
apply (rule divmod-rel-div)
apply (auto simp add: divmod-rel-def)
done

```

```

lemma div-neg-neg-trivial: [| a ≤ (0::int); b < a |] ==> a div b = 0
apply (rule divmod-rel-div)

```

**apply** (*auto simp add: divmod-rel-def*)  
**done**

**lemma** *div-pos-neg-trivial*:  $[(0::int) < a; a+b \leq 0] \implies a \text{ div } b = -1$   
**apply** (*rule divmod-rel-div*)  
**apply** (*auto simp add: divmod-rel-def*)  
**done**

**lemma** *mod-pos-pos-trivial*:  $[(0::int) \leq a; a < b] \implies a \text{ mod } b = a$   
**apply** (*rule-tac q = 0 in divmod-rel-mod*)  
**apply** (*auto simp add: divmod-rel-def*)  
**done**

**lemma** *mod-neg-neg-trivial*:  $[a \leq (0::int); b < a] \implies a \text{ mod } b = a$   
**apply** (*rule-tac q = 0 in divmod-rel-mod*)  
**apply** (*auto simp add: divmod-rel-def*)  
**done**

**lemma** *mod-pos-neg-trivial*:  $[(0::int) < a; a+b \leq 0] \implies a \text{ mod } b = a+b$   
**apply** (*rule-tac q = -1 in divmod-rel-mod*)  
**apply** (*auto simp add: divmod-rel-def*)  
**done**

There is no *mod-neg-pos-trivial*.

**lemma** *zdiv-zminus-zminus* [*simp*]:  $(-a) \text{ div } (-b) = a \text{ div } (b::int)$   
**apply** (*case-tac b = 0, simp*)  
**apply** (*simp add: divmod-rel-div-mod [THEN divmod-rel-neg, simplified,*  
*THEN divmod-rel-div, THEN sym]*)

**done**

**lemma** *zmod-zminus-zminus* [*simp*]:  $(-a) \text{ mod } (-b) = -(a \text{ mod } (b::int))$   
**apply** (*case-tac b = 0, simp*)  
**apply** (*subst divmod-rel-div-mod [THEN divmod-rel-neg, simplified, THEN divmod-rel-mod],*  
*auto*)  
**done**

### 31.6 Laws for div and mod with Unary Minus

**lemma** *zminus1-lemma*:  
 $\text{divmod-rel } a \ b \ (q, r) \implies \text{divmod-rel } (-a) \ b \ (\text{if } r=0 \text{ then } -q \text{ else } -q - 1,$   
 $\text{if } r=0 \text{ then } 0 \text{ else } b-r)$   
**by** (*force simp add: split-ifs divmod-rel-def linorder-neq-iff right-diff-distrib*)

```

lemma zdiv-zminus1-eq-if:
   $b \neq (0::int)$ 
   $\implies (-a) \text{ div } b =$ 
     $(\text{if } a \bmod b = 0 \text{ then } -(a \text{ div } b) \text{ else } -(a \text{ div } b) - 1)$ 
by (blast intro: divmod-rel-div-mod [THEN zminus1-lemma, THEN divmod-rel-div])

lemma zmod-zminus1-eq-if:
   $(-a::int) \bmod b = (\text{if } a \bmod b = 0 \text{ then } 0 \text{ else } b - (a \bmod b))$ 
apply (case-tac b = 0, simp)
apply (blast intro: divmod-rel-div-mod [THEN zminus1-lemma, THEN divmod-rel-mod])
done

lemma zmod-zminus1-not-zero:
  fixes  $k \ l :: int$ 
  shows  $-k \bmod l \neq 0 \implies k \bmod l \neq 0$ 
  unfolding zmod-zminus1-eq-if by auto

lemma zdiv-zminus2:  $a \text{ div } (-b) = (-a::int) \text{ div } b$ 
by (cut-tac a = -a in zdiv-zminus-zminus, auto)

lemma zmod-zminus2:  $a \bmod (-b) = -((-a::int) \bmod b)$ 
by (cut-tac a = -a and b = b in zmod-zminus-zminus, auto)

lemma zdiv-zminus2-eq-if:
   $b \neq (0::int)$ 
   $\implies a \text{ div } (-b) =$ 
     $(\text{if } a \bmod b = 0 \text{ then } -(a \text{ div } b) \text{ else } -(a \text{ div } b) - 1)$ 
by (simp add: zdiv-zminus1-eq-if zdiv-zminus2)

lemma zmod-zminus2-eq-if:
   $a \bmod (-b::int) = (\text{if } a \bmod b = 0 \text{ then } 0 \text{ else } (a \bmod b) - b)$ 
by (simp add: zmod-zminus1-eq-if zmod-zminus2)

lemma zmod-zminus2-not-zero:
  fixes  $k \ l :: int$ 
  shows  $k \bmod -l \neq 0 \implies k \bmod l \neq 0$ 
  unfolding zmod-zminus2-eq-if by auto

```

### 31.7 Division of a Number by Itself

```

lemma self-quotient-aux1:  $[(0::int) < a; a = r + a*q; r < a] \implies 1 \leq q$ 
apply (subgoal-tac 0 < a*q)
apply (simp add: zero-less-mult-iff, arith)
done

lemma self-quotient-aux2:  $[(0::int) < a; a = r + a*q; 0 \leq r] \implies q \leq 1$ 
apply (subgoal-tac 0 \leq a*(1-q))
apply (simp add: zero-le-mult-iff)
apply (simp add: right-diff-distrib)

```

done

**lemma** *self-quotient*:  $[\mid \text{divmod-rel } a \ a \ (q, r); \ a \neq (0::\text{int}) \mid] \implies q = 1$   
**apply** (*simp add: split-ifs divmod-rel-def linorder-neq-iff*)  
**apply** (*rule order-antisym, safe, simp-all*)  
**apply** (*rule-tac [3] a = -a and r = -r in self-quotient-aux1*)  
**apply** (*rule-tac a = -a and r = -r in self-quotient-aux2*)  
**apply** (*force intro: self-quotient-aux1 self-quotient-aux2 simp add: add-commute*) +  
**done**

**lemma** *self-remainder*:  $[\mid \text{divmod-rel } a \ a \ (q, r); \ a \neq (0::\text{int}) \mid] \implies r = 0$   
**apply** (*frule self-quotient, assumption*)  
**apply** (*simp add: divmod-rel-def*)  
**done**

**lemma** *zdiv-self* [*simp*]:  $a \neq 0 \implies a \text{ div } a = (1::\text{int})$   
**by** (*simp add: divmod-rel-div-mod [THEN self-quotient]*)

**lemma** *zmod-self* [*simp*]:  $a \text{ mod } a = (0::\text{int})$   
**apply** (*case-tac a = 0, simp*)  
**apply** (*simp add: divmod-rel-div-mod [THEN self-remainder]*)  
**done**

### 31.8 Computation of Division and Remainder

**lemma** *zdiv-zero* [*simp*]:  $(0::\text{int}) \text{ div } b = 0$   
**by** (*simp add: div-def divmod-def*)

**lemma** *div-eq-minus1*:  $(0::\text{int}) < b \implies -1 \text{ div } b = -1$   
**by** (*simp add: div-def divmod-def*)

**lemma** *zmod-zero* [*simp*]:  $(0::\text{int}) \text{ mod } b = 0$   
**by** (*simp add: mod-def divmod-def*)

**lemma** *zmod-minus1*:  $(0::\text{int}) < b \implies -1 \text{ mod } b = b - 1$   
**by** (*simp add: mod-def divmod-def*)

a positive, b positive

**lemma** *div-pos-pos*:  $[\mid 0 < a; \ 0 \leq b \mid] \implies a \text{ div } b = \text{fst } (\text{posDivAlg } a \ b)$   
**by** (*simp add: div-def divmod-def*)

**lemma** *mod-pos-pos*:  $[\mid 0 < a; \ 0 \leq b \mid] \implies a \text{ mod } b = \text{snd } (\text{posDivAlg } a \ b)$   
**by** (*simp add: mod-def divmod-def*)

a negative, b positive

**lemma** *div-neg-pos*:  $[\mid a < 0; \ 0 < b \mid] \implies a \text{ div } b = \text{fst } (\text{negDivAlg } a \ b)$   
**by** (*simp add: div-def divmod-def*)

**lemma** *mod-neg-pos*:  $\llbracket a < 0; \ 0 < b \rrbracket \implies a \bmod b = \text{snd } (\text{negDivAlg } a \ b)$   
**by** (*simp add: mod-def divmod-def*)

a positive, b negative

**lemma** *div-pos-neg*:  
 $\llbracket 0 < a; \ b < 0 \rrbracket \implies a \text{ div } b = \text{fst } (\text{negateSnd } (\text{negDivAlg } (-a) \ (-b)))$   
**by** (*simp add: div-def divmod-def*)

**lemma** *mod-pos-neg*:  
 $\llbracket 0 < a; \ b < 0 \rrbracket \implies a \bmod b = \text{snd } (\text{negateSnd } (\text{negDivAlg } (-a) \ (-b)))$   
**by** (*simp add: mod-def divmod-def*)

a negative, b negative

**lemma** *div-neg-neg*:  
 $\llbracket a < 0; \ b \leq 0 \rrbracket \implies a \text{ div } b = \text{fst } (\text{negateSnd } (\text{posDivAlg } (-a) \ (-b)))$   
**by** (*simp add: div-def divmod-def*)

**lemma** *mod-neg-neg*:  
 $\llbracket a < 0; \ b \leq 0 \rrbracket \implies a \bmod b = \text{snd } (\text{negateSnd } (\text{posDivAlg } (-a) \ (-b)))$   
**by** (*simp add: mod-def divmod-def*)

Simplify expresions in which div and mod combine numerical constants

**lemma** *divmod-reII*:  
 $\llbracket a == b * q + r; \text{ if } 0 < b \text{ then } 0 \leq r \wedge r < b \text{ else } b < r \wedge r \leq 0 \rrbracket$   
 $\implies \text{divmod-rel } a \ b \ (q, r)$   
**unfolding** *divmod-rel-def* **by** *simp*

**lemmas** *divmod-rel-div-eq* = *divmod-reII* [*THEN* *divmod-rel-div*, *THEN* *eq-reflection*]

**lemmas** *divmod-rel-mod-eq* = *divmod-reII* [*THEN* *divmod-rel-mod*, *THEN* *eq-reflection*]

**lemmas** *arithmetic-simps* =

*arith-simps*

*add-special*

*OrderedGroup.add-0-left*

*OrderedGroup.add-0-right*

*mult-zero-left*

*mult-zero-right*

*mult-1-left*

*mult-1-right*

**ML**  $\ll$

*local*

*val* *mk-number* = *HOLogic.mk-number* *HOLogic.intT*;

*fun* *mk-cert* *u k l* =  $\text{@}\{\text{term plus} :: \text{int} \Rightarrow \text{int} \Rightarrow \text{int}\} \$$

$(\text{@}\{\text{term times} :: \text{int} \Rightarrow \text{int} \Rightarrow \text{int}\} \$ u \$ \text{mk-number } k) \$$

*mk-number l*;

*fun* *prove ctxt prop* = *Goal.prove ctxt*  $\llbracket \rrbracket$  *prop*

$(K (ALLGOALS (\text{full-simp-tac } (\text{HOL-basic-ss addsimps } \text{@}\{\text{thms arithmetic-simps}\}))));$

*fun* *binary-proc* *proc ss ct* =

```

    (case Thm.term-of ct of
      - $ t $ u =>
        (case try (pairself ('(snd o HOLogic.dest-number))) (t, u) of
          SOME args => proc (Simplifier.the-context ss) args
          | NONE => NONE)
      | - => NONE);
  in
    fun divmod-proc rule = binary-proc (fn ctxt => fn ((m, t), (n, u)) =>
      if n = 0 then NONE
      else let val (k, l) = Integer.div-mod m n;
            in SOME (rule OF [prove ctxt (Logic.mk-equals (t, mk-cert u k l))] end);
    end
  >>

```

```

simproc-setup binary-int-div (number-of m div number-of n :: int) =
  << K (divmod-proc (@{thm divmod-rel-div-eq})) >>

```

```

simproc-setup binary-int-mod (number-of m mod number-of n :: int) =
  << K (divmod-proc (@{thm divmod-rel-mod-eq})) >>

```

```

lemmas posDivAlg-eqn-number-of [simp] =
  posDivAlg-eqn [of number-of v number-of w, standard]

```

```

lemmas negDivAlg-eqn-number-of [simp] =
  negDivAlg-eqn [of number-of v number-of w, standard]

```

Special-case simplification

```

lemma zmod-minus1-right [simp]: a mod (-1::int) = 0
apply (cut-tac a = a and b = -1 in neg-mod-sign)
apply (cut-tac [2] a = a and b = -1 in neg-mod-bound)
apply (auto simp del: neg-mod-sign neg-mod-bound)
done

```

```

lemma zdiv-minus1-right [simp]: a div (-1::int) = -a
by (cut-tac a = a and b = -1 in zmod-zdiv-equality, auto)

```

```

lemmas div-pos-pos-1-number-of [simp] =
  div-pos-pos [OF int-0-less-1, of number-of w, standard]

```

```

lemmas div-pos-neg-1-number-of [simp] =
  div-pos-neg [OF int-0-less-1, of number-of w, standard]

```

```

lemmas mod-pos-pos-1-number-of [simp] =
  mod-pos-pos [OF int-0-less-1, of number-of w, standard]

```

```

lemmas mod-pos-neg-1-number-of [simp] =
  mod-pos-neg [OF int-0-less-1, of number-of w, standard]

```

```
lemmas posDivAlg-eqn-1-number-of [simp] =
  posDivAlg-eqn [of concl: 1 number-of w, standard]
```

```
lemmas negDivAlg-eqn-1-number-of [simp] =
  negDivAlg-eqn [of concl: 1 number-of w, standard]
```

### 31.9 Monotonicity in the First Argument (Dividend)

```
lemma zdiv-mono1: [| a ≤ a'; 0 < (b::int) |] ==> a div b ≤ a' div b
  apply (cut-tac a = a and b = b in zmod-zdiv-equality)
  apply (cut-tac a = a' and b = b in zmod-zdiv-equality)
  apply (rule unique-quotient-lemma)
  apply (erule subst)
  apply (erule subst, simp-all)
done
```

```
lemma zdiv-mono1-neg: [| a ≤ a'; (b::int) < 0 |] ==> a' div b ≤ a div b
  apply (cut-tac a = a and b = b in zmod-zdiv-equality)
  apply (cut-tac a = a' and b = b in zmod-zdiv-equality)
  apply (rule unique-quotient-lemma-neg)
  apply (erule subst)
  apply (erule subst, simp-all)
done
```

#### 31.10 Monotonicity in the Second Argument (Divisor)

```
lemma q-pos-lemma:
  [| 0 ≤ b*q' + r'; r' < b'; 0 < b' |] ==> 0 ≤ (q'::int)
  apply (subgoal-tac 0 < b'*(q' + 1) )
  apply (simp add: zero-less-mult-iff)
  apply (simp add: right-distrib)
done
```

```
lemma zdiv-mono2-lemma:
  [| b*q + r = b'*q' + r'; 0 ≤ b'*q' + r';
    r' < b'; 0 ≤ r; 0 < b'; b' ≤ b |]
  ==> q ≤ (q'::int)
  apply (frule q-pos-lemma, assumption+)
  apply (subgoal-tac b*q < b*(q' + 1) )
  apply (simp add: mult-less-cancel-left)
  apply (subgoal-tac b*q = r' - r + b'*q')
  prefer 2 apply simp
  apply (simp (no-asm-simp) add: right-distrib)
  apply (subst add-commute, rule zadd-zless-mono, arith)
  apply (rule mult-right-mono, auto)
done
```

```
lemma zdiv-mono2:
```

```

  [| (0::int) ≤ a; 0 < b'; b' ≤ b |] ==> a div b ≤ a div b'
apply (subgoal-tac b ≠ 0)
prefer 2 apply arith
apply (cut-tac a = a and b = b in zmod-zdiv-equality)
apply (cut-tac a = a and b = b' in zmod-zdiv-equality)
apply (rule zdiv-mono2-lemma)
apply (erule subst)
apply (erule subst, simp-all)
done

```

**lemma** *q-neg-lemma*:

```

  [| b'*q' + r' < 0; 0 ≤ r'; 0 < b' |] ==> q' ≤ (0::int)
apply (subgoal-tac b'*q' < 0)
apply (simp add: mult-less-0-iff, arith)
done

```

**lemma** *zdiv-mono2-neg-lemma*:

```

  [| b*q + r = b'*q' + r'; b'*q' + r' < 0;
    r < b; 0 ≤ r'; 0 < b'; b' ≤ b |]
  ==> q' ≤ (q::int)
apply (frule q-neg-lemma, assumption+)
apply (subgoal-tac b*q' < b*(q + 1) )
  apply (simp add: mult-less-cancel-left)
  apply (simp add: right-distrib)
apply (subgoal-tac b*q' ≤ b'*q')
  prefer 2 apply (simp add: mult-right-mono-neg, arith)
done

```

**lemma** *zdiv-mono2-neg*:

```

  [| a < (0::int); 0 < b'; b' ≤ b |] ==> a div b' ≤ a div b
apply (cut-tac a = a and b = b in zmod-zdiv-equality)
apply (cut-tac a = a and b = b' in zmod-zdiv-equality)
apply (rule zdiv-mono2-neg-lemma)
apply (erule subst)
apply (erule subst, simp-all)
done

```

### 31.11 More Algebraic Laws for div and mod

proving  $(a*b) \text{ div } c = a * (b \text{ div } c) + a * (b \text{ mod } c)$

**lemma** *zmult1-lemma*:

```

  [| divmod-rel b c (q, r); c ≠ 0 |]
  ==> divmod-rel (a * b) c (a*q + a*r div c, a*r mod c)
by (force simp add: split-ifs divmod-rel-def linorder-neq-iff right-distrib)

```

**lemma** *zdiv-zmult1-eq*:  $(a*b) \text{ div } c = a*(b \text{ div } c) + a*(b \text{ mod } c) \text{ div } (c::int)$

```

apply (case-tac c = 0, simp)
apply (blast intro: divmod-rel-div-mod [THEN zmult1-lemma, THEN divmod-rel-div])
done

```



```

lemma zmod-zmult1-eq:  $(a*b) \bmod c = a*(b \bmod c) \bmod (c::int)$ 
apply (case-tac  $c = 0$ , simp)
apply (blast intro: divmod-rel-div-mod [THEN zmult1-lemma, THEN divmod-rel-mod])
done

```

```

lemma zmod-zdiv-trivial:  $(a \bmod b) \operatorname{div} b = (0::int)$ 
apply (case-tac  $b = 0$ , simp)
apply (auto simp add: linorder-neq-iff div-pos-pos-trivial div-neg-neg-trivial)
done

```

proving  $(a+b) \operatorname{div} c = a \operatorname{div} c + b \operatorname{div} c + ((a \bmod c + b \bmod c) \operatorname{div} c)$

```

lemma zadd1-lemma:
  [| divmod-rel  $a \ c \ (aq, ar)$ ; divmod-rel  $b \ c \ (bq, br)$ ;  $c \neq 0$  |]
  ==> divmod-rel  $(a+b) \ c \ (aq + bq + (ar+br) \operatorname{div} c, (ar+br) \bmod c)$ 
by (force simp add: split-ifs divmod-rel-def linorder-neq-iff right-distrib)

```

```

lemma zdiv-zadd1-eq:
   $(a+b) \operatorname{div} (c::int) = a \operatorname{div} c + b \operatorname{div} c + ((a \bmod c + b \bmod c) \operatorname{div} c)$ 
apply (case-tac  $c = 0$ , simp)
apply (blast intro: zadd1-lemma [OF divmod-rel-div-mod divmod-rel-div-mod] divmod-rel-div)
done

```

**instance** *int* :: *ring-div*

**proof**

```

  fix  $a \ b \ c :: int$ 
  assume not0:  $b \neq 0$ 
  show  $(a + c * b) \operatorname{div} b = c + a \operatorname{div} b$ 
    unfolding zdiv-zadd1-eq [of  $a \ c * b$ ] using not0
    by (simp add: zmod-zmult1-eq zmod-zdiv-trivial zdiv-zmult1-eq)
qed auto

```

**lemma** *posDivAlg-div-mod*:

```

  assumes  $k \geq 0$ 
  and  $l \geq 0$ 
  shows  $\operatorname{posDivAlg} \ k \ l = (k \operatorname{div} l, k \bmod l)$ 
proof (cases  $l = 0$ )
  case True then show ?thesis by (simp add: posDivAlg.simps)
next
  case False with assms posDivAlg-correct
    have divmod-rel  $k \ l \ (fst \ (\operatorname{posDivAlg} \ k \ l), snd \ (\operatorname{posDivAlg} \ k \ l))$ 
    by simp
    from divmod-rel-div [OF this  $\langle l \neq 0 \rangle$ ] divmod-rel-mod [OF this  $\langle l \neq 0 \rangle$ ]
    show ?thesis by simp
qed

```

**lemma** *negDivAlg-div-mod*:

**assumes**  $k < 0$

```

    and  $l > 0$ 
    shows  $\text{negDivAlg } k \ l = (k \ \text{div } l, k \ \text{mod } l)$ 
  proof -
    from assms have  $l \neq 0$  by simp
    from assms negDivAlg-correct
      have  $\text{divmod-rel } k \ l \ (\text{fst } (\text{negDivAlg } k \ l), \text{snd } (\text{negDivAlg } k \ l))$ 
      by simp
    from divmod-rel-div [OF this  $\langle l \neq 0 \rangle$ ] divmod-rel-mod [OF this  $\langle l \neq 0 \rangle$ ]
    show ?thesis by simp
  qed

```

```

lemma zmod-eq-0-iff:  $(m \ \text{mod } d = 0) = (\text{EX } q::\text{int}. m = d*q)$ 
by (simp add: dvd-eq-mod-eq-0 [symmetric] dvd-def)

```

```

lemmas zmod-eq-0D [dest!] = zmod-eq-0-iff [THEN iffD1]

```

### 31.12 Proving $a \ \text{div } (b * c) = a \ \text{div } b \ \text{div } c$

first, four lemmas to bound the remainder for the cases  $b \nmid 0$  and  $b \mid 0$

```

lemma zmult2-lemma-aux1:  $[\mid (0::\text{int}) < c; \ b < r; \ r \leq 0 \mid] ==> b*c < b*(q \ \text{mod } c) + r$ 
apply (subgoal-tac  $b * (c - q \ \text{mod } c) < r * 1$ )
  apply (simp add: algebra-simps)
  apply (rule order-le-less-trans)
  apply (erule-tac [2] mult-strict-right-mono)
  apply (rule mult-left-mono-neg)
  using add1-zle-eq[of  $q \ \text{mod } c$ ] apply (simp add: algebra-simps pos-mod-bound)
  apply (simp)
  apply (simp)
done

```

```

lemma zmult2-lemma-aux2:
   $[\mid (0::\text{int}) < c; \ b < r; \ r \leq 0 \mid] ==> b * (q \ \text{mod } c) + r \leq 0$ 
apply (subgoal-tac  $b * (q \ \text{mod } c) \leq 0$ )
  apply arith
  apply (simp add: mult-le-0-iff)
done

```

```

lemma zmult2-lemma-aux3:  $[\mid (0::\text{int}) < c; \ 0 \leq r; \ r < b \mid] ==> 0 \leq b * (q \ \text{mod } c) + r$ 
apply (subgoal-tac  $0 \leq b * (q \ \text{mod } c)$ )
  apply arith
  apply (simp add: zero-le-mult-iff)
done

```

```

lemma zmult2-lemma-aux4:  $[\mid (0::\text{int}) < c; \ 0 \leq r; \ r < b \mid] ==> b * (q \ \text{mod } c) + r < b * c$ 
apply (subgoal-tac  $r * 1 < b * (c - q \ \text{mod } c)$ )

```

```

apply (simp add: right-diff-distrib)
apply (rule order-less-le-trans)
apply (erule mult-strict-right-mono)
apply (rule-tac [2] mult-left-mono)
apply simp
using add1-zle-eq[of q mod c] apply (simp add: algebra-simps pos-mod-bound)
apply simp
done

lemma zmult2-lemma: [| divmod-rel a b (q, r); b ≠ 0; 0 < c |]
  ==> divmod-rel a (b * c) (q div c, b*(q mod c) + r)
by (auto simp add: mult-ac divmod-rel-def linorder-neq-iff
  zero-less-mult-iff right-distrib [symmetric]
  zmult2-lemma-aux1 zmult2-lemma-aux2 zmult2-lemma-aux3
  zmult2-lemma-aux4)

lemma zdiv-zmult2-eq: (0::int) < c ==> a div (b*c) = (a div b) div c
apply (case-tac b = 0, simp)
apply (force simp add: divmod-rel-div-mod [THEN zmult2-lemma, THEN divmod-rel-div])
done

lemma zmod-zmult2-eq:
  (0::int) < c ==> a mod (b*c) = b*(a div b mod c) + a mod b
apply (case-tac b = 0, simp)
apply (force simp add: divmod-rel-div-mod [THEN zmult2-lemma, THEN divmod-rel-mod])
done

### 31.13 Cancellation of Common Factors in div

lemma zdiv-zmult-zmult1-aux1:
  [| (0::int) < b; c ≠ 0 |] ==> (c*a) div (c*b) = a div b
by (subst zdiv-zmult2-eq, auto)

lemma zdiv-zmult-zmult1-aux2:
  [| b < (0::int); c ≠ 0 |] ==> (c*a) div (c*b) = a div b
apply (subgoal-tac (c * (-a)) div (c * (-b)) = (-a) div (-b) )
apply (rule-tac [2] zdiv-zmult-zmult1-aux1, auto)
done

lemma zdiv-zmult-zmult1: c ≠ (0::int) ==> (c*a) div (c*b) = a div b
apply (case-tac b = 0, simp)
apply (auto simp add: linorder-neq-iff zdiv-zmult-zmult1-aux1 zdiv-zmult-zmult1-aux2)
done

lemma zdiv-zmult-zmult1-if[simp]:
  (k*m) div (k*n) = (if k = (0::int) then 0 else m div n)
by (simp add: zdiv-zmult-zmult1)

```

### 31.14 Distribution of Factors over mod

**lemma** *zmod-zmult-zmult1-aux1*:

$[[ (0::int) < b; \ c \neq 0 ]] \implies (c*a) \bmod (c*b) = c * (a \bmod b)$   
**by** (*subst zmod-zmult2-eq, auto*)

**lemma** *zmod-zmult-zmult1-aux2*:

$[[ b < (0::int); \ c \neq 0 ]] \implies (c*a) \bmod (c*b) = c * (a \bmod b)$   
**apply** (*subgoal-tac (c \* (-a)) mod (c \* (-b)) = c \* ((-a) mod (-b))*)  
**apply** (*rule-tac [2] zmod-zmult-zmult1-aux1, auto*)  
**done**

**lemma** *zmod-zmult-zmult1*:  $(c*a) \bmod (c*b) = (c::int) * (a \bmod b)$   
**apply** (*case-tac b = 0, simp*)  
**apply** (*case-tac c = 0, simp*)  
**apply** (*auto simp add: linorder-neq-iff zmod-zmult-zmult1-aux1 zmod-zmult-zmult1-aux2*)  
**done**

**lemma** *zmod-zmult-zmult2*:  $(a*c) \bmod (b*c) = (a \bmod b) * (c::int)$   
**apply** (*cut-tac c = c in zmod-zmult-zmult1*)  
**apply** (*auto simp add: mult-commute*)  
**done**

### 31.15 Splitting Rules for div and mod

The proofs of the two lemmas below are essentially identical

**lemma** *split-pos-lemma*:

$0 < k \implies$   
 $P(n \bmod k :: int)(n \bmod k) = (\forall i j. \ 0 \leq j \ \& \ j < k \ \& \ n = k*i + j \implies P \ i \ j)$   
**apply** (*rule iffI, clarify*)  
**apply** (*erule-tac P=P ?x ?y in rev-mp*)  
**apply** (*subst mod-add-eq*)  
**apply** (*subst zdiv-zadd1-eq*)  
**apply** (*simp add: div-pos-pos-trivial mod-pos-pos-trivial*)

converse direction

**apply** (*drule-tac x = n div k in spec*)  
**apply** (*drule-tac x = n mod k in spec, simp*)  
**done**

**lemma** *split-neg-lemma*:

$k < 0 \implies$   
 $P(n \bmod k :: int)(n \bmod k) = (\forall i j. \ k < j \ \& \ j \leq 0 \ \& \ n = k*i + j \implies P \ i \ j)$   
**apply** (*rule iffI, clarify*)  
**apply** (*erule-tac P=P ?x ?y in rev-mp*)  
**apply** (*subst mod-add-eq*)  
**apply** (*subst zdiv-zadd1-eq*)  
**apply** (*simp add: div-neg-neg-trivial mod-neg-neg-trivial*)

converse direction

```

apply (drule-tac  $x = n \text{ div } k$  in spec)
apply (drule-tac  $x = n \text{ mod } k$  in spec, simp)
done

```

```

lemma split-zdiv:
   $P(n \text{ div } k :: \text{int}) =$ 
     $((k = 0 \longrightarrow P\ 0) \ \&$ 
       $(0 < k \longrightarrow (\forall i\ j. 0 \leq j \ \& \ j < k \ \& \ n = k*i + j \longrightarrow P\ i)) \ \&$ 
       $(k < 0 \longrightarrow (\forall i\ j. k < j \ \& \ j \leq 0 \ \& \ n = k*i + j \longrightarrow P\ i)))$ 
apply (case-tac  $k=0$ , simp)
apply (simp only: linorder-neq-iff)
apply (erule disjE)
apply (simp-all add: split-pos-lemma [of concl:  $\%x\ y. P\ x$ ]
      split-neg-lemma [of concl:  $\%x\ y. P\ x$ ])
done

```

```

lemma split-zmod:
   $P(n \text{ mod } k :: \text{int}) =$ 
     $((k = 0 \longrightarrow P\ n) \ \&$ 
       $(0 < k \longrightarrow (\forall i\ j. 0 \leq j \ \& \ j < k \ \& \ n = k*i + j \longrightarrow P\ j)) \ \&$ 
       $(k < 0 \longrightarrow (\forall i\ j. k < j \ \& \ j \leq 0 \ \& \ n = k*i + j \longrightarrow P\ j)))$ 
apply (case-tac  $k=0$ , simp)
apply (simp only: linorder-neq-iff)
apply (erule disjE)
apply (simp-all add: split-pos-lemma [of concl:  $\%x\ y. P\ y$ ]
      split-neg-lemma [of concl:  $\%x\ y. P\ y$ ])
done

```

```

declare split-zdiv [of - - number-of k, simplified, standard, arith-split]
declare split-zmod [of - - number-of k, simplified, standard, arith-split]

```

### 31.16 Speeding up the Division Algorithm with Shifting

computing div by shifting

```

lemma pos-zdiv-mult-2:  $(0 :: \text{int}) \leq a \implies (1 + 2*b) \text{ div } (2*a) = b \text{ div } a$ 

```

**proof** *cases*

**assume**  $a=0$

**thus** *?thesis* **by** *simp*

**next**

**assume**  $a \neq 0$  **and**  $le\text{-}a: 0 \leq a$

**hence**  $a\text{-pos}: 1 \leq a$  **by** *arith*

**hence**  $one\text{-less-}a2: 1 < 2 * a$  **by** *arith*

**hence**  $le\text{-}2a: 2 * (1 + b \text{ mod } a) \leq 2 * a$

**unfolding** *mult-le-cancel-left*

**by** (*simp add: add1-zle-eq add-commute [of 1]*)

**with**  $a\text{-pos}$  **have**  $0 \leq b \text{ mod } a$  **by** *simp*

**hence**  $le\text{-}addm: 0 \leq 1 \text{ mod } (2*a) + 2*(b \text{ mod } a)$

**by** (*simp add: mod-pos-pos-trivial one-less-a2*)

```

with le-2a
have (1 mod (2*a) + 2*(b mod a)) div (2*a) = 0
  by (simp add: div-pos-pos-trivial le-addm mod-pos-pos-trivial one-less-a2
        right-distrib)
thus ?thesis
  by (subst zdiv-zadd1-eq,
        simp add: zdiv-zmult-zmult1 zmod-zmult-zmult1 one-less-a2
        div-pos-pos-trivial)
qed

lemma neg-zdiv-mult-2:  $a \leq (0::int) \implies (1 + 2*b) \text{ div } (2*a) = (b+1) \text{ div } a$ 
apply (subgoal-tac (1 + 2* (-b - 1)) div (2 * (-a)) = (-b - 1) div (-a) )
apply (rule-tac [2] pos-zdiv-mult-2)
apply (auto simp add: minus-mult-right [symmetric] right-diff-distrib)
apply (subgoal-tac (-1 - (2 * b)) = - (1 + (2 * b)))
apply (simp only: zdiv-zminus-zminus diff-minus minus-add-distrib [symmetric],
        simp)
done

lemma zdiv-number-of-Bit0 [simp]:
  number-of (Int.Bit0 v) div number-of (Int.Bit0 w) =
    number-of v div (number-of w :: int)
by (simp only: number-of-eq numeral-simps) simp

lemma zdiv-number-of-Bit1 [simp]:
  number-of (Int.Bit1 v) div number-of (Int.Bit0 w) =
    (if (0::int) ≤ number-of w
      then number-of v div (number-of w)
      else (number-of v + (1::int)) div (number-of w))
apply (simp only: number-of-eq numeral-simps UNIV-I split: split-if)
apply (simp add: zdiv-zmult-zmult1 pos-zdiv-mult-2 neg-zdiv-mult-2 add-ac)
done

```

### 31.17 Computing mod by Shifting (proofs resemble those for div)

```

lemma pos-zmod-mult-2:
   $(0::int) \leq a \implies (1 + 2*b) \text{ mod } (2*a) = 1 + 2 * (b \text{ mod } a)$ 
apply (case-tac a = 0, simp)
apply (subgoal-tac 1 < a * 2)
  prefer 2 apply arith
apply (subgoal-tac 2* (1 + b mod a) ≤ 2*a)
  apply (rule-tac [2] mult-left-mono)
apply (auto simp add: add-commute [of 1] mult-commute add1-zle-eq
        pos-mod-bound)
apply (subst mod-add-eq)
apply (simp add: zmod-zmult-zmult2 mod-pos-pos-trivial)
apply (rule mod-pos-pos-trivial)
apply (auto simp add: mod-pos-pos-trivial ring-distrib)

```

**apply** (*subgoal-tac*  $0 \leq b \bmod a$ , *arith*, *simp*)  
**done**

**lemma** *neg-zmod-mult-2*:

$a \leq (0::int) \implies (1 + 2*b) \bmod (2*a) = 2 * ((b+1) \bmod a) - 1$   
**apply** (*subgoal-tac*  $(1 + 2*(-b - 1)) \bmod (2*(-a)) =$   
 $1 + 2*(-b - 1) \bmod (-a))$ )  
**apply** (*rule-tac* [2] *pos-zmod-mult-2*)  
**apply** (*auto simp add: right-diff-distrib*)  
**apply** (*subgoal-tac*  $(-1 - (2 * b)) = -(1 + (2 * b))$ )  
**prefer** 2 **apply** *simp*  
**apply** (*simp only: zmod-zminus-zminus diff-minus minus-add-distrib [symmetric]*)  
**done**

**lemma** *zmod-number-of-Bit0 [simp]*:

$\text{number-of } (Int.Bit0\ v) \bmod \text{number-of } (Int.Bit0\ w) =$   
 $(2::int) * (\text{number-of } v \bmod \text{number-of } w)$   
**apply** (*simp only: number-of-eq numeral-simps*)  
**apply** (*simp add: zmod-zmult-zmult1 pos-zmod-mult-2*  
 $\text{neg-zmod-mult-2 add-ac}$ )  
**done**

**lemma** *zmod-number-of-Bit1 [simp]*:

$\text{number-of } (Int.Bit1\ v) \bmod \text{number-of } (Int.Bit0\ w) =$   
 $(\text{if } (0::int) \leq \text{number-of } w$   
 $\text{then } 2 * (\text{number-of } v \bmod \text{number-of } w) + 1$   
 $\text{else } 2 * ((\text{number-of } v + (1::int)) \bmod \text{number-of } w) - 1)$   
**apply** (*simp only: number-of-eq numeral-simps*)  
**apply** (*simp add: zmod-zmult-zmult1 pos-zmod-mult-2*  
 $\text{neg-zmod-mult-2 add-ac}$ )  
**done**

### 31.18 Quotients of Signs

**lemma** *div-neg-pos-less0*:  $[[\ a < (0::int); \ 0 < b \ ]] \implies a \text{ div } b < 0$   
**apply** (*subgoal-tac*  $a \text{ div } b \leq -1$ , *force*)  
**apply** (*rule order-trans*)  
**apply** (*rule-tac*  $a' = -1$  **in** *zdiv-mono1*)  
**apply** (*auto simp add: div-eq-minus1*)  
**done**

**lemma** *div-nonneg-neg-le0*:  $[[\ (0::int) \leq a; \ b < 0 \ ]] \implies a \text{ div } b \leq 0$   
**by** (*drule zdiv-mono1-neg, auto*)

**lemma** *div-nonpos-pos-le0*:  $[[\ (a::int) \leq 0; \ b > 0 \ ]] \implies a \text{ div } b \leq 0$   
**by** (*drule zdiv-mono1, auto*)

**lemma** *pos-imp-zdiv-nonneg-iff*:  $(0::int) < b \implies (0 \leq a \text{ div } b) = (0 \leq a)$   
**apply** *auto*

```

apply (drule-tac [2] zdiv-mono1)
apply (auto simp add: linorder-neq-iff)
apply (simp (no-asm-use) add: linorder-not-less [symmetric])
apply (blast intro: div-neg-pos-less0)
done

```

```

lemma neg-imp-zdiv-nonneg-iff:
   $b < (0::int) \implies (0 \leq a \text{ div } b) = (a \leq (0::int))$ 
apply (subst zdiv-zminus-zminus [symmetric])
apply (subst pos-imp-zdiv-nonneg-iff, auto)
done

```

```

lemma pos-imp-zdiv-neg-iff:  $(0::int) < b \implies (a \text{ div } b < 0) = (a < 0)$ 
by (simp add: linorder-not-le [symmetric] pos-imp-zdiv-nonneg-iff)

```

```

lemma neg-imp-zdiv-neg-iff:  $b < (0::int) \implies (a \text{ div } b < 0) = (0 < a)$ 
by (simp add: linorder-not-le [symmetric] neg-imp-zdiv-nonneg-iff)

```

### 31.19 The Divides Relation

```

lemmas zdvd-iff-zmod-eq-0-number-of [simp] =
  dvd-eq-mod-eq-0 [of number-of x::int number-of y::int, standard]

```

```

lemma zdvd-anti-sym:
   $0 < m \implies 0 < n \implies m \text{ dvd } n \implies n \text{ dvd } m \implies m = (n::int)$ 
apply (simp add: dvd-def, auto)
apply (simp add: mult-assoc zero-less-mult-iff zmult-eq-1-iff)
done

```

```

lemma zdvd-dvd-eq: assumes  $a \neq 0$  and  $(a::int) \text{ dvd } b$  and  $b \text{ dvd } a$ 
shows  $|a| = |b|$ 

```

**proof**—

```

from  $\langle a \text{ dvd } b \rangle$  obtain  $k$  where  $k:b = a*k$  unfolding dvd-def by blast
from  $\langle b \text{ dvd } a \rangle$  obtain  $k'$  where  $k':a = b*k'$  unfolding dvd-def by blast
from  $k \ k'$  have  $a = a*k*k'$  by simp
with mult-cancel-left1[where  $c=a$  and  $b=k*k'$ ]
have  $k*k':k*k' = 1$  using  $\langle a \neq 0 \rangle$  by (simp add: mult-assoc)
hence  $k = 1 \wedge k' = 1 \vee k = -1 \wedge k' = -1$  by (simp add: zmult-eq-1-iff)
thus ?thesis using  $k \ k'$  by auto

```

**qed**

```

lemma zdvd-zdiffD:  $k \text{ dvd } m - n \implies k \text{ dvd } n \implies k \text{ dvd } (m::int)$ 
apply (subgoal-tac  $m = n + (m - n)$ )
apply (erule ssubst)
apply (blast intro: dvd-add, simp)
done

```



```

lemma zdvd-reduce:  $(k \text{ dvd } n + k * m) = (k \text{ dvd } (n::int))$ 
apply (rule iffI)
apply (erule-tac [2] dvd-add)
apply (subgoal-tac  $n = (n + k * m) - k * m$ )
apply (erule ssubst)
apply (erule dvd-diff)
apply (simp-all)
done

lemma zdvd-zmod:  $f \text{ dvd } m ==> f \text{ dvd } (n::int) ==> f \text{ dvd } m \bmod n$ 
apply (simp add: dvd-def)
apply (auto simp add: zmod-zmult-zmult1)
done

lemma zdvd-zmod-imp-zdvd:  $k \text{ dvd } m \bmod n ==> k \text{ dvd } n ==> k \text{ dvd } (m::int)$ 
apply (subgoal-tac  $k \text{ dvd } n * (m \text{ div } n) + m \bmod n$ )
apply (simp add: zmod-zdiv-equality [symmetric])
apply (simp only: dvd-add dvd-mult2)
done

lemma zdvd-not-zless:  $0 < m ==> m < n ==> \neg n \text{ dvd } (m::int)$ 
apply (auto elim!: dvdE)
apply (subgoal-tac  $0 < n$ )
prefer 2
apply (blast intro: order-less-trans)
apply (simp add: zero-less-mult-iff)
apply (subgoal-tac  $n * k < n * 1$ )
apply (erule mult-less-cancel-left [THEN iffD1], auto)
done

lemma zmult-div-cancel:  $(n::int) * (m \text{ div } n) = m - (m \bmod n)$ 
using zmod-zdiv-equality[where  $a=m$  and  $b=n$ ]
by (simp add: algebra-simps)

lemma zdvd-mult-div-cancel:  $(n::int) \text{ dvd } m ==> n * (m \text{ div } n) = m$ 
apply (subgoal-tac  $m \bmod n = 0$ )
apply (simp add: zmult-div-cancel)
apply (simp only: dvd-eq-mod-eq-0)
done

lemma zdvd-mult-cancel: assumes  $d:k * m \text{ dvd } k * n$  and  $kz:k \neq (0::int)$ 
shows  $m \text{ dvd } n$ 
proof –
from  $d$  obtain  $h$  where  $h: k*n = k*m * h$  unfolding dvd-def by blast
{assume  $n \neq m*h$  hence  $k*n \neq k*(m*h)$  using  $kz$  by simp
with  $h$  have False by (simp add: mult-assoc)}
hence  $n = m * h$  by blast
thus ?thesis by simp
qed

```

```

theorem ex-nat:  $(\exists x::nat. P\ x) = (\exists x::int. 0 \leq x \wedge P\ (nat\ x))$ 
apply (simp split add: split-nat)
apply (rule iffI)
apply (erule exE)
apply (rule-tac x = int x in exI)
apply simp
apply (erule exE)
apply (rule-tac x = nat x in exI)
apply (erule conjE)
apply (erule-tac x = nat x in allE)
apply simp
done

theorem zdvd-int:  $(x\ dvd\ y) = (int\ x\ dvd\ int\ y)$ 
proof –
  have  $\bigwedge k. int\ y = int\ x * k \implies x\ dvd\ y$ 
  proof –
    fix k
    assume A:  $int\ y = int\ x * k$ 
    then show  $x\ dvd\ y$  proof (cases k)
      case (1 n) with A have  $y = x * n$  by (simp add: zmult-int)
      then show ?thesis ..
    next
      case (2 n) with A have  $int\ y = int\ x * (-\ int\ (Suc\ n))$  by simp
      also have  $\dots = -(int\ x * int\ (Suc\ n))$  by (simp only: mult-minus-right)
      also have  $\dots = -\ int\ (x * Suc\ n)$  by (simp only: zmult-int)
      finally have  $-\ int\ (x * Suc\ n) = int\ y$  ..
      then show ?thesis by (simp only: negative-eq-positive) auto
    qed
  qed
  then show ?thesis by (auto elim!: dvdE simp only: dvd-triv-left int-mult)
qed

lemma zdvd1-eq[simp]:  $(x::int)\ dvd\ 1 = (|x| = 1)$ 
proof
  assume d:  $x\ dvd\ 1$  hence  $int\ (nat\ |x|)\ dvd\ int\ (nat\ 1)$  by simp
  hence  $nat\ |x|\ dvd\ 1$  by (simp add: zdvd-int)
  hence  $nat\ |x| = 1$  by simp
  thus  $|x| = 1$  by (cases x < 0, auto)
next
  assume  $|x|=1$  thus  $x\ dvd\ 1$ 
  by(cases x < 0, simp-all add: minus-equation-iff dvd-eq-mod-eq-0)
qed
lemma zdvd-mult-cancel1:
  assumes  $mp:m \neq (0::int)$  shows  $(m * n\ dvd\ m) = (|n| = 1)$ 
proof
  assume n1:  $|n| = 1$  thus  $m * n\ dvd\ m$ 

```

```

  by (cases n > 0, auto simp add: minus-dvd-iff minus-equation-iff)
next
  assume H: m * n dvd m hence H2: m * n dvd m * 1 by simp
  from zdvd-mult-cancel[OF H2 mp] show |n| = 1 by (simp only: zdvd1-eq)
qed

```

```

lemma int-dvd-iff: (int m dvd z) = (m dvd nat (abs z))
  unfolding zdvd-int by (cases z ≥ 0) simp-all

```

```

lemma dvd-int-iff: (z dvd int m) = (nat (abs z) dvd m)
  unfolding zdvd-int by (cases z ≥ 0) simp-all

```

```

lemma nat-dvd-iff: (nat z dvd m) = (if 0 ≤ z then (z dvd int m) else m = 0)
  by (auto simp add: dvd-int-iff)

```

```

lemma zdvd-imp-le: [| z dvd n; 0 < n |] ==> z ≤ (n::int)
  apply (rule-tac z=n in int-cases)
  apply (auto simp add: dvd-int-iff)
  apply (rule-tac z=z in int-cases)
  apply (auto simp add: dvd-imp-le)
done

```

```

lemma zpower-zmod: ((x::int) mod m) ^ y mod m = x ^ y mod m
  apply (induct y, auto)
  apply (rule zmod-zmult1-eq [THEN trans])
  apply (simp (no-asm-simp))
  apply (rule mod-mult-eq [symmetric])
done

```

```

lemma zdiv-int: int (a div b) = (int a) div (int b)
  apply (subst split-div, auto)
  apply (subst split-zdiv, auto)
  apply (rule-tac a=int (b * i) + int j and b=int b and r=int j and r'=ja in
    IntDiv.unique-quotient)
  apply (auto simp add: IntDiv.divmod-rel-def of-nat-mult)
done

```

```

lemma zmod-int: int (a mod b) = (int a) mod (int b)
  apply (subst split-mod, auto)
  apply (subst split-zmod, auto)
  apply (rule-tac a=int (b * i) + int j and b=int b and q=int i and q'=ia
    in unique-remainder)
  apply (auto simp add: IntDiv.divmod-rel-def of-nat-mult)
done

```

```

lemma abs-div: (y::int) dvd x ==> abs (x div y) = abs x div abs y
  by (unfold dvd-def, cases y=0, auto simp add: abs-mult)

```

Suggested by Matthias Daum

```

lemma int-power-div-base:
   $\llbracket 0 < m; 0 < k \rrbracket \implies k \wedge m \text{ div } k = (k::\text{int}) \wedge (m - \text{Suc } 0)$ 
apply (subgoal-tac  $k \wedge m = k \wedge ((m - \text{Suc } 0) + \text{Suc } 0)$ )
apply (erule ssubst)
apply (simp only: power-add)
apply simp-all
done

by Brian Huffman

lemma zminus-zmod:  $-(x::\text{int}) \bmod m \bmod m = -x \bmod m$ 
by (rule mod-minus-eq [symmetric])

lemma zdiff-zmod-left:  $(x \bmod m - y) \bmod m = (x - y) \bmod (m::\text{int})$ 
by (rule mod-diff-left-eq [symmetric])

lemma zdiff-zmod-right:  $(x - y \bmod m) \bmod m = (x - y) \bmod (m::\text{int})$ 
by (rule mod-diff-right-eq [symmetric])

lemmas zmod-simps =
  mod-add-left-eq [symmetric]
  mod-add-right-eq [symmetric]
  IntDiv.zmod-zmult1-eq [symmetric]
  mod-mult-left-eq [symmetric]
  IntDiv.zpower-zmod
  zminus-zmod zdiff-zmod-left zdiff-zmod-right

```

Distributive laws for function *nat*.

```

lemma nat-div-distrib:  $0 \leq x \implies \text{nat } (x \text{ div } y) = \text{nat } x \text{ div } \text{nat } y$ 
apply (rule linorder-cases [of  $y \ 0$ ])
apply (simp add: div-nonneg-neg-le0)
apply simp
apply (simp add: nat-eq-iff pos-imp-zdiv-nonneg-iff zdiv-int)
done

```

```

lemma nat-mod-distrib:
   $\llbracket 0 \leq x; 0 \leq y \rrbracket \implies \text{nat } (x \bmod y) = \text{nat } x \bmod \text{nat } y$ 
apply (case-tac  $y = 0$ , simp add: DIVISION-BY-ZERO)
apply (simp add: nat-eq-iff zmod-int)
done

```

Suggested by Matthias Daum

```

lemma int-div-less-self:  $\llbracket 0 < x; 1 < k \rrbracket \implies x \text{ div } k < (x::\text{int})$ 
apply (subgoal-tac  $\text{nat } x \text{ div } \text{nat } k < \text{nat } x$ )
apply (simp (asm-lr) add: nat-div-distrib [symmetric])
apply (rule Divides.div-less-dividend, simp-all)
done

```

code generator setup

**context** *ring-1*  
**begin**

**lemma** *of-int-num* [code]:

*of-int*  $k = (\text{if } k = 0 \text{ then } 0 \text{ else if } k < 0 \text{ then}$   
    $- \text{of-int } (-k) \text{ else let}$   
      $(l, m) = \text{divmod } k \ 2;$   
      $l' = \text{of-int } l$   
    $\text{in if } m = 0 \text{ then } l' + l' \text{ else } l' + l' + 1)$

**proof** –

**have** *aux1*:  $k \bmod (2::\text{int}) \neq (0::\text{int}) \implies$   
 $\text{of-int } k = \text{of-int } (k \text{ div } 2 * 2 + 1)$

**proof** –

**have**  $k \bmod 2 < 2$  **by** (*auto intro: pos-mod-bound*)  
**moreover have**  $0 \leq k \bmod 2$  **by** (*auto intro: pos-mod-sign*)  
**moreover assume**  $k \bmod 2 \neq 0$   
**ultimately have**  $k \bmod 2 = 1$  **by** *arith*  
**moreover have**  $\text{of-int } k = \text{of-int } (k \text{ div } 2 * 2 + k \bmod 2)$  **by** *simp*  
**ultimately show** *?thesis* **by** *auto*

**qed**

**have** *aux2*:  $\bigwedge x. \text{of-int } 2 * x = x + x$

**proof** –

**fix**  $x$   
**have** *int2*:  $(2::\text{int}) = 1 + 1$  **by** *arith*  
**show**  $\text{of-int } 2 * x = x + x$   
**unfolding** *int2 of-int-add left-distrib* **by** *simp*

**qed**

**have** *aux3*:  $\bigwedge x. x * \text{of-int } 2 = x + x$

**proof** –

**fix**  $x$   
**have** *int2*:  $(2::\text{int}) = 1 + 1$  **by** *arith*  
**show**  $x * \text{of-int } 2 = x + x$   
**unfolding** *int2 of-int-add right-distrib* **by** *simp*

**qed**

**from** *aux1* **show** *?thesis* **by** (*auto simp add: divmod-mod-div Let-def aux2 aux3*)

**qed**

**end**

**lemma** *zmod-eq-dvd-iff*:  $(x::\text{int}) \bmod n = y \bmod n \longleftrightarrow n \text{ dvd } x - y$

**proof**

**assume**  $H: x \bmod n = y \bmod n$   
**hence**  $x \bmod n - y \bmod n = 0$  **by** *simp*  
**hence**  $(x \bmod n - y \bmod n) \bmod n = 0$  **by** *simp*  
**hence**  $(x - y) \bmod n = 0$  **by** (*simp add: mod-diff-eq[symmetric]*)  
**thus**  $n \text{ dvd } x - y$  **by** (*simp add: dvd-eq-mod-eq-0*)

**next**

**assume**  $H: n \text{ dvd } x - y$   
**then obtain**  $k$  **where**  $k: x - y = n * k$  **unfolding** *dvd-def* **by** *blast*

hence  $x = n*k + y$  **by** *simp*  
 hence  $x \bmod n = (n*k + y) \bmod n$  **by** *simp*  
 thus  $x \bmod n = y \bmod n$  **by** (*simp add: mod-add-left-eq*)  
**qed**

**lemma** *nat-mod-eq-lemma*: **assumes**  $xyn: (x::nat) \bmod n = y \bmod n$  **and**  $xy:y \leq x$   
**shows**  $\exists q. x = y + n * q$   
**proof** –  
 from  $xy$  **have**  $th: int\ x - int\ y = int\ (x - y)$  **by** *simp*  
 from  $xyn$  **have**  $int\ x \bmod int\ n = int\ y \bmod int\ n$   
**by** (*simp add: zmod-int[symmetric]*)  
 hence  $int\ n \text{ dvd } int\ x - int\ y$  **by** (*simp only: zmod-eq-dvd-iff[symmetric]*)  
 hence  $n \text{ dvd } x - y$  **by** (*simp add: th zdvd-int*)  
 then show *?thesis* **using**  $xy$  **unfolding** *dvd-def* **apply** *clarsimp* **apply** (*rule-tac*  $x=k$  **in** *exI*) **by** *arith*  
**qed**

**lemma** *nat-mod-eq-iff*:  $(x::nat) \bmod n = y \bmod n \longleftrightarrow (\exists q1\ q2. x + n * q1 = y + n * q2)$   
 (**is** *?lhs = ?rhs*)  
**proof**  
 assume  $H: x \bmod n = y \bmod n$   
 {**assume**  $xy: x \leq y$   
 from  $H$  **have**  $th: y \bmod n = x \bmod n$  **by** *simp*  
 from *nat-mod-eq-lemma*[*OF th xy*] **have** *?rhs*  
**apply** *clarify* **apply** (*rule-tac*  $x=q$  **in** *exI*) **by** (*rule exI[where x=0], simp*)}  
 moreover  
 {**assume**  $xy: y \leq x$   
 from *nat-mod-eq-lemma*[*OF H xy*] **have** *?rhs*  
**apply** *clarify* **apply** (*rule-tac*  $x=0$  **in** *exI*) **by** (*rule-tac*  $x=q$  **in** *exI*, *simp*)}  
 ultimately **show** *?rhs* **using** *linear[of x y]* **by** *blast*  
**next**  
 assume *?rhs* **then obtain**  $q1\ q2$  **where**  $q12: x + n * q1 = y + n * q2$  **by** *blast*  
 hence  $(x + n * q1) \bmod n = (y + n * q2) \bmod n$  **by** *simp*  
 thus *?lhs* **by** *simp*  
**qed**

### 31.20 Simproc setup

**use** *Tools/int-factor-simprocs.ML*

### 31.21 Code generation

**definition** *pdivmod* ::  $int \Rightarrow int \Rightarrow int \times int$  **where**  
 $pdivmod\ k\ l = (|k| \text{ div } |l|, |k| \bmod |l|)$

**lemma** *pdivmod-posDivAlg* [*code*]:  
 $pdivmod\ k\ l = (if\ l = 0\ then\ (0, |k|)\ else\ posDivAlg\ |k|\ |l|)$   
**by** (*subst posDivAlg-div-mod*) (*simp-all add: pdivmod-def*)

```

lemma divmod-pdivmod:  $\text{divmod } k \ l = (\text{if } k = 0 \text{ then } (0, 0) \text{ else if } l = 0 \text{ then } (0, k) \text{ else}$ 
   $\text{apsnd } ((\text{op } *) (\text{sgn } l)) (\text{if } 0 < l \wedge 0 \leq k \vee l < 0 \wedge k < 0$ 
     $\text{then pdivmod } k \ l$ 
     $\text{else (let } (r, s) = \text{pdivmod } k \ l \text{ in}$ 
       $\text{if } s = 0 \text{ then } (-r, 0) \text{ else } (-r - 1, |l| - s)))$ 
proof –
  have aux:  $\bigwedge q::\text{int. } -k = l * q \longleftrightarrow k = l * -q$  by auto
  show ?thesis
    by (simp add: divmod-mod-div pdivmod-def)
      (auto simp add: aux not-less not-le zdiv-zminus1-eq-if
        zmod-zminus1-eq-if zdiv-zminus2-eq-if zmod-zminus2-eq-if)
qed

```

```

lemma divmod-code [code]:  $\text{divmod } k \ l = (\text{if } k = 0 \text{ then } (0, 0) \text{ else if } l = 0 \text{ then}$ 
   $(0, k) \text{ else}$ 
   $\text{apsnd } ((\text{op } *) (\text{sgn } l)) (\text{if } \text{sgn } k = \text{sgn } l$ 
     $\text{then pdivmod } k \ l$ 
     $\text{else (let } (r, s) = \text{pdivmod } k \ l \text{ in}$ 
       $\text{if } s = 0 \text{ then } (-r, 0) \text{ else } (-r - 1, |l| - s)))$ 
proof –
  have  $k \neq 0 \implies l \neq 0 \implies 0 < l \wedge 0 \leq k \vee l < 0 \wedge k < 0 \longleftrightarrow \text{sgn } k = \text{sgn } l$ 
    by (auto simp add: not-less sgn-if)
  then show ?thesis by (simp add: divmod-pdivmod)
qed

```

**code-modulename** *SML*

*IntDiv Integer*

**code-modulename** *OCaml*

*IntDiv Integer*

**code-modulename** *Haskell*

*IntDiv Integer*

**end**

## 32 NatBin: Binary arithmetic for the natural numbers

```

theory NatBin
imports IntDiv
uses (Tools/nat-simprocs.ML)
begin

```

Arithmetic for naturals is reduced to that for the non-negative integers.

**instantiation** *nat* :: *number*  
**begin**

**definition**  
*nat-number-of-def* [*code inline*, *code del*]: *number-of* *v* = *nat* (*number-of* *v*)

**instance** ..

**end**

**lemma** [*code post*]:  
*nat* (*number-of* *v*) = *number-of* *v*  
**unfolding** *nat-number-of-def* ..

**abbreviation** (*xsymbols*)  
*power2* :: 'a::power => 'a ((<sup>-2</sup>) [1000] 999) **where**  
 $x^2 == x^{\wedge}2$

**notation** (*latex output*)  
*power2* ((<sup>-2</sup>) [1000] 999)

**notation** (*HTML output*)  
*power2* ((<sup>-2</sup>) [1000] 999)

### 32.1 Predicate for negative binary numbers

**definition** *neg* :: *int* => *bool* **where**  
*neg* *Z* <math>\longleftrightarrow Z < 0</math>

**lemma** *not-neg-int* [*simp*]:  $\sim \text{neg } (\text{of-nat } n)$   
**by** (*simp add: neg-def*)

**lemma** *neg-zminus-int* [*simp*]: *neg* ( $- (\text{of-nat } (\text{Suc } n))$ )  
**by** (*simp add: neg-def neg-less-0-iff-less del: of-nat-Suc*)

**lemmas** *neg-eq-less-0* = *neg-def*

**lemma** *not-neg-eq-ge-0*:  $(\sim \text{neg } x) = (0 \leq x)$   
**by** (*simp add: neg-def linorder-not-less*)

To simplify inequalities when *Numerall* can get simplified to 1

**lemma** *not-neg-0*:  $\sim \text{neg } 0$   
**by** (*simp add: One-int-def neg-def*)

**lemma** *not-neg-1*:  $\sim \text{neg } 1$   
**by** (*simp add: neg-def linorder-not-less zero-le-one*)

**lemma** *neg-nat*: *neg* *z* ==> *nat* *z* = 0  
**by** (*simp add: neg-def order-less-imp-le*)



**lemma** *not-neg-nat*:  $\sim \text{neg } z \implies \text{of-nat } (\text{nat } z) = z$   
**by** (*simp add: linorder-not-less neg-def*)

If *Numeral0* is rewritten to 0 then this rule can’t be applied: *Numeral0* IS *number-of Pls*

**lemma** *not-neg-number-of-Pls*:  $\sim \text{neg } (\text{number-of Int.Pl})$   
**by** (*simp add: neg-def*)

**lemma** *neg-number-of-Min*:  $\text{neg } (\text{number-of Int.Min})$   
**by** (*simp add: neg-def*)

**lemma** *neg-number-of-Bit0*:  
 $\text{neg } (\text{number-of } (\text{Int.Bit0 } w)) = \text{neg } (\text{number-of } w)$   
**by** (*simp add: neg-def*)

**lemma** *neg-number-of-Bit1*:  
 $\text{neg } (\text{number-of } (\text{Int.Bit1 } w)) = \text{neg } (\text{number-of } w)$   
**by** (*simp add: neg-def*)

**lemmas** *neg-simps* [*simp*] =  
*not-neg-0 not-neg-1*  
*not-neg-number-of-Pls neg-number-of-Min*  
*neg-number-of-Bit0 neg-number-of-Bit1*

### 32.2 Function *nat*: Coercion from Type *int* to *nat*

**declare** *nat-0* [*simp*] *nat-1* [*simp*]

**lemma** *nat-number-of* [*simp*]:  $\text{nat } (\text{number-of } w) = \text{number-of } w$   
**by** (*simp add: nat-number-of-def*)

**lemma** *nat-numeral-0-eq-0* [*simp*]: *Numeral0* = (0::nat)  
**by** (*simp add: nat-number-of-def*)

**lemma** *nat-numeral-1-eq-1* [*simp*]: *Numeral1* = (1::nat)  
**by** (*simp add: nat-1 nat-number-of-def*)

**lemma** *numeral-1-eq-Suc-0*: *Numeral1* = *Suc 0*  
**by** (*simp add: nat-numeral-1-eq-1*)

**lemma** *numeral-2-eq-2*: 2 = *Suc (Suc 0)*  
**apply** (*unfold nat-number-of-def*)  
**apply** (*rule nat-2*)  
**done**

### 32.3 Function *int*: Coercion from Type *nat* to *int*

**lemma** *int-nat-number-of* [*simp*]:

