

ZF

Lawrence C Paulson and others

December 17, 2025

Contents

1	Base of Zermelo-Fraenkel Set Theory	3
1.1	Signature	3
1.2	Bounded Quantifiers	3
1.3	Variations on Replacement	3
1.4	General union and intersection	4
1.5	Finite sets and binary operations	4
1.6	Axioms	5
1.7	Definite descriptions – via Replace over the set "1"	5
1.8	Ordered Pairing	6
1.9	Relations and Functions	7
1.10	ASCII syntax	8
1.11	Substitution	9
1.12	Bounded universal quantifier	9
1.13	Bounded existential quantifier	9
1.14	Rules for subsets	10
1.15	Rules for equality	11
1.16	Rules for Replace – the derived form of replacement	11
1.17	Rules for RepFun	12
1.18	Rules for Collect – forming a subset by separation	12
1.19	Rules for Unions	13
1.20	Rules for Unions of families	13
1.21	Rules for the empty set	13
1.22	Rules for Inter	14
1.23	Rules for Intersections of families	14
1.24	Rules for Powersets	15
1.25	Cantor's Theorem: There is no surjection from a set to its powerset.	15

2	Unordered Pairs	15
2.1	Unordered Pairs: constant <i>Upair</i>	15
2.2	Rules for Binary Union, Defined via <i>Upair</i>	16
2.3	Rules for Binary Intersection, Defined via <i>Upair</i>	16
2.4	Rules for Set Difference, Defined via <i>Upair</i>	16
2.5	Rules for <i>cons</i>	17
2.6	Singletons	17
2.7	Descriptions	18
2.8	Conditional Terms: <i>if-then-else</i>	19
2.9	Consequences of Foundation	20
2.10	Rules for Successor	20
2.11	Miniscoping of the Bounded Universal Quantifier	21
2.12	Miniscoping of the Bounded Existential Quantifier	22
2.13	Miniscoping of the Replacement Operator	23
2.14	Miniscoping of Unions	23
2.15	Miniscoping of Intersections	24
2.16	Other simprules	25
3	Ordered Pairs	25
3.1	Sigma: Disjoint Union of a Family of Sets	26
3.2	Projections <i>fst</i> and <i>snd</i>	27
3.3	The Eliminator, <i>split</i>	28
3.4	A version of <i>split</i> for Formulae: Result Type <i>o</i>	28
4	Basic Equalities and Inclusions	28
4.1	Bounded Quantifiers	29
4.2	Converse of a Relation	29
4.3	Finite Set Constructions Using <i>cons</i>	30
4.4	Binary Intersection	31
4.5	Binary Union	32
4.6	Set Difference	34
4.7	Big Union and Intersection	35
4.8	Unions and Intersections of Families	37
4.9	Image of a Set under a Function or Relation	43
4.10	Inverse Image of a Set under a Function or Relation	44
4.11	Powerset Operator	46
4.12	RepFun	46
4.13	Collect	47
5	Least and Greatest Fixed Points; the Knaster-Tarski Theorem	48
5.1	Monotone Operators	48
5.2	Proof of Knaster-Tarski Theorem using <i>lfp</i>	49
5.3	General Induction Rule for Least Fixedpoints	50

5.4	Proof of Knaster-Tarski Theorem using <i>gfp</i>	52
5.5	Coinduction Rules for Greatest Fixed Points	53
6	Booleans in Zermelo-Fraenkel Set Theory	54
6.1	Laws About 'not'	56
6.2	Laws About 'and'	56
6.3	Laws About 'or'	57
7	Disjoint Sums	57
7.1	Rules for the <i>Part</i> Primitive	58
7.2	Rules for Disjoint Sums	58
7.3	The Eliminator: <i>case</i>	60
7.4	More Rules for <i>Part(A, h)</i>	60
8	Functions, Function Spaces, Lambda-Abstraction	61
8.1	The Pi Operator: Dependent Function Space	61
8.2	Function Application	62
8.3	Lambda Abstraction	64
8.4	Extensionality	65
8.5	Images of Functions	66
8.6	Properties of <i>restrict(f, A)</i>	67
8.7	Unions of Functions	68
8.8	Domain and Range of a Function or Relation	69
8.9	Extensions of Functions	69
8.10	Function Updates	70
8.11	Monotonicity Theorems	71
8.11.1	Replacement in its Various Forms	71
8.11.2	Standard Products, Sums and Function Spaces	71
8.11.3	Converse, Domain, Range, Field	72
8.11.4	Images	72
9	Quine-Inspired Ordered Pairs and Disjoint Sums	73
9.1	Quine ordered pairing	74
9.1.1	QSigma: Disjoint union of a family of sets Generalizes Cartesian product	75
9.1.2	Projections: <i>qfst</i> , <i>qsnd</i>	75
9.1.3	Eliminator: <i>qsplit</i>	76
9.1.4	<i>qsplit</i> for predicates: result type <i>o</i>	76
9.1.5	<i>qconverse</i>	76
9.2	The Quine-inspired notion of disjoint sum	77
9.2.1	Eliminator – <i>qcase</i>	78
9.2.2	Monotonicity	79

10 Injections, Surjections, Bijections, Composition	79
10.1 Surjective Function Space	80
10.2 Injective Function Space	81
10.3 Bijections	81
10.4 Identity Function	82
10.5 Converse of a Function	83
10.6 Converses of Injections, Surjections, Bijections	84
10.7 Composition of Two Relations	84
10.8 Domain and Range – see Suppes, Section 3.1	85
10.9 Other Results	85
10.10 Composition Preserves Functions, Injections, and Surjections	85
10.11 Dual Properties of <i>inj</i> and <i>surj</i>	87
10.11.1 Inverses of Composition	87
10.11.2 Proving that a Function is a Bijection	88
10.11.3 Unions of Functions	88
10.11.4 Restrictions as Surjections and Bijections	89
10.11.5 Lemmas for Ramsey’s Theorem	89
11 Relations: Their General Properties and Transitive Closure	90
11.1 General properties of relations	91
11.1.1 irreflexivity	91
11.1.2 symmetry	91
11.1.3 antisymmetry	91
11.1.4 transitivity	91
11.2 Transitive closure of a relation	92
12 Well-Founded Recursion	97
12.1 Well-Founded Relations	98
12.1.1 Equivalences between <i>wf</i> and <i>wf-on</i>	98
12.1.2 Introduction Rules for <i>wf-on</i>	99
12.1.3 Well-founded Induction	99
12.2 Basic Properties of Well-Founded Relations	100
12.3 The Predicate <i>is-recfun</i>	101
12.4 Recursion: Main Existence Lemma	102
12.5 Unfolding <i>wftrec</i> (<i>r</i> , <i>a</i> , <i>H</i>)	103
12.5.1 Removal of the Premise <i>trans</i> (<i>r</i>)	104
13 Transitive Sets and Ordinals	104
13.1 Rules for Transset	105
13.1.1 Three Neat Characterisations of Transset	105
13.1.2 Consequences of Downwards Closure	105
13.1.3 Closure Properties	106
13.2 Lemmas for Ordinals	106
13.3 The Construction of Ordinals: 0, succ, Union	107

13.4	$<$ is 'less Than' for Ordinals	108
13.5	Natural Deduction Rules for Memrel	110
13.6	Transfinite Induction	111
14	Fundamental properties of the epsilon ordering ($<$ on ordinals)	111
14.0.1	Proving That $<$ is a Linear Ordering on the Ordinals	111
14.0.2	Some Rewrite Rules for $<$, \leq	112
14.1	Results about Less-Than or Equals	113
14.1.1	Transitivity Laws	113
14.1.2	Union and Intersection	114
14.2	Results about Limits	115
14.3	Limit Ordinals – General Properties	117
14.3.1	Traditional 3-Way Case Analysis on Ordinals	118
15	Special quantifiers	120
15.1	Quantifiers and union operator for ordinals	120
15.1.1	simplification of the new quantifiers	120
15.1.2	Union over ordinals	121
15.1.3	universal quantifier for ordinals	122
15.1.4	existential quantifier for ordinals	122
15.1.5	Rules for Ordinal-Indexed Unions	123
15.2	Quantification over a class	123
15.2.1	Relativized universal quantifier	124
15.2.2	Relativized existential quantifier	124
15.2.3	One-point rule for bounded quantifiers	126
15.2.4	Sets as Classes	126
16	The Natural numbers As a Least Fixed Point	127
16.1	Injectivity Properties and Induction	128
16.2	Variations on Mathematical Induction	130
16.3	quasinat: to allow a case-split rule for <i>nat-case</i>	131
16.4	Recursion on the Natural Numbers	132
17	Inductive and Coinductive Definitions	133
18	Epsilon Induction and Recursion	135
18.1	Basic Closure Properties	136
18.2	Leastness of <i>eclose</i>	136
18.3	Epsilon Recursion	137
18.4	Rank	139
18.5	Corollaries of Leastness	141

19 Partial and Total Orderings: Basic Definitions and Properties	143
19.1 Immediate Consequences of the Definitions	144
19.2 Restricting an Ordering's Domain	145
19.3 Empty and Unit Domains	146
19.3.1 Relations over the Empty Set	146
19.3.2 The Empty Relation Well-Orders the Unit Set	147
19.4 Order-Isomorphisms	147
19.5 Main results of Kunen, Chapter 1 section 6	150
19.6 Towards Kunen's Theorem 6.3: Linearity of the Similarity Relation	152
19.7 Miscellaneous Results by Krzysztof Grabczewski	155
19.8 Lemmas for the Reflexive Orders	156
20 Combining Orderings: Foundations of Ordinal Arithmetic	157
20.1 Addition of Relations – Disjoint Sum	157
20.1.1 Rewrite rules. Can be used to obtain introduction rules	157
20.1.2 Elimination Rule	158
20.1.3 Type checking	158
20.1.4 Linearity	158
20.1.5 Well-foundedness	158
20.1.6 An <i>ord-iso</i> congruence law	159
20.1.7 Associativity	160
20.2 Multiplication of Relations – Lexicographic Product	160
20.2.1 Rewrite rule. Can be used to obtain introduction rules	160
20.2.2 Type checking	160
20.2.3 Linearity	160
20.2.4 Well-foundedness	160
20.2.5 An <i>ord-iso</i> congruence law	161
20.2.6 Distributive law	162
20.2.7 Associativity	162
20.3 Inverse Image of a Relation	163
20.3.1 Rewrite rule	163
20.3.2 Type checking	163
20.3.3 Partial Ordering Properties	163
20.3.4 Linearity	163
20.3.5 Well-foundedness	164
20.4 Every well-founded relation is a subset of some inverse image of an ordinal	165
20.5 Other Results	166
20.5.1 The Empty Relation	166
20.5.2 The "measure" relation is useful with wfrec	166
20.5.3 Well-foundedness of Unions	167
20.5.4 Bijections involving Powersets	168

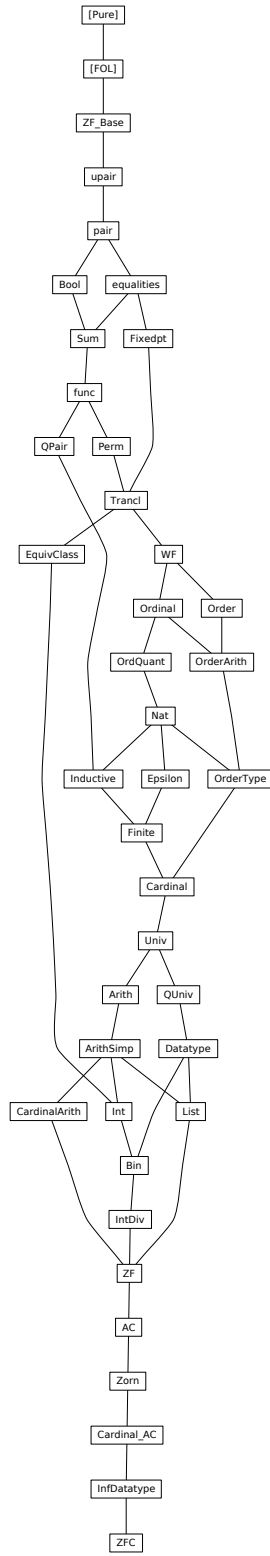
21 Order Types and Ordinal Arithmetic	168
21.1 Proofs needing the combination of Ordinal.thy and Order.thy	169
21.2 Ordermap and ordertype	170
21.2.1 Unfolding of ordermap	170
21.2.2 Showing that ordermap, ordertype yield ordinals . . .	170
21.2.3 ordermap preserves the orderings in both directions .	171
21.2.4 Isomorphisms involving ordertype	171
21.2.5 Basic equalities for ordertype	172
21.2.6 A fundamental unfolding law for ordertype.	173
21.3 Alternative definition of ordinal	174
21.4 Ordinal Addition	174
21.4.1 Order Type calculations for radd	174
21.4.2 ordify: trivial coercion to an ordinal	175
21.4.3 Basic laws for ordinal addition	175
21.4.4 Ordinal addition with successor – via associativity! .	178
21.5 Ordinal Subtraction	181
21.6 Ordinal Multiplication	182
21.6.1 A useful unfolding law	182
21.6.2 Basic laws for ordinal multiplication	184
21.6.3 Ordering/monotonicity properties of ordinal multipli- cation	185
21.7 The Relation Lt	187
22 Finite Powerset Operator and Finite Function Space	188
22.1 Finite Powerset Operator	188
22.2 Finite Function Space	190
22.3 The Contents of a Singleton Set	192
23 Cardinal Numbers Without the Axiom of Choice	192
23.1 The Schroeder-Bernstein Theorem	193
23.2 lesspoll: contributions by Krzysztof Grabczewski	195
23.3 Basic Properties of Cardinals	198
23.4 The finite cardinals	203
23.5 The first infinite cardinal: Omega, or nat	205
23.6 Towards Cardinal Arithmetic	207
23.7 Lemmas by Krzysztof Grabczewski	208
23.8 Finite and infinite sets	209
24 The Cumulative Hierarchy and a Small Universe for Recursive Types	216
24.1 Immediate Consequences of the Definition of $Vfrom(A, i)$. .	216
24.1.1 Monotonicity	216
24.1.2 A fundamental equality: $Vfrom$ does not require or- dinals!	217

24.2	Basic Closure Properties	217
24.2.1	Finite sets and ordered pairs	218
24.3	0, Successor and Limit Equations for $Vfrom$	218
24.4	$Vfrom$ applied to Limit Ordinals	219
24.4.1	Closure under Disjoint Union	220
24.5	Properties assuming $Transset(A)$	221
24.5.1	Products	222
24.5.2	Disjoint Sums, or Quine Ordered Pairs	222
24.5.3	Function Space!	223
24.6	The Set $Vset(i)$	223
24.6.1	Characterisation of the elements of $Vset(i)$	223
24.6.2	Reasoning about Sets in Terms of Their Elements' Ranks	224
24.6.3	Set Up an Environment for Simplification	225
24.6.4	Recursion over Vset Levels!	225
24.7	The Datatype Universe: $univ(A)$	226
24.7.1	The Set $univ(A)$ as a Limit	226
24.8	Closure Properties for $univ(A)$	227
24.8.1	Closure under Unordered and Ordered Pairs	227
24.8.2	The Natural Numbers	227
24.8.3	Instances for 1 and 2	228
24.8.4	Closure under Disjoint Union	228
24.9	Finite Branching Closure Properties	229
24.9.1	Closure under Finite Powerset	229
24.9.2	Closure under Finite Powers: Functions from a Natural Number	229
24.9.3	Closure under Finite Function Space	229
24.10*	For QUniv. Properties of $Vfrom$ analogous to the "take-lemma" *	230
25	A Small Universe for Lazy Recursive Types	231
25.1	Properties involving $Transset$ and Sum	231
25.2	Introduction and Elimination Rules	232
25.3	Closure Properties	232
25.4	Quine Disjoint Sum	233
25.5	Closure for Quine-Inspired Products and Sums	233
25.6	Quine Disjoint Sum	234
25.7	The Natural Numbers	234
25.8	"Take-Lemma" Rules	234
26	Datatype and CoDatatype Definitions	235

27 Arithmetic Operators and Their Definitions	237
27.1 <i>natify</i> , the Coercion to <i>nat</i>	239
27.2 Typing rules	240
27.3 Addition	241
27.4 Monotonicity of Addition	243
27.5 Multiplication	246
28 Arithmetic with simplification	248
28.1 Arithmetic simplification	248
28.1.1 Examples	249
28.2 Difference	250
28.3 Remainder	251
28.4 Division	252
28.5 Further Facts about Remainder	254
28.6 Additional theorems about \leq	255
28.7 Cancellation Laws for Common Factors in Comparisons . . .	256
28.8 More Lemmas about Remainder	258
28.8.1 More Lemmas About Difference	260
29 Lists in Zermelo-Fraenkel Set Theory	261
29.1 The function <i>zip</i>	278
30 Equivalence Relations	287
30.1 Suppes, Theorem 70: r is an equiv relation iff <i>converse</i> (r) O $r = r$	287
30.2 Defining Unary Operations upon Equivalence Classes	289
30.3 Defining Binary Operations upon Equivalence Classes	290
31 The Integers as Equivalence Classes Over Pairs of Natural Numbers	291
31.1 Proving that <i>intrel</i> is an equivalence relation	293
31.2 Collapsing rules: to remove <i>intify</i> from arithmetic expressions	294
31.3 <i>zminus</i> : unary negation on <i>int</i>	295
31.4 <i>znegative</i> : the test for negative integers	296
31.5 <i>nat-of</i> : Coercion of an Integer to a Natural Number	297
31.6 <i>zmagnitude</i> : magnitude of an integer, as a natural number . .	297
31.7 ($\$+$): addition on <i>int</i>	299
31.8 ($\$*$): Integer Multiplication	301
31.9 The "Less Than" Relation	304
31.10 Less Than or Equals	306
31.11 More subtraction laws (for <i>zcompare-rls</i>)	307
31.12 Monotonicity and Cancellation Results for Instantiation of the CancelNumerals Simprocs	308
31.13 Comparison laws	309