```

    int (number-of v) =
      (if neg (number-of v :: int) then 0
       else (number-of v :: int))
  unfolding nat-number-of-def number-of-is-id neg-def
  by simp

```

### 32.3.1 Successor

```

lemma Suc-nat-eq-nat-zadd1: (0::int) <= z ==> Suc (nat z) = nat (1 + z)
apply (rule sym)
apply (simp add: nat-eq-iff int-Suc)
done

```

```

lemma Suc-nat-number-of-add:
  Suc (number-of v + n) =
    (if neg (number-of v :: int) then 1+n else number-of (Int.succ v) + n)
  unfolding nat-number-of-def number-of-is-id neg-def numeral-simps
  by (simp add: Suc-nat-eq-nat-zadd1 add-ac)

```

```

lemma Suc-nat-number-of [simp]:
  Suc (number-of v) =
    (if neg (number-of v :: int) then 1 else number-of (Int.succ v))
  apply (cut-tac n = 0 in Suc-nat-number-of-add)
  apply (simp cong del: if-weak-cong)
  done

```

### 32.3.2 Addition

```

lemma add-nat-number-of [simp]:
  (number-of v :: nat) + number-of v' =
    (if v < Int.Pls then number-of v'
     else if v' < Int.Pls then number-of v
     else number-of (v + v'))
  unfolding nat-number-of-def number-of-is-id numeral-simps
  by (simp add: nat-add-distrib)

```

```

lemma nat-number-of-add-1 [simp]:
  number-of v + (1::nat) =
    (if v < Int.Pls then 1 else number-of (Int.succ v))
  unfolding nat-number-of-def number-of-is-id numeral-simps
  by (simp add: nat-add-distrib)

```

```

lemma nat-1-add-number-of [simp]:
  (1::nat) + number-of v =
    (if v < Int.Pls then 1 else number-of (Int.succ v))
  unfolding nat-number-of-def number-of-is-id numeral-simps
  by (simp add: nat-add-distrib)

```

```

lemma nat-1-add-1 [simp]: 1 + 1 = (2::nat)
  by (rule int-int-eq [THEN iffD1]) simp

```

### 32.3.3 Subtraction

**lemma** *diff-nat-eq-if*:

$\text{nat } z - \text{nat } z' =$   
 (if *neg*  $z'$  then  $\text{nat } z$   
 else let  $d = z - z'$  in  
 if *neg*  $d$  then 0 else  $\text{nat } d$ )

**by** (*simp add: Let-def nat-diff-distrib [symmetric] neg-eq-less-0 not-neg-eq-ge-0*)

**lemma** *diff-nat-number-of [simp]*:

$(\text{number-of } v :: \text{nat}) - \text{number-of } v' =$   
 (if  $v' < \text{Int.Pls}$  then  $\text{number-of } v$   
 else let  $d = \text{number-of } (v + \text{uminus } v')$  in  
 if *neg*  $d$  then 0 else  $\text{nat } d$ )

**unfolding** *nat-number-of-def number-of-is-id numeral-simps neg-def*

**by** *auto*

**lemma** *nat-number-of-diff-1 [simp]*:

$\text{number-of } v - (1 :: \text{nat}) =$   
 (if  $v \leq \text{Int.Pls}$  then 0 else  $\text{number-of } (\text{Int.pred } v)$ )

**unfolding** *nat-number-of-def number-of-is-id numeral-simps*

**by** *auto*

### 32.3.4 Multiplication

**lemma** *mult-nat-number-of [simp]*:

$(\text{number-of } v :: \text{nat}) * \text{number-of } v' =$   
 (if  $v < \text{Int.Pls}$  then 0 else  $\text{number-of } (v * v')$ )

**unfolding** *nat-number-of-def number-of-is-id numeral-simps*

**by** (*simp add: nat-mult-distrib*)

### 32.3.5 Quotient

**lemma** *div-nat-number-of [simp]*:

$(\text{number-of } v :: \text{nat}) \text{ div } \text{number-of } v' =$   
 (if *neg*  $(\text{number-of } v :: \text{int})$  then 0  
 else  $\text{nat } (\text{number-of } v \text{ div } \text{number-of } v')$ )

**unfolding** *nat-number-of-def number-of-is-id neg-def*

**by** (*simp add: nat-div-distrib*)

**lemma** *one-div-nat-number-of [simp]*:

$\text{Suc } 0 \text{ div } \text{number-of } v' = \text{nat } (1 \text{ div } \text{number-of } v')$

**by** (*simp del: nat-numeral-1-eq-1 add: numeral-1-eq-Suc-0 [symmetric]*)

### 32.3.6 Remainder

**lemma** *mod-nat-number-of [simp]*:

$(\text{number-of } v :: \text{nat}) \text{ mod } \text{number-of } v' =$   
 (if *neg*  $(\text{number-of } v :: \text{int})$  then 0

```

      else if neg (number-of v' :: int) then number-of v
      else nat (number-of v mod number-of v')
unfolding nat-number-of-def number-of-is-id neg-def
by (simp add: nat-mod-distrib)

```

```

lemma one-mod-nat-number-of [simp]:
  Suc 0 mod number-of v' =
    (if neg (number-of v' :: int) then Suc 0
     else nat (1 mod number-of v'))
by (simp del: nat-numeral-1-eq-1 add: numeral-1-eq-Suc-0 [symmetric])

```

### 32.3.7 Divisibility

```

lemmas dvd-eq-mod-eq-0-number-of =
  dvd-eq-mod-eq-0 [of number-of x number-of y, standard]

```

```

declare dvd-eq-mod-eq-0-number-of [simp]

```

**ML**

```

⟦
  val nat-number-of-def = thmnat-number-of-def;

  val nat-number-of = thmnat-number-of;
  val nat-numeral-0-eq-0 = thmnat-numeral-0-eq-0;
  val nat-numeral-1-eq-1 = thmnat-numeral-1-eq-1;
  val numeral-1-eq-Suc-0 = thmnumeral-1-eq-Suc-0;
  val numeral-2-eq-2 = thmnumeral-2-eq-2;
  val nat-div-distrib = thmnat-div-distrib;
  val nat-mod-distrib = thmnat-mod-distrib;
  val int-nat-number-of = thmint-nat-number-of;
  val Suc-nat-eq-nat-zadd1 = thmSuc-nat-eq-nat-zadd1;
  val Suc-nat-number-of-add = thmSuc-nat-number-of-add;
  val Suc-nat-number-of = thmSuc-nat-number-of;
  val add-nat-number-of = thmadd-nat-number-of;
  val diff-nat-eq-if = thmdiff-nat-eq-if;
  val diff-nat-number-of = thmdiff-nat-number-of;
  val mult-nat-number-of = thmmult-nat-number-of;
  val div-nat-number-of = thmdiv-nat-number-of;
  val mod-nat-number-of = thmmod-nat-number-of;
⟧

```

## 32.4 Comparisons

### 32.4.1 Equals (=)

```

lemma eq-nat-nat-iff:
  [| (0::int) <= z; 0 <= z' |] ==> (nat z = nat z') = (z=z')
by (auto elim!: nonneg-eq-int)

```

```

lemma eq-nat-number-of [simp]:

```

```

((number-of v :: nat) = number-of v') =
  (if neg (number-of v :: int) then (number-of v' :: int) ≤ 0
   else if neg (number-of v' :: int) then (number-of v :: int) = 0
   else v = v')
unfolding nat-number-of-def number-of-is-id neg-def
by auto

```

### 32.4.2 Less-than (i)

```

lemma less-nat-number-of [simp]:
  (number-of v :: nat) < number-of v'  $\longleftrightarrow$ 
  (if v < v' then Int.Pls < v' else False)
unfolding nat-number-of-def number-of-is-id numeral-simps
by auto

```

### 32.4.3 Less-than-or-equal

```

lemma le-nat-number-of [simp]:
  (number-of v :: nat) ≤ number-of v'  $\longleftrightarrow$ 
  (if v ≤ v' then True else v ≤ Int.Pls)
unfolding nat-number-of-def number-of-is-id numeral-simps
by auto

```

**lemmas** numerals = nat-numeral-0-eq-0 nat-numeral-1-eq-1 numeral-2-eq-2

## 32.5 Powers with Numeric Exponents

We cannot refer to the number  $2::'a$  in *Ring-and-Field.thy*. We cannot prove general results about the numeral  $-1::'a$ , so we have to use  $-(1::'a)$  instead.

```

lemma power2-eq-square: (a::'a::recpower)2 = a * a
by (simp add: numeral-2-eq-2 Power.power-Suc)

```

```

lemma zero-power2 [simp]: (0::'a::{semiring-1,recpower})2 = 0
by (simp add: power2-eq-square)

```

```

lemma one-power2 [simp]: (1::'a::{semiring-1,recpower})2 = 1
by (simp add: power2-eq-square)

```

```

lemma power3-eq-cube: (x::'a::recpower) ^ 3 = x * x * x
apply (subgoal-tac 3 = Suc (Suc (Suc 0)))
apply (erule ssubst)
apply (simp add: power-Suc mult-ac)
apply (unfold nat-number-of-def)
apply (subst nat-eq-iff)
apply simp
done

```

Squares of literal numerals will be evaluated.

**lemmas** *power2-eq-square-number-of* =  
           *power2-eq-square* [of number-of *w*, standard]  
**declare** *power2-eq-square-number-of* [simp]

**lemma** *zero-le-power2*[simp]:  $0 \leq (a^2 :: 'a :: \{\text{ordered-idom}, \text{recpower}\})$   
   **by** (simp add: *power2-eq-square*)

**lemma** *zero-less-power2*[simp]:  
    $(0 < a^2) = (a \neq (0 :: 'a :: \{\text{ordered-idom}, \text{recpower}\}))$   
   **by** (force simp add: *power2-eq-square* *zero-less-mult-iff* *linorder-neq-iff*)

**lemma** *power2-less-0*[simp]:  
   **fixes**  $a :: 'a :: \{\text{ordered-idom}, \text{recpower}\}$   
   **shows**  $\sim (a^2 < 0)$   
**by** (force simp add: *power2-eq-square* *mult-less-0-iff*)

**lemma** *zero-eq-power2*[simp]:  
    $(a^2 = 0) = (a = (0 :: 'a :: \{\text{ordered-idom}, \text{recpower}\}))$   
   **by** (force simp add: *power2-eq-square* *mult-eq-0-iff*)

**lemma** *abs-power2*[simp]:  
    $\text{abs}(a^2) = (a^2 :: 'a :: \{\text{ordered-idom}, \text{recpower}\})$   
   **by** (simp add: *power2-eq-square* *abs-mult* *abs-mult-self*)

**lemma** *power2-abs*[simp]:  
    $(\text{abs } a)^2 = (a^2 :: 'a :: \{\text{ordered-idom}, \text{recpower}\})$   
   **by** (simp add: *power2-eq-square* *abs-mult-self*)

**lemma** *power2-minus*[simp]:  
    $(- a)^2 = (a^2 :: 'a :: \{\text{comm-ring-1}, \text{recpower}\})$   
   **by** (simp add: *power2-eq-square*)

**lemma** *power2-le-imp-le*:  
   **fixes**  $x y :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}$   
   **shows**  $\llbracket x^2 \leq y^2; 0 \leq y \rrbracket \implies x \leq y$   
**unfolding** *numeral-2-eq-2* **by** (rule *power-le-imp-le-base*)

**lemma** *power2-less-imp-less*:  
   **fixes**  $x y :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}$   
   **shows**  $\llbracket x^2 < y^2; 0 \leq y \rrbracket \implies x < y$   
**by** (rule *power-less-imp-less-base*)

**lemma** *power2-eq-imp-eq*:  
   **fixes**  $x y :: 'a :: \{\text{ordered-semidom}, \text{recpower}\}$   
   **shows**  $\llbracket x^2 = y^2; 0 \leq x; 0 \leq y \rrbracket \implies x = y$   
**unfolding** *numeral-2-eq-2* **by** (erule (2) *power-eq-imp-eq-base*, simp)

**lemma** *power-minus1-even*[simp]:  $(-1) ^ (2*n) = (1::'a::\{comm-ring-1,recpower\})$   
**proof** (*induct n*)  
 case 0 **show** ?case **by** *simp*  
**next**  
 case (*Suc n*) **then show** ?case **by** (*simp add: power-Suc power-add*)  
**qed**

**lemma** *power-minus1-odd*:  $(-1) ^ Suc(2*n) = -(1::'a::\{comm-ring-1,recpower\})$   
**by** (*simp add: power-Suc*)

**lemma** *power-even-eq*:  $(a::'a::recpower) ^ (2*n) = (a ^ n) ^ 2$   
**by** (*subst mult-commute*) (*simp add: power-mult*)

**lemma** *power-odd-eq*:  $(a::int) ^ Suc(2*n) = a * (a ^ n) ^ 2$   
**by** (*simp add: power-even-eq*)

**lemma** *power-minus-even* [simp]:  
 $(-a) ^ (2*n) = (a::'a::\{comm-ring-1,recpower\}) ^ (2*n)$   
**by** (*simp add: power-minus1-even power-minus [of a]*)

**lemma** *zero-le-even-power'*[simp]:  
 $0 \leq (a::'a::\{ordered-idom,recpower\}) ^ (2*n)$   
**proof** (*induct n*)  
 case 0  
 show ?case **by** (*simp add: zero-le-one*)  
**next**  
 case (*Suc n*)  
 have  $a ^ (2 * Suc n) = (a*a) * a ^ (2*n)$   
 by (*simp add: mult-ac power-add power2-eq-square*)  
 thus ?case  
 by (*simp add: prems zero-le-mult-iff*)  
**qed**

**lemma** *odd-power-less-zero*:  
 $(a::'a::\{ordered-idom,recpower\}) < 0 ==> a ^ Suc(2*n) < 0$   
**proof** (*induct n*)  
 case 0  
 then show ?case **by** *simp*  
**next**  
 case (*Suc n*)  
 have  $a ^ Suc (2 * Suc n) = (a*a) * a ^ Suc(2*n)$   
 by (*simp add: mult-ac power-add power2-eq-square*)  
 thus ?case  
 by (*simp del: power-Suc add: prems mult-less-0-iff mult-neg-neg*)  
**qed**

**lemma** *odd-0-le-power-imp-0-le*:  
 $0 \leq a ^ Suc(2*n) ==> 0 \leq (a::'a::\{ordered-idom,recpower\})$   
**apply** (*insert odd-power-less-zero [of a n]*)

**apply** (*force simp add: linorder-not-less [symmetric]*)  
**done**

Simprules for comparisons where common factors can be cancelled.

**lemmas** *zero-compare-simps* =  
*add-strict-increasing add-strict-increasing2 add-increasing*  
*zero-le-mult-iff zero-le-divide-iff*  
*zero-less-mult-iff zero-less-divide-iff*  
*mult-le-0-iff divide-le-0-iff*  
*mult-less-0-iff divide-less-0-iff*  
*zero-le-power2 power2-less-0*

### 32.5.1 Nat

**lemma** *Suc-pred'*:  $0 < n \implies n = \text{Suc}(n - 1)$   
**by** (*simp add: numerals*)

**lemmas** *expand-Suc* = *Suc-pred'* [*of number-of v, standard*]

### 32.5.2 Arith

**lemma** *Suc-eq-add-numeral-1*:  $\text{Suc } n = n + 1$   
**by** (*simp add: numerals*)

**lemma** *Suc-eq-add-numeral-1-left*:  $\text{Suc } n = 1 + n$   
**by** (*simp add: numerals*)

**lemma** *add-eq-if*:  $(m::\text{nat}) + n = (\text{if } m=0 \text{ then } n \text{ else } \text{Suc } ((m - 1) + n))$   
**unfolding** *One-nat-def* **by** (*cases m*) *simp-all*

**lemma** *mult-eq-if*:  $(m::\text{nat}) * n = (\text{if } m=0 \text{ then } 0 \text{ else } n + ((m - 1) * n))$   
**unfolding** *One-nat-def* **by** (*cases m*) *simp-all*

**lemma** *power-eq-if*:  $(p \wedge m::\text{nat}) = (\text{if } m=0 \text{ then } 1 \text{ else } p * (p \wedge (m - 1)))$   
**unfolding** *One-nat-def* **by** (*cases m*) *simp-all*

## 32.6 Comparisons involving (0::nat)

Simplification already does  $n < (0::'a)$ ,  $n \leq (0::'a)$  and  $(0::'a) \leq n$ .

**lemma** *eq-number-of-0* [*simp*]:  
 $\text{number-of } v = (0::\text{nat}) \iff v \leq \text{Int.Pl}$   
**unfolding** *nat-number-of-def number-of-is-id numeral-simps*  
**by** *auto*

**lemma** *eq-0-number-of* [*simp*]:  
 $(0::\text{nat}) = \text{number-of } v \iff v \leq \text{Int.Pl}$



**by** (rule trans [OF eq-sym-conv eq-number-of-0])

**lemma** *less-0-number-of* [simp]:  
 $(0 :: \text{nat}) < \text{number-of } v \iff \text{Int.Pl} < v$   
**unfolding** *nat-number-of-def number-of-is-id numeral-simps*  
**by** *simp*

**lemma** *neg-imp-number-of-eq-0*:  $\text{neg } (\text{number-of } v :: \text{int}) \implies \text{number-of } v = (0 :: \text{nat})$   
**by** (simp del: *nat-numeral-0-eq-0 add: nat-numeral-0-eq-0 [symmetric]*)

### 32.7 Comparisons involving *Suc*

**lemma** *eq-number-of-Suc* [simp]:  
 $(\text{number-of } v = \text{Suc } n) =$   
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$   
 $\text{if neg } pv \text{ then False else nat } pv = n)$   
**apply** (simp only: *simp-thms Let-def neg-eq-less-0 linorder-not-less*  
*number-of-pred nat-number-of-def*  
*split add: split-if*)  
**apply** (rule-tac  $x = \text{number-of } v$  **in** *spec*)  
**apply** (auto simp add: *nat-eq-iff*)  
**done**

**lemma** *Suc-eq-number-of* [simp]:  
 $(\text{Suc } n = \text{number-of } v) =$   
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$   
 $\text{if neg } pv \text{ then False else nat } pv = n)$   
**by** (rule trans [OF eq-sym-conv eq-number-of-Suc])

**lemma** *less-number-of-Suc* [simp]:  
 $(\text{number-of } v < \text{Suc } n) =$   
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$   
 $\text{if neg } pv \text{ then True else nat } pv < n)$   
**apply** (simp only: *simp-thms Let-def neg-eq-less-0 linorder-not-less*  
*number-of-pred nat-number-of-def*  
*split add: split-if*)  
**apply** (rule-tac  $x = \text{number-of } v$  **in** *spec*)  
**apply** (auto simp add: *nat-less-iff*)  
**done**

**lemma** *less-Suc-number-of* [simp]:  
 $(\text{Suc } n < \text{number-of } v) =$   
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$   
 $\text{if neg } pv \text{ then False else } n < \text{nat } pv)$   
**apply** (simp only: *simp-thms Let-def neg-eq-less-0 linorder-not-less*  
*number-of-pred nat-number-of-def*  
*split add: split-if*)  
**apply** (rule-tac  $x = \text{number-of } v$  **in** *spec*)

**apply** (*auto simp add: zless-nat-eq-int-zless*)  
**done**

**lemma** *le-number-of-Suc* [*simp*]:  
 (number-of  $v \leq \text{Suc } n$ ) =  
 (let  $pv = \text{number-of } (\text{Int.pred } v)$  in  
   if  $\text{neg } pv$  then *True* else  $\text{nat } pv \leq n$ )  
**by** (*simp add: Let-def less-Suc-number-of linorder-not-less [symmetric]*)

**lemma** *le-Suc-number-of* [*simp*]:  
 ( $\text{Suc } n \leq \text{number-of } v$ ) =  
 (let  $pv = \text{number-of } (\text{Int.pred } v)$  in  
   if  $\text{neg } pv$  then *False* else  $n \leq \text{nat } pv$ )  
**by** (*simp add: Let-def less-number-of-Suc linorder-not-less [symmetric]*)

**lemma** *eq-number-of-Pls-Min*: (*Natural0 :: int*)  $\sim = \text{number-of Int.Min}$   
**by** *auto*

### 32.8 Max and Min Combined with *Suc*

**lemma** *max-number-of-Suc* [*simp*]:  
 max ( $\text{Suc } n$ ) (number-of  $v$ ) =  
 (let  $pv = \text{number-of } (\text{Int.pred } v)$  in  
   if  $\text{neg } pv$  then  $\text{Suc } n$  else  $\text{Suc}(\text{max } n (\text{nat } pv))$ )  
**apply** (*simp only: Let-def neg-eq-less-0 number-of-pred nat-number-of-def*  
   *split add: split-if nat.split*)  
**apply** (*rule-tac x = number-of v in spec*)  
**apply** *auto*  
**done**

**lemma** *max-Suc-number-of* [*simp*]:  
 max (number-of  $v$ ) ( $\text{Suc } n$ ) =  
 (let  $pv = \text{number-of } (\text{Int.pred } v)$  in  
   if  $\text{neg } pv$  then  $\text{Suc } n$  else  $\text{Suc}(\text{max } n (\text{nat } pv))$ )  
**apply** (*simp only: Let-def neg-eq-less-0 number-of-pred nat-number-of-def*  
   *split add: split-if nat.split*)  
**apply** (*rule-tac x = number-of v in spec*)  
**apply** *auto*  
**done**

**lemma** *min-number-of-Suc* [*simp*]:  
 min ( $\text{Suc } n$ ) (number-of  $v$ ) =  
 (let  $pv = \text{number-of } (\text{Int.pred } v)$  in  
   if  $\text{neg } pv$  then 0 else  $\text{Suc}(\text{min } n (\text{nat } pv))$ )  
**apply** (*simp only: Let-def neg-eq-less-0 number-of-pred nat-number-of-def*  
   *split add: split-if nat.split*)  
**apply** (*rule-tac x = number-of v in spec*)  
**apply** *auto*

done

**lemma** *min-Suc-number-of [simp]*:

*min (number-of v) (Suc n) =*  
*(let pv = number-of (Int.pred v) in*  
*if neg pv then 0 else Suc(min (nat pv) n))*

**apply** (*simp only: Let-def neg-eq-less-0 number-of-pred nat-number-of-def*  
*split add: split-if nat.split*)

**apply** (*rule-tac x = number-of v in spec*)

**apply** *auto*

done

### 32.9 Literal arithmetic involving powers

**lemma** *nat-power-eq: (0::int) <= z ==> nat (z^n) = nat z ^ n*

**apply** (*induct n*)

**apply** (*simp-all (no-asm-simp) add: nat-mult-distrib*)

done

**lemma** *power-nat-number-of:*

*(number-of v :: nat) ^ n =*  
*(if neg (number-of v :: int) then 0^n else nat ((number-of v :: int) ^ n))*

**by** (*simp only: simp-thms neg-nat not-neg-eq-ge-0 nat-number-of-def nat-power-eq*  
*split add: split-if cong: imp-cong*)

**lemmas** *power-nat-number-of-number-of = power-nat-number-of [of - number-of*  
*w, standard]*

**declare** *power-nat-number-of-number-of [simp]*

For arbitrary rings

**lemma** *power-number-of-even:*

**fixes** *z :: 'a::{number-ring,recpower}*

**shows** *z ^ number-of (Int.Bit0 w) = (let w = z ^ (number-of w) in w \* w)*

**unfolding** *Let-def nat-number-of-def number-of-Bit0*

**apply** (*rule-tac x = number-of w in spec, clarify*)

**apply** (*case-tac (0::int) <= x*)

**apply** (*auto simp add: nat-mult-distrib power-even-eq power2-eq-square*)

done

**lemma** *power-number-of-odd:*

**fixes** *z :: 'a::{number-ring,recpower}*

**shows** *z ^ number-of (Int.Bit1 w) = (if (0::int) <= number-of w*  
*then (let w = z ^ (number-of w) in z \* w \* w) else 1)*

**unfolding** *Let-def nat-number-of-def number-of-Bit1*

**apply** (*rule-tac x = number-of w in spec, auto*)

**apply** (*simp only: nat-add-distrib nat-mult-distrib*)

**apply** *simp*

**apply** (*auto simp add: nat-add-distrib nat-mult-distrib power-even-eq power2-eq-square*)

```
neg-nat power-Suc)
done
```

```
lemmas zpower-number-of-even = power-number-of-even [where 'a=int]
lemmas zpower-number-of-odd = power-number-of-odd [where 'a=int]
```

```
lemmas power-number-of-even-number-of [simp] =
  power-number-of-even [of number-of v, standard]
```

```
lemmas power-number-of-odd-number-of [simp] =
  power-number-of-odd [of number-of v, standard]
```

**ML**

```
<<
val numeral-ss = @{simpset} addsimps @{thms numerals};

val nat-bin-arith-setup =
  Lin-Arith.map-data
    (fn {add-mono-thms, mult-mono-thms, inj-thms, lessD, neqE, simpset} =>
      {add-mono-thms = add-mono-thms, mult-mono-thms = mult-mono-thms,
       inj-thms = inj-thms,
       lessD = lessD, neqE = neqE,
       simpset = simpset addsimps @{thms neg-simps} @
         [@{thm Suc-nat-number-of}, @{thm int-nat-number-of}]})
>>
```

```
declaration << K nat-bin-arith-setup >>
```

```
declare split-div[of - - number-of k, standard, arith-split]
declare split-mod[of - - number-of k, standard, arith-split]
```

```
lemma nat-number-of-Pls: Numeral0 = (0::nat)
  by (simp add: number-of-Pls nat-number-of-def)
```

```
lemma nat-number-of-Min: number-of Int.Min = (0::nat)
  apply (simp only: number-of-Min nat-number-of-def nat-zminus-int)
  done
```

```
lemma nat-number-of-Bit0:
  number-of (Int.Bit0 w) = (let n::nat = number-of w in n + n)
  unfolding nat-number-of-def number-of-is-id numeral-simps Let-def
  by auto
```

```
lemma nat-number-of-Bit1:
  number-of (Int.Bit1 w) =
    (if neg (number-of w :: int) then 0
```

else let  $n = \text{number-of } w \text{ in } \text{Suc } (n + n)$   
**unfolding** *nat-number-of-def number-of-is-id numeral-simps neg-def Let-def*  
**by** *auto*

**lemmas** *nat-number =*  
*nat-number-of-Pls nat-number-of-Min*  
*nat-number-of-Bit0 nat-number-of-Bit1*

**lemma** *Let-Suc [simp]: Let (Suc n) f == f (Suc n)*  
**by** (*simp add: Let-def*)

**lemma** *power-m1-even:  $(-1) ^ (2*n) = (1::'a::\{\text{number-ring}, \text{recpower}\})$*   
**by** (*simp add: power-mult power-Suc*)

**lemma** *power-m1-odd:  $(-1) ^ \text{Suc}(2*n) = (-1::'a::\{\text{number-ring}, \text{recpower}\})$*   
**by** (*simp add: power-mult power-Suc*)

### 32.10 Literal arithmetic and *of-nat*

**lemma** *of-nat-double:*  
 $0 \leq x \implies \text{of-nat } (\text{nat } (2 * x)) = \text{of-nat } (\text{nat } x) + \text{of-nat } (\text{nat } x)$   
**by** (*simp only: mult-2 nat-add-distrib of-nat-add*)

**lemma** *nat-numeral-m1-eq-0:  $-1 = (0::\text{nat})$*   
**by** (*simp only: nat-number-of-def*)

**lemma** *of-nat-number-of-lemma:*  
 $\text{of-nat } (\text{number-of } v :: \text{nat}) =$   
 $(\text{if } 0 \leq (\text{number-of } v :: \text{int})$   
 $\text{then } (\text{number-of } v :: 'a :: \text{number-ring})$   
 $\text{else } 0)$   
**by** (*simp add: int-number-of-def nat-number-of-def number-of-eq of-nat-nat*)

**lemma** *of-nat-number-of-eq [simp]:*  
 $\text{of-nat } (\text{number-of } v :: \text{nat}) =$   
 $(\text{if neg } (\text{number-of } v :: \text{int}) \text{ then } 0$   
 $\text{else } (\text{number-of } v :: 'a :: \text{number-ring}))$   
**by** (*simp only: of-nat-number-of-lemma neg-def, simp*)

### 32.11 Lemmas for the Combination and Cancellation Simprocs

**lemma** *nat-number-of-add-left:*  
 $\text{number-of } v + (\text{number-of } v' + (k::\text{nat})) =$   
 $(\text{if neg } (\text{number-of } v :: \text{int}) \text{ then } \text{number-of } v' + k$   
 $\text{else if neg } (\text{number-of } v' :: \text{int}) \text{ then } \text{number-of } v + k$   
 $\text{else } \text{number-of } (v + v') + k)$   
**unfolding** *nat-number-of-def number-of-is-id neg-def*  
**by** *auto*

**lemma** *nat-number-of-mult-left*:  

$$\text{number-of } v * (\text{number-of } v' * (k::\text{nat})) =$$

$$(\text{if } v < \text{Int.Pl} \text{ then } 0$$

$$\text{else number-of } (v * v') * k)$$
**by** *simp*

### 32.11.1 For combine-numerals

**lemma** *left-add-mult-distrib*:  $i*u + (j*u + k) = (i+j)*u + (k::\text{nat})$   
**by** (*simp add: add-mult-distrib*)

### 32.11.2 For cancel-numerals

**lemma** *nat-diff-add-eq1*:  

$$j \leq (i::\text{nat}) \implies ((i*u + m) - (j*u + n)) = (((i-j)*u + m) - n)$$
**by** (*simp split add: nat-diff-split add: add-mult-distrib*)

**lemma** *nat-diff-add-eq2*:  

$$i \leq (j::\text{nat}) \implies ((i*u + m) - (j*u + n)) = (m - ((j-i)*u + n))$$
**by** (*simp split add: nat-diff-split add: add-mult-distrib*)

**lemma** *nat-eq-add-iff1*:  

$$j \leq (i::\text{nat}) \implies (i*u + m = j*u + n) = ((i-j)*u + m = n)$$
**by** (*auto split add: nat-diff-split simp add: add-mult-distrib*)

**lemma** *nat-eq-add-iff2*:  

$$i \leq (j::\text{nat}) \implies (i*u + m = j*u + n) = (m = (j-i)*u + n)$$
**by** (*auto split add: nat-diff-split simp add: add-mult-distrib*)

**lemma** *nat-less-add-iff1*:  

$$j \leq (i::\text{nat}) \implies (i*u + m < j*u + n) = ((i-j)*u + m < n)$$
**by** (*auto split add: nat-diff-split simp add: add-mult-distrib*)

**lemma** *nat-less-add-iff2*:  

$$i \leq (j::\text{nat}) \implies (i*u + m < j*u + n) = (m < (j-i)*u + n)$$
**by** (*auto split add: nat-diff-split simp add: add-mult-distrib*)

**lemma** *nat-le-add-iff1*:  

$$j \leq (i::\text{nat}) \implies (i*u + m \leq j*u + n) = ((i-j)*u + m \leq n)$$
**by** (*auto split add: nat-diff-split simp add: add-mult-distrib*)

**lemma** *nat-le-add-iff2*:  

$$i \leq (j::\text{nat}) \implies (i*u + m \leq j*u + n) = (m \leq (j-i)*u + n)$$
**by** (*auto split add: nat-diff-split simp add: add-mult-distrib*)

### 32.11.3 For cancel-numeral-factors

**lemma** *nat-mult-le-cancel1*:  $(0::\text{nat}) < k \implies (k*m \leq k*n) = (m \leq n)$   
**by** *auto*

**lemma** *nat-mult-less-cancel1*:  $(0::nat) < k \implies (k*m < k*n) = (m < n)$   
**by** *auto*

**lemma** *nat-mult-eq-cancel1*:  $(0::nat) < k \implies (k*m = k*n) = (m = n)$   
**by** *auto*

**lemma** *nat-mult-div-cancel1*:  $(0::nat) < k \implies (k*m) \text{ div } (k*n) = (m \text{ div } n)$   
**by** *auto*

**lemma** *nat-mult-dvd-cancel-disj*[*simp*]:  
 $(k*m) \text{ dvd } (k*n) = (k=0 \mid m \text{ dvd } (n::nat))$   
**by**(*auto simp: dvd-eq-mod-eq-0 mod-mult-distrib2[symmetric]*)

**lemma** *nat-mult-dvd-cancel1*:  $0 < k \implies (k*m) \text{ dvd } (k*n::nat) = (m \text{ dvd } n)$   
**by**(*auto*)

### 32.11.4 For cancel-factor

**lemma** *nat-mult-le-cancel-disj*:  $(k*m \leq k*n) = ((0::nat) < k \longrightarrow m \leq n)$   
**by** *auto*

**lemma** *nat-mult-less-cancel-disj*:  $(k*m < k*n) = ((0::nat) < k \ \& \ m < n)$   
**by** *auto*

**lemma** *nat-mult-eq-cancel-disj*:  $(k*m = k*n) = (k = (0::nat) \mid m = n)$   
**by** *auto*

**lemma** *nat-mult-div-cancel-disj*[*simp*]:  
 $(k*m) \text{ div } (k*n) = (\text{if } k = (0::nat) \text{ then } 0 \text{ else } m \text{ div } n)$   
**by** (*simp add: nat-mult-div-cancel1*)

## 32.12 Simprocs for the Naturals

**use** *Tools/nat-simprocs.ML*

**declaration**  $\ll K \text{ nat-simprocs-setup} \gg$

### 32.12.1 For simplifying $Suc\ m - K$ and $K - Suc\ m$

Where K above is a literal

**lemma** *Suc-diff-eq-diff-pred*:  $Numeral0 < n \implies Suc\ m - n = m - (n - Numeral1)$   
**by** (*simp add: numeral-0-eq-0 numeral-1-eq-1 split add: nat-diff-split*)

Now just instantiating  $n$  to *number-of*  $v$  does the right simplification, but with some redundant inequality tests.

**lemma** *neg-number-of-pred-iff-0*:  
 $neg\ (\text{number-of } (Int.pred\ v)::int) = (\text{number-of } v = (0::nat))$   
**apply** (*subgoal-tac neg (number-of (Int.pred v)) = (number-of v < Suc 0)* )

```

apply (simp only: less-Suc-eq-le le-0-eq)
apply (subst less-number-of-Suc, simp)
done

```

No longer required as a simprule because of the *inverse-fold* simproc

```

lemma Suc-diff-number-of:
  Int.Pls < v ==>
    Suc m - (number-of v) = m - (number-of (Int.pred v))
apply (subst Suc-diff-eq-diff-pred)
apply simp
apply (simp del: nat-numeral-1-eq-1)
apply (auto simp only: diff-nat-number-of less-0-number-of [symmetric]
      neg-number-of-pred-iff-0)
done

```

```

lemma diff-Suc-eq-diff-pred: m - Suc n = (m - 1) - n
by (simp add: numerals split add: nat-diff-split)

```

### 32.12.2 For *nat-case* and *nat-rec*

```

lemma nat-case-number-of [simp]:
  nat-case a f (number-of v) =
    (let pv = number-of (Int.pred v) in
     if neg pv then a else f (nat pv))
by (simp split add: nat.split add: Let-def neg-number-of-pred-iff-0)

```

```

lemma nat-case-add-eq-if [simp]:
  nat-case a f ((number-of v) + n) =
    (let pv = number-of (Int.pred v) in
     if neg pv then nat-case a f n else f (nat pv + n))
apply (subst add-eq-if)
apply (simp split add: nat.split
      del: nat-numeral-1-eq-1
      add: nat-numeral-1-eq-1 [symmetric]
      numeral-1-eq-Suc-0 [symmetric]
      neg-number-of-pred-iff-0)
done

```

```

lemma nat-rec-number-of [simp]:
  nat-rec a f (number-of v) =
    (let pv = number-of (Int.pred v) in
     if neg pv then a else f (nat pv) (nat-rec a f (nat pv)))
apply (case-tac (number-of v) :: nat)
apply (simp-all (no-asm-simp) add: Let-def neg-number-of-pred-iff-0)
apply (simp split add: split-if-asm)
done

```

```

lemma nat-rec-add-eq-if [simp]:
  nat-rec a f (number-of v + n) =

```



```

      (let pv = number-of (Int.pred v) in
      if neg pv then nat-rec a f n
      else f (nat pv + n) (nat-rec a f (nat pv + n)))
apply (subst add-eq-if)
apply (simp split add: nat.split
      del: nat-numeral-1-eq-1
      add: nat-numeral-1-eq-1 [symmetric]
      numeral-1-eq-Suc-0 [symmetric]
      neg-number-of-pred-iff-0)
done

```

### 32.12.3 Various Other Lemmas

Evens and Odds, for Mutilated Chess Board

Lemmas for specialist use, NOT as default simprules

```

lemma nat-mult-2: 2 * z = (z+z::nat)
proof –
  have 2*z = (1 + 1)*z by simp
  also have ... = z+z by (simp add: left-distrib)
  finally show ?thesis .
qed

```

```

lemma nat-mult-2-right: z * 2 = (z+z::nat)
by (subst mult-commute, rule nat-mult-2)

```

Case analysis on  $n < (2::'a)$

```

lemma less-2-cases: (n::nat) < 2 ==> n = 0 | n = Suc 0
by arith

```

```

lemma div2-Suc-Suc [simp]: Suc(Suc m) div 2 = Suc (m div 2)
by arith

```

```

lemma add-self-div-2 [simp]: (m + m) div 2 = (m::nat)
by (simp add: nat-mult-2 [symmetric])

```

```

lemma mod2-Suc-Suc [simp]: Suc(Suc(m)) mod 2 = m mod 2
apply (subgoal-tac m mod 2 < 2)
apply (erule less-2-cases [THEN disjE])
apply (simp-all (no-asm-simp) add: Let-def mod-Suc nat-1)
done

```

```

lemma mod2-gr-0 [simp]: !!m::nat. (0 < m mod 2) = (m mod 2 = 1)
apply (subgoal-tac m mod 2 < 2)
apply (force simp del: mod-less-divisor, simp)
done

```

Removal of Small Numerals: 0, 1 and (in additive positions) 2

```

lemma add-2-eq-Suc [simp]: 2 + n = Suc (Suc n)

```

**by** *simp*

**lemma** *add-2-eq-Suc'* [*simp*]:  $n + 2 = \text{Suc } (\text{Suc } n)$   
**by** *simp*

Can be used to eliminate long strings of Sucs, but not by default

**lemma** *Suc3-eq-add-3*:  $\text{Suc } (\text{Suc } (\text{Suc } n)) = 3 + n$   
**by** *simp*

These lemmas collapse some needless occurrences of Suc: at least three Sucs, since two and fewer are rewritten back to Suc again! We already have some rules to simplify operands smaller than 3.

**lemma** *div-Suc-eq-div-add3* [*simp*]:  $m \text{ div } (\text{Suc } (\text{Suc } (\text{Suc } n))) = m \text{ div } (3+n)$   
**by** (*simp add: Suc3-eq-add-3*)

**lemma** *mod-Suc-eq-mod-add3* [*simp*]:  $m \text{ mod } (\text{Suc } (\text{Suc } (\text{Suc } n))) = m \text{ mod } (3+n)$   
**by** (*simp add: Suc3-eq-add-3*)

**lemma** *Suc-div-eq-add3-div*:  $(\text{Suc } (\text{Suc } (\text{Suc } m))) \text{ div } n = (3+m) \text{ div } n$   
**by** (*simp add: Suc3-eq-add-3*)

**lemma** *Suc-mod-eq-add3-mod*:  $(\text{Suc } (\text{Suc } (\text{Suc } m))) \text{ mod } n = (3+m) \text{ mod } n$   
**by** (*simp add: Suc3-eq-add-3*)

**lemmas** *Suc-div-eq-add3-div-number-of* =  
*Suc-div-eq-add3-div* [*of - number-of v, standard*]  
**declare** *Suc-div-eq-add3-div-number-of* [*simp*]

**lemmas** *Suc-mod-eq-add3-mod-number-of* =  
*Suc-mod-eq-add3-mod* [*of - number-of v, standard*]  
**declare** *Suc-mod-eq-add3-mod-number-of* [*simp*]

**end**

### 33 Groebner-Basis: Semiring normalization and Groebner Bases

**theory** *Groebner-Basis*  
**imports** *NatBin*  
**uses**  
*Tools/Groebner-Basis/misc.ML*  
*Tools/Groebner-Basis/normalizer-data.ML*  
(*Tools/Groebner-Basis/normalizer.ML*)  
(*Tools/Groebner-Basis/groebner.ML*)  
**begin**

### 33.1 Semiring normalization

**setup** *NormalizerData.setup*

**locale** *gb-semiring* =

**fixes** *add mul pwr r0 r1*

**assumes** *add-a: (add x (add y z) = add (add x y) z)*

**and** *add-c: add x y = add y x* **and** *add-0: add r0 x = x*

**and** *mul-a: mul x (mul y z) = mul (mul x y) z* **and** *mul-c: mul x y = mul y x*

**and** *mul-1: mul r1 x = x* **and** *mul-0: mul r0 x = r0*

**and** *mul-d: mul x (add y z) = add (mul x y) (mul x z)*

**and** *pwr-0: pwr x 0 = r1* **and** *pwr-Suc: pwr x (Suc n) = mul x (pwr x n)*

**begin**

**lemma** *mul-pwr: mul (pwr x p) (pwr x q) = pwr x (p + q)*

**proof** (*induct p*)

**case** *0*

**then show** *?case* **by** (*auto simp add: pwr-0 mul-1*)

**next**

**case** *Suc*

**from** *this [symmetric]* **show** *?case*

**by** (*auto simp add: pwr-Suc mul-1 mul-a*)

**qed**

**lemma** *pwr-mul: pwr (mul x y) q = mul (pwr x q) (pwr y q)*

**proof** (*induct q arbitrary: x y, auto simp add: pwr-0 pwr-Suc mul-1*)

**fix** *q x y*

**assume**  $\bigwedge x y. \text{pwr } (mul\ x\ y)\ q = mul\ (pwr\ x\ q)\ (pwr\ y\ q)$

**have**  $mul\ (mul\ x\ y)\ (mul\ (pwr\ x\ q)\ (pwr\ y\ q)) = mul\ x\ (mul\ y\ (mul\ (pwr\ x\ q)\ (pwr\ y\ q)))$

**by** (*simp add: mul-a*)

**also have**  $\dots = (mul\ (mul\ y\ (mul\ (pwr\ y\ q)\ (pwr\ x\ q)))\ x)$  **by** (*simp add: mul-c*)

**also have**  $\dots = (mul\ (mul\ y\ (pwr\ y\ q))\ (mul\ (pwr\ x\ q)\ x))$  **by** (*simp add: mul-a*)

**finally show**  $mul\ (mul\ x\ y)\ (mul\ (pwr\ x\ q)\ (pwr\ y\ q)) =$

$mul\ (mul\ x\ (pwr\ x\ q))\ (mul\ y\ (pwr\ y\ q))$  **by** (*simp add: mul-c*)

**qed**

**lemma** *pwr-pwr: pwr (pwr x p) q = pwr x (p \* q)*

**proof** (*induct p arbitrary: q*)

**case** *0*

**show** *?case* **using** *pwr-Suc mul-1 pwr-0* **by** (*induct q*) *auto*

**next**

**case** *Suc*

**thus** *?case* **by** (*auto simp add: mul-pwr [symmetric] pwr-mul pwr-Suc*)

**qed**

#### 33.1.1 Declaring the abstract theory

**lemma** *semiring-ops:*

**shows**  $TERM (add\ x\ y)$  **and**  $TERM (mul\ x\ y)$  **and**  $TERM (pwr\ x\ n)$   
**and**  $TERM\ r0$  **and**  $TERM\ r1$  .

**lemma** *semiring-rules*:

$add\ (mul\ a\ m)\ (mul\ b\ m) = mul\ (add\ a\ b)\ m$   
 $add\ (mul\ a\ m)\ m = mul\ (add\ a\ r1)\ m$   
 $add\ m\ (mul\ a\ m) = mul\ (add\ a\ r1)\ m$   
 $add\ m\ m = mul\ (add\ r1\ r1)\ m$   
 $add\ r0\ a = a$   
 $add\ a\ r0 = a$   
 $mul\ a\ b = mul\ b\ a$   
 $mul\ (add\ a\ b)\ c = add\ (mul\ a\ c)\ (mul\ b\ c)$   
 $mul\ r0\ a = r0$   
 $mul\ a\ r0 = r0$   
 $mul\ r1\ a = a$   
 $mul\ a\ r1 = a$   
 $mul\ (mul\ lx\ ly)\ (mul\ rx\ ry) = mul\ (mul\ lx\ rx)\ (mul\ ly\ ry)$   
 $mul\ (mul\ lx\ ly)\ (mul\ rx\ ry) = mul\ lx\ (mul\ ly\ (mul\ rx\ ry))$   
 $mul\ (mul\ lx\ ly)\ (mul\ rx\ ry) = mul\ rx\ (mul\ (mul\ lx\ ly)\ ry)$   
 $mul\ (mul\ lx\ ly)\ rx = mul\ (mul\ lx\ rx)\ ly$   
 $mul\ (mul\ lx\ ly)\ rx = mul\ lx\ (mul\ ly\ rx)$   
 $mul\ lx\ (mul\ rx\ ry) = mul\ (mul\ lx\ rx)\ ry$   
 $mul\ lx\ (mul\ rx\ ry) = mul\ rx\ (mul\ lx\ ry)$   
 $add\ (add\ a\ b)\ (add\ c\ d) = add\ (add\ a\ c)\ (add\ b\ d)$   
 $add\ (add\ a\ b)\ c = add\ a\ (add\ b\ c)$   
 $add\ a\ (add\ c\ d) = add\ c\ (add\ a\ d)$   
 $add\ (add\ a\ b)\ c = add\ (add\ a\ c)\ b$   
 $add\ a\ c = add\ c\ a$   
 $add\ a\ (add\ c\ d) = add\ (add\ a\ c)\ d$   
 $mul\ (pwr\ x\ p)\ (pwr\ x\ q) = pwr\ x\ (p + q)$   
 $mul\ x\ (pwr\ x\ q) = pwr\ x\ (Suc\ q)$   
 $mul\ (pwr\ x\ q)\ x = pwr\ x\ (Suc\ q)$   
 $mul\ x\ x = pwr\ x\ 2$   
 $pwr\ (mul\ x\ y)\ q = mul\ (pwr\ x\ q)\ (pwr\ y\ q)$   
 $pwr\ (pwr\ x\ p)\ q = pwr\ x\ (p * q)$   
 $pwr\ x\ 0 = r1$   
 $pwr\ x\ 1 = x$   
 $mul\ x\ (add\ y\ z) = add\ (mul\ x\ y)\ (mul\ x\ z)$   
 $pwr\ x\ (Suc\ q) = mul\ x\ (pwr\ x\ q)$   
 $pwr\ x\ (2*n) = mul\ (pwr\ x\ n)\ (pwr\ x\ n)$   
 $pwr\ x\ (Suc\ (2*n)) = mul\ x\ (mul\ (pwr\ x\ n)\ (pwr\ x\ n))$

**proof** –

**show**  $add\ (mul\ a\ m)\ (mul\ b\ m) = mul\ (add\ a\ b)\ m$  **using** *mul-d mul-c* **by** *simp*  
**next show**  $add\ (mul\ a\ m)\ m = mul\ (add\ a\ r1)\ m$  **using** *mul-d mul-c mul-1* **by** *simp*  
**next show**  $add\ m\ (mul\ a\ m) = mul\ (add\ a\ r1)\ m$  **using** *mul-c mul-d mul-1 add-c* **by** *simp*  
**next show**  $add\ m\ m = mul\ (add\ r1\ r1)\ m$  **using** *mul-c mul-d mul-1* **by** *simp*  
**next show**  $add\ r0\ a = a$  **using** *add-0* **by** *simp*

```

next show add a r0 = a using add-0 add-c by simp
next show mul a b = mul b a using mul-c by simp
next show mul (add a b) c = add (mul a c) (mul b c) using mul-c mul-d by
simp
next show mul r0 a = r0 using mul-0 by simp
next show mul a r0 = r0 using mul-0 mul-c by simp
next show mul r1 a = a using mul-1 by simp
next show mul a r1 = a using mul-1 mul-c by simp
next show mul (mul lx ly) (mul rx ry) = mul (mul lx rx) (mul ly ry)
  using mul-c mul-a by simp
next show mul (mul lx ly) (mul rx ry) = mul lx (mul ly (mul rx ry))
  using mul-a by simp
next
  have mul (mul lx ly) (mul rx ry) = mul (mul rx ry) (mul lx ly) by (rule mul-c)
  also have ... = mul rx (mul ry (mul lx ly)) using mul-a by simp
  finally
    show mul (mul lx ly) (mul rx ry) = mul rx (mul (mul lx ly) ry)
      using mul-c by simp
next show mul (mul lx ly) rx = mul (mul lx rx) ly using mul-c mul-a by simp
next
  show mul (mul lx ly) rx = mul lx (mul ly rx) by (simp add: mul-a)
next show mul lx (mul rx ry) = mul (mul lx rx) ry by (simp add: mul-a)
next show mul lx (mul rx ry) = mul rx (mul lx ry) by (simp add: mul-a, simp
add: mul-c)
next show add (add a b) (add c d) = add (add a c) (add b d)
  using add-c add-a by simp
next show add (add a b) c = add a (add b c) using add-a by simp
next show add a (add c d) = add c (add a d)
  apply (simp add: add-a) by (simp only: add-c)
next show add (add a b) c = add (add a c) b using add-a add-c by simp
next show add a c = add c a by (rule add-c)
next show add a (add c d) = add (add a c) d using add-a by simp
next show mul (pwr x p) (pwr x q) = pwr x (p + q) by (rule mul-pwr)
next show mul x (pwr x q) = pwr x (Suc q) using pwr-Suc by simp
next show mul (pwr x q) x = pwr x (Suc q) using pwr-Suc mul-c by simp
next show mul x x = pwr x 2 by (simp add: nat-number pwr-Suc pwr-0 mul-1
mul-c)
next show pwr (mul x y) q = mul (pwr x q) (pwr y q) by (rule pwr-mul)
next show pwr (pwr x p) q = pwr x (p * q) by (rule pwr-pwr)
next show pwr x 0 = r1 using pwr-0 .
next show pwr x 1 = x unfolding One-nat-def by (simp add: nat-number
pwr-Suc pwr-0 mul-1 mul-c)
next show mul x (add y z) = add (mul x y) (mul x z) using mul-d by simp
next show pwr x (Suc q) = mul x (pwr x q) using pwr-Suc by simp
next show pwr x (2 * n) = mul (pwr x n) (pwr x n) by (simp add: nat-number
mul-pwr)
next show pwr x (Suc (2 * n)) = mul x (mul (pwr x n) (pwr x n))
  by (simp add: nat-number pwr-Suc mul-pwr)
qed

```

```

lemmas gb-semiring-axioms' =
  gb-semiring-axioms [normalizer
    semiring ops: semiring-ops
    semiring rules: semiring-rules]

```

```

end

```

```

interpretation class-semiring: gb-semiring
  op + op * op ^ 0::'a::{comm-semiring-1, recpower} 1
proof qed (auto simp add: algebra-simps power-Suc)

```

```

lemmas nat-arith =
  add-nat-number-of
  diff-nat-number-of
  mult-nat-number-of
  eq-nat-number-of
  less-nat-number-of

```

```

lemma not-iszero-Numeral1:  $\neg$  iszero (Numeral1::'a::number-ring)
  by (simp add: numeral-1-eq-1)

```

```

lemmas comp-arith =
  Let-def arith-simps nat-arith rel-simps neg-simps if-False
  if-True add-0 add-Suc add-number-of-left mult-number-of-left
  numeral-1-eq-1 [symmetric] Suc-eq-add-numeral-1
  numeral-0-eq-0 [symmetric] numerals [symmetric]
  iszero-simps not-iszero-Numeral1

```

```

lemmas semiring-norm = comp-arith

```

```

ML <<
  local

```

```

  open Conv;

```

```

  fun numeral-is-const ct = can HOLogic.dest-number (Thm.term-of ct);

```

```

  fun int-of-rat x =
    (case Rat.quotient-of-rat x of (i, 1)  $\Rightarrow$  i
    | -  $\Rightarrow$  error int-of-rat: bad int);

```

```

  val numeral-conv =
    Simplifier.rewrite (HOL-basic-ss addsimps @ {thms semiring-norm}) then-conv
    Simplifier.rewrite (HOL-basic-ss addsimps
      (@ {thms numeral-1-eq-1} @ @ {thms numeral-0-eq-0} @ @ {thms numerals(1-2)}));

```

```

  in

```

```

fun normalizer-funs key =
  NormalizerData.funs key
  {is-const = fn phi => numeral-is-const,
   dest-const = fn phi => fn ct =>
     Rat.rat-of-int (snd
       (HOLogic.dest-number (Thm.term-of ct)
         handle TERM - => error ring-dest-const)),
   mk-const = fn phi => fn cT => fn x => Numeral.mk-cnumber cT (int-of-rat
     x),
   conv = fn phi => K numeral-conv}

end
>>

declaration << normalizer-funs @{thm class-semiring.gb-semiring-axioms'} >>

locale gb-ring = gb-semiring +
  fixes sub :: 'a ⇒ 'a ⇒ 'a
  and neg :: 'a ⇒ 'a
  assumes neg-mul: neg x = mul (neg r1) x
  and sub-add: sub x y = add x (neg y)
begin

lemma ring-ops: shows TERM (sub x y) and TERM (neg x) .

lemmas ring-rules = neg-mul sub-add

lemmas gb-ring-axioms' =
  gb-ring-axioms [normalizer
    semiring ops: semiring-ops
    semiring rules: semiring-rules
    ring ops: ring-ops
    ring rules: ring-rules]

end

interpretation class-ring: gb-ring op + op * op ^
  0::'a::{comm-semiring-1,recpower,number-ring} 1 op - uminus
proof qed simp-all

declaration << normalizer-funs @{thm class-ring.gb-ring-axioms'} >>

use Tools/Groebner-Basis/normalizer.ML

```

```

method-setup sring-norm = ⟨⟨
  Scan.succeed (SIMPLE-METHOD' o Normalizer.semiring-normalize-tac)
  ⟩⟩ semiring normalizer

```

```

locale gb-field = gb-ring +
  fixes divide :: 'a ⇒ 'a ⇒ 'a
  and inverse:: 'a ⇒ 'a
  assumes divide-inverse: divide x y = mul x (inverse y)
  and inverse-divide: inverse x = divide r1 x
begin

```

```

lemma field-ops: shows TERM (divide x y) and TERM (inverse x) .

```

```

lemmas field-rules = divide-inverse inverse-divide

```

```

lemmas gb-field-axioms' =
  gb-field-axioms [normalizer
    semiring ops: semiring-ops
    semiring rules: semiring-rules
    ring ops: ring-ops
    ring rules: ring-rules
    field ops: field-ops
    field rules: field-rules]

```

```

end

```

### 33.2 Groebner Bases

```

locale semiringb = gb-semiring +
  assumes add-cancel: add (x::'a) y = add x z ⟷ y = z
  and add-mul-solve: add (mul w y) (mul x z) =
    add (mul w z) (mul x y) ⟷ w = x ∨ y = z
begin

```

```

lemma noteq-reduce: a ≠ b ∧ c ≠ d ⟷ add (mul a c) (mul b d) ≠ add (mul a
d) (mul b c)

```

```

proof–

```

```

  have a ≠ b ∧ c ≠ d ⟷ ¬ (a = b ∨ c = d) by simp
  also have ... ⟷ add (mul a c) (mul b d) ≠ add (mul a d) (mul b c)
  using add-mul-solve by blast
  finally show a ≠ b ∧ c ≠ d ⟷ add (mul a c) (mul b d) ≠ add (mul a d) (mul
b c)
  by simp

```

```

qed

```

```

lemma add-scale-eq-noteq: [[r ≠ r0 ; (a = b) ∧ ~(c = d)]]
  ⇒ add a (mul r c) ≠ add b (mul r d)

```

```

proof(clarify)

```



```

assume nz:  $r \neq r0$  and cnd:  $c \neq d$ 
and eq:  $\text{add } b \ (\text{mul } r \ c) = \text{add } b \ (\text{mul } r \ d)$ 
hence  $\text{mul } r \ c = \text{mul } r \ d$  using cnd add-cancel by simp
hence  $\text{add } (\text{mul } r0 \ d) \ (\text{mul } r \ c) = \text{add } (\text{mul } r0 \ c) \ (\text{mul } r \ d)$ 
using mul-0 add-cancel by simp
thus False using add-mul-solve nz cnd by simp
qed

```

```

lemma add-r0-iff:  $x = \text{add } x \ a \longleftrightarrow a = r0$ 
proof –
  have  $a = r0 \longleftrightarrow \text{add } x \ a = \text{add } x \ r0$  by (simp add: add-cancel)
  thus  $x = \text{add } x \ a \longleftrightarrow a = r0$  by (auto simp add: add-c add-0)
qed

```

```

declare gb-semiring-axioms' [normalizer del]

```

```

lemmas semiringb-axioms' = semiringb-axioms [normalizer
  semiring ops: semiring-ops
  semiring rules: semiring-rules
  idom rules: noteq-reduce add-scale-eq-noteq]

```

```

end

```

```

locale ringb = semiringb + gb-ring +
  assumes subr0-iff:  $\text{sub } x \ y = r0 \longleftrightarrow x = y$ 
begin

```

```

declare gb-ring-axioms' [normalizer del]

```

```

lemmas ringb-axioms' = ringb-axioms [normalizer
  semiring ops: semiring-ops
  semiring rules: semiring-rules
  ring ops: ring-ops
  ring rules: ring-rules
  idom rules: noteq-reduce add-scale-eq-noteq
  ideal rules: subr0-iff add-r0-iff]

```

```

end

```

```

lemma no-zero-divisors-neq0:
  assumes az:  $(a::'a::\text{no-zero-divisors}) \neq 0$ 
  and ab:  $a*b = 0$  shows  $b = 0$ 
proof –
  { assume bz:  $b \neq 0$ 
    from no-zero-divisors [OF az bz] ab have False by blast }
  thus  $b = 0$  by blast
qed

```

```

interpretation class-ringb: ringb
  op + op * op ^ 0::'a::{idom,recpower,number-ring} 1 op - uminus
proof(unfold-locales, simp add: algebra-simps power-Suc, auto)
  fix w x y z ::'a::{idom,recpower,number-ring}
  assume p: w * y + x * z = w * z + x * y and ynz: y ≠ z
  hence ynz': y - z ≠ 0 by simp
  from p have w * y + x * z - w * z - x * y = 0 by simp
  hence w * (y - z) - x * (y - z) = 0 by (simp add: algebra-simps)
  hence (y - z) * (w - x) = 0 by (simp add: algebra-simps)
  with no-zero-divisors-neq0 [OF ynz']
  have w - x = 0 by blast
  thus w = x by simp
qed

declaration ⟨⟨ normalizer-funs @ {thm class-ringb.ringb-axioms'} ⟩⟩

interpretation natgb: semiringb
  op + op * op ^ 0::nat 1
proof (unfold-locales, simp add: algebra-simps power-Suc)
  fix w x y z ::nat
  { assume p: w * y + x * z = w * z + x * y and ynz: y ≠ z
    hence y < z ∨ y > z by arith
    moreover {
      assume lt: y < z hence ∃ k. z = y + k ∧ k > 0 by (rule-tac x=z - y in
exI, auto)
      then obtain k where kp: k>0 and yz: z = y + k by blast
      from p have (w * y + x * y) + x * k = (w * y + x * y) + w * k by (simp add:
yz algebra-simps)
      hence x * k = w * k by simp
      hence w = x using kp by (simp add: mult-cancel2) }
    moreover {
      assume lt: y > z hence ∃ k. y = z + k ∧ k > 0 by (rule-tac x=y - z in exI,
auto)
      then obtain k where kp: k>0 and yz: y = z + k by blast
      from p have (w * z + x * z) + w * k = (w * z + x * z) + x * k by (simp add:
yz algebra-simps)
      hence w * k = x * k by simp
      hence w = x using kp by (simp add: mult-cancel2)}
    ultimately have w=x by blast }
  thus (w * y + x * z = w * z + x * y) = (w = x ∨ y = z) by auto
qed

declaration ⟨⟨ normalizer-funs @ {thm natgb.semiringb-axioms'} ⟩⟩

locale fieldgb = ringb + gb-field
begin

declare gb-field-axioms' [normalizer del]

```

```

lemmas fieldgb-axioms' = fieldgb-axioms [normalizer
  semiring ops: semiring-ops
  semiring rules: semiring-rules
  ring ops: ring-ops
  ring rules: ring-rules
  field ops: field-ops
  field rules: field-rules
  idom rules: noteq-reduce add-scale-eq-noteq
  ideal rules: subr0-iff add-r0-iff]

```

**end**

```

lemmas bool-simps = simp-thms(1-34)

```

```

lemma dnf:

```

$$(P \ \& \ (Q \mid R)) = ((P \ \& \ Q) \mid (P \ \& \ R)) \quad ((Q \mid R) \ \& \ P) = ((Q \ \& \ P) \mid (R \ \& \ P))$$

$$(P \wedge Q) = (Q \wedge P) \quad (P \vee Q) = (Q \vee P)$$

**by** blast+

```

lemmas weak-dnf-simps = dnf bool-simps

```

```

lemma nnf-simps:

```

$$(\neg(P \wedge Q)) = (\neg P \vee \neg Q) \quad (\neg(P \vee Q)) = (\neg P \wedge \neg Q) \quad (P \longrightarrow Q) = (\neg P \vee Q)$$

$$(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg Q)) \quad (\neg \neg(P)) = P$$

**by** blast+

```

lemma PFalse:

```

$$P \equiv \text{False} \implies \neg P$$

$$\neg P \implies (P \equiv \text{False})$$

**by** auto

```

use Tools/Groebner-Basis/groebner.ML

```

```

method-setup algebra =

```

```

  <<

```

```

  let

```

```

    fun keyword k = Scan.lift (Args.$$$ k -- Args.colon) >> K ()

```

```

    val addN = add

```

```

    val delN = del

```

```

    val any-keyword = keyword addN || keyword delN

```

```

    val thms = Scan.repeat (Scan.unless any-keyword Attrib.multi-thm) >> flat;

```

```

  in

```

```

    ((Scan.optional (keyword addN |-- thms) []) --

```

```

     (Scan.optional (keyword delN |-- thms) [])) >>

```

```

    (fn (add-ths, del-ths) => fn ctxt =>

```

```

      SIMPLE-METHOD' (Groebner.algebra-tac add-ths del-ths ctxt))

```

```

  end

```

```

  >> solve polynomial equations over (semi)rings and ideal membership problems using
  Groebner bases

```

```

declare dvd-def[algebra]

```

```

declare dvd-eq-mod-eq-0[symmetric, algebra]
declare mod-div-trivial[algebra]
declare mod-mod-trivial[algebra]
declare conjunct1[OF DIVISION-BY-ZERO, algebra]
declare conjunct2[OF DIVISION-BY-ZERO, algebra]
declare zmod-zdiv-equality[symmetric, algebra]
declare zdiv-zmod-equality[symmetric, algebra]
declare zdiv-zminus-zminus[algebra]
declare zmod-zminus-zminus[algebra]
declare zdiv-zminus2[algebra]
declare zmod-zminus2[algebra]
declare zdiv-zero[algebra]
declare zmod-zero[algebra]
declare mod-by-1[algebra]
declare div-by-1[algebra]
declare zmod-minus1-right[algebra]
declare zdiv-minus1-right[algebra]
declare mod-div-trivial[algebra]
declare mod-mod-trivial[algebra]
declare mod-mult-self2-is-0[algebra]
declare mod-mult-self1-is-0[algebra]
declare zmod-eq-0-iff[algebra]
declare dvd-0-left-iff[algebra]
declare zdvd1-eq[algebra]
declare zmod-eq-dvd-iff[algebra]
declare nat-mod-eq-iff[algebra]

```

### 33.3 Groebner Bases for fields

**interpretation** class-fieldgb:

fieldgb op + op \* op ^ 0::'a::{field,recpower,number-ring} 1 op - uminus op /  
inverse **apply** (unfold-locales) **by** (simp-all add: divide-inverse)

**lemma** divide-Numeral1: (x::'a::{field,number-ring}) / Numeral1 = x **by** simp

**lemma** divide-Numeral0: (x::'a::{field,number-ring, division-by-zero}) / Numeral0  
= 0

**by** simp

**lemma** mult-frac-frac: ((x::'a::{field,division-by-zero}) / y) \* (z / w) = (x\*z) /  
(y\*w)

**by** simp

**lemma** mult-frac-num: ((x::'a::{field, division-by-zero}) / y) \* z = (x\*z) / y

**by** simp

**lemma** mult-num-frac: ((x::'a::{field, division-by-zero}) / y) \* z = (x\*z) / y

**by** simp

**lemma** Numeral1-eq1-nat: (1::nat) = Numeral1 **by** simp

**lemma** add-frac-num:  $y \neq 0 \implies (x::'a::{field, division-by-zero}) / y + z = (x + z*y) / y$

```

  by (simp add: add-divide-distrib)
lemma add-num-frac:  $y \neq 0 \implies z + (x :: 'a :: \{field, division-by-zero\}) / y = (x + z * y) / y$ 
  by (simp add: add-divide-distrib)
ML⟨⟨ let open Conv in fconv-rule (arg-conv (arg1-conv (rewr-conv (mk-meta-eq
@{thm mult-commute})))) (@{thm divide-inverse} RS sym)end⟩⟩
ML⟨⟨
local
  val zr = @{cpat 0}
  val zT = ctyp-of-term zr
  val geq = @{cpat op =}
  val eqT = Thm.dest-ctyp (ctyp-of-term geq) |> hd
  val add-frac-eq = mk-meta-eq @{thm add-frac-eq}
  val add-frac-num = mk-meta-eq @{thm add-frac-num}
  val add-num-frac = mk-meta-eq @{thm add-num-frac}

  fun prove-nz ss T t =
    let
      val z = instantiate-cterm ([ (zT, T)], []) zr
      val eq = instantiate-cterm ([ (eqT, T)], []) geq
      val th = Simplifier.rewrite (ss addsimps simp-thms)
        (Thm.capply @{cterm Trueprop} (Thm.capply @{cterm Not}
          (Thm.capply (Thm.capply eq t) z)))
      in equal-elim (symmetric th) TrueI
    end

  fun proc phi ss ct =
    let
      val ((x,y),(w,z)) =
        (Thm.dest-binop #> (fn (a,b) => (Thm.dest-binop a, Thm.dest-binop b)))
    ct
      val - = map (HOLogic.dest-number o term-of) [x,y,z,w]
      val T = ctyp-of-term x
      val [y-nz, z-nz] = map (prove-nz ss T) [y, z]
      val th = instantiate' [SOME T] (map SOME [y,z,x,w]) add-frac-eq
      in SOME (implies-elim (implies-elim th y-nz) z-nz)
    end
      handle CTERM - => NONE | TERM - => NONE | THM - => NONE

  fun proc2 phi ss ct =
    let
      val (l,r) = Thm.dest-binop ct
      val T = ctyp-of-term l
      in (case (term-of l, term-of r) of
        (Const(@{const-name HOL.divide},-)$-,-) =>
          let val (x,y) = Thm.dest-binop l val z = r
            val - = map (HOLogic.dest-number o term-of) [x,y,z]
            val ynz = prove-nz ss T y
            in SOME (implies-elim (instantiate' [SOME T] (map SOME [y,x,z])

```

```

add-frac-num) ynz)
  end
  | (-, Const (@{const-name HOL.divide},-)$.$.-) =>
    let val (x,y) = Thm.dest-binop r val z = l
        val - = map (HOLLogic.dest-number o term-of) [x,y,z]
        val ynz = prove-nz ss T y
        in SOME (implies-elim (instantiate' [SOME T] (map SOME [y,z,x])
add-num-frac) ynz)
    end
  | - => NONE)
end
handle CTERM - => NONE | TERM - => NONE | THM - => NONE

fun is-number (Const(@{const-name HOL.divide},-)$.$.b) = is-number a andalso
is-number b
  | is-number t = can HOLLogic.dest-number t

val is-number = is-number o term-of

fun proc3 phi ss ct =
  (case term-of ct of
    Const(@{const-name HOL.less},-)$(Const(@{const-name HOL.divide},-)$.$.-)-
=>
    let
      val ((a,b),c) = Thm.dest-binop ct |>> Thm.dest-binop
      val - = map is-number [a,b,c]
      val T = ctyp-of-term c
      val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm divide-less-eq}
      in SOME (mk-meta-eq th) end
    | Const(@{const-name HOL.less-eq},-)$(Const(@{const-name HOL.divide},-)$.$.-)-
=>
    let
      val ((a,b),c) = Thm.dest-binop ct |>> Thm.dest-binop
      val - = map is-number [a,b,c]
      val T = ctyp-of-term c
      val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm divide-le-eq}
      in SOME (mk-meta-eq th) end
    | Const(op =,-)$(Const(@{const-name HOL.divide},-)$.$.-)- =>
    let
      val ((a,b),c) = Thm.dest-binop ct |>> Thm.dest-binop
      val - = map is-number [a,b,c]
      val T = ctyp-of-term c
      val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm divide-eq-eq}
      in SOME (mk-meta-eq th) end
    | Const(@{const-name HOL.less},-)$.$(Const(@{const-name HOL.divide},-)$.$.-)-
=>
    let
      val (a,(b,c)) = Thm.dest-binop ct ||> Thm.dest-binop
      val - = map is-number [a,b,c]

```

```

    val T = ctyp-of-term c
    val th = instantiate' [SOME T] (map SOME [a,b,c]) @ {thm less-divide-eq}
    in SOME (mk-meta-eq th) end
  | Const(@ {const-name HOL.less-eq},-) $- $(Const(@ {const-name HOL.divide},-) $- $-)
=>
  let
    val (a,(b,c)) = Thm.dest-binop ct ||> Thm.dest-binop
    val - = map is-number [a,b,c]
    val T = ctyp-of-term c
    val th = instantiate' [SOME T] (map SOME [a,b,c]) @ {thm le-divide-eq}
    in SOME (mk-meta-eq th) end
  | Const(op =,-) $- $(Const(@ {const-name HOL.divide},-) $- $-) =>
  let
    val (a,(b,c)) = Thm.dest-binop ct ||> Thm.dest-binop
    val - = map is-number [a,b,c]
    val T = ctyp-of-term c
    val th = instantiate' [SOME T] (map SOME [a,b,c]) @ {thm eq-divide-eq}
    in SOME (mk-meta-eq th) end
  | - => NONE)
  handle TERM - => NONE | CTERM - => NONE | THM - => NONE

val add-frac-frac-simproc =
  make-simproc {lhss = [@ {cpat (?x::?'a::field)/?y + (?w::?'a::field)/?z}],
    name = add-frac-frac-simproc,
    proc = proc, identifier = []}

val add-frac-num-simproc =
  make-simproc {lhss = [@ {cpat (?x::?'a::field)/?y + ?z}, @ {cpat ?z +
    (?x::?'a::field)/?y}],
    name = add-frac-num-simproc,
    proc = proc2, identifier = []}

val ord-frac-simproc =
  make-simproc
    {lhss = [@ {cpat (?a::(?'a::{field, ord}))/?b < ?c},
      @ {cpat (?a::(?'a::{field, ord}))/?b ≤ ?c},
      @ {cpat ?c < (?a::(?'a::{field, ord}))/?b},
      @ {cpat ?c ≤ (?a::(?'a::{field, ord}))/?b},
      @ {cpat ?c = ((?a::(?'a::{field, ord}))/?b)},
      @ {cpat ((?a::(?'a::{field, ord}))/ ?b) = ?c}],
    name = ord-frac-simproc, proc = proc3, identifier = []}

local
open Conv
in

val ths = [@ {thm mult-numeral-1}, @ {thm mult-numeral-1-right},
  @ {thm divide-Numeral1},
  @ {thm Ring-and-Field.divide-zero}, @ {thm divide-Numeral0},

```

```

    @{thm divide-divide-eq-left}, @{thm mult-frac-frac},
    @{thm mult-num-frac}, @{thm mult-frac-num},
    @{thm mult-frac-frac}, @{thm times-divide-eq-right},
    @{thm times-divide-eq-left}, @{thm divide-divide-eq-right},
    @{thm diff-def}, @{thm minus-divide-left},
    @{thm Numeral1-eq1-nat}, @{thm add-divide-distrib} RS sym,
    @{thm divide-inverse} RS sym, @{thm inverse-divide},
    fconv-rule (arg-conv (arg1-conv (rewr-conv (mk-meta-eq @{thm mult-commute}))))

    (@{thm divide-inverse} RS sym)]

val comp-conv = (Simplifier.rewrite
  (HOL-basic-ss addsimps @{thms Groebner-Basis.comp-arith}
    addsimps ths addsimps simp-thms
    addsimprocs field-cancel-numeral-factors
    addsimprocs [add-frac-frac-simproc, add-frac-num-simproc,
      ord-frac-simproc]
    addcongs [@{thm if-weak-cong}])))
then-conv (Simplifier.rewrite (HOL-basic-ss addsimps
  [@{thm numeral-1-eq-1},@{thm numeral-0-eq-0}] @ @{thms numerals(1-2)}))
end

fun numeral-is-const ct =
  case term-of ct of
    Const (@{const-name HOL.divide},-) $ a $ b =>
      can HOLogic.dest-number a andalso can HOLogic.dest-number b
  | Const (@{const-name HOL.inverse},-) $ t => can HOLogic.dest-number t
  | t => can HOLogic.dest-number t

fun dest-const ct = ((case term-of ct of
  Const (@{const-name HOL.divide},-) $ a $ b =>
    Rat.rat-of-quotient (snd (HOLogic.dest-number a), snd (HOLogic.dest-number
b))
  | Const (@{const-name HOL.inverse},-) $ t =>
    Rat.inv (Rat.rat-of-int (snd (HOLogic.dest-number t)))
  | t => Rat.rat-of-int (snd (HOLogic.dest-number t)))
  handle TERM - => error ring-dest-const)

fun mk-const phi cT x =
  let val (a, b) = Rat.quotient-of-rat x
  in if b = 1 then Numeral.mk-cnumber cT a
    else Thm.capply
      (Thm.capply (Drule.ctrm-rule (instantiate' [SOME cT] []) @ {cpat op /})
        (Numeral.mk-cnumber cT a))
      (Numeral.mk-cnumber cT b)
  end

in
  val field-comp-conv = comp-conv;

```



```

val fieldgb-declaration =
  NormalizerData.funs @{thm class-fieldgb.fieldgb-axioms'}
  {is-const = K numeral-is-const,
   dest-const = K dest-const,
   mk-const = mk-const,
   conv = K (K comp-conv)}
end;
>>

```

```

declaration fieldgb-declaration

```

```

end

```

### 34 SetInterval: Set intervals

```

theory SetInterval
imports Int
begin

```

```

context ord
begin

```

```

definition
  lessThan    :: 'a => 'a set ((1{..<})) where
  {..} == {x. x < u}

```

```

definition
  atMost      :: 'a => 'a set ((1{..})) where
  {..u} == {x. x ≤ u}

```

```

definition
  greaterThan :: 'a => 'a set ((1{<..})) where
  {l<..} == {x. l < x}

```

```

definition
  atLeast     :: 'a => 'a set ((1{-..})) where
  {l..} == {x. l ≤ x}

```

```

definition
  greaterThanLessThan :: 'a => 'a => 'a set ((1{<..<})) where
  {l<..} == {l<..} Int {..}

```

```

definition
  atLeastLessThan :: 'a => 'a => 'a set ((1{-..<})) where
  {l..} == {l..} Int {..}

```

```

definition
  greaterThanAtMost :: 'a => 'a => 'a set ((1{<..})) where
  {l<..u} == {l<..} Int {..u}

```

**definition**

*atLeastAtMost* :: 'a => 'a => 'a set      ((1{...})) **where**  
 {l..u} == {l..} Int {..u}

**end**

A note of warning when using  $\{..<n\}$  on type *nat*: it is equivalent to  $\{0..<n\}$  but some lemmas involving  $\{m..<n\}$  may not exist in  $\{..<n\}$ -form as well.

**syntax**

@UNION-le :: 'a => 'a => 'b set => 'b set      ((3UN -<= - / -) 10)  
 @UNION-less :: 'a => 'a => 'b set => 'b set      ((3UN -< - / -) 10)  
 @INTER-le :: 'a => 'a => 'b set => 'b set      ((3INT -<= - / -) 10)  
 @INTER-less :: 'a => 'a => 'b set => 'b set      ((3INT -< - / -) 10)

**syntax** (*xsymbols*)

@UNION-le :: 'a => 'a => 'b set => 'b set      ((3U -<= - / -) 10)  
 @UNION-less :: 'a => 'a => 'b set => 'b set      ((3U -< - / -) 10)  
 @INTER-le :: 'a => 'a => 'b set => 'b set      ((3I -<= - / -) 10)  
 @INTER-less :: 'a => 'a => 'b set => 'b set      ((3I -< - / -) 10)

**syntax** (*latex output*)

@UNION-le :: 'a => 'a => 'b set => 'b set      ((3U (00- ≤ -) / -) 10)  
 @UNION-less :: 'a => 'a => 'b set => 'b set      ((3U (00- < -) / -) 10)  
 @INTER-le :: 'a => 'a => 'b set => 'b set      ((3I (00- ≤ -) / -) 10)  
 @INTER-less :: 'a => 'a => 'b set => 'b set      ((3I (00- < -) / -) 10)

**translations**

UN  $i \leq n$ . A == UN  $i:\{..n\}$ . A  
 UN  $i < n$ . A == UN  $i:\{..<n\}$ . A  
 INT  $i \leq n$ . A == INT  $i:\{..n\}$ . A  
 INT  $i < n$ . A == INT  $i:\{..<n\}$ . A

**34.1 Various equivalences**

**lemma** (in *ord*) *lessThan-iff* [*iff*]: ( $i$ : *lessThan*  $k$ ) = ( $i < k$ )  
**by** (*simp add: lessThan-def*)

**lemma** *Compl-lessThan* [*simp*]:

!! $k$ :: 'a::linorder.  $\neg$ *lessThan*  $k$  = *atLeast*  $k$

**apply** (*auto simp add: lessThan-def atLeast-def*)  
**done**

**lemma** *single-Diff-lessThan* [*simp*]: !! $k$ :: 'a::order.  $\{k\} - \text{lessThan } k = \{k\}$   
**by** *auto*

**lemma** (in *ord*) *greaterThan-iff* [*iff*]: ( $i$ : *greaterThan*  $k$ ) = ( $k < i$ )  
**by** (*simp add: greaterThan-def*)

```

lemma Compl-greaterThan [simp]:
  !!k:: 'a::linorder. -greaterThan k = atMost k
  by (auto simp add: greaterThan-def atMost-def)

lemma Compl-atMost [simp]: !!k:: 'a::linorder. -atMost k = greaterThan k
apply (subst Compl-greaterThan [symmetric])
apply (rule double-complement)
done

lemma (in ord) atLeast-iff [iff]: (i: atLeast k) = (k<=i)
by (simp add: atLeast-def)

lemma Compl-atLeast [simp]:
  !!k:: 'a::linorder. -atLeast k = lessThan k
  by (auto simp add: lessThan-def atLeast-def)

lemma (in ord) atMost-iff [iff]: (i: atMost k) = (i<=k)
by (simp add: atMost-def)

lemma atMost-Int-atLeast: !!n:: 'a::order. atMost n Int atLeast n = {n}
by (blast intro: order-antisym)

```

### 34.2 Logical Equivalences for Set Inclusion and Equality

```

lemma atLeast-subset-iff [iff]:
  (atLeast x ⊆ atLeast y) = (y ≤ (x::'a::order))
by (blast intro: order-trans)

lemma atLeast-eq-iff [iff]:
  (atLeast x = atLeast y) = (x = (y::'a::linorder))
by (blast intro: order-antisym order-trans)

lemma greaterThan-subset-iff [iff]:
  (greaterThan x ⊆ greaterThan y) = (y ≤ (x::'a::linorder))
apply (auto simp add: greaterThan-def)
apply (subst linorder-not-less [symmetric], blast)
done

lemma greaterThan-eq-iff [iff]:
  (greaterThan x = greaterThan y) = (x = (y::'a::linorder))
apply (rule iffI)
apply (erule equalityE)
apply simp-all
done

lemma atMost-subset-iff [iff]: (atMost x ⊆ atMost y) = (x ≤ (y::'a::order))
by (blast intro: order-trans)

lemma atMost-eq-iff [iff]: (atMost x = atMost y) = (x = (y::'a::linorder))

```

**by** (*blast intro: order-antisym order-trans*)

**lemma** *lessThan-subset-iff* [*iff*]:

$(\text{lessThan } x \subseteq \text{lessThan } y) = (x \leq (y::'a::\text{linorder}))$

**apply** (*auto simp add: lessThan-def*)

**apply** (*subst linorder-not-less [symmetric], blast*)

**done**

**lemma** *lessThan-eq-iff* [*iff*]:

$(\text{lessThan } x = \text{lessThan } y) = (x = (y::'a::\text{linorder}))$

**apply** (*rule iffI*)

**apply** (*erule equalityE*)

**apply** *simp-all*

**done**

### 34.3 Two-sided intervals

**context** *ord*

**begin**

**lemma** *greaterThanLessThan-iff* [*simp, noatp*]:

$(i : \{l <..<u\}) = (l < i \ \& \ i < u)$

**by** (*simp add: greaterThanLessThan-def*)

**lemma** *atLeastLessThan-iff* [*simp, noatp*]:

$(i : \{l..<u\}) = (l \leq i \ \& \ i < u)$

**by** (*simp add: atLeastLessThan-def*)

**lemma** *greaterThanAtMost-iff* [*simp, noatp*]:

$(i : \{l <..u\}) = (l < i \ \& \ i \leq u)$

**by** (*simp add: greaterThanAtMost-def*)

**lemma** *atLeastAtMost-iff* [*simp, noatp*]:

$(i : \{l..u\}) = (l \leq i \ \& \ i \leq u)$

**by** (*simp add: atLeastAtMost-def*)

The above four lemmas could be declared as *iffs*. If we do so, a call to *blast* in Hyperreal/Star.ML, lemma *STAR-Int* seems to take forever (more than one hour).

**end**

#### 34.3.1 Emptiness and singletons

**context** *order*

**begin**

**lemma** *atLeastAtMost-empty* [*simp*]:  $n < m \implies \{m..n\} = \{\}$

**by** (*auto simp add: atLeastAtMost-def atMost-def atLeast-def*)

**lemma** *atLeastLessThan-empty*[simp]:  $n \leq m \implies \{m..<n\} = \{\}$   
**by** (auto simp add: *atLeastLessThan-def*)

**lemma** *greaterThanAtMost-empty*[simp]:  $l \leq k \implies \{k<..l\} = \{\}$   
**by**(auto simp:*greaterThanAtMost-def greaterThan-def atMost-def*)

**lemma** *greaterThanLessThan-empty*[simp]:  $l \leq k \implies \{k<..  
**by**(auto simp:*greaterThanLessThan-def greaterThan-def lessThan-def*)$

**lemma** *atLeastAtMost-singleton* [simp]:  $\{a..a\} = \{a\}$   
**by** (auto simp add: *atLeastAtMost-def atMost-def atLeast-def*)

**end**

### 34.4 Intervals of natural numbers

#### 34.4.1 The Constant *lessThan*

**lemma** *lessThan-0* [simp]:  $\text{lessThan } (0::\text{nat}) = \{\}$   
**by** (simp add: *lessThan-def*)

**lemma** *lessThan-Suc*:  $\text{lessThan } (\text{Suc } k) = \text{insert } k (\text{lessThan } k)$   
**by** (simp add: *lessThan-def less-Suc-eq, blast*)

**lemma** *lessThan-Suc-atMost*:  $\text{lessThan } (\text{Suc } k) = \text{atMost } k$   
**by** (simp add: *lessThan-def atMost-def less-Suc-eq-le*)

**lemma** *UN-lessThan-UNIV*:  $(UN\ m::\text{nat}.\ \text{lessThan } m) = UNIV$   
**by** *blast*

#### 34.4.2 The Constant *greaterThan*

**lemma** *greaterThan-0* [simp]:  $\text{greaterThan } 0 = \text{range } \text{Suc}$   
**apply** (simp add: *greaterThan-def*)  
**apply** (*blast dest: gr0-conv-Suc [THEN iffD1]*)  
**done**

**lemma** *greaterThan-Suc*:  $\text{greaterThan } (\text{Suc } k) = \text{greaterThan } k - \{\text{Suc } k\}$   
**apply** (simp add: *greaterThan-def*)  
**apply** (auto elim: *linorder-neqE*)  
**done**

**lemma** *INT-greaterThan-UNIV*:  $(INT\ m::\text{nat}.\ \text{greaterThan } m) = \{\}$   
**by** *blast*

#### 34.4.3 The Constant *atLeast*

**lemma** *atLeast-0* [simp]:  $\text{atLeast } (0::\text{nat}) = UNIV$   
**by** (*unfold atLeast-def UNIV-def, simp*)

```

lemma atLeast-Suc: atLeast (Suc k) = atLeast k - {k}
apply (simp add: atLeast-def)
apply (simp add: Suc-le-eq)
apply (simp add: order-le-less, blast)
done

```

```

lemma atLeast-Suc-greaterThan: atLeast (Suc k) = greaterThan k
  by (auto simp add: greaterThan-def atLeast-def less-Suc-eq-le)

```

```

lemma UN-atLeast-UNIV: (UN m::nat. atLeast m) = UNIV
by blast

```

#### 34.4.4 The Constant *atMost*

```

lemma atMost-0 [simp]: atMost (0::nat) = {0}
by (simp add: atMost-def)

```

```

lemma atMost-Suc: atMost (Suc k) = insert (Suc k) (atMost k)
apply (simp add: atMost-def)
apply (simp add: less-Suc-eq order-le-less, blast)
done

```

```

lemma UN-atMost-UNIV: (UN m::nat. atMost m) = UNIV
by blast

```

#### 34.4.5 The Constant *atLeastLessThan*

The orientation of the following 2 rules is tricky. The lhs is defined in terms of the rhs. Hence the chosen orientation makes sense in this theory — the reverse orientation complicates proofs (eg nontermination). But outside, when the definition of the lhs is rarely used, the opposite orientation seems preferable because it reduces a specific concept to a more general one.

```

lemma atLeast0LessThan: {0::nat..n} = {..n}
by (simp add: lessThan-def atLeastLessThan-def)

```

```

lemma atLeast0AtMost: {0..n::nat} = {..n}
by (simp add: atMost-def atLeastAtMost-def)

```

```

declare atLeast0LessThan[symmetric, code unfold]
          atLeast0AtMost[symmetric, code unfold]

```

```

lemma atLeastLessThan0: {m..n::nat} = {}
by (simp add: atLeastLessThan-def)

```

#### 34.4.6 Intervals of nats with *Suc*

Not a simprule because the RHS is too messy.

```

lemma atLeastLessThanSuc:

```

$\{m..<Suc\ n\} = (if\ m \leq n\ then\ insert\ n\ \{m..<n\}\ else\ \{\})$   
**by** (*auto simp add: atLeastLessThan-def*)

**lemma** *atLeastLessThan-singleton [simp]*:  $\{m..<Suc\ m\} = \{m\}$   
**by** (*auto simp add: atLeastLessThan-def*)

**lemma** *atLeastLessThanSuc-atLeastAtMost*:  $\{l..<Suc\ u\} = \{l..u\}$   
**by** (*simp add: lessThan-Suc-atMost atLeastAtMost-def atLeastLessThan-def*)

**lemma** *atLeastSucAtMost-greaterThanAtMost*:  $\{Suc\ l..u\} = \{l<..u\}$   
**by** (*simp add: atLeast-Suc-greaterThan atLeastAtMost-def greaterThanAtMost-def*)

**lemma** *atLeastSucLessThan-greaterThanLessThan*:  $\{Suc\ l..<u\} = \{l<..<u\}$   
**by** (*simp add: atLeast-Suc-greaterThan atLeastLessThan-def greaterThanLessThan-def*)

**lemma** *atLeastAtMostSuc-conv*:  $m \leq Suc\ n \implies \{m..Suc\ n\} = insert\ (Suc\ n)\ \{m..n\}$   
**by** (*auto simp add: atLeastAtMost-def*)

### 34.4.7 Image

**lemma** *image-add-atLeastAtMost*:  
 $(\%n::nat. n+k) ' \{i..j\} = \{i+k..j+k\}$  (**is**  $?A = ?B$ )

**proof**

**show**  $?A \subseteq ?B$  **by** *auto*

**next**

**show**  $?B \subseteq ?A$

**proof**

**fix**  $n$  **assume**  $a: n : ?B$

**hence**  $n - k : \{i..j\}$  **by** *auto*

**moreover** **have**  $n = (n - k) + k$  **using**  $a$  **by** *auto*

**ultimately** **show**  $n : ?A$  **by** *blast*

**qed**

**qed**

**lemma** *image-add-atLeastLessThan*:  
 $(\%n::nat. n+k) ' \{i..<j\} = \{i+k..<j+k\}$  (**is**  $?A = ?B$ )

**proof**

**show**  $?A \subseteq ?B$  **by** *auto*

**next**

**show**  $?B \subseteq ?A$

**proof**

**fix**  $n$  **assume**  $a: n : ?B$

**hence**  $n - k : \{i..<j\}$  **by** *auto*

**moreover** **have**  $n = (n - k) + k$  **using**  $a$  **by** *auto*

**ultimately** **show**  $n : ?A$  **by** *blast*

**qed**

qed

**corollary** *image-Suc-atLeastAtMost*[simp]:

*Suc* ‘  $\{i..j\} = \{Suc\ i..Suc\ j\}$

**using** *image-add-atLeastAtMost*[**where**  $k = Suc\ 0$ ] **by** *simp*

**corollary** *image-Suc-atLeastLessThan*[simp]:

*Suc* ‘  $\{i..<j\} = \{Suc\ i..<Suc\ j\}$

**using** *image-add-atLeastLessThan*[**where**  $k = Suc\ 0$ ] **by** *simp*

**lemma** *image-add-int-atLeastLessThan*:

(%*x*.  $x + (l::int)$ ) ‘  $\{0..<u-l\} = \{l..<u\}$

**apply** (*auto simp add: image-def*)

**apply** (*rule-tac*  $x = x - l$  **in** *beXI*)

**apply** *auto*

**done**

#### 34.4.8 Finiteness

**lemma** *finite-lessThan* [*iff*]: **fixes**  $k :: nat$  **shows** *finite*  $\{..<k\}$

**by** (*induct*  $k$ ) (*simp-all add: lessThan-Suc*)

**lemma** *finite-atMost* [*iff*]: **fixes**  $k :: nat$  **shows** *finite*  $\{..k\}$

**by** (*induct*  $k$ ) (*simp-all add: atMost-Suc*)

**lemma** *finite-greaterThanLessThan* [*iff*]:

**fixes**  $l :: nat$  **shows** *finite*  $\{l<..<u\}$

**by** (*simp add: greaterThanLessThan-def*)

**lemma** *finite-atLeastLessThan* [*iff*]:

**fixes**  $l :: nat$  **shows** *finite*  $\{l..<u\}$

**by** (*simp add: atLeastLessThan-def*)

**lemma** *finite-greaterThanAtMost* [*iff*]:

**fixes**  $l :: nat$  **shows** *finite*  $\{l<..u\}$

**by** (*simp add: greaterThanAtMost-def*)

**lemma** *finite-atLeastAtMost* [*iff*]:

**fixes**  $l :: nat$  **shows** *finite*  $\{l..u\}$

**by** (*simp add: atLeastAtMost-def*)

A bounded set of natural numbers is finite.

**lemma** *bounded-nat-set-is-finite*:

(*ALL*  $i:N$ .  $i < (n::nat)$ )  $\implies$  *finite*  $N$

**apply** (*rule finite-subset*)

**apply** (*rule-tac* [2] *finite-lessThan*, *auto*)

**done**

**lemma** *finite-less-ub*:



```

!!f::nat=>nat. (!!n. n ≤ f n) ==> finite {n. f n ≤ u}
by (rule-tac B={..u} in finite-subset, auto intro: order-trans)

```

Any subset of an interval of natural numbers the size of the subset is exactly that interval.

```

lemma subset-card-intvl-is-intvl:
  A ≤ {k.. $k + \text{card } A$ }  $\implies A = \{k.. $k + \text{card } A$ \} (is PROP ?P)
proof cases
  assume finite A
  thus PROP ?P
proof (induct A rule:finite-linorder-induct)
  case empty thus ?case by auto
next
  case (insert A b)
  moreover hence  $b \sim A$  by auto
  moreover have  $A \leq \{k.. $k + \text{card } A$ \}$  and  $b = k + \text{card } A$ 
    using  $\langle b \sim A \rangle$  insert by fastsimp+
  ultimately show ?case by auto
qed
next
  assume  $\sim$  finite A thus PROP ?P by simp
qed$ 
```

### 34.4.9 Cardinality

```

lemma card-lessThan [simp]: card {.. $u$ } =  $u$ 
by (induct u, simp-all add: lessThan-Suc)

```

```

lemma card-atMost [simp]: card {.. $u$ } =  $\text{Suc } u$ 
by (simp add: lessThan-Suc-atMost [THEN sym])

```

```

lemma card-atLeastLessThan [simp]: card  $\{l.. $u$ \}$  =  $u - l$ 
apply (subgoal-tac card  $\{l.. $u$ \}$  = card  $\{.. $u-l$ \}$ )
apply (erule ssubst, rule card-lessThan)
apply (subgoal-tac ( $\%x. x + l$ ) ‘  $\{.. $u-l$ \}$  =  $\{l.. $u$ \}$ )
apply (erule subst)
apply (rule card-image)
apply (simp add: inj-on-def)
apply (auto simp add: image-def atLeastLessThan-def lessThan-def)
apply (rule-tac  $x = x - l$  in exI)
apply arith
done

```

```

lemma card-atLeastAtMost [simp]: card  $\{l.. $u$ \}$  =  $\text{Suc } u - l$ 
by (subst atLeastLessThanSuc-atLeastAtMost [THEN sym], simp)

```

```

lemma card-greaterThanAtMost [simp]: card  $\{l<.. $u$ \}$  =  $u - l$ 
by (subst atLeastSucAtMost-greaterThanAtMost [THEN sym], simp)

```

**lemma** *card-greaterThanLessThan* [simp]:  $\text{card } \{l < .. < u\} = u - \text{Suc } l$   
**by** (subst *atLeastSucLessThan-greaterThanLessThan* [THEN sym], simp)

**lemma** *ex-bij-betw-nat-finite*:  
 $\text{finite } M \implies \exists h. \text{bij-betw } h \ \{0 .. < \text{card } M\} \ M$   
**apply** (drule *finite-imp-nat-seg-image-inj-on*)  
**apply** (auto simp: *atLeast0LessThan*[symmetric] *lessThan-def*[symmetric] *card-image*  
*bij-betw-def*)  
**done**

**lemma** *ex-bij-betw-finite-nat*:  
 $\text{finite } M \implies \exists h. \text{bij-betw } h \ M \ \{0 .. < \text{card } M\}$   
**by** (blast dest: *ex-bij-betw-nat-finite* *bij-betw-inv*)

### 34.5 Intervals of integers

**lemma** *atLeastLessThanPlusOne-atLeastAtMost-int*:  $\{l .. < u + 1\} = \{l .. (u :: \text{int})\}$   
**by** (auto simp add: *atLeastAtMost-def* *atLeastLessThan-def*)

**lemma** *atLeastPlusOneAtMost-greaterThanAtMost-int*:  $\{l + 1 .. u\} = \{l < .. (u :: \text{int})\}$   
**by** (auto simp add: *atLeastAtMost-def* *greaterThanAtMost-def*)

**lemma** *atLeastPlusOneLessThan-greaterThanLessThan-int*:  
 $\{l + 1 .. < u\} = \{l < .. < u :: \text{int}\}$   
**by** (auto simp add: *atLeastLessThan-def* *greaterThanLessThan-def*)

#### 34.5.1 Finiteness

**lemma** *image-atLeastZeroLessThan-int*:  $0 \leq u \implies$   
 $\{(0 :: \text{int}) .. < u\} = \text{int } ' \{.. < \text{nat } u\}$   
**apply** (unfold *image-def* *lessThan-def*)  
**apply** auto  
**apply** (rule-tac  $x = \text{nat } x$  in *exI*)  
**apply** (auto simp add: *zless-nat-conj* *zless-nat-eq-int-zless* [THEN sym])  
**done**

**lemma** *finite-atLeastZeroLessThan-int*:  $\text{finite } \{(0 :: \text{int}) .. < u\}$   
**apply** (case-tac  $0 \leq u$ )  
**apply** (subst *image-atLeastZeroLessThan-int*, assumption)  
**apply** (rule *finite-imageI*)  
**apply** auto  
**done**

**lemma** *finite-atLeastLessThan-int* [iff]:  $\text{finite } \{l .. < u :: \text{int}\}$   
**apply** (subgoal-tac ( $\%x. x + l$ ) ‘  $\{0 .. < u - l\} = \{l .. < u\}$  )  
**apply** (erule subst)  
**apply** (rule *finite-imageI*)  
**apply** (rule *finite-atLeastZeroLessThan-int*)  
**apply** (rule *image-add-int-atLeastLessThan*)

done

**lemma** *finite-atLeastAtMost-int* [iff]: *finite*  $\{l..(u::int)\}$   
**by** (*subst atLeastLessThanPlusOne-atLeastAtMost-int [THEN sym], simp*)

**lemma** *finite-greaterThanAtMost-int* [iff]: *finite*  $\{l<..(u::int)\}$   
**by** (*subst atLeastPlusOneAtMost-greaterThanAtMost-int [THEN sym], simp*)

**lemma** *finite-greaterThanLessThan-int* [iff]: *finite*  $\{l<..u::int\}$   
**by** (*subst atLeastPlusOneLessThan-greaterThanLessThan-int [THEN sym], simp*)

### 34.5.2 Cardinality

**lemma** *card-atLeastZeroLessThan-int*: *card*  $\{(0::int)..<u\} = \text{nat } u$   
**apply** (*case-tac*  $0 \leq u$ )  
**apply** (*subst image-atLeastZeroLessThan-int, assumption*)  
**apply** (*subst card-image*)  
**apply** (*auto simp add: inj-on-def*)  
**done**

**lemma** *card-atLeastLessThan-int* [simp]: *card*  $\{l..u\} = \text{nat } (u - l)$   
**apply** (*subgoal-tac*  $\text{card } \{l..u\} = \text{card } \{0..u-l\}$ )  
**apply** (*erule ssubst, rule card-atLeastZeroLessThan-int*)  
**apply** (*subgoal-tac*  $(\%x. x + l) \text{ ‘ } \{0..u-l\} = \{l..u\}$ )  
**apply** (*erule ssubst*)  
**apply** (*rule card-image*)  
**apply** (*simp add: inj-on-def*)  
**apply** (*rule image-add-int-atLeastLessThan*)  
**done**

**lemma** *card-atLeastAtMost-int* [simp]: *card*  $\{l..u\} = \text{nat } (u - l + 1)$   
**apply** (*subst atLeastLessThanPlusOne-atLeastAtMost-int [THEN sym]*)  
**apply** (*auto simp add: algebra-simps*)  
**done**

**lemma** *card-greaterThanAtMost-int* [simp]: *card*  $\{l<..u\} = \text{nat } (u - l)$   
**by** (*subst atLeastPlusOneAtMost-greaterThanAtMost-int [THEN sym], simp*)

**lemma** *card-greaterThanLessThan-int* [simp]: *card*  $\{l<..u\} = \text{nat } (u - (l + 1))$   
**by** (*subst atLeastPlusOneLessThan-greaterThanLessThan-int [THEN sym], simp*)

**lemma** *finite-M-bounded-by-nat*: *finite*  $\{k. P \ k \wedge k < (i::nat)\}$   
**proof** –  
**have**  $\{k. P \ k \wedge k < i\} \subseteq \{..i\}$  **by** *auto*  
**with** *finite-lessThan[of i]* **show** ?thesis **by** (*simp add: finite-subset*)  
**qed**

**lemma** *card-less*:  
**assumes** *zero-in-M*:  $0 \in M$

**shows**  $\text{card } \{k \in M. k < \text{Suc } i\} \neq 0$

**proof** –

**from** *zero-in-M* **have**  $\{k \in M. k < \text{Suc } i\} \neq \{\}$  **by** *auto*

**with** *finite-M-bounded-by-nat* **show** *?thesis* **by** (*auto simp add: card-eq-0-iff*)

**qed**

**lemma** *card-less-Suc2*:  $0 \notin M \implies \text{card } \{k. \text{Suc } k \in M \wedge k < i\} = \text{card } \{k \in M. k < \text{Suc } i\}$

**apply** (*rule card-bij-eq [of Suc - -  $\lambda x. x - \text{Suc } 0$ ]*)

**apply** *simp*

**apply** *fastsimp*

**apply** *auto*

**apply** (*rule inj-on-diff-nat*)

**apply** *auto*

**apply** (*case-tac x*)

**apply** *auto*

**apply** (*case-tac xa*)

**apply** *auto*

**apply** (*case-tac xa*)

**apply** *auto*

**done**

**lemma** *card-less-Suc*:

**assumes** *zero-in-M*:  $0 \in M$

**shows**  $\text{Suc } (\text{card } \{k. \text{Suc } k \in M \wedge k < i\}) = \text{card } \{k \in M. k < \text{Suc } i\}$

**proof** –

**from** *assms* **have**  $0 \in \{k \in M. k < \text{Suc } i\}$  **by** *simp*

**hence**  $c: \{k \in M. k < \text{Suc } i\} = \text{insert } 0 (\{k \in M. k < \text{Suc } i\} - \{0\})$

**by** (*auto simp only: insert-Diff*)

**have**  $b: \{k \in M. k < \text{Suc } i\} - \{0\} = \{k \in M - \{0\}. k < \text{Suc } i\}$  **by** *auto*

**from** *finite-M-bounded-by-nat* [*of  $\lambda x. x \in M \text{ Suc } i$* ] **have**  $\text{Suc } (\text{card } \{k. \text{Suc } k \in M \wedge k < i\}) = \text{card } (\text{insert } 0 (\{k \in M. k < \text{Suc } i\} - \{0\}))$

**apply** (*subst card-insert*)

**apply** *simp-all*

**apply** (*subst b*)

**apply** (*subst card-less-Suc2[symmetric]*)

**apply** *simp-all*

**done**

**with**  $c$  **show** *?thesis* **by** *simp*

**qed**

## 34.6 Lemmas useful with the summation operator *setsum*

For examples, see *Algebra/poly/UnivPoly2.thy*

### 34.6.1 Disjoint Unions

Singletons and open intervals

**lemma** *ivl-disj-un-singleton*:

$$\begin{aligned} & \{l::'a::\text{linorder}\} \text{ Un } \{l<..\} = \{l..\} \\ & \{..\} \text{ Un } \{u::'a::\text{linorder}\} = \{..u\} \\ & (l::'a::\text{linorder}) < u ==> \{l\} \text{ Un } \{l<..\} = \{l..\} \\ & (l::'a::\text{linorder}) < u ==> \{l<..\} \text{ Un } \{u\} = \{l<..u\} \\ & (l::'a::\text{linorder}) <= u ==> \{l\} \text{ Un } \{l<..u\} = \{l..u\} \\ & (l::'a::\text{linorder}) <= u ==> \{l..\} \text{ Un } \{u\} = \{l..u\} \end{aligned}$$

**by** *auto*

One- and two-sided intervals

**lemma** *ivl-disj-un-one*:

$$\begin{aligned} & (l::'a::\text{linorder}) < u ==> \{..l\} \text{ Un } \{l<..\} = \{..\} \\ & (l::'a::\text{linorder}) <= u ==> \{..\} = \{..\} \\ & (l::'a::\text{linorder}) <= u ==> \{..l\} \text{ Un } \{l<..u\} = \{..u\} \\ & (l::'a::\text{linorder}) <= u ==> \{..\} \text{ Un } \{u..\} = \{l<..\} \\ & (l::'a::\text{linorder}) <= u ==> \{l..u\} \text{ Un } \{u<..\} = \{l..\} \\ & (l::'a::\text{linorder}) <= u ==> \{l..\} \text{ Un } \{u..\} = \{l..\} \end{aligned}$$

**by** *auto*

Two- and two-sided intervals

**lemma** *ivl-disj-un-two*:

$$\begin{aligned} & [(l::'a::\text{linorder}) < m; m <= u] ==> \{l<..\} \text{ Un } \{m..\} = \{l<..\} \\ & [(l::'a::\text{linorder}) <= m; m < u] ==> \{l<..m\} \text{ Un } \{m<..\} = \{l<..\} \\ & [(l::'a::\text{linorder}) <= m; m <= u] ==> \{l..\} = \{l..\} \\ & [(l::'a::\text{linorder}) <= m; m < u] ==> \{l..m\} \text{ Un } \{m<..\} = \{l<..\} \\ & [(l::'a::\text{linorder}) < m; m <= u] ==> \{l<..$$

**by** *auto*

**lemmas** *ivl-disj-un* = *ivl-disj-un-singleton* *ivl-disj-un-one* *ivl-disj-un-two*

### 34.6.2 Disjoint Intersections

Singletons and open intervals

**lemma** *ivl-disj-int-singleton*:

$$\begin{aligned} & \{l::'a::\text{order}\} \text{ Int } \{l<..\} = \{\} \\ & \{..\} \text{ Int } \{u\} = \{\} \\ & \{l\} \text{ Int } \{l<..\} = \{\} \\ & \{l<..\} \text{ Int } \{u\} = \{\} \\ & \{l\} \text{ Int } \{l<..u\} = \{\} \\ & \{l..\} \text{ Int } \{u\} = \{\} \end{aligned}$$

**by** *simp+*

One- and two-sided intervals

**lemma** *ivl-disj-int-one*:  
 $\{..l::'a::order\} \text{ Int } \{l<..\} = \{\}$   
 $\{..\} \text{ Int } \{l<..\} = \{\}$   
 $\{..l\} \text{ Int } \{l<..\} = \{\}$   
 $\{..\} \text{ Int } \{l..u\} = \{\}$   
 $\{l<..\} \text{ Int } \{u<..\} = \{\}$   
 $\{l<..\} \text{ Int } \{u..\} = \{\}$   
 $\{l..u\} \text{ Int } \{u<..\} = \{\}$   
 $\{l..u\} \text{ Int } \{u..\} = \{\}$   
**by** *auto*

Two- and two-sided intervals

**lemma** *ivl-disj-int-two*:  
 $\{l::'a::order<..\} \text{ Int } \{m..\} = \{\}$   
 $\{l<..\} \text{ Int } \{m<..\} = \{\}$   
 $\{l..\} \text{ Int } \{m..\} = \{\}$   
 $\{l..m\} \text{ Int } \{m<..\} = \{\}$   
 $\{l<..\} \text{ Int } \{m..u\} = \{\}$   
 $\{l<..\} \text{ Int } \{m<..\} = \{\}$   
 $\{l..\} \text{ Int } \{m..u\} = \{\}$   
 $\{l..m\} \text{ Int } \{m<..\} = \{\}$   
**by** *auto*

**lemmas** *ivl-disj-int* = *ivl-disj-int-singleton ivl-disj-int-one ivl-disj-int-two*

### 34.6.3 Some Differences

**lemma** *ivl-diff[simp]*:  
 $i \leq n \implies \{i..\} - \{i..\} = \{n..\}$   
**by** (*auto*)

### 34.6.4 Some Subset Conditions

**lemma** *ivl-subset [simp,noatp]*:  
 $(\{i..\} \subseteq \{m..\}) = (j \leq i \mid m \leq i \ \& \ j \leq (n::'a::linorder))$   
**apply** (*auto simp:linorder-not-le*)  
**apply** (*rule ccontr*)  
**apply** (*insert linorder-le-less-linear[of i n]*)  
**apply** (*clarsimp simp:linorder-not-le*)  
**apply** (*fastsimp*)  
**done**

## 34.7 Summation indexed over intervals

**syntax**

*-from-to-setsum* :: *idt*  $\Rightarrow$  *'a*  $\Rightarrow$  *'a*  $\Rightarrow$  *'b*  $\Rightarrow$  *'b* ((*SUM* - = -../ -) [*0,0,0,10*] *10*)  
*-from-upto-setsum* :: *idt*  $\Rightarrow$  *'a*  $\Rightarrow$  *'a*  $\Rightarrow$  *'b*  $\Rightarrow$  *'b* ((*SUM* - = -..<./ -) [*0,0,0,10*] *10*)  
*-upt-setsum* :: *idt*  $\Rightarrow$  *'a*  $\Rightarrow$  *'b*  $\Rightarrow$  *'b* ((*SUM* -<./ -) [*0,0,10*] *10*)  
*-upto-setsum* :: *idt*  $\Rightarrow$  *'a*  $\Rightarrow$  *'b*  $\Rightarrow$  *'b* ((*SUM* -<=./ -) [*0,0,10*] *10*)

**syntax** (*xsymbols*)

-from-to-setsum ::  $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{I}\sum - = \dots / -) [0,0,0,10] \ 10)$   
 -from-upto-setsum ::  $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{I}\sum - = \dots < / -) [0,0,0,10] \ 10)$   
 -upt-setsum ::  $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{I}\sum - < / -) [0,0,10] \ 10)$   
 -upto-setsum ::  $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{I}\sum - \leq / -) [0,0,10] \ 10)$

**syntax** (*HTML output*)

-from-to-setsum ::  $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{I}\sum - = \dots / -) [0,0,0,10] \ 10)$   
 -from-upto-setsum ::  $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{I}\sum - = \dots < / -) [0,0,0,10] \ 10)$   
 -upt-setsum ::  $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{I}\sum - < / -) [0,0,10] \ 10)$   
 -upto-setsum ::  $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{I}\sum - \leq / -) [0,0,10] \ 10)$

**syntax** (*latex-sum output*)

-from-to-setsum ::  $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$   
 $((\mathcal{I}\sum - = -) [0,0,0,10] \ 10)$   
 -from-upto-setsum ::  $idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$   
 $((\mathcal{I}\sum - < -) [0,0,0,10] \ 10)$   
 -upt-setsum ::  $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$   
 $((\mathcal{I}\sum - < -) [0,0,10] \ 10)$   
 -upto-setsum ::  $idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$   
 $((\mathcal{I}\sum - \leq -) [0,0,10] \ 10)$

**translations**

$\sum x=a..b. t == \text{CONST setsum } (\%x. t) \{a..b\}$   
 $\sum x=a..<b. t == \text{CONST setsum } (\%x. t) \{a..<b\}$   
 $\sum i \leq n. t == \text{CONST setsum } (\lambda i. t) \{..n\}$   
 $\sum i < n. t == \text{CONST setsum } (\lambda i. t) \{..<n\}$

The above introduces some pretty alternative syntaxes for summation over intervals:

Old	New	L <sup>A</sup> T <sub>E</sub> X
$\sum x \in \{a..b\}. e$	$\sum x = a..b. e$	$\sum_x^b =_a e$
$\sum x \in \{a..<b\}. e$	$\sum x = a..<b. e$	$\sum_x^{<b} =_a e$
$\sum x \in \{..b\}. e$	$\sum x \leq b. e$	$\sum_x \leq_b e$
$\sum x \in \{..<b\}. e$	$\sum x < b. e$	$\sum_x <_b e$

The left column shows the term before introduction of the new syntax, the middle column shows the new (default) syntax, and the right column shows a special syntax. The latter is only meaningful for latex output and has to be activated explicitly by setting the print mode to *latex-sum* (e.g. via *mode = latex-sum* in antiquotations). It is not the default L<sup>A</sup>T<sub>E</sub>X output because it only works well with italic-style formulae, not tt-style.

Note that for uniformity on *nat* it is better to use  $\sum x = 0..<n. e$  rather than  $\sum x < n. e$ : *setsum* may not provide all lemmas available for  $\{m..<n\}$  also in the special form for  $\{..<n\}$ .

This congruence rule should be used for sums over intervals as the standard

theorem *setsum-cong* does not work well with the simplifier who adds the unsimplified premise  $x \in B$  to the context.

**lemma** *setsum-ivl-cong*:

$\llbracket a = c; b = d; !!x. \llbracket c \leq x; x < d \rrbracket \implies f x = g x \rrbracket \implies$   
 $\text{setsum } f \{a..<b\} = \text{setsum } g \{c..<d\}$   
**by** (*rule setsum-cong, simp-all*)

**lemma** *setsum-atMost-Suc[simp]*:  $(\sum i \leq \text{Suc } n. f i) = (\sum i \leq n. f i) + f(\text{Suc } n)$   
**by** (*simp add: atMost-Suc add-ac*)

**lemma** *setsum-lessThan-Suc[simp]*:  $(\sum i < \text{Suc } n. f i) = (\sum i < n. f i) + f n$   
**by** (*simp add: lessThan-Suc add-ac*)

**lemma** *setsum-cl-ivl-Suc[simp]*:

$\text{setsum } f \{m.. \text{Suc } n\} = (\text{if } \text{Suc } n < m \text{ then } 0 \text{ else } \text{setsum } f \{m..n\} + f(\text{Suc } n))$   
**by** (*auto simp: add-ac atLeastAtMostSuc-conv*)

**lemma** *setsum-op-ivl-Suc[simp]*:

$\text{setsum } f \{m..<\text{Suc } n\} = (\text{if } n < m \text{ then } 0 \text{ else } \text{setsum } f \{m..<n\} + f(n))$   
**by** (*auto simp: add-ac atLeastLessThanSuc*)

**lemma** *setsum-head*:

**fixes**  $n :: \text{nat}$   
**assumes**  $mn: m \leq n$   
**shows**  $(\sum x \in \{m..n\}. P x) = P m + (\sum x \in \{m < .. n\}. P x)$  (**is**  $?lhs = ?rhs$ )  
**proof** –  
**from**  $mn$   
**have**  $\{m..n\} = \{m\} \cup \{m < .. n\}$   
**by** (*auto intro: ivl-disj-un-singleton*)  
**hence**  $?lhs = (\sum x \in \{m\} \cup \{m < .. n\}. P x)$   
**by** (*simp add: atLeast0LessThan*)  
**also have**  $\dots = ?rhs$  **by** *simp*  
**finally show**  $?thesis$  .  
**qed**

**lemma** *setsum-head-Suc*:

$m \leq n \implies \text{setsum } f \{m..n\} = f m + \text{setsum } f \{\text{Suc } m..n\}$   
**by** (*simp add: setsum-head atLeastSucAtMost-greaterThanAtMost*)

**lemma** *setsum-head-upt-Suc*:

$m < n \implies \text{setsum } f \{m..<n\} = f m + \text{setsum } f \{\text{Suc } m..<n\}$   
**apply** (*insert setsum-head-Suc[of m n – Suc 0 f]*)  
**apply** (*simp add: atLeastLessThanSuc-atLeastAtMost[symmetric] algebra-simps*)  
**done**



**lemma** *setsum-add-nat-ivl*:  $\llbracket m \leq n; n \leq p \rrbracket \implies$   
 $\text{setsum } f \{m..<n\} + \text{setsum } f \{n..<p\} = \text{setsum } f \{m..<p::\text{nat}\}$   
**by** (*simp add:setsum-Un-disjoint[symmetric] ivl-disj-int ivl-disj-un*)

**lemma** *setsum-diff-nat-ivl*:  
**fixes**  $f :: \text{nat} \Rightarrow 'a::\text{ab-group-add}$   
**shows**  $\llbracket m \leq n; n \leq p \rrbracket \implies$   
 $\text{setsum } f \{m..<p\} - \text{setsum } f \{m..<n\} = \text{setsum } f \{n..<p\}$   
**using** *setsum-add-nat-ivl [of m n p f, symmetric]*  
**apply** (*simp add: add-ac*)  
**done**

### 34.8 Shifting bounds

**lemma** *setsum-shift-bounds-nat-ivl*:  
 $\text{setsum } f \{m+k..<n+k\} = \text{setsum } (\%i. f(i+k))\{m..<n::\text{nat}\}$   
**by** (*induct n, auto simp:atLeastLessThanSuc*)

**lemma** *setsum-shift-bounds-cl-nat-ivl*:  
 $\text{setsum } f \{m+k..n+k\} = \text{setsum } (\%i. f(i+k))\{m..n::\text{nat}\}$   
**apply** (*insert setsum-reindex[OF inj-on-add-nat, where h=f and B = {m..n}])*  
**apply** (*simp add:image-add-atLeastAtMost o-def*)  
**done**

**corollary** *setsum-shift-bounds-cl-Suc-ivl*:  
 $\text{setsum } f \{\text{Suc } m..\text{Suc } n\} = \text{setsum } (\%i. f(\text{Suc } i))\{m..n\}$   
**by** (*simp add:setsum-shift-bounds-cl-nat-ivl[where k=Suc 0, simplified]*)

**corollary** *setsum-shift-bounds-Suc-ivl*:  
 $\text{setsum } f \{\text{Suc } m..<\text{Suc } n\} = \text{setsum } (\%i. f(\text{Suc } i))\{m..<n\}$   
**by** (*simp add:setsum-shift-bounds-nat-ivl[where k=Suc 0, simplified]*)

**lemma** *setsum-shift-lb-Suc0-0*:  
 $f(0::\text{nat}) = (0::\text{nat}) \implies \text{setsum } f \{\text{Suc } 0..k\} = \text{setsum } f \{0..k\}$   
**by**(*simp add:setsum-head-Suc*)

**lemma** *setsum-shift-lb-Suc0-0-upt*:  
 $f(0::\text{nat}) = 0 \implies \text{setsum } f \{\text{Suc } 0..<k\} = \text{setsum } f \{0..<k\}$   
**apply**(*cases k*)**apply** *simp*  
**apply**(*simp add:setsum-head-upt-Suc*)  
**done**

### 34.9 The formula for geometric sums

**lemma** *geometric-sum*:  
 $x \sim 1 \implies (\sum_{i=0..<n} x^i) =$   
 $(x^n - 1) / (x - 1::'a::\{\text{field}, \text{recpower}\})$   
**by** (*induct n*) (*simp-all add:field-simps power-Suc*)

### 34.10 The formula for arithmetic sums

**lemma** *gauss-sum*:

$$((1::'a::\text{comm-semiring-1}) + 1) * (\sum i \in \{1..n\}. \text{of-nat } i) = \text{of-nat } n * ((\text{of-nat } n) + 1)$$

**proof** (*induct n*)

case 0

show ?case by simp

next

case (*Suc n*)

then show ?case by (simp add: algebra-simps)

qed

**theorem** *arith-series-general*:

$$((1::'a::\text{comm-semiring-1}) + 1) * (\sum i \in \{..<n\}. a + \text{of-nat } i * d) = \text{of-nat } n * (a + (a + \text{of-nat}(n - 1) * d))$$

**proof** *cases*

assume *ngt1*:  $n > 1$

let ?*I* =  $\lambda i. \text{of-nat } i$  and ?*n* =  $\text{of-nat } n$

have

$$\begin{aligned} (\sum i \in \{..<n\}. a + ?I \ i * d) &= \\ ((\sum i \in \{..<n\}. a) + (\sum i \in \{..<n\}. ?I \ i * d)) & \\ \text{by (rule setsum-addf)} & \end{aligned}$$

also from *ngt1* have ... = ?*n*\**a* +  $(\sum i \in \{..<n\}. ?I \ i * d)$  by simp

also from *ngt1* have ... = (?*n*\**a* +  $d * (\sum i \in \{1..<n\}. ?I \ i)$ )

unfolding *One-nat-def*

by (simp add: setsum-right-distrib atLeast0LessThan[symmetric] setsum-shift-lb-Suc0-0-upt mult-ac)

also have  $(1+1)*... = (1+1)*?n*a + d*(1+1)*(\sum i \in \{1..<n\}. ?I \ i)$

by (simp add: left-distrib right-distrib)

also from *ngt1* have  $\{1..<n\} = \{1..n - 1\}$

by (cases *n*) (auto simp: atLeastLessThanSuc-atLeastAtMost)

also from *ngt1*

have  $(1+1)*?n*a + d*(1+1)*(\sum i \in \{1..n - 1\}. ?I \ i) = ((1+1)*?n*a + d*?I \ (n - 1)*?I \ n)$

by (simp only: mult-ac gauss-sum [of  $n - 1$ ], unfold *One-nat-def*)

(simp add: mult-ac trans [OF add-commute of-nat-Suc [symmetric]])

finally show ?thesis by (simp add: algebra-simps)

next

assume  $\neg(n > 1)$

hence  $n = 1 \vee n = 0$  by auto

thus ?thesis by (auto simp: algebra-simps)

qed

**lemma** *arith-series-nat*:

$$\text{Suc } (\text{Suc } 0) * (\sum i \in \{..<n\}. a + i * d) = n * (a + (a + (n - 1) * d))$$

**proof** –

have

$$\begin{aligned} ((1::\text{nat}) + 1) * (\sum i \in \{..<n::\text{nat}\}. a + \text{of-nat}(i) * d) &= \\ \text{of-nat}(n) * (a + (a + \text{of-nat}(n - 1) * d)) & \end{aligned}$$

```

    by (rule arith-series-general)
  thus ?thesis
    unfolding One-nat-def by (auto simp add: of-nat-id)
qed

```

**lemma** *arith-series-int*:

```

  (2::int) * (∑ i∈{.. $n$ }.  $a + \text{of-nat } i * d$ ) =
  of-nat  $n$  * ( $a + (a + \text{of-nat}(n - 1) * d)$ )
proof –
  have
    ((1::int) + 1) * (∑ i∈{.. $n$ }.  $a + \text{of-nat } i * d$ ) =
    of-nat( $n$ ) * ( $a + (a + \text{of-nat}(n - 1) * d)$ )
    by (rule arith-series-general)
  thus ?thesis by simp
qed

```

**lemma** *sum-diff-distrib*:

```

  fixes  $P::\text{nat} \Rightarrow \text{nat}$ 
  shows
     $\forall x. Q\ x \leq P\ x \implies$ 
     $(\sum x < n. P\ x) - (\sum x < n. Q\ x) = (\sum x < n. P\ x - Q\ x)$ 
proof (induct  $n$ )
  case 0 show ?case by simp
next
  case (Suc  $n$ )

```

```

  let ?lhs = ( $\sum x < n. P\ x$ ) - ( $\sum x < n. Q\ x$ )
  let ?rhs =  $\sum x < n. P\ x - Q\ x$ 

```

```

  from Suc have ?lhs = ?rhs by simp

```

**moreover**

```

  from Suc have ?lhs +  $P\ n - Q\ n = ?rhs + (P\ n - Q\ n)$  by simp

```

**moreover**

**from** *Suc* **have**

```

  ( $\sum x < n. P\ x$ ) +  $P\ n - ((\sum x < n. Q\ x) + Q\ n) = ?rhs + (P\ n - Q\ n)$ 

```

```

  by (subst diff-diff-left[symmetric],

```

```

    subst diff-add-assoc2)

```

```

    (auto simp: diff-add-assoc2 intro: setsum-mono)

```

**ultimately**

```

  show ?case by simp

```

**qed**

### 34.11 Products indexed over intervals

**syntax**

```

-from-to-setprod ::  $\text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$  ( $((\text{PROD } - = \dots / -) [0,0,0,10] 10)$ )
-from-upto-setprod ::  $\text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$  ( $((\text{PROD } - = \dots < - / -) [0,0,0,10] 10)$ )
-upt-setprod ::  $\text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$  ( $((\text{PROD } - < - / -) [0,0,10] 10)$ )

```

```

-upto-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((PROD -<= - / -) [0,0,10] 10)
syntax (xsymbols)
  -from-to-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((∑Π - = -..- / -) [0,0,0,10] 10)
  -from-upto-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((∑Π - = -..<- / -) [0,0,0,10]
10)
  -upt-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((∑Π -<- / -) [0,0,10] 10)
  -upto-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((∑Π -≤- / -) [0,0,10] 10)
syntax (HTML output)
  -from-to-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((∑Π - = -..- / -) [0,0,0,10] 10)
  -from-upto-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((∑Π - = -..<- / -) [0,0,0,10]
10)
  -upt-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((∑Π -<- / -) [0,0,10] 10)
  -upto-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((∑Π -≤- / -) [0,0,10] 10)
syntax (latex-prod output)
  -from-to-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b
  ((∑Π - = - -) [0,0,0,10] 10)
  -from-upto-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b
  ((∑Π - ≤ - -) [0,0,0,10] 10)
  -upt-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b
  ((∑Π - < - -) [0,0,10] 10)
  -upto-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b
  ((∑Π - ≤ - -) [0,0,10] 10)

translations
  Π x=a..b. t == CONST setprod (%x. t) {a..b}
  Π x=a..<b. t == CONST setprod (%x. t) {a..<b}
  Π i≤n. t == CONST setprod (λi. t) {..n}
  Π i<n. t == CONST setprod (λi. t) {..<n}

end

```

## 35 Presburger: Decision Procedure for Presburger Arithmetic

```

theory Presburger
imports Groebner-Basis SetInterval
uses
  Tools/Qelim/qelim.ML
  Tools/Qelim/cooper-data.ML
  Tools/Qelim/generated-cooper.ML
  (Tools/Qelim/cooper.ML)
  (Tools/Qelim/presburger.ML)
begin

setup CooperData.setup

```

### 35.1 The $-\infty$ and $+\infty$ Properties

**lemma** *minf*:

$$\begin{aligned} & \llbracket \exists (z :: 'a :: \text{linorder}). \forall x < z. P\ x = P'\ x; \exists z. \forall x < z. Q\ x = Q'\ x \rrbracket \\ & \implies \exists z. \forall x < z. (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x) \\ & \llbracket \exists (z :: 'a :: \text{linorder}). \forall x < z. P\ x = P'\ x; \exists z. \forall x < z. Q\ x = Q'\ x \rrbracket \\ & \implies \exists z. \forall x < z. (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x) \\ & \exists (z :: 'a :: \{\text{linorder}\}). \forall x < z. (x = t) = \text{False} \\ & \exists (z :: 'a :: \{\text{linorder}\}). \forall x < z. (x \neq t) = \text{True} \\ & \exists (z :: 'a :: \{\text{linorder}\}). \forall x < z. (x < t) = \text{True} \\ & \exists (z :: 'a :: \{\text{linorder}\}). \forall x < z. (x \leq t) = \text{True} \\ & \exists (z :: 'a :: \{\text{linorder}\}). \forall x < z. (x > t) = \text{False} \\ & \exists (z :: 'a :: \{\text{linorder}\}). \forall x < z. (x \geq t) = \text{False} \\ & \exists z. \forall (x :: 'a :: \{\text{linorder}, \text{plus}, \text{Ring-and-Field.dvd}\}) < z. (d\ \text{dvd}\ x + s) = (d\ \text{dvd}\ x + s) \\ & \exists z. \forall (x :: 'a :: \{\text{linorder}, \text{plus}, \text{Ring-and-Field.dvd}\}) < z. (\neg d\ \text{dvd}\ x + s) = (\neg d\ \text{dvd}\ x + s) \\ & \exists z. \forall x < z. F = F \\ & \text{by } ((\text{erule } \text{exE}, \text{erule } \text{exE}, \text{rule-tac } x = \min\ z\ \text{za in } \text{exI}, \text{simp})+, (\text{rule-tac } x = t \text{ in } \text{exI}, \text{fastsimp})+) \text{ simp-all} \end{aligned}$$

**lemma** *pinf*:

$$\begin{aligned} & \llbracket \exists (z :: 'a :: \text{linorder}). \forall x > z. P\ x = P'\ x; \exists z. \forall x > z. Q\ x = Q'\ x \rrbracket \\ & \implies \exists z. \forall x > z. (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x) \\ & \llbracket \exists (z :: 'a :: \text{linorder}). \forall x > z. P\ x = P'\ x; \exists z. \forall x > z. Q\ x = Q'\ x \rrbracket \\ & \implies \exists z. \forall x > z. (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x) \\ & \exists (z :: 'a :: \{\text{linorder}\}). \forall x > z. (x = t) = \text{False} \\ & \exists (z :: 'a :: \{\text{linorder}\}). \forall x > z. (x \neq t) = \text{True} \\ & \exists (z :: 'a :: \{\text{linorder}\}). \forall x > z. (x < t) = \text{False} \\ & \exists (z :: 'a :: \{\text{linorder}\}). \forall x > z. (x \leq t) = \text{False} \\ & \exists (z :: 'a :: \{\text{linorder}\}). \forall x > z. (x > t) = \text{True} \\ & \exists (z :: 'a :: \{\text{linorder}\}). \forall x > z. (x \geq t) = \text{True} \\ & \exists z. \forall (x :: 'a :: \{\text{linorder}, \text{plus}, \text{Ring-and-Field.dvd}\}) > z. (d\ \text{dvd}\ x + s) = (d\ \text{dvd}\ x + s) \\ & \exists z. \forall (x :: 'a :: \{\text{linorder}, \text{plus}, \text{Ring-and-Field.dvd}\}) > z. (\neg d\ \text{dvd}\ x + s) = (\neg d\ \text{dvd}\ x + s) \\ & \exists z. \forall x > z. F = F \\ & \text{by } ((\text{erule } \text{exE}, \text{erule } \text{exE}, \text{rule-tac } x = \max\ z\ \text{za in } \text{exI}, \text{simp})+, (\text{rule-tac } x = t \text{ in } \text{exI}, \text{fastsimp})+) \text{ simp-all} \end{aligned}$$

**lemma** *inf-period*:

$$\begin{aligned} & \llbracket \forall x\ k. P\ x = P\ (x - k * D); \forall x\ k. Q\ x = Q\ (x - k * D) \rrbracket \\ & \implies \forall x\ k. (P\ x \wedge Q\ x) = (P\ (x - k * D) \wedge Q\ (x - k * D)) \\ & \llbracket \forall x\ k. P\ x = P\ (x - k * D); \forall x\ k. Q\ x = Q\ (x - k * D) \rrbracket \\ & \implies \forall x\ k. (P\ x \vee Q\ x) = (P\ (x - k * D) \vee Q\ (x - k * D)) \\ & (d :: 'a :: \{\text{comm-ring}, \text{Ring-and-Field.dvd}\})\ \text{dvd}\ D \implies \forall x\ k. (d\ \text{dvd}\ x + t) = (d\ \text{dvd}\ (x - k * D) + t) \\ & (d :: 'a :: \{\text{comm-ring}, \text{Ring-and-Field.dvd}\})\ \text{dvd}\ D \implies \forall x\ k. (\neg d\ \text{dvd}\ x + t) = (\neg d\ \text{dvd}\ (x - k * D) + t) \\ & \forall x\ k. F = F \end{aligned}$$

**apply** (*auto elim!*: *dvdE simp add: algebra-simps*)  
**unfolding** *mult-assoc* [*symmetric*] *left-distrib* [*symmetric*] *left-diff-distrib* [*symmetric*]  
**unfolding** *dvd-def mult-commute* [*of d*]  
**by** *auto*

### 35.2 The A and B sets

**lemma** *bset*:

$$\begin{aligned} & \llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow P\ x \longrightarrow P(x - D) ; \\ & \quad \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow Q\ x \longrightarrow Q(x - D) \rrbracket \Longrightarrow \\ & \quad \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (P\ x \wedge Q\ x) \longrightarrow (P(x - D) \wedge Q(x - D)) \\ & \llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow P\ x \longrightarrow P(x - D) ; \\ & \quad \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow Q\ x \longrightarrow Q(x - D) \rrbracket \Longrightarrow \\ & \quad \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (P\ x \vee Q\ x) \longrightarrow (P(x - D) \vee Q(x - D)) \\ & \llbracket D > 0 ; t - 1 \in B \rrbracket \Longrightarrow (\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x = t) \longrightarrow (x - D = t)) \\ & \llbracket D > 0 ; t \in B \rrbracket \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \neq t) \longrightarrow (x - D \neq t)) \\ & D > 0 \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x < t) \longrightarrow (x - D < t)) \\ & D > 0 \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \leq t) \longrightarrow (x - D \leq t)) \\ & \llbracket D > 0 ; t \in B \rrbracket \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x > t) \longrightarrow (x - D > t)) \\ & \llbracket D > 0 ; t - 1 \in B \rrbracket \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \geq t) \longrightarrow (x - D \geq t)) \\ & d\ dvd\ D \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (d\ dvd\ x+t) \longrightarrow (d\ dvd\ (x - D) + t)) \\ & d\ dvd\ D \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (\neg d\ dvd\ x+t) \longrightarrow (\neg d\ dvd\ (x - D) + t)) \\ & \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow F \longrightarrow F \end{aligned}$$

**proof** (*blast, blast*)

**assume** *dp*:  $D > 0$  **and** *tB*:  $t - 1 \in B$

**show**  $(\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x = t) \longrightarrow (x - D = t))$

**apply** (*rule allI, rule impI,erule ballE*[**where**  $x=1$ ],*erule ballE*[**where**  $x=t-1$ ])

**apply** *algebra using dp tB by simp-all*

**next**

**assume** *dp*:  $D > 0$  **and** *tB*:  $t \in B$

**show**  $(\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \neq t) \longrightarrow (x - D \neq t))$

**apply** (*rule allI, rule impI,erule ballE*[**where**  $x=D$ ],*erule ballE*[**where**  $x=t$ ])

**apply** *algebra*

**using** *dp tB by simp-all*

**next**

**assume** *dp*:  $D > 0$  **thus**  $(\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x < t) \longrightarrow (x - D < t))$  **by** *arith*

**next**

**assume**  $dp: D > 0$  **thus**  $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \leq t) \longrightarrow$   
 $(x - D \leq t)$  **by** *arith*  
**next**  
**assume**  $dp: D > 0$  **and**  $tB: t \in B$   
**{fix**  $x$  **assume**  $nob: \forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j$  **and**  $g: x > t$  **and**  $ng: \neg (x$   
 $- D) > t$   
**hence**  $x - t \leq D$  **and**  $1 \leq x - t$  **by** *simp+*  
**hence**  $\exists j \in \{1 \dots D\}. x - t = j$  **by** *auto*  
**hence**  $\exists j \in \{1 \dots D\}. x = t + j$  **by** (*simp add: algebra-simps*)  
**with**  $nob$   $tB$  **have** *False* **by** *simp*  
**thus**  $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x > t) \longrightarrow (x - D > t)$  **by** *blast*  
**next**  
**assume**  $dp: D > 0$  **and**  $tB: t - 1 \in B$   
**{fix**  $x$  **assume**  $nob: \forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j$  **and**  $g: x \geq t$  **and**  $ng: \neg (x$   
 $- D) \geq t$   
**hence**  $x - (t - 1) \leq D$  **and**  $1 \leq x - (t - 1)$  **by** *simp+*  
**hence**  $\exists j \in \{1 \dots D\}. x - (t - 1) = j$  **by** *auto*  
**hence**  $\exists j \in \{1 \dots D\}. x = (t - 1) + j$  **by** (*simp add: algebra-simps*)  
**with**  $nob$   $tB$  **have** *False* **by** *simp*  
**thus**  $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \geq t) \longrightarrow (x - D \geq t)$  **by** *blast*  
**next**  
**assume**  $d: d \text{ dvd } D$   
**{fix**  $x$  **assume**  $H: d \text{ dvd } x + t$  **with**  $d$  **have**  $d \text{ dvd } (x - D) + t$  **by** *algebra*  
**thus**  $\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (d \text{ dvd } x + t) \longrightarrow (d \text{ dvd } (x$   
 $- D) + t)$  **by** *simp*  
**next**  
**assume**  $d: d \text{ dvd } D$   
**{fix**  $x$  **assume**  $H: \neg (d \text{ dvd } x + t)$  **with**  $d$  **have**  $\neg d \text{ dvd } (x - D) + t$   
**by** (*clarsimp simp add: dvd-def,erule-tac x = ka + k in allE,simp add:*  
*algebra-simps*)  
**thus**  $\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (\neg d \text{ dvd } x + t) \longrightarrow (\neg d \text{ dvd}$   
 $(x - D) + t)$  **by** *auto*  
**qed** *blast*

**lemma** *aset*:

$\llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow P x \longrightarrow P(x + D) ;$   
 $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow Q x \longrightarrow Q(x + D) \rrbracket \Longrightarrow$   
 $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (P x \wedge Q x) \longrightarrow (P(x + D) \wedge Q(x$   
 $+ D))$   
 $\llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow P x \longrightarrow P(x + D) ;$   
 $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow Q x \longrightarrow Q(x + D) \rrbracket \Longrightarrow$   
 $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (P x \vee Q x) \longrightarrow (P(x + D) \vee Q(x +$   
 $D))$   
 $\llbracket D > 0 ; t + 1 \in A \rrbracket \Longrightarrow (\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x = t) \longrightarrow (x$   
 $+ D = t))$   
 $\llbracket D > 0 ; t \in A \rrbracket \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \neq t) \longrightarrow$   
 $(x + D \neq t))$   
 $\llbracket D > 0 ; t \in A \rrbracket \Longrightarrow (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x < t) \longrightarrow$   
 $(x + D < t))$

$\llbracket D > 0; t + 1 \in A \rrbracket \implies (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \leq t) \longrightarrow (x + D \leq t))$   
 $D > 0 \implies (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x > t) \longrightarrow (x + D > t))$   
 $D > 0 \implies (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \geq t) \longrightarrow (x + D \geq t))$   
 $d \text{ dvd } D \implies (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (d \text{ dvd } x + t) \longrightarrow (d \text{ dvd } (x + D) + t))$   
 $d \text{ dvd } D \implies (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (\neg d \text{ dvd } x + t) \longrightarrow (\neg d \text{ dvd } (x + D) + t))$   
 $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow F \longrightarrow F$   
**proof** (*blast*, *blast*)  
**assume** *dp*:  $D > 0$  **and** *tA*:  $t + 1 \in A$   
**show**  $(\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x = t) \longrightarrow (x + D = t))$   
**apply** (*rule allI*, *rule impI*, *erule ballE*[**where**  $x=1$ ], *erule ballE*[**where**  $x=t + 1$ ])  
**using** *dp tA* **by** *simp-all*  
**next**  
**assume** *dp*:  $D > 0$  **and** *tA*:  $t \in A$   
**show**  $(\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \neq t) \longrightarrow (x + D \neq t))$   
**apply** (*rule allI*, *rule impI*, *erule ballE*[**where**  $x=D$ ], *erule ballE*[**where**  $x=t$ ])  
**using** *dp tA* **by** *simp-all*  
**next**  
**assume** *dp*:  $D > 0$  **thus**  $(\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x > t) \longrightarrow (x + D > t))$  **by** *arith*  
**next**  
**assume** *dp*:  $D > 0$  **thus**  $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \geq t) \longrightarrow (x + D \geq t)$  **by** *arith*  
**next**  
**assume** *dp*:  $D > 0$  **and** *tA*:  $t \in A$   
**{fix** *x* **assume** *nob*:  $\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j$  **and** *g*:  $x < t$  **and** *ng*:  $\neg (x + D) < t$   
**hence**  $t - x \leq D$  **and**  $1 \leq t - x$  **by** *simp+*  
**hence**  $\exists j \in \{1 \dots D\}. t - x = j$  **by** *auto*  
**hence**  $\exists j \in \{1 \dots D\}. x = t - j$  **by** (*auto simp add: algebra-simps*)  
**with** *nob tA* **have** *False* **by** *simp*  
**thus**  $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x < t) \longrightarrow (x + D < t)$  **by** *blast*  
**next**  
**assume** *dp*:  $D > 0$  **and** *tA*:  $t + 1 \in A$   
**{fix** *x* **assume** *nob*:  $\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j$  **and** *g*:  $x \leq t$  **and** *ng*:  $\neg (x + D) \leq t$   
**hence**  $(t + 1) - x \leq D$  **and**  $1 \leq (t + 1) - x$  **by** (*simp-all add: algebra-simps*)  
**hence**  $\exists j \in \{1 \dots D\}. (t + 1) - x = j$  **by** *auto*  
**hence**  $\exists j \in \{1 \dots D\}. x = (t + 1) - j$  **by** (*auto simp add: algebra-simps*)  
**with** *nob tA* **have** *False* **by** *simp*  
**thus**  $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \leq t) \longrightarrow (x + D \leq t)$  **by** *blast*  
**next**  
**assume** *d*:  $d \text{ dvd } D$   
**{fix** *x* **assume** *H*:  $d \text{ dvd } x + t$  **with** *d* **have**  $d \text{ dvd } (x + D) + t$



```

    by (clarsimp simp add: dvd-def, rule-tac x = ka + k in exI, simp add:
algebra-simps) }
    thus  $\forall (x::int). (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (d \text{ dvd } x+t) \longrightarrow (d \text{ dvd } (x$ 
 $+ D) + t)$  by simp
next
  assume d: d dvd D
  {fix x assume H:  $\neg(d \text{ dvd } x + t)$  with d have  $\neg d \text{ dvd } (x + D) + t$ 
    by (clarsimp simp add: dvd-def, rule-tac x = ka - k in allE, simp add:
algebra-simps) }
    thus  $\forall (x::int). (\forall j \in \{1 .. D\}. \forall b \in A. x \neq b - j) \longrightarrow (\neg d \text{ dvd } x+t) \longrightarrow (\neg d \text{ dvd } (x$ 
 $+ D) + t)$  by auto
qed blast

```

### 35.3 Cooper’s Theorem $-\infty$ and $+\infty$ Version

#### 35.3.1 First some trivial facts about periodic sets or predicates

lemma *periodic-finite-ex*:

```

assumes dpos:  $(0::int) < d$  and modd:  $\text{ALL } x \ k. P \ x = P(x - k*d)$ 
shows  $(EX \ x. P \ x) = (EX \ j : \{1..d\}. P \ j)$ 
(is ?LHS = ?RHS)

```

proof

assume ?LHS

then obtain x where  $P \ x ..$

have  $x \bmod d = x - (x \text{ div } d)*d$  by (simp add: zmod-zdiv-equality mult-ac eq-diff-eq)

hence  $P \bmod: P \ x = P(x \bmod d)$  using modd by simp

show ?RHS

proof (cases)

assume  $x \bmod d = 0$

hence  $P \ 0$  using  $P \ P \bmod$  by simp

moreover have  $P \ 0 = P(0 - (-1)*d)$  using modd by blast

ultimately have  $P \ d$  by simp

moreover have  $d : \{1..d\}$  using dpos by (simp add: atLeastAtMost-iff)

ultimately show ?RHS ..

next

assume  $\text{not } 0: x \bmod d \neq 0$

have  $P(x \bmod d)$  using  $dpos \ P \ P \bmod$  by (simp add: pos-mod-sign pos-mod-bound)

moreover have  $x \bmod d : \{1..d\}$

proof -

from dpos have  $0 \leq x \bmod d$  by (rule pos-mod-sign)

moreover from dpos have  $x \bmod d < d$  by (rule pos-mod-bound)

ultimately show ?thesis using not0 by (simp add: atLeastAtMost-iff)

qed

ultimately show ?RHS ..

qed

qed auto

#### 35.3.2 The $-\infty$ Version

lemma *decr-lemma*:  $0 < (d::int) \implies x - (\text{abs}(x-z)+1) * d < z$

**by**(*induct rule: int-gr-induct, simp-all add:int-distrib*)

**lemma** *incr-lemma*:  $0 < (d::int) \implies z < x + (abs(x-z)+1) * d$   
**by**(*induct rule: int-gr-induct, simp-all add:int-distrib*)

**theorem** *int-induct*[*case-names base step1 step2*]:

**assumes**

*base*:  $P(k::int)$  **and** *step1*:  $\bigwedge i. \llbracket k \leq i; P\ i \rrbracket \implies P(i+1)$  **and**

*step2*:  $\bigwedge i. \llbracket k \geq i; P\ i \rrbracket \implies P(i-1)$

**shows**  $P\ i$

**proof** –

**have**  $i \leq k \vee i \geq k$  **by** *arith*

**thus** *?thesis* **using** *prems int-ge-induct*[**where**  $P=P$  **and**  $k=k$  **and**  $i=i$ ] *int-le-induct*[**where**  $P=P$  **and**  $k=k$  **and**  $i=i$ ] **by** *blast*

**qed**

**lemma** *decr-mult-lemma*:

**assumes** *dpos*:  $(0::int) < d$  **and** *minus*:  $\forall x. P\ x \longrightarrow P(x-d)$  **and** *knneg*:  $0 \leq k$

**shows**  $ALL\ x. P\ x \longrightarrow P(x-k*d)$

**using** *knneg*

**proof** (*induct rule:int-ge-induct*)

**case** *base* **thus** *?case* **by** *simp*

**next**

**case** (*step i*)

{**fix**  $x$

**have**  $P\ x \longrightarrow P(x-i*d)$  **using** *step.hyps* **by** *blast*

**also have**  $\dots \longrightarrow P(x-(i+1)*d)$  **using** *minus*[*THEN spec*, of  $x-i*d$ ]

**by** (*simp add:int-distrib OrderedGroup.diff-diff-eq[symmetric]*)

**ultimately have**  $P\ x \longrightarrow P(x-(i+1)*d)$  **by** *blast*}

**thus** *?case* ..

**qed**

**lemma** *minusinfinity*:

**assumes** *dpos*:  $0 < d$  **and**

*P1eqP1*:  $ALL\ x\ k. P1\ x = P1(x-k*d)$  **and** *ePeqP1*:  $EX\ z::int. ALL\ x. x < z \longrightarrow (P\ x = P1\ x)$

**shows**  $(EX\ x. P1\ x) \longrightarrow (EX\ x. P\ x)$

**proof**

**assume** *eP1*:  $EX\ x. P1\ x$

**then obtain**  $x$  **where**  $P1\ x$  ..

**from** *ePeqP1* **obtain**  $z$  **where** *P1eqP*:  $ALL\ x. x < z \longrightarrow (P\ x = P1\ x)$  ..

**let**  $?w = x - (abs(x-z)+1) * d$

**from** *dpos* **have**  $w: ?w < z$  **by**(*rule decr-lemma*)

**have**  $P1\ x = P1\ ?w$  **using** *P1eqP1* **by** *blast*

**also have**  $\dots = P(?w)$  **using**  $w\ P1eqP$  **by** *blast*

**finally have**  $P\ ?w$  **using** *P1* **by** *blast*

**thus**  $EX\ x. P\ x$  ..

**qed**

**lemma** *cpmi*:  
**assumes** *dp*:  $0 < D$  **and** *p1*:  $\exists z. \forall x < z. P\ x = P'\ x$   
**and** *nb*:  $\forall x. (\forall j \in \{1..D\}. \forall (b::int) \in B. x \neq b+j) \longrightarrow P\ (x) \longrightarrow P\ (x - D)$   
**and** *pd*:  $\forall x\ k. P'\ x = P'\ (x - k * D)$   
**shows**  $(\exists x. P\ x) = ((\exists j \in \{1..D\}. P'\ j) \mid (\exists j \in \{1..D\}. \exists b \in B. P\ (b+j)))$   
**(is**  $?L = (?R1 \vee ?R2)$ **)**  
**proof**–  
**{assume** *?R2* **hence** *?L* **by** *blast***}**  
**moreover**  
**{assume** *H*: *?R1* **hence** *?L* **using** *minusinfinity*[*OF dp pd p1*] *periodic-finite-ex*[*OF dp pd*] **by** *simp***}**  
**moreover**  
**{ fix** *x*  
**assume** *P*:  $P\ x$  **and** *H*:  $\neg ?R2$   
**{fix** *y* **assume**  $\neg (\exists j \in \{1..D\}. \exists b \in B. P\ (b + j))$  **and** *P*:  $P\ y$   
**hence**  $\sim (EX\ (j::int) : \{1..D\}. EX\ (b::int) : B. y = b+j)$  **by** *auto*  
**with** *nb* *P* **have**  $P\ (y - D)$  **by** *auto* **}**  
**hence**  $ALL\ x. \sim (EX\ (j::int) : \{1..D\}. EX\ (b::int) : B. P(b+j)) \longrightarrow P\ (x)$   
 $\longrightarrow P\ (x - D)$  **by** *blast*  
**with** *H P* **have** *th*:  $\forall x. P\ x \longrightarrow P\ (x - D)$  **by** *auto*  
**from** *p1* **obtain** *z* **where**  $z: ALL\ x. x < z \longrightarrow (P\ x = P'\ x)$  **by** *blast*  
**let** *?y*  $= x - (|x - z| + 1) * D$   
**have** *zp*:  $0 \leq (|x - z| + 1)$  **by** *arith*  
**from** *dp* **have** *yz*:  $?y < z$  **using** *decr-lemma*[*OF dp*] **by** *simp*  
**from** *z*[*rule-format*, *OF yz*] *decr-mult-lemma*[*OF dp th zp*, *rule-format*, *OF P*]  
**have** *th2*:  $P'\ ?y$  **by** *auto*  
**with** *periodic-finite-ex*[*OF dp pd*]  
**have** *?R1* **by** *blast***}**  
**ultimately show** *?thesis* **by** *blast*  
**qed**

### 35.3.3 The $+\infty$ Version

**lemma** *plusinfinity*:  
**assumes** *dpos*:  $(0::int) < d$  **and**  
*P1eqP1*:  $\forall x\ k. P'\ x = P'\ (x - k * d)$  **and** *ePeqP1*:  $\exists z. \forall x > z. P\ x = P'\ x$   
**shows**  $(\exists x. P'\ x) \longrightarrow (\exists x. P\ x)$   
**proof**  
**assume** *eP1*:  $EX\ x. P'\ x$   
**then obtain** *x* **where** *P1*:  $P'\ x ..$   
**from** *ePeqP1* **obtain** *z* **where** *P1eqP*:  $\forall x > z. P\ x = P'\ x ..$   
**let** *?w'*  $= x + (abs(x-z)+1) * d$   
**let** *?w*  $= x - (-(abs(x-z) + 1)) * d$   
**have** *ww'*[*simp*]:  $?w = ?w'$  **by** (*simp add: algebra-simps*)  
**from** *dpos* **have** *w*:  $?w > z$  **by** (*simp only: ww' incr-lemma*)  
**hence**  $P'\ x = P'\ ?w$  **using** *P1eqP1* **by** *blast*  
**also have**  $... = P(?w)$  **using** *w P1eqP* **by** *blast*

finally have  $P \text{ ?}w$  using  $P1$  by *blast*  
 thus  $EX\ x. P\ x$  ..  
 qed

lemma *incr-mult-lemma*:

assumes  $dpos: (0::int) < d$  and  $plus: ALL\ x::int. P\ x \longrightarrow P(x + d)$  and  $knneg: 0 \leq k$   
 shows  $ALL\ x. P\ x \longrightarrow P(x + k*d)$   
 using *knneg*  
 proof (induct rule: *int-ge-induct*)  
 case base thus ?case by *simp*  
 next  
 case (step  $i$ )  
 {fix  $x$   
 have  $P\ x \longrightarrow P(x + i * d)$  using *step.hyps* by *blast*  
 also have  $\dots \longrightarrow P(x + (i + 1) * d)$  using *plus*[*THEN spec, of  $x + i * d$* ]  
 by (*simp add: int-distrib zadd-ac*)  
 ultimately have  $P\ x \longrightarrow P(x + (i + 1) * d)$  by *blast*}  
 thus ?case ..  
 qed

lemma *cpai*:

assumes  $dp: 0 < D$  and  $p1: \exists z. \forall x > z. P\ x = P'\ x$   
 and  $nb: \forall x. (\forall j \in \{1..D\}. \forall (b::int) \in A. x \neq b - j) \longrightarrow P(x) \longrightarrow P(x + D)$   
 and  $pd: \forall x\ k. P'\ x = P'(x - k*D)$   
 shows  $(\exists x. P\ x) = ((\exists j \in \{1..D\}. P'\ j) \mid (\exists j \in \{1..D\}. \exists b \in A. P(b - j)))$   
 (is ?L = (?R1  $\vee$  ?R2))  
 proof—  
 {assume ?R2 hence ?L by *blast*}  
 moreover  
 {assume  $H: ?R1$  hence ?L using *plusinfinity*[*OF dp pd p1*] *periodic-finite-ex*[*OF dp pd*] by *simp*}  
 moreover  
 {fix  $x$   
 assume  $P: P\ x$  and  $H: \neg ?R2$   
 {fix  $y$  assume  $\neg (\exists j \in \{1..D\}. \exists b \in A. P(b - j))$  and  $P: P\ y$   
 hence  $\sim (EX\ (j::int) : \{1..D\}. EX\ (b::int) : A. y = b - j)$  by *auto*  
 with  $nb\ P$  have  $P(y + D)$  by *auto* }  
 hence  $ALL\ x. \sim (EX\ (j::int) : \{1..D\}. EX\ (b::int) : A. P(b - j)) \longrightarrow P(x)$   
 $\longrightarrow P(x + D)$  by *blast*  
 with  $H\ P$  have  $th: \forall x. P\ x \longrightarrow P(x + D)$  by *auto*  
 from  $p1$  obtain  $z$  where  $z: ALL\ x. x > z \longrightarrow (P\ x = P'\ x)$  by *blast*  
 let  $?y = x + (|x - z| + 1)*D$   
 have  $zp: 0 \leq (|x - z| + 1)$  by *arith*  
 from  $dp$  have  $yz: ?y > z$  using *incr-lemma*[*OF dp*] by *simp*  
 from  $z$ [*rule-format, OF yz*] *incr-mult-lemma*[*OF dp th zp, rule-format, OF P*]  
 have  $th2: P'\ ?y$  by *auto*  
 with *periodic-finite-ex*[*OF dp pd*]

```

  have ?R1 by blast}
  ultimately show ?thesis by blast
qed

```

```

lemma simp-from-to: {i..j::int} = (if j < i then {} else insert i {i+1..j})
apply(simp add:atLeastAtMost-def atLeast-def atMost-def)
apply(fastsimp)
done

```

```

theorem unity-coeff-ex: ( $\exists (x::'a::\{\text{semiring-0}, \text{Ring-and-Field.dvd}\}). P (l * x) \equiv$ 
 $(\exists x. l \text{ dvd } (x + 0) \wedge P x)$ )
  apply (rule eq-reflection [symmetric])
  apply (rule iffI)
  defer
  apply (erule exE)
  apply (rule-tac x = l * x in exI)
  apply (simp add: dvd-def)
  apply (rule-tac x = x in exI, simp)
  apply (erule exE)
  apply (erule conjE)
  apply simp
  apply (erule dvdE)
  apply (rule-tac x = k in exI)
  apply simp
done

```

```

lemma zdvd-mono: assumes not0: (k::int)  $\neq$  0
shows ((m::int) dvd t)  $\equiv$  (k*m dvd k*t)
  using not0 by (simp add: dvd-def)

```

```

lemma uminus-dvd-conv: (d dvd (t::int))  $\equiv$  (-d dvd t) (d dvd (t::int))  $\equiv$  (d dvd
-t)
  by simp-all

```

Theorems for transforming predicates on nat to predicates on int

```

lemma all-nat: ( $\forall x::\text{nat}. P x$ ) = ( $\forall x::\text{int}. 0 \leq x \longrightarrow P (\text{nat } x)$ )
  by (simp split add: split-nat)

```

```

lemma ex-nat: ( $\exists x::\text{nat}. P x$ ) = ( $\exists x::\text{int}. 0 \leq x \wedge P (\text{nat } x)$ )
  apply (auto split add: split-nat)
  apply (rule-tac x=int x in exI, simp)
  apply (rule-tac x = nat x in exI,erule-tac x = nat x in allE, simp)
done

```

```

lemma zdiff-int-split: P (int (x - y)) =
  ((y  $\leq$  x  $\longrightarrow$  P (int x - int y))  $\wedge$  (x < y  $\longrightarrow$  P 0))
  by (case-tac y  $\leq$  x, simp-all add: zdiff-int)

```

```

lemma number-of1: (0::int) <= number-of n  $\implies$  (0::int) <= number-of (Int.Bit0 n)  $\wedge$  (0::int) <= number-of (Int.Bit1 n)
by simp
lemma number-of2: (0::int) <= Numeral0 by simp
lemma Suc-plus1: Suc n = n + 1 by simp

```

Specific instances of congruence rules, to prevent simplifier from looping.

```

theorem imp-le-cong: (0 <= x  $\implies$  P = P')  $\implies$  (0 <= (x::int)  $\longrightarrow$  P) = (0 <= x  $\longrightarrow$  P') by simp

```

```

theorem conj-le-cong: (0 <= x  $\implies$  P = P')  $\implies$  (0 <= (x::int)  $\wedge$  P) = (0 <= x  $\wedge$  P')

```

```

by (simp cong: conj-cong)

```

```

lemma int-eq-number-of-eq:

```

```

  (((number-of v)::int) = (number-of w)) = iszero ((number-of (v + (uminus w)))::int)

```

```

by (rule eq-number-of-eq)

```

```

declare dvd-eq-mod-eq-0[symmetric, presburger]

```

```

declare mod-1[presburger]

```

```

declare mod-0[presburger]

```

```

declare mod-by-1[presburger]

```

```

declare zmod-zero[presburger]

```

```

declare zmod-self[presburger]

```

```

declare mod-self[presburger]

```

```

declare mod-by-0[presburger]

```

```

declare mod-div-trivial[presburger]

```

```

declare div-mod-equality2[presburger]

```

```

declare div-mod-equality[presburger]

```

```

declare mod-div-equality2[presburger]

```

```

declare mod-div-equality[presburger]

```

```

declare mod-mult-self1[presburger]

```

```

declare mod-mult-self2[presburger]

```

```

declare zdiv-zmod-equality2[presburger]

```

```

declare zdiv-zmod-equality[presburger]

```

```

declare mod2-Suc-Suc[presburger]

```

```

lemma [presburger]: (a::int) div 0 = 0 and [presburger]: a mod 0 = a

```

```

by simp-all

```

```

use Tools/Qelim/cooper.ML

```

```

oracle linzqe-oracle = Coopereif.cooper-oracle

```

```

use Tools/Qelim/presburger.ML

```

```

setup  $\ll$  Arith-Data.add-tactic Presburger arithmetic (K (Presburger.cooper-tac true [] []))  $\gg$ 

```

```

method-setup presburger =  $\ll$ 

```

```

  let

```

```

fun keyword k = Scan.lift (Args.$$$ k -- Args.colon) >> K ()
fun simple-keyword k = Scan.lift (Args.$$$ k) >> K ()
val addN = add
val delN = del
val elimN = elim
val any-keyword = keyword addN || keyword delN || simple-keyword elimN
val thms = Scan.repeat (Scan.unless any-keyword Attrib.multi-thm) >> flat;
in
  Scan.optional (simple-keyword elimN >> K false) true --
  Scan.optional (keyword addN |-- thms) [] --
  Scan.optional (keyword delN |-- thms) [] >>
  (fn ((elim, add-ths), del-ths) => fn ctxt =>
    SIMPLE-METHOD' (Presburger.cooper-tac elim add-ths del-ths ctxt))
end
>> Cooper's algorithm for Presburger arithmetic

lemma [presburger, algebra]: m mod 2 = (1::nat)  $\longleftrightarrow$   $\neg$  2 dvd m by presburger
lemma [presburger, algebra]: m mod 2 = Suc 0  $\longleftrightarrow$   $\neg$  2 dvd m by presburger
lemma [presburger, algebra]: m mod (Suc (Suc 0)) = (1::nat)  $\longleftrightarrow$   $\neg$  2 dvd m by
presburger
lemma [presburger, algebra]: m mod (Suc (Suc 0)) = Suc 0  $\longleftrightarrow$   $\neg$  2 dvd m by
presburger
lemma [presburger, algebra]: m mod 2 = (1::int)  $\longleftrightarrow$   $\neg$  2 dvd m by presburger

lemma zdvd-period:
  fixes a d :: int
  assumes advdd: a dvd d
  shows a dvd (x + t)  $\longleftrightarrow$  a dvd ((x + c * d) + t)
  using advdd
  apply -
  apply (rule iffI)
  by algebra+

end

```

## 36 Recdef: TFL: recursive function definitions

```

theory Recdef
imports FunDef Plain
uses
  (Tools/TFL/casesplit.ML)
  (Tools/TFL/utis.ML)
  (Tools/TFL/usyntax.ML)
  (Tools/TFL/dcterm.ML)
  (Tools/TFL/thms.ML)
  (Tools/TFL/rules.ML)
  (Tools/TFL/thry.ML)

```

```

(Tools/TFL/tfl.ML)
(Tools/TFL/post.ML)
(Tools/recdef-package.ML)

```

```
begin
```

\* This form avoids giant explosions in proofs. NOTE USE OF ==

```
lemma def-wfrec: [| f==wfrec r H; wf(r) |] ==> f(a) = H (cut f r a) a
```

```
apply auto
```

```
apply (blast intro: wfrec)
```

```
done
```

```
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))
```

```
apply clarify
```

```
apply (rule-tac r = R and P = P and a = x in wf-induct, assumption, blast)
```

```
done
```

```
lemma tfl-cut-apply: ALL f R. (x,a):R --> (cut f R a)(x) = f(x)
```

```
apply clarify
```

```
apply (rule cut-apply, assumption)
```

```
done
```

```
lemma tfl-wfrec:
```

```
ALL M R f. (f==wfrec R M) --> wf R --> (ALL x. f x = M (cut f R x) x)
```

```
apply clarify
```

```
apply (erule wfrec)
```

```
done
```

```
lemma tfl-eq-True: (x = True) --> x
```

```
by blast
```

```
lemma tfl-rev-eq-mp: (x = y) --> y --> x
```

```
by blast
```

```
lemma tfl-simp-thm: (x --> y) --> (x = x') --> (x' --> y)
```

```
by blast
```

```
lemma tfl-P-imp-P-iff-True: P ==> P = True
```

```
by blast
```

```
lemma tfl-imp-trans: (A --> B) ==> (B --> C) ==> (A --> C)
```

```
by blast
```

```
lemma tfl-disj-assoc: (a ∨ b) ∨ c == a ∨ (b ∨ c)
```

```
by simp
```

```
lemma tfl-disjE: P ∨ Q ==> P --> R ==> Q --> R ==> R
```



```

by blast

lemma tfl-exE:  $\exists x. P\ x ==> \forall x. P\ x \dashv\dashv Q ==> Q$ 
by blast

use Tools/TFL/casesplit.ML
use Tools/TFL/utls.ML
use Tools/TFL/usyntax.ML
use Tools/TFL/dcterm.ML
use Tools/TFL/thms.ML
use Tools/TFL/rules.ML
use Tools/TFL/thry.ML
use Tools/TFL/tfl.ML
use Tools/TFL/post.ML
use Tools/recdef-package.ML
setup RecdefPackage.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 INT-cong UN-cong bex-cong ball-cong imp-cong

lemmas [recdef-wf] =
  wf-trancl
  wf-less-than
  wf-lex-prod
  wf-inv-image
  wf-measure
  wf-pred-nat
  wf-same-fst
  wf-empty

end

```

### 37 Hilbert-Choice: Hilbert’s Epsilon-Operator and the Axiom of Choice

```

theory Hilbert-Choice
imports Nat Wellfounded Plain
uses (Tools/meson.ML) (Tools/specification-package.ML)
begin

```

### 37.1 Hilbert’s epsilon

#### axiomatization

$Eps :: ('a \Rightarrow bool) \Rightarrow 'a$

#### where

$someI: P\ x \Rightarrow P\ (Eps\ P)$

#### syntax (epsilon)

$-Eps :: [pttrn, bool] \Rightarrow 'a \quad ((\exists \epsilon \text{ -./ -}) [0, 10] 10)$

#### syntax (HOL)

$-Eps :: [pttrn, bool] \Rightarrow 'a \quad ((\exists @ \text{ -./ -}) [0, 10] 10)$

#### syntax

$-Eps :: [pttrn, bool] \Rightarrow 'a \quad ((\exists SOME \text{ -./ -}) [0, 10] 10)$

#### translations

$SOME\ x.\ P == CONST\ Eps\ (\%x.\ P)$

#### print-translation $\ll$

$(*$  to avoid eta-contraction of body  $*)$   
 $[(\@ \{const\text{-}syntax\ Eps\}, fn\ [Abs\ abs] \Rightarrow$   
 $\quad let\ val\ (x,t) = atomic\text{-}abs\text{-}tr'\ abs$   
 $\quad in\ Syntax.const\ \text{-}Eps\ \$\ x\ \$\ t\ end)]$   
 $\gg$

#### constdefs

$inv :: ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$   
 $inv(f :: 'a \Rightarrow 'b) == \%y.\ SOME\ x.\ f\ x = y$

$Inv :: 'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$   
 $Inv\ A\ f == \%x.\ SOME\ y.\ y \in A \ \&\ f\ y = x$

### 37.2 Hilbert’s Epsilon-operator

Easier to apply than *someI* if the witness comes from an existential formula

**lemma** *someI-ex* [*elim?*]:  $\exists x.\ P\ x \Rightarrow P\ (SOME\ x.\ P\ x)$

**apply** (*erule exE*)

**apply** (*erule someI*)

**done**

Easier to apply than *someI* because the conclusion has only one occurrence of *P*.

**lemma** *someI2*:  $[\![\ P\ a;\ \! \! x.\ P\ x \Rightarrow Q\ x\ ]\!] \Rightarrow Q\ (SOME\ x.\ P\ x)$

**by** (*blast intro: someI*)

Easier to apply than *someI2* if the witness comes from an existential formula

**lemma** *someI2-ex*:  $[\![\ \exists a.\ P\ a;\ \! \! x.\ P\ x \Rightarrow Q\ x\ ]\!] \Rightarrow Q\ (SOME\ x.\ P\ x)$

**by** (*blast intro: someI2*)

**lemma** *some-equality* [*intro*]:

$[\![\ P\ a;\ \! \! x.\ P\ x \Rightarrow x=a\ ]\!] \Rightarrow (SOME\ x.\ P\ x) = a$

**by** (*blast intro: someI2*)

**lemma** *some1-equality*:  $[[ \text{EX!}x. P\ x; P\ a \ ]] ==> (\text{SOME } x. P\ x) = a$   
**by** (*blast intro: some-equality*)

**lemma** *some-eq-ex*:  $P\ (\text{SOME } x. P\ x) = (\exists x. P\ x)$   
**by** (*blast intro: someI*)

**lemma** *some-eq-trivial* [*simp*]:  $(\text{SOME } y. y=x) = x$   
**apply** (*rule some-equality*)  
**apply** (*rule refl, assumption*)  
**done**

**lemma** *some-sym-eq-trivial* [*simp*]:  $(\text{SOME } y. x=y) = x$   
**apply** (*rule some-equality*)  
**apply** (*rule refl*)  
**apply** (*erule sym*)  
**done**

### 37.3 Axiom of Choice, Proved Using the Description Operator

Used in *Tools/meson.ML*

**lemma** *choice*:  $\forall x. \exists y. Q\ x\ y ==> \exists f. \forall x. Q\ x\ (f\ x)$   
**by** (*fast elim: someI*)

**lemma** *bchoice*:  $\forall x \in S. \exists y. Q\ x\ y ==> \exists f. \forall x \in S. Q\ x\ (f\ x)$   
**by** (*fast elim: someI*)

### 37.4 Function Inverse

**lemma** *inv-id* [*simp*]:  $\text{inv } id = id$   
**by** (*simp add: inv-def id-def*)

A one-to-one function has an inverse.

**lemma** *inv-f-f* [*simp*]:  $\text{inj } f ==> \text{inv } f\ (f\ x) = x$   
**by** (*simp add: inv-def inj-eq*)

**lemma** *inv-f-eq*:  $[[ \text{inj } f; f\ x = y \ ]] ==> \text{inv } f\ y = x$   
**apply** (*erule subst*)  
**apply** (*erule inv-f-f*)  
**done**

**lemma** *inj-imp-inv-eq*:  $[[ \text{inj } f; \forall x. f(g\ x) = x \ ]] ==> \text{inv } f = g$   
**by** (*blast intro: ext inv-f-eq*)

But is it useful?

**lemma** *inj-transfer*:

```

    assumes injf: inj f and minor: !!y. y ∈ range(f) ==> P(inv f y)
    shows P x
  proof -
    have f x ∈ range f by auto
    hence P(inv f (f x)) by (rule minor)
    thus P x by (simp add: inv-f-f [OF injf])
  qed

```

```

lemma inj-iff: (inj f) = (inv f o f = id)
apply (simp add: o-def expand-fun-eq)
apply (blast intro: inj-on-inverseI inv-f-f)
done

```

```

lemma inv-o-cancel[simp]: inj f ==> inv f o f = id
by (simp add: inj-iff)

```

```

lemma o-inv-o-cancel[simp]: inj f ==> g o inv f o f = g
by (simp add: o-assoc[symmetric])

```

```

lemma inv-image-cancel[simp]:
  inj f ==> inv f ‘ f ‘ S = S
by (simp add: image-compose[symmetric])

```

```

lemma inj-imp-surj-inv: inj f ==> surj (inv f)
by (blast intro: surjI inv-f-f)

```

```

lemma f-inv-f: y ∈ range(f) ==> f(inv f y) = y
apply (simp add: inv-def)
apply (fast intro: someI)
done

```

```

lemma surj-f-inv-f: surj f ==> f(inv f y) = y
by (simp add: f-inv-f surj-range)

```

```

lemma inv-injective:
  assumes eq: inv f x = inv f y
    and x: x: range f
    and y: y: range f
  shows x=y
  proof -
    have f (inv f x) = f (inv f y) using eq by simp
    thus ?thesis by (simp add: f-inv-f x y)
  qed

```

```

lemma inj-on-inv: A <= range(f) ==> inj-on (inv f) A
by (fast intro: inj-onI elim: inv-injective injD)

```

```

lemma surj-imp-inj-inv: surj f ==> inj (inv f)

```

**by** (*simp add: inj-on-inv surj-range*)

**lemma** *surj-iff*:  $(\text{surj } f) = (f \circ \text{inv } f = \text{id})$   
**apply** (*simp add: o-def expand-fun-eq*)  
**apply** (*blast intro: surjI surj-f-inv-f*)  
**done**

**lemma** *surj-imp-inv-eq*:  $[\text{surj } f; \forall x. g(f\ x) = x] \implies \text{inv } f = g$   
**apply** (*rule ext*)  
**apply** (*drule-tac x = inv f x in spec*)  
**apply** (*simp add: surj-f-inv-f*)  
**done**

**lemma** *bij-imp-bij-inv*:  $\text{bij } f \implies \text{bij } (\text{inv } f)$   
**by** (*simp add: bij-def inj-imp-surj-inv surj-imp-inj-inv*)

**lemma** *inv-equality*:  $[\forall x. g(f\ x) = x; \forall y. f(g\ y) = y] \implies \text{inv } f = g$   
**apply** (*rule ext*)  
**apply** (*auto simp add: inv-def*)  
**done**

**lemma** *inv-inv-eq*:  $\text{bij } f \implies \text{inv } (\text{inv } f) = f$   
**apply** (*rule inv-equality*)  
**apply** (*auto simp add: bij-def surj-f-inv-f*)  
**done**

**lemma** *o-inv-distrib*:  $[\text{bij } f; \text{bij } g] \implies \text{inv } (f \circ g) = \text{inv } g \circ \text{inv } f$   
**apply** (*rule inv-equality*)  
**apply** (*auto simp add: bij-def surj-f-inv-f*)  
**done**

**lemma** *image-surj-f-inv-f*:  $\text{surj } f \implies f \circ (\text{inv } f \circ A) = A$   
**by** (*simp add: image-eq-UN surj-f-inv-f*)

**lemma** *image-inv-f-f*:  $\text{inj } f \implies (\text{inv } f) \circ (f \circ A) = A$   
**by** (*simp add: image-eq-UN*)

**lemma** *inv-image-comp*:  $\text{inj } f \implies \text{inv } f \circ (f \circ X) = X$   
**by** (*auto simp add: image-def*)

**lemma** *bij-image-Collect-eq*:  $\text{bij } f \implies f \circ \text{Collect } P = \{y. P(\text{inv } f\ y)\}$   
**apply** *auto*  
**apply** (*force simp add: bij-is-inj*)  
**apply** (*blast intro: bij-is-surj [THEN surj-f-inv-f, symmetric]*)  
**done**

```

lemma bij-vimage-eq-inv-image:  $\text{bij } f \implies f^{-1} A = \text{inv } f^{-1} A$ 
apply (auto simp add: bij-is-surj [THEN surj-f-inv-f])
apply (blast intro: bij-is-inj [THEN inv-f-f, symmetric])
done

```

### 37.5 Inverse of a PI-function (restricted domain)

```

lemma Inv-f-f:  $[\text{inj-on } f \text{ } A; x \in A] \implies \text{Inv } A \text{ } f (f x) = x$ 
apply (simp add: Inv-def inj-on-def)
apply (blast intro: someI2)
done

```

```

lemma f-Inv-f:  $y \in f^{-1} A \implies f (\text{Inv } A \text{ } f y) = y$ 
apply (simp add: Inv-def)
apply (fast intro: someI2)
done

```

```

lemma Inv-injective:
  assumes eq:  $\text{Inv } A \text{ } f x = \text{Inv } A \text{ } f y$ 
  and x:  $x \in f^{-1} A$ 
  and y:  $y \in f^{-1} A$ 
  shows  $x=y$ 
proof –
  have  $f (\text{Inv } A \text{ } f x) = f (\text{Inv } A \text{ } f y)$  using eq by simp
  thus ?thesis by (simp add: f-Inv-f x y)
qed

```

```

lemma inj-on-Inv:  $B \subseteq f^{-1} A \implies \text{inj-on } (\text{Inv } A \text{ } f) \text{ } B$ 
apply (rule inj-onI)
apply (blast intro: inj-onI dest: Inv-injective injD)
done

```

```

lemma Inv-mem:  $[\text{inv } f^{-1} A = B; x \in B] \implies \text{Inv } A \text{ } f x \in A$ 
apply (simp add: Inv-def)
apply (fast intro: someI2)
done

```

```

lemma Inv-f-eq:  $[\text{inj-on } f \text{ } A; f x = y; x \in A] \implies \text{Inv } A \text{ } f y = x$ 
apply (erule subst)
apply (erule Inv-f-f, assumption)
done

```

```

lemma Inv-comp:
   $[\text{inj-on } f \text{ } (g^{-1} A); \text{inj-on } g \text{ } A; x \in f^{-1} (g^{-1} A)] \implies$ 
   $\text{Inv } A (f \circ g) x = (\text{Inv } A \text{ } g \circ \text{Inv } (g^{-1} A) \text{ } f) x$ 
apply simp
apply (rule Inv-f-eq)
apply (fast intro: comp-inj-on)
apply (simp add: f-Inv-f Inv-mem)

```

```

apply (simp add: Inv-mem)
done

```

```

lemma bij-betw-Inv: bij-betw f A B  $\implies$  bij-betw (Inv A f) B A
apply (auto simp add: bij-betw-def inj-on-Inv Inv-mem)
apply (simp add: image-compose [symmetric] o-def)
apply (simp add: image-def Inv-f-f)
done

```

### 37.6 Other Consequences of Hilbert’s Epsilon

Hilbert’s Epsilon and the *split* Operator

Looping *simp* rule

```

lemma split-paired-Eps: (SOME x. P x) = (SOME (a,b). P(a,b))
by simp

```

```

lemma Eps-split: Eps (split P) = (SOME xy. P (fst xy) (snd xy))
by (simp add: split-def)

```

```

lemma Eps-split-eq [simp]: (@(x',y'). x = x' & y = y') = (x,y)
by blast

```

A relation is wellfounded iff it has no infinite descending chain

```

lemma wf-iff-no-infinite-down-chain:
  wf r = ( $\sim(\exists f. \forall i. (f(Suc\ i), f\ i) \in r)$ )
apply (simp only: wf-eq-minimal)
apply (rule iffI)
apply (rule notI)
apply (erule exE)
apply (erule tac x = {w.  $\exists i. w=f\ i$ } in allE, blast)
apply (erule contrapos-np, simp, clarify)
apply (subgoal-tac  $\forall n. \text{nat-rec } x\ (\%i\ y. @z. z:Q \ \&\ (z,y):r)\ n \in Q$ )
apply (rule-tac x = nat-rec x (%i y. @z. z:Q & (z,y):r) in exI)
apply (rule allI, simp)
apply (rule someI2-ex, blast, blast)
apply (rule allI)
apply (induct-tac n, simp-all)
apply (rule someI2-ex, blast+)
done

```

```

lemma wf-no-infinite-down-chainE:
  assumes wf r obtains k where (f (Suc k), f k)  $\notin$  r
using (wf r) wf-iff-no-infinite-down-chain[of r] by blast

```

A dynamically-scoped fact for TFL

```

lemma tfl-some:  $\forall P\ x. P\ x \dashv\dashv P\ (Eps\ P)$ 
by (blast intro: someI)

```

### 37.7 Least value operator

**constdefs**

$LeastM :: [ 'a ==> 'b::ord, 'a ==> bool ] ==> 'a$   
 $LeastM\ m\ P == SOME\ x.\ P\ x \ \& \ (\forall y.\ P\ y \dashrightarrow m\ x \leq m\ y)$

**syntax**

$-LeastM :: [pttrn, 'a ==> 'b::ord, bool] ==> 'a \quad (LEAST - WRT \cdot, \cdot [0, 4, 10]$   
 $10)$

**translations**

$LEAST\ x\ WRT\ m.\ P == LeastM\ m\ (\%x.\ P)$

**lemma** *LeastMI2*:

$P\ x ==> (!y.\ P\ y ==> m\ x \leq m\ y)$   
 $==> (!x.\ P\ x ==> \forall y.\ P\ y \dashrightarrow m\ x \leq m\ y ==> Q\ x)$   
 $==> Q\ (LeastM\ m\ P)$   
**apply** (*simp add: LeastM-def*)  
**apply** (*rule someI2-ex, blast, blast*)  
**done**

**lemma** *LeastM-equality*:

$P\ k ==> (!x.\ P\ x ==> m\ k \leq m\ x)$   
 $==> m\ (LEAST\ x\ WRT\ m.\ P\ x) = (m\ k::'a::order)$   
**apply** (*rule LeastMI2, assumption, blast*)  
**apply** (*blast intro!: order-antisym*)  
**done**

**lemma** *wf-linord-ex-has-least*:

$wf\ r ==> \forall x\ y.\ ((x,y):r^+)=((y,x)^\sim:r^*) ==> P\ k$   
 $==> \exists x.\ P\ x \ \& \ (!y.\ P\ y \dashrightarrow (m\ x, m\ y):r^*)$   
**apply** (*drule wf-trancl [THEN wf-eq-minimal [THEN iffD1]]*)  
**apply** (*drule-tac x = m'Collect P in spec, force*)  
**done**

**lemma** *ex-has-least-nat*:

$P\ k ==> \exists x.\ P\ x \ \& \ (\forall y.\ P\ y \dashrightarrow m\ x \leq (m\ y::nat))$   
**apply** (*simp only: pred-nat-trancl-eq-le [symmetric]*)  
**apply** (*rule wf-pred-nat [THEN wf-linord-ex-has-least]*)  
**apply** (*simp add: less-eq linorder-not-le pred-nat-trancl-eq-le, assumption*)  
**done**

**lemma** *LeastM-nat-lemma*:

$P\ k ==> P\ (LeastM\ m\ P) \ \& \ (\forall y.\ P\ y \dashrightarrow m\ (LeastM\ m\ P) \leq (m\ y::nat))$   
**apply** (*simp add: LeastM-def*)  
**apply** (*rule someI-ex*)  
**apply** (*erule ex-has-least-nat*)  
**done**

**lemmas** *LeastM-natI = LeastM-nat-lemma [THEN conjunct1, standard]*



**lemma** *LeastM-nat-le*:  $P\ x \implies m\ (LeastM\ m\ P) \leq (m\ x::nat)$   
**by** (rule *LeastM-nat-lemma* [THEN *conjunct2*, THEN *spec*, THEN *mp*], *assumption*, *assumption*)

### 37.8 Greatest value operator

#### constdefs

*GreatestM* ::  $[a \Rightarrow b::ord, a \Rightarrow bool] \Rightarrow a$   
*GreatestM*  $m\ P == SOME\ x.\ P\ x \ \& (\forall y.\ P\ y \longrightarrow m\ y \leq m\ x)$

*Greatest* ::  $(a::ord \Rightarrow bool) \Rightarrow a$     (**binder** *GREATEST* 10)  
*Greatest* == *GreatestM* (% $x$ .  $x$ )

#### syntax

*GreatestM* ::  $[pttrn, a \Rightarrow b::ord, bool] \Rightarrow a$   
 (*GREATEST* - WRT -. - [0, 4, 10] 10)

#### translations

*GREATEST*  $x$  WRT  $m$ .  $P == GreatestM\ m\ (\%x.\ P)$

#### lemma *GreatestMI2*:

$P\ x \implies (!y.\ P\ y \implies m\ y \leq m\ x)$   
 $\implies (!x.\ P\ x \implies \forall y.\ P\ y \longrightarrow m\ y \leq m\ x \implies Q\ x)$   
 $\implies Q\ (GreatestM\ m\ P)$   
**apply** (*simp add: GreatestM-def*)  
**apply** (*rule someI2-ex, blast, blast*)  
**done**

#### lemma *GreatestM-equality*:

$P\ k \implies (!x.\ P\ x \implies m\ x \leq m\ k)$   
 $\implies m\ (GREATEST\ x\ WRT\ m.\ P\ x) = (m\ k::a::order)$   
**apply** (*rule-tac*  $m = m$  **in** *GreatestMI2*, *assumption*, *blast*)  
**apply** (*blast intro!: order-antisym*)  
**done**

#### lemma *Greatest-equality*:

$P\ (k::a::order) \implies (!x.\ P\ x \implies x \leq k) \implies (GREATEST\ x.\ P\ x) = k$   
**apply** (*simp add: Greatest-def*)  
**apply** (*erule GreatestM-equality, blast*)  
**done**

#### lemma *ex-has-greatest-nat-lemma*:

$P\ k \implies \forall x.\ P\ x \longrightarrow (\exists y.\ P\ y \ \& \sim ((m\ y::nat) \leq m\ x))$   
 $\implies \exists y.\ P\ y \ \& \sim (m\ y < m\ k + n)$   
**apply** (*induct*  $n$ , *force*)  
**apply** (*force simp add: le-Suc-eq*)  
**done**

#### lemma *ex-has-greatest-nat*:

```

P k ==> ∀ y. P y --> m y < b
==> ∃ x. P x & (∀ y. P y --> (m y::nat) <= m x)
apply (rule ccontr)
apply (cut-tac P = P and n = b - m k in ex-has-greatest-nat-lemma)
apply (subgoal-tac [3] m k <= b, auto)
done

lemma GreatestM-nat-lemma:
  P k ==> ∀ y. P y --> m y < b
  ==> P (GreatestM m P) & (∀ y. P y --> (m y::nat) <= m (GreatestM m
P))
apply (simp add: GreatestM-def)
apply (rule someI-ex)
apply (erule ex-has-greatest-nat, assumption)
done

lemmas GreatestM-natI = GreatestM-nat-lemma [THEN conjunct1, standard]

lemma GreatestM-nat-le:
  P x ==> ∀ y. P y --> m y < b
  ==> (m x::nat) <= m (GreatestM m P)
apply (blast dest: GreatestM-nat-lemma [THEN conjunct2, THEN spec, of P])
done

```

Specialization to *GREATEST*.