31.13.1	More inequality lemmas	309
31.13.2	The next several equations are permutative: watch out!	309
32	Arithmetic on Binary Integers	310
32.0.1	The Carry and Borrow Functions, <i>bin-succ</i> and <i>bin-pred</i>	313
32.0.2	<i>bin-minus</i> : Unary Negation of Binary Integers	313
32.0.3	<i>bin-add</i> : Binary Addition	313
32.0.4	<i>bin-mult</i> : Binary Multiplication	314
32.1	Computations	315
32.2	Simplification Rules for Comparison of Binary Numbers . . .	316
32.2.1	Examples	324
33	The Division Operators Div and Mod	325
33.1	Uniqueness and monotonicity of quotients and remainders . .	332
33.2	Correctness of posDivAlg, the Division Algorithm for $a \geq 0$ and $b > 0$	333
33.3	Some convenient biconditionals for products of signs	334
33.4	Correctness of negDivAlg, the division algorithm for $a < 0$ and $b > 0$	336
33.5	Existence shown by proving the division algorithm to be correct	338
33.6	division of a number by itself	344
33.7	Computation of division and remainder	345
33.8	Monotonicity in the first argument (divisor)	348
33.9	Monotonicity in the second argument (dividend)	348
33.10	More algebraic laws for zdiv and zmod	351
33.11	proving $a \text{ zdiv } (b * c) = (a \text{ zdiv } b) \text{ zdiv } c$	354
33.12	Cancellation of common factors in "zdiv"	356
33.13	Distribution of factors over "zmod"	356
34	Cardinal Arithmetic Without the Axiom of Choice	358
34.1	Cardinal addition	360
34.1.1	Cardinal addition is commutative	360
34.1.2	Cardinal addition is associative	360
34.1.3	0 is the identity for addition	361
34.1.4	Addition by another cardinal	361
34.1.5	Monotonicity of addition	361
34.1.6	Addition of finite cardinals is "ordinary" addition . . .	362
34.2	Cardinal multiplication	362
34.2.1	Cardinal multiplication is commutative	362
34.2.2	Cardinal multiplication is associative	363
34.2.3	Cardinal multiplication distributes over addition . . .	363
34.2.4	Multiplication by 0 yields 0	364
34.2.5	1 is the identity for multiplication	364
34.3	Some inequalities for multiplication	364

34.3.1	Multiplication by a non-zero cardinal	364
34.3.2	Monotonicity of multiplication	365
34.4	Multiplication of finite cardinals is "ordinary" multiplication .	365
34.5	Infinite Cardinals are Limit Ordinals	366
34.5.1	Establishing the well-ordering	368
34.5.2	Characterising initial segments of the well-ordering . .	368
34.5.3	The cardinality of initial segments	369
34.5.4	Toward's Kunen's Corollary 10.13 (1)	372
34.6	For Every Cardinal Number There Exists A Greater One . .	373
34.7	Basic Properties of Successor Cardinals	374
34.7.1	Removing elements from a finite set decreases its car- dinality	375
34.7.2	Theorems by Krzysztof Grabczewski, proofs by lcp . .	376
35	Main ZF Theory: Everything Except AC	377
35.1	Iteration of the function F	377
35.2	Transfinite Recursion	378
36	The Axiom of Choice	379
37	Zorn's Lemma	380
37.1	Mathematical Preamble	381
37.2	The Transfinite Construction	381
37.3	Some Properties of the Transfinite Construction	381
37.4	Hausdorff's Theorem: Every Set Contains a Maximal Chain .	383
37.5	Zorn's Lemma: If All Chains in S Have Upper Bounds In S , then S contains a Maximal Element	385
37.6	Zermelo's Theorem: Every Set can be Well-Ordered	386
37.7	Zorn's Lemma for Partial Orders	388
38	Cardinal Arithmetic Using AC	390
38.1	Strengthened Forms of Existing Theorems on Cardinals . . .	390
38.2	The relationship between cardinality and le-pollence	391
38.3	Other Applications of AC	392
38.4	The Main Result for Infinite-Branching Datatypes	394
39	Infinite-Branching Datatype Definitions	395



1 Base of Zermelo-Fraenkel Set Theory

```
theory ZF-Base
imports FOL
begin
```

1.1 Signature

```
declare [[eta-contract = false]]
```

```
typedecl i
instance i :: term ..
```

```
axiomatization mem :: [i, i]  $\Rightarrow$  o (infixl  $\in$  50) — membership relation
and zero :: i ( $\langle 0 \rangle$ ) — the empty set
and Pow :: i  $\Rightarrow$  i — power sets
and Inf :: i — infinite set
and Union :: i  $\Rightarrow$  i ( $\langle \langle \text{open-block notation} = \langle \text{prefix } \bigcup \rangle \bigcup - \rangle \rangle$  [90] 90)
and PrimReplace :: [i, [i, i]  $\Rightarrow$  o]  $\Rightarrow$  i
```

```
abbreviation not-mem :: [i, i]  $\Rightarrow$  o (infixl  $\notin$  50) — negated membership
relation
where  $x \notin y \equiv \neg (x \in y)$ 
```

1.2 Bounded Quantifiers

```
definition Ball :: [i, i  $\Rightarrow$  o]  $\Rightarrow$  o
where Ball(A, P)  $\equiv \forall x. x \in A \longrightarrow P(x)$ 
```

```
definition Bex :: [i, i  $\Rightarrow$  o]  $\Rightarrow$  o
where Bex(A, P)  $\equiv \exists x. x \in A \wedge P(x)$ 
```

syntax

```
-Ball :: [pttrn, i, o]  $\Rightarrow$  o ( $\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \forall \in \rangle \forall - \in - / - \rangle \rangle$  10)
-Bex :: [pttrn, i, o]  $\Rightarrow$  o ( $\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \exists \in \rangle \exists - \in - / - \rangle \rangle$  10)
```

syntax-consts

```
-Ball  $\equiv$  Ball and
-Bex  $\equiv$  Bex
```

translations

```
 $\forall x \in A. P \equiv \text{CONST Ball}(A, \lambda x. P)$ 
 $\exists x \in A. P \equiv \text{CONST Bex}(A, \lambda x. P)$ 
```

1.3 Variations on Replacement

```
definition Replace :: [i, [i, i]  $\Rightarrow$  o]  $\Rightarrow$  i
where Replace(A, P)  $\equiv \text{PrimReplace}(A, \lambda x y. (\exists ! z. P(x, z)) \wedge P(x, y))$ 
```

syntax

```
-Replace :: [pttrn, pttrn, i, o]  $\Rightarrow$  i ( $\langle \langle \text{indent} = 1 \text{ notation} = \langle \text{mixfix relational replacement} \rangle \{ - / - \in -, - \} \rangle \rangle$ )
```

syntax-consts

-*Replace* \Rightarrow *Replace*

translations

$\{y. x \in A, Q\} \Rightarrow \text{CONST } \text{Replace}(A, \lambda x y. Q)$

definition *RepFun* :: $[i, i \Rightarrow i] \Rightarrow i$

where $\text{RepFun}(A, f) \equiv \{y . x \in A, y = f(x)\}$

syntax

-*RepFun* :: $[i, \text{pttrn}, i] \Rightarrow i$ ($\langle \langle \text{indent} = 1 \text{ notation} = \langle \text{mixfix functional replace-ment} \rangle \rangle \{- ./ - \in -\} \rangle [51, 0, 51] \rangle$)

syntax-consts

-*RepFun* \Rightarrow *RepFun*

translations

$\{b. x \in A\} \Rightarrow \text{CONST } \text{RepFun}(A, \lambda x. b)$

definition *Collect* :: $[i, i \Rightarrow o] \Rightarrow i$

where $\text{Collect}(A, P) \equiv \{y . x \in A, x = y \wedge P(x)\}$

syntax

-*Collect* :: $[\text{pttrn}, i, o] \Rightarrow i$ ($\langle \langle \text{indent} = 1 \text{ notation} = \langle \text{mixfix set comprehension} \rangle \rangle \{- \in - ./ -\} \rangle \rangle$)

syntax-consts

-*Collect* \Rightarrow *Collect*

translations

$\{x \in A. P\} \Rightarrow \text{CONST } \text{Collect}(A, \lambda x. P)$

1.4 General union and intersection

definition *Inter* :: $i \Rightarrow i$ ($\langle \langle \text{open-block notation} = \langle \text{prefix } \bigcap \rangle \bigcap - \rangle [90] 90 \rangle$)

where $\bigcap(A) \equiv \{x \in \bigcup(A) . \forall y \in A. x \in y\}$

syntax

-*UNION* :: $[\text{pttrn}, i, i] \Rightarrow i$ ($\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \bigcup \in \rangle \bigcup - \in - ./ - \rangle 10 \rangle$)

-*INTER* :: $[\text{pttrn}, i, i] \Rightarrow i$ ($\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \bigcap \in \rangle \bigcap - \in - ./ - \rangle 10 \rangle$)

syntax-consts

-*UNION* == *Union* and

-*INTER* == *Inter*

translations

$\bigcup_{x \in A} B == \text{CONST } \text{Union}(\{B. x \in A\})$

$\bigcap_{x \in A} B == \text{CONST } \text{Inter}(\{B. x \in A\})$

1.5 Finite sets and binary operations

definition *Upair* :: $[i, i] \Rightarrow i$

where $\text{Upair}(a, b) \equiv \{y. x \in \text{Pow}(\text{Pow}(\theta)), (x = \theta \wedge y = a) \mid (x = \text{Pow}(\theta) \wedge y = b)\}$

definition *Subset* :: $[i, i] \Rightarrow o$ (**infixl** $\langle \subseteq \rangle$ 50) — subset relation
where *subset-def*: $A \subseteq B \equiv \forall x \in A. x \in B$

definition *Diff* :: $[i, i] \Rightarrow i$ (**infixl** $\langle - \rangle$ 65) — set difference
where $A - B \equiv \{ x \in A \mid \neg(x \in B) \}$

definition *Un* :: $[i, i] \Rightarrow i$ (**infixl** $\langle \cup \rangle$ 65) — binary union
where $A \cup B \equiv \bigcup (U\text{pair}(A, B))$

definition *Int* :: $[i, i] \Rightarrow i$ (**infixl** $\langle \cap \rangle$ 70) — binary intersection
where $A \cap B \equiv \bigcap (U\text{pair}(A, B))$

definition *cons* :: $[i, i] \Rightarrow i$
where $\text{cons}(a, A) \equiv U\text{pair}(a, A) \cup A$

definition *succ* :: $i \Rightarrow i$
where $\text{succ}(i) \equiv \text{cons}(i, i)$

nonterminal *is*

syntax

:: $i \Rightarrow is$ ($\langle - \rangle$)
-Enum :: $[i, is] \Rightarrow is$ ($\langle -, / - \rangle$)
-Finset :: $is \Rightarrow i$ ($\langle (\langle \text{indent}=1 \text{ notation}=\langle \text{mixfix set enumeration} \rangle \{ - \} \rangle) \rangle$)

translations

$\{x, xs\} == \text{CONST } \text{cons}(x, \{xs\})$
 $\{x\} == \text{CONST } \text{cons}(x, \emptyset)$

1.6 Axioms

axiomatization

where

extension: $A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$ **and**
Union-iff: $A \in \bigcup(C) \longleftrightarrow (\exists B \in C. A \in B)$ **and**
Pow-iff: $A \in \text{Pow}(B) \longleftrightarrow A \subseteq B$ **and**

infinity: $0 \in \text{Inf} \wedge (\forall y \in \text{Inf}. \text{succ}(y) \in \text{Inf})$ **and**

foundation: $A = \emptyset \vee (\exists x \in A. \forall y \in x. y \notin A)$ **and**

replacement: $(\forall x \in A. \forall y z. P(x, y) \wedge P(x, z) \longrightarrow y = z) \implies$
 $b \in \text{PrimReplace}(A, P) \longleftrightarrow (\exists x \in A. P(x, b))$

1.7 Definite descriptions – via Replace over the set "1"

definition *The* :: $(i \Rightarrow o) \Rightarrow i$ (**binder** $\langle \text{THE} \rangle$ 10)
where *the-def*: $\text{The}(P) \equiv \bigcup (\{y \mid x \in \{0\}, P(y)\})$

definition $If :: [o, i, i] \Rightarrow i$ ($\langle \langle notation = \langle mixfix\ if\ then\ else \rangle \rangle if\ (-) / then\ (-) / else\ (-) \rangle [10] 10$)
where $if\text{-}def: if\ P\ then\ a\ else\ b \equiv THE\ z.\ P \wedge z=a \mid \neg P \wedge z=b$

abbreviation (*input*)
 $old\text{-}if :: [o, i, i] \Rightarrow i$ ($\langle if\ '(-,-,-)' \rangle$)
where $if(P,a,b) \equiv If(P,a,b)$

1.8 Ordered Pairing

definition $Pair :: [i, i] \Rightarrow i$
where $Pair(a,b) \equiv \{\{a,a\}, \{a,b\}\}$

definition $fst :: i \Rightarrow i$
where $fst(p) \equiv THE\ a.\ \exists b.\ p = Pair(a, b)$

definition $snd :: i \Rightarrow i$
where $snd(p) \equiv THE\ b.\ \exists a.\ p = Pair(a, b)$

definition $split :: [[i, i] \Rightarrow 'a, i] \Rightarrow 'a::\{\}$ — for pattern-matching
where $split(c) \equiv \lambda p.\ c(fst(p), snd(p))$

nonterminal *tuple-args*

syntax

$:: i \Rightarrow tuple\text{-}args\ (\langle \cdot \rangle)$
 $-Tuple\text{-}args :: [i, tuple\text{-}args] \Rightarrow tuple\text{-}args\ (\langle \cdot, / \cdot \rangle)$
 $-Tuple :: [i, tuple\text{-}args] \Rightarrow i$ ($\langle \langle indent = 1\ notation = \langle mixfix\ tuple\ enumeration \rangle \rangle \langle \cdot, / \cdot \rangle \rangle$)

translations

$\langle x, y, z \rangle == \langle x, \langle y, z \rangle \rangle$
 $\langle x, y \rangle == CONST\ Pair(x, y)$

nonterminal *patterns*

syntax

$-pattern :: patterns \Rightarrow pptrn$ ($\langle \langle open\text{-}block\ notation = \langle pattern\ tuple \rangle \rangle \langle \cdot \rangle \rangle$)
 $:: pptrn \Rightarrow patterns\ (\langle \cdot \rangle)$
 $-patterns :: [pptrn, patterns] \Rightarrow patterns\ (\langle \cdot, / \cdot \rangle)$

syntax-consts

$-pattern\ -patterns == split$

translations

$\lambda \langle x, y, zs \rangle. b == CONST\ split(\lambda x\ \langle y, zs \rangle. b)$
 $\lambda \langle x, y \rangle. b == CONST\ split(\lambda x\ y. b)$

definition $Sigma :: [i, i \Rightarrow i] \Rightarrow i$
where $Sigma(A,B) \equiv \bigcup_{x \in A} \bigcup_{y \in B(x)} \{\langle x, y \rangle\}$

abbreviation $cart\text{-}prod :: [i, i] \Rightarrow i$ (**infixr** $\langle \times \rangle$ 80) — Cartesian product

where $A \times B \equiv \text{Sigma}(A, \lambda-. B)$

1.9 Relations and Functions

definition $\text{converse} :: i \Rightarrow i$
 where $\text{converse}(r) \equiv \{z. w \in r, \exists x y. w = \langle x, y \rangle \wedge z = \langle y, x \rangle\}$

definition $\text{domain} :: i \Rightarrow i$
 where $\text{domain}(r) \equiv \{x. w \in r, \exists y. w = \langle x, y \rangle\}$

definition $\text{range} :: i \Rightarrow i$
 where $\text{range}(r) \equiv \text{domain}(\text{converse}(r))$

definition $\text{field} :: i \Rightarrow i$
 where $\text{field}(r) \equiv \text{domain}(r) \cup \text{range}(r)$

definition $\text{relation} :: i \Rightarrow o$ — recognizes sets of pairs
 where $\text{relation}(r) \equiv \forall z \in r. \exists x y. z = \langle x, y \rangle$

definition $\text{function} :: i \Rightarrow o$ — recognizes functions; can have non-pairs
 where $\text{function}(r) \equiv \forall x y. \langle x, y \rangle \in r \longrightarrow (\forall y'. \langle x, y' \rangle \in r \longrightarrow y = y')$

definition $\text{Image} :: [i, i] \Rightarrow i$ (**infixl** $\langle \langle \rangle 90$) — image
 where $\text{image-def}: r \text{ “ } A \equiv \{y \in \text{range}(r). \exists x \in A. \langle x, y \rangle \in r\}$

definition $\text{vimage} :: [i, i] \Rightarrow i$ (**infixl** $\langle \langle \rangle 90$) — inverse image
 where $\text{vimage-def}: r \text{ - “ } A \equiv \text{converse}(r) \text{ “ } A$

definition $\text{restrict} :: [i, i] \Rightarrow i$
 where $\text{restrict}(r, A) \equiv \{z \in r. \exists x \in A. \exists y. z = \langle x, y \rangle\}$

definition $\text{Lambda} :: [i, i \Rightarrow i] \Rightarrow i$
 where $\text{lam-def}: \text{Lambda}(A, b) \equiv \{\langle x, b(x) \rangle. x \in A\}$

definition $\text{apply} :: [i, i] \Rightarrow i$ (**infixl** $\langle \langle \rangle 90$) — function application
 where $f'a \equiv \bigcup (f' \text{ “ } \{a\})$

definition $\text{Pi} :: [i, i \Rightarrow i] \Rightarrow i$
 where $\text{Pi}(A, B) \equiv \{f \in \text{Pow}(\text{Sigma}(A, B)). A \subseteq \text{domain}(f) \wedge \text{function}(f)\}$

abbreviation $\text{function-space} :: [i, i] \Rightarrow i$ (**infixr** $\langle \langle \rangle 60$) — function space
 where $A \rightarrow B \equiv \text{Pi}(A, \lambda-. B)$

syntax

-PROD :: [pttrn, i, i] ⇒ i (⟨⟨indent=3 notation=⟨mixfix Π ∈⟩⟩Π -∈-./ -)⟩ 10)