```

lemma GreatestI: P (k::nat) ==> ∀ y. P y --> y < b ==> P (GREATEST
x. P x)
apply (simp add: Greatest-def)
apply (rule GreatestM-natI, auto)
done

lemma Greatest-le:
  P x ==> ∀ y. P y --> y < b ==> (x::nat) <= (GREATEST x. P x)
apply (simp add: Greatest-def)
apply (rule GreatestM-nat-le, auto)
done

```

## 37.9 The Meson proof procedure

### 37.9.1 Negation Normal Form

de Morgan laws

```

lemma meson-not-conjD: ~(P & Q) ==> ~P | ~Q
and meson-not-disjD: ~(P | Q) ==> ~P & ~Q
and meson-not-notD: ~~P ==> P
and meson-not-allD: !!P. ~(∀ x. P(x)) ==> ∃ x. ~P(x)
and meson-not-exD: !!P. ~(∃ x. P(x)) ==> ∀ x. ~P(x)
by fast+

```

Removal of  $-->$  and  $<->$  (positive and negative occurrences)

**lemma** *meson-imp-to-disjD*:  $P-->Q ==> \sim P \mid Q$   
**and** *meson-not-impD*:  $\sim(P-->Q) ==> P \ \& \ \sim Q$   
**and** *meson-iff-to-disjD*:  $P=Q ==> (\sim P \mid Q) \ \& \ (\sim Q \mid P)$   
**and** *meson-not-iffD*:  $\sim(P=Q) ==> (P \mid Q) \ \& \ (\sim P \mid \sim Q)$   
 — Much more efficient than  $P \wedge \neg Q \vee Q \wedge \neg P$  for computing CNF  
**and** *meson-not-refl-disj-D*:  $x \sim = x \mid P ==> P$   
**by** *fast+*

### 37.9.2 Pulling out the existential quantifiers

Conjunction

**lemma** *meson-conj-exD1*:  $!!P \ Q. (\exists x. P(x)) \ \& \ Q ==> \exists x. P(x) \ \& \ Q$   
**and** *meson-conj-exD2*:  $!!P \ Q. P \ \& \ (\exists x. Q(x)) ==> \exists x. P \ \& \ Q(x)$   
**by** *fast+*

Disjunction

**lemma** *meson-disj-exD*:  $!!P \ Q. (\exists x. P(x)) \ \mid (\exists x. Q(x)) ==> \exists x. P(x) \ \mid \ Q(x)$   
 — DO NOT USE with forall-Skolemization: makes fewer schematic variables!!  
 — With ex-Skolemization, makes fewer Skolem constants  
**and** *meson-disj-exD1*:  $!!P \ Q. (\exists x. P(x)) \ \mid \ Q ==> \exists x. P(x) \ \mid \ Q$   
**and** *meson-disj-exD2*:  $!!P \ Q. P \ \mid (\exists x. Q(x)) ==> \exists x. P \ \mid \ Q(x)$   
**by** *fast+*

### 37.9.3 Generating clauses for the Meson Proof Procedure

Disjunctions

**lemma** *meson-disj-assoc*:  $(P \mid Q) \mid R ==> P \mid (Q \mid R)$   
**and** *meson-disj-comm*:  $P \mid Q ==> Q \mid P$   
**and** *meson-disj-FalseD1*:  $False \mid P ==> P$   
**and** *meson-disj-FalseD2*:  $P \mid False ==> P$   
**by** *fast+*

## 37.10 Lemmas for Meson, the Model Elimination Procedure

Generation of contrapositives

Inserts negated disjunct after removing the negation; P is a literal. Model elimination requires assuming the negation of every attempted subgoal, hence the negated disjuncts.

**lemma** *make-neg-rule*:  $\sim P \mid Q ==> ((\sim P ==> P) ==> Q)$   
**by** *blast*

Version for Plaisted’s ”Postive refinement” of the Meson procedure

**lemma** *make-refined-neg-rule*:  $\sim P \mid Q ==> (P ==> Q)$   
**by** *blast*

$P$  should be a literal

**lemma** *make-pos-rule*:  $P|Q \implies ((P \implies \sim P) \implies Q)$   
**by** *blast*

Versions of *make-neg-rule* and *make-pos-rule* that don’t insert new assumptions, for ordinary resolution.

**lemmas** *make-neg-rule'* = *make-refined-neg-rule*

**lemma** *make-pos-rule'*:  $[|P|Q; \sim P|] \implies Q$   
**by** *blast*

Generation of a goal clause – put away the final literal

**lemma** *make-neg-goal*:  $\sim P \implies ((\sim P \implies P) \implies \text{False})$   
**by** *blast*

**lemma** *make-pos-goal*:  $P \implies ((P \implies \sim P) \implies \text{False})$   
**by** *blast*

### 37.10.1 Lemmas for Forward Proof

There is a similarity to congruence rules

**lemma** *conj-forward*:  $[|P' \& Q'; P' \implies P; Q' \implies Q|] \implies P \& Q$   
**by** *blast*

**lemma** *disj-forward*:  $[|P'|Q'; P' \implies P; Q' \implies Q|] \implies P|Q$   
**by** *blast*

**lemma** *disj-forward2*:  
 $[|P'|Q'; P' \implies P; [|Q'; P \implies \text{False}|] \implies Q|] \implies P|Q$   
**apply** *blast*  
**done**

**lemma** *all-forward*:  $[|\forall x. P'(x); !x. P'(x) \implies P(x)|] \implies \forall x. P(x)$   
**by** *blast*

**lemma** *ex-forward*:  $[|\exists x. P'(x); !x. P'(x) \implies P(x)|] \implies \exists x. P(x)$   
**by** *blast*

Many of these bindings are used by the ATP linkup, and not just by legacy proof scripts.

**ML**

$\ll$   
 $\text{val inv-def} = \text{thm inv-def};$   
 $\text{val Inv-def} = \text{thm Inv-def};$

$\text{val someI} = \text{thm someI};$

```

val someI-ex = thm someI-ex;
val someI2 = thm someI2;
val someI2-ex = thm someI2-ex;
val some-equality = thm some-equality;
val some1-equality = thm some1-equality;
val some-eq-ex = thm some-eq-ex;
val some-eq-trivial = thm some-eq-trivial;
val some-sym-eq-trivial = thm some-sym-eq-trivial;
val choice = thm choice;
val bchoice = thm bchoice;
val inv-id = thm inv-id;
val inv-f-f = thm inv-f-f;
val inv-f-eq = thm inv-f-eq;
val inj-imp-inv-eq = thm inj-imp-inv-eq;
val inj-transfer = thm inj-transfer;
val inj-iff = thm inj-iff;
val inj-imp-surj-inv = thm inj-imp-surj-inv;
val f-inv-f = thm f-inv-f;
val surj-f-inv-f = thm surj-f-inv-f;
val inv-injective = thm inv-injective;
val inj-on-inv = thm inj-on-inv;
val surj-imp-inj-inv = thm surj-imp-inj-inv;
val surj-iff = thm surj-iff;
val surj-imp-inv-eq = thm surj-imp-inv-eq;
val bij-imp-bij-inv = thm bij-imp-bij-inv;
val inv-equality = thm inv-equality;
val inv-inv-eq = thm inv-inv-eq;
val o-inv-distrib = thm o-inv-distrib;
val image-surj-f-inv-f = thm image-surj-f-inv-f;
val image-inv-f-f = thm image-inv-f-f;
val inv-image-comp = thm inv-image-comp;
val bij-image-Collect-eq = thm bij-image-Collect-eq;
val bij-vimage-eq-inv-image = thm bij-vimage-eq-inv-image;
val Inv-f-f = thm Inv-f-f;
val f-Inv-f = thm f-Inv-f;
val Inv-injective = thm Inv-injective;
val inj-on-Inv = thm inj-on-Inv;
val split-paired-Eps = thm split-paired-Eps;
val Eps-split = thm Eps-split;
val Eps-split-eq = thm Eps-split-eq;
val wf-iff-no-infinite-down-chain = thm wf-iff-no-infinite-down-chain;
val Inv-mem = thm Inv-mem;
val Inv-f-eq = thm Inv-f-eq;
val Inv-comp = thm Inv-comp;
val tfl-some = thm tfl-some;
val make-neg-rule = thm make-neg-rule;
val make-refined-neg-rule = thm make-refined-neg-rule;
val make-pos-rule = thm make-pos-rule;
val make-neg-rule' = thm make-neg-rule';

```

```

val make-pos-rule' = thm make-pos-rule';
val make-neg-goal = thm make-neg-goal;
val make-pos-goal = thm make-pos-goal;
val conj-forward = thm conj-forward;
val disj-forward = thm disj-forward;
val disj-forward2 = thm disj-forward2;
val all-forward = thm all-forward;
val ex-forward = thm ex-forward;
>>

```

### 37.11 Meson package

```
use Tools/meson.ML
```

```
setup Meson.setup
```

### 37.12 Specification package – Hilbertized version

```

lemma exE-some: [| Ex P ; c == Eps P |] ==> P c
  by (simp only: someI-ex)

```

```
use Tools/specification-package.ML
```

```
end
```

## 38 ATP-Linkup: The Isabelle-ATP Linkup

```
theory ATP-Linkup
```

```
imports Divides Record Hilbert-Choice Plain
```

```
uses
```

```

  Tools/polyhash.ML
  Tools/res-clause.ML
  (Tools/res-axioms.ML)
  (Tools/res-hol-clause.ML)
  (Tools/res-reconstruct.ML)
  (Tools/res-atp.ML)
  (Tools/atp-manager.ML)
  (Tools/atp-wrapper.ML)
  ~/src/Tools/Metis/metis.ML
  (Tools/metis-tools.ML)

```

```
begin
```

```

definition COMBI :: 'a ==> 'a
  where COMBI P == P

```

```

definition COMBK :: 'a ==> 'b ==> 'a
  where COMBK P Q == P

```

**definition** *COMBB* :: ( $'b \Rightarrow 'c \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$   
**where** *COMBB*  $P\ Q\ R == P\ (Q\ R)$

**definition** *COMBC* :: ( $'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'b \Rightarrow 'a \Rightarrow 'c$   
**where** *COMBC*  $P\ Q\ R == P\ R\ Q$

**definition** *COMBS* :: ( $'a \Rightarrow 'b \Rightarrow 'c \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$   
**where** *COMBS*  $P\ Q\ R == P\ R\ (Q\ R)$

**definition** *fequal* ::  $'a \Rightarrow 'a \Rightarrow \text{bool}$   
**where** *fequal*  $X\ Y == (X=Y)$

**lemma** *fequal-imp-equal*:  $\text{fequal}\ X\ Y \Rightarrow X=Y$   
**by** (*simp add: fequal-def*)

**lemma** *equal-imp-fequal*:  $X=Y \Rightarrow \text{fequal}\ X\ Y$   
**by** (*simp add: fequal-def*)

These two represent the equivalence between Boolean equality and iff. They can't be converted to clauses automatically, as the iff would be expanded...

**lemma** *iff-positive*:  $P \mid Q \mid P=Q$   
**by** *blast*

**lemma** *iff-negative*:  $\sim P \mid \sim Q \mid P=Q$   
**by** *blast*

Theorems for translation to combinators

**lemma** *abs-S*:  $(\%x. (f\ x)\ (g\ x)) == \text{COMBS}\ f\ g$   
**apply** (*rule eq-reflection*)  
**apply** (*rule ext*)  
**apply** (*simp add: COMBS-def*)  
**done**

**lemma** *abs-I*:  $(\%x. x) == \text{COMBI}$   
**apply** (*rule eq-reflection*)  
**apply** (*rule ext*)  
**apply** (*simp add: COMBI-def*)  
**done**

**lemma** *abs-K*:  $(\%x. y) == \text{COMBK}\ y$   
**apply** (*rule eq-reflection*)  
**apply** (*rule ext*)  
**apply** (*simp add: COMBK-def*)  
**done**

**lemma** *abs-B*:  $(\%x. a\ (g\ x)) == \text{COMBB}\ a\ g$   
**apply** (*rule eq-reflection*)  
**apply** (*rule ext*)  
**apply** (*simp add: COMBB-def*)

done

```
lemma abs-C: (%x. (f x) b) == COMBC f b
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBC-def)
done
```

### 38.1 Setup of external ATPs

```
use Tools/res-axioms.ML setup ResAxioms.setup
use Tools/res-hol-clause.ML
use Tools/res-reconstruct.ML setup ResReconstruct.setup
use Tools/res-atp.ML
```

```
use Tools/atp-manager.ML
use Tools/atp-wrapper.ML
```

basic provers

```
setup << AtpManager.add-prover spass AtpWrapper.spass >>
setup << AtpManager.add-prover vampire AtpWrapper.vampire >>
setup << AtpManager.add-prover e AtpWrapper.e prover >>
```

provers with structured output

```
setup << AtpManager.add-prover vampire-full AtpWrapper.vampire-full >>
setup << AtpManager.add-prover e-full AtpWrapper.e prover-full >>
```

on some problems better results

```
setup << AtpManager.add-prover spass-no-tc (AtpWrapper.spass-opts 40 false) >>
```

remote provers via SystemOnTPTP

```
setup << AtpManager.add-prover remote-vampire
  (AtpWrapper.remote-prover -s Vampire---9.0) >>
setup << AtpManager.add-prover remote-spass
  (AtpWrapper.remote-prover -s SPASS---3.01) >>
setup << AtpManager.add-prover remote-e
  (AtpWrapper.remote-prover -s EP---1.0) >>
```

### 38.2 The Metis prover

```
use Tools/metis-tools.ML
setup MetisTools.setup
```

end

## 39 List: The datatype of finite lists

theory List



```

imports Plain Relation-Power Presburger Recdef ATP-Linkup
uses Tools/string-syntax.ML
begin

```

```

datatype 'a list =
  Nil    ([])
  | Cons 'a 'a list  (infixr # 65)

```

### 39.1 Basic list processing functions

#### consts

```

filter:: ('a => bool) => 'a list => 'a list
concat:: 'a list list => 'a list
foldl :: ('b => 'a => 'b) => 'b => 'a list => 'b
foldr :: ('a => 'b => 'b) => 'a list => 'b => 'b
hd:: 'a list => 'a
tl:: 'a list => 'a list
last:: 'a list => 'a
butlast :: 'a list => 'a list
set :: 'a list => 'a set
map :: ('a=>'b) => ('a list => 'b list)
listsum :: 'a list => 'a::monoid-add
list-update :: 'a list => nat => 'a => 'a list
take:: nat => 'a list => 'a list
drop:: nat => 'a list => 'a list
takeWhile :: ('a => bool) => 'a list => 'a list
dropWhile :: ('a => bool) => 'a list => 'a list
rev :: 'a list => 'a list
zip :: 'a list => 'b list => ('a * 'b) list
upt :: nat => nat => nat list ((1[-.</-]))
remdups :: 'a list => 'a list
remove1 :: 'a => 'a list => 'a list
removeAll :: 'a => 'a list => 'a list
distinct:: 'a list => bool
replicate :: nat => 'a => 'a list
splice :: 'a list => 'a list => 'a list

```

**nonterminals** lupdbinds lupdbind

#### syntax

```

— list Enumeration
@list :: args => 'a list  ([[(-)])

— Special syntax for filter
@filter :: [pttrn, 'a list, bool] => 'a list  ((1[-<--./ -]))

— list update
-lupdbind:: ['a, 'a] => lupdbind  ((2- :=/ -))

```

```

:: lupdbind => lupdbinds    (-)
-lupdbinds :: [lupdbind, lupdbinds] => lupdbinds    (-,/ -)
-LUpdate :: ['a, lupdbinds] => 'a    (-/[(-) [900,0] 900)

```

**translations**

```

[x, xs] == x#[xs]
[x] == x#[ ]
[x<-xs . P] == filter (%x. P) xs

-LUpdate xs (-lupdbinds b bs) == -LUpdate (-LUpdate xs b) bs
xs[i:=x] == list-update xs i x

```

**syntax** (*xsymbols*)

```
@filter :: [pttrn, 'a list, bool] => 'a list((1[-<- - ./ -]))
```

**syntax** (*HTML output*)

```
@filter :: [pttrn, 'a list, bool] => 'a list((1[-<- - ./ -]))
```

Function *size* is overloaded for all datatypes. Users may refer to the list version as *length*.

**abbreviation**

```

length :: 'a list => nat where
length == size

```

**primrec**

```
hd(x#xs) = x
```

**primrec**

```

tl([]) = []
tl(x#xs) = xs

```

**primrec**

```
last(x#xs) = (if xs=[] then x else last xs)
```

**primrec**

```

butlast [] = []
butlast(x#xs) = (if xs=[] then [] else x#butlast xs)

```

**primrec**

```

set [] = {}
set (x#xs) = insert x (set xs)

```

**primrec**

```

map f [] = []
map f (x#xs) = f(x)#map f xs

```

**primrec**

```
append :: 'a list => 'a list => 'a list (infixr @ 65)
```

**where**

*append-Nil*:  $[] @ ys = ys$   
*append-Cons*:  $(x \# xs) @ ys = x \# xs @ ys$

**primrec**

*rev*( $[]$ ) =  $[]$   
*rev*( $x \# xs$ ) = *rev*( $xs$ ) @  $[x]$

**primrec**

*filter*  $P$   $[]$  =  $[]$   
*filter*  $P$  ( $x \# xs$ ) = (if  $P$   $x$  then  $x \# \text{filter } P \text{ } xs$  else *filter*  $P$   $xs$ )

**primrec**

*foldl-Nil*: *foldl*  $f$   $a$   $[]$  =  $a$   
*foldl-Cons*: *foldl*  $f$   $a$  ( $x \# xs$ ) = *foldl*  $f$  ( $f$   $a$   $x$ )  $xs$

**primrec**

*foldr*  $f$   $[]$   $a$  =  $a$   
*foldr*  $f$  ( $x \# xs$ )  $a$  =  $f$   $x$  (*foldr*  $f$   $xs$   $a$ )

**primrec**

*concat*( $[]$ ) =  $[]$   
*concat*( $x \# xs$ ) =  $x @ \text{concat}(xs)$

**primrec**

*listsum*  $[]$  = 0  
*listsum* ( $x \# xs$ ) =  $x + \text{listsum } xs$

**primrec**

*drop-Nil*: *drop*  $n$   $[]$  =  $[]$   
*drop-Cons*: *drop*  $n$  ( $x \# xs$ ) = (case  $n$  of 0 =>  $x \# xs$  | *Suc*( $m$ ) => *drop*  $m$   $xs$ )  
— Warning: simpset does not contain this definition, but separate theorems for  $n = 0$  and  $n = \text{Suc } k$

**primrec**

*take-Nil*: *take*  $n$   $[]$  =  $[]$   
*take-Cons*: *take*  $n$  ( $x \# xs$ ) = (case  $n$  of 0 =>  $[]$  | *Suc*( $m$ ) =>  $x \# \text{take } m \text{ } xs$ )  
— Warning: simpset does not contain this definition, but separate theorems for  $n = 0$  and  $n = \text{Suc } k$

**primrec** *nth* :: 'a list => nat => 'a (**infixl** ! 100) **where**

*nth-Cons*:  $(x \# xs) ! n$  = (case  $n$  of 0 =>  $x$  | (*Suc*  $k$ ) =>  $xs ! k$ )  
— Warning: simpset does not contain this definition, but separate theorems for  $n = 0$  and  $n = \text{Suc } k$

**primrec**

$[] [i := v]$  =  $[]$   
 $(x \# xs) [i := v]$  = (case  $i$  of 0 =>  $v \# xs$  | *Suc*  $j$  =>  $x \# xs [j := v]$ )

**primrec**

*takeWhile*  $P \ [] = []$   
*takeWhile*  $P \ (x\#xs) = (\text{if } P \ x \text{ then } x\#\text{takeWhile } P \ xs \text{ else } [])$

**primrec**

*dropWhile*  $P \ [] = []$   
*dropWhile*  $P \ (x\#xs) = (\text{if } P \ x \text{ then } \text{dropWhile } P \ xs \text{ else } x\#xs)$

**primrec**

*zip*  $xs \ [] = []$   
*zip-Cons*:  $\text{zip } xs \ (y\#ys) = (\text{case } xs \text{ of } [] \Rightarrow [] \mid z\#zs \Rightarrow (z,y)\#\text{zip } zs \ ys)$   
 — Warning: simpset does not contain this definition, but separate theorems for  
 $xs = []$  and  $xs = z \# zs$

**primrec**

*upt-0*:  $[i..<0] = []$   
*upt-Suc*:  $[i..<(\text{Suc } j)] = (\text{if } i \leq j \text{ then } [i..<j] @ [j] \text{ else } [])$

**primrec**

*distinct*  $[] = \text{True}$   
*distinct*  $(x\#xs) = (x \sim: \text{set } xs \wedge \text{distinct } xs)$

**primrec**

*remdups*  $[] = []$   
*remdups*  $(x\#xs) = (\text{if } x : \text{set } xs \text{ then } \text{remdups } xs \text{ else } x \# \text{remdups } xs)$

**primrec**

*remove1*  $x \ [] = []$   
*remove1*  $x \ (y\#xs) = (\text{if } x=y \text{ then } xs \text{ else } y \# \text{remove1 } x \ xs)$

**primrec**

*removeAll*  $x \ [] = []$   
*removeAll*  $x \ (y\#xs) = (\text{if } x=y \text{ then } \text{removeAll } x \ xs \text{ else } y \# \text{removeAll } x \ xs)$

**primrec**

*replicate-0*:  $\text{replicate } 0 \ x = []$   
*replicate-Suc*:  $\text{replicate } (\text{Suc } n) \ x = x \# \text{replicate } n \ x$

**definition**

*rotate1*  $:: 'a \text{ list} \Rightarrow 'a \text{ list}$  **where**  
*rotate1*  $xs = (\text{case } xs \text{ of } [] \Rightarrow [] \mid x\#xs \Rightarrow xs @ [x])$

**definition**

*rotate*  $:: \text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$  **where**  
*rotate*  $n = \text{rotate1} \ ^n$

**definition**

*list-all2*  $:: ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow \text{bool}$  **where**  
 $[\text{code del}]: \text{list-all2 } P \ xs \ ys =$   
 $(\text{length } xs = \text{length } ys \wedge (\forall (x, y) \in \text{set } (\text{zip } xs \ ys). P \ x \ y))$

**definition**

*sublist* :: 'a list => nat set => 'a list **where**  
*sublist* *xs* *A* = *map fst (filter (λp. snd p ∈ A) (zip xs [0..*size xs*]))*

**primrec**

*splice* [] *ys* = *ys*  
*splice* (*x#xs*) *ys* = (if *ys*=[] then *x#xs* else *x* # *hd ys* # *splice xs (tl ys)*)  
 — Warning: *simpset* does not contain the second eqn but a derived one.

Figure 1 shows characteristic examples that should give an intuitive understanding of the above functions.

The following simple sort functions are intended for proofs, not for efficient implementations.

**context** *linorder*  
**begin**

**fun** *sorted* :: 'a list ⇒ bool **where**  
*sorted* [] ⇔ True |  
*sorted* [*x*] ⇔ True |  
*sorted* (*x#y#zs*) ⇔ *x* ≤ *y* ∧ *sorted* (*y#zs*)

**primrec** *insort* :: 'a ⇒ 'a list ⇒ 'a list **where**  
*insort* *x* [] = [*x*] |  
*insort* *x* (*y#ys*) = (if *x* ≤ *y* then (*x#y#ys*) else *y#(insort x ys)*)

**primrec** *sort* :: 'a list ⇒ 'a list **where**  
*sort* [] = [] |  
*sort* (*x#xs*) = *insort x (sort xs)*

**end**

### 39.1.1 List comprehension

Input syntax for Haskell-like list comprehension notation. Typical example:  $[(x,y). x \leftarrow xs, y \leftarrow ys, x \neq y]$ , the list of all pairs of distinct elements from *xs* and *ys*. The syntax is as in Haskell, except that | becomes a dot (like in Isabelle’s set comprehension):  $[e. x \leftarrow xs, \dots]$  rather than  $[e \mid x \leftarrow xs, \dots]$ .

The qualifiers after the dot are

**generators**  $p \leftarrow xs$ , where *p* is a pattern and *xs* an expression of list type,  
 or

**guards** *b*, where *b* is a boolean expression.

Just like in Haskell, list comprehension is just a shorthand. To avoid misunderstandings, the translation into desugared form is not reversed upon

```

[a, b] @ [c, d] = [a, b, c, d]
length [a, b, c] = 3
set [a, b, c] = {a, b, c}
map f [a, b, c] = [f a, f b, f c]
rev [a, b, c] = [c, b, a]
hd [a, b, c, d] = a
tl [a, b, c, d] = [b, c, d]
last [a, b, c, d] = d
butlast [a, b, c, d] = [a, b, c]
filter (λn::nat. n < 2) [0, 2, 1] = [0, 1]
concat [[a, b], [c, d, e], [], [f]] = [a, b, c, d, e, f]
foldl f x [a, b, c] = f (f (f x a) b) c
foldr f [a, b, c] x = f a (f b (f c x))
zip [a, b, c] [x, y, z] = [(a, x), (b, y), (c, z)]
zip [a, b] [x, y, z] = [(a, x), (b, y)]
splice [a, b, c] [x, y, z] = [a, x, b, y, c, z]
splice [a, b, c, d] [x, y] = [a, x, b, y, c, d]
take 2 [a, b, c, d] = [a, b]
take 6 [a, b, c, d] = [a, b, c, d]
drop 2 [a, b, c, d] = [c, d]
drop 6 [a, b, c, d] = []
takeWhile (λn. n < 3) [1, 2, 3, 0] = [1, 2]
dropWhile (λn. n < 3) [1, 2, 3, 0] = [3, 0]
distinct [2, 0, 1]
remdups [2, 0, 2, 1, 2] = [0, 1, 2]
remove1 2 [2, 0, 2, 1, 2] = [0, 2, 1, 2]
removeAll 2 [2, 0, 2, 1, 2] = [0, 1]
[a, b, c, d] ! 2 = c
[a, b, c, d][2 := x] = [a, b, x, d]
sublist [a, b, c, d, e] {0, 2, 3} = [a, c, d]
rotate1 [a, b, c, d] = [b, c, d, a]
rotate 3 [a, b, c, d] = [d, a, b, c]
replicate 4 a = [a, a, a, a]
[2..<5] = [2, 3, 4]
listsum [1, 2, 3] = 6

```

Figure 1: Characteristic examples

output. Note that the translation of  $[e. x \leftarrow xs]$  is optimized to  $map (\lambda x. e) xs$ .

It is easy to write short list comprehensions which stand for complex expressions. During proofs, they may become unreadable (and mangled). In such cases it can be advisable to introduce separate definitions for the list comprehensions in question.

**nonterminals** *lc-qual lc-quals*

**syntax**

```
-listcompr :: 'a ⇒ lc-qual ⇒ lc-quals ⇒ 'a list  ([- . --)
-lc-gen  :: 'a ⇒ 'a list ⇒ lc-qual  (- <- -)
-lc-test :: bool ⇒ lc-qual  (-)
```

```
-lc-end  :: lc-quals  ()
-lc-quals :: lc-qual ⇒ lc-quals ⇒ lc-quals  (, --)
-lc-abs  :: 'a => 'b list => 'b list
```

**syntax** (*xsymbols*)

```
-lc-gen :: 'a ⇒ 'a list ⇒ lc-qual  (- <- -)
```

**syntax** (*HTML output*)

```
-lc-gen :: 'a ⇒ 'a list ⇒ lc-qual  (- <- -)
```

**parse-translation** (*advanced*)  $\ll$

*let*

```
val NilC = Syntax.const @{const-name Nil};
val ConsC = Syntax.const @{const-name Cons};
val mapC = Syntax.const @{const-name map};
val concatC = Syntax.const @{const-name concat};
val IfC = Syntax.const @{const-name If};
fun singl x = ConsC $ x $ NilC;
```

```
fun pat-tr ctxt p e opti = (* %x. case x of p => e | - => [] *)
```

*let*

```
val x = Free (Name.variant (fold Term.add-free-names [p, e] []) x, dummyT);
val e = if opti then singl e else e;
val case1 = Syntax.const -case1 $ p $ e;
val case2 = Syntax.const -case1 $ Syntax.const Term.dummy-patternN
              $ NilC;
val cs = Syntax.const -case2 $ case1 $ case2
val ft = DatatypeCase.case-tr false DatatypePackage.datatype-of-constr
          ctxt [x, cs]
```

*in lambda x ft end;*

```
fun abs-tr ctxt (p as Free(s,T)) e opti =
  let val thy = ProofContext.theory-of ctxt;
      val s' = Sign.intern-const thy s
```

```

      in if Sign.declared-const thy s'
        then (pat-tr ctxt p e opti, false)
        else (lambda p e, true)
      end
    | abs-tr ctxt p e opti = (pat-tr ctxt p e opti, false);

fun lc-tr ctxt [e, Const(-lc-test,-)]$b, qs] =
  let val res = case qs of Const(-lc-end,-) => singl e
    | Const(-lc-quals,-)$q$qs => lc-tr ctxt [e,q,qs];
  in IfC $ b $ res $ NilC end
| lc-tr ctxt [e, Const(-lc-gen,-)] $ p $ es, Const(-lc-end,-)] =
  (case abs-tr ctxt p e true of
    (f,true) => mapC $ f $ es
    | (f,false) => concatC $ (mapC $ f $ es))
| lc-tr ctxt [e, Const(-lc-gen,-)] $ p $ es, Const(-lc-quals,-)$q$qs] =
  let val e' = lc-tr ctxt [e,q,qs];
  in concatC $ (mapC $ (fst(abs-tr ctxt p e' false)) $ es) end

in [(-listcompr, lc-tr)] end
>>

```

### 39.1.2 [] and op #

**lemma** *not-Cons-self* [simp]:

$xs \neq x \# xs$

**by** (induct xs) auto

**lemmas** *not-Cons-self2* [simp] = *not-Cons-self* [symmetric]

**lemma** *neq-Nil-conv*:  $(xs \neq []) = (\exists y \text{ ys. } xs = y \# \text{ ys})$

**by** (induct xs) auto

**lemma** *length-induct*:

$(\bigwedge xs. \forall \text{ ys. } \text{length } \text{ys} < \text{length } xs \longrightarrow P \text{ ys} \Longrightarrow P \text{ xs}) \Longrightarrow P \text{ xs}$

**by** (rule *measure-induct* [of length]) iprover

### 39.1.3 length

Needs to come before @ because of theorem *append-eq-append-conv*.

**lemma** *length-append* [simp]:  $\text{length } (xs @ \text{ys}) = \text{length } xs + \text{length } \text{ys}$

**by** (induct xs) auto

**lemma** *length-map* [simp]:  $\text{length } (\text{map } f \text{ xs}) = \text{length } \text{xs}$

**by** (induct xs) auto

**lemma** *length-rev* [simp]:  $\text{length } (\text{rev } \text{xs}) = \text{length } \text{xs}$

**by** (induct xs) auto

**lemma** *length-tl* [simp]:  $\text{length } (\text{tl } \text{xs}) = \text{length } \text{xs} - 1$



**by** (*cases xs*) *auto*

**lemma** *length-0-conv* [*iff*]:  $(\text{length } xs = 0) = (xs = [])$   
**by** (*induct xs*) *auto*

**lemma** *length-greater-0-conv* [*iff*]:  $(0 < \text{length } xs) = (xs \neq [])$   
**by** (*induct xs*) *auto*

**lemma** *length-pos-if-in-set*:  $x : \text{set } xs \implies \text{length } xs > 0$   
**by** *auto*

**lemma** *length-Suc-conv*:  
 $(\text{length } xs = \text{Suc } n) = (\exists y \text{ } ys. xs = y \# ys \wedge \text{length } ys = n)$   
**by** (*induct xs*) *auto*

**lemma** *Suc-length-conv*:  
 $(\text{Suc } n = \text{length } xs) = (\exists y \text{ } ys. xs = y \# ys \wedge \text{length } ys = n)$   
**apply** (*induct xs, simp, simp*)  
**apply** *blast*  
**done**

**lemma** *impossible-Cons*:  $\text{length } xs <= \text{length } ys \implies xs = x \# ys = \text{False}$   
**by** (*induct xs*) *auto*

**lemma** *list-induct2* [*consumes 1, case-names Nil Cons*]:  
 $\text{length } xs = \text{length } ys \implies P [] [] \implies$   
 $(\bigwedge x \text{ } xs \text{ } y \text{ } ys. \text{length } xs = \text{length } ys \implies P \text{ } xs \text{ } ys \implies P (x \# xs) (y \# ys))$   
 $\implies P \text{ } xs \text{ } ys$   
**proof** (*induct xs arbitrary: ys*)  
**case** *Nil* **then show** ?*case* **by** *simp*  
**next**  
**case** (*Cons x xs ys*) **then show** ?*case* **by** (*cases ys*) *simp-all*  
**qed**

**lemma** *list-induct3* [*consumes 2, case-names Nil Cons*]:  
 $\text{length } xs = \text{length } ys \implies \text{length } ys = \text{length } zs \implies P [] [] [] \implies$   
 $(\bigwedge x \text{ } xs \text{ } y \text{ } ys \text{ } z \text{ } zs. \text{length } xs = \text{length } ys \implies \text{length } ys = \text{length } zs \implies P \text{ } xs \text{ } ys \text{ } zs$   
 $\implies P (x \# xs) (y \# ys) (z \# zs))$   
 $\implies P \text{ } xs \text{ } ys \text{ } zs$   
**proof** (*induct xs arbitrary: ys zs*)  
**case** *Nil* **then show** ?*case* **by** *simp*  
**next**  
**case** (*Cons x xs ys zs*) **then show** ?*case* **by** (*cases ys, simp-all*)  
*(cases zs, simp-all)*  
**qed**

**lemma** *list-induct2'*:  
 $\llbracket P [] [];$   
 $\bigwedge x \text{ } xs. P (x \# xs) [];$

```

 $\bigwedge y \text{ } ys. P \ [] \ (y \# ys);$ 
 $\bigwedge x \text{ } xs \text{ } y \text{ } ys. P \ xs \ ys \implies P \ (x \# xs) \ (y \# ys) \ []$ 
 $\implies P \ xs \ ys$ 
by (induct xs arbitrary: ys) (case-tac x, auto)+

```

**lemma** *neq-if-length-neq*:  $\text{length } xs \neq \text{length } ys \implies (xs = ys) == \text{False}$   
**by** (rule Eq-FalseI) auto

**simproc-setup** *list-neq* ((xs::'a list) = ys) =  $\langle\langle$   
 (\*  
*Reduces xs=ys to False if xs and ys cannot be of the same length.*  
*This is the case if the atomic sublists of one are a submultiset*  
*of those of the other list and there are fewer Cons's in one than the other.*  
 \*)

let

```

fun len (Const(@{const-name Nil},-)) acc = acc
| len (Const(@{const-name Cons},-) $ - $ xs) (ts,n) = len xs (ts,n+1)
| len (Const(@{const-name append},-) $ xs $ ys) acc = len xs (len ys acc)
| len (Const(@{const-name rev},-) $ xs) acc = len xs acc
| len (Const(@{const-name map},-) $ - $ xs) acc = len xs acc
| len t (ts,n) = (t::ts,n);

```

fun list-neq - ss ct =

```

let
  val (Const(-,eqT) $ lhs $ rhs) = Thm.term-of ct;
  val (ls,m) = len lhs ([],0) and (rs,n) = len rhs ([],0);
  fun prove-neq() =
    let
      val Type(-,listT::-) = eqT;
      val size = HOLogic.size-const listT;
      val eq-len = HOLogic.mk-eq (size $ lhs, size $ rhs);
      val neq-len = HOLogic.mk-Trueprop (HOLogic.Not $ eq-len);
      val thm = Goal.prove (Simplifier.the-context ss) [] [] neq-len
        (K (simp-tac (Simplifier.inherit-context ss @ {simpset}) 1));
      in SOME (thm RS @ {thm neq-if-length-neq}) end
    in
      if m < n andalso submultiset (op aconv) (ls,rs) orelse
        n < m andalso submultiset (op aconv) (rs,ls)
      then prove-neq() else NONE
    end;
in list-neq end;

```

### 39.1.4 @ – append

**lemma** *append-assoc* [simp]:  $(xs @ ys) @ zs = xs @ (ys @ zs)$   
**by** (induct xs) auto

**lemma** *append-Nil2* [*simp*]:  $xs @ [] = xs$

**by** (*induct xs*) *auto*

**lemma** *append-is-Nil-conv* [*iff*]:  $(xs @ ys = []) = (xs = [] \wedge ys = [])$

**by** (*induct xs*) *auto*

**lemma** *Nil-is-append-conv* [*iff*]:  $([] = xs @ ys) = (xs = [] \wedge ys = [])$

**by** (*induct xs*) *auto*

**lemma** *append-self-conv* [*iff*]:  $(xs @ ys = xs) = (ys = [])$

**by** (*induct xs*) *auto*

**lemma** *self-append-conv* [*iff*]:  $(xs = xs @ ys) = (ys = [])$

**by** (*induct xs*) *auto*

**lemma** *append-eq-append-conv* [*simp*, *noatp*]:

$length\ xs = length\ ys \vee length\ us = length\ vs$

$\implies (xs @ us = ys @ vs) = (xs = ys \wedge us = vs)$

**apply** (*induct xs arbitrary: ys*)

**apply** (*case-tac ys, simp, force*)

**apply** (*case-tac ys, force, simp*)

**done**

**lemma** *append-eq-append-conv2*:  $(xs @ ys = zs @ ts) =$

$(EX\ us.\ xs = zs @ us \ \&\ us @ ys = ts \mid xs @ us = zs \ \&\ ys = us @ ts)$

**apply** (*induct xs arbitrary: ys zs ts*)

**apply** *fastsimp*

**apply** (*case-tac zs*)

**apply** *simp*

**apply** *fastsimp*

**done**

**lemma** *same-append-eq* [*iff*]:  $(xs @ ys = xs @ zs) = (ys = zs)$

**by** *simp*

**lemma** *append1-eq-conv* [*iff*]:  $(xs @ [x] = ys @ [y]) = (xs = ys \wedge x = y)$

**by** *simp*

**lemma** *append-same-eq* [*iff*]:  $(ys @ xs = zs @ xs) = (ys = zs)$

**by** *simp*

**lemma** *append-self-conv2* [*iff*]:  $(xs @ ys = ys) = (xs = [])$

**using** *append-same-eq* [*of* - -  $[]$ ] **by** *auto*

**lemma** *self-append-conv2* [*iff*]:  $(ys = xs @ ys) = (xs = [])$

**using** *append-same-eq* [*of*  $[]$ ] **by** *auto*

**lemma** *hd-Cons-tl* [*simp*, *noatp*]:  $xs \neq [] \implies hd\ xs \# tl\ xs = xs$

**by** (*induct xs*) *auto*

**lemma** *hd-append*:  $hd\ (xs\ @\ ys) = (if\ xs = []\ then\ hd\ ys\ else\ hd\ xs)$   
**by** (*induct xs*) *auto*

**lemma** *hd-append2* [*simp*]:  $xs \neq [] \implies hd\ (xs\ @\ ys) = hd\ xs$   
**by** (*simp add: hd-append split: list.split*)

**lemma** *tl-append*:  $tl\ (xs\ @\ ys) = (case\ xs\ of\ [] \Rightarrow tl\ ys \mid z\#\!zs \Rightarrow zs\ @\ ys)$   
**by** (*simp split: list.split*)

**lemma** *tl-append2* [*simp*]:  $xs \neq [] \implies tl\ (xs\ @\ ys) = tl\ xs\ @\ ys$   
**by** (*simp add: tl-append split: list.split*)

**lemma** *Cons-eq-append-conv*:  $x\#\!xs = ys@zs =$   
 $(ys = [] \ \&\ x\#\!xs = zs \mid (EX\ ys'.\ x\#\!ys' = ys \ \&\ xs = ys'\!@zs))$   
**by**(*cases ys*) *auto*

**lemma** *append-eq-Cons-conv*:  $(ys@zs = x\#\!xs) =$   
 $(ys = [] \ \&\ zs = x\#\!xs \mid (EX\ ys'.\ ys = x\#\!ys' \ \&\ ys'\!@zs = xs))$   
**by**(*cases ys*) *auto*

Trivial rules for solving @-equations automatically.

**lemma** *eq-Nil-appendI*:  $xs = ys \implies xs = []\ @\ ys$   
**by** *simp*

**lemma** *Cons-eq-appendI*:  
 $[| x\ \#\!xs1 = ys; xs = xs1\ @\ zs |] \implies x\ \#\!xs = ys\ @\ zs$   
**by** (*drule sym*) *simp*

**lemma** *append-eq-appendI*:  
 $[| xs\ @\ xs1 = zs; ys = xs1\ @\ us |] \implies xs\ @\ ys = zs\ @\ us$   
**by** (*drule sym*) *simp*

Simplification procedure for all list equalities. Currently only tries to rearrange @ to see if - both lists end in a singleton list, - or both lists end in the same list.

**ML**  $\langle\langle$   
*local*

*fun* *last* (*cons as Const*(@{*const-name Cons*},-) \$ - \$ *xs*) =  
 $(case\ xs\ of\ Const(@\{const-name\ Nil\},-) \Rightarrow cons \mid - \Rightarrow last\ xs)$   
 $\mid last\ (Const(@\{const-name\ append\},-) \$ - \$ ys) = last\ ys$   
 $\mid last\ t = t;$

*fun* *list1* (*Const*(@{*const-name Cons*},-) \$ - \$ *Const*(@{*const-name Nil*},-)) = *true*  
 $\mid list1\ - = false;$

```

fun butlast ((cons as Const(@{const-name Cons},-) $ x) $ xs) =
  (case xs of Const(@{const-name Nil},-) => xs | - => cons $ butlast xs)
| butlast ((app as Const(@{const-name append},-) $ xs) $ ys) = app $ butlast ys
| butlast xs = Const(@{const-name Nil},fastype-of xs);

val rearr-ss = HOL-basic-ss addsimps [@{thm append-assoc},
  @{thm append-Nil}, @{thm append-Cons}];

fun list-eq ss (F as (eq as Const(-,eqT)) $ lhs $ rhs) =
  let
    val lastl = last lhs and lastr = last rhs;
    fun rearr conv =
      let
        val lhs1 = butlast lhs and rhs1 = butlast rhs;
        val Type(-,listT::-) = eqT
        val appT = [listT,listT] ---> listT
        val app = Const(@{const-name append},appT)
        val F2 = eq $ (app$lhs1$lastl) $ (app$rhs1$lastr)
        val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (F,F2));
        val thm = Goal.prove (Simplifier.the-context ss) [] [] eq
          (K (simp-tac (Simplifier.inherit-context ss rearr-ss) 1));
        in SOME ((conv RS (thm RS trans)) RS eq-reflection) end;
      in
        if list1 lastl andalso list1 lastr then rearr @{thm append1-eq-conv}
        else if lastl aconv lastr then rearr @{thm append-same-eq}
        else NONE
      end;
  in
    val list-eq-simproc =
      Simplifier.simproc (the-context ()) list-eq [(xs::'a list) = ys] (K list-eq);
  end;

  Addsimprocs [list-eq-simproc];
  >>

```

### 39.1.5 map

**lemma** map-ext: (!x. x : set xs --> f x = g x) ==> map f xs = map g xs  
**by** (induct xs) simp-all

**lemma** map-ident [simp]: map ( $\lambda x. x$ ) = ( $\lambda xs. xs$ )  
**by** (rule ext, induct-tac xs) auto

**lemma** map-append [simp]: map f (xs @ ys) = map f xs @ map f ys  
**by** (induct xs) auto

**lemma** *map-compose*:  $\text{map } (f \circ g) \text{ } xs = \text{map } f \text{ } (\text{map } g \text{ } xs)$   
**by** (*induct xs*) (*auto simp add: o-def*)

**lemma** *rev-map*:  $\text{rev } (\text{map } f \text{ } xs) = \text{map } f \text{ } (\text{rev } xs)$   
**by** (*induct xs*) *auto*

**lemma** *map-eq-conv*[*simp*]:  $(\text{map } f \text{ } xs = \text{map } g \text{ } xs) = (!x : \text{set } xs. f \text{ } x = g \text{ } x)$   
**by** (*induct xs*) *auto*

**lemma** *map-cong* [*fundef-cong, recdef-cong*]:  
 $xs = ys \implies (!x. x : \text{set } ys \implies f \text{ } x = g \text{ } x) \implies \text{map } f \text{ } xs = \text{map } g \text{ } ys$   
— a congruence rule for *map*  
**by** *simp*

**lemma** *map-is-Nil-conv* [*iff*]:  $(\text{map } f \text{ } xs = []) = (xs = [])$   
**by** (*cases xs*) *auto*

**lemma** *Nil-is-map-conv* [*iff*]:  $([] = \text{map } f \text{ } xs) = (xs = [])$   
**by** (*cases xs*) *auto*

**lemma** *map-eq-Cons-conv*:  
 $(\text{map } f \text{ } xs = y \# ys) = (\exists z \text{ } zs. xs = z \# zs \wedge f \text{ } z = y \wedge \text{map } f \text{ } zs = ys)$   
**by** (*cases xs*) *auto*

**lemma** *Cons-eq-map-conv*:  
 $(x \# xs = \text{map } f \text{ } ys) = (\exists z \text{ } zs. ys = z \# zs \wedge x = f \text{ } z \wedge xs = \text{map } f \text{ } zs)$   
**by** (*cases ys*) *auto*

**lemmas** *map-eq-Cons-D* = *map-eq-Cons-conv* [*THEN iffD1*]  
**lemmas** *Cons-eq-map-D* = *Cons-eq-map-conv* [*THEN iffD1*]  
**declare** *map-eq-Cons-D* [*dest!*] *Cons-eq-map-D* [*dest!*]

**lemma** *ex-map-conv*:  
 $(\text{EX } xs. ys = \text{map } f \text{ } xs) = (\text{ALL } y : \text{set } ys. \text{EX } x. y = f \text{ } x)$   
**by**(*induct ys, auto simp add: Cons-eq-map-conv*)

**lemma** *map-eq-imp-length-eq*:  
**assumes**  $\text{map } f \text{ } xs = \text{map } f \text{ } ys$   
**shows**  $\text{length } xs = \text{length } ys$   
**using** *assms* **proof** (*induct ys arbitrary: xs*)  
**case** *Nil* **then show** ?*case* **by** *simp*  
**next**  
**case** (*Cons y ys*) **then obtain** *z zs* **where**  $xs = z \# zs$  **by** *auto*  
**from** *Cons xs* **have**  $\text{map } f \text{ } zs = \text{map } f \text{ } ys$  **by** *simp*  
**moreover with** *Cons* **have**  $\text{length } zs = \text{length } ys$  **by** *blast*  
**with** *xs* **show** ?*case* **by** *simp*  
**qed**

**lemma** *map-inj-on*:

$[[ \text{map } f \text{ } xs = \text{map } f \text{ } ys; \text{inj-on } f \text{ } (\text{set } xs \text{ } Un \text{ } \text{set } ys) \text{ } ]]$

$\implies xs = ys$

**apply**(*frule map-eq-imp-length-eq*)

**apply**(*rotate-tac -1*)

**apply**(*induct rule:list-induct2*)

**apply** *simp*

**apply**(*simp*)

**apply** (*blast intro:sym*)

**done**

**lemma** *inj-on-map-eq-map*:

$\text{inj-on } f \text{ } (\text{set } xs \text{ } Un \text{ } \text{set } ys) \implies (\text{map } f \text{ } xs = \text{map } f \text{ } ys) = (xs = ys)$

**by**(*blast dest:map-inj-on*)

**lemma** *map-injective*:

$\text{map } f \text{ } xs = \text{map } f \text{ } ys \implies \text{inj } f \implies xs = ys$

**by** (*induct ys arbitrary: xs*) (*auto dest!:injD*)

**lemma** *inj-map-eq-map[simp]*:  $\text{inj } f \implies (\text{map } f \text{ } xs = \text{map } f \text{ } ys) = (xs = ys)$

**by**(*blast dest:map-injective*)

**lemma** *inj-mapI*:  $\text{inj } f \implies \text{inj } (\text{map } f)$

**by** (*iprover dest: map-injective injD intro: inj-onI*)

**lemma** *inj-mapD*:  $\text{inj } (\text{map } f) \implies \text{inj } f$

**apply** (*unfold inj-on-def, clarify*)

**apply** (*erule-tac x = [x] in ballE*)

**apply** (*erule-tac x = [y] in ballE, simp, blast*)

**apply** *blast*

**done**

**lemma** *inj-map[iff]*:  $\text{inj } (\text{map } f) = \text{inj } f$

**by** (*blast dest: inj-mapD intro: inj-mapI*)

**lemma** *inj-on-mapI*:  $\text{inj-on } f \text{ } (\bigcup (\text{set } 'A)) \implies \text{inj-on } (\text{map } f) \text{ } A$

**apply**(*rule inj-onI*)

**apply**(*erule map-inj-on*)

**apply**(*blast intro:inj-onI dest:inj-onD*)

**done**

**lemma** *map-idI*:  $(\bigwedge x. x \in \text{set } xs \implies f \text{ } x = x) \implies \text{map } f \text{ } xs = xs$

**by** (*induct xs, auto*)

**lemma** *map-fun-upd [simp]*:  $y \notin \text{set } xs \implies \text{map } (f(y:=v)) \text{ } xs = \text{map } f \text{ } xs$

**by** (*induct xs*) *auto*

**lemma** *map-fst-zip[simp]*:

$\text{length } xs = \text{length } ys \implies \text{map fst } (\text{zip } xs \text{ } ys) = xs$

**by** (*induct rule:list-induct2, simp-all*)

**lemma** *map-snd-zip* [*simp*]:

$length\ xs = length\ ys \implies map\ snd\ (zip\ xs\ ys) = ys$

**by** (*induct rule:list-induct2, simp-all*)

### 39.1.6 *rev*

**lemma** *rev-append* [*simp*]:  $rev\ (xs\ @\ ys) = rev\ ys\ @\ rev\ xs$

**by** (*induct xs*) *auto*

**lemma** *rev-rev-ident* [*simp*]:  $rev\ (rev\ xs) = xs$

**by** (*induct xs*) *auto*

**lemma** *rev-swap*:  $(rev\ xs = ys) = (xs = rev\ ys)$

**by** *auto*

**lemma** *rev-is-Nil-conv* [*iff*]:  $(rev\ xs = []) = (xs = [])$

**by** (*induct xs*) *auto*

**lemma** *Nil-is-rev-conv* [*iff*]:  $([] = rev\ xs) = (xs = [])$

**by** (*induct xs*) *auto*

**lemma** *rev-singleton-conv* [*simp*]:  $(rev\ xs = [x]) = (xs = [x])$

**by** (*cases xs*) *auto*

**lemma** *singleton-rev-conv* [*simp*]:  $([x] = rev\ xs) = (xs = [x])$

**by** (*cases xs*) *auto*

**lemma** *rev-is-rev-conv* [*iff*]:  $(rev\ xs = rev\ ys) = (xs = ys)$

**apply** (*induct xs arbitrary: ys, force*)

**apply** (*case-tac ys, simp, force*)

**done**

**lemma** *inj-on-rev* [*iff*]: *inj-on* *rev A*

**by** (*simp add:inj-on-def*)

**lemma** *rev-induct* [*case-names Nil snoc*]:

$[[]\ P\ []; !!x\ xs.\ P\ xs \implies P\ (xs\ @\ [x])\ [] \implies P\ xs]$

**apply** (*simplesubst rev-rev-ident[symmetric]*)

**apply** (*rule-tac list = rev xs in list.induct, simp-all*)

**done**

**lemma** *rev-exhaust* [*case-names Nil snoc*]:

$(xs = [] \implies P) \implies (!!ys\ y.\ xs = ys\ @\ [y] \implies P) \implies P$

**by** (*induct xs rule: rev-induct*) *auto*

**lemmas** *rev-cases = rev-exhaust*



**lemma** *rev-eq-Cons-iff* [iff]:  $(\text{rev } xs = y \# ys) = (xs = \text{rev } ys @ [y])$   
**by** (rule *rev-cases* [of *xs*]) *auto*

### 39.1.7 set

**lemma** *finite-set* [iff]: *finite* (set *xs*)  
**by** (induct *xs*) *auto*

**lemma** *set-append* [simp]:  $\text{set } (xs @ ys) = (\text{set } xs \cup \text{set } ys)$   
**by** (induct *xs*) *auto*

**lemma** *hd-in-set* [simp]:  $xs \neq [] \implies \text{hd } xs : \text{set } xs$   
**by** (cases *xs*) *auto*

**lemma** *set-subset-Cons*:  $\text{set } xs \subseteq \text{set } (x \# xs)$   
**by** *auto*

**lemma** *set-ConsD*:  $y \in \text{set } (x \# xs) \implies y = x \vee y \in \text{set } xs$   
**by** *auto*

**lemma** *set-empty* [iff]:  $(\text{set } xs = \{\}) = (xs = [])$   
**by** (induct *xs*) *auto*

**lemma** *set-empty2* [iff]:  $(\{\} = \text{set } xs) = (xs = [])$   
**by** (induct *xs*) *auto*

**lemma** *set-rev* [simp]:  $\text{set } (\text{rev } xs) = \text{set } xs$   
**by** (induct *xs*) *auto*

**lemma** *set-map* [simp]:  $\text{set } (\text{map } f \text{ } xs) = f'(\text{set } xs)$   
**by** (induct *xs*) *auto*

**lemma** *set-filter* [simp]:  $\text{set } (\text{filter } P \text{ } xs) = \{x. x : \text{set } xs \wedge P \text{ } x\}$   
**by** (induct *xs*) *auto*

**lemma** *set-upt* [simp]:  $\text{set}[i..<j] = \{k. i \leq k \wedge k < j\}$   
**apply** (induct *j*, *simp-all*)  
**apply** (erule *ssubst*, *auto*)  
**done**

**lemma** *split-list*:  $x : \text{set } xs \implies \exists ys \text{ } zs. xs = ys @ x \# zs$   
**proof** (induct *xs*)  
  **case** *Nil* **thus** ?*case* **by** *simp*  
**next**  
  **case** *Cons* **thus** ?*case* **by** (auto intro: *Cons-eq-appendI*)  
**qed**

**lemma** *in-set-conv-decomp*:  $x \in \text{set } xs \longleftrightarrow (\exists ys \text{ } zs. xs = ys @ x \# zs)$

```

by (auto elim: split-list)

lemma split-list-first:  $x : \text{set } xs \implies \exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } ys$ 
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons a xs)
  show ?case
  proof cases
    assume  $x = a$  thus ?case using Cons by fastsimp
  next
    assume  $x \neq a$  thus ?case using Cons by (fastsimp intro!: Cons-eq-appendI)
  qed
qed

lemma in-set-conv-decomp-first:
   $(x : \text{set } xs) = (\exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } ys)$ 
  by (auto dest!: split-list-first)

lemma split-list-last:  $x : \text{set } xs \implies \exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } zs$ 
proof (induct xs rule: rev-induct)
  case Nil thus ?case by simp
next
  case (snoc a xs)
  show ?case
  proof cases
    assume  $x = a$  thus ?case using snoc by simp (metis ex-in-conv set-empty2)
  next
    assume  $x \neq a$  thus ?case using snoc by fastsimp
  qed
qed

lemma in-set-conv-decomp-last:
   $(x : \text{set } xs) = (\exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } zs)$ 
  by (auto dest!: split-list-last)

lemma split-list-prop:  $\exists x \in \text{set } xs. P\ x \implies \exists ys\ x\ zs. xs = ys @ x \# zs \ \&\ P\ x$ 
proof (induct xs)
  case Nil thus ?case by simp
next
  case Cons thus ?case
    by (simp add: Bex-def) (metis append-Cons append.simps(1))
qed

lemma split-list-propE:
  assumes  $\exists x \in \text{set } xs. P\ x$ 
  obtains  $ys\ x\ zs$  where  $xs = ys @ x \# zs$  and  $P\ x$ 
  using split-list-prop [OF assms] by blast

```

```

lemma split-list-first-prop:
   $\exists x \in \text{set } xs. P\ x \implies$ 
   $\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P\ x \wedge (\forall y \in \text{set } ys. \neg P\ y)$ 
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons x xs)
  show ?case
  proof cases
    assume  $P\ x$ 
    thus ?thesis by simp
    (metis Un-upper1 contra-subsetD in-set-conv-decomp-first self-append-conv2
set-append)
  next
    assume  $\neg P\ x$ 
    hence  $\exists x \in \text{set } xs. P\ x$  using Cons(2) by simp
    thus ?thesis using  $\langle \neg P\ x \rangle$  Cons(1) by (metis append-Cons set-ConsD)
  qed
qed

```

```

lemma split-list-first-propE:
  assumes  $\exists x \in \text{set } xs. P\ x$ 
  obtains  $ys\ x\ zs$  where  $xs = ys @ x \# zs$  and  $P\ x$  and  $\forall y \in \text{set } ys. \neg P\ y$ 
using split-list-first-prop [OF assms] by blast

```

```

lemma split-list-first-prop-iff:
   $(\exists x \in \text{set } xs. P\ x) \longleftrightarrow$ 
   $(\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P\ x \wedge (\forall y \in \text{set } ys. \neg P\ y))$ 
by (rule, erule split-list-first-prop) auto

```

```

lemma split-list-last-prop:
   $\exists x \in \text{set } xs. P\ x \implies$ 
   $\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P\ x \wedge (\forall z \in \text{set } zs. \neg P\ z)$ 
proof(induct xs rule:rev-induct)
  case Nil thus ?case by simp
next
  case (snoc x xs)
  show ?case
  proof cases
    assume  $P\ x$  thus ?thesis by (metis emptyE set-empty)
  next
    assume  $\neg P\ x$ 
    hence  $\exists x \in \text{set } xs. P\ x$  using snoc(2) by simp
    thus ?thesis using  $\langle \neg P\ x \rangle$  snoc(1) by fastsimp
  qed
qed

```

```

lemma split-list-last-propE:
  assumes  $\exists x \in \text{set } xs. P\ x$ 

```

**obtains**  $ys\ x\ zs$  **where**  $xs = ys @ x \# zs$  **and**  $P\ x$  **and**  $\forall z \in set\ zs. \neg P\ z$   
**using** *split-list-last-prop* [*OF assms*] **by** *blast*

**lemma** *split-list-last-prop-iff*:

$(\exists x \in set\ xs. P\ x) \longleftrightarrow$

$(\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P\ x \wedge (\forall z \in set\ zs. \neg P\ z))$

**by** (*metis split-list-last-prop* [**where**  $P=P$ ] *in-set-conv-decomp*)

**lemma** *finite-list*:  $finite\ A \implies \exists xs. set\ xs = A$

**by** (*erule finite-induct*)

(*auto simp add: set.simps(2) [symmetric] simp del: set.simps(2)*)

**lemma** *card-length*:  $card\ (set\ xs) \leq length\ xs$

**by** (*induct xs*) (*auto simp add: card-insert-if*)

**lemma** *set-minus-filter-out*:

$set\ xs - \{y\} = set\ (filter\ (\lambda x. \neg (x = y))\ xs)$

**by** (*induct xs*) *auto*

### 39.1.8 *filter*

**lemma** *filter-append* [*simp*]:  $filter\ P\ (xs @ ys) = filter\ P\ xs @ filter\ P\ ys$

**by** (*induct xs*) *auto*

**lemma** *rev-filter*:  $rev\ (filter\ P\ xs) = filter\ P\ (rev\ xs)$

**by** (*induct xs*) *simp-all*

**lemma** *filter-filter* [*simp*]:  $filter\ P\ (filter\ Q\ xs) = filter\ (\lambda x. Q\ x \wedge P\ x)\ xs$

**by** (*induct xs*) *auto*

**lemma** *length-filter-le* [*simp*]:  $length\ (filter\ P\ xs) \leq length\ xs$

**by** (*induct xs*) (*auto simp add: le-SucI*)

**lemma** *sum-length-filter-compl*:

$length\ (filter\ P\ xs) + length\ (filter\ (\%x. \neg P\ x)\ xs) = length\ xs$

**by** (*induct xs*) *simp-all*

**lemma** *filter-True* [*simp*]:  $\forall x \in set\ xs. P\ x \implies filter\ P\ xs = xs$

**by** (*induct xs*) *auto*

**lemma** *filter-False* [*simp*]:  $\forall x \in set\ xs. \neg P\ x \implies filter\ P\ xs = []$

**by** (*induct xs*) *auto*

**lemma** *filter-empty-conv*:  $(filter\ P\ xs = []) = (\forall x \in set\ xs. \neg P\ x)$

**by** (*induct xs*) *simp-all*

**lemma** *filter-id-conv*:  $(filter\ P\ xs = xs) = (\forall x \in set\ xs. P\ x)$

**apply** (*induct xs*)

**apply** *auto*