-SUM :: [pttrn, i, i] ⇒ i (⟨⟨indent=3 notation=⟨mixfix ∑ ∈⟩⟩∑ -∈-./ -)⟩ 10)

-lam :: [pttrn, i, i] ⇒ i (⟨⟨indent=3 notation=⟨mixfix λ ∈⟩⟩λ -∈-./ -)⟩ 10)

syntax-consts

-PROD == Pi and

-SUM == Sigma and

-lam == Lambda

translations

$\prod_{x \in A} B == \text{CONST } \text{Pi}(A, \lambda x. B)$

$\sum_{x \in A} B == \text{CONST } \text{Sigma}(A, \lambda x. B)$

$\lambda x \in A. f == \text{CONST } \text{Lambda}(A, \lambda x. f)$

1.10 ASCII syntax**notation (ASCII)**

cart-prod (infixr ⟨*⟩ 80) and

Int (infixl ⟨Int⟩ 70) and

Un (infixl ⟨Un⟩ 65) and

function-space (infixr ⟨->⟩ 60) and

Subset (infixl ⟨<=>⟩ 50) and

mem (infixl ⟨:⟩ 50) and

not-mem (infixl ⟨¬:⟩ 50)

syntax (ASCII)

-Ball :: [pttrn, i, o] ⇒ o (⟨⟨indent=3 notation=⟨binder ALL:⟩⟩ALL -:-./ -)⟩ 10)

-Bex :: [pttrn, i, o] ⇒ o (⟨⟨indent=3 notation=⟨binder EX:⟩⟩EX -:-./ -)⟩ 10)

-Collect :: [pttrn, i, o] ⇒ i (⟨⟨indent=1 notation=⟨mixfix set comprehension⟩⟩{-: -./ -})⟩)

-Replace :: [pttrn, pttrn, i, o] ⇒ i (⟨⟨indent=1 notation=⟨mixfix relational replacement⟩⟩{-: -./ -})⟩)

-RepFun :: [i, pttrn, i] ⇒ i (⟨⟨indent=1 notation=⟨mixfix functional replacement⟩⟩{-: -./ -})⟩ [51,0,51])

-UNION :: [pttrn, i, i] ⇒ i (⟨⟨indent=3 notation=⟨binder UN:⟩⟩UN -:-./ -)⟩ 10)

-INTER :: [pttrn, i, i] ⇒ i (⟨⟨indent=3 notation=⟨binder INT:⟩⟩INT -:-./ -)⟩ 10)

-PROD :: [pttrn, i, i] ⇒ i (⟨⟨indent=3 notation=⟨binder PROD:⟩⟩PROD -:-./ -)⟩ 10)

-SUM :: [pttrn, i, i] ⇒ i (⟨⟨indent=3 notation=⟨binder SUM:⟩⟩SUM -:-./ -)⟩ 10)

-lam :: [pttrn, i, i] ⇒ i (⟨⟨indent=3 notation=⟨binder lam:⟩⟩lam -:-./ -)⟩ 10)

-*Tuple* :: $[i, \text{tuple-args}] \Rightarrow i$ ($\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix tuple enumeration} \rangle \langle -, / - \rangle \rangle$)
 -*pattern* :: $\text{patterns} \Rightarrow \text{pttrn}$ ($\langle \langle -, \rangle \rangle$)

1.11 Substitution

lemma *subst-elem*: $\llbracket b \in A; a = b \rrbracket \Longrightarrow a \in A$
by (*erule ssust, assumption*)

1.12 Bounded universal quantifier

lemma *ballI* [*intro!*]: $\llbracket \bigwedge x. x \in A \Longrightarrow P(x) \rrbracket \Longrightarrow \forall x \in A. P(x)$
by (*simp add: Ball-def*)

lemmas *strip = impI allI ballI*

lemma *bspec* [*dest?*]: $\llbracket \forall x \in A. P(x); x: A \rrbracket \Longrightarrow P(x)$
by (*simp add: Ball-def*)

lemma *rev-ballE* [*elim*]:
 $\llbracket \forall x \in A. P(x); x \notin A \Longrightarrow Q; P(x) \Longrightarrow Q \rrbracket \Longrightarrow Q$
by (*simp add: Ball-def, blast*)

lemma *ballE*: $\llbracket \forall x \in A. P(x); P(x) \Longrightarrow Q; x \notin A \Longrightarrow Q \rrbracket \Longrightarrow Q$
by *blast*

lemma *rev-bspec*: $\llbracket x: A; \forall x \in A. P(x) \rrbracket \Longrightarrow P(x)$
by (*simp add: Ball-def*)

lemma *ball-triv* [*simp*]: $(\forall x \in A. P) \longleftrightarrow ((\exists x. x \in A) \longrightarrow P)$
by (*simp add: Ball-def*)

lemma *ball-cong* [*cong*]:
 $\llbracket A = A'; \bigwedge x. x \in A' \Longrightarrow P(x) \longleftrightarrow P'(x) \rrbracket \Longrightarrow (\forall x \in A. P(x)) \longleftrightarrow (\forall x \in A'. P'(x))$
by (*simp add: Ball-def*)

lemma *atomize-ball*:
 $(\bigwedge x. x \in A \Longrightarrow P(x)) \equiv \text{Trueprop } (\forall x \in A. P(x))$
by (*simp only: Ball-def atomize-all atomize-imp*)

lemmas [*symmetric, rulify*] = *atomize-ball*
and [*symmetric, defn*] = *atomize-ball*

1.13 Bounded existential quantifier

lemma *bexI* [*intro*]: $\llbracket P(x); x: A \rrbracket \Longrightarrow \exists x \in A. P(x)$

by (*simp add: Bex-def, blast*)

lemma *rev-bexI*: $\llbracket x \in A; P(x) \rrbracket \implies \exists x \in A. P(x)$
by *blast*

lemma *bexCI*: $\llbracket \forall x \in A. \neg P(x) \implies P(a); a: A \rrbracket \implies \exists x \in A. P(x)$
by *blast*

lemma *bexE [elim!]*: $\llbracket \exists x \in A. P(x); \bigwedge x. \llbracket x \in A; P(x) \rrbracket \implies Q \rrbracket \implies Q$
by (*simp add: Bex-def, blast*)

lemma *bex-triv [simp]*: $(\exists x \in A. P) \longleftrightarrow ((\exists x. x \in A) \wedge P)$
by (*simp add: Bex-def*)

lemma *bex-cong [cong]*:
 $\llbracket A=A'; \bigwedge x. x \in A' \implies P(x) \longleftrightarrow P'(x) \rrbracket$
 $\implies (\exists x \in A. P(x)) \longleftrightarrow (\exists x \in A'. P'(x))$
by (*simp add: Bex-def cong: conj-cong*)

1.14 Rules for subsets

lemma *subsetI [intro!]*:
 $(\bigwedge x. x \in A \implies x \in B) \implies A \subseteq B$
by (*simp add: subset-def*)

lemma *subsetD [elim]*: $\llbracket A \subseteq B; c \in A \rrbracket \implies c \in B$
unfolding *subset-def*
apply (*erule bspec, assumption*)
done

lemma *subsetCE [elim]*:
 $\llbracket A \subseteq B; c \notin A \implies P; c \in B \implies P \rrbracket \implies P$
by (*simp add: subset-def, blast*)

lemma *rev-subsetD*: $\llbracket c \in A; A \subseteq B \rrbracket \implies c \in B$
by *blast*

lemma *contra-subsetD*: $\llbracket A \subseteq B; c \notin B \rrbracket \implies c \notin A$
by *blast*

lemma *rev-contra-subsetD*: $\llbracket c \notin B; A \subseteq B \rrbracket \implies c \notin A$
by *blast*

lemma *subset-refl* [*simp*]: $A \subseteq A$
by *blast*

lemma *subset-trans*: $\llbracket A \subseteq B; B \subseteq C \rrbracket \Longrightarrow A \subseteq C$
by *blast*

lemma *subset-iff*:
 $A \subseteq B \longleftrightarrow (\forall x. x \in A \longrightarrow x \in B)$
by *auto*

For calculations

declare *subsetD* [*trans*] *rev-subsetD* [*trans*] *subset-trans* [*trans*]

1.15 Rules for equality

lemma *equalityI* [*intro*]: $\llbracket A \subseteq B; B \subseteq A \rrbracket \Longrightarrow A = B$
by (*rule extension* [*THEN iffD2*], *rule conjI*)

lemma *equality-iffI*: $(\bigwedge x. x \in A \longleftrightarrow x \in B) \Longrightarrow A = B$
by (*rule equalityI*, *blast+*)

lemmas *equalityD1* = *extension* [*THEN iffD1*, *THEN conjunct1*]
lemmas *equalityD2* = *extension* [*THEN iffD1*, *THEN conjunct2*]

lemma *equalityE*: $\llbracket A = B; \llbracket A \subseteq B; B \subseteq A \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$
by (*blast dest: equalityD1 equalityD2*)

lemma *equalityCE*:
 $\llbracket A = B; \llbracket c \in A; c \in B \rrbracket \Longrightarrow P; \llbracket c \notin A; c \notin B \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$
by (*erule equalityE*, *blast*)

lemma *equality-iffD*:
 $A = B \Longrightarrow (\bigwedge x. x \in A \longleftrightarrow x \in B)$
by *auto*

1.16 Rules for Replace – the derived form of replacement

lemma *Replace-iff*:
 $b \in \{y. x \in A, P(x, y)\} \longleftrightarrow (\exists x \in A. P(x, b) \wedge (\forall y. P(x, y) \longrightarrow y = b))$
unfolding *Replace-def*
by (*rule replacement* [*THEN iff-trans*], *blast+*)

lemma *ReplaceI* [*intro*]:
 $\llbracket P(x, b); x: A; \bigwedge y. P(x, y) \Longrightarrow y = b \rrbracket \Longrightarrow$
 $b \in \{y. x \in A, P(x, y)\}$
by (*rule Replace-iff* [*THEN iffD2*], *blast*)

lemma *ReplaceE*:

$$\llbracket b \in \{y. x \in A, P(x,y)\};$$

$$\bigwedge x. \llbracket x: A; P(x,b); \forall y. P(x,y) \longrightarrow y=b \rrbracket \Longrightarrow R$$

$$\rrbracket \Longrightarrow R$$
by (*rule Replace-iff [THEN iffD1, THEN bexE], simp+*)

lemma *ReplaceE2 [elim!]*:

$$\llbracket b \in \{y. x \in A, P(x,y)\};$$

$$\bigwedge x. \llbracket x: A; P(x,b) \rrbracket \Longrightarrow R$$

$$\rrbracket \Longrightarrow R$$
by (*erule ReplaceE, blast*)

lemma *Replace-cong [cong]*:

$$\llbracket A=B; \bigwedge x y. x \in B \Longrightarrow P(x,y) \longleftrightarrow Q(x,y) \rrbracket \Longrightarrow \text{Replace}(A,P) = \text{Replace}(B,Q)$$
apply (*rule equality-iffI*)
apply (*simp add: Replace-iff*)
done

1.17 Rules for RepFun

lemma *RepFunI*: $a \in A \Longrightarrow f(a) \in \{f(x). x \in A\}$
by (*simp add: RepFun-def Replace-iff, blast*)

lemma *RepFun-eqI [intro]*: $\llbracket b=f(a); a \in A \rrbracket \Longrightarrow b \in \{f(x). x \in A\}$
by (*blast intro: RepFunI*)

lemma *RepFunE [elim!]*:

$$\llbracket b \in \{f(x). x \in A\};$$

$$\bigwedge x. \llbracket x \in A; b=f(x) \rrbracket \Longrightarrow P \rrbracket \Longrightarrow$$

$$P$$
by (*simp add: RepFun-def Replace-iff, blast*)

lemma *RepFun-cong [cong]*:

$$\llbracket A=B; \bigwedge x. x \in B \Longrightarrow f(x)=g(x) \rrbracket \Longrightarrow \text{RepFun}(A,f) = \text{RepFun}(B,g)$$
by (*simp add: RepFun-def*)

lemma *RepFun-iff [simp]*: $b \in \{f(x). x \in A\} \longleftrightarrow (\exists x \in A. b=f(x))$
by (*unfold Bex-def, blast*)

lemma *triv-RepFun [simp]*: $\{x. x \in A\} = A$
by *blast*

1.18 Rules for Collect – forming a subset by separation

lemma *separation [simp]*: $a \in \{x \in A. P(x)\} \longleftrightarrow a \in A \wedge P(a)$
by (*auto simp: Collect-def*)

lemma *CollectI* [*intro!*]: $\llbracket a \in A; P(a) \rrbracket \implies a \in \{x \in A. P(x)\}$
by *simp*

lemma *CollectE* [*elim!*]: $\llbracket a \in \{x \in A. P(x)\}; \llbracket a \in A; P(a) \rrbracket \implies R \rrbracket \implies R$
by *simp*

lemma *CollectD1*: $a \in \{x \in A. P(x)\} \implies a \in A$ **and** *CollectD2*: $a \in \{x \in A. P(x)\} \implies P(a)$
by *auto*

lemma *Collect-cong* [*cong*]:
 $\llbracket A=B; \bigwedge x. x \in B \implies P(x) \longleftrightarrow Q(x) \rrbracket$
 $\implies \text{Collect}(A, \lambda x. P(x)) = \text{Collect}(B, \lambda x. Q(x))$
by (*simp add: Collect-def*)

1.19 Rules for Unions

declare *Union-iff* [*simp*]

lemma *UnionI* [*intro*]: $\llbracket B: C; A: B \rrbracket \implies A: \bigcup(C)$
by *auto*

lemma *UnionE* [*elim!*]: $\llbracket A \in \bigcup(C); \bigwedge B. \llbracket A: B; B: C \rrbracket \implies R \rrbracket \implies R$
by *auto*

1.20 Rules for Unions of families

lemma *UN-iff* [*simp*]: $b \in (\bigcup x \in A. B(x)) \longleftrightarrow (\exists x \in A. b \in B(x))$
by *blast*

lemma *UN-I*: $\llbracket a: A; b: B(a) \rrbracket \implies b: (\bigcup x \in A. B(x))$
by *force*

lemma *UN-E* [*elim!*]:
 $\llbracket b \in (\bigcup x \in A. B(x)); \bigwedge x. \llbracket x: A; b: B(x) \rrbracket \implies R \rrbracket \implies R$
by *blast*

lemma *UN-cong*:
 $\llbracket A=B; \bigwedge x. x \in B \implies C(x)=D(x) \rrbracket \implies (\bigcup x \in A. C(x)) = (\bigcup x \in B. D(x))$
by *simp*

1.21 Rules for the empty set

lemma *not-mem-empty* [*simp*]: $a \notin 0$
using *foundation* **by** (*best dest: equalityD2*)

lemmas *emptyE* [*elim!*] = *not-mem-empty* [*THEN notE*]

lemma *empty-subsetI* [*simp*]: $0 \subseteq A$
by *blast*

lemma *equals0I*: $\llbracket \bigwedge y. y \in A \implies \text{False} \rrbracket \implies A = 0$
by *blast*

lemma *equals0D* [*dest*]: $A = 0 \implies a \notin A$
by *blast*

declare *sym* [*THEN equals0D, dest*]

lemma *not-emptyI*: $a \in A \implies A \neq 0$
by *blast*

lemma *not-emptyE*: $\llbracket A \neq 0; \bigwedge x. x \in A \implies R \rrbracket \implies R$
by *blast*

1.22 Rules for Inter

lemma *Inter-iff*: $A \in \bigcap (C) \longleftrightarrow (\forall x \in C. A: x) \wedge C \neq 0$
by (*force simp: Inter-def*)

lemma *InterI* [*intro!*]:
 $\llbracket \bigwedge x. x: C \implies A: x; C \neq 0 \rrbracket \implies A \in \bigcap (C)$
by (*simp add: Inter-iff*)

lemma *InterD* [*elim, Pure.elim*]: $\llbracket A \in \bigcap (C); B \in C \rrbracket \implies A \in B$
by (*force simp: Inter-def*)

lemma *InterE* [*elim*]:
 $\llbracket A \in \bigcap (C); B \notin C \implies R; A \in B \implies R \rrbracket \implies R$
by (*auto simp: Inter-def*)

1.23 Rules for Intersections of families

lemma *INT-iff*: $b \in (\bigcap x \in A. B(x)) \longleftrightarrow (\forall x \in A. b \in B(x)) \wedge A \neq 0$
by (*force simp add: Inter-def*)

lemma *INT-I*: $\llbracket \bigwedge x. x: A \implies b: B(x); A \neq 0 \rrbracket \implies b: (\bigcap x \in A. B(x))$
by *blast*

lemma *INT-E*: $\llbracket b \in (\bigcap x \in A. B(x)); a: A \rrbracket \implies b \in B(a)$
by *blast*

lemma *INT-cong*:
 $\llbracket A=B; \bigwedge x. x \in B \implies C(x)=D(x) \rrbracket \implies (\bigcap_{x \in A}. C(x)) = (\bigcap_{x \in B}. D(x))$
by *simp*

1.24 Rules for Powersets

lemma *PowI*: $A \subseteq B \implies A \in \text{Pow}(B)$
by (*erule Pow-iff [THEN iffD2]*)

lemma *PowD*: $A \in \text{Pow}(B) \implies A \subseteq B$
by (*erule Pow-iff [THEN iffD1]*)

declare *Pow-iff* [*iff*]

lemmas *Pow-bottom* = *empty-subsetI [THEN PowI]* — $0 \in \text{Pow}(B)$

lemmas *Pow-top* = *subset-refl [THEN PowI]* — $A \in \text{Pow}(A)$

1.25 Cantor's Theorem: There is no surjection from a set to its powerset.

lemma *cantor*: $\exists S \in \text{Pow}(A). \forall x \in A. b(x) \neq S$
by (*best elim!*: *equalityCE del: ReplaceI RepFun-eqI*)

end

2 Unordered Pairs

theory *upair*
imports *ZF-Base*
keywords *print-tcset :: diag*
begin

ML-file $\langle \text{Tools/typechk.ML} \rangle$

2.1 Unordered Pairs: constant *Upair*

lemma *Upair-iff* [*simp*]: $c \in \text{Upair}(a,b) \longleftrightarrow (c=a \mid c=b)$
by (*unfold Upair-def, blast*)

lemma *UpairI1*: $a \in \text{Upair}(a,b)$
by *simp*

lemma *UpairI2*: $b \in \text{Upair}(a,b)$
by *simp*

lemma *UpairE*: $\llbracket a \in \text{Upair}(b,c); a=b \implies P; a=c \implies P \rrbracket \implies P$
by (*simp, blast*)

2.2 Rules for Binary Union, Defined via *Upair*

lemma *Un-iff* [*simp*]: $c \in A \cup B \longleftrightarrow (c \in A \mid c \in B)$
apply (*simp add: Un-def*)
apply (*blast intro: UpairI1 UpairI2 elim: UpairE*)
done

lemma *UnI1*: $c \in A \implies c \in A \cup B$
by *simp*

lemma *UnI2*: $c \in B \implies c \in A \cup B$
by *simp*

declare *UnI1* [*elim?*] *UnI2* [*elim?*]

lemma *UnE* [*elim!*]: $\llbracket c \in A \cup B; c \in A \implies P; c \in B \implies P \rrbracket \implies P$
by (*simp, blast*)

lemma *UnE'*: $\llbracket c \in A \cup B; c \in A \implies P; \llbracket c \in B; c \notin A \rrbracket \implies P \rrbracket \implies P$
by (*simp, blast*)

lemma *UnCI* [*intro!*]: $(c \notin B \implies c \in A) \implies c \in A \cup B$
by (*simp, blast*)

2.3 Rules for Binary Intersection, Defined via *Upair*

lemma *Int-iff* [*simp*]: $c \in A \cap B \longleftrightarrow (c \in A \wedge c \in B)$
unfolding *Int-def*
apply (*blast intro: UpairI1 UpairI2 elim: UpairE*)
done

lemma *IntI* [*intro!*]: $\llbracket c \in A; c \in B \rrbracket \implies c \in A \cap B$
by *simp*

lemma *IntD1*: $c \in A \cap B \implies c \in A$
by *simp*

lemma *IntD2*: $c \in A \cap B \implies c \in B$
by *simp*

lemma *IntE* [*elim!*]: $\llbracket c \in A \cap B; \llbracket c \in A; c \in B \rrbracket \implies P \rrbracket \implies P$
by *simp*

2.4 Rules for Set Difference, Defined via *Upair*

lemma *Diff-iff* [*simp*]: $c \in A - B \longleftrightarrow (c \in A \wedge c \notin B)$
by (*unfold Diff-def, blast*)

lemma *DiffI* [*intro!*]: $\llbracket c \in A; c \notin B \rrbracket \implies c \in A - B$
by *simp*

lemma *DiffD1*: $c \in A - B \implies c \in A$
by *simp*

lemma *DiffD2*: $c \in A - B \implies c \notin B$
by *simp*

lemma *DiffE* [*elim!*]: $\llbracket c \in A - B; \llbracket c \in A; c \notin B \rrbracket \implies P \rrbracket \implies P$
by *simp*

2.5 Rules for *cons*

lemma *cons-iff* [*simp*]: $a \in \text{cons}(b, A) \longleftrightarrow (a=b \mid a \in A)$
unfolding *cons-def*
apply (*blast intro: UpairI1 UpairI2 elim: UpairE*)
done

lemma *consI1* [*simp, TC*]: $a \in \text{cons}(a, B)$
by *simp*

lemma *consI2*: $a \in B \implies a \in \text{cons}(b, B)$
by *simp*

lemma *consE* [*elim!*]: $\llbracket a \in \text{cons}(b, A); a=b \implies P; a \in A \implies P \rrbracket \implies P$
by (*simp, blast*)

lemma *consE'*:
 $\llbracket a \in \text{cons}(b, A); a=b \implies P; \llbracket a \in A; a \neq b \rrbracket \implies P \rrbracket \implies P$
by (*simp, blast*)

lemma *consCI* [*intro!*]: $(a \notin B \implies a=b) \implies a \in \text{cons}(b, B)$
by (*simp, blast*)

lemma *cons-not-0* [*simp*]: $\text{cons}(a, B) \neq 0$
by (*blast elim: equalityE*)

lemmas *cons-neq-0* = *cons-not-0* [*THEN notE*]

declare *cons-not-0* [*THEN not-sym, simp*]

2.6 Singletons

lemma *singleton-iff*: $a \in \{b\} \longleftrightarrow a=b$
by *simp*

lemma *singletonI* [intro!]: $a \in \{a\}$
by (*rule consI1*)

lemmas *singletonE* = *singleton-iff* [THEN *iffD1*, *elim-format*, *elim!*]

2.7 Descriptions

lemma *the-equality* [intro]:
 $\llbracket P(a); \bigwedge x. P(x) \implies x=a \rrbracket \implies (THE\ x. P(x)) = a$
unfolding *the-def*
apply (*fast dest: subst*)
done

lemma *the-equality2*: $\llbracket \exists !x. P(x); P(a) \rrbracket \implies (THE\ x. P(x)) = a$
by *blast*

lemma *theI*: $\exists !x. P(x) \implies P(THE\ x. P(x))$
apply (*erule exIE*)
apply (*subst the-equality*)
apply (*blast+*)
done

lemma *the-0*: $\neg (\exists !x. P(x)) \implies (THE\ x. P(x))=0$
unfolding *the-def*
apply (*blast elim!: ReplaceE*)
done

lemma *theI2*:
assumes *p1*: $\neg Q(0) \implies \exists !x. P(x)$
and *p2*: $\bigwedge x. P(x) \implies Q(x)$
shows $Q(THE\ x. P(x))$
apply (*rule classical*)
apply (*rule p2*)
apply (*rule theI*)
apply (*rule classical*)
apply (*rule p1*)
apply (*erule the-0 [THEN subst], assumption*)
done

lemma *the-eq-trivial* [simp]: $(THE\ x. x = a) = a$
by *blast*

lemma *the-eq-trivial2* [simp]: $(THE\ x. a = x) = a$

by *blast*

2.8 Conditional Terms: *if-then-else*

lemma *if-true* [*simp*]: (if True then a else b) = a
by (unfold *if-def*, *blast*)

lemma *if-false* [*simp*]: (if False then a else b) = b
by (unfold *if-def*, *blast*)

lemma *if-cong*:
 $\llbracket P \longleftrightarrow Q; \quad Q \Longrightarrow a=c; \quad \neg Q \Longrightarrow b=d \rrbracket$
 $\Longrightarrow (if\ P\ then\ a\ else\ b) = (if\ Q\ then\ c\ else\ d)$
by (*simp add: if-def cong add: conj-cong*)

lemma *if-weak-cong*: $P \longleftrightarrow Q \Longrightarrow (if\ P\ then\ x\ else\ y) = (if\ Q\ then\ x\ else\ y)$
by *simp*

lemma *if-P*: $P \Longrightarrow (if\ P\ then\ a\ else\ b) = a$
by (unfold *if-def*, *blast*)

lemma *if-not-P*: $\neg P \Longrightarrow (if\ P\ then\ a\ else\ b) = b$
by (unfold *if-def*, *blast*)

lemma *split-if* [*split*]:
 $P(if\ Q\ then\ x\ else\ y) \longleftrightarrow ((Q \longrightarrow P(x)) \wedge (\neg Q \longrightarrow P(y)))$
by (*case-tac Q, simp-all*)

lemmas *split-if-eq1* = *split-if* [of $\lambda x. x = b$] **for** *b*
lemmas *split-if-eq2* = *split-if* [of $\lambda x. a = x$] **for** *a*

lemmas *split-if-mem1* = *split-if* [of $\lambda x. x \in b$] **for** *b*
lemmas *split-if-mem2* = *split-if* [of $\lambda x. a \in x$] **for** *a*

lemmas *split-ifs* = *split-if-eq1 split-if-eq2 split-if-mem1 split-if-mem2*

lemma *if-iff*: $a: (if\ P\ then\ x\ else\ y) \longleftrightarrow P \wedge a \in x \mid \neg P \wedge a \in y$
by *simp*

lemma *if-type* [*TC*]:
 $\llbracket P \Longrightarrow a \in A; \quad \neg P \Longrightarrow b \in A \rrbracket \Longrightarrow (if\ P\ then\ a\ else\ b): A$
by *simp*

lemma *split-if-asm*: $P(\text{if } Q \text{ then } x \text{ else } y) \longleftrightarrow (\neg((Q \wedge \neg P(x)) \mid (\neg Q \wedge \neg P(y))))$
by *simp*

lemmas *if-splits* = *split-if split-if-asm*

2.9 Consequences of Foundation

lemma *mem-asym*: $\llbracket a \in b; \neg P \implies b \in a \rrbracket \implies P$
apply (*rule classical*)
apply (*rule-tac A1 = {a,b} in foundation [THEN disjE]*)
apply (*blast elim!: equalityE*)
done

lemma *mem-irrefl*: $a \in a \implies P$
by (*blast intro: mem-asym*)

lemma *mem-not-refl*: $a \notin a$
apply (*rule notI*)
apply (*erule mem-irrefl*)
done

lemma *mem-imp-not-eq*: $a \in A \implies a \neq A$
by (*blast elim!: mem-irrefl*)

lemma *eq-imp-not-mem*: $a=A \implies a \notin A$
by (*blast intro: elim: mem-irrefl*)

2.10 Rules for Successor

lemma *succ-iff*: $i \in \text{succ}(j) \longleftrightarrow i=j \mid i \in j$
by (*unfold succ-def, blast*)

lemma *succI1* [*simp*]: $i \in \text{succ}(i)$
by (*simp add: succ-iff*)

lemma *succI2*: $i \in j \implies i \in \text{succ}(j)$
by (*simp add: succ-iff*)

lemma *succE* [*elim!*]:
 $\llbracket i \in \text{succ}(j); i=j \implies P; i \in j \implies P \rrbracket \implies P$
apply (*simp add: succ-iff, blast*)
done

lemma *succCI* [intro!]: $(i \notin j \implies i=j) \implies i \in \text{succ}(j)$
by (*simp add: succ-iff, blast*)

lemma *succ-not-0* [simp]: $\text{succ}(n) \neq 0$
by (*blast elim!: equalityE*)

lemmas *succ-neq-0 = succ-not-0* [THEN *notE, elim!*]

declare *succ-not-0* [THEN *not-sym, simp*]
declare *sym* [THEN *succ-neq-0, elim!*]

lemmas *succ-subsetD = succI1* [THEN [2] *subsetD*]

lemmas *succ-neq-self = succI1* [THEN *mem-imp-not-eq, THEN not-sym*]

lemma *succ-inject-iff* [simp]: $\text{succ}(m) = \text{succ}(n) \longleftrightarrow m=n$
by (*blast elim: mem-asym elim!: equalityE*)

lemmas *succ-inject = succ-inject-iff* [THEN *iffD1, dest!*]

2.11 Miniscoping of the Bounded Universal Quantifier

lemma *ball-simps1*:

$(\forall x \in A. P(x) \wedge Q) \longleftrightarrow (\forall x \in A. P(x)) \wedge (A=0 \mid Q)$
 $(\forall x \in A. P(x) \mid Q) \longleftrightarrow ((\forall x \in A. P(x)) \mid Q)$
 $(\forall x \in A. P(x) \longrightarrow Q) \longleftrightarrow ((\exists x \in A. P(x)) \longrightarrow Q)$
 $(\neg(\forall x \in A. P(x))) \longleftrightarrow (\exists x \in A. \neg P(x))$
 $(\forall x \in 0. P(x)) \longleftrightarrow \text{True}$
 $(\forall x \in \text{succ}(i). P(x)) \longleftrightarrow P(i) \wedge (\forall x \in i. P(x))$
 $(\forall x \in \text{cons}(a, B). P(x)) \longleftrightarrow P(a) \wedge (\forall x \in B. P(x))$
 $(\forall x \in \text{RepFun}(A, f). P(x)) \longleftrightarrow (\forall y \in A. P(f(y)))$
 $(\forall x \in \bigcup (A). P(x)) \longleftrightarrow (\forall y \in A. \forall x \in y. P(x))$

by *blast+*

lemma *ball-simps2*:

$(\forall x \in A. P \wedge Q(x)) \longleftrightarrow (A=0 \mid P) \wedge (\forall x \in A. Q(x))$
 $(\forall x \in A. P \mid Q(x)) \longleftrightarrow (P \mid (\forall x \in A. Q(x)))$
 $(\forall x \in A. P \longrightarrow Q(x)) \longleftrightarrow (P \longrightarrow (\forall x \in A. Q(x)))$

by *blast+*

lemma *ball-simps3*:

$(\forall x \in \text{Collect}(A, Q). P(x)) \longleftrightarrow (\forall x \in A. Q(x) \longrightarrow P(x))$

by *blast+*

lemmas *ball-simps* [simp] = *ball-simps1 ball-simps2 ball-simps3*

lemma *ball-conj-distrib*:

$$(\forall x \in A. P(x) \wedge Q(x)) \longleftrightarrow ((\forall x \in A. P(x)) \wedge (\forall x \in A. Q(x)))$$

by *blast*

2.12 Miniscoping of the Bounded Existential Quantifier

lemma *bex-simps1*:

$$\begin{aligned} (\exists x \in A. P(x) \wedge Q) &\longleftrightarrow ((\exists x \in A. P(x)) \wedge Q) \\ (\exists x \in A. P(x) \mid Q) &\longleftrightarrow (\exists x \in A. P(x)) \mid (A \neq 0 \wedge Q) \\ (\exists x \in A. P(x) \longrightarrow Q) &\longleftrightarrow ((\forall x \in A. P(x)) \longrightarrow (A \neq 0 \wedge Q)) \\ (\exists x \in 0. P(x)) &\longleftrightarrow \text{False} \\ (\exists x \in \text{succ}(i). P(x)) &\longleftrightarrow P(i) \mid (\exists x \in i. P(x)) \\ (\exists x \in \text{cons}(a, B). P(x)) &\longleftrightarrow P(a) \mid (\exists x \in B. P(x)) \\ (\exists x \in \text{RepFun}(A, f). P(x)) &\longleftrightarrow (\exists y \in A. P(f(y))) \\ (\exists x \in \bigcup(A). P(x)) &\longleftrightarrow (\exists y \in A. \exists x \in y. P(x)) \\ (\neg(\exists x \in A. P(x))) &\longleftrightarrow (\forall x \in A. \neg P(x)) \end{aligned}$$

by *blast+*

lemma *bex-simps2*:

$$\begin{aligned} (\exists x \in A. P \wedge Q(x)) &\longleftrightarrow (P \wedge (\exists x \in A. Q(x))) \\ (\exists x \in A. P \mid Q(x)) &\longleftrightarrow (A \neq 0 \wedge P) \mid (\exists x \in A. Q(x)) \\ (\exists x \in A. P \longrightarrow Q(x)) &\longleftrightarrow ((A = 0 \mid P) \longrightarrow (\exists x \in A. Q(x))) \end{aligned}$$

by *blast+*

lemma *bex-simps3*:

$$(\exists x \in \text{Collect}(A, Q). P(x)) \longleftrightarrow (\exists x \in A. Q(x) \wedge P(x))$$

by *blast*

lemmas *bex-simps* [*simp*] = *bex-simps1 bex-simps2 bex-simps3*

lemma *bex-disj-distrib*:

$$(\exists x \in A. P(x) \mid Q(x)) \longleftrightarrow ((\exists x \in A. P(x)) \mid (\exists x \in A. Q(x)))$$

by *blast*

lemma *bex-triv-one-point1* [*simp*]: $(\exists x \in A. x = a) \longleftrightarrow (a \in A)$

by *blast*

lemma *bex-triv-one-point2* [*simp*]: $(\exists x \in A. a = x) \longleftrightarrow (a \in A)$

by *blast*

lemma *bex-one-point1* [*simp*]: $(\exists x \in A. x = a \wedge P(x)) \longleftrightarrow (a \in A \wedge P(a))$

by *blast*

lemma *bex-one-point2* [*simp*]: $(\exists x \in A. a = x \wedge P(x)) \longleftrightarrow (a \in A \wedge P(a))$

by *blast*

lemma *ball-one-point1* [simp]: $(\forall x \in A. x=a \longrightarrow P(x)) \longleftrightarrow (a \in A \longrightarrow P(a))$
by *blast*

lemma *ball-one-point2* [simp]: $(\forall x \in A. a=x \longrightarrow P(x)) \longleftrightarrow (a \in A \longrightarrow P(a))$
by *blast*

2.13 Miniscoping of the Replacement Operator

These cover both *Replace* and *Collect*

lemma *Rep-simps* [simp]:
 $\{x. y \in 0, R(x,y)\} = 0$
 $\{x \in 0. P(x)\} = 0$
 $\{x \in A. Q\} = (\text{if } Q \text{ then } A \text{ else } 0)$
 $\text{RepFun}(0,f) = 0$
 $\text{RepFun}(\text{succ}(i),f) = \text{cons}(f(i), \text{RepFun}(i,f))$
 $\text{RepFun}(\text{cons}(a,B),f) = \text{cons}(f(a), \text{RepFun}(B,f))$
by (*simp-all*, *blast+*)