```

apply(cut-tac  $P=P$  and  $xs=xs$  in length-filter-le)
apply simp
done

```

```

lemma filter-map:
  filter  $P$  (map  $f$   $xs$ ) = map  $f$  (filter ( $P \circ f$ )  $xs$ )
by (induct  $xs$ ) simp-all

```

```

lemma length-filter-map[simp]:
  length (filter  $P$  (map  $f$   $xs$ )) = length (filter ( $P \circ f$ )  $xs$ )
by (simp add:filter-map)

```

```

lemma filter-is-subset [simp]: set (filter  $P$   $xs$ )  $\leq$  set  $xs$ 
by auto

```

```

lemma length-filter-less:
   $\llbracket x : \text{set } xs; \sim P\ x \rrbracket \implies \text{length}(\text{filter } P\ xs) < \text{length } xs$ 
proof (induct  $xs$ )
  case Nil thus ?case by simp
next
  case (Cons  $x$   $xs$ ) thus ?case
    apply (auto split:split-if-asm)
    using length-filter-le[of  $P$   $xs$ ] apply arith
  done
qed

```

```

lemma length-filter-conv-card:
  length (filter  $p$   $xs$ ) = card { $i$ .  $i < \text{length } xs \ \& \ p(xs!i)$ }
proof (induct  $xs$ )
  case Nil thus ?case by simp
next
  case (Cons  $x$   $xs$ )
    let ? $S$  = { $i$ .  $i < \text{length } xs \ \& \ p(xs!i)$ }
    have fin: finite ? $S$  by (fast intro: bounded-nat-set-is-finite)
    show ?case (is ? $l$  = card ? $S'$ )
    proof (cases)
      assume  $p\ x$ 
      hence eq: ? $S'$  = insert 0 (Suc ‘ ? $S$ )
      by (auto simp: image-def split:nat.split dest:gr0-implies-Suc)
      have length (filter  $p$  ( $x \# xs$ )) = Suc (card ? $S$ )
      using Cons ⟨ $p\ x$ ⟩ by simp
      also have ... = Suc (card (Suc ‘ ? $S$ )) using fin
      by (simp add: card-image inj-Suc)
      also have ... = card ? $S'$  using eq fin
      by (simp add:card-insert-if) (simp add:image-def)
      finally show ?thesis .
    next
      assume  $\neg p\ x$ 
      hence eq: ? $S'$  = Suc ‘ ? $S$ 

```

```

    by(auto simp add: image-def split:nat.split elim:lessE)
  have length (filter p (x # xs)) = card ?S
    using Cons (¬ p x) by simp
  also have ... = card (Suc ‘ ?S) using fin
    by (simp add: card-image inj-Suc)
  also have ... = card ?S' using eq fin
    by (simp add: card-insert-if)
  finally show ?thesis .
qed
qed

```

**lemma** *Cons-eq-filterD*:

```

x # xs = filter P ys ⟹
  ∃ us vs. ys = us @ x # vs ∧ (∀ u ∈ set us. ¬ P u) ∧ P x ∧ xs = filter P vs
  (is - ⟹ ∃ us vs. ?P ys us vs)
proof(induct ys)
  case Nil thus ?case by simp
next
  case (Cons y ys)
  show ?case (is ∃ x. ?Q x)
  proof cases
    assume Py: P y
    show ?thesis
    proof cases
      assume x = y
      with Py Cons.prem1 have ?Q [] by simp
      then show ?thesis ..
    next
      assume x ≠ y
      with Py Cons.prem1 show ?thesis by simp
    qed
  next
    assume ¬ P y
    with Cons obtain us vs where ?P (y # ys) (y # us) vs by fastsimp
    then have ?Q (y # us) by simp
    then show ?thesis ..
  qed
qed

```

**lemma** *filter-eq-ConsD*:

```

filter P ys = x # xs ⟹
  ∃ us vs. ys = us @ x # vs ∧ (∀ u ∈ set us. ¬ P u) ∧ P x ∧ xs = filter P vs
by(rule Cons-eq-filterD) simp

```

**lemma** *filter-eq-Cons-iff*:

```

(filter P ys = x # xs) =
  (∃ us vs. ys = us @ x # vs ∧ (∀ u ∈ set us. ¬ P u) ∧ P x ∧ xs = filter P vs)
by(auto dest:filter-eq-ConsD)

```

**lemma** *Cons-eq-filter-iff*:  
 $(x \# xs = \text{filter } P \text{ } ys) =$   
 $(\exists us \text{ } vs. \text{ } ys = us @ x \# vs \wedge (\forall u \in \text{set } us. \neg P u) \wedge P x \wedge xs = \text{filter } P \text{ } vs)$   
**by** (*auto dest:Cons-eq-filterD*)

**lemma** *filter-cong*[*fundef-cong*, *recdef-cong*]:  
 $xs = ys \implies (\bigwedge x. x \in \text{set } ys \implies P x = Q x) \implies \text{filter } P \text{ } xs = \text{filter } Q \text{ } ys$   
**apply** *simp*  
**apply** (*erule thin-rl*)  
**by** (*induct ys*) *simp-all*

### 39.1.9 List partitioning

**primrec** *partition* ::  $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \times 'a \text{ list}$  **where**  
 $\text{partition } P \text{ } [] = ([], [])$   
 $| \text{partition } P \text{ } (x \# xs) =$   
 $(\text{let } (yes, no) = \text{partition } P \text{ } xs$   
 $\text{in if } P x \text{ then } (x \# yes, no) \text{ else } (yes, x \# no))$

**lemma** *partition-filter1*:  
 $\text{fst } (\text{partition } P \text{ } xs) = \text{filter } P \text{ } xs$   
**by** (*induct xs*) (*auto simp add: Let-def split-def*)

**lemma** *partition-filter2*:  
 $\text{snd } (\text{partition } P \text{ } xs) = \text{filter } (\text{Not } o P) \text{ } xs$   
**by** (*induct xs*) (*auto simp add: Let-def split-def*)

**lemma** *partition-P*:  
**assumes**  $\text{partition } P \text{ } xs = (yes, no)$   
**shows**  $(\forall p \in \text{set } yes. P p) \wedge (\forall p \in \text{set } no. \neg P p)$   
**proof** –  
**from** *assms* **have**  $yes = \text{fst } (\text{partition } P \text{ } xs)$  **and**  $no = \text{snd } (\text{partition } P \text{ } xs)$   
**by** *simp-all*  
**then show** *?thesis* **by** (*simp-all add: partition-filter1 partition-filter2*)  
**qed**

**lemma** *partition-set*:  
**assumes**  $\text{partition } P \text{ } xs = (yes, no)$   
**shows**  $\text{set } yes \cup \text{set } no = \text{set } xs$   
**proof** –  
**from** *assms* **have**  $yes = \text{fst } (\text{partition } P \text{ } xs)$  **and**  $no = \text{snd } (\text{partition } P \text{ } xs)$   
**by** *simp-all*  
**then show** *?thesis* **by** (*auto simp add: partition-filter1 partition-filter2*)  
**qed**

### 39.1.10 concat

**lemma** *concat-append* [*simp*]:  $\text{concat } (xs @ ys) = \text{concat } xs @ \text{concat } ys$   
**by** (*induct xs*) *auto*

**lemma** *concat-eq-Nil-conv* [simp]:  $(\text{concat } xss = []) = (\forall xs \in \text{set } xss. xs = [])$   
**by** (induct xss) auto

**lemma** *Nil-eq-concat-conv* [simp]:  $([] = \text{concat } xss) = (\forall xs \in \text{set } xss. xs = [])$   
**by** (induct xss) auto

**lemma** *set-concat* [simp]:  $\text{set } (\text{concat } xs) = (\bigcup x:\text{set } xs. \text{set } x)$   
**by** (induct xs) auto

**lemma** *concat-map-singleton*[simp]:  $\text{concat}(\text{map } (\%x. [f x]) xs) = \text{map } f xs$   
**by** (induct xs) auto

**lemma** *map-concat*:  $\text{map } f (\text{concat } xs) = \text{concat } (\text{map } (\text{map } f) xs)$   
**by** (induct xs) auto

**lemma** *filter-concat*:  $\text{filter } p (\text{concat } xs) = \text{concat } (\text{map } (\text{filter } p) xs)$   
**by** (induct xs) auto

**lemma** *rev-concat*:  $\text{rev } (\text{concat } xs) = \text{concat } (\text{map } \text{rev } (\text{rev } xs))$   
**by** (induct xs) auto

### 39.1.11 *nth*

**lemma** *nth-Cons-0* [simp, code]:  $(x \# xs)!0 = x$   
**by** auto

**lemma** *nth-Cons-Suc* [simp, code]:  $(x \# xs)!(\text{Suc } n) = xs!n$   
**by** auto

**declare** *nth.simps* [simp del]

**lemma** *nth-append*:  
 $(xs @ ys)!n = (\text{if } n < \text{length } xs \text{ then } xs!n \text{ else } ys!(n - \text{length } xs))$   
**apply** (induct xs arbitrary: n, simp)  
**apply** (case-tac n, auto)  
**done**

**lemma** *nth-append-length* [simp]:  $(xs @ x \# ys) ! \text{length } xs = x$   
**by** (induct xs) auto

**lemma** *nth-append-length-plus*[simp]:  $(xs @ ys) ! (\text{length } xs + n) = ys ! n$   
**by** (induct xs) auto

**lemma** *nth-map* [simp]:  $n < \text{length } xs \implies (\text{map } f xs)!n = f(xs!n)$   
**apply** (induct xs arbitrary: n, simp)  
**apply** (case-tac n, auto)  
**done**

**lemma** *hd-conv-nth*:  $xs \neq [] \implies \text{hd } xs = xs!0$



**by**(*cases xs*) *simp-all*

**lemma** *list-eq-iff-nth-eq*:

(*xs* = *ys*) = (*length xs* = *length ys*  $\wedge$  (*ALL* *i* < *length xs*. *xs*!*i* = *ys*!*i*))  
**apply**(*induct xs arbitrary: ys*)  
**apply** *force*  
**apply**(*case-tac ys*)  
**apply** *simp*  
**apply**(*simp add:nth-Cons split:nat.split*)**apply** *blast*  
**done**

**lemma** *set-conv-nth*: *set xs* = {*xs*!*i* | *i*. *i* < *length xs*}

**apply** (*induct xs, simp, simp*)  
**apply** *safe*  
**apply** (*metis nat-case-0 nth.simps zero-less-Suc*)  
**apply** (*metis less-Suc-eq-0-disj nth-Cons-Suc*)  
**apply** (*case-tac i, simp*)  
**apply** (*metis diff-Suc-Suc nat-case-Suc nth.simps zero-less-diff*)  
**done**

**lemma** *in-set-conv-nth*: (*x*  $\in$  *set xs*) = ( $\exists$  *i* < *length xs*. *xs*!*i* = *x*)

**by**(*auto simp:set-conv-nth*)

**lemma** *list-ball-nth*: [*n* < *length xs*; !*x* : *set xs*. *P x*]  $\implies$  *P*(*xs*!*n*)

**by** (*auto simp add: set-conv-nth*)

**lemma** *nth-mem* [*simp*]: *n* < *length xs*  $\implies$  *xs*!*n* : *set xs*

**by** (*auto simp add: set-conv-nth*)

**lemma** *all-nth-imp-all-set*:

[*!i* < *length xs*. *P*(*xs*!*i*); *x* : *set xs*]  $\implies$  *P x*

**by** (*auto simp add: set-conv-nth*)

**lemma** *all-set-conv-all-nth*:

( $\forall x \in \text{set } xs. P x$ ) = ( $\forall i. i < \text{length } xs \longrightarrow P (xs ! i)$ )

**by** (*auto simp add: set-conv-nth*)

**lemma** *rev-nth*:

*n* < *size xs*  $\implies$  *rev xs* ! *n* = *xs* ! (*length xs* - *Suc n*)

**proof** (*induct xs arbitrary: n*)

**case** *Nil* **thus** ?*case* **by** *simp*

**next**

**case** (*Cons x xs*)

**hence** *n*: *n* < *Suc* (*length xs*) **by** *simp*

**moreover**

{ **assume** *n* < *length xs*

**with** *n* **obtain** *n'* **where** *length xs* - *n* = *Suc n'*

**by** (*cases length xs - n, auto*)

```

moreover
then have  $\text{length } xs - \text{Suc } n = n'$  by simp
ultimately
have  $xs ! (\text{length } xs - \text{Suc } n) = (x \# xs) ! (\text{length } xs - n)$  by simp
}
ultimately
show ?case by (clarsimp simp add: Cons nth-append)
qed

```

### 39.1.12 *list-update*

**lemma** *length-list-update* [*simp*]:  $\text{length}(xs[i:=x]) = \text{length } xs$   
**by** (*induct xs arbitrary: i*) (*auto split: nat.split*)

**lemma** *nth-list-update*:  
 $i < \text{length } xs \implies (xs[i:=x])!j = (\text{if } i = j \text{ then } x \text{ else } xs!j)$   
**by** (*induct xs arbitrary: i j*) (*auto simp add: nth-Cons split: nat.split*)

**lemma** *nth-list-update-eq* [*simp*]:  $i < \text{length } xs \implies (xs[i:=x])!i = x$   
**by** (*simp add: nth-list-update*)

**lemma** *nth-list-update-neq* [*simp*]:  $i \neq j \implies xs[i:=x]!j = xs!j$   
**by** (*induct xs arbitrary: i j*) (*auto simp add: nth-Cons split: nat.split*)

**lemma** *list-update-id* [*simp*]:  $xs[i := xs!i] = xs$   
**by** (*induct xs arbitrary: i*) (*simp-all split: nat.splits*)

**lemma** *list-update-beyond* [*simp*]:  $\text{length } xs \leq i \implies xs[i:=x] = xs$   
**apply** (*induct xs arbitrary: i*)  
**apply** *simp*  
**apply** (*case-tac i*)  
**apply** *simp-all*  
**done**

**lemma** *list-update-same-conv*:  
 $i < \text{length } xs \implies (xs[i := x] = xs) = (xs!i = x)$   
**by** (*induct xs arbitrary: i*) (*auto split: nat.split*)

**lemma** *list-update-append1*:  
 $i < \text{size } xs \implies (xs @ ys)[i:=x] = xs[i:=x] @ ys$   
**apply** (*induct xs arbitrary: i, simp*)  
**apply** (*simp split: nat.split*)  
**done**

**lemma** *list-update-append*:  
 $(xs @ ys)[n := x] =$   
 $(\text{if } n < \text{length } xs \text{ then } xs[n := x] @ ys \text{ else } xs @ (ys[n - \text{length } xs := x]))$   
**by** (*induct xs arbitrary: n*) (*auto split: nat.splits*)

**lemma** *list-update-length* [simp]:  
 $(xs @ x \# ys)[length\ xs := y] = (xs @ y \# ys)$   
**by** (induct xs, auto)

**lemma** *update-zip*:  
 $length\ xs = length\ ys ==>$   
 $(zip\ xs\ ys)[i:=xy] = zip\ (xs[i:=fst\ xy])\ (ys[i:=snd\ xy])$   
**by** (induct ys arbitrary: i xy xs) (auto, case-tac xs, auto split: nat.split)

**lemma** *set-update-subset-insert*:  $set(xs[i:=x]) \leq insert\ x\ (set\ xs)$   
**by** (induct xs arbitrary: i) (auto split: nat.split)

**lemma** *set-update-subsetI*:  $[| set\ xs \leq A; x:A |] ==> set(xs[i := x]) \leq A$   
**by** (blast dest!: set-update-subset-insert [THEN subsetD])

**lemma** *set-update-memI*:  $n < length\ xs \implies x \in set\ (xs[n := x])$   
**by** (induct xs arbitrary: n) (auto split: nat.splits)

**lemma** *list-update-overwrite*:  
 $xs\ [i := x, i := y] = xs\ [i := y]$   
**apply** (induct xs arbitrary: i)  
**apply** simp  
**apply** (case-tac i)  
**apply** simp-all  
**done**

**lemma** *list-update-swap*:  
 $i \neq i' \implies xs\ [i := x, i' := x'] = xs\ [i' := x', i := x]$   
**apply** (induct xs arbitrary: i i')  
**apply** simp  
**apply** (case-tac i, case-tac i')  
**apply** auto  
**apply** (case-tac i')  
**apply** auto  
**done**

**lemma** *list-update-code* [code]:  
 $[] [i := y] = []$   
 $(x \# xs)[0 := y] = y \# xs$   
 $(x \# xs)[Suc\ i := y] = x \# xs[i := y]$   
**by** simp-all

### 39.1.13 last and butlast

**lemma** *last-snoc* [simp]:  $last\ (xs @ [x]) = x$   
**by** (induct xs) auto

**lemma** *butlast-snoc* [simp]:  $butlast\ (xs @ [x]) = xs$   
**by** (induct xs) auto

**lemma** *last-ConsL*:  $xs = [] \implies \text{last}(x \# xs) = x$   
**by** (*simp add: last.simps*)

**lemma** *last-ConsR*:  $xs \neq [] \implies \text{last}(x \# xs) = \text{last } xs$   
**by** (*simp add: last.simps*)

**lemma** *last-append*:  $\text{last}(xs @ ys) = (\text{if } ys = [] \text{ then } \text{last } xs \text{ else } \text{last } ys)$   
**by** (*induct xs*) (*auto*)

**lemma** *last-appendL[simp]*:  $ys = [] \implies \text{last}(xs @ ys) = \text{last } xs$   
**by** (*simp add: last-append*)

**lemma** *last-appendR[simp]*:  $ys \neq [] \implies \text{last}(xs @ ys) = \text{last } ys$   
**by** (*simp add: last-append*)

**lemma** *hd-rev*:  $xs \neq [] \implies \text{hd}(\text{rev } xs) = \text{last } xs$   
**by** (*rule rev-exhaust[of xs]*) *simp-all*

**lemma** *last-rev*:  $xs \neq [] \implies \text{last}(\text{rev } xs) = \text{hd } xs$   
**by** (*cases xs*) *simp-all*

**lemma** *last-in-set[simp]*:  $as \neq [] \implies \text{last } as \in \text{set } as$   
**by** (*induct as*) *auto*

**lemma** *length-butlast [simp]*:  $\text{length } (\text{butlast } xs) = \text{length } xs - 1$   
**by** (*induct xs rule: rev-induct*) *auto*

**lemma** *butlast-append*:  
 $\text{butlast } (xs @ ys) = (\text{if } ys = [] \text{ then } \text{butlast } xs \text{ else } xs @ \text{butlast } ys)$   
**by** (*induct xs arbitrary: ys*) *auto*

**lemma** *append-butlast-last-id [simp]*:  
 $xs \neq [] \implies \text{butlast } xs @ [\text{last } xs] = xs$   
**by** (*induct xs*) *auto*

**lemma** *in-set-butlastD*:  $x : \text{set } (\text{butlast } xs) \implies x : \text{set } xs$   
**by** (*induct xs*) (*auto split: split-if-asm*)

**lemma** *in-set-butlast-appendI*:  
 $x : \text{set } (\text{butlast } xs) \mid x : \text{set } (\text{butlast } ys) \implies x : \text{set } (\text{butlast } (xs @ ys))$   
**by** (*auto dest: in-set-butlastD simp add: butlast-append*)

**lemma** *last-drop[simp]*:  $n < \text{length } xs \implies \text{last } (\text{drop } n \text{ } xs) = \text{last } xs$   
**apply** (*induct xs arbitrary: n*)  
**apply** *simp*  
**apply** (*auto split: nat.split*)  
**done**

**lemma** *last-conv-nth*:  $xs \neq [] \implies \text{last } xs = xs!(\text{length } xs - 1)$   
**by** (*induct xs*) (*auto simp: neq-Nil-conv*)

**lemma** *butlast-conv-take*:  $\text{butlast } xs = \text{take } (\text{length } xs - 1) \ xs$   
**by** (*induct xs, simp, case-tac xs, simp-all*)

### 39.1.14 take and drop

**lemma** *take-0* [*simp*]:  $\text{take } 0 \ xs = []$   
**by** (*induct xs*) *auto*

**lemma** *drop-0* [*simp*]:  $\text{drop } 0 \ xs = xs$   
**by** (*induct xs*) *auto*

**lemma** *take-Suc-Cons* [*simp*]:  $\text{take } (\text{Suc } n) \ (x \# xs) = x \# \text{take } n \ xs$   
**by** *simp*

**lemma** *drop-Suc-Cons* [*simp*]:  $\text{drop } (\text{Suc } n) \ (x \# xs) = \text{drop } n \ xs$   
**by** *simp*

**declare** *take-Cons* [*simp del*] **and** *drop-Cons* [*simp del*]

**lemma** *take-1-Cons* [*simp*]:  $\text{take } 1 \ (x \# xs) = [x]$   
**unfolding** *One-nat-def* **by** *simp*

**lemma** *drop-1-Cons* [*simp*]:  $\text{drop } 1 \ (x \# xs) = xs$   
**unfolding** *One-nat-def* **by** *simp*

**lemma** *take-Suc*:  $xs \sim [] \implies \text{take } (\text{Suc } n) \ xs = \text{hd } xs \# \text{take } n \ (\text{tl } xs)$   
**by** (*clarsimp simp add: neq-Nil-conv*)

**lemma** *drop-Suc*:  $\text{drop } (\text{Suc } n) \ xs = \text{drop } n \ (\text{tl } xs)$   
**by** (*cases xs, simp-all*)

**lemma** *take-tl*:  $\text{take } n \ (\text{tl } xs) = \text{tl } (\text{take } (\text{Suc } n) \ xs)$   
**by** (*induct xs arbitrary: n*) *simp-all*

**lemma** *drop-tl*:  $\text{drop } n \ (\text{tl } xs) = \text{tl } (\text{drop } n \ xs)$   
**by** (*induct xs arbitrary: n, simp-all add: drop-Cons drop-Suc split: nat.split*)

**lemma** *tl-take*:  $\text{tl } (\text{take } n \ xs) = \text{take } (n - 1) \ (\text{tl } xs)$   
**by** (*cases n, simp, cases xs, auto*)

**lemma** *tl-drop*:  $\text{tl } (\text{drop } n \ xs) = \text{drop } n \ (\text{tl } xs)$   
**by** (*simp only: drop-tl*)

**lemma** *nth-via-drop*:  $\text{drop } n \ xs = y \# ys \implies xs!n = y$   
**apply** (*induct xs arbitrary: n, simp*)  
**apply** (*simp add: drop-Cons nth-Cons split: nat.splits*)

done

**lemma** *take-Suc-conv-app-nth*:

$i < \text{length } xs \implies \text{take } (\text{Suc } i) \text{ } xs = \text{take } i \text{ } xs @ [xs!i]$

**apply** (*induct xs arbitrary: i, simp*)

**apply** (*case-tac i, auto*)

done

**lemma** *drop-Suc-conv-tl*:

$i < \text{length } xs \implies (xs!i) \# (\text{drop } (\text{Suc } i) \text{ } xs) = \text{drop } i \text{ } xs$

**apply** (*induct xs arbitrary: i, simp*)

**apply** (*case-tac i, auto*)

done

**lemma** *length-take [simp]*:  $\text{length } (\text{take } n \text{ } xs) = \min (\text{length } xs) \text{ } n$

**by** (*induct n arbitrary: xs*) (*auto, case-tac xs, auto*)

**lemma** *length-drop [simp]*:  $\text{length } (\text{drop } n \text{ } xs) = (\text{length } xs - n)$

**by** (*induct n arbitrary: xs*) (*auto, case-tac xs, auto*)

**lemma** *take-all [simp]*:  $\text{length } xs \leq n \implies \text{take } n \text{ } xs = xs$

**by** (*induct n arbitrary: xs*) (*auto, case-tac xs, auto*)

**lemma** *drop-all [simp]*:  $\text{length } xs \leq n \implies \text{drop } n \text{ } xs = []$

**by** (*induct n arbitrary: xs*) (*auto, case-tac xs, auto*)

**lemma** *take-append [simp]*:

$\text{take } n \text{ } (xs @ ys) = (\text{take } n \text{ } xs @ \text{take } (n - \text{length } xs) \text{ } ys)$

**by** (*induct n arbitrary: xs*) (*auto, case-tac xs, auto*)

**lemma** *drop-append [simp]*:

$\text{drop } n \text{ } (xs @ ys) = \text{drop } n \text{ } xs @ \text{drop } (n - \text{length } xs) \text{ } ys$

**by** (*induct n arbitrary: xs*) (*auto, case-tac xs, auto*)

**lemma** *take-take [simp]*:  $\text{take } n \text{ } (\text{take } m \text{ } xs) = \text{take } (\min n \text{ } m) \text{ } xs$

**apply** (*induct m arbitrary: xs n, auto*)

**apply** (*case-tac xs, auto*)

**apply** (*case-tac n, auto*)

done

**lemma** *drop-drop [simp]*:  $\text{drop } n \text{ } (\text{drop } m \text{ } xs) = \text{drop } (n + m) \text{ } xs$

**apply** (*induct m arbitrary: xs, auto*)

**apply** (*case-tac xs, auto*)

done

**lemma** *take-drop*:  $\text{take } n \text{ } (\text{drop } m \text{ } xs) = \text{drop } m \text{ } (\text{take } (n + m) \text{ } xs)$

**apply** (*induct m arbitrary: xs n, auto*)

**apply** (*case-tac xs, auto*)

done

```

lemma drop-take: drop n (take m xs) = take (m-n) (drop n xs)
apply (induct xs arbitrary: m n)
  apply simp
apply (simp add: take-Cons drop-Cons split:nat.split)
done

```

```

lemma append-take-drop-id [simp]: take n xs @ drop n xs = xs
apply (induct n arbitrary: xs, auto)
apply (case-tac xs, auto)
done

```

```

lemma take-eq-Nil[simp]: (take n xs = []) = (n = 0 ∨ xs = [])
apply (induct xs arbitrary: n)
  apply simp
apply (simp add: take-Cons split:nat.split)
done

```

```

lemma drop-eq-Nil[simp]: (drop n xs = []) = (length xs ≤ n)
apply (induct xs arbitrary: n)
apply simp
apply (simp add: drop-Cons split:nat.split)
done

```

```

lemma take-map: take n (map f xs) = map f (take n xs)
apply (induct n arbitrary: xs, auto)
apply (case-tac xs, auto)
done

```

```

lemma drop-map: drop n (map f xs) = map f (drop n xs)
apply (induct n arbitrary: xs, auto)
apply (case-tac xs, auto)
done

```

```

lemma rev-take: rev (take i xs) = drop (length xs - i) (rev xs)
apply (induct xs arbitrary: i, auto)
apply (case-tac i, auto)
done

```

```

lemma rev-drop: rev (drop i xs) = take (length xs - i) (rev xs)
apply (induct xs arbitrary: i, auto)
apply (case-tac i, auto)
done

```

```

lemma nth-take [simp]: i < n ==> (take n xs)!i = xs!i
apply (induct xs arbitrary: i n, auto)
apply (case-tac n, blast)
apply (case-tac i, auto)
done

```

```

lemma nth-drop [simp]:
   $n + i \leq \text{length } xs \implies (\text{drop } n \text{ } xs)!i = xs!(n + i)$ 
apply (induct n arbitrary: xs i, auto)
apply (case-tac xs, auto)
done

lemma butlast-take:
   $n \leq \text{length } xs \implies \text{butlast } (\text{take } n \text{ } xs) = \text{take } (n - 1) \text{ } xs$ 
by (simp add: butlast-conv-take min-max.inf-absorb1 min-max.inf-absorb2)

lemma butlast-drop:  $\text{butlast } (\text{drop } n \text{ } xs) = \text{drop } n \text{ } (\text{butlast } xs)$ 
by (simp add: butlast-conv-take drop-take add-ac)

lemma take-butlast:  $n < \text{length } xs \implies \text{take } n \text{ } (\text{butlast } xs) = \text{take } n \text{ } xs$ 
by (simp add: butlast-conv-take min-max.inf-absorb1)

lemma drop-butlast:  $\text{drop } n \text{ } (\text{butlast } xs) = \text{butlast } (\text{drop } n \text{ } xs)$ 
by (simp add: butlast-conv-take drop-take add-ac)

lemma hd-drop-conv-nth:  $\llbracket xs \neq []; n < \text{length } xs \rrbracket \implies \text{hd}(\text{drop } n \text{ } xs) = xs!n$ 
by(simp add: hd-conv-nth)

lemma set-take-subset:  $\text{set}(\text{take } n \text{ } xs) \subseteq \text{set } xs$ 
by(induct xs arbitrary: n)(auto simp:take-Cons split:nat.split)

lemma set-drop-subset:  $\text{set}(\text{drop } n \text{ } xs) \subseteq \text{set } xs$ 
by(induct xs arbitrary: n)(auto simp:drop-Cons split:nat.split)

lemma in-set-takeD:  $x : \text{set}(\text{take } n \text{ } xs) \implies x : \text{set } xs$ 
using set-take-subset by fast

lemma in-set-dropD:  $x : \text{set}(\text{drop } n \text{ } xs) \implies x : \text{set } xs$ 
using set-drop-subset by fast

lemma append-eq-conv-conj:
   $(xs @ ys = zs) = (xs = \text{take } (\text{length } xs) \text{ } zs \wedge ys = \text{drop } (\text{length } xs) \text{ } zs)$ 
apply (induct xs arbitrary: zs, simp, clarsimp)
apply (case-tac zs, auto)
done

lemma take-add:
   $i + j \leq \text{length}(xs) \implies \text{take } (i + j) \text{ } xs = \text{take } i \text{ } xs @ \text{take } j \text{ } (\text{drop } i \text{ } xs)$ 
apply (induct xs arbitrary: i, auto)
apply (case-tac i, simp-all)
done

lemma append-eq-append-conv-if:
   $(xs_1 @ xs_2 = ys_1 @ ys_2) =$ 

```



```

    (if size  $xs_1 \leq$  size  $ys_1$ 
      then  $xs_1 = \text{take } (\text{size } xs_1) \text{ } ys_1 \wedge xs_2 = \text{drop } (\text{size } xs_1) \text{ } ys_1 @ ys_2$ 
      else  $\text{take } (\text{size } ys_1) \text{ } xs_1 = ys_1 \wedge \text{drop } (\text{size } ys_1) \text{ } xs_1 @ xs_2 = ys_2$ )
  apply (induct  $xs_1$  arbitrary:  $ys_1$ )
  apply simp
  apply (case-tac  $ys_1$ )
  apply simp-all
  done

```

```

lemma take-hd-drop:
   $n < \text{length } xs \implies \text{take } n \text{ } xs @ [\text{hd } (\text{drop } n \text{ } xs)] = \text{take } (\text{Suc } n) \text{ } xs$ 
  apply (induct  $xs$  arbitrary:  $n$ )
  apply simp
  apply (simp add: drop-Cons split: nat.split)
  done

```

```

lemma id-take-nth-drop:
   $i < \text{length } xs \implies xs = \text{take } i \text{ } xs @ xs!i \# \text{drop } (\text{Suc } i) \text{ } xs$ 
  proof -
    assume  $si: i < \text{length } xs$ 
    hence  $xs = \text{take } (\text{Suc } i) \text{ } xs @ \text{drop } (\text{Suc } i) \text{ } xs$  by auto
    moreover
    from  $si$  have  $\text{take } (\text{Suc } i) \text{ } xs = \text{take } i \text{ } xs @ [xs!i]$ 
    apply (rule-tac take-Suc-conv-app-nth) by arith
    ultimately show ?thesis by auto
  qed

```

```

lemma upd-conv-take-nth-drop:
   $i < \text{length } xs \implies xs[i:=a] = \text{take } i \text{ } xs @ a \# \text{drop } (\text{Suc } i) \text{ } xs$ 
  proof -
    assume  $i: i < \text{length } xs$ 
    have  $xs[i:=a] = (\text{take } i \text{ } xs @ xs!i \# \text{drop } (\text{Suc } i) \text{ } xs)[i:=a]$ 
    by (rule arg-cong[OF id-take-nth-drop[OF  $i$ ]])
    also have  $\dots = \text{take } i \text{ } xs @ a \# \text{drop } (\text{Suc } i) \text{ } xs$ 
    using  $i$  by (simp add: list-update-append)
    finally show ?thesis .
  qed

```

```

lemma nth-drop':
   $i < \text{length } xs \implies xs ! i \# \text{drop } (\text{Suc } i) \text{ } xs = \text{drop } i \text{ } xs$ 
  apply (induct  $i$  arbitrary:  $xs$ )
  apply (simp add: neq-Nil-conv)
  apply (erule exE)+
  apply simp
  apply (case-tac  $xs$ )
  apply simp-all
  done

```

**39.1.15** *takeWhile* and *dropWhile*

**lemma** *takeWhile-dropWhile-id* [simp]:  $\text{takeWhile } P \text{ } xs @ \text{dropWhile } P \text{ } xs = xs$   
**by** (induct xs) auto

**lemma** *takeWhile-append1* [simp]:  
 $[[x : \text{set } xs; \sim P(x)]] \implies \text{takeWhile } P \text{ } (xs @ ys) = \text{takeWhile } P \text{ } xs$   
**by** (induct xs) auto

**lemma** *takeWhile-append2* [simp]:  
 $(!!x. x : \text{set } xs \implies P x) \implies \text{takeWhile } P \text{ } (xs @ ys) = xs @ \text{takeWhile } P \text{ } ys$   
**by** (induct xs) auto

**lemma** *takeWhile-tail*:  $\neg P x \implies \text{takeWhile } P \text{ } (xs @ (x \# l)) = \text{takeWhile } P \text{ } xs$   
**by** (induct xs) auto

**lemma** *dropWhile-append1* [simp]:  
 $[[x : \text{set } xs; \sim P(x)]] \implies \text{dropWhile } P \text{ } (xs @ ys) = (\text{dropWhile } P \text{ } xs) @ ys$   
**by** (induct xs) auto

**lemma** *dropWhile-append2* [simp]:  
 $(!!x. x : \text{set } xs \implies P(x)) \implies \text{dropWhile } P \text{ } (xs @ ys) = \text{dropWhile } P \text{ } ys$   
**by** (induct xs) auto

**lemma** *set-takeWhileD*:  $x : \text{set } (\text{takeWhile } P \text{ } xs) \implies x : \text{set } xs \wedge P x$   
**by** (induct xs) (auto split: split-if-asm)

**lemma** *takeWhile-eq-all-conv*[simp]:  
 $(\text{takeWhile } P \text{ } xs = xs) = (\forall x \in \text{set } xs. P x)$   
**by**(induct xs, auto)

**lemma** *dropWhile-eq-Nil-conv*[simp]:  
 $(\text{dropWhile } P \text{ } xs = []) = (\forall x \in \text{set } xs. P x)$   
**by**(induct xs, auto)

**lemma** *dropWhile-eq-Cons-conv*:  
 $(\text{dropWhile } P \text{ } xs = y \# ys) = (xs = \text{takeWhile } P \text{ } xs @ y \# ys \ \& \ \neg P y)$   
**by**(induct xs, auto)

The following two lemmas could be generalized to an arbitrary property.

**lemma** *takeWhile-neq-rev*:  $[[\text{distinct } xs; x \in \text{set } xs]] \implies$   
 $\text{takeWhile } (\lambda y. y \neq x) \text{ } (\text{rev } xs) = \text{rev } (\text{tl } (\text{dropWhile } (\lambda y. y \neq x) \text{ } xs))$   
**by**(induct xs) (auto simp: takeWhile-tail[where l=[]])

**lemma** *dropWhile-neq-rev*:  $[[\text{distinct } xs; x \in \text{set } xs]] \implies$   
 $\text{dropWhile } (\lambda y. y \neq x) \text{ } (\text{rev } xs) = x \# \text{rev } (\text{takeWhile } (\lambda y. y \neq x) \text{ } xs)$   
**apply**(induct xs)  
**apply** simp  
**apply** auto  
**apply**(subst dropWhile-append2)

**apply** *auto*  
**done**

**lemma** *takeWhile-not-last*:  
 $\llbracket xs \neq []; \text{distinct } xs \rrbracket \implies \text{takeWhile } (\lambda y. y \neq \text{last } xs) \text{ } xs = \text{butlast } xs$   
**apply** (*induct xs*)  
**apply** *simp*  
**apply** (*case-tac xs*)  
**apply** (*auto*)  
**done**

**lemma** *takeWhile-cong* [*fundef-cong, recdef-cong*]:  
 $\llbracket l = k; !!x. x : \text{set } l \implies P \ x = Q \ x \rrbracket$   
 $\implies \text{takeWhile } P \ l = \text{takeWhile } Q \ k$   
**by** (*induct k arbitrary: l*) (*simp-all*)

**lemma** *dropWhile-cong* [*fundef-cong, recdef-cong*]:  
 $\llbracket l = k; !!x. x : \text{set } l \implies P \ x = Q \ x \rrbracket$   
 $\implies \text{dropWhile } P \ l = \text{dropWhile } Q \ k$   
**by** (*induct k arbitrary: l, simp-all*)

### 39.1.16 *zip*

**lemma** *zip-Nil* [*simp*]:  $\text{zip } [] \text{ } ys = []$   
**by** (*induct ys*) *auto*

**lemma** *zip-Cons-Cons* [*simp*]:  $\text{zip } (x \# xs) \ (y \# ys) = (x, y) \# \text{zip } xs \ ys$   
**by** *simp*

**declare** *zip-Cons* [*simp del*]

**lemma** *zip-Cons1*:  
 $\text{zip } (x \# xs) \ ys = (\text{case } ys \text{ of } [] \Rightarrow [] \mid y \# ys \Rightarrow (x, y) \# \text{zip } xs \ ys)$   
**by** (*auto split:list.split*)

**lemma** *length-zip* [*simp*]:  
 $\text{length } (\text{zip } xs \ ys) = \min (\text{length } xs) (\text{length } ys)$   
**by** (*induct xs ys rule:list-induct2'*) *auto*

**lemma** *zip-append1*:  
 $\text{zip } (xs \ @ \ ys) \ zs =$   
 $\text{zip } xs \ (\text{take } (\text{length } xs) \ zs) \ @ \ \text{zip } ys \ (\text{drop } (\text{length } xs) \ zs)$   
**by** (*induct xs zs rule:list-induct2'*) *auto*

**lemma** *zip-append2*:  
 $\text{zip } xs \ (ys \ @ \ zs) =$   
 $\text{zip } (\text{take } (\text{length } ys) \ xs) \ ys \ @ \ \text{zip } (\text{drop } (\text{length } ys) \ xs) \ zs$   
**by** (*induct xs ys rule:list-induct2'*) *auto*

**lemma** *zip-append* [*simp*]:  
 $[[ \text{length } xs = \text{length } us; \text{length } ys = \text{length } vs ]] ==>$   
 $\text{zip } (xs@ys) (us@vs) = \text{zip } xs\ us\ @\ \text{zip } ys\ vs$   
**by** (*simp add: zip-append1*)

**lemma** *zip-rev*:  
 $\text{length } xs = \text{length } ys ==> \text{zip } (\text{rev } xs) (\text{rev } ys) = \text{rev } (\text{zip } xs\ ys)$   
**by** (*induct rule:list-induct2, simp-all*)

**lemma** *map-zip-map*:  
 $\text{map } f\ (\text{zip } (\text{map } g\ xs)\ ys) = \text{map } (\lambda(x,y). f(g\ x,\ y))\ (\text{zip } xs\ ys)$   
**apply** (*induct xs arbitrary:ys*) **apply** *simp*  
**apply** (*case-tac ys*)  
**apply** *simp-all*  
**done**

**lemma** *map-zip-map2*:  
 $\text{map } f\ (\text{zip } xs\ (\text{map } g\ ys)) = \text{map } (\lambda(x,y). f(x,\ g\ y))\ (\text{zip } xs\ ys)$   
**apply** (*induct xs arbitrary:ys*) **apply** *simp*  
**apply** (*case-tac ys*)  
**apply** *simp-all*  
**done**

**lemma** *nth-zip* [*simp*]:  
 $[[ i < \text{length } xs; i < \text{length } ys ]] ==> (\text{zip } xs\ ys)!i = (xs!i,\ ys!i)$   
**apply** (*induct ys arbitrary: i xs, simp*)  
**apply** (*case-tac xs*)  
**apply** (*simp-all add: nth.simps split: nat.split*)  
**done**

**lemma** *set-zip*:  
 $\text{set } (\text{zip } xs\ ys) = \{(xs!i,\ ys!i) \mid i. i < \min(\text{length } xs)\ (\text{length } ys)\}$   
**by** (*simp add: set-conv-nth cong: rev-conj-cong*)

**lemma** *zip-update*:  
 $\text{length } xs = \text{length } ys ==> \text{zip } (xs[i:=x])\ (ys[i:=y]) = (\text{zip } xs\ ys)[i:=(x,y)]$   
**by** (*rule sym, simp add: update-zip*)

**lemma** *zip-replicate* [*simp*]:  
 $\text{zip } (\text{replicate } i\ x)\ (\text{replicate } j\ y) = \text{replicate } (\min\ i\ j)\ (x,y)$   
**apply** (*induct i arbitrary: j, auto*)  
**apply** (*case-tac j, auto*)  
**done**

**lemma** *take-zip*:  
 $\text{take } n\ (\text{zip } xs\ ys) = \text{zip } (\text{take } n\ xs)\ (\text{take } n\ ys)$   
**apply** (*induct n arbitrary: xs ys*)  
**apply** *simp*  
**apply** (*case-tac xs, simp*)

**apply** (*case-tac* *ys*, *simp-all*)  
**done**

**lemma** *drop-zip*:  
 $\text{drop } n \text{ (zip } xs \text{ } ys) = \text{zip (drop } n \text{ } xs) \text{ (drop } n \text{ } ys)$   
**apply** (*induct* *n* *arbitrary*: *xs* *ys*)  
**apply** *simp*  
**apply** (*case-tac* *xs*, *simp*)  
**apply** (*case-tac* *ys*, *simp-all*)  
**done**

**lemma** *set-zip-leftD*:  
 $(x,y) \in \text{set (zip } xs \text{ } ys) \implies x \in \text{set } xs$   
**by** (*induct* *xs* *ys* *rule*:*list-induct2*<sup>^</sup>) *auto*

**lemma** *set-zip-rightD*:  
 $(x,y) \in \text{set (zip } xs \text{ } ys) \implies y \in \text{set } ys$   
**by** (*induct* *xs* *ys* *rule*:*list-induct2*<sup>^</sup>) *auto*

**lemma** *in-set-zipE*:  
 $(x,y) : \text{set (zip } xs \text{ } ys) \implies ([x : \text{set } xs; y : \text{set } ys] \implies R) \implies R$   
**by**(*blast* *dest*: *set-zip-leftD* *set-zip-rightD*)

**lemma** *zip-map-fst-snd*:  
 $\text{zip (map fst } zs) \text{ (map snd } zs) = zs$   
**by** (*induct* *zs*) *simp-all*

**lemma** *zip-eq-conv*:  
 $\text{length } xs = \text{length } ys \implies \text{zip } xs \text{ } ys = zs \iff \text{map fst } zs = xs \wedge \text{map snd } zs = ys$   
**by** (*auto* *simp* *add*: *zip-map-fst-snd*)

### 39.1.17 *list-all2*

**lemma** *list-all2-lengthD* [*intro?*]:  
 $\text{list-all2 } P \text{ } xs \text{ } ys \implies \text{length } xs = \text{length } ys$   
**by** (*simp* *add*: *list-all2-def*)

**lemma** *list-all2-Nil* [*iff*, *code*]: *list-all2* *P* [] *ys* = (*ys* = [])  
**by** (*simp* *add*: *list-all2-def*)

**lemma** *list-all2-Nil2* [*iff*, *code*]: *list-all2* *P* *xs* [] = (*xs* = [])  
**by** (*simp* *add*: *list-all2-def*)

**lemma** *list-all2-Cons* [*iff*, *code*]:  
 $\text{list-all2 } P \text{ (} x \# xs \text{) (} y \# ys \text{) = (} P \text{ } x \text{ } y \wedge \text{list-all2 } P \text{ } xs \text{ } ys \text{)}$   
**by** (*auto* *simp* *add*: *list-all2-def*)

**lemma** *list-all2-Cons1*:  
 $\text{list-all2 } P \text{ (} x \# xs \text{) } ys = (\exists z \text{ } zs. \text{ } ys = z \# zs \wedge P \text{ } x \text{ } z \wedge \text{list-all2 } P \text{ } xs \text{ } zs)$

**by** (*cases ys*) *auto*

**lemma** *list-all2-Cons2*:

*list-all2 P xs (y # ys) = ( $\exists z zs. xs = z \# zs \wedge P z y \wedge list-all2 P zs ys$ )*

**by** (*cases xs*) *auto*

**lemma** *list-all2-rev [iff]*:

*list-all2 P (rev xs) (rev ys) = list-all2 P xs ys*

**by** (*simp add: list-all2-def zip-rev cong: conj-cong*)

**lemma** *list-all2-rev1*:

*list-all2 P (rev xs) ys = list-all2 P xs (rev ys)*

**by** (*subst list-all2-rev [symmetric]*) *simp*

**lemma** *list-all2-append1*:

*list-all2 P (xs @ ys) zs =*

*(EX us vs. zs = us @ vs  $\wedge$  length us = length xs  $\wedge$  length vs = length ys  $\wedge$*

*list-all2 P xs us  $\wedge$  list-all2 P ys vs)*

**apply** (*simp add: list-all2-def zip-append1*)

**apply** (*rule iffI*)

**apply** (*rule-tac x = take (length xs) zs in exI*)

**apply** (*rule-tac x = drop (length xs) zs in exI*)

**apply** (*force split: nat-diff-split simp add: min-def, clarify*)

**apply** (*simp add: ball-Un*)

**done**

**lemma** *list-all2-append2*:

*list-all2 P xs (ys @ zs) =*

*(EX us vs. xs = us @ vs  $\wedge$  length us = length ys  $\wedge$  length vs = length zs  $\wedge$*

*list-all2 P us ys  $\wedge$  list-all2 P vs zs)*

**apply** (*simp add: list-all2-def zip-append2*)

**apply** (*rule iffI*)

**apply** (*rule-tac x = take (length ys) xs in exI*)

**apply** (*rule-tac x = drop (length ys) xs in exI*)

**apply** (*force split: nat-diff-split simp add: min-def, clarify*)

**apply** (*simp add: ball-Un*)

**done**

**lemma** *list-all2-append*:

*length xs = length ys  $\implies$*

*list-all2 P (xs@us) (ys@vs) = (list-all2 P xs ys  $\wedge$  list-all2 P us vs)*

**by** (*induct rule:list-induct2, simp-all*)

**lemma** *list-all2-appendI [intro?, trans]*:

$\llbracket list-all2 P a b; list-all2 P c d \rrbracket \implies list-all2 P (a@c) (b@d)$

**by** (*simp add: list-all2-append list-all2-lengthD*)

**lemma** *list-all2-conv-all-nth*:

*list-all2 P xs ys =*

(length xs = length ys  $\wedge$  ( $\forall i < \text{length } xs. P (xs!i) (ys!i)$ ))  
**by** (force simp add: list-all2-def set-zip)

**lemma** list-all2-trans:

**assumes** tr:  $!!a \ b \ c. P1 \ a \ b \implies P2 \ b \ c \implies P3 \ a \ c$   
**shows**  $!!bs \ cs. \text{list-all2 } P1 \ as \ bs \implies \text{list-all2 } P2 \ bs \ cs \implies \text{list-all2 } P3 \ as \ cs$   
 (**is**  $!!bs \ cs. PROP \ ?Q \ as \ bs \ cs$ )

**proof** (induct as)

**fix** x xs bs **assume** I1:  $!!bs \ cs. PROP \ ?Q \ xs \ bs \ cs$

**show**  $!!cs. PROP \ ?Q \ (x \# \ xs) \ bs \ cs$

**proof** (induct bs)

**fix** y ys cs **assume** I2:  $!!cs. PROP \ ?Q \ (x \# \ xs) \ ys \ cs$

**show**  $PROP \ ?Q \ (x \# \ xs) \ (y \# \ ys) \ cs$

**by** (induct cs) (auto intro: tr I1 I2)

**qed** simp

**qed** simp

**lemma** list-all2-all-nthI [intro?]:

$\text{length } a = \text{length } b \implies (\bigwedge n. n < \text{length } a \implies P (a!n) (b!n)) \implies \text{list-all2 } P \ a \ b$

**by** (simp add: list-all2-conv-all-nth)

**lemma** list-all2I:

$\forall x \in \text{set } (\text{zip } a \ b). \text{split } P \ x \implies \text{length } a = \text{length } b \implies \text{list-all2 } P \ a \ b$

**by** (simp add: list-all2-def)

**lemma** list-all2-nthD:

$\llbracket \text{list-all2 } P \ xs \ ys; p < \text{size } xs \rrbracket \implies P (xs!p) (ys!p)$

**by** (simp add: list-all2-conv-all-nth)

**lemma** list-all2-nthD2:

$\llbracket \text{list-all2 } P \ xs \ ys; p < \text{size } ys \rrbracket \implies P (xs!p) (ys!p)$

**by** (frule list-all2-lengthD) (auto intro: list-all2-nthD)

**lemma** list-all2-map1:

$\text{list-all2 } P \ (\text{map } f \ as) \ bs = \text{list-all2 } (\lambda x \ y. P (f \ x) \ y) \ as \ bs$

**by** (simp add: list-all2-conv-all-nth)

**lemma** list-all2-map2:

$\text{list-all2 } P \ as \ (\text{map } f \ bs) = \text{list-all2 } (\lambda x \ y. P \ x \ (f \ y)) \ as \ bs$

**by** (auto simp add: list-all2-conv-all-nth)

**lemma** list-all2-refl [intro?]:

$(\bigwedge x. P \ x \ x) \implies \text{list-all2 } P \ xs \ xs$

**by** (simp add: list-all2-conv-all-nth)

**lemma** list-all2-update-cong:

$\llbracket i < \text{size } xs; \text{list-all2 } P \ xs \ ys; P \ x \ y \rrbracket \implies \text{list-all2 } P \ (xs[i:=x]) \ (ys[i:=y])$

**by** (simp add: list-all2-conv-all-nth nth-list-update)

**lemma** *list-all2-update-cong2*:

$\llbracket \text{list-all2 } P \text{ } xs \text{ } ys; P \text{ } x \text{ } y; i < \text{length } ys \rrbracket \implies \text{list-all2 } P \text{ } (xs[i:=x]) \text{ } (ys[i:=y])$   
**by** (*simp add: list-all2-lengthD list-all2-update-cong*)

**lemma** *list-all2-takeI* [*simp,intro?*]:

$\text{list-all2 } P \text{ } xs \text{ } ys \implies \text{list-all2 } P \text{ } (\text{take } n \text{ } xs) \text{ } (\text{take } n \text{ } ys)$   
**apply** (*induct xs arbitrary: n ys*)  
**apply** *simp*  
**apply** (*clarsimp simp add: list-all2-Cons1*)  
**apply** (*case-tac n*)  
**apply** *auto*  
**done**

**lemma** *list-all2-dropI* [*simp,intro?*]:

$\text{list-all2 } P \text{ } as \text{ } bs \implies \text{list-all2 } P \text{ } (\text{drop } n \text{ } as) \text{ } (\text{drop } n \text{ } bs)$   
**apply** (*induct as arbitrary: n bs, simp*)  
**apply** (*clarsimp simp add: list-all2-Cons1*)  
**apply** (*case-tac n, simp, simp*)  
**done**

**lemma** *list-all2-mono* [*intro?*]:

$\text{list-all2 } P \text{ } xs \text{ } ys \implies (\bigwedge xs \text{ } ys. P \text{ } xs \text{ } ys \implies Q \text{ } xs \text{ } ys) \implies \text{list-all2 } Q \text{ } xs \text{ } ys$   
**apply** (*induct xs arbitrary: ys, simp*)  
**apply** (*case-tac ys, auto*)  
**done**

**lemma** *list-all2-eq*:

$xs = ys \iff \text{list-all2 } (op =) \text{ } xs \text{ } ys$   
**by** (*induct xs ys rule: list-induct2'*) *auto*

### 39.1.18 foldl and foldr

**lemma** *foldl-append* [*simp*]:

$\text{foldl } f \text{ } a \text{ } (xs @ ys) = \text{foldl } f \text{ } (\text{foldl } f \text{ } a \text{ } xs) \text{ } ys$   
**by** (*induct xs arbitrary: a*) *auto*

**lemma** *foldr-append*[*simp*]:  $\text{foldr } f \text{ } (xs @ ys) \text{ } a = \text{foldr } f \text{ } xs \text{ } (\text{foldr } f \text{ } ys \text{ } a)$

**by** (*induct xs*) *auto*

**lemma** *foldr-map*:  $\text{foldr } g \text{ } (\text{map } f \text{ } xs) \text{ } a = \text{foldr } (g \circ f) \text{ } xs \text{ } a$

**by**(*induct xs*) *simp-all*

For efficient code generation: avoid intermediate list.

**lemma** *foldl-map*[*code unfold*]:

$\text{foldl } g \text{ } a \text{ } (\text{map } f \text{ } xs) = \text{foldl } (\%a \text{ } x. g \text{ } a \text{ } (f \text{ } x)) \text{ } a \text{ } xs$   
**by**(*induct xs arbitrary:a*) *simp-all*

**lemma** *foldl-cong* [*fundef-cong, recdef-cong*]:

$\llbracket a = b; l = k; !!a \text{ } x. x : \text{set } l \implies f \text{ } a \text{ } x = g \text{ } a \text{ } x \rrbracket$



$\Rightarrow \text{foldl } f \ a \ l = \text{foldl } g \ b \ k$   
**by** (induct  $k$  arbitrary:  $a \ b \ l$ ) simp-all

**lemma** foldr-cong [fundef-cong, recdef-cong]:  
 $\llbracket a = b; l = k; \forall x. x : \text{set } l \Rightarrow f \ x \ a = g \ x \ a \rrbracket$   
 $\Rightarrow \text{foldr } f \ l \ a = \text{foldr } g \ k \ b$   
**by** (induct  $k$  arbitrary:  $a \ b \ l$ ) simp-all

**lemma** (in semigroup-add) foldl-assoc:  
**shows**  $\text{foldl } op + (x+y) \ zs = x + (\text{foldl } op + y \ zs)$   
**by** (induct  $zs$  arbitrary:  $y$ ) (simp-all add:add-assoc)

**lemma** (in monoid-add) foldl-absorb0:  
**shows**  $x + (\text{foldl } op + 0 \ zs) = \text{foldl } op + x \ zs$   
**by** (induct  $zs$ ) (simp-all add:foldl-assoc)

The “First Duality Theorem” in Bird & Wadler:

**lemma** foldl-foldr1-lemma:  
 $\text{foldl } op + a \ xs = a + \text{foldr } op + xs \ (0::'a::\text{monoid-add})$   
**by** (induct  $xs$  arbitrary:  $a$ ) (auto simp:add-assoc)

**corollary** foldl-foldr1:  
 $\text{foldl } op + 0 \ xs = \text{foldr } op + xs \ (0::'a::\text{monoid-add})$   
**by** (simp add:foldl-foldr1-lemma)

The “Third Duality Theorem” in Bird & Wadler:

**lemma** foldr-foldl:  $\text{foldr } f \ xs \ a = \text{foldl } (\%x \ y. f \ y \ x) \ a \ (\text{rev } xs)$   
**by** (induct  $xs$ ) auto

**lemma** foldl-foldr:  $\text{foldl } f \ a \ xs = \text{foldr } (\%x \ y. f \ y \ x) \ (\text{rev } xs) \ a$   
**by** (simp add: foldr-foldl [of  $\%x \ y. f \ y \ x \ \text{rev } xs$ ])

**lemma** (in ab-semigroup-add) foldr-conv-foldl:  $\text{foldr } op + xs \ a = \text{foldl } op + a \ xs$   
**by** (induct  $xs$ , auto simp add: foldl-assoc add-commute)

Note:  $n \leq \text{foldl } (op \ +) \ n \ ns$  looks simpler, but is more difficult to use because it requires an additional transitivity step.

**lemma** start-le-sum:  $(m::\text{nat}) \leq n \Rightarrow m \leq \text{foldl } (op \ +) \ n \ ns$   
**by** (induct  $ns$  arbitrary:  $n$ ) auto

**lemma** elem-le-sum:  $(n::\text{nat}) : \text{set } ns \Rightarrow n \leq \text{foldl } (op \ +) \ 0 \ ns$   
**by** (force intro: start-le-sum simp add: in-set-conv-decomp)

**lemma** sum-eq-0-conv [iff]:  
 $(\text{foldl } (op \ +) \ (m::\text{nat}) \ ns = 0) = (m = 0 \wedge (\forall n \in \text{set } ns. n = 0))$   
**by** (induct  $ns$  arbitrary:  $m$ ) auto

**lemma** foldr-invariant:  
 $\llbracket Q \ x ; \forall x \in \text{set } xs. P \ x ; \forall x \ y. P \ x \wedge Q \ y \longrightarrow Q \ (f \ x \ y) \rrbracket \Longrightarrow Q \ (\text{foldr } f \ xs \ x)$

**by** (*induct xs, simp-all*)

**lemma** *foldl-invariant*:

$\llbracket Q\ x ; \forall\ x \in \text{set}\ xs.\ P\ x ; \forall\ x\ y.\ P\ x \wedge Q\ y \longrightarrow Q\ (f\ y\ x) \rrbracket \implies Q\ (\text{foldl}\ f\ x\ xs)$   
**by** (*induct xs arbitrary: x, simp-all*)

*foldl* and *concat*

**lemma** *foldl-conv-concat*:

*foldl* (*op @*) *xs xss* = *xs @ concat xss*

**proof** (*induct xss arbitrary: xs*)

**case** *Nil* **show** ?*case* **by** *simp*

**next**

**interpret** *monoid-add* [] *op @* **proof** **qed** *simp-all*

**case** *Cons* **then show** ?*case* **by** (*simp add: foldl-absorb0*)

**qed**

**lemma** *concat-conv-foldl*: *concat xss* = *foldl* (*op @*) [] *xss*

**by** (*simp add: foldl-conv-concat*)

### 39.1.19 List summation: *listsum* and $\sum$

**lemma** *listsum-append* [*simp*]: *listsum* (*xs @ ys*) = *listsum xs* + *listsum ys*

**by** (*induct xs*) (*simp-all add: add-assoc*)

**lemma** *listsum-rev* [*simp*]:

**fixes** *xs* :: 'a::comm-monoid-add list

**shows** *listsum* (*rev xs*) = *listsum xs*

**by** (*induct xs*) (*simp-all add: add-ac*)

**lemma** *listsum-foldr*: *listsum xs* = *foldr* (*op +*) *xs 0*

**by** (*induct xs*) *auto*

**lemma** *length-concat*: *length* (*concat xss*) = *listsum* (*map length xss*)

**by** (*induct xss*) *simp-all*

For efficient code generation — *listsum* is not tail recursive but *foldl* is.

**lemma** *listsum[code unfold]*: *listsum xs* = *foldl* (*op +*) *0 xs*

**by**(*simp add: listsum-foldr foldl-foldr1*)

Some syntactic sugar for summing a function over a list:

**syntax**

*-listsum* :: *pttrn* => 'a list => 'b => 'b ((*3SUM* -<-.-) [0, 51, 10] 10)

**syntax** (*xsymbols*)

*-listsum* :: *pttrn* => 'a list => 'b => 'b ((*3Σ* -<-.-) [0, 51, 10] 10)

**syntax** (*HTML output*)

*-listsum* :: *pttrn* => 'a list => 'b => 'b ((*3Σ* -<-.-) [0, 51, 10] 10)

**translations** — Beware of argument permutation!

*SUM* *x* <- *xs*. *b* == *CONST listsum* (*map* (%*x*. *b*) *xs*)

$$\sum x \leftarrow xs. b == \text{CONST listsum } (\text{map } (\%x. b) xs)$$

**lemma** *listsum-triv*:  $(\sum x \leftarrow xs. r) = \text{of-nat } (\text{length } xs) * r$   
**by** (*induct xs*) (*simp-all add: left-distrib*)

**lemma** *listsum-0* [*simp*]:  $(\sum x \leftarrow xs. 0) = 0$   
**by** (*induct xs*) (*simp-all add: left-distrib*)

For non-Abelian groups *xs* needs to be reversed on one side:

**lemma** *uminus-listsum-map*:  
**fixes** *f* :: 'a  $\Rightarrow$  'b::ab-group-add  
**shows**  $-\text{listsum } (\text{map } f xs) = (\text{listsum } (\text{map } (\text{uminus } o f) xs))$   
**by** (*induct xs*) *simp-all*

### 39.1.20 *upt*

**lemma** *upt-rec[code]*:  $[i..<j] = (\text{if } i < j \text{ then } i \# [\text{Suc } i..<j] \text{ else } [])$   
— *simp* does not terminate!  
**by** (*induct j*) *auto*

**lemma** *upt-conv-Nil* [*simp*]:  $j \leq i \implies [i..<j] = []$   
**by** (*subst upt-rec*) *simp*

**lemma** *upt-eq-Nil-conv* [*simp*]:  $([i..<j] = []) = (j = 0 \vee j \leq i)$   
**by**(*induct j*)*simp-all*

**lemma** *upt-eq-Cons-conv*:  
 $([i..<j] = x \# xs) = (i < j \ \& \ i = x \ \& \ [i+1..<j] = xs)$   
**apply**(*induct j arbitrary: x xs*)  
**apply** *simp*  
**apply**(*clarsimp simp add: append-eq-Cons-conv*)  
**apply** *arith*  
**done**

**lemma** *upt-Suc-append*:  $i \leq j \implies [i..<(\text{Suc } j)] = [i..<j] @ [j]$   
— Only needed if *upt-Suc* is deleted from the simpset.  
**by** *simp*

**lemma** *upt-conv-Cons*:  $i < j \implies [i..<j] = i \# [\text{Suc } i..<j]$   
**by** (*simp add: upt-rec*)

**lemma** *upt-add-eq-append*:  $i \leq j \implies [i..<j+k] = [i..<j] @ [j..<j+k]$   
— LOOPS as a simprule, since  $j \leq j$ .  
**by** (*induct k*) *auto*

**lemma** *length-upt* [*simp*]:  $\text{length } [i..<j] = j - i$   
**by** (*induct j*) (*auto simp add: Suc-diff-le*)

**lemma** *nth-upt* [*simp*]:  $i + k < j \implies [i..<j] ! k = i + k$

```

apply (induct j)
apply (auto simp add: less-Suc-eq nth-append split: nat-diff-split)
done

```

```

lemma hd-upt[simp]: i < j ==> hd[i..j] = i
by(simp add:upt-conv-Cons)

```

```

lemma last-upt[simp]: i < j ==> last[i..j] = j - 1
apply(cases j)
apply simp
by(simp add:upt-Suc-append)

```

```

lemma take-upt [simp]: i+m <= n ==> take m [i..n] = [i..i+m]
apply (induct m arbitrary: i, simp)
apply (subst upt-rec)
apply (rule sym)
apply (subst upt-rec)
apply (simp del: upt.simps)
done

```

```

lemma drop-upt[simp]: drop m [i..j] = [i+m..j]
apply(induct j)
apply auto
done

```

```

lemma map-Suc-upt: map Suc [m..n] = [Suc m..Suc n]
by (induct n) auto

```

```

lemma nth-map-upt: i < n-m ==> (map f [m..n]) ! i = f(m+i)
apply (induct n m arbitrary: i rule: diff-induct)
prefer 3 apply (subst map-Suc-upt[symmetric])
apply (auto simp add: less-diff-conv nth-upt)
done

```

```

lemma nth-take-lemma:
  k <= length xs ==> k <= length ys ==>
  (!!i. i < k --> xs!i = ys!i) ==> take k xs = take k ys
apply (atomize, induct k arbitrary: xs ys)
apply (simp-all add: less-Suc-eq-0-disj all-conj-distrib, clarify)

```

Both lists must be non-empty

```

apply (case-tac xs, simp)
apply (case-tac ys, clarify)
apply (simp (no-asm-use))
apply clarify

```

prenexing's needed, not miniscoping

```

apply (simp (no-asm-use) add: all-simps [symmetric] del: all-simps)

```

**apply** *blast*  
**done**

**lemma** *nth-equalityI*:  
 $\llbracket \text{length } xs = \text{length } ys; \text{ALL } i < \text{length } xs. xs!i = ys!i \rrbracket \implies xs = ys$   
**apply** (*frule* *nth-take-lemma* [*OF le-refl eq-imp-le*])  
**apply** (*simp-all* *add: take-all*)  
**done**

**lemma** *map-nth*:  
 $\text{map } (\lambda i. xs ! i) [0..<\text{length } xs] = xs$   
**by** (*rule* *nth-equalityI*, *auto*)

**lemma** *list-all2-antisym*:  
 $\llbracket (\bigwedge x y. \llbracket P x y; Q y x \rrbracket \implies x = y); \text{list-all2 } P \text{ } xs \text{ } ys; \text{list-all2 } Q \text{ } ys \text{ } xs \rrbracket$   
 $\implies xs = ys$   
**apply** (*simp* *add: list-all2-conv-all-nth*)  
**apply** (*rule* *nth-equalityI*, *blast*, *simp*)  
**done**

**lemma** *take-equalityI*:  $(\forall i. \text{take } i \text{ } xs = \text{take } i \text{ } ys) \implies xs = ys$   
— The famous take-lemma.  
**apply** (*drule-tac*  $x = \max (\text{length } xs) (\text{length } ys)$  **in** *spec*)  
**apply** (*simp* *add: le-max-iff-disj take-all*)  
**done**

**lemma** *take-Cons'*:  
 $\text{take } n \text{ } (x \# xs) = (\text{if } n = 0 \text{ then } [] \text{ else } x \# \text{take } (n - 1) \text{ } xs)$   
**by** (*cases* *n*) *simp-all*

**lemma** *drop-Cons'*:  
 $\text{drop } n \text{ } (x \# xs) = (\text{if } n = 0 \text{ then } x \# xs \text{ else } \text{drop } (n - 1) \text{ } xs)$   
**by** (*cases* *n*) *simp-all*

**lemma** *nth-Cons'*:  $(x \# xs)!n = (\text{if } n = 0 \text{ then } x \text{ else } xs!(n - 1))$   
**by** (*cases* *n*) *simp-all*

**lemmas** *take-Cons-number-of* = *take-Cons'*[*of number-of v,standard*]  
**lemmas** *drop-Cons-number-of* = *drop-Cons'*[*of number-of v,standard*]  
**lemmas** *nth-Cons-number-of* = *nth-Cons'*[*of - - number-of v,standard*]

**declare** *take-Cons-number-of* [*simp*]  
*drop-Cons-number-of* [*simp*]  
*nth-Cons-number-of* [*simp*]

**39.1.21** *distinct and remdups***lemma** *distinct-append* [simp]: $distinct\ (xs\ @\ ys) = (distinct\ xs \wedge distinct\ ys \wedge set\ xs \cap set\ ys = \{\})$ **by** (induct *xs*) auto**lemma** *distinct-rev*[simp]:  $distinct(rev\ xs) = distinct\ xs$ **by**(induct *xs*) auto**lemma** *set-remdups* [simp]:  $set\ (remdups\ xs) = set\ xs$ **by** (induct *xs*) (auto simp add: insert-absorb)**lemma** *distinct-remdups* [iff]:  $distinct\ (remdups\ xs)$ **by** (induct *xs*) auto**lemma** *distinct-remdups-id*:  $distinct\ xs ==> remdups\ xs = xs$ **by** (induct *xs*, auto)**lemma** *remdups-id-iff-distinct* [simp]:  $remdups\ xs = xs \longleftrightarrow distinct\ xs$ **by** (metis *distinct-remdups distinct-remdups-id*)**lemma** *finite-distinct-list*:  $finite\ A \implies \exists x. set\ xs = A \ \& \ distinct\ xs$ **by** (metis *distinct-remdups finite-list set-remdups*)**lemma** *remdups-eq-nil-iff* [simp]:  $(remdups\ x = []) = (x = [])$ **by** (induct *x*, auto)**lemma** *remdups-eq-nil-right-iff* [simp]:  $([] = remdups\ x) = (x = [])$ **by** (induct *x*, auto)**lemma** *length-remdups-leq*[iff]:  $length(remdups\ xs) \leq length\ xs$ **by** (induct *xs*) auto**lemma** *length-remdups-eq*[iff]: $(length\ (remdups\ xs) = length\ xs) = (remdups\ xs = xs)$ **apply**(induct *xs*)**apply** auto**apply**(subgoal-tac  $length\ (remdups\ xs) \leq length\ xs$ )**apply** arith**apply**(rule *length-remdups-leq*)**done****lemma** *distinct-map*: $distinct(map\ f\ xs) = (distinct\ xs \ \& \ inj-on\ f\ (set\ xs))$ **by** (induct *xs*) auto**lemma** *distinct-filter* [simp]:  $distinct\ xs ==> distinct\ (filter\ P\ xs)$ **by** (induct *xs*) auto

```

lemma distinct-upt[simp]: distinct[i..<j]
by (induct j) auto

lemma distinct-take[simp]: distinct xs  $\implies$  distinct (take i xs)
apply (induct xs arbitrary: i)
apply simp
apply (case-tac i)
apply simp-all
apply (blast dest:in-set-takeD)
done

lemma distinct-drop[simp]: distinct xs  $\implies$  distinct (drop i xs)
apply (induct xs arbitrary: i)
apply simp
apply (case-tac i)
apply simp-all
done

lemma distinct-list-update:
assumes d: distinct xs and a:  $a \notin \text{set } xs - \{xs!i\}$ 
shows distinct (xs[i:=a])
proof (cases i < length xs)
  case True
    with a have  $a \notin \text{set } (take\ i\ xs\ @\ xs\ !\ i\ \# \ drop\ (Suc\ i)\ xs) - \{xs!i\}$ 
    apply (drule-tac id-take-nth-drop) by simp
    with d True show ?thesis
    apply (simp add: upd-conv-take-nth-drop)
    apply (drule subst [OF id-take-nth-drop]) apply assumption
    apply simp apply (cases a = xs!i) apply simp by blast
  next
    case False with d show ?thesis by auto
qed

It is best to avoid this indexed version of distinct, but sometimes it is useful.

lemma distinct-conv-nth:
 $distinct\ xs = (\forall i < size\ xs. \forall j < size\ xs. i \neq j \longrightarrow xs!i \neq xs!j)$ 
apply (induct xs, simp, simp)
apply (rule iffI, clarsimp)
apply (case-tac i)
apply (case-tac j, simp)
apply (simp add: set-conv-nth)
apply (case-tac j)
apply (clarsimp simp add: set-conv-nth, simp)
apply (rule conjI)

apply (clarsimp simp add: set-conv-nth)
apply (erule-tac x = 0 in allE, simp)
apply (erule-tac x = Suc i in allE, simp, clarsimp)

```

```

apply (erule-tac  $x = \text{Suc } i$  in  $\text{allE}$ ,  $\text{simp}$ )
apply (erule-tac  $x = \text{Suc } j$  in  $\text{allE}$ ,  $\text{simp}$ )
done

```

```

lemma nth-eq-iff-index-eq:
   $\llbracket \text{distinct } xs; i < \text{length } xs; j < \text{length } xs \rrbracket \implies (xs!i = xs!j) = (i = j)$ 
by(auto  $\text{simp}$ : distinct-conv-nth)

```

```

lemma distinct-card:  $\text{distinct } xs \implies \text{card } (\text{set } xs) = \text{size } xs$ 
by (induct  $xs$ ) auto

```

```

lemma card-distinct:  $\text{card } (\text{set } xs) = \text{size } xs \implies \text{distinct } xs$ 
proof (induct  $xs$ )
  case Nil thus ?case by  $\text{simp}$ 
next
  case (Cons  $x$   $xs$ )
  show ?case
  proof (cases  $x \in \text{set } xs$ )
    case False with Cons show ?thesis by  $\text{simp}$ 
  next
    case True with Cons.prems
    have  $\text{card } (\text{set } xs) = \text{Suc } (\text{length } xs)$ 
      by ( $\text{simp}$  add: card-insert-if split: split-if-asm)
    moreover have  $\text{card } (\text{set } xs) \leq \text{length } xs$  by (rule card-length)
    ultimately have False by  $\text{simp}$ 
    thus ?thesis ..
  qed
qed

```

```

lemma not-distinct-decomp:  $\sim \text{distinct } ws \implies \exists x y z. ws = xs @ [y] @ ys @ [y] @ zs$ 
apply (induct  $n == \text{length } ws$  arbitrary:ws) apply  $\text{simp}$ 
apply (case-tac  $ws$ ) apply  $\text{simp}$ 
apply ( $\text{simp}$  split:split-if-asm)
apply (metis Cons-eq-appendI eq-Nil-appendI split-list)
done

```

```

lemma length-remdups-concat:
   $\text{length}(\text{remdups}(\text{concat } xss)) = \text{card}(\bigcup xs \in \text{set } xss. \text{set } xs)$ 
by( $\text{simp}$  add: set-concat distinct-card[symmetric])

```

### 39.1.22 *remove1*

```

lemma remove1-append:
   $\text{remove1 } x (xs @ ys) =$ 
  (if  $x \in \text{set } xs$  then  $\text{remove1 } x xs @ ys$  else  $xs @ \text{remove1 } x ys$ )
by (induct  $xs$ ) auto

```

```

lemma in-set-remove1[ $\text{simp}$ ]:

```



```

   $a \neq b \implies a : \text{set}(\text{remove1 } b \text{ } xs) = (a : \text{set } xs)$ 
apply (induct xs)
apply auto
done

```

```

lemma set-remove1-subset:  $\text{set}(\text{remove1 } x \text{ } xs) \leq \text{set } xs$ 
apply(induct xs)
  apply simp
apply simp
apply blast
done

```

```

lemma set-remove1-eq [simp]:  $\text{distinct } xs \implies \text{set}(\text{remove1 } x \text{ } xs) = \text{set } xs - \{x\}$ 
apply(induct xs)
  apply simp
apply simp
apply blast
done

```

```

lemma length-remove1:
   $\text{length}(\text{remove1 } x \text{ } xs) = (\text{if } x : \text{set } xs \text{ then } \text{length } xs - 1 \text{ else } \text{length } xs)$ 
apply (induct xs)
  apply (auto dest!:length-pos-if-in-set)
done

```

```

lemma remove1-filter-not[simp]:
   $\neg P \ x \implies \text{remove1 } x \text{ } (\text{filter } P \text{ } xs) = \text{filter } P \text{ } xs$ 
by(induct xs) auto

```

```

lemma notin-set-remove1[simp]:  $x \notin \text{set } xs \implies x \notin \text{set}(\text{remove1 } y \text{ } xs)$ 
apply(insert set-remove1-subset)
apply fast
done

```

```

lemma distinct-remove1[simp]:  $\text{distinct } xs \implies \text{distinct}(\text{remove1 } x \text{ } xs)$ 
by (induct xs) simp-all

```

### 39.1.23 *removeAll*

```

lemma removeAll-append[simp]:
   $\text{removeAll } x \text{ } (xs @ ys) = \text{removeAll } x \text{ } xs @ \text{removeAll } x \text{ } ys$ 
by (induct xs) auto

```

```

lemma set-removeAll[simp]:  $\text{set}(\text{removeAll } x \text{ } xs) = \text{set } xs - \{x\}$ 
by (induct xs) auto

```

```

lemma removeAll-id[simp]:  $x \notin \text{set } xs \implies \text{removeAll } x \text{ } xs = xs$ 
by (induct xs) auto

```

**lemma** *removeAll-filter-not*[simp]:  
 $\neg P x \implies \text{removeAll } x \ (\text{filter } P \ xs) = \text{filter } P \ xs$   
**by**(*induct xs*) *auto*

**lemma** *distinct-remove1-removeAll*:  
 $\text{distinct } xs \implies \text{remove1 } x \ xs = \text{removeAll } x \ xs$   
**by** (*induct xs*) *simp-all*

**lemma** *map-removeAll-inj-on*:  $\text{inj-on } f \ (\text{insert } x \ (\text{set } xs)) \implies$   
 $\text{map } f \ (\text{removeAll } x \ xs) = \text{removeAll } (f \ x) \ (\text{map } f \ xs)$   
**by** (*induct xs*) (*simp-all add:inj-on-def*)

**lemma** *map-removeAll-inj*:  $\text{inj } f \implies$   
 $\text{map } f \ (\text{removeAll } x \ xs) = \text{removeAll } (f \ x) \ (\text{map } f \ xs)$   
**by**(*metis map-removeAll-inj-on subset-inj-on subset-UNIV*)

### 39.1.24 replicate

**lemma** *length-replicate* [simp]:  $\text{length } (\text{replicate } n \ x) = n$   
**by** (*induct n*) *auto*

**lemma** *map-replicate* [simp]:  $\text{map } f \ (\text{replicate } n \ x) = \text{replicate } n \ (f \ x)$   
**by** (*induct n*) *auto*

**lemma** *replicate-app-Cons-same*:  
 $(\text{replicate } n \ x) @ (x \ \# \ xs) = x \ \# \ \text{replicate } n \ x @ xs$   
**by** (*induct n*) *auto*

**lemma** *rev-replicate* [simp]:  $\text{rev } (\text{replicate } n \ x) = \text{replicate } n \ x$   
**apply** (*induct n, simp*)  
**apply** (*simp add: replicate-app-Cons-same*)  
**done**

**lemma** *replicate-add*:  $\text{replicate } (n + m) \ x = \text{replicate } n \ x @ \text{replicate } m \ x$   
**by** (*induct n*) *auto*

Courtesy of Matthias Daum:

**lemma** *append-replicate-commute*:  
 $\text{replicate } n \ x @ \text{replicate } k \ x = \text{replicate } k \ x @ \text{replicate } n \ x$   
**apply** (*simp add: replicate-add [THEN sym]*)  
**apply** (*simp add: add-commute*)  
**done**

**lemma** *hd-replicate* [simp]:  $n \neq 0 \implies \text{hd } (\text{replicate } n \ x) = x$   
**by** (*induct n*) *auto*

**lemma** *tl-replicate* [*simp*]:  $n \neq 0 \implies \text{tl } (\text{replicate } n \ x) = \text{replicate } (n - 1) \ x$   
**by** (*induct n*) *auto*

**lemma** *last-replicate* [*simp*]:  $n \neq 0 \implies \text{last } (\text{replicate } n \ x) = x$   
**by** (*atomize (full)*, *induct n*) *auto*

**lemma** *nth-replicate* [*simp*]:  $i < n \implies (\text{replicate } n \ x)!i = x$   
**apply** (*induct n arbitrary: i, simp*)  
**apply** (*simp add: nth-Cons split: nat.split*)  
**done**

Courtesy of Matthias Daum (2 lemmas):

**lemma** *take-replicate* [*simp*]:  $\text{take } i \ (\text{replicate } k \ x) = \text{replicate } (\min i \ k) \ x$   
**apply** (*case-tac k ≤ i*)  
**apply** (*simp add: min-def*)  
**apply** (*drule not-leE*)  
**apply** (*simp add: min-def*)  
**apply** (*subgoal-tac replicate k x = replicate i x @ replicate (k - i) x*)  
**apply** *simp*  
**apply** (*simp add: replicate-add [symmetric]*)  
**done**

**lemma** *drop-replicate* [*simp*]:  $\text{drop } i \ (\text{replicate } k \ x) = \text{replicate } (k - i) \ x$   
**apply** (*induct k arbitrary: i*)  
**apply** *simp*  
**apply** *clarsimp*  
**apply** (*case-tac i*)  
**apply** *simp*  
**apply** *clarsimp*  
**done**

**lemma** *set-replicate-Suc*:  $\text{set } (\text{replicate } (\text{Suc } n) \ x) = \{x\}$   
**by** (*induct n*) *auto*

**lemma** *set-replicate* [*simp*]:  $n \neq 0 \implies \text{set } (\text{replicate } n \ x) = \{x\}$   
**by** (*fast dest!: not0-implies-Suc intro!: set-replicate-Suc*)

**lemma** *set-replicate-conv-if*:  $\text{set } (\text{replicate } n \ x) = (\text{if } n = 0 \text{ then } \{\} \text{ else } \{x\})$   
**by** *auto*

**lemma** *in-set-replicateD*:  $x : \text{set } (\text{replicate } n \ y) \implies x = y$   
**by** (*simp add: set-replicate-conv-if split: split-if-asm*)

**lemma** *replicate-append-same*:  
 $\text{replicate } i \ x \ @ \ [x] = x \ \# \ \text{replicate } i \ x$   
**by** (*induct i*) *simp-all*

**lemma** *map-replicate-trivial*:

$\text{map } (\lambda i. x) [0..<i] = \text{replicate } i x$   
**by** (*induct i*) (*simp-all add: replicate-append-same*)

**lemma** *replicate-empty[simp]*:  $(\text{replicate } n x = []) \longleftrightarrow n=0$   
**by** (*induct n*) *auto*

**lemma** *empty-replicate[simp]*:  $([] = \text{replicate } n x) \longleftrightarrow n=0$   
**by** (*induct n*) *auto*

**lemma** *replicate-eq-replicate[simp]*:  
 $(\text{replicate } m x = \text{replicate } n y) \longleftrightarrow (m=n \ \& \ (m \neq 0 \longrightarrow x=y))$   
**apply** (*induct m arbitrary: n*)  
**apply** *simp*  
**apply** (*induct-tac n*)  
**apply** *auto*  
**done**

### 39.1.25 *rotate1* and *rotate*

**lemma** *rotate-simps[simp]*:  $\text{rotate1 } [] = [] \wedge \text{rotate1 } (x\#xs) = xs @ [x]$   
**by** (*simp add:rotate1-def*)

**lemma** *rotate0[simp]*:  $\text{rotate } 0 = \text{id}$   
**by** (*simp add:rotate-def*)

**lemma** *rotate-Suc[simp]*:  $\text{rotate } (\text{Suc } n) xs = \text{rotate1 } (\text{rotate } n xs)$   
**by** (*simp add:rotate-def*)

**lemma** *rotate-add*:  
 $\text{rotate } (m+n) = \text{rotate } m \circ \text{rotate } n$   
**by** (*simp add:rotate-def funpow-add*)

**lemma** *rotate-rotate*:  $\text{rotate } m (\text{rotate } n xs) = \text{rotate } (m+n) xs$   
**by** (*simp add:rotate-add*)

**lemma** *rotate1-rotate-swap*:  $\text{rotate1 } (\text{rotate } n xs) = \text{rotate } n (\text{rotate1 } xs)$   
**by** (*simp add:rotate-def funpow-swap1*)

**lemma** *rotate1-length01[simp]*:  $\text{length } xs \leq 1 \implies \text{rotate1 } xs = xs$   
**by** (*cases xs*) *simp-all*

**lemma** *rotate-length01[simp]*:  $\text{length } xs \leq 1 \implies \text{rotate } n xs = xs$   
**apply** (*induct n*)  
**apply** *simp*  
**apply** (*simp add:rotate-def*)  
**done**

**lemma** *rotate1-hd-tl*:  $xs \neq [] \implies \text{rotate1 } xs = \text{tl } xs @ [\text{hd } xs]$

**by**(*simp add:rotate1-def split:list.split*)

**lemma** *rotate-drop-take*:

*rotate n xs = drop (n mod length xs) xs @ take (n mod length xs) xs*  
**apply**(*induct n*)  
**apply** *simp*  
**apply**(*simp add:rotate-def*)  
**apply**(*cases xs = []*)  
**apply** (*simp*)  
**apply**(*case-tac n mod length xs = 0*)  
**apply**(*simp add:mod-Suc*)  
**apply**(*simp add: rotate1-hd-tl drop-Suc take-Suc*)  
**apply**(*simp add:mod-Suc rotate1-hd-tl drop-Suc[symmetric] drop-tl[symmetric]*  
*take-hd-drop linorder-not-le*)  
**done**

**lemma** *rotate-conv-mod*: *rotate n xs = rotate (n mod length xs) xs*  
**by**(*simp add:rotate-drop-take*)

**lemma** *rotate-id[simp]*: *n mod length xs = 0  $\implies$  rotate n xs = xs*  
**by**(*simp add:rotate-drop-take*)

**lemma** *length-rotate1[simp]*: *length(rotate1 xs) = length xs*  
**by**(*simp add:rotate1-def split:list.split*)

**lemma** *length-rotate[simp]*: *length(rotate n xs) = length xs*  
**by** (*induct n arbitrary: xs*) (*simp-all add:rotate-def*)

**lemma** *distinct1-rotate[simp]*: *distinct(rotate1 xs) = distinct xs*  
**by**(*simp add:rotate1-def split:list.split blast*)

**lemma** *distinct-rotate[simp]*: *distinct(rotate n xs) = distinct xs*  
**by** (*induct n*) (*simp-all add:rotate-def*)

**lemma** *rotate-map*: *rotate n (map f xs) = map f (rotate n xs)*  
**by**(*simp add:rotate-drop-take take-map drop-map*)

**lemma** *set-rotate1[simp]*: *set(rotate1 xs) = set xs*  
**by**(*simp add:rotate1-def split:list.split*)

**lemma** *set-rotate[simp]*: *set(rotate n xs) = set xs*  
**by** (*induct n*) (*simp-all add:rotate-def*)

**lemma** *rotate1-is-Nil-conv[simp]*: *(rotate1 xs = []) = (xs = [])*  
**by**(*simp add:rotate1-def split:list.split*)

**lemma** *rotate-is-Nil-conv[simp]*: *(rotate n xs = []) = (xs = [])*  
**by** (*induct n*) (*simp-all add:rotate-def*)

**lemma** *rotate-rev*:

```

  rotate n (rev xs) = rev (rotate (length xs - (n mod length xs)) xs)
apply (simp add: rotate-drop-take rev-drop rev-take)
apply (cases length xs = 0)
  apply simp
apply (cases n mod length xs = 0)
  apply simp
apply (simp add: rotate-drop-take rev-drop rev-take)
done

```

**lemma** *hd-rotate-conv-nth*:  $xs \neq [] \implies \text{hd}(\text{rotate } n \text{ } xs) = xs!(n \bmod \text{length } xs)$

```

apply (simp add: rotate-drop-take hd-append hd-drop-conv-nth hd-conv-nth)
apply (subgoal-tac length xs  $\neq$  0)
  prefer 2 apply simp
using mod-less-divisor[of length xs n] by arith

```

### 39.1.26 *sublist* — a generalization of *nth* to sets

**lemma** *sublist-empty* [simp]:  $\text{sublist } xs \ \{\} = []$

```

by (auto simp add: sublist-def)

```

**lemma** *sublist-nil* [simp]:  $\text{sublist } [] \ A = []$

```

by (auto simp add: sublist-def)

```

**lemma** *length-sublist*:

```

  length (sublist xs I) = card {i. i < length xs  $\wedge$  i : I}
by (simp add: sublist-def length-filter-conv-card cong: conj-cong)

```

**lemma** *sublist-shift-lemma-Suc*:

```

  map fst (filter (%p. P (Suc (snd p))) (zip xs is)) =
    map fst (filter (%p. P (snd p)) (zip xs (map Suc is)))
apply (induct xs arbitrary: is)
  apply simp
apply (case-tac is)
  apply simp
apply simp
done

```

**lemma** *sublist-shift-lemma*:

```

  map fst [p <- zip xs [i..i + length xs] . snd p : A] =
    map fst [p <- zip xs [0..length xs] . snd p + i : A]
by (induct xs rule: rev-induct) (simp-all add: add-commute)

```

**lemma** *sublist-append*:

```

  sublist (l @ l') A = sublist l A @ sublist l' {j. j + length l : A}
apply (unfold sublist-def)
apply (induct l' rule: rev-induct, simp)
apply (simp add: upt-add-eq-append[of 0] zip-append sublist-shift-lemma)
apply (simp add: add-commute)

```

done

**lemma** *sublist-Cons*:

*sublist* ( $x \# l$ )  $A = (if\ 0:A\ then\ [x]\ else\ [])\ @\ sublist\ l\ \{j.\ Suc\ j : A\}$

**apply** (*induct*  $l$  *rule*: *rev-induct*)

**apply** (*simp* *add*: *sublist-def*)

**apply** (*simp* *del*: *append-Cons* *add*: *append-Cons[symmetric]* *sublist-append*)

done

**lemma** *set-sublist*:  $set(sublist\ xs\ I) = \{xs!i \mid i.<size\ xs \wedge i \in I\}$

**apply**(*induct*  $xs$  *arbitrary*:  $I$ )

**apply**(*auto* *simp*: *sublist-Cons* *nth-Cons* *split*:*nat.split* *dest*!: *gr0-implies-Suc*)

done

**lemma** *set-sublist-subset*:  $set(sublist\ xs\ I) \subseteq set\ xs$

**by**(*auto* *simp* *add*:*set-sublist*)

**lemma** *notin-set-sublistI*[*simp*]:  $x \notin set\ xs \implies x \notin set(sublist\ xs\ I)$

**by**(*auto* *simp* *add*:*set-sublist*)

**lemma** *in-set-sublistD*:  $x \in set(sublist\ xs\ I) \implies x \in set\ xs$

**by**(*auto* *simp* *add*:*set-sublist*)

**lemma** *sublist-singleton* [*simp*]:  $sublist\ [x]\ A = (if\ 0 : A\ then\ [x]\ else\ [])$

**by** (*simp* *add*: *sublist-Cons*)

**lemma** *distinct-sublistI*[*simp*]:  $distinct\ xs \implies distinct(sublist\ xs\ I)$

**apply**(*induct*  $xs$  *arbitrary*:  $I$ )

**apply** *simp*

**apply**(*auto* *simp* *add*:*sublist-Cons*)

done

**lemma** *sublist-upt-eq-take* [*simp*]:  $sublist\ l\ \{.. $n\} = take\ n\ l$$

**apply** (*induct*  $l$  *rule*: *rev-induct*, *simp*)

**apply** (*simp* *split*: *nat-diff-split* *add*: *sublist-append*)

done

**lemma** *filter-in-sublist*:

$distinct\ xs \implies filter\ (\%x.\ x \in set(sublist\ xs\ s))\ xs = sublist\ xs\ s$

**proof** (*induct*  $xs$  *arbitrary*:  $s$ )

**case** *Nil* **thus** ?*case* **by** *simp*

**next**

**case** (*Cons*  $a\ xs$ )

**moreover** **hence** ! $x.\ x : set\ xs \longrightarrow x \neq a$  **by** *auto*

**ultimately** **show** ?*case* **by**(*simp* *add*: *sublist-Cons* *cong*:*filter-cong*)

qed

**39.1.27** *splice*

**lemma** *splice-Nil2* [*simp*, *code*]:

*splice xs [] = xs*

**by** (*cases xs*) *simp-all*

**lemma** *splice-Cons-Cons* [*simp*, *code*]:

*splice (x#xs) (y#ys) = x # y # splice xs ys*

**by** *simp*

**declare** *splice.simps*(2) [*simp del*, *code del*]

**lemma** *length-splice*[*simp*]: *length (splice xs ys) = length xs + length ys*

**apply**(*induct xs arbitrary: ys*) **apply** *simp*

**apply**(*case-tac ys*)

**apply** *auto*

**done**

**39.1.28** *Infiniteness*

**lemma** *finite-maxlen*:

*finite (M::'a list set) ==> EX n. ALL s:M. size s < n*

**proof** (*induct rule: finite.induct*)

**case emptyI** **show** ?*case* **by** *simp*

**next**

**case** (*insertI M xs*)

**then obtain** *n* **where**  $\forall s \in M. \text{length } s < n$  **by** *blast*

**hence** *ALL s:insert xs M. size s < max n (size xs) + 1* **by** *auto*

**thus** ?*case* ..

**qed**

**lemma** *infinite-UNIV-listI*:  $\sim \text{finite}(\text{UNIV}::'a \text{ list set})$

**apply**(*rule notI*)

**apply**(*drule finite-maxlen*)

**apply** (*metis UNIV-I length-replicate less-not-refl*)

**done**

**39.2** *Sorting*

Currently it is not shown that *sort* returns a permutation of its input because the nicest proof is via multisets, which are not yet available. Alternatively one could define a function that counts the number of occurrences of an element in a list and use that instead of multisets to state the correctness property.

**context** *linorder*

**begin**

**lemma** *sorted-Cons*: *sorted (x#xs) = (sorted xs & (ALL y:set xs. x <= y))*

**apply**(*induct xs arbitrary: x*) **apply** *simp*



**by** *simp* (*blast intro: order-trans*)

**lemma** *sorted-append*:

*sorted* (*xs@ys*) = (*sorted xs* & *sorted ys* & ( $\forall x \in \text{set } xs. \forall y \in \text{set } ys. x \leq y$ ))  
**by** (*induct xs*) (*auto simp add:sorted-Cons*)

**lemma** *set-insort*: *set*(*insort x xs*) = *insert x* (*set xs*)

**by** (*induct xs*) *auto*

**lemma** *set-sort*[*simp*]: *set*(*sort xs*) = *set xs*

**by** (*induct xs*) (*simp-all add:set-insort*)

**lemma** *distinct-insort*: *distinct* (*insort x xs*) = ( $x \notin \text{set } xs \wedge \text{distinct } xs$ )

**by**(*induct xs*)(*auto simp:set-insort*)

**lemma** *distinct-sort*[*simp*]: *distinct* (*sort xs*) = *distinct xs*

**by**(*induct xs*)(*simp-all add:distinct-insort set-sort*)

**lemma** *sorted-insort*: *sorted* (*insort x xs*) = *sorted xs*

**apply** (*induct xs*)

**apply**(*auto simp:sorted-Cons set-insort*)

**done**

**theorem** *sorted-sort*[*simp*]: *sorted* (*sort xs*)

**by** (*induct xs*) (*auto simp:sorted-insort*)

**lemma** *insort-is-Cons*:  $\forall x \in \text{set } xs. a \leq x \implies \text{insort } a \text{ } xs = a \# xs$

**by** (*cases xs*) *auto*

**lemma** *sorted-remove1*: *sorted xs*  $\implies$  *sorted* (*remove1 a xs*)

**by** (*induct xs, auto simp add: sorted-Cons*)

**lemma** *insort-remove1*:  $\llbracket a \in \text{set } xs; \text{sorted } xs \rrbracket \implies \text{insort } a \text{ } (\text{remove1 } a \text{ } xs) = xs$

**by** (*induct xs, auto simp add: sorted-Cons insort-is-Cons*)

**lemma** *sorted-remdups*[*simp*]:

*sorted l*  $\implies$  *sorted* (*remdups l*)

**by** (*induct l*) (*auto simp: sorted-Cons*)

**lemma** *sorted-distinct-set-unique*:

**assumes** *sorted xs distinct xs sorted ys distinct ys set xs = set ys*

**shows** *xs = ys*

**proof** –

**from** *assms* **have** *1: length xs = length ys* **by** (*auto dest!: distinct-card*)

**from** *assms* **show** *?thesis*

**proof**(*induct rule:list-induct2[OF 1]*)

**case** *1* **show** *?case* **by** *simp*

**next**

**case** *2* **thus** *?case* **by** (*simp add:sorted-Cons*)

```

      (metis Diff-insert-absorb antisym insertE insert-iff)
    qed
  qed

lemma finite-sorted-distinct-unique:
  shows finite A  $\implies$  EX! xs. set xs = A & sorted xs & distinct xs
  apply (drule finite-distinct-list)
  apply clarify
  apply (rule-tac a=sort xs in ex1I)
  apply (auto simp: sorted-distinct-set-unique)
  done

lemma sorted-take:
  sorted xs  $\implies$  sorted (take n xs)
proof (induct xs arbitrary: n rule: sorted.induct)
  case 1 show ?case by simp
next
  case 2 show ?case by (cases n) simp-all
next
  case (3 x y xs)
  then have x  $\leq$  y by simp
  show ?case proof (cases n)
    case 0 then show ?thesis by simp
  next
    case (Suc m)
    with 3 have sorted (take m (y # xs)) by simp
    with Suc  $\langle x \leq y \rangle$  show ?thesis by (cases m) simp-all
  qed
qed

lemma sorted-drop:
  sorted xs  $\implies$  sorted (drop n xs)
proof (induct xs arbitrary: n rule: sorted.induct)
  case 1 show ?case by simp
next
  case 2 show ?case by (cases n) simp-all
next
  case 3 then show ?case by (cases n) simp-all
qed

end

lemma sorted-upt[simp]: sorted[i..

```

**39.2.1** *sorted-list-of-set*

This function maps (finite) linearly ordered sets to sorted lists. Warning: in most cases it is not a good idea to convert from sets to lists but one should convert in the other direction (via *set*).

**context** *linorder*  
**begin**

**definition**

*sorted-list-of-set* :: 'a set  $\Rightarrow$  'a list **where**  
[code del]: *sorted-list-of-set* A == THE xs. set xs = A & sorted xs & distinct xs

**lemma** *sorted-list-of-set*[simp]: finite A  $\implies$   
  set(*sorted-list-of-set* A) = A &  
  sorted(*sorted-list-of-set* A) & distinct(*sorted-list-of-set* A)  
**apply**(simp add:*sorted-list-of-set-def*)  
**apply**(rule the1I2)  
**apply**(simp-all add: finite-sorted-distinct-unique)  
**done**

**lemma** *sorted-list-of-empty*[simp]: *sorted-list-of-set* {} = []  
**unfolding** *sorted-list-of-set-def*  
**apply**(subst the-equality[of - []])  
**apply** simp-all  
**done**

**end**

**39.2.2** *upto*: the generic interval-list

**class** *finite-intvl-succ* = *linorder* +  
**fixes** *successor* :: 'a  $\Rightarrow$  'a  
**assumes** *finite-intvl*: finite{a..b}  
**and** *successor-incr*: a < *successor* a  
**and** *ord-discrete*:  $\neg(\exists x. a < x \ \& \ x < \text{successor } a)$

**context** *finite-intvl-succ*  
**begin**

**definition**

*upto* :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a list ((1[-./-])) **where**  
*upto* i j == *sorted-list-of-set* {i..j}

**lemma** *upto*[simp]: set[a..b] = {a..b} & sorted[a..b] & distinct[a..b]  
**by**(simp add:*upto-def* *finite-intvl*)

**lemma** *insert-intvl*:  $i \leq j \implies \text{insert } i \ \{\text{successor } i..j\} = \{i..j\}$   
**apply**(insert *successor-incr*[of i])  
**apply**(auto simp: *atLeastAtMost-def* *atLeast-def* *atMost-def*)

```

apply(metis ord-discrete less-le not-le)
done

```

```

lemma sorted-list-of-set-rec:  $i \leq j \implies$ 
  sorted-list-of-set  $\{i..j\} = i \#$  sorted-list-of-set  $\{\text{successor } i..j\}$ 
apply(simp add:sorted-list-of-set-def upto-def)
apply (rule the1-equality[OF finite-sorted-distinct-unique])
apply (simp add:finite-intvl)
apply(rule the1I2[OF finite-sorted-distinct-unique])
apply (simp add:finite-intvl)
apply (simp add: sorted-Cons insert-intvl Ball-def)
apply (metis successor-incr leD less-imp-le order-trans)
done

```

```

lemma sorted-list-of-set-rec2:  $i \leq j \implies$ 
  sorted-list-of-set  $\{i..\text{successor } j\} =$ 
  sorted-list-of-set  $\{i..j\} @ [\text{successor } j]$ 
apply(simp add:sorted-list-of-set-def upto-def)
apply (rule the1-equality[OF finite-sorted-distinct-unique])
apply (simp add:finite-intvl)
apply(rule the1I2[OF finite-sorted-distinct-unique])
apply (simp add:finite-intvl)
apply (simp add: sorted-append Ball-def expand-set-eq)
apply(rule conjI)
apply (metis eq-iff leD linear not-leE ord-discrete order-trans successor-incr)
apply (metis leD linear order-trans successor-incr)
done

```

```

lemma upto-rec[code]:  $[i..j] = (\text{if } i \leq j \text{ then } i \# [\text{successor } i..j] \text{ else } [])$ 
by(simp add: upto-def sorted-list-of-set-rec)

```

```

lemma upto-empty[simp]:  $j < i \implies [i..j] = []$ 
by(simp add: upto-rec)

```

```

lemma upto-rec2:  $i \leq j \implies [i..\text{successor } j] = [i..j] @ [\text{successor } j]$ 
by(simp add: upto-def sorted-list-of-set-rec2)

```

```

end

```

The integers are an instance of the above class:

```

instantiation int:: finite-intvl-succ
begin

```

```

definition
  successor-int-def: successor = (%i::int. i+1)

```

```

instance
by intro-classes (simp-all add: successor-int-def)

```

**end**

Now  $[i..j]$  is defined for integers.

**hide (open)** *const successor*

**lemma** *upto-rec2-int*:  $(i::int) \leq j \implies [i..j+1] = [i..j] @ [j+1]$   
**by**(rule *upto-rec2*[**where**  $'a = int$ , *simplified successor-int-def*])

### 39.2.3 *lists*: the list-forming operator over sets

**inductive-set**

*lists* ::  $'a \text{ set} \implies 'a \text{ list set}$

**for**  $A :: 'a \text{ set}$

**where**

*Nil* [*intro!*]:  $[] : \text{lists } A$

| *Cons* [*intro!*,*noatp*]:  $[a : A; l : \text{lists } A] \implies a \# l : \text{lists } A$

**inductive-cases** *listsE* [*elim!*,*noatp*]:  $x \# l : \text{lists } A$

**inductive-cases** *listspE* [*elim!*,*noatp*]:  $\text{listsp } A (x \# l)$

**lemma** *listsp-mono* [*mono*]:  $A \leq B \implies \text{listsp } A \leq \text{listsp } B$

**by** (rule *predicate1I*, erule *listsp.induct*, blast+)

**lemmas** *lists-mono* = *listsp-mono* [*to-set pred-subset-eq*]

**lemma** *listsp-infI*:

**assumes**  $l : \text{listsp } A \text{ } l$  **shows**  $\text{listsp } B \text{ } l \implies \text{listsp } (inf \ A \ B) \text{ } l$  **using**  $l$

**by** *induct blast+*

**lemmas** *lists-IntI* = *listsp-infI* [*to-set*]

**lemma** *listsp-inf-eq* [*simp*]:  $\text{listsp } (inf \ A \ B) = inf \ (\text{listsp } A) \ (\text{listsp } B)$

**proof** (rule *mono-inf* [**where**  $f = \text{listsp}$ , *THEN order-antisym*])

**show** *mono listsp* **by** (*simp add: mono-def listsp-mono*)

**show**  $inf \ (\text{listsp } A) \ (\text{listsp } B) \leq \text{listsp } (inf \ A \ B)$  **by** (*blast intro!: listsp-infI predicate1I*)

**qed**

**lemmas** *listsp-conj-eq* [*simp*] = *listsp-inf-eq* [*simplified inf-fun-eq inf-bool-eq*]

**lemmas** *lists-Int-eq* [*simp*] = *listsp-inf-eq* [*to-set pred-equals-eq*]

**lemma** *append-in-listsp-conv* [*iff*]:

$(\text{listsp } A \ (xs @ ys)) = (\text{listsp } A \ xs \wedge \text{listsp } A \ ys)$

**by** (*induct xs*) *auto*

**lemmas** *append-in-lists-conv* [*iff*] = *append-in-listsp-conv* [*to-set*]

**lemma** *in-listsp-conv-set*:  $(\text{listsp } A \ xs) = (\forall x \in \text{set } xs. A \ x)$

— eliminate *listsp* in favour of *set*  
**by** (*induct xs*) *auto*

**lemmas** *in-lists-conv-set* = *in-listsp-conv-set* [*to-set*]

**lemma** *in-listspD* [*dest!*,*noatp*]: *listsp A xs ==>  $\forall x \in \text{set } xs. A x$*   
**by** (*rule in-listsp-conv-set* [*THEN iffD1*])

**lemmas** *in-listsD* [*dest!*,*noatp*] = *in-listspD* [*to-set*]

**lemma** *in-listspI* [*intro!*,*noatp*]:  $\forall x \in \text{set } xs. A x ==> \text{listsp } A xs$   
**by** (*rule in-listsp-conv-set* [*THEN iffD2*])

**lemmas** *in-listsI* [*intro!*,*noatp*] = *in-listspI* [*to-set*]

**lemma** *lists-UNIV* [*simp*]: *lists UNIV = UNIV*  
**by** *auto*

### 39.2.4 Inductive definition for membership

**inductive** *ListMem* :: '*a*  $\Rightarrow$  '*a* list  $\Rightarrow$  bool  
**where**

*elem*: *ListMem* *x* (*x* # *xs*)  
| *insert*: *ListMem* *x* *xs*  $\Longrightarrow$  *ListMem* *x* (*y* # *xs*)

**lemma** *ListMem-iff*: (*ListMem* *x* *xs*) = (*x*  $\in$  *set* *xs*)  
**apply** (*rule iffI*)  
**apply** (*induct set*: *ListMem*)  
**apply** *auto*  
**apply** (*induct xs*)  
**apply** (*auto intro*: *ListMem.intros*)  
**done**

### 39.2.5 Lists as Cartesian products

*set-Cons* *A* *Xs*: the set of lists with head drawn from *A* and tail drawn from *Xs*.

**constdefs**

*set-Cons* :: '*a* set  $\Rightarrow$  '*a* list set  $\Rightarrow$  '*a* list set  
*set-Cons* *A* *XS* == {*z*.  $\exists x xs. z = x \# xs \ \& \ x \in A \ \& \ xs \in XS$ }  
**declare** *set-Cons-def* [*code del*]

**lemma** *set-Cons-sing-Nil* [*simp*]: *set-Cons* *A* {[]} = (%*x*. [*x*]) '*A*  
**by** (*auto simp add*: *set-Cons-def*)

Yields the set of lists, all of the same length as the argument and with elements drawn from the corresponding element of the argument.

**consts** *listset* :: '*a* set list  $\Rightarrow$  '*a* list set

**primrec**

$listset [] = \{\}\}$   
 $listset(A\#As) = set-Cons\ A\ (listset\ As)$

### 39.3 Relations on Lists

#### 39.3.1 Length Lexicographic Ordering

These orderings preserve well-foundedness: shorter lists precede longer lists. These ordering are not used in dictionaries.

**consts**  $lexn :: ('a * 'a) set \Rightarrow nat \Rightarrow ('a list * 'a list) set$   
 — The lexicographic ordering for lists of the specified length

**primrec**

$lexn\ r\ 0 = \{\}$   
 $lexn\ r\ (Suc\ n) =$   
 $(prod-fun\ (\%(x, xs). x\#xs)\ (\%(x, xs). x\#xs)\ ' (r\ < *lex* > lexn\ r\ n))\ Int$   
 $\{(xs, ys). length\ xs = Suc\ n \wedge length\ ys = Suc\ n\}$

**constdefs**

$lex :: ('a \times 'a) set \Rightarrow ('a list \times 'a list) set$   
 $lex\ r == \bigcup n. lexn\ r\ n$   
 — Holds only between lists of the same length

$lenlex :: ('a \times 'a) set \Rightarrow ('a list \times 'a list) set$   
 $lenlex\ r == inv-image\ (less-than\ < *lex* > lex\ r)\ (\%xs. (length\ xs, xs))$   
 — Compares lists by their length and then lexicographically

**declare**  $lex-def\ [code\ del]$

**lemma**  $wf-lexn$ :  $wf\ r \implies wf\ (lexn\ r\ n)$   
**apply**  $(induct\ n, simp, simp)$   
**apply**  $(rule\ wf-subset)$   
**prefer** 2 **apply**  $(rule\ Int-lower1)$   
**apply**  $(rule\ wf-prod-fun-image)$   
**prefer** 2 **apply**  $(rule\ inj-onI, auto)$   
**done**

**lemma**  $lexn-length$ :

$(xs, ys) : lexn\ r\ n \implies length\ xs = n \wedge length\ ys = n$   
**by**  $(induct\ n\ arbitrary: xs\ ys)\ auto$

**lemma**  $wf-lex\ [intro!]$ :  $wf\ r \implies wf\ (lex\ r)$   
**apply**  $(unfold\ lex-def)$   
**apply**  $(rule\ wf-UN)$   
**apply**  $(blast\ intro: wf-lexn, clarify)$   
**apply**  $(rename-tac\ m\ n)$   
**apply**  $(subgoal-tac\ m\ \neq\ n)$   
**prefer** 2 **apply**  $blast$

**apply** (*blast dest: lexn-length not-sym*)  
**done**

**lemma** *lexn-conv*:

*lexn r n =*  
 $\{(xs,ys). \text{length } xs = n \wedge \text{length } ys = n \wedge$   
 $(\exists xys\ x\ y\ xs'\ ys'. xs = xys @ x \# xs' \wedge ys = xys @ y \# ys' \wedge (x, y):r)\}$   
**apply** (*induct n, simp*)  
**apply** (*simp add: image-Collect lex-prod-def, safe, blast*)  
**apply** (*rule-tac x = ab # xys in exI, simp*)  
**apply** (*case-tac xys, simp-all, blast*)  
**done**

**lemma** *lex-conv*:

*lex r =*  
 $\{(xs,ys). \text{length } xs = \text{length } ys \wedge$   
 $(\exists xys\ x\ y\ xs'\ ys'. xs = xys @ x \# xs' \wedge ys = xys @ y \# ys' \wedge (x, y):r)\}$   
**by** (*force simp add: lex-def lexn-conv*)

**lemma** *wf-lenlex [intro!]*: *wf r ==> wf (lenlex r)*  
**by** (*unfold lenlex-def blast*)

**lemma** *lenlex-conv*:

*lenlex r =*  $\{(xs,ys). \text{length } xs < \text{length } ys \mid$   
 $\text{length } xs = \text{length } ys \wedge (xs, ys) : \text{lex } r\}$   
**by** (*simp add: lenlex-def Id-on-def lex-prod-def inv-image-def*)

**lemma** *Nil-notin-lex [iff]*:  $([], ys) \notin \text{lex } r$   
**by** (*simp add: lex-conv*)

**lemma** *Nil2-notin-lex [iff]*:  $(xs, []) \notin \text{lex } r$   
**by** (*simp add: lex-conv*)

**lemma** *Cons-in-lex [simp]*:

$((x \# xs, y \# ys) : \text{lex } r) =$   
 $((x, y) : r \wedge \text{length } xs = \text{length } ys \mid x = y \wedge (xs, ys) : \text{lex } r)$   
**apply** (*simp add: lex-conv*)  
**apply** (*rule iffI*)  
**prefer 2 apply** (*blast intro: Cons-eq-appendI, clarify*)  
**apply** (*case-tac xys, simp, simp*)  
**apply** *blast*  
**done**

### 39.3.2 Lexicographic Ordering

Classical lexicographic ordering on lists, ie. "a"  $\leq$  "ab"  $\leq$  "b". This ordering does *not* preserve well-foundedness. Author: N. Voelker, March 2005.

**constdefs**

*lexord* ::  $('a * 'a)\text{set} \Rightarrow ('a\ \text{list} * 'a\ \text{list})\ \text{set}$



$\text{lexord } r == \{(x,y). \exists a v. y = x @ a \# v \vee$   
 $(\exists u a b v w. (a,b) \in r \wedge x = u @ (a \# v) \wedge y = u @ (b \# w))\}$   
**declare** *lexord-def* [*code del*]

**lemma** *lexord-Nil-left[simp]*:  $([],y) \in \text{lexord } r = (\exists a x. y = a \# x)$   
**by** (*unfold lexord-def, induct-tac y, auto*)

**lemma** *lexord-Nil-right[simp]*:  $(x,[]) \notin \text{lexord } r$   
**by** (*unfold lexord-def, induct-tac x, auto*)

**lemma** *lexord-cons-cons[simp]*:  
 $((a \# x, b \# y) \in \text{lexord } r) = ((a,b) \in r \mid (a = b \ \& \ (x,y) \in \text{lexord } r))$   
**apply** (*unfold lexord-def, safe, simp-all*)  
**apply** (*case-tac u, simp, simp*)  
**apply** (*case-tac u, simp, clarsimp, blast, blast, clarsimp*)  
**apply** (*erule-tac x=b \# u in allE*)  
**by** *force*

**lemmas** *lexord-simps* = *lexord-Nil-left lexord-Nil-right lexord-cons-cons*

**lemma** *lexord-append-rightI*:  $\exists b z. y = b \# z \implies (x, x @ y) \in \text{lexord } r$   
**by** (*induct-tac x, auto*)

**lemma** *lexord-append-left-rightI*:  
 $(a,b) \in r \implies (u @ a \# x, u @ b \# y) \in \text{lexord } r$   
**by** (*induct-tac u, auto*)

**lemma** *lexord-append-leftI*:  $(u,v) \in \text{lexord } r \implies (x @ u, x @ v) \in \text{lexord } r$   
**by** (*induct x, auto*)

**lemma** *lexord-append-leftD*:  
 $\llbracket (x @ u, x @ v) \in \text{lexord } r; (! a. (a,a) \notin r) \rrbracket \implies (u,v) \in \text{lexord } r$   
**by** (*erule rev-mp, induct-tac x, auto*)

**lemma** *lexord-take-index-conv*:  
 $((x,y) : \text{lexord } r) =$   
 $((\text{length } x < \text{length } y \wedge \text{take } (\text{length } x) y = x) \vee$   
 $(\exists i. i < \min(\text{length } x)(\text{length } y) \ \& \ \text{take } i x = \text{take } i y \ \& \ (x!i,y!i) \in r))$   
**apply** (*unfold lexord-def Let-def, clarsimp*)  
**apply** (*rule-tac f = (% a b. a \vee b) in arg-cong2*)  
**apply** *auto*  
**apply** (*rule-tac x=hd (drop (length x) y) in exI*)  
**apply** (*rule-tac x=tl (drop (length x) y) in exI*)  
**apply** (*erule subst, simp add: min-def*)  
**apply** (*rule-tac x=length u in exI, simp*)  
**apply** (*rule-tac x=take i x in exI*)  
**apply** (*rule-tac x=x ! i in exI*)  
**apply** (*rule-tac x=y ! i in exI, safe*)  
**apply** (*rule-tac x=drop (Suc i) x in exI*)

```

apply (drule sym, simp add: drop-Suc-conv-tl)
apply (rule-tac x=drop (Suc i) y in exI)
by (simp add: drop-Suc-conv-tl)

```

— lexord is extension of partial ordering List.lex

```

lemma lexord-lex: (x,y) ∈ lex r = ((x,y) ∈ lexord r ∧ length x = length y)
apply (rule-tac x = y in spec)
apply (induct-tac x, clarsimp)
by (clarify, case-tac x, simp, force)

```

```

lemma lexord-irreflexive: (! x. (x,x) ∉ r) ⇒ (y,y) ∉ lexord r
by (induct y, auto)

```

**lemma** lexord-trans:

```

  [| (x, y) ∈ lexord r; (y, z) ∈ lexord r; trans r |] ⇒ (x, z) ∈ lexord r
apply (erule rev-mp)+
apply (rule-tac x = x in spec)
apply (rule-tac x = z in spec)
apply (induct-tac y, simp, clarify)
apply (case-tac xa, erule ssubst)
apply (erule allE, erule allE) — avoid simp recursion
apply (case-tac x, simp, simp)
apply (case-tac x, erule allE, erule allE, simp)
apply (erule-tac x = listb in allE)
apply (erule-tac x = lista in allE, simp)
apply (unfold trans-def)
by blast

```

```

lemma lexord-transI: trans r ⇒ trans (lexord r)
by (rule transI, drule lexord-trans, blast)

```

```

lemma lexord-linear: (! a b. (a,b) ∈ r | a = b | (b,a) ∈ r) ⇒ (x,y) : lexord r | x
= y | (y,x) : lexord r
apply (rule-tac x = y in spec)
apply (induct-tac x, rule allI)
apply (case-tac x, simp, simp)
apply (rule allI, case-tac x, simp, simp)
by blast

```

### 39.4 Lexicographic combination of measure functions

These are useful for termination proofs

**definition**

```

  measures fs = inv-image (lex less-than) (%a. map (%f. f a) fs)

```

```

lemma wf-measures[recdef-wf, simp]: wf (measures fs)
unfolding measures-def
by blast

```

```

lemma in-measures[simp]:
   $(x, y) \in \text{measures } [] = \text{False}$ 
   $(x, y) \in \text{measures } (f \# fs)$ 
     $= (f x < f y \vee (f x = f y \wedge (x, y) \in \text{measures } fs))$ 
unfolding measures-def
by auto

```

```

lemma measures-less:  $f x < f y \implies (x, y) \in \text{measures } (f \# fs)$ 
by simp

```

```

lemma measures-lesseq:  $f x \leq f y \implies (x, y) \in \text{measures } fs \implies (x, y) \in \text{measures } (f \# fs)$ 
by auto

```

### 39.4.1 Lifting a Relation on List Elements to the Lists

```

inductive-set
  listrel :: ('a * 'a) set => ('a list * 'a list) set
  for r :: ('a * 'a) set
where
  Nil:  $([], []) \in \text{listrel } r$ 
  | Cons:  $[(x, y) \in r; (xs, ys) \in \text{listrel } r] \implies (x \# xs, y \# ys) \in \text{listrel } r$ 

```

```

inductive-cases listrel-Nil1 [elim!]:  $([], xs) \in \text{listrel } r$ 
inductive-cases listrel-Nil2 [elim!]:  $(xs, []) \in \text{listrel } r$ 
inductive-cases listrel-Cons1 [elim!]:  $(y \# ys, xs) \in \text{listrel } r$ 
inductive-cases listrel-Cons2 [elim!]:  $(xs, y \# ys) \in \text{listrel } r$ 

```

```

lemma listrel-mono:  $r \subseteq s \implies \text{listrel } r \subseteq \text{listrel } s$ 
apply clarify
apply (erule listrel.induct)
apply (blast intro: listrel.intros)+
done

```

```

lemma listrel-subset:  $r \subseteq A \times A \implies \text{listrel } r \subseteq \text{lists } A \times \text{lists } A$ 
apply clarify
apply (erule listrel.induct, auto)
done

```

```

lemma listrel-refl-on:  $\text{refl-on } A \ r \implies \text{refl-on } (\text{lists } A) \ (\text{listrel } r)$ 
apply (simp add: refl-on-def listrel-subset Ball-def)
apply (rule allI)
apply (induct-tac x)
apply (auto intro: listrel.intros)
done

```

```

lemma listrel-sym:  $\text{sym } r \implies \text{sym } (\text{listrel } r)$ 
apply (auto simp add: sym-def)

```

```

apply (erule listrel.induct)
apply (blast intro: listrel.intros)+
done

```

```

lemma listrel-trans: trans r  $\implies$  trans (listrel r)
apply (simp add: trans-def)
apply (intro allI)
apply (rule impI)
apply (erule listrel.induct)
apply (blast intro: listrel.intros)+
done

```

```

theorem equiv-listrel: equiv A r  $\implies$  equiv (lists A) (listrel r)
by (simp add: equiv-def listrel-refl-on listrel-sym listrel-trans)

```

```

lemma listrel-Nil [simp]: listrel r “ {} = {}
by (blast intro: listrel.intros)

```

```

lemma listrel-Cons:
  listrel r “ {x#xs} = set-Cons (r “ {x}) (listrel r “ {xs})
by (auto simp add: set-Cons-def intro: listrel.intros)

```

## 39.5 Miscellany

### 39.5.1 Characters and strings

```

datatype nibble =
  Nibble0 | Nibble1 | Nibble2 | Nibble3 | Nibble4 | Nibble5 | Nibble6 | Nibble7
  | Nibble8 | Nibble9 | NibbleA | NibbleB | NibbleC | NibbleD | NibbleE | NibbleF

```

```

lemma UNIV-nibble:
  UNIV = {Nibble0, Nibble1, Nibble2, Nibble3, Nibble4, Nibble5, Nibble6, Nibble7,
    Nibble8, Nibble9, NibbleA, NibbleB, NibbleC, NibbleD, NibbleE, NibbleF} (is -
    = ?A)
proof (rule UNIV-eq-I)
  fix x show x  $\in$  ?A by (cases x) simp-all
qed

```

```

instance nibble :: finite
by default (simp add: UNIV-nibble)

```

```

datatype char = Char nibble nibble
  — Note: canonical order of character encoding coincides with standard term
  ordering

```

```

lemma UNIV-char:
  UNIV = image (split Char) (UNIV  $\times$  UNIV)
proof (rule UNIV-eq-I)
  fix x show x  $\in$  image (split Char) (UNIV  $\times$  UNIV) by (cases x) auto
qed

```