2.14 Miniscoping of Unions

lemma *UN-simps1*:
 $(\bigcup x \in C. \text{cons}(a, B(x))) = (\text{if } C=0 \text{ then } 0 \text{ else } \text{cons}(a, \bigcup x \in C. B(x)))$
 $(\bigcup x \in C. A(x) \cup B') = (\text{if } C=0 \text{ then } 0 \text{ else } (\bigcup x \in C. A(x)) \cup B')$
 $(\bigcup x \in C. A' \cup B(x)) = (\text{if } C=0 \text{ then } 0 \text{ else } A' \cup (\bigcup x \in C. B(x)))$
 $(\bigcup x \in C. A(x) \cap B') = ((\bigcup x \in C. A(x)) \cap B')$
 $(\bigcup x \in C. A' \cap B(x)) = (A' \cap (\bigcup x \in C. B(x)))$
 $(\bigcup x \in C. A(x) - B') = ((\bigcup x \in C. A(x)) - B')$
 $(\bigcup x \in C. A' - B(x)) = (\text{if } C=0 \text{ then } 0 \text{ else } A' - (\bigcap x \in C. B(x)))$
apply (*simp-all add: Inter-def*)
apply (*blast intro!: equalityI*)
done

lemma *UN-simps2*:
 $(\bigcup x \in \bigcup (A). B(x)) = (\bigcup y \in A. \bigcup x \in y. B(x))$
 $(\bigcup z \in (\bigcup x \in A. B(x)). C(z)) = (\bigcup x \in A. \bigcup z \in B(x). C(z))$
 $(\bigcup x \in \text{RepFun}(A,f). B(x)) = (\bigcup a \in A. B(f(a)))$
by *blast+*

lemmas *UN-simps* [simp] = *UN-simps1 UN-simps2*

Opposite of miniscoping: pull the operator out

lemma *UN-extend-simps1*:
 $(\bigcup x \in C. A(x)) \cup B = (\text{if } C=0 \text{ then } B \text{ else } (\bigcup x \in C. A(x) \cup B))$
 $((\bigcup x \in C. A(x)) \cap B) = (\bigcup x \in C. A(x) \cap B)$
 $((\bigcup x \in C. A(x)) - B) = (\bigcup x \in C. A(x) - B)$
apply *simp-all*
apply *blast+*
done

lemma *UN-extend-simps2*:

$$\begin{aligned}
& \text{cons}(a, \bigcup_{x \in C}. B(x)) = (\text{if } C=0 \text{ then } \{a\} \text{ else } (\bigcup_{x \in C}. \text{cons}(a, B(x)))) \\
& A \cup (\bigcup_{x \in C}. B(x)) = (\text{if } C=0 \text{ then } A \text{ else } (\bigcup_{x \in C}. A \cup B(x))) \\
& (A \cap (\bigcup_{x \in C}. B(x))) = (\bigcup_{x \in C}. A \cap B(x)) \\
& A - (\bigcap_{x \in C}. B(x)) = (\text{if } C=0 \text{ then } A \text{ else } (\bigcup_{x \in C}. A - B(x))) \\
& (\bigcup_{y \in A}. \bigcup_{x \in y}. B(x)) = (\bigcup_{x \in \bigcup(A)}. B(x)) \\
& (\bigcup_{a \in A}. B(f(a))) = (\bigcup_{x \in \text{RepFun}(A, f)}. B(x))
\end{aligned}$$

apply (*simp-all add: Inter-def*)

apply (*blast intro!: equalityI*) +

done

lemma *UN-UN-extend*:

$$(\bigcup_{x \in A}. \bigcup_{z \in B(x)}. C(z)) = (\bigcup_{z \in (\bigcup_{x \in A}. B(x))}. C(z))$$

by *blast*

lemmas *UN-extend-simps = UN-extend-simps1 UN-extend-simps2 UN-UN-extend*

2.15 Miniscoping of Intersections

lemma *INT-simps1*:

$$\begin{aligned}
& (\bigcap_{x \in C}. A(x) \cap B) = (\bigcap_{x \in C}. A(x)) \cap B \\
& (\bigcap_{x \in C}. A(x) - B) = (\bigcap_{x \in C}. A(x)) - B \\
& (\bigcap_{x \in C}. A(x) \cup B) = (\text{if } C=0 \text{ then } 0 \text{ else } (\bigcap_{x \in C}. A(x)) \cup B)
\end{aligned}$$

by (*simp-all add: Inter-def, blast+*)

lemma *INT-simps2*:

$$\begin{aligned}
& (\bigcap_{x \in C}. A \cap B(x)) = A \cap (\bigcap_{x \in C}. B(x)) \\
& (\bigcap_{x \in C}. A - B(x)) = (\text{if } C=0 \text{ then } 0 \text{ else } A - (\bigcup_{x \in C}. B(x))) \\
& (\bigcap_{x \in C}. \text{cons}(a, B(x))) = (\text{if } C=0 \text{ then } 0 \text{ else } \text{cons}(a, \bigcap_{x \in C}. B(x))) \\
& (\bigcap_{x \in C}. A \cup B(x)) = (\text{if } C=0 \text{ then } 0 \text{ else } A \cup (\bigcap_{x \in C}. B(x)))
\end{aligned}$$

apply (*simp-all add: Inter-def*)

apply (*blast intro!: equalityI*) +

done

lemmas *INT-simps [simp] = INT-simps1 INT-simps2*

Opposite of miniscoping: pull the operator out

lemma *INT-extend-simps1*:

$$\begin{aligned}
& (\bigcap_{x \in C}. A(x)) \cap B = (\bigcap_{x \in C}. A(x) \cap B) \\
& (\bigcap_{x \in C}. A(x)) - B = (\bigcap_{x \in C}. A(x) - B) \\
& (\bigcap_{x \in C}. A(x)) \cup B = (\text{if } C=0 \text{ then } B \text{ else } (\bigcap_{x \in C}. A(x) \cup B))
\end{aligned}$$

apply (*simp-all add: Inter-def, blast+*)

done

lemma *INT-extend-simps2*:

$$\begin{aligned}
& A \cap (\bigcap_{x \in C}. B(x)) = (\bigcap_{x \in C}. A \cap B(x)) \\
& A - (\bigcup_{x \in C}. B(x)) = (\text{if } C=0 \text{ then } A \text{ else } (\bigcap_{x \in C}. A - B(x))) \\
& \text{cons}(a, \bigcap_{x \in C}. B(x)) = (\text{if } C=0 \text{ then } \{a\} \text{ else } (\bigcap_{x \in C}. \text{cons}(a, B(x)))) \\
& A \cup (\bigcap_{x \in C}. B(x)) = (\text{if } C=0 \text{ then } A \text{ else } (\bigcap_{x \in C}. A \cup B(x)))
\end{aligned}$$

```

apply (simp-all add: Inter-def)
apply (blast intro!: equalityI)+
done

```

```

lemmas INT-extend-simps = INT-extend-simps1 INT-extend-simps2

```

2.16 Other simprules

```

lemma misc-simps [simp]:
   $0 \cup A = A$ 
   $A \cup 0 = A$ 
   $0 \cap A = 0$ 
   $A \cap 0 = 0$ 
   $0 - A = 0$ 
   $A - 0 = A$ 
   $\bigcup (0) = 0$ 
   $\bigcup (\text{cons}(b,A)) = b \cup \bigcup (A)$ 
   $\bigcap (\{b\}) = b$ 
by blast+

end

```

3 Ordered Pairs

```

theory pair imports upair
begin

```

```

ML-file <simpdata.ML>

```

```

setup <
  map-theory-simpset
  (Simplifier.set-mksimps (fn ctxt => map mk-eq o ZF-atomize o Variable.gen-all
    ctxt)
    #> Simplifier.add-cong @{thm if-weak-cong})
  >

```

```

ML <val ZF-ss = simpset-of context>

```

```

simproc-setup defined-Bex ( $\exists x \in A. P(x) \wedge Q(x)$ ) = <
  K (Quantifier1.rearrange-Bex (fn ctxt => unfold-tac ctxt @{thms Bex-def}))
  >

```

```

simproc-setup defined-Ball ( $\forall x \in A. P(x) \longrightarrow Q(x)$ ) = <
  K (Quantifier1.rearrange-Ball (fn ctxt => unfold-tac ctxt @{thms Ball-def}))
  >

```

```

lemma singleton-eq-iff [iff]:  $\{a\} = \{b\} \longleftrightarrow a=b$ 
by (rule extension [THEN iff-trans], blast)

lemma doubleton-eq-iff:  $\{a,b\} = \{c,d\} \longleftrightarrow (a=c \wedge b=d) \mid (a=d \wedge b=c)$ 
by (rule extension [THEN iff-trans], blast)

lemma Pair-iff [simp]:  $\langle a,b \rangle = \langle c,d \rangle \longleftrightarrow a=c \wedge b=d$ 
by (simp add: Pair-def doubleton-eq-iff, blast)

lemmas Pair-inject = Pair-iff [THEN iffD1, THEN conjE, elim!]

lemmas Pair-inject1 = Pair-iff [THEN iffD1, THEN conjunct1]
lemmas Pair-inject2 = Pair-iff [THEN iffD1, THEN conjunct2]

lemma Pair-not-0:  $\langle a,b \rangle \neq 0$ 
  unfolding Pair-def
apply (blast elim: equalityE)
done

lemmas Pair-neq-0 = Pair-not-0 [THEN notE, elim!]

declare sym [THEN Pair-neq-0, elim!]

lemma Pair-neq-fst:  $\langle a,b \rangle = a \implies P$ 
proof (unfold Pair-def)
  assume eq:  $\{\{a, a\}, \{a, b\}\} = a$ 
  have  $\{a, a\} \in \{\{a, a\}, \{a, b\}\}$  by (rule consI1)
  hence  $\{a, a\} \in a$  by (simp add: eq)
  moreover have  $a \in \{a, a\}$  by (rule consI1)
  ultimately show  $P$  by (rule mem-asym)
qed

lemma Pair-neq-snd:  $\langle a,b \rangle = b \implies P$ 
proof (unfold Pair-def)
  assume eq:  $\{\{a, a\}, \{a, b\}\} = b$ 
  have  $\{a, b\} \in \{\{a, a\}, \{a, b\}\}$  by blast
  hence  $\{a, b\} \in b$  by (simp add: eq)
  moreover have  $b \in \{a, b\}$  by blast
  ultimately show  $P$  by (rule mem-asym)
qed

```

3.1 Sigma: Disjoint Union of a Family of Sets

Generalizes Cartesian product

```

lemma Sigma-iff [simp]:  $\langle a,b \rangle : \text{Sigma}(A,B) \longleftrightarrow a \in A \wedge b \in B(a)$ 
by (simp add: Sigma-def)

lemma SigmaI [TC,intro!]:  $\llbracket a \in A; b \in B(a) \rrbracket \implies \langle a,b \rangle \in \text{Sigma}(A,B)$ 
by simp

```

lemmas $SigmaD1 = Sigma\text{-iff}$ [THEN $iffD1$, THEN $conjunct1$]
lemmas $SigmaD2 = Sigma\text{-iff}$ [THEN $iffD1$, THEN $conjunct2$]

lemma $SigmaE$ [elim!]:
 $\llbracket c \in Sigma(A,B);$
 $\bigwedge x y. \llbracket x \in A; y \in B(x); c = \langle x, y \rangle \rrbracket \implies P$
 $\rrbracket \implies P$
by (unfold $Sigma\text{-def}$, blast)

lemma $SigmaE2$ [elim!]:
 $\llbracket \langle a, b \rangle \in Sigma(A,B);$
 $\llbracket a \in A; b \in B(a) \rrbracket \implies P$
 $\rrbracket \implies P$
by (unfold $Sigma\text{-def}$, blast)

lemma $Sigma\text{-cong}$:
 $\llbracket A = A'; \bigwedge x. x \in A' \implies B(x) = B'(x) \rrbracket \implies$
 $Sigma(A,B) = Sigma(A',B')$
by (simp add: $Sigma\text{-def}$)

lemma $Sigma\text{-empty1}$ [simp]: $Sigma(0,B) = 0$
by blast

lemma $Sigma\text{-empty2}$ [simp]: $A * 0 = 0$
by blast

lemma $Sigma\text{-empty-iff}$: $A * B = 0 \longleftrightarrow A = 0 \mid B = 0$
by blast

3.2 Projections fst and snd

lemma $fst\text{-conv}$ [simp]: $fst(\langle a, b \rangle) = a$
by (simp add: $fst\text{-def}$)

lemma $snd\text{-conv}$ [simp]: $snd(\langle a, b \rangle) = b$
by (simp add: $snd\text{-def}$)

lemma $fst\text{-type}$ [TC]: $p \in Sigma(A,B) \implies fst(p) \in A$
by auto

lemma $snd\text{-type}$ [TC]: $p \in Sigma(A,B) \implies snd(p) \in B(fst(p))$
by auto

lemma $Pair\text{-fst-snd-eq}$: $a \in Sigma(A,B) \implies \langle fst(a), snd(a) \rangle = a$
by auto

3.3 The Eliminator, *split*

lemma *split* [*simp*]: $\text{split}(\lambda x y. c(x,y), \langle a,b \rangle) \equiv c(a,b)$
by (*simp add: split-def*)

lemma *split-type* [*TC*]:
 $\llbracket p \in \text{Sigma}(A,B);$
 $\bigwedge x y. \llbracket x \in A; y \in B(x) \rrbracket \implies c(x,y):C(\langle x,y \rangle)$
 $\rrbracket \implies \text{split}(\lambda x y. c(x,y), p) \in C(p)$
by (*erule SigmaE, auto*)

lemma *expand-split*:
 $u \in A*B \implies$
 $R(\text{split}(c,u)) \longleftrightarrow (\forall x \in A. \forall y \in B. u = \langle x,y \rangle \longrightarrow R(c(x,y)))$
by (*auto simp add: split-def*)

3.4 A version of *split* for Formulae: Result Type *o*

lemma *splitI*: $R(a,b) \implies \text{split}(R, \langle a,b \rangle)$
by (*simp add: split-def*)

lemma *splitE*:
 $\llbracket \text{split}(R,z); z \in \text{Sigma}(A,B);$
 $\bigwedge x y. \llbracket z = \langle x,y \rangle; R(x,y) \rrbracket \implies P$
 $\rrbracket \implies P$
by (*auto simp add: split-def*)

lemma *splitD*: $\text{split}(R, \langle a,b \rangle) \implies R(a,b)$
by (*simp add: split-def*)

Complex rules for Sigma.

lemma *split-paired-Bex-Sigma* [*simp*]:
 $(\exists z \in \text{Sigma}(A,B). P(z)) \longleftrightarrow (\exists x \in A. \exists y \in B(x). P(\langle x,y \rangle))$
by *blast*

lemma *split-paired-Ball-Sigma* [*simp*]:
 $(\forall z \in \text{Sigma}(A,B). P(z)) \longleftrightarrow (\forall x \in A. \forall y \in B(x). P(\langle x,y \rangle))$
by *blast*

end

4 Basic Equalities and Inclusions

theory *equalities* **imports** *pair* **begin**

These cover union, intersection, converse, domain, range, etc. Philippe de Groote proved many of the inclusions.

lemma *in-mono*: $A \subseteq B \implies x \in A \longrightarrow x \in B$

by *blast*

lemma *the-eq-0* [*simp*]: $(THE\ x.\ False) = 0$
 by (*blast intro: the-0*)

4.1 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*: $(\forall x \in A \cup B. P(x)) \longleftrightarrow (\forall x \in A. P(x)) \wedge (\forall x \in B. P(x))$
 by *blast*

lemma *bex-Un*: $(\exists x \in A \cup B. P(x)) \longleftrightarrow (\exists x \in A. P(x)) \mid (\exists x \in B. P(x))$
 by *blast*

lemma *ball-UN*: $(\forall z \in (\bigcup x \in A. B(x)). P(z)) \longleftrightarrow (\forall x \in A. \forall z \in B(x). P(z))$
 by *blast*

lemma *bex-UN*: $(\exists z \in (\bigcup x \in A. B(x)). P(z)) \longleftrightarrow (\exists x \in A. \exists z \in B(x). P(z))$
 by *blast*

4.2 Converse of a Relation

lemma *converse-iff* [*simp*]: $\langle a, b \rangle \in converse(r) \longleftrightarrow \langle b, a \rangle \in r$
 by (*unfold converse-def, blast*)

lemma *converseI* [*intro!*]: $\langle a, b \rangle \in r \implies \langle b, a \rangle \in converse(r)$
 by (*unfold converse-def, blast*)

lemma *converseD*: $\langle a, b \rangle \in converse(r) \implies \langle b, a \rangle \in r$
 by (*unfold converse-def, blast*)

lemma *converseE* [*elim!*]:

$$\begin{aligned} & \llbracket yx \in converse(r); \\ & \quad \bigwedge x\ y. \llbracket yx = \langle y, x \rangle; \langle x, y \rangle \in r \rrbracket \implies P \rrbracket \\ & \implies P \end{aligned}$$

 by (*unfold converse-def, blast*)

lemma *converse-converse*: $r \subseteq Sigma(A, B) \implies converse(converse(r)) = r$
 by *blast*

lemma *converse-type*: $r \subseteq A * B \implies converse(r) \subseteq B * A$
 by *blast*

lemma *converse-prod* [*simp*]: $converse(A * B) = B * A$
 by *blast*

lemma *converse-empty* [*simp*]: $converse(0) = 0$

by *blast*

lemma *converse-subset-iff*:

$$A \subseteq \text{Sigma}(X, Y) \implies \text{converse}(A) \subseteq \text{converse}(B) \longleftrightarrow A \subseteq B$$

by *blast*

4.3 Finite Set Constructions Using *cons*

lemma *cons-subsetI*: $\llbracket a \in C; B \subseteq C \rrbracket \implies \text{cons}(a, B) \subseteq C$

by *blast*

lemma *subset-consI*: $B \subseteq \text{cons}(a, B)$

by *blast*

lemma *cons-subset-iff* [*iff*]: $\text{cons}(a, B) \subseteq C \longleftrightarrow a \in C \wedge B \subseteq C$

by *blast*

lemmas *cons-subsetE* = *cons-subset-iff* [*THEN iffD1*, *THEN conjE*]

lemma *subset-empty-iff*: $A \subseteq 0 \longleftrightarrow A = 0$

by *blast*

lemma *subset-cons-iff*: $C \subseteq \text{cons}(a, B) \longleftrightarrow C \subseteq B \mid (a \in C \wedge C - \{a\} \subseteq B)$

by *blast*

lemma *cons-eq*: $\{a\} \cup B = \text{cons}(a, B)$

by *blast*

lemma *cons-commute*: $\text{cons}(a, \text{cons}(b, C)) = \text{cons}(b, \text{cons}(a, C))$

by *blast*

lemma *cons-absorb*: $a: B \implies \text{cons}(a, B) = B$

by *blast*

lemma *cons-Diff*: $a: B \implies \text{cons}(a, B - \{a\}) = B$

by *blast*

lemma *Diff-cons-eq*: $\text{cons}(a, B) - C = (\text{if } a \in C \text{ then } B - C \text{ else } \text{cons}(a, B - C))$

by *auto*

lemma *equal-singleton*: $\llbracket a: C; \bigwedge y. y \in C \implies y = b \rrbracket \implies C = \{b\}$

by *blast*

lemma [*simp*]: $\text{cons}(a, \text{cons}(a, B)) = \text{cons}(a, B)$

by *blast*

lemma *singleton-subsetI*: $a \in C \implies \{a\} \subseteq C$
by *blast*

lemma *singleton-subsetD*: $\{a\} \subseteq C \implies a \in C$
by *blast*

lemma *subset-succI*: $i \subseteq \text{succ}(i)$
by *blast*

lemma *succ-subsetI*: $\llbracket i \in j; i \subseteq j \rrbracket \implies \text{succ}(i) \subseteq j$
by (*unfold succ-def, blast*)

lemma *succ-subsetE*:
 $\llbracket \text{succ}(i) \subseteq j; \llbracket i \in j; i \subseteq j \rrbracket \implies P \rrbracket \implies P$
by (*unfold succ-def, blast*)

lemma *succ-subset-iff*: $\text{succ}(a) \subseteq B \longleftrightarrow (a \subseteq B \wedge a \in B)$
by (*unfold succ-def, blast*)

4.4 Binary Intersection

lemma *Int-subset-iff*: $C \subseteq A \cap B \longleftrightarrow C \subseteq A \wedge C \subseteq B$
by *blast*

lemma *Int-lower1*: $A \cap B \subseteq A$
by *blast*

lemma *Int-lower2*: $A \cap B \subseteq B$
by *blast*

lemma *Int-greatest*: $\llbracket C \subseteq A; C \subseteq B \rrbracket \implies C \subseteq A \cap B$
by *blast*

lemma *Int-cons*: $\text{cons}(a, B) \cap C \subseteq \text{cons}(a, B \cap C)$
by *blast*

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*

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-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 *subset-Int-iff*: $A \subseteq B \longleftrightarrow A \cap B = A$

by (*blast elim! equalityE*)

lemma *subset-Int-iff2*: $A \subseteq B \longleftrightarrow B \cap A = A$

by (*blast elim! equalityE*)

lemma *Int-Diff-eq*: $C \subseteq A \implies (A - B) \cap C = C - B$

by *blast*

lemma *Int-cons-left*:

$\text{cons}(a, A) \cap B = (\text{if } a \in B \text{ then } \text{cons}(a, A \cap B) \text{ else } A \cap B)$

by *auto*

lemma *Int-cons-right*:

$A \cap \text{cons}(a, B) = (\text{if } a \in A \text{ then } \text{cons}(a, A \cap B) \text{ else } A \cap B)$

by *auto*

lemma *cons-Int-distrib*: $\text{cons}(x, A \cap B) = \text{cons}(x, A) \cap \text{cons}(x, B)$

by *auto*

4.5 Binary Union

lemma *Un-subset-iff*: $A \cup B \subseteq C \longleftrightarrow A \subseteq C \wedge B \subseteq C$

by *blast*

lemma *Un-upper1*: $A \subseteq A \cup B$

by *blast*

lemma *Un-upper2*: $B \subseteq A \cup B$

by *blast*

lemma *Un-least*: $\llbracket A \subseteq C; B \subseteq C \rrbracket \implies A \cup B \subseteq C$

by *blast*

lemma *Un-cons*: $\text{cons}(a, B) \cup C = \text{cons}(a, B \cup C)$

by *blast*

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*

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-Int-distrib*: $(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$

by *blast*

lemma *subset-Un-iff*: $A \subseteq B \longleftrightarrow A \cup B = B$

by (*blast elim!:* *equalityE*)

lemma *subset-Un-iff2*: $A \subseteq B \longleftrightarrow B \cup A = B$

by (*blast elim!:* *equalityE*)

lemma *Un-empty* [*iff*]: $(A \cup B = 0) \longleftrightarrow (A = 0 \wedge B = 0)$

by *blast*

lemma *Un-eq-Union*: $A \cup B = \bigcup(\{A, B\})$

by *blast*

4.6 Set Difference

lemma *Diff-subset*: $A - B \subseteq A$
by *blast*

lemma *Diff-contains*: $\llbracket C \subseteq A; C \cap B = 0 \rrbracket \implies C \subseteq A - B$
by *blast*

lemma *subset-Diff-cons-iff*: $B \subseteq A - \text{cons}(c, C) \iff B \subseteq A - C \wedge c \notin B$
by *blast*

lemma *Diff-cancel*: $A - A = 0$
by *blast*

lemma *Diff-triv*: $A \cap B = 0 \implies A - B = A$
by *blast*

lemma *empty-Diff* [*simp*]: $0 - A = 0$
by *blast*

lemma *Diff-0* [*simp*]: $A - 0 = A$
by *blast*

lemma *Diff-eq-0-iff*: $A - B = 0 \iff A \subseteq B$
by (*blast elim: equalityE*)

lemma *Diff-cons*: $A - \text{cons}(a, B) = A - B - \{a\}$
by *blast*

lemma *Diff-cons2*: $A - \text{cons}(a, B) = A - \{a\} - B$
by *blast*

lemma *Diff-disjoint*: $A \cap (B - A) = 0$
by *blast*

lemma *Diff-partition*: $A \subseteq B \implies A \cup (B - A) = B$
by *blast*

lemma *subset-Un-Diff*: $A \subseteq B \cup (A - B)$
by *blast*

lemma *double-complement*: $\llbracket A \subseteq B; B \subseteq C \rrbracket \implies B - (C - A) = A$
by *blast*

lemma *double-complement-Un*: $(A \cup B) - (B - A) = A$
by *blast*

lemma *Un-Int-crazy*:

$(A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A)$
apply *blast*
done

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 *Un-Int-assoc-iff*: $(A \cap B) \cup C = A \cap (B \cup C) \longleftrightarrow C \subseteq A$
by (*blast elim!:* *equalityE*)

4.7 Big Union and Intersection

lemma *Union-subset-iff*: $\bigcup(A) \subseteq C \longleftrightarrow (\forall x \in A. x \subseteq C)$
by *blast*

lemma *Union-upper*: $B \in A \implies B \subseteq \bigcup(A)$
by *blast*

lemma *Union-least*: $\llbracket \bigwedge x. x \in A \implies x \subseteq C \rrbracket \implies \bigcup(A) \subseteq C$
by *blast*

lemma *Union-cons* [*simp*]: $\bigcup(\text{cons}(a, B)) = a \cup \bigcup(B)$
by *blast*

lemma *Union-Un-distrib*: $\bigcup(A \cup B) = \bigcup(A) \cup \bigcup(B)$
by *blast*

lemma *Union-Int-subset*: $\bigcup(A \cap B) \subseteq \bigcup(A) \cap \bigcup(B)$
by *blast*

lemma *Union-disjoint*: $\bigcup(C) \cap A = 0 \longleftrightarrow (\forall B \in C. B \cap A = 0)$
by (*blast elim!:* *equalityE*)

lemma *Union-empty-iff*: $\bigcup(A) = 0 \longleftrightarrow (\forall B \in A. B = 0)$
by *blast*

lemma *Int-Union2*: $\bigcup(B) \cap A = (\bigcup C \in B. C \cap A)$
by *blast*

lemma *Inter-subset-iff*: $A \neq 0 \implies C \subseteq \bigcap(A) \longleftrightarrow (\forall x \in A. C \subseteq x)$
by *blast*

lemma *Inter-lower*: $B \in A \implies \bigcap(A) \subseteq B$
by *blast*

lemma *Inter-greatest*: $\llbracket A \neq 0; \bigwedge x. x \in A \implies C \subseteq x \rrbracket \implies C \subseteq \bigcap(A)$
by *blast*

lemma *INT-lower*: $x \in A \implies (\bigcap x \in A. B(x)) \subseteq B(x)$
by *blast*

lemma *INT-greatest*: $\llbracket A \neq 0; \bigwedge x. x \in A \implies C \subseteq B(x) \rrbracket \implies C \subseteq (\bigcap x \in A. B(x))$
by *force*

lemma *Inter-0* [*simp*]: $\bigcap(0) = 0$
by (*unfold Inter-def, blast*)

lemma *Inter-Un-subset*:
 $\llbracket z \in A; z \in B \rrbracket \implies \bigcap(A) \cup \bigcap(B) \subseteq \bigcap(A \cap B)$
by *blast*

lemma *Inter-Un-distrib*:
 $\llbracket A \neq 0; B \neq 0 \rrbracket \implies \bigcap(A \cup B) = \bigcap(A) \cap \bigcap(B)$
by *blast*

lemma *Union-singleton*: $\bigcup(\{b\}) = b$
by *blast*

lemma *Inter-singleton*: $\bigcap(\{b\}) = b$
by *blast*

lemma *Inter-cons* [*simp*]:
 $\bigcap(\text{cons}(a, B)) = (\text{if } B = 0 \text{ then } a \text{ else } a \cap \bigcap(B))$
by *force*

4.8 Unions and Intersections of Families

lemma *subset-UN-iff-eq*: $A \subseteq (\bigcup_{i \in I}. B(i)) \longleftrightarrow A = (\bigcup_{i \in I}. A \cap B(i))$
by (*blast elim! equalityE*)

lemma *UN-subset-iff*: $(\bigcup_{x \in A}. B(x)) \subseteq C \longleftrightarrow (\forall x \in A. B(x) \subseteq C)$
by *blast*

lemma *UN-upper*: $x \in A \implies B(x) \subseteq (\bigcup_{x \in A}. B(x))$
by (*erule RepFunI [THEN Union-upper]*)

lemma *UN-least*: $\llbracket \bigwedge x. x \in A \implies B(x) \subseteq C \rrbracket \implies (\bigcup_{x \in A}. B(x)) \subseteq C$
by *blast*

lemma *Union-eq-UN*: $\bigcup (A) = (\bigcup_{x \in A}. x)$
by *blast*

lemma *Inter-eq-INT*: $\bigcap (A) = (\bigcap_{x \in A}. x)$
by (*unfold Inter-def, blast*)

lemma *UN-0 [simp]*: $(\bigcup_{i \in 0}. A(i)) = 0$
by *blast*

lemma *UN-singleton*: $(\bigcup_{x \in A}. \{x\}) = A$
by *blast*

lemma *UN-Un*: $(\bigcup_{i \in A \cup B}. C(i)) = (\bigcup_{i \in A}. C(i)) \cup (\bigcup_{i \in B}. C(i))$
by *blast*

lemma *INT-Un*: $(\bigcap_{i \in I \cup J}. A(i)) =$
 $(\text{if } I=0 \text{ then } \bigcap_{j \in J}. A(j)$
 $\text{else if } J=0 \text{ then } \bigcap_{i \in I}. A(i)$
 $\text{else } ((\bigcap_{i \in I}. A(i)) \cap (\bigcap_{j \in J}. A(j))))$
by (*simp, blast intro! equalityI*)

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 *Int-UN-distrib*: $B \cap (\bigcup_{i \in I}. A(i)) = (\bigcup_{i \in I}. B \cap A(i))$
by *blast*

lemma *Un-INT-distrib*: $I \neq 0 \implies B \cup (\bigcap_{i \in I}. A(i)) = (\bigcap_{i \in I}. B \cup A(i))$
by *auto*

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*: $\llbracket I \neq 0; J \neq 0 \rrbracket \implies$

$(\bigcap_{i \in I}. A(i)) \cup (\bigcap_{j \in J}. B(j)) = (\bigcap_{i \in I}. \bigcap_{j \in J}. A(i) \cup B(j))$
by *auto*

lemma *UN-constant [simp]*: $(\bigcup_{y \in A}. c) = (\text{if } A=0 \text{ then } 0 \text{ else } c)$
by *force*

lemma *INT-constant [simp]*: $(\bigcap_{y \in A}. c) = (\text{if } A=0 \text{ then } 0 \text{ else } c)$
by *force*

lemma *UN-RepFun [simp]*: $(\bigcup_{y \in \text{RepFun}(A,f)}. B(y)) = (\bigcup_{x \in A}. B(f(x)))$
by *blast*

lemma *INT-RepFun [simp]*: $(\bigcap_{x \in \text{RepFun}(A,f)}. B(x)) = (\bigcap_{a \in A}. B(f(a)))$
by (*auto simp add: Inter-def*)

lemma *INT-Union-eq*:
 $0 \notin A \implies (\bigcap_{x \in \bigcup(A)}. B(x)) = (\bigcap_{y \in A}. \bigcap_{x \in y}. B(x))$
apply (*subgoal-tac $\forall x \in A. x \neq 0$*)
prefer 2 **apply** *blast*
apply (*force simp add: Inter-def ball-conj-distrib*)
done

lemma *INT-UN-eq*:
 $(\forall x \in A. B(x) \neq 0) \implies (\bigcap_{z \in (\bigcup_{x \in A}. B(x))}. C(z)) = (\bigcap_{x \in A}. \bigcap_{z \in B(x)}. C(z))$
apply (*subst INT-Union-eq, blast*)
apply (*simp add: Inter-def*)
done

lemma *UN-Un-distrib*:
 $(\bigcup_{i \in I}. A(i) \cup B(i)) = (\bigcup_{i \in I}. A(i)) \cup (\bigcup_{i \in I}. B(i))$
by *blast*

lemma *INT-Int-distrib*:
 $I \neq 0 \implies (\bigcap_{i \in I}. A(i) \cap B(i)) = (\bigcap_{i \in I}. A(i)) \cap (\bigcap_{i \in I}. B(i))$
by (*blast elim!: not-emptyE*)

lemma *UN-Int-subset*:
 $(\bigcup_{z \in I \cap J}. A(z)) \subseteq (\bigcup_{z \in I}. A(z)) \cap (\bigcup_{z \in J}. A(z))$
by *blast*

lemma *Diff-UN*: $I \neq 0 \implies B - (\bigcup_{i \in I}. A(i)) = (\bigcap_{i \in I}. B - A(i))$
by (*blast elim!: not-emptyE*)

lemma *Diff-INT*: $I \neq 0 \implies B - (\bigcap_{i \in I}. A(i)) = (\bigcup_{i \in I}. B - A(i))$
by (*blast elim! not-emptyE*)

lemma *Sigma-cons1*: $\text{Sigma}(\text{cons}(a, B), C) = (\{a\} * C(a)) \cup \text{Sigma}(B, C)$
by *blast*

lemma *Sigma-cons2*: $A * \text{cons}(b, B) = A * \{b\} \cup A * B$
by *blast*

lemma *Sigma-succ1*: $\text{Sigma}(\text{succ}(A), B) = (\{A\} * B(A)) \cup \text{Sigma}(A, B)$
by *blast*

lemma *Sigma-succ2*: $A * \text{succ}(B) = A * \{B\} \cup A * B$
by *blast*

lemma *SUM-UN-distrib1*:
 $(\sum x \in (\bigcup_{y \in A}. C(y)). B(x)) = (\bigcup_{y \in A}. \sum x \in C(y). B(x))$
by *blast*

lemma *SUM-UN-distrib2*:
 $(\sum i \in I. \bigcup_{j \in J}. C(i, j)) = (\bigcup_{j \in J}. \sum i \in I. C(i, j))$
by *blast*

lemma *SUM-Un-distrib1*:
 $(\sum i \in I \cup J. C(i)) = (\sum i \in I. C(i)) \cup (\sum j \in J. C(j))$
by *blast*

lemma *SUM-Un-distrib2*:
 $(\sum i \in I. A(i) \cup B(i)) = (\sum i \in I. A(i)) \cup (\sum i \in I. B(i))$
by *blast*

lemma *prod-Un-distrib2*: $I * (A \cup B) = I * A \cup I * B$
by (*rule SUM-Un-distrib2*)

lemma *SUM-Int-distrib1*:
 $(\sum i \in I \cap J. C(i)) = (\sum i \in I. C(i)) \cap (\sum j \in J. C(j))$
by *blast*

lemma *SUM-Int-distrib2*:
 $(\sum i \in I. A(i) \cap B(i)) = (\sum i \in I. A(i)) \cap (\sum i \in I. B(i))$
by *blast*

lemma *prod-Int-distrib2*: $I * (A \cap B) = I * A \cap I * B$
by (*rule SUM-Int-distrib2*)

lemma *SUM-eq-UN*: $(\sum_{i \in I}. A(i)) = (\bigcup_{i \in I}. \{i\} * A(i))$
by *blast*

lemma *times-subset-iff*:
 $(A' * B' \subseteq A * B) \longleftrightarrow (A' = 0 \mid B' = 0 \mid (A' \subseteq A) \wedge (B' \subseteq B))$
by *blast*

lemma *Int-Sigma-eq*:
 $(\sum x \in A'. B'(x)) \cap (\sum x \in A. B(x)) = (\sum x \in A' \cap A. B'(x) \cap B(x))$
by *blast*

lemma *domain-iff*: $a: \text{domain}(r) \longleftrightarrow (\exists y. \langle a, y \rangle \in r)$
by (*unfold domain-def, blast*)

lemma *domainI [intro]*: $\langle a, b \rangle \in r \implies a: \text{domain}(r)$
by (*unfold domain-def, blast*)

lemma *domainE [elim!]*:
 $\llbracket a \in \text{domain}(r); \bigwedge y. \langle a, y \rangle \in r \implies P \rrbracket \implies P$
by (*unfold domain-def, blast*)

lemma *domain-subset*: $\text{domain}(\text{Sigma}(A, B)) \subseteq A$
by *blast*

lemma *domain-of-prod*: $b \in B \implies \text{domain}(A * B) = A$
by *blast*

lemma *domain-0 [simp]*: $\text{domain}(0) = 0$
by *blast*

lemma *domain-cons [simp]*: $\text{domain}(\text{cons}(\langle a, b \rangle, r)) = \text{cons}(a, \text{domain}(r))$
by *blast*

lemma *domain-Un-eq [simp]*: $\text{domain}(A \cup B) = \text{domain}(A) \cup \text{domain}(B)$
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-UN*: $\text{domain}(\bigcup_{x \in A}. B(x)) = (\bigcup_{x \in A}. \text{domain}(B(x)))$

by *blast*

lemma *domain-Union*: $\text{domain}(\bigcup(A)) = (\bigcup_{x \in A} \text{domain}(x))$
by *blast*

lemma *rangeI* [*intro*]: $\langle a, b \rangle \in r \implies b \in \text{range}(r)$
 unfolding *range-def*
apply (*erule converseI* [*THEN domainI*])
done

lemma *rangeE* [*elim!*]: $\llbracket b \in \text{range}(r); \bigwedge x. \langle x, b \rangle \in r \implies P \rrbracket \implies P$
by (*unfold range-def, blast*)

lemma *range-subset*: $\text{range}(A * B) \subseteq B$
 unfolding *range-def*
apply (*subst converse-prod*)
apply (*rule domain-subset*)
done

lemma *range-of-prod*: $a \in A \implies \text{range}(A * B) = B$
by *blast*

lemma *range-0* [*simp*]: $\text{range}(0) = 0$
by *blast*

lemma *range-cons* [*simp*]: $\text{range}(\text{cons}(\langle a, b \rangle, r)) = \text{cons}(b, \text{range}(r))$
by *blast*

lemma *range-Un-eq* [*simp*]: $\text{range}(A \cup B) = \text{range}(A) \cup \text{range}(B)$
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 *domain-converse* [*simp*]: $\text{domain}(\text{converse}(r)) = \text{range}(r)$
by *blast*

lemma *range-converse* [*simp*]: $\text{range}(\text{converse}(r)) = \text{domain}(r)$
by *blast*

lemma *fieldI1*: $\langle a, b \rangle \in r \implies a \in \text{field}(r)$
by (*unfold field-def, blast*)

lemma *fieldI2*: $\langle a, b \rangle \in r \implies b \in \text{field}(r)$
by (*unfold field-def, blast*)

lemma *fieldCI* [*intro*]:
 $(\neg \langle c, a \rangle \in r \implies \langle a, b \rangle \in r) \implies a \in \text{field}(r)$
apply (*unfold field-def, blast*)
done

lemma *fieldE* [*elim!*]:
 $\llbracket a \in \text{field}(r);$
 $\bigwedge x. \langle a, x \rangle \in r \implies P;$
 $\bigwedge x. \langle x, a \rangle \in r \implies P \rrbracket \implies P$
by (*unfold field-def, blast*)

lemma *field-subset*: $\text{field}(A * B) \subseteq A \cup B$
by *blast*

lemma *domain-subset-field*: $\text{domain}(r) \subseteq \text{field}(r)$
unfolding *field-def*
apply (*rule Un-upper1*)
done

lemma *range-subset-field*: $\text{range}(r) \subseteq \text{field}(r)$
unfolding *field-def*
apply (*rule Un-upper2*)
done

lemma *domain-times-range*: $r \subseteq \text{Sigma}(A, B) \implies r \subseteq \text{domain}(r) * \text{range}(r)$
by *blast*

lemma *field-times-field*: $r \subseteq \text{Sigma}(A, B) \implies r \subseteq \text{field}(r) * \text{field}(r)$
by *blast*

lemma *relation-field-times-field*: $\text{relation}(r) \implies r \subseteq \text{field}(r) * \text{field}(r)$
by (*simp add: relation-def, blast*)

lemma *field-of-prod*: $\text{field}(A * A) = A$
by *blast*

lemma *field-0* [*simp*]: $\text{field}(0) = 0$
by *blast*

lemma *field-cons* [*simp*]: $\text{field}(\text{cons}(\langle a, b \rangle, r)) = \text{cons}(a, \text{cons}(b, \text{field}(r)))$
by *blast*

lemma *field-Un-eq* [*simp*]: $\text{field}(A \cup B) = \text{field}(A) \cup \text{field}(B)$

by *blast*

lemma *field-Int-subset*: $\text{field}(A \cap B) \subseteq \text{field}(A) \cap \text{field}(B)$
by *blast*

lemma *field-Diff-subset*: $\text{field}(A) - \text{field}(B) \subseteq \text{field}(A - B)$
by *blast*

lemma *field-converse* [*simp*]: $\text{field}(\text{converse}(r)) = \text{field}(r)$
by *blast*

lemma *rel-Union*: $(\forall x \in S. \exists A B. x \subseteq A * B) \implies$
 $\bigcup(S) \subseteq \text{domain}(\bigcup(S)) * \text{range}(\bigcup(S))$
by *blast*

lemma *rel-Un*: $\llbracket r \subseteq A * B; s \subseteq C * D \rrbracket \implies (r \cup s) \subseteq (A \cup C) * (B \cup D)$
by *blast*

lemma *domain-Diff-eq*: $\llbracket \langle a, c \rangle \in r; c \neq b \rrbracket \implies \text{domain}(r - \{\langle a, b \rangle\}) = \text{domain}(r)$
by *blast*

lemma *range-Diff-eq*: $\llbracket \langle c, b \rangle \in r; c \neq a \rrbracket \implies \text{range}(r - \{\langle a, b \rangle\}) = \text{range}(r)$
by *blast*

4.9 Image of a Set under a Function or Relation

lemma *image-iff*: $b \in r''A \longleftrightarrow (\exists x \in A. \langle x, b \rangle \in r)$
by (*unfold image-def*, *blast*)

lemma *image-singleton-iff*: $b \in r''\{a\} \longleftrightarrow \langle a, b \rangle \in r$
by (*rule image-iff* [*THEN iff-trans*], *blast*)

lemma *imageI* [*intro*]: $\llbracket \langle a, b \rangle \in r; a \in A \rrbracket \implies b \in r''A$
by (*unfold image-def*, *blast*)

lemma *imageE* [*elim!*]:
 $\llbracket b: r''A; \bigwedge x. \llbracket \langle x, b \rangle \in r; x \in A \rrbracket \implies P \rrbracket \implies P$
by (*unfold image-def*, *blast*)

lemma *image-subset*: $r \subseteq A * B \implies r''C \subseteq B$
by *blast*

lemma *image-0* [*simp*]: $r''0 = 0$
by *blast*

lemma *image-Un* [*simp*]: $r''(A \cup B) = (r''A) \cup (r''B)$
by *blast*

lemma *image-UN*: $r^{-1}(\bigcup_{x \in A} B(x)) = \bigcup_{x \in A} r^{-1} B(x)$
by *blast*

lemma *Collect-image-eq*:
 $\{z \in \text{Sigma}(A, B). P(z)\}^{-1} C = (\bigcup x \in A. \{y \in B(x). x \in C \wedge P(\langle x, y \rangle)\})^{-1}$
by *blast*

lemma *image-Int-subset*: $r^{-1}(A \cap B) \subseteq (r^{-1}A) \cap (r^{-1}B)$
by *blast*

lemma *image-Int-square-subset*: $(r \cap A * A)^{-1}B \subseteq (r^{-1}B) \cap A$
by *blast*

lemma *image-Int-square*: $B \subseteq A \implies (r \cap A * A)^{-1}B = (r^{-1}B) \cap A$
by *blast*

lemma *image-0-left* [*simp*]: $0^{-1}A = 0$
by *blast*

lemma *image-Un-left*: $(r \cup s)^{-1}A = (r^{-1}A) \cup (s^{-1}A)$
by *blast*

lemma *image-Int-subset-left*: $(r \cap s)^{-1}A \subseteq (r^{-1}A) \cap (s^{-1}A)$
by *blast*

4.10 Inverse Image of a Set under a Function or Relation

lemma *vimage-iff*:
 $a \in r^{-1}B \iff (\exists y \in B. \langle a, y \rangle \in r)$
by (*unfold vimage-def image-def converse-def*, *blast*)

lemma *vimage-singleton-iff*: $a \in r^{-1}\{b\} \iff \langle a, b \rangle \in r$
by (*rule vimage-iff* [*THEN iff-trans*], *blast*)

lemma *vimageI* [*intro*]: $\llbracket \langle a, b \rangle \in r; b \in B \rrbracket \implies a \in r^{-1}B$
by (*unfold vimage-def*, *blast*)

lemma *vimageE* [*elim!*]:
 $\llbracket a: r^{-1}B; \bigwedge x. \llbracket \langle a, x \rangle \in r; x \in B \rrbracket \implies P \rrbracket \implies P$
apply (*unfold vimage-def*, *blast*)
done

lemma *vimage-subset*: $r \subseteq A * B \implies r^{-1}C \subseteq A$
unfolding *vimage-def*
apply (*erule converse-type* [*THEN image-subset*])
done

lemma *vimage-0* [simp]: $r-“0 = 0$

by *blast*

lemma *vimage-Un* [simp]: $r-“(A \cup B) = (r-“A) \cup (r-“B)$

by *blast*

lemma *vimage-Int-subset*: $r-“(A \cap B) \subseteq (r-“A) \cap (r-“B)$

by *blast*

lemma *vimage-eq-UN*: $f-“B = (\bigcup y \in B. f-“\{y\})$

by *blast*

lemma *function-vimage-Int*:

$function(f) \implies f-“(A \cap B) = (f-“A) \cap (f-“B)$

by (*unfold function-def*, *blast*)

lemma *function-vimage-Diff*: $function(f) \implies f-“(A-B) = (f-“A) - (f-“B)$

by (*unfold function-def*, *blast*)

lemma *function-image-vimage*: $function(f) \implies f-“(f-“A) \subseteq A$

by (*unfold function-def*, *blast*)

lemma *vimage-Int-square-subset*: $(r \cap A * A)-“B \subseteq (r-“B) \cap A$

by *blast*

lemma *vimage-Int-square*: $B \subseteq A \implies (r \cap A * A)-“B = (r-“B) \cap A$

by *blast*

lemma *vimage-0-left* [simp]: $0-“A = 0$

by *blast*

lemma *vimage-Un-left*: $(r \cup s)-“A = (r-“A) \cup (s-“A)$

by *blast*

lemma *vimage-Int-subset-left*: $(r \cap s)-“A \subseteq (r-“A) \cap (s-“A)$

by *blast*

lemma *converse-Un* [simp]: $converse(A \cup B) = converse(A) \cup converse(B)$

by *blast*

lemma *converse-Int* [simp]: $converse(A \cap B) = converse(A) \cap converse(B)$

by *blast*

lemma *converse-Diff* [simp]: $\text{converse}(A - B) = \text{converse}(A) - \text{converse}(B)$
by *blast*

lemma *converse-UN* [simp]: $\text{converse}(\bigcup x \in A. B(x)) = (\bigcup x \in A. \text{converse}(B(x)))$
by *blast*

lemma *converse-INT* [simp]:
 $\text{converse}(\bigcap x \in A. B(x)) = (\bigcap x \in A. \text{converse}(B(x)))$
apply (*unfold Inter-def, blast*)
done

4.11 Powerset Operator

lemma *Pow-0* [simp]: $\text{Pow}(\emptyset) = \{\emptyset\}$
by *blast*

lemma *Pow-insert*: $\text{Pow}(\text{cons}(a, A)) = \text{Pow}(A) \cup \{\text{cons}(a, X) \mid X \in \text{Pow}(A)\}$
apply (*rule equalityI, safe*)
apply (*erule swap*)
apply (*rule-tac a = x-{\a} in RepFun-eqI, auto*)
done

lemma *Un-Pow-subset*: $\text{Pow}(A) \cup \text{Pow}(B) \subseteq \text{Pow}(A \cup B)$
by *blast*

lemma *UN-Pow-subset*: $(\bigcup x \in A. \text{Pow}(B(x))) \subseteq \text{Pow}(\bigcup x \in A. B(x))$
by *blast*

lemma *subset-Pow-Union*: $A \subseteq \text{Pow}(\bigcup(A))$
by *blast*

lemma *Union-Pow-eq* [simp]: $\bigcup(\text{Pow}(A)) = A$
by *blast*

lemma *Union-Pow-iff*: $\bigcup(A) \in \text{Pow}(B) \longleftrightarrow A \in \text{Pow}(\text{Pow}(B))$
by *blast*

lemma *Pow-Int-eq* [simp]: $\text{Pow}(A \cap B) = \text{Pow}(A) \cap \text{Pow}(B)$
by *blast*

lemma *Pow-INT-eq*: $A \neq \emptyset \implies \text{Pow}(\bigcap x \in A. B(x)) = (\bigcap x \in A. \text{Pow}(B(x)))$
by (*blast elim!: not-emptyE*)

4.12 RepFun

lemma *RepFun-subset*: $\llbracket \bigwedge x. x \in A \implies f(x) \in B \rrbracket \implies \{f(x) \mid x \in A\} \subseteq B$
by *blast*

lemma *RepFun-eq-0-iff* [simp]: $\{f(x).x \in A\} = 0 \longleftrightarrow A = 0$

by *blast*

lemma *RepFun-constant* [simp]: $\{c. x \in A\} = (\text{if } A = 0 \text{ then } 0 \text{ else } \{c\})$

by *force*

4.13 Collect

lemma *Collect-subset*: $\text{Collect}(A, P) \subseteq A$

by *blast*

lemma *Collect-Un*: $\text{Collect}(A \cup B, P) = \text{Collect}(A, P) \cup \text{Collect}(B, P)$

by *blast*

lemma *Collect-Int*: $\text{Collect}(A \cap B, P) = \text{Collect}(A, P) \cap \text{Collect}(B, P)$

by *blast*

lemma *Collect-Diff*: $\text{Collect}(A - B, P) = \text{Collect}(A, P) - \text{Collect}(B, P)$

by *blast*

lemma *Collect-cons*: $\{x \in \text{cons}(a, B). P(x)\} =$
 $(\text{if } P(a) \text{ then } \text{cons}(a, \{x \in B. P(x)\}) \text{ else } \{x \in B. P(x)\})$

by (*simp*, *blast*)

lemma *Int-Collect-self-eq*: $A \cap \text{Collect}(A, P) = \text{Collect}(A, P)$

by *blast*

lemma *Collect-Collect-eq* [simp]:

$\text{Collect}(\text{Collect}(A, P), Q) = \text{Collect}(A, \lambda x. P(x) \wedge Q(x))$

by *blast*

lemma *Collect-Int-Collect-eq*:

$\text{Collect}(A, P) \cap \text{Collect}(A, Q) = \text{Collect}(A, \lambda x. P(x) \wedge Q(x))$

by *blast*

lemma *Collect-Union-eq* [simp]:

$\text{Collect}(\bigcup x \in A. B(x), P) = \bigcup x \in A. \text{Collect}(B(x), P)$

by *blast*

lemma *Collect-Int-left*: $\{x \in A. P(x)\} \cap B = \{x \in A \cap B. P(x)\}$

by *blast*

lemma *Collect-Int-right*: $A \cap \{x \in B. P(x)\} = \{x \in A \cap B. P(x)\}$

by *blast*

lemma *Collect-disj-eq*: $\{x \in A. P(x) \mid Q(x)\} = \text{Collect}(A, P) \cup \text{Collect}(A, Q)$

by *blast*

lemma *Collect-conj-eq*: $\{x \in A. P(x) \wedge Q(x)\} = \text{Collect}(A, P) \cap \text{Collect}(A, Q)$
by *blast*

lemmas *subset-SIs* = *subset-refl cons-subsetI subset-consI*
Union-least UN-least Un-least
Inter-greatest Int-greatest RepFun-subset
Un-upper1 Un-upper2 Int-lower1 Int-lower2

ML \langle
val subset-cs =
claset-of (**context**
delrules [$\text{@}\{thm\ subsetI\}$, $\text{@}\{thm\ subsetCE\}$]
addSIs $\text{@}\{thms\ subset-SIs\}$
addIs [$\text{@}\{thm\ Union-upper\}$, $\text{@}\{thm\ Inter-lower\}$]
addSEs [$\text{@}\{thm\ cons-subsetE\}$]);
val ZF-cs = *claset-of* (**context** *delrules* [$\text{@}\{thm\ equalityI\}$]);
 \rangle
end

5 Least and Greatest Fixed Points; the Knaster-Tarski Theorem

theory *Fixedpt* **imports** *equalities* **begin**

definition

bnd-mono :: $[i, i \Rightarrow i] \Rightarrow o$ **where**
 $bnd-mono(D, h) \equiv h(D) \leq D \wedge (\forall W X. W \leq X \longrightarrow X \leq D \longrightarrow h(W) \subseteq h(X))$

definition

lfp :: $[i, i \Rightarrow i] \Rightarrow i$ **where**
 $lfp(D, h) \equiv \bigcap (\{X: Pow(D). h(X) \subseteq X\})$

definition

gfp :: $[i, i \Rightarrow i] \Rightarrow i$ **where**
 $gfp(D, h) \equiv \bigcup (\{X: Pow(D). X \subseteq h(X)\})$

The theorem is proved in the lattice of subsets of D , namely $Pow(D)$, with Inter as the greatest lower bound.

5.1 Monotone Operators

lemma *bnd-monoI*:

$\llbracket h(D) \leq D; \bigwedge W X. \llbracket W \leq D; X \leq D; W \leq X \rrbracket \Longrightarrow h(W) \subseteq h(X)$

$\llbracket \implies \text{bnd-mono}(D, h) \rrbracket$
by (*unfold bnd-mono-def, clarify, blast*)

lemma *bnd-monoD1*: $\text{bnd-mono}(D, h) \implies h(D) \subseteq D$
unfolding *bnd-mono-def*
apply (*erule conjunct1*)
done

lemma *bnd-monoD2*: $\llbracket \text{bnd-mono}(D, h); W \leq X; X \leq D \rrbracket \implies h(W) \subseteq h(X)$
by (*unfold bnd-mono-def, blast*)

lemma *bnd-mono-subset*:
 $\llbracket \text{bnd-mono}(D, h); X \leq D \rrbracket \implies h(X) \subseteq D$
by (*unfold bnd-mono-def, clarify, blast*)

lemma *bnd-mono-Un*:
 $\llbracket \text{bnd-mono}(D, h); A \subseteq D; B \subseteq D \rrbracket \implies h(A) \cup h(B) \subseteq h(A \cup B)$
unfolding *bnd-mono-def*
apply (*rule Un-least, blast+*)
done

lemma *bnd-mono-UN*:
 $\llbracket \text{bnd-mono}(D, h); \forall i \in I. A(i) \subseteq D \rrbracket$
 $\implies (\bigcup_{i \in I} h(A(i))) \subseteq h(\bigcup_{i \in I} A(i))$
unfolding *bnd-mono-def*
apply (*rule UN-least*)
apply (*elim conjE*)
apply (*drule-tac x=A(i) in spec*)
apply (*drule-tac x=($\bigcup_{i \in I} A(i)$) in spec*)
apply *blast*
done

lemma *bnd-mono-Int*:
 $\llbracket \text{bnd-mono}(D, h); A \subseteq D; B \subseteq D \rrbracket \implies h(A \cap B) \subseteq h(A) \cap h(B)$
apply (*rule Int-greatest*)
apply (*erule bnd-monoD2, rule Int-lower1, assumption*)
apply (*erule bnd-monoD2, rule Int-lower2, assumption*)
done

5.2 Proof of Knaster-Tarski Theorem using *lfp*

lemma *lfp-lowerbound*:
 $\llbracket h(A) \subseteq A; A \leq D \rrbracket \implies \text{lfp}(D, h) \subseteq A$
by (*unfold lfp-def, blast*)

lemma *lfp-subset*: $\text{lfp}(D, h) \subseteq D$

by (*unfold lfp-def Inter-def, blast*)

lemma *def-lfp-subset*: $A \equiv \text{lfp}(D, h) \implies A \subseteq D$
apply *simp*
apply (*rule lfp-subset*)
done

lemma *lfp-greatest*:
 $\llbracket h(D) \subseteq D; \bigwedge X. \llbracket h(X) \subseteq X; X \leq D \rrbracket \implies A \leq X \rrbracket \implies A \subseteq \text{lfp}(D, h)$
by (*unfold lfp-def, blast*)

lemma *lfp-lemma1*:
 $\llbracket \text{bnd-mono}(D, h); h(A) \leq A; A \leq D \rrbracket \implies h(\text{lfp}(D, h)) \subseteq A$
apply (*erule bnd-monoD2 [THEN subset-trans]*)
apply (*rule lfp-lowerbound, assumption+*)
done

lemma *lfp-lemma2*: $\text{bnd-mono}(D, h) \implies h(\text{lfp}(D, h)) \subseteq \text{lfp}(D, h)$
apply (*rule bnd-monoD1 [THEN lfp-greatest]*)
apply (*rule-tac [2] lfp-lemma1*)
apply (*assumption+*)
done

lemma *lfp-lemma3*:
 $\text{bnd-mono}(D, h) \implies \text{lfp}(D, h) \subseteq h(\text{lfp}(D, h))$
apply (*rule lfp-lowerbound*)
apply (*rule bnd-monoD2, assumption*)
apply (*rule lfp-lemma2, assumption*)
apply (*erule-tac [2] bnd-mono-subset*)
apply (*rule lfp-subset*)+
done

lemma *lfp-unfold*: $\text{bnd-mono}(D, h) \implies \text{lfp}(D, h) = h(\text{lfp}(D, h))$
apply (*rule equalityI*)
apply (*erule lfp-lemma3*)
apply (*erule lfp-lemma2*)
done

lemma *def-lfp-unfold*:
 $\llbracket A \equiv \text{lfp}(D, h); \text{bnd-mono}(D, h) \rrbracket \implies A = h(A)$
apply *simp*
apply (*erule lfp-unfold*)
done

5.3 General Induction Rule for Least Fixedpoints

lemma *Collect-is-pre-fixedpt*:

$\llbracket \text{bnd-mono}(D, h); \bigwedge x. x \in h(\text{Collect}(\text{lfp}(D, h), P)) \implies P(x) \rrbracket$
 $\implies h(\text{Collect}(\text{lfp}(D, h), P)) \subseteq \text{Collect}(\text{lfp}(D, h), P)$
by (blast intro: lfp-lemma2 [THEN subsetD] bnd-monoD2 [THEN subsetD]
lfp-subset [THEN subsetD])

lemma *induct*:
 $\llbracket \text{bnd-mono}(D, h); a \in \text{lfp}(D, h);$
 $\bigwedge x. x \in h(\text{Collect}(\text{lfp}(D, h), P)) \implies P(x) \rrbracket$
 $\implies P(a)$
apply (rule *Collect-is-pre-fixedpt*
[THEN lfp-lowerbound, THEN subsetD, THEN CollectD2])
apply (rule-tac [3] lfp-subset [THEN Collect-subset [THEN subset-trans]],
blast+)
done

lemma *def-induct*:
 $\llbracket A \equiv \text{lfp}(D, h); \text{bnd-mono}(D, h); a:A;$
 $\bigwedge x. x \in h(\text{Collect}(A, P)) \implies P(x) \rrbracket$
 $\implies P(a)$
by (rule *induct*, blast+)

lemma *lfp-Int-lowerbound*:
 $\llbracket h(D \cap A) \subseteq A; \text{bnd-mono}(D, h) \rrbracket \implies \text{lfp}(D, h) \subseteq A$
apply (rule lfp-lowerbound [THEN subset-trans])
apply (erule bnd-mono-subset [THEN Int-greatest], blast+)
done

lemma *lfp-mono*:
assumes *hmono*: $\text{bnd-mono}(D, h)$
and *imono*: $\text{bnd-mono}(E, i)$
and *subhi*: $\bigwedge X. X \leq D \implies h(X) \subseteq i(X)$
shows $\text{lfp}(D, h) \subseteq \text{lfp}(E, i)$
apply (rule bnd-monoD1 [THEN lfp-greatest])
apply (rule imono)
apply (rule hmono [THEN [2] lfp-Int-lowerbound])
apply (rule Int-lower1 [THEN subhi, THEN subset-trans])
apply (rule imono [THEN bnd-monoD2, THEN subset-trans], auto)
done

lemma *lfp-mono2*:
 $\llbracket i(D) \subseteq D; \bigwedge X. X \leq D \implies h(X) \subseteq i(X) \rrbracket \implies \text{lfp}(D, h) \subseteq \text{lfp}(D, i)$
apply (rule lfp-greatest, assumption)
apply (rule lfp-lowerbound, blast, assumption)
done

```

lemma lfp-cong:
   $\llbracket D=D'; \bigwedge X. X \subseteq D' \implies h(X) = h'(X) \rrbracket \implies \text{lfp}(D,h) = \text{lfp}(D',h')$ 
apply (simp add: lfp-def)
apply (rule-tac t=Inter in subst-context)
apply (rule Collect-cong, simp-all)
done

```

5.4 Proof of Knaster-Tarski Theorem using *gfp*

```

lemma gfp-upperbound:  $\llbracket A \subseteq h(A); A \leq D \rrbracket \implies A \subseteq \text{gfp}(D,h)$ 
  unfolding gfp-def
apply (rule PowI [THEN CollectI, THEN Union-upper])
apply (assumption+)
done

```

```

lemma gfp-subset:  $\text{gfp}(D,h) \subseteq D$ 
by (unfold gfp-def, blast)

```

```

lemma def-gfp-subset:  $A \equiv \text{gfp}(D,h) \implies A \subseteq D$ 
apply simp
apply (rule gfp-subset)
done

```

```

lemma gfp-least:
   $\llbracket \text{bnd-mono}(D,h); \bigwedge X. \llbracket X \subseteq h(X); X \leq D \rrbracket \implies X \leq A \rrbracket \implies$ 
   $\text{gfp}(D,h) \subseteq A$ 
  unfolding gfp-def
apply (blast dest: bnd-monoD1)
done

```

```

lemma gfp-lemma1:
   $\llbracket \text{bnd-mono}(D,h); A \leq h(A); A \leq D \rrbracket \implies A \subseteq h(\text{gfp}(D,h))$ 
apply (rule subset-trans, assumption)
apply (erule bnd-monoD2)
apply (rule-tac [2] gfp-subset)
apply (simp add: gfp-upperbound)
done

```

```

lemma gfp-lemma2:  $\text{bnd-mono}(D,h) \implies \text{gfp}(D,h) \subseteq h(\text{gfp}(D,h))$ 
apply (rule gfp-least)
apply (rule-tac [2] gfp-lemma1)
apply (assumption+)
done

```

```

lemma gfp-lemma3:
   $\text{bnd-mono}(D,h) \implies h(\text{gfp}(D,h)) \subseteq \text{gfp}(D,h)$ 
apply (rule gfp-upperbound)

```

```

apply (rule bnd-monoD2, assumption)
apply (rule gfp-lemma2, assumption)
apply (erule bnd-mono-subset, rule gfp-subset)+
done

```

```

lemma gfp-unfold: bnd-mono( $D, h$ )  $\implies$   $\text{gfp}(D, h) = h(\text{gfp}(D, h))$ 
apply (rule equalityI)
apply (erule gfp-lemma2)
apply (erule gfp-lemma3)
done

```

```

lemma def-gfp-unfold:
   $\llbracket A \equiv \text{gfp}(D, h); \text{bnd-mono}(D, h) \rrbracket \implies A = h(A)$ 
apply simp
apply (erule gfp-unfold)
done

```

5.5 Coinduction Rules for Greatest Fixed Points

```

lemma weak-coinduct:  $\llbracket a: X; X \subseteq h(X); X \subseteq D \rrbracket \implies a \in \text{gfp}(D, h)$ 
by (blast intro: gfp-upperbound [THEN subsetD])

```

```

lemma coinduct-lemma:
   $\llbracket X \subseteq h(X \cup \text{gfp}(D, h)); X \subseteq D; \text{bnd-mono}(D, h) \rrbracket \implies$ 
   $X \cup \text{gfp}(D, h) \subseteq h(X \cup \text{gfp}(D, h))$ 
apply (erule Un-least)
apply (rule gfp-lemma2 [THEN subset-trans], assumption)
apply (rule Un-upper2 [THEN subset-trans])
apply (rule bnd-mono-Un, assumption+)
apply (rule gfp-subset)
done

```

```

lemma coinduct:
   $\llbracket \text{bnd-mono}(D, h); a: X; X \subseteq h(X \cup \text{gfp}(D, h)); X \subseteq D \rrbracket$ 
   $\implies a \in \text{gfp}(D, h)$ 
apply (rule weak-coinduct)
apply (erule-tac [2] coinduct-lemma)
apply (simp-all add: gfp-subset Un-subset-iff)
done

```

```

lemma def-coinduct:
   $\llbracket A \equiv \text{gfp}(D, h); \text{bnd-mono}(D, h); a: X; X \subseteq h(X \cup A); X \subseteq D \rrbracket \implies$ 
   $a \in A$ 
apply simp
apply (rule coinduct, assumption+)
done

```

```

lemma def-Collect-coinduct:
   $\llbracket A \equiv \text{gfp}(D, \lambda w. \text{Collect}(D, P(w))); \text{bnd-mono}(D, \lambda w. \text{Collect}(D, P(w)));$ 
     $a: X; X \subseteq D; \bigwedge z. z: X \implies P(X \cup A, z) \rrbracket \implies$ 
     $a \in A$ 
apply (rule def-coinduct, assumption+, blast+)
done

```

```

lemma gfp-mono:
   $\llbracket \text{bnd-mono}(D, h); D \subseteq E;$ 
     $\bigwedge X. X \leq D \implies h(X) \subseteq i(X) \rrbracket \implies \text{gfp}(D, h) \subseteq \text{gfp}(E, i)$ 
apply (rule gfp-upperbound)
apply (rule gfp-lemma2 [THEN subset-trans], assumption)
apply (blast del: subsetI intro: gfp-subset)
apply (blast del: subsetI intro: subset-trans gfp-subset)
done

end

```

6 Booleans in Zermelo-Fraenkel Set Theory

theory *Bool* **imports** *pair* **begin**

abbreviation
one $\langle 1 \rangle$ **where**
 $1 \equiv \text{succ}(0)$

abbreviation
two $\langle 2 \rangle$ **where**
 $2 \equiv \text{succ}(1)$

2 is equal to bool, but is used as a number rather than a type.

definition *bool* $\equiv \{0, 1\}$

definition *cond*(*b, c, d*) $\equiv \text{if}(b=1, c, d)$

definition *not*(*b*) $\equiv \text{cond}(b, 0, 1)$

definition
and $:: [i, i] \Rightarrow i$ (**infixl** $\langle \text{and} \rangle$ 70) **where**
 $a \text{ and } b \equiv \text{cond}(a, b, 0)$

definition
or $:: [i, i] \Rightarrow i$ (**infixl** $\langle \text{or} \rangle$ 65) **where**
 $a \text{ or } b \equiv \text{cond}(a, 1, b)$

definition

$xor :: [i,i] \Rightarrow i$ (**infixl** $\langle xor \rangle$ 65) **where**
 $a xor b \equiv cond(a, not(b), b)$

lemmas $bool-defs = bool-def cond-def$

lemma $singleton-0$: $\{0\} = 1$
by ($simp$ add : $succ-def$)

lemma $bool-1I$ [$simp, TC$]: $1 \in bool$
by ($simp$ add : $bool-defs$)

lemma $bool-0I$ [$simp, TC$]: $0 \in bool$
by ($simp$ add : $bool-defs$)

lemma $one-not-0$: $1 \neq 0$
by ($simp$ add : $bool-defs$)

lemmas $one-neq-0 = one-not-0$ [$THEN notE$]

lemma $boolE$:
 $\llbracket c: bool; \ c=1 \implies P; \ c=0 \implies P \rrbracket \implies P$
by ($simp$ add : $bool-defs, blast$)

lemma $cond-1$ [$simp$]: $cond(1, c, d) = c$
by ($simp$ add : $bool-defs$)

lemma $cond-0$ [$simp$]: $cond(0, c, d) = d$
by ($simp$ add : $bool-defs$)

lemma $cond-type$ [TC]: $\llbracket b: bool; \ c: A(1); \ d: A(0) \rrbracket \implies cond(b, c, d): A(b)$
by ($simp$ add : $bool-defs, blast$)

lemma $cond-simple-type$: $\llbracket b: bool; \ c: A; \ d: A \rrbracket \implies cond(b, c, d): A$
by ($simp$ add : $bool-defs$)

lemma $def-cond-1$: $\llbracket \bigwedge b. j(b) \equiv cond(b, c, d) \rrbracket \implies j(1) = c$
by $simp$

lemma $def-cond-0$: $\llbracket \bigwedge b. j(b) \equiv cond(b, c, d) \rrbracket \implies j(0) = d$
by $simp$

lemmas *not-1* = *not-def* [*THEN def-cond-1, simp*]
lemmas *not-0* = *not-def* [*THEN def-cond-0, simp*]

lemmas *and-1* = *and-def* [*THEN def-cond-1, simp*]
lemmas *and-0* = *and-def* [*THEN def-cond-0, simp*]

lemmas *or-1* = *or-def* [*THEN def-cond-1, simp*]
lemmas *or-0* = *or-def* [*THEN def-cond-0, simp*]

lemmas *xor-1* = *xor-def* [*THEN def-cond-1, simp*]
lemmas *xor-0* = *xor-def* [*THEN def-cond-0, simp*]

lemma *not-type* [*TC*]: $a:\text{bool} \implies \text{not}(a) \in \text{bool}$
by (*simp add: not-def*)

lemma *and-type* [*TC*]: $\llbracket a:\text{bool}; b:\text{bool} \rrbracket \implies a \text{ and } b \in \text{bool}$
by (*simp add: and-def*)

lemma *or-type* [*TC*]: $\llbracket a:\text{bool}; b:\text{bool} \rrbracket \implies a \text{ or } b \in \text{bool}$
by (*simp add: or-def*)

lemma *xor-type* [*TC*]: $\llbracket a:\text{bool}; b:\text{bool} \rrbracket \implies a \text{ xor } b \in \text{bool}$
by (*simp add: xor-def*)

lemmas *bool-typechecks* = *bool-1I bool-0I cond-type not-type and-type*
or-type xor-type

6.1 Laws About 'not'

lemma *not-not* [*simp*]: $a:\text{bool} \implies \text{not}(\text{not}(a)) = a$
by (*elim boolE, auto*)

lemma *not-and* [*simp*]: $a:\text{bool} \implies \text{not}(a \text{ and } b) = \text{not}(a) \text{ or } \text{not}(b)$
by (*elim boolE, auto*)

lemma *not-or* [*simp*]: $a:\text{bool} \implies \text{not}(a \text{ or } b) = \text{not}(a) \text{ and } \text{not}(b)$
by (*elim boolE, auto*)

6.2 Laws About 'and