```

instance char :: finite
  by default (simp add: UNIV-char)

lemma size-char [code, simp]:
  size (c::char) = 0 by (cases c) simp

lemma char-size [code, simp]:
  char-size (c::char) = 0 by (cases c) simp

primrec nibble-pair-of-char :: char  $\Rightarrow$  nibble  $\times$  nibble where
  nibble-pair-of-char (Char n m) = (n, m)

declare nibble-pair-of-char.simps [code del]

setup ⟨⟨
  let
    val nibbles = map (Thm.ctrm-of @{theory} o HOLogic.mk-nibble) (0 upto 15);
    val thms = map-product
      (fn n => fn m => Drule.instantiate' [] [SOME n, SOME m] @{thm nibble-pair-of-char.simps})
      nibbles nibbles;
  in
    PureThy.note-thmss Thm.lemmaK [((Binding.name nibble-pair-of-char.simps,
    []), [(thms, [])])]
    #-> (fn [(-, thms)] => fold-rev Code.add-eqn thms)
  end
  ⟩⟩

lemma char-case-nibble-pair [code, code inline]:
  char-case f = split f o nibble-pair-of-char
  by (simp add: expand-fun-eq split: char.split)

lemma char-rec-nibble-pair [code, code inline]:
  char-rec f = split f o nibble-pair-of-char
  unfolding char-case-nibble-pair [symmetric]
  by (simp add: expand-fun-eq split: char.split)

types string = char list

syntax
  -Char :: xstr => char    (CHR -)
  -String :: xstr => string  (-)

setup StringSyntax.setup

```

### 39.6 Size function

```

lemma [measure-function]: is-measure f  $\implies$  is-measure (list-size f)
by (rule is-measure-trivial)

```

**lemma** *[measure-function]: is-measure f  $\implies$  is-measure (option-size f)*  
**by** (rule *is-measure-trivial*)

**lemma** *list-size-estimation[termination-simp]:*  
 $x \in \text{set } xs \implies y < f\ x \implies y < \text{list-size } f\ xs$   
**by** (induct *xs*) auto

**lemma** *list-size-estimation'[termination-simp]:*  
 $x \in \text{set } xs \implies y \leq f\ x \implies y \leq \text{list-size } f\ xs$   
**by** (induct *xs*) auto

**lemma** *list-size-map[simp]: list-size f (map g xs) = list-size (f o g) xs*  
**by** (induct *xs*) auto

**lemma** *list-size-pointwise[termination-simp]:*  
 $(\bigwedge x. x \in \text{set } xs \implies f\ x < g\ x) \implies \text{list-size } f\ xs \leq \text{list-size } g\ xs$   
**by** (induct *xs*) force+

## 39.7 Code generator

### 39.7.1 Setup

**types-code**

*list* (- list)

**attach** (term-of)  $\ll$   
*fun* term-of-list *f T* = *HOLogic.mk-list T o map f*;  
 $\gg$

**attach** (test)  $\ll$   
*fun* gen-list' *aG aT i j* = *frequency*  
 [(*i*, *fn* () =>  
   *let*  
     *val* (*x*, *t*) = *aG j*;  
     *val* (*xs*, *ts*) = *gen-list' aG aT (i-1) j*  
     *in* (*x* :: *xs*, *fn* () => *HOLogic.cons-const aT \$ t () \$ ts ()*) end),  
 (1, *fn* () => ([], *fn* () => *HOLogic.nil-const aT*))] ()  
*and* gen-list *aG aT i* = *gen-list' aG aT i i*;  
 $\gg$

*char* (string)

**attach** (term-of)  $\ll$   
*val* term-of-char = *HOLogic.mk-char o ord*;  
 $\gg$   
**attach** (test)  $\ll$   
*fun* gen-char *i* =  
   *let* *val j* = *random-range (ord a) (Int.min (ord a + i, ord z))*  
   *in* (*chr j*, *fn* () => *HOLogic.mk-char j*) end;  
 $\gg$

**consts-code** *Cons* ((- ::/ -))

**code-type** *list*  
 (*SML* - *list*)  
 (*OCaml* - *list*)  
 (*Haskell* ![-])

**code-reserved** *SML*  
*list*

**code-reserved** *OCaml*  
*list*

**code-const** *Nil*  
 (*SML* [])  
 (*OCaml* [])  
 (*Haskell* [])

**ML** ⟨⟨  
*local*

*open Basic-Code-Thingol;*

```
fun implode-list naming t = case pairself
  (Code-Thingol.lookup-const naming) (@{const-name Nil}, @){const-name Cons})
  of (SOME nil', SOME cons') => let
    fun dest-cons (IConst (c, -) '$ t1 '$ t2) =
      if c = cons'
      then SOME (t1, t2)
      else NONE
    | dest-cons - = NONE;
    val (ts, t') = Code-Thingol.unfoldr dest-cons t;
  in case t'
    of IConst (c, -) => if c = nil' then SOME ts else NONE
    | - => NONE
  end
  | - => NONE
```

```
fun decode-char naming (IConst (c1, -), IConst (c2, -)) = (case map-filter
  (Code-Thingol.lookup-const naming)[@{const-name Nibble0}, @){const-name Nib-
ble1},
  @){const-name Nibble2}, @){const-name Nibble3},
  @){const-name Nibble4}, @){const-name Nibble5},
  @){const-name Nibble6}, @){const-name Nibble7},
  @){const-name Nibble8}, @){const-name Nibble9},
  @){const-name NibbleA}, @){const-name NibbleB},
  @){const-name NibbleC}, @){const-name NibbleD},
  @){const-name NibbleE}, @){const-name NibbleF}]
  of nibbles' as [-, -, -, -, -, -, -, -, -, -, -, -, -, -] => let
    fun idx c = find-index (curry (op =) c) nibbles';
    fun decode ~1 - = NONE
```

```

      | decode - ~ 1 = NONE
      | decode n m = SOME (chr (n * 16 + m));
    in decode (idx c1) (idx c2) end
  | - => NONE)
| decode-char - = NONE

fun implode-string naming mk-char mk-string ts = case
  Code-Thingol.lookup-const naming @{const-name Char}
of SOME char' => let
  fun implode-char (IConst (c, -) '$ t1 '$ t2) =
    if c = char' then decode-char naming (t1, t2) else NONE
  | implode-char - = NONE;
  val ts' = map implode-char ts;
  in if forall is-some ts'
    then (SOME o Code-Printer.str o mk-string o implode o map-filter I) ts'
    else NONE
  end
| - => NONE;

fun default-list (target-fxy, target-cons) pr fxy t1 t2 =
  Code-Printer.brackify-infix (target-fxy, Code-Printer.R) fxy [
    pr (Code-Printer.INFX (target-fxy, Code-Printer.X)) t1,
    Code-Printer.str target-cons,
    pr (Code-Printer.INFX (target-fxy, Code-Printer.R)) t2
  ];

fun pretty-list literals =
  let
    val mk-list = Code-Printer.literal-list literals;
    fun pretty pr naming thm vars fxy [(t1, -), (t2, -)] =
      case Option.map (cons t1) (implode-list naming t2)
      of SOME ts => mk-list (map (pr vars Code-Printer.NOBR) ts)
      | NONE => default-list (Code-Printer.infix-cons literals) (pr vars) fxy t1
  t2;
  in (2, pretty) end;

fun pretty-list-string literals =
  let
    val mk-list = Code-Printer.literal-list literals;
    val mk-char = Code-Printer.literal-char literals;
    val mk-string = Code-Printer.literal-string literals;
    fun pretty pr naming thm vars fxy [(t1, -), (t2, -)] =
      case Option.map (cons t1) (implode-list naming t2)
      of SOME ts => (case implode-string naming mk-char mk-string ts
        of SOME p => p
        | NONE => mk-list (map (pr vars Code-Printer.NOBR) ts))
      | NONE => default-list (Code-Printer.infix-cons literals) (pr vars) fxy t1
  t2;
  in (2, pretty) end;

```



```

fun pretty-char literals =
  let
    val mk-char = Code-Printer.literal-char literals;
    fun pretty - naming thm - - [(t1, -), (t2, -)] =
      case decode-char naming (t1, t2)
      of SOME c => (Code-Printer.str o mk-char) c
       | NONE => Code-Printer.nerror thm Illegal character expression;
  in (2, pretty) end;

fun pretty-message literals =
  let
    val mk-char = Code-Printer.literal-char literals;
    val mk-string = Code-Printer.literal-string literals;
    fun pretty - naming thm - - [(t, -)] =
      case implode-list naming t
      of SOME ts => (case implode-string naming mk-char mk-string ts
                      of SOME p => p
                       | NONE => Code-Printer.nerror thm Illegal message expression)
       | NONE => Code-Printer.nerror thm Illegal message expression;
  in (1, pretty) end;

in

fun add-literal-list target thy =
  let
    val pr = pretty-list (Code-Target.the-literals thy target);
  in
    thy
    |> Code-Target.add-syntax-const target @{const-name Cons} (SOME pr)
  end;

fun add-literal-list-string target thy =
  let
    val pr = pretty-list-string (Code-Target.the-literals thy target);
  in
    thy
    |> Code-Target.add-syntax-const target @{const-name Cons} (SOME pr)
  end;

fun add-literal-char target thy =
  let
    val pr = pretty-char (Code-Target.the-literals thy target);
  in
    thy
    |> Code-Target.add-syntax-const target @{const-name Char} (SOME pr)
  end;

fun add-literal-message str target thy =

```

```

    let
      val pr = pretty-message (Code-Target.the-literals thy target);
    in
      thy
      |> Code-Target.add-syntax-const target str (SOME pr)
    end;

end;
>>

setup <<
  fold (fn target => add-literal-list target) [SML, OCaml, Haskell]
>>

code-instance list :: eq
  (Haskell -)

code-const eq-class.eq :: 'a::eq list => 'a list => bool
  (Haskell infixl 4 ==)

setup <<
  let

    fun list-codegen thy defs dep thynome b t gr =
      let
        val ts = HLogic.dest-list t;
        val (-, gr') = Codegen.invoke-tycodegen thy defs dep thynome false
          (fastype-of t) gr;
        val (ps, gr'') = fold-map
          (Codegen.invoke-codegen thy defs dep thynome false) ts gr'
        in SOME (Pretty.list [ ] ps, gr'') end handle TERM - => NONE;

    fun char-codegen thy defs dep thynome b t gr =
      let
        val i = HLogic.dest-char t;
        val (-, gr') = Codegen.invoke-tycodegen thy defs dep thynome false
          (fastype-of t) gr;
        in SOME (Codegen.str (ML-Syntax.print-string (chr i)), gr')
        end handle TERM - => NONE;

      in
        Codegen.add-codegen list-codegen list-codegen
        #> Codegen.add-codegen char-codegen char-codegen
      end;
  >>

```

### 39.7.2 Generation of efficient code

**primrec**

*member* :: 'a  $\Rightarrow$  'a list  $\Rightarrow$  bool (**infixl** mem 55)

**where**

*x mem* []  $\longleftrightarrow$  False

| *x mem* (y#ys)  $\longleftrightarrow$  x = y  $\vee$  x mem ys

**primrec**

*null* :: 'a list  $\Rightarrow$  bool

**where**

*null* [] = True

| *null* (x#xs) = False

**primrec**

*list-inter* :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list

**where**

*list-inter* [] bs = []

| *list-inter* (a#as) bs =

(if a  $\in$  set bs then a # *list-inter* as bs else *list-inter* as bs)

**primrec**

*list-all* :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  ('a list  $\Rightarrow$  bool)

**where**

*list-all* P [] = True

| *list-all* P (x#xs) = (P x  $\wedge$  *list-all* P xs)

**primrec**

*list-ex* :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  bool

**where**

*list-ex* P [] = False

| *list-ex* P (x#xs) = (P x  $\vee$  *list-ex* P xs)

**primrec**

*filtermap* :: ('a  $\Rightarrow$  'b option)  $\Rightarrow$  'a list  $\Rightarrow$  'b list

**where**

*filtermap* f [] = []

| *filtermap* f (x#xs) =

(case f x of None  $\Rightarrow$  *filtermap* f xs

| Some y  $\Rightarrow$  y # *filtermap* f xs)

**primrec**

*map-filter* :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  'b list

**where**

*map-filter* f P [] = []

| *map-filter* f P (x#xs) =

(if P x then f x # *map-filter* f P xs else *map-filter* f P xs)

**primrec**

*length-unique* :: 'a list  $\Rightarrow$  nat

**where**

*length-unique* [] = 0

```
| length-unique (x#xs) =
  (if x ∈ set xs then length-unique xs else Suc (length-unique xs))
```

Only use *mem* for generating executable code. Otherwise use  $x \in \text{set } xs$  instead — it is much easier to reason about. The same is true for *list-all* and *list-ex*: write  $\forall x \in \text{set } xs$  and  $\exists x \in \text{set } xs$  instead because the HOL quantifiers are already known to the automatic provers. In fact, the declarations in the code subsection make sure that  $\in$ ,  $\forall x \in \text{set } xs$  and  $\exists x \in \text{set } xs$  are implemented efficiently.

Efficient emptiness check is implemented by *null*.

The functions *filtermap* and *map-filter* are just there to generate efficient code. Do not use them for modelling and proving.

```
lemma rev-foldl-cons [code]:
  rev xs = foldl (λxs x. x # xs) [] xs
proof (induct xs)
  case Nil then show ?case by simp
next
  case Cons
  {
    fix x xs ys
    have foldl (λxs x. x # xs) ys xs @ [x]
      = foldl (λxs x. x # xs) (ys @ [x]) xs
    by (induct xs arbitrary: ys) auto
  }
  note aux = this
  show ?case by (induct xs) (auto simp add: Cons aux)
qed
```

```
lemma mem-iff [code post]:
  x mem xs ⟷ x ∈ set xs
by (induct xs) auto
```

```
lemmas in-set-code [code unfold] = mem-iff [symmetric]
```

```
lemma empty-null [code inline]:
  xs = [] ⟷ null xs
by (cases xs) simp-all
```

```
lemmas null-empty [code post] =
  empty-null [symmetric]
```

```
lemma list-inter-conv:
  set (list-inter xs ys) = set xs ∩ set ys
by (induct xs) auto
```

```
lemma list-all-iff [code post]:
  list-all P xs ⟷ (∀ x ∈ set xs. P x)
by (induct xs) auto
```

**lemmas** *list-ball-code* [code unfold] = *list-all-iff* [symmetric]

**lemma** *list-all-append* [simp]:

$\text{list-all } P \ (xs \ @ \ ys) \longleftrightarrow (\text{list-all } P \ xs \wedge \text{list-all } P \ ys)$

**by** (induct xs) auto

**lemma** *list-all-rev* [simp]:

$\text{list-all } P \ (\text{rev } xs) \longleftrightarrow \text{list-all } P \ xs$

**by** (simp add: list-all-iff)

**lemma** *list-all-length*:

$\text{list-all } P \ xs \longleftrightarrow (\forall n < \text{length } xs. P \ (xs \ ! \ n))$

**unfolding** *list-all-iff* **by** (auto intro: all-nth-imp-all-set)

**lemma** *list-ex-iff* [code post]:

$\text{list-ex } P \ xs \longleftrightarrow (\exists x \in \text{set } xs. P \ x)$

**by** (induct xs) simp-all

**lemmas** *list-bex-code* [code unfold] =

*list-ex-iff* [symmetric]

**lemma** *list-ex-length*:

$\text{list-ex } P \ xs \longleftrightarrow (\exists n < \text{length } xs. P \ (xs \ ! \ n))$

**unfolding** *list-ex-iff* *set-conv-nth* **by** auto

**lemma** *filtermap-conv*:

$\text{filtermap } f \ xs = \text{map } (\lambda x. \text{the } (f \ x)) \ (\text{filter } (\lambda x. f \ x \neq \text{None}) \ xs)$

**by** (induct xs) (simp-all split: option.split)

**lemma** *map-filter-conv* [simp]:

$\text{map-filter } f \ P \ xs = \text{map } f \ (\text{filter } P \ xs)$

**by** (induct xs) auto

**lemma** *length-remdups-length-unique* [code inline]:

$\text{length } (\text{remdups } xs) = \text{length-unique } xs$

**by** (induct xs) simp-all

**hide** (open) *const length-unique*

Code for bounded quantification and summation over nats.

**lemma** *atMost-upto* [code unfold]:

$\{..n\} = \text{set } [0..<\text{Suc } n]$

**by** auto

**lemma** *atLeast-upt* [code unfold]:

$\{..<n\} = \text{set } [0..<n]$

**by** auto

**lemma** *greaterThanLessThan-upt* [code unfold]:  
 $\{n < .. < m\} = \text{set } [\text{Suc } n .. < m]$   
**by** *auto*

**lemma** *atLeastLessThan-upt* [code unfold]:  
 $\{n .. < m\} = \text{set } [n .. < m]$   
**by** *auto*

**lemma** *greaterThanAtMost-upt* [code unfold]:  
 $\{n < .. m\} = \text{set } [\text{Suc } n .. < \text{Suc } m]$   
**by** *auto*

**lemma** *atLeastAtMost-upt* [code unfold]:  
 $\{n .. m\} = \text{set } [n .. < \text{Suc } m]$   
**by** *auto*

**lemma** *all-nat-less-eq* [code unfold]:  
 $(\forall m < n :: \text{nat}. P\ m) \longleftrightarrow (\forall m \in \{0 .. < n\}. P\ m)$   
**by** *auto*

**lemma** *ex-nat-less-eq* [code unfold]:  
 $(\exists m < n :: \text{nat}. P\ m) \longleftrightarrow (\exists m \in \{0 .. < n\}. P\ m)$   
**by** *auto*

**lemma** *all-nat-less* [code unfold]:  
 $(\forall m \leq n :: \text{nat}. P\ m) \longleftrightarrow (\forall m \in \{0 .. n\}. P\ m)$   
**by** *auto*

**lemma** *ex-nat-less* [code unfold]:  
 $(\exists m \leq n :: \text{nat}. P\ m) \longleftrightarrow (\exists m \in \{0 .. n\}. P\ m)$   
**by** *auto*

**lemma** *setsum-set-distinct-conv-listsum*:  
 $\text{distinct } xs \implies \text{setsum } f\ (\text{set } xs) = \text{listsum } (\text{map } f\ xs)$   
**by** (*induct xs*) *simp-all*

**lemma** *setsum-set-upt-conv-listsum* [code unfold]:  
 $\text{setsum } f\ (\text{set } [m .. < n]) = \text{listsum } (\text{map } f\ [m .. < n])$   
**by** (*rule setsum-set-distinct-conv-listsum*) *simp*

Code for summation over ints.

**lemma** *greaterThanLessThan-upto* [code unfold]:  
 $\{i < .. < j :: \text{int}\} = \text{set } [i+1 .. j - 1]$   
**by** *auto*

**lemma** *atLeastLessThan-upto* [code unfold]:  
 $\{i .. < j :: \text{int}\} = \text{set } [i .. j - 1]$   
**by** *auto*

```

lemma greaterThanAtMost-upto [code unfold]:
  {i<..j::int} = set [i+1..j]
by auto

lemma atLeastAtMost-upto [code unfold]:
  {i..j::int} = set [i..j]
by auto

lemma setsum-set-upto-conv-listsum [code unfold]:
  setsum f (set [i..j::int]) = listsum (map f [i..j)
by (rule setsum-set-distinct-conv-listsum) simp

end

```

## 40 Code-Message: Monolithic strings (message strings) for code generation

```

theory Code-Message
imports Plain ~~/src/HOL/List
begin

```

### 40.1 Datatype of messages

```

datatype message-string = STR string

lemmas [code del] = message-string.recs message-string.cases

lemma [code]: size (s::message-string) = 0
  by (cases s) simp-all

lemma [code]: message-string-size (s::message-string) = 0
  by (cases s) simp-all

```

### 40.2 ML interface

```

ML ⟨⟨
  structure Message-String =
  struct

    fun mk s = @{term STR} $ HOLogic.mk-string s;

    end;
  ⟩⟩

```

### 40.3 Code serialization

```

code-type message-string
  (SML string)

```

```

(OCaml string)
(Haskell String)

setup <<
  fold (fn target => add-literal-message @{const-name STR} target)
    [SML, OCaml, Haskell]
>>

code-reserved SML string
code-reserved OCaml string

code-instance message-string :: eq
  (Haskell -)

code-const eq-class.eq :: message-string ⇒ message-string ⇒ bool
  (SML !((- : string) = -))
  (OCaml !((- : string) = -))
  (Haskell infixl 4 ==)

end

```

## 41 Typerep: Reflecting Pure types into HOL

```

theory Typerep
imports Plain List Code-Message
begin

datatype typerep = Typerep message-string typerep list

class typerep =
  fixes typerep :: 'a::{} itself ⇒ typerep
begin

definition typerep-of :: 'a ⇒ typerep where
  [simp]: typerep-of x = typerep TYPE('a)

end

setup <<
  let
    fun typerep-tr (*-TYPEREP*) [ty] =
      Lexicon.const @{const-syntax typerep} $ (Lexicon.const -constrain $ Lexi-
con.const TYPE $
        (Lexicon.const itself $ ty))
    | typerep-tr (*-TYPEREP*) ts = raise TERM (typerep-tr, ts);
    fun typerep-tr' show-sorts (*typerep*)
      (Type (fun, [Type (itself, [T]), -])) (Const (@{const-syntax TYPE}, -) ::
ts) =

```



```

      Term.list-comb (Lexicon.const -TYPEREP $ Syntax.term-of-typ show-sorts
T, ts)
    | typerep-tr' - T ts = raise Match;
in
  Sign.add-syntax-i
    [(-TYPEREP, SimpleSyntax.read-typ type => logic, Delimfix (1TYPEREP/(1'(-'))))]
  #> Sign.add-trfuns ([], [(-TYPEREP, typerep-tr)], [], [])
  #> Sign.add-trfunsT [(@{const-syntax typerep}, typerep-tr')]
end
>>

```

```

ML <<
structure Typerep =
struct

```

```

fun mk f (Type (tyco, tys)) =
  @{term Typerep} $ Message-String.mk tyco
  $ HOLogic.mk-list @{typ typerep} (map (mk f) tys)
| mk f (TFree v) =
  f v;

fun typerep ty =
  Const (@{const-name typerep}, Term.itselfT ty --> @{typ typerep})
  $ Logic.mk-type ty;

fun add-def tyco thy =
  let
    val sorts = replicate (Sign.arity-number thy tyco) @{sort typerep};
    val vs = Name.names Name.context 'a sorts;
    val ty = Type (tyco, map TFree vs);
    val lhs = Const (@{const-name typerep}, Term.itselfT ty --> @{typ typerep})
      $ Free (T, Term.itselfT ty);
    val rhs = mk (typerep o TFree) ty;
    val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs));
  in
    thy
    |> TheoryTarget.instantiation ([tyco], vs, @{sort typerep})
    |> '(fn lthy => Syntax.check-term lthy eq)
    |-> (fn eq => Specification.definition (NONE, (Attrib.empty-binding, eq)))
    |> snd
    |> Class.prove-instantiation-instance (K (Class.intro-classes-tac []))
    |> LocalTheory.exit-global
  end;

fun perhaps-add-def tyco thy =
  let
    val inst = can (Sorts.mg-domain (Sign.classes-of thy) tyco) @{sort typerep}
  in if inst then thy else add-def tyco thy end;

```

```

end;
>>

setup <<
  Typerep.add-def @{type-name prop}
  #> Typerep.add-def @{type-name fun}
  #> Typerep.add-def @{type-name itself}
  #> Typerep.add-def @{type-name bool}
  #> TypedefPackage.interpretation Typerep.perhaps-add-def
>>

lemma [code]:
  eq-class.eq (Typerep tyco1 tys1) (Typerep tyco2 tys2)  $\longleftrightarrow$  eq-class.eq tyco1 tyco2
   $\wedge$  list-all2 eq-class.eq tys1 tys2
  by (auto simp add: equals-eq [symmetric] list-all2-eq [symmetric])

code-type typerep
  (SML Term.typ)

code-const Typerep
  (SML Term.Type/ (-, -))

code-reserved SML Term

hide (open) const typerep Typerep

end

```

## 42 Code-Eval: Term evaluation using the generic code generator

```

theory Code-Eval
imports Plain Typerep
begin

```

### 42.1 Term representation

#### 42.1.1 Terms and class *term-of*

```

datatype term = dummy-term

```

```

definition Const :: message-string  $\Rightarrow$  typerep  $\Rightarrow$  term where
  Const - - = dummy-term

```

```

definition App :: term  $\Rightarrow$  term  $\Rightarrow$  term where
  App - - = dummy-term

```

```

code-datatype Const App

```

```

class term-of = typerep +
  fixes term-of :: 'a::{}  $\Rightarrow$  term

```

```

lemma term-of-anything: term-of  $x \equiv t$ 
  by (rule eq-reflection) (cases term-of  $x$ , cases  $t$ , simp)

```

```

ML <<
  structure Eval =
  struct

```

```

    fun mk-term f g (Const (c, ty)) =
      @{term Const} $ Message-String.mk c $ g ty
    | mk-term f g (t1 $ t2) =
      @{term App} $ mk-term f g t1 $ mk-term f g t2
    | mk-term f g (Free v) = f v
    | mk-term f g (Bound i) = Bound i
    | mk-term f g (Abs (v, -, t)) = Abs (v, @{typ term}, mk-term f g t);

    fun mk-term-of ty t = Const (@{const-name term-of}, ty --> @{typ term}) $ t;

  end;
>>

```

#### 42.1.2 term-of instances

```

setup <<
  let
    fun add-term-of-def ty vs tyco thy =
      let
        val lhs = Const (@{const-name term-of}, ty --> @{typ term})
          $ Free (x, ty);
        val rhs = @{term undefined :: term};
        val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs));
        fun triv-name-of t = (fst o dest-Free o fst o strip-comb o fst
          o HOLogic.dest-eq o HOLogic.dest-Trueprop) t ^ -triv;
      in
        thy
        |> TheoryTarget.instantiation ([tyco], vs, @{sort term-of})
        |> '(fn lthy => Syntax.check-term lthy eq)
        |> (fn eq => Specification.definition (NONE, ((Binding.name (triv-name-of
          eq), []), eq)))
        |> snd
        |> Class.prove-instantiation-instance (K (Class.intro-classes-tac []))
        |> LocalTheory.exit-global
      end;
    fun interpretator (tyco, (raw-vs, -)) thy =
      let
        val has-inst = can (Sorts.mg-domain (Sign.classes-of thy) tyco) @{sort

```

```

term-of};
  val constrain-sort =
    curry (Sorts.inter-sort (Sign.classes-of thy)) @ {sort term-of};
  val vs = (map o apsnd) constrain-sort raw-vs;
  val ty = Type (tyco, map TFree vs);
in
  thy
  |> Typerep.perhaps-add-def tyco
  |> not has-inst ? add-term-of-def ty vs tyco
end;
in
  Code.type-interpretation interpretator
end
>>

setup <<
let
  fun mk-term-of-eq ty vs tyco (c, tys) =
    let
      val t = list-comb (Const (c, tys ----> ty),
        map Free (Name.names Name.context a tys));
    in (map-aterms (fn Free (v, ty) => Var ((v, 0), ty) | t => t) t, Eval.mk-term
      (fn (v, ty) => Eval.mk-term-of ty (Var ((v, 0), ty)))
      (Typerep.mk (fn (v, sort) => Typerep.typerep (TFree (v, sort)))) t)
    end;
  fun prove-term-of-eq ty eq thy =
    let
      val cty = Thm.ctyp-of thy ty;
      val (arg, rhs) = pairself (Thm.cterm-of thy) eq;
      val thm = @ {thm term-of-anything}
        |> Drule.instantiate' [SOME cty] [SOME arg, SOME rhs]
        |> Thm.verifyT;
    in
      thy
      |> Code.add-eqn thm
    end;
  fun interpretator (tyco, (raw-vs, raw-cs)) thy =
    let
      val constrain-sort =
        curry (Sorts.inter-sort (Sign.classes-of thy)) @ {sort term-of};
      val vs = (map o apsnd) constrain-sort raw-vs;
      val cs = (map o apsnd o map o map-atyps)
        (fn TFree (v, sort) => TFree (v, constrain-sort sort)) raw-cs;
      val ty = Type (tyco, map TFree vs);
      val eqs = map (mk-term-of-eq ty vs tyco) cs;
      val const = AxCClass.param-of-inst thy (@ {const-name term-of}, tyco);
    in
      thy
      |> Code.del-eqns const
    end
  end
end

```

```

    |> fold (prove-term-of-eq ty) eqs
  end;
in
  Code.type-interpretation interpretator
end
>>

```

### 42.1.3 Code generator setup

**lemmas** [code del] = term.recs term.cases term.size

**lemma** [code, code del]: eq-class.eq (t1::term) t2  $\longleftrightarrow$  eq-class.eq t1 t2 ..

**lemma** [code, code del]: (term-of :: typerep  $\Rightarrow$  term) = term-of ..

**lemma** [code, code del]: (term-of :: term  $\Rightarrow$  term) = term-of ..

**lemma** [code, code del]: (term-of :: message-string  $\Rightarrow$  term) = term-of ..

**lemma** [code, code del]:

(Code-Eval.term-of :: 'a::{type, term-of} Predicate.pred  $\Rightarrow$  Code-Eval.term) = Code-Eval.term-of ..

**lemma** [code, code del]:

(Code-Eval.term-of :: 'a::{type, term-of} Predicate.seq  $\Rightarrow$  Code-Eval.term) = Code-Eval.term-of ..

**lemma** term-of-char [unfolded typerep-fun-def typerep-char-def typerep-nibble-def, code]: Code-Eval.term-of c =

```

  (let (n, m) = nibble-pair-of-char c
  in Code-Eval.App (Code-Eval.App (Code-Eval.Const (STR "Pair")) (TYPEREP(nibble
 $\Rightarrow$  nibble  $\Rightarrow$  char))))
  (Code-Eval.term-of n)) (Code-Eval.term-of m))
  by (subst term-of-anything) rule

```

**code-type** term

(SML Term.term)

**code-const** Const and App

(SML Term.Const/ (-, -) and Term.\$/ (-, -))

**code-const** term-of :: message-string  $\Rightarrow$  term

(SML Message'-String.mk)

## 42.2 Evaluation setup

**ML** <<

signature EVAL =

sig

val mk-term: ((string \* typ)  $\rightarrow$  term)  $\rightarrow$  (typ  $\rightarrow$  term)  $\rightarrow$  term  $\rightarrow$  term

val eval-ref: (unit  $\rightarrow$  term) option ref

val eval-term: theory  $\rightarrow$  term  $\rightarrow$  term

end;

structure Eval : EVAL =

```

struct

open Eval;

val eval-ref = ref (NONE : (unit -> term) option);

fun eval-term thy t =
  t
  |> Eval.mk-term-of (fastype-of t)
  |> (fn t => Code-ML.eval-term (Eval.eval-ref, eval-ref) thy t [])
  |> Code.postprocess-term thy;

end;
>>

setup <<
  Value.add-evaluator (code, Eval.eval-term o ProofContext.theory-of)
>>

42.2.1 Syntax

print-translation <<
let
  val term = Const (<TERM>, dummyT);
  fun tr1' [-, -] = term;
  fun tr2' [] = term;
in
  [(@{const-syntax Const}, tr1'),
   (@{const-syntax App}, tr1'),
   (@{const-syntax dummy-term}, tr2')]
end
>>

hide const dummy-term
hide (open) const Const App
hide (open) const term-of

end

```

## 43 Map: Maps

```

theory Map
imports List
begin

types ('a,'b) ~=> = 'a => 'b option (infixr 0)
translations (type) a ~=> b <= (type) a => b option

```

**syntax** (*xsymbols*)

$\sim=> :: [type, type] => type$  (**infixr**  $\rightarrow$  0)

**abbreviation**

*empty* :: 'a  $\sim=>$  'b **where**

*empty* == %x. None

**definition**

*map-comp* :: ('b  $\sim=>$  'c)  $\Rightarrow$  ('a  $\sim=>$  'b)  $\Rightarrow$  ('a  $\sim=>$  'c) (**infixl** *o'-m* 55)

**where**

*f o-m g* = ( $\lambda k.$  case *g k* of None  $\Rightarrow$  None | Some *v*  $\Rightarrow$  *f v*)

**notation** (*xsymbols*)

*map-comp* (**infixl**  $\circ_m$  55)

**definition**

*map-add* :: ('a  $\sim=>$  'b)  $\Rightarrow$  ('a  $\sim=>$  'b)  $\Rightarrow$  ('a  $\sim=>$  'b) (**infixl** ++ 100)

**where**

*m1* ++ *m2* = ( $\lambda x.$  case *m2 x* of None  $\Rightarrow$  *m1 x* | Some *y*  $\Rightarrow$  Some *y*)

**definition**

*restrict-map* :: ('a  $\sim=>$  'b)  $\Rightarrow$  'a set  $\Rightarrow$  ('a  $\sim=>$  'b) (**infixl** |' 110) **where**

*m* | 'A = ( $\lambda x.$  if *x* : A then *m x* else None)

**notation** (*latex output*)

*restrict-map* ( $-|$  [111,110] 110)

**definition**

*dom* :: ('a  $\sim=>$  'b)  $\Rightarrow$  'a set **where**

*dom m* = {*a*. *m a*  $\sim=$  None}

**definition**

*ran* :: ('a  $\sim=>$  'b)  $\Rightarrow$  'b set **where**

*ran m* = {*b*. EX *a*. *m a* = Some *b*}

**definition**

*map-le* :: ('a  $\sim=>$  'b)  $\Rightarrow$  ('a  $\sim=>$  'b)  $\Rightarrow$  bool (**infix**  $\subseteq_m$  50) **where**

(*m*<sub>1</sub>  $\subseteq_m$  *m*<sub>2</sub>) = ( $\forall a \in \text{dom } m_1. m_1 a = m_2 a$ )

**consts**

*map-of* :: ('a \* 'b) list  $\Rightarrow$  'a  $\sim=>$  'b

*map-upds* :: ('a  $\sim=>$  'b)  $\Rightarrow$  'a list  $\Rightarrow$  'b list  $\Rightarrow$  ('a  $\sim=>$  'b)

**nonterminals**

*maplets maplet*

**syntax**

*-maplet* :: ['a, 'a]  $\Rightarrow$  maplet

(- /|->/ -)

*-maplets* :: ['a, 'a]  $\Rightarrow$  maplet

(- /[[->]]/ -)

```

      :: maplet => maplets          (-)
-Maplets :: [maplet, maplets] => maplets (-,/ -)
-MapUpd  :: ['a ~=> 'b, maplets] => 'a ~=> 'b (-/'(-) [900,0]900)
-Map     :: maplets => 'a ~=> 'b      ((1[-]))

```

**syntax** (*xsymbols*)

```

-maplet  :: ['a, 'a] => maplet      (- /⊢/ -)
-maplets :: ['a, 'a] => maplet      (- /⊢]/ -)

```

**translations**

```

-MapUpd m (-Maplets xy ms) == -MapUpd (-MapUpd m xy) ms
-MapUpd m (-maplet x y)    == m(x:=Some y)
-MapUpd m (-maplets x y)   == map-upds m x y
-Map ms                    == -MapUpd (CONST empty) ms
-Map (-Maplets ms1 ms2)    <= -MapUpd (-Map ms1) ms2
-Maplets ms1 (-Maplets ms2 ms3) <= -Maplets (-Maplets ms1 ms2) ms3

```

**primrec**

```

map-of [] = empty
map-of (p#ps) = (map-of ps)(fst p |-> snd p)

```

**declare** *map-of.simps* [*code del*]

**lemma** *map-of-Cons-code* [*code*]:

```

map-of [] k = None
map-of ((l, v) # ps) k = (if l = k then Some v else map-of ps k)
by simp-all

```

**defs**

```

map-upds-def [code]: m(xs [|->] ys) == m ++ map-of (rev(zip xs ys))

```

### 43.1 empty

**lemma** *empty-upd-none* [*simp*]: *empty*(*x* := *None*) = *empty*

**by** (*rule ext*) *simp*

### 43.2 map-upd

**lemma** *map-upd-triv*: *t* *k* = *Some* *x* ==> *t*(*k*|->*x*) = *t*

**by** (*rule ext*) *simp*

**lemma** *map-upd-nonempty* [*simp*]: *t*(*k*|->*x*) ~ = *empty*

**proof**

```

  assume t(k ⊢ x) = empty
  then have (t(k ⊢ x)) k = None by simp
  then show False by simp

```

**qed**

**lemma** *map-upd-eqD1*:

```

  assumes m(a⊢x) = n(a⊢y)

```



```

  shows  $x = y$ 
proof -
  from prems have  $(m(a \mapsto x))\ a = (n(a \mapsto y))\ a$  by simp
  then show ?thesis by simp
qed

```

```

lemma map-upd-Some-unfold:
   $((m(a|->b))\ x = \text{Some } y) = (x = a \wedge b = y \vee x \neq a \wedge m\ x = \text{Some } y)$ 
by auto

```

```

lemma image-map-upd [simp]:  $x \notin A \implies m(x \mapsto y) \text{ ` } A = m \text{ ` } A$ 
by auto

```

```

lemma finite-range-updI:  $\text{finite } (\text{range } f) \implies \text{finite } (\text{range } (f(a|->b)))$ 
unfolding image-def
apply (simp (no-asm-use) add:full-SetCompr-eq)
apply (rule finite-subset)
prefer 2 apply assumption
apply (auto)
done

```

### 43.3 map-of

```

lemma map-of-eq-None-iff:
   $(\text{map-of } xys\ x = \text{None}) = (x \notin \text{fst ` } (\text{set } xys))$ 
by (induct xys) simp-all

```

```

lemma map-of-is-SomeD:  $\text{map-of } xys\ x = \text{Some } y \implies (x,y) \in \text{set } xys$ 
apply (induct xys)
  apply simp
apply (clarsimp split: if-splits)
done

```

```

lemma map-of-eq-Some-iff [simp]:
   $\text{distinct}(\text{map fst } xys) \implies (\text{map-of } xys\ x = \text{Some } y) = ((x,y) \in \text{set } xys)$ 
apply (induct xys)
  apply simp
apply (auto simp: map-of-eq-None-iff [symmetric])
done

```

```

lemma Some-eq-map-of-iff [simp]:
   $\text{distinct}(\text{map fst } xys) \implies (\text{Some } y = \text{map-of } xys\ x) = ((x,y) \in \text{set } xys)$ 
by (auto simp del: map-of-eq-Some-iff simp add: map-of-eq-Some-iff [symmetric])

```

```

lemma map-of-is-SomeI [simp]:  $\llbracket \text{distinct}(\text{map fst } xys); (x,y) \in \text{set } xys \rrbracket$ 
 $\implies \text{map-of } xys\ x = \text{Some } y$ 
apply (induct xys)
  apply simp
apply force

```

done

**lemma** *map-of-zip-is-None* [simp]:

$length\ xs = length\ ys \implies (map-of\ (zip\ xs\ ys)\ x = None) = (x \notin set\ xs)$   
**by** (induct rule: list-induct2) simp-all

**lemma** *map-of-zip-is-Some*:

**assumes**  $length\ xs = length\ ys$   
**shows**  $x \in set\ xs \longleftrightarrow (\exists y. map-of\ (zip\ xs\ ys)\ x = Some\ y)$   
**using** *assms* **by** (induct rule: list-induct2) simp-all

**lemma** *map-of-zip-upd*:

**fixes**  $x :: 'a$  **and**  $xs :: 'a\ list$  **and**  $ys\ zs :: 'b\ list$   
**assumes**  $length\ ys = length\ xs$   
**and**  $length\ zs = length\ xs$   
**and**  $x \notin set\ xs$   
**and**  $map-of\ (zip\ xs\ ys)(x \mapsto y) = map-of\ (zip\ xs\ zs)(x \mapsto z)$   
**shows**  $map-of\ (zip\ xs\ ys) = map-of\ (zip\ xs\ zs)$

**proof**

**fix**  $x' :: 'a$   
**show**  $map-of\ (zip\ xs\ ys)\ x' = map-of\ (zip\ xs\ zs)\ x'$   
**proof** (cases  $x = x'$ )  
**case** *True*  
**from** *assms* *True* *map-of-zip-is-None* [of  $xs\ ys\ x'$ ]  
**have**  $map-of\ (zip\ xs\ ys)\ x' = None$  **by** *simp*  
**moreover from** *assms* *True* *map-of-zip-is-None* [of  $xs\ zs\ x'$ ]  
**have**  $map-of\ (zip\ xs\ zs)\ x' = None$  **by** *simp*  
**ultimately show** *?thesis* **by** *simp*  
**next**  
**case** *False* **from** *assms*  
**have**  $(map-of\ (zip\ xs\ ys)(x \mapsto y))\ x' = (map-of\ (zip\ xs\ zs)(x \mapsto z))\ x'$  **by**  
*auto*  
**with** *False* **show** *?thesis* **by** *simp*  
**qed**  
**qed**

**lemma** *map-of-zip-inject*:

**assumes**  $length\ ys = length\ xs$   
**and**  $length\ zs = length\ xs$   
**and** *dist: distinct*  $xs$   
**and**  $map-of\ (zip\ xs\ ys) = map-of\ (zip\ xs\ zs)$   
**shows**  $ys = zs$   
**using** *assms*(1) *assms*(2)[*symmetric*] **using** *dist* *map-of* **proof** (induct  $ys\ xs\ zs$   
rule: list-induct3)  
**case** *Nil* **show** *?case* **by** *simp*  
**next**  
**case** (*Cons*  $y\ ys\ x\ xs\ z\ zs$ )  
**from**  $\langle map-of\ (zip\ (x\#\!xs)\ (y\#\!ys)) = map-of\ (zip\ (x\#\!xs)\ (z\#\!zs)) \rangle$   
**have**  $map-of\ (zip\ xs\ ys)(x \mapsto y) = map-of\ (zip\ xs\ zs)(x \mapsto z)$  **by** *simp*

**from** *Cons* **have**  $\text{length } ys = \text{length } xs$  **and**  $\text{length } zs = \text{length } xs$   
**and**  $x \notin \text{set } xs$  **by** *simp-all*  
**then have**  $\text{map-of } (\text{zip } xs \ ys) = \text{map-of } (\text{zip } xs \ zs)$  **using** *map-of* **by** (*rule*  
*map-of-zip-upd*)  
**with** *Cons.hyps*  $\langle \text{distinct } (x \ \# \ xs) \rangle$  **have**  $ys = zs$  **by** *simp*  
**moreover from** *map-of* **have**  $y = z$  **by** (*rule map-upd-eqD1*)  
**ultimately show**  $?case$  **by** *simp*  
**qed**

**lemma** *finite-range-map-of*:  $\text{finite } (\text{range } (\text{map-of } xys))$   
**apply** (*induct xys*)  
**apply** (*simp-all add: image-constant*)  
**apply** (*rule finite-subset*)  
**prefer 2 apply** *assumption*  
**apply** *auto*  
**done**

**lemma** *map-of-SomeD*:  $\text{map-of } xs \ k = \text{Some } y \implies (k, y) \in \text{set } xs$   
**by** (*induct xs*) (*simp, atomize (full), auto*)

**lemma** *map-of-mapk-SomeI*:  
 $\text{inj } f \implies \text{map-of } t \ k = \text{Some } x \implies$   
 $\text{map-of } (\text{map } (\text{split } (\%k. \text{Pair } (f \ k)))) \ t \ (f \ k) = \text{Some } x$   
**by** (*induct t*) (*auto simp add: inj-eq*)

**lemma** *weak-map-of-SomeI*:  $(k, x) : \text{set } l \implies \exists x. \text{map-of } l \ k = \text{Some } x$   
**by** (*induct l*) *auto*

**lemma** *map-of-filter-in*:  
 $\text{map-of } xs \ k = \text{Some } z \implies P \ k \ z \implies \text{map-of } (\text{filter } (\text{split } P) \ xs) \ k = \text{Some } z$   
**by** (*induct xs*) *auto*

**lemma** *map-of-map*:  $\text{map-of } (\text{map } (\% (a, b). (a, f \ b)) \ xs) \ x = \text{Option.map } f \ (\text{map-of } xs \ x)$   
**by** (*induct xs*) *auto*

#### 43.4 *Option.map* related

**lemma** *option-map-o-empty* [*simp*]:  $\text{Option.map } f \ o \ \text{empty} = \text{empty}$   
**by** (*rule ext*) *simp*

**lemma** *option-map-o-map-upd* [*simp*]:  
 $\text{Option.map } f \ o \ m(a|->b) = (\text{Option.map } f \ o \ m)(a|->f \ b)$   
**by** (*rule ext*) *simp*

#### 43.5 *map-comp* related

**lemma** *map-comp-empty* [*simp*]:  
 $m \circ_m \text{empty} = \text{empty}$   
 $\text{empty} \circ_m m = \text{empty}$

**by** (*auto simp add: map-comp-def intro: ext split: option.splits*)

**lemma** *map-comp-simps* [*simp*]:

$m2\ k = \text{None} \implies (m1\ \circ_m\ m2)\ k = \text{None}$

$m2\ k = \text{Some}\ k' \implies (m1\ \circ_m\ m2)\ k = m1\ k'$

**by** (*auto simp add: map-comp-def*)

**lemma** *map-comp-Some-iff*:

$((m1\ \circ_m\ m2)\ k = \text{Some}\ v) = (\exists k'.\ m2\ k = \text{Some}\ k' \wedge m1\ k' = \text{Some}\ v)$

**by** (*auto simp add: map-comp-def split: option.splits*)

**lemma** *map-comp-None-iff*:

$((m1\ \circ_m\ m2)\ k = \text{None}) = (m2\ k = \text{None} \vee (\exists k'.\ m2\ k = \text{Some}\ k' \wedge m1\ k' = \text{None}))$

**by** (*auto simp add: map-comp-def split: option.splits*)

### 43.6 ++

**lemma** *map-add-empty*[*simp*]:  $m\ ++\ \text{empty} = m$

**by**(*simp add: map-add-def*)

**lemma** *empty-map-add*[*simp*]:  $\text{empty}\ ++\ m = m$

**by** (*rule ext*) (*simp add: map-add-def split: option.split*)

**lemma** *map-add-assoc*[*simp*]:  $m1\ ++\ (m2\ ++\ m3) = (m1\ ++\ m2)\ ++\ m3$

**by** (*rule ext*) (*simp add: map-add-def split: option.split*)

**lemma** *map-add-Some-iff*:

$((m\ ++\ n)\ k = \text{Some}\ x) = (n\ k = \text{Some}\ x \mid n\ k = \text{None} \ \&\ m\ k = \text{Some}\ x)$

**by** (*simp add: map-add-def split: option.split*)

**lemma** *map-add-SomeD* [*dest!*]:

$(m\ ++\ n)\ k = \text{Some}\ x \implies n\ k = \text{Some}\ x \vee n\ k = \text{None} \wedge m\ k = \text{Some}\ x$

**by** (*rule map-add-Some-iff* [*THEN iffD1*])

**lemma** *map-add-find-right* [*simp*]:  $!!xx.\ n\ k = \text{Some}\ xx \implies (m\ ++\ n)\ k = \text{Some}\ xx$

**by** (*subst map-add-Some-iff*) *fast*

**lemma** *map-add-None* [*iff*]:  $((m\ ++\ n)\ k = \text{None}) = (n\ k = \text{None} \ \&\ m\ k = \text{None})$

**by** (*simp add: map-add-def split: option.split*)

**lemma** *map-add-upd*[*simp*]:  $f\ ++\ g(x|->y) = (f\ ++\ g)(x|->y)$

**by** (*rule ext*) (*simp add: map-add-def*)

**lemma** *map-add-upds*[*simp*]:  $m1\ ++\ (m2(xs[\mapsto]ys)) = (m1\ ++\ m2)(xs[\mapsto]ys)$

**by** (*simp add: map-upds-def*)

```

lemma map-of-append[simp]: map-of (xs @ ys) = map-of ys ++ map-of xs
unfolding map-add-def
apply (induct xs)
  apply simp
apply (rule ext)
apply (simp split add: option.split)
done

```

```

lemma finite-range-map-of-map-add:
  finite (range f) ==> finite (range (f ++ map-of l))
apply (induct l)
  apply (auto simp del: fun-upd-apply)
apply (erule finite-range-updI)
done

```

```

lemma inj-on-map-add-dom [iff]:
  inj-on (m ++ m') (dom m') = inj-on m' (dom m')
by (fastsimp simp: map-add-def dom-def inj-on-def split: option.splits)

```

### 43.7 restrict-map

```

lemma restrict-map-to-empty [simp]: m|'{} = empty
by (simp add: restrict-map-def)

```

```

lemma restrict-map-empty [simp]: empty|'D = empty
by (simp add: restrict-map-def)

```

```

lemma restrict-in [simp]: x ∈ A ==> (m|'A) x = m x
by (simp add: restrict-map-def)

```

```

lemma restrict-out [simp]: x ∉ A ==> (m|'A) x = None
by (simp add: restrict-map-def)

```

```

lemma ran-restrictD: y ∈ ran (m|'A) ==> ∃ x ∈ A. m x = Some y
by (auto simp: restrict-map-def ran-def split: split-if-asm)

```

```

lemma dom-restrict [simp]: dom (m|'A) = dom m ∩ A
by (auto simp: restrict-map-def dom-def split: split-if-asm)

```

```

lemma restrict-upd-same [simp]: m(x ↦ y)|'(-{x}) = m|'(-{x})
by (rule ext) (auto simp: restrict-map-def)

```

```

lemma restrict-restrict [simp]: m|'A|'B = m|'A ∩ B
by (rule ext) (auto simp: restrict-map-def)

```

```

lemma restrict-fun-upd [simp]:
  m(x := y)|'D = (if x ∈ D then (m|'(D - {x}))(x := y) else m|'D)
by (simp add: restrict-map-def expand-fun-eq)

```

**lemma** *fun-upd-None-restrict* [simp]:  
 $(m|'D)(x := \text{None}) = (\text{if } x:D \text{ then } m|'(D - \{x\}) \text{ else } m|'D)$   
**by** (simp add: restrict-map-def expand-fun-eq)

**lemma** *fun-upd-restrict*:  $(m|'D)(x := y) = (m|'(D - \{x\}))(x := y)$   
**by** (simp add: restrict-map-def expand-fun-eq)

**lemma** *fun-upd-restrict-conv* [simp]:  
 $x \in D \implies (m|'D)(x := y) = (m|'(D - \{x\}))(x := y)$   
**by** (simp add: restrict-map-def expand-fun-eq)

### 43.8 map-upds

**lemma** *map-upds-Nil1* [simp]:  $m([] \llbracket - \> \rrbracket bs) = m$   
**by** (simp add: map-upds-def)

**lemma** *map-upds-Nil2* [simp]:  $m(as \llbracket - \> \rrbracket []) = m$   
**by** (simp add: map-upds-def)

**lemma** *map-upds-Cons* [simp]:  $m(a \# as \llbracket - \> \rrbracket b \# bs) = (m(a|->b))(as \llbracket - \> \rrbracket bs)$   
**by** (simp add: map-upds-def)

**lemma** *map-upds-append1* [simp]:  $\bigwedge ys m. \text{size } xs < \text{size } ys \implies$   
 $m(xs @ [x] \llbracket \mapsto \rrbracket ys) = m(xs \llbracket \mapsto \rrbracket ys)(x \mapsto ys! \text{size } xs)$   
**apply** (induct xs)  
**apply** (clarsimp simp add: neq-Nil-conv)  
**apply** (case-tac ys)  
**apply** simp  
**apply** simp  
**done**

**lemma** *map-upds-list-update2-drop* [simp]:  
 $\llbracket \text{size } xs \leq i; i < \text{size } ys \rrbracket$   
 $\implies m(xs \llbracket \mapsto \rrbracket ys[i:=y]) = m(xs \llbracket \mapsto \rrbracket ys)$   
**apply** (induct xs arbitrary: m ys i)  
**apply** simp  
**apply** (case-tac ys)  
**apply** simp  
**apply** (simp split: nat.split)  
**done**

**lemma** *map-upd-upds-conv-if*:  
 $(f(x|->y))(xs \llbracket - \> \rrbracket ys) =$   
 $(\text{if } x : \text{set}(\text{take } (\text{length } ys) xs) \text{ then } f(xs \llbracket - \> \rrbracket ys)$   
 $\text{else } (f(xs \llbracket - \> \rrbracket ys))(x|->y))$   
**apply** (induct xs arbitrary: x y ys f)  
**apply** simp  
**apply** (case-tac ys)  
**apply** (auto split: split-if simp: fun-upd-twist)

done

**lemma** *map-upds-twist* [simp]:

$a \sim: \text{set } as \implies m(a|->b)(as[|->]bs) = m(as[|->]bs)(a|->b)$   
**using** *set-take-subset* **by** (*fastsimp simp add: map-upd-upds-conv-if*)

**lemma** *map-upds-apply-nontin* [simp]:

$x \sim: \text{set } xs \implies (f(xs[|->]ys)) \ x = f \ x$   
**apply** (*induct xs arbitrary: ys*)  
**apply** *simp*  
**apply** (*case-tac ys*)  
**apply** (*auto simp: map-upd-upds-conv-if*)  
done

**lemma** *fun-upds-append-drop* [simp]:

$\text{size } xs = \text{size } ys \implies m(xs@zs[\mapsto]ys) = m(xs[\mapsto]ys)$   
**apply** (*induct xs arbitrary: m ys*)  
**apply** *simp*  
**apply** (*case-tac ys*)  
**apply** *simp-all*  
done

**lemma** *fun-upds-append2-drop* [simp]:

$\text{size } xs = \text{size } ys \implies m(xs[\mapsto]ys@zs) = m(xs[\mapsto]ys)$   
**apply** (*induct xs arbitrary: m ys*)  
**apply** *simp*  
**apply** (*case-tac ys*)  
**apply** *simp-all*  
done

**lemma** *restrict-map-upds*[simp]:

$\llbracket \text{length } xs = \text{length } ys; \text{set } xs \subseteq D \rrbracket$   
 $\implies m(xs[\mapsto]ys)|'D = (m|'(D - \text{set } xs))(xs[\mapsto]ys)$   
**apply** (*induct xs arbitrary: m ys*)  
**apply** *simp*  
**apply** (*case-tac ys*)  
**apply** *simp*  
**apply** (*simp add: Diff-insert [symmetric] insert-absorb*)  
**apply** (*simp add: map-upd-upds-conv-if*)  
done

### 43.9 dom

**lemma** *domI*:  $m \ a = \text{Some } b \implies a : \text{dom } m$   
**by**(*simp add: dom-def*)

**lemma** *domD*:  $a : \text{dom } m \implies \exists b. m \ a = \text{Some } b$

**by** (*cases m a*) (*auto simp add: dom-def*)

**lemma** *domIff* [*iff*, *simp del*]: (*a : dom m*) = (*m a*  $\sim$  *None*)  
**by**(*simp add:dom-def*)

**lemma** *dom-empty* [*simp*]: *dom empty* = {}  
**by**(*simp add:dom-def*)

**lemma** *dom-fun-upd* [*simp*]:  
 $\text{dom}(f(x := y)) = (\text{if } y = \text{None} \text{ then } \text{dom } f - \{x\} \text{ else } \text{insert } x (\text{dom } f))$   
**by**(*auto simp add:dom-def*)

**lemma** *dom-map-of*:  $\text{dom}(\text{map-of } xys) = \{x. \exists y. (x,y) : \text{set } xys\}$   
**by** (*induct xys*) (*auto simp del: fun-upd-apply*)

**lemma** *dom-map-of-conv-image-fst*:  
 $\text{dom}(\text{map-of } xys) = \text{fst} \text{ ` } (\text{set } xys)$   
**by**(*force simp: dom-map-of*)

**lemma** *dom-map-of-zip* [*simp*]: [*length xs = length ys; distinct xs*]  $\implies$   
 $\text{dom}(\text{map-of}(\text{zip } xs \ ys)) = \text{set } xs$   
**by** (*induct rule: list-induct2*) *simp-all*

**lemma** *finite-dom-map-of*: *finite* (*dom* (*map-of l*))  
**by** (*induct l*) (*auto simp add: dom-def insert-Collect [symmetric]*)

**lemma** *dom-map-upds* [*simp*]:  
 $\text{dom}(m(xs[->]ys)) = \text{set}(\text{take } (\text{length } ys) \ xs) \cup \text{dom } m$   
**apply** (*induct xs arbitrary: m ys*)  
**apply** *simp*  
**apply** (*case-tac ys*)  
**apply** *auto*  
**done**

**lemma** *dom-map-add* [*simp*]:  $\text{dom}(m++n) = \text{dom } n \cup \text{dom } m$   
**by**(*auto simp:dom-def*)

**lemma** *dom-override-on* [*simp*]:  
 $\text{dom}(\text{override-on } f \ g \ A) =$   
 $(\text{dom } f - \{a. a : A - \text{dom } g\}) \cup \{a. a : A \text{ Int } \text{dom } g\}$   
**by**(*auto simp: dom-def override-on-def*)

**lemma** *map-add-comm*:  $\text{dom } m1 \cap \text{dom } m2 = \{\} \implies m1++m2 = m2++m1$   
**by** (*rule ext*) (*force simp: map-add-def dom-def split: option.split*)

**lemma** *dom-const* [*simp*]:  
 $\text{dom } (\lambda x. \text{Some } y) = \text{UNIV}$   
**by** *auto*



**lemma** *dom-if*:

$\text{dom } (\lambda x. \text{ if } P \ x \text{ then } f \ x \text{ else } g \ x) = \text{dom } f \cap \{x. P \ x\} \cup \text{dom } g \cap \{x. \neg P \ x\}$   
**by** (*auto split: if-splits*)

**lemma** *finite-map-freshness*:

$\text{finite } (\text{dom } (f :: 'a \rightarrow 'b)) \implies \neg \text{finite } (UNIV :: 'a \text{ set}) \implies$   
 $\exists x. f \ x = \text{None}$   
**by**(*bestsimp dest:ex-new-if-finite*)

**lemma** *dom-minus*:

$f \ x = \text{None} \implies \text{dom } f - \text{insert } x \ A = \text{dom } f - A$   
**unfolding** *dom-def* **by** *simp*

**lemma** *insert-dom*:

$f \ x = \text{Some } y \implies \text{insert } x \ (\text{dom } f) = \text{dom } f$   
**unfolding** *dom-def* **by** *auto*

#### 43.10 *ran*

**lemma** *ranI*:  $m \ a = \text{Some } b \implies b : \text{ran } m$

**by**(*auto simp: ran-def*)

**lemma** *ran-empty* [*simp*]:  $\text{ran } \text{empty} = \{\}$

**by**(*auto simp: ran-def*)

**lemma** *ran-map-upd* [*simp*]:  $m \ a = \text{None} \implies \text{ran}(m(a|->b)) = \text{insert } b \ (\text{ran } m)$

**unfolding** *ran-def*

**apply** *auto*

**apply** (*subgoal-tac aa ~ = a*)

**apply** *auto*

**done**

#### 43.11 *map-le*

**lemma** *map-le-empty* [*simp*]:  $\text{empty} \subseteq_m g$

**by** (*simp add: map-le-def*)

**lemma** *upd-None-map-le* [*simp*]:  $f(x := \text{None}) \subseteq_m f$

**by** (*force simp add: map-le-def*)

**lemma** *map-le-upd*[*simp*]:  $f \subseteq_m g \implies f(a := b) \subseteq_m g(a := b)$

**by** (*fastsimp simp add: map-le-def*)

**lemma** *map-le-imp-upd-le* [*simp*]:  $m1 \subseteq_m m2 \implies m1(x := \text{None}) \subseteq_m m2(x \mapsto y)$

**by** (*force simp add: map-le-def*)

```

lemma map-le-upds [simp]:
   $f \subseteq_m g \implies f(as \llbracket - \rceil bs) \subseteq_m g(as \llbracket - \rceil bs)$ 
apply (induct as arbitrary: f g bs)
apply simp
apply (case-tac bs)
apply auto
done

lemma map-le-implies-dom-le:  $(f \subseteq_m g) \implies (dom\ f \subseteq dom\ g)$ 
by (fastsimp simp add: map-le-def dom-def)

lemma map-le-refl [simp]:  $f \subseteq_m f$ 
by (simp add: map-le-def)

lemma map-le-trans[trans]:  $\llbracket m1 \subseteq_m m2; m2 \subseteq_m m3 \rrbracket \implies m1 \subseteq_m m3$ 
by (auto simp add: map-le-def dom-def)

lemma map-le-antisym:  $\llbracket f \subseteq_m g; g \subseteq_m f \rrbracket \implies f = g$ 
unfolding map-le-def
apply (rule ext)
apply (case-tac  $x \in dom\ f$ , simp)
apply (case-tac  $x \in dom\ g$ , simp, fastsimp)
done

lemma map-le-map-add [simp]:  $f \subseteq_m (g ++ f)$ 
by (fastsimp simp add: map-le-def)

lemma map-le-iff-map-add-commute:  $(f \subseteq_m f ++ g) = (f ++ g = g ++ f)$ 
by (fastsimp simp: map-add-def map-le-def expand-fun-eq split: option.splits)

lemma map-add-le-mapE:  $f ++ g \subseteq_m h \implies g \subseteq_m h$ 
by (fastsimp simp add: map-le-def map-add-def dom-def)

lemma map-add-le-mapI:  $\llbracket f \subseteq_m h; g \subseteq_m h; f \subseteq_m f ++ g \rrbracket \implies f ++ g \subseteq_m h$ 
by (clarsimp simp add: map-le-def map-add-def dom-def split: option.splits)

end

```

## 44 Refute: Refute

```

theory Refute
imports Hilbert-Choice List Record
uses
  Tools/prop-logic.ML
  Tools/sat-solver.ML
  Tools/refute.ML
  Tools/refute-isar.ML

```

begin

setup *Refute.setup*

```
(* ----- *)
(* 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): *)
(* ----- *)
(* 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.
(*)
(*)
(*) See 'HOL/SAT.thy' for default values.
(*)
(*)
(*) 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/Tools/refute_isar.ML     Adds 'refute'/'refute_params' to Isabelle's
(*)                             syntax
(*) HOL/Refute.thy               This file: loads the ML files, basic setup,
(*)                             documentation
(*) HOL/SAT.thy                  Sets default parameters
(*) HOL/ex/RefuteExamples.thy    Examples
(*) -----

```

end

## 45 SAT: Reconstructing external resolution proofs for propositional logic

```

theory SAT
imports Refute
uses
  Tools/cnf-funcs.ML
  Tools/sat-funcs.ML
begin

```

Late package setup: default values for `refute`, see also theory *Refute*.

**refute-params**

```
[itself=1,
 minsize=1,
 maxsize=8,
 maxvars=10000,
 maxtime=60,
 satsolver=auto]
```

**ML**  $\langle\langle$  *structure sat = SATFunc(structure cnf = cnf);*  $\rangle\rangle$

**method-setup** *sat* =  $\langle\langle$  *Scan.succeed (K (SIMPLE-METHOD' sat.sat-tac))*  $\rangle\rangle$   
*SAT solver*

**method-setup** *satx* =  $\langle\langle$  *Scan.succeed (K (SIMPLE-METHOD' sat.satx-tac))*  $\rangle\rangle$   
*SAT solver (with definitional CNF)*

**end**

## 46 Main: Main HOL

**theory** *Main*

**imports** *Plain Code-Eval Map Recdef SAT*

**begin**

Classical Higher-order Logic – only “Main”, excluding real and complex numbers etc.

See further [\[1\]](#)

**end**

## References

- [1] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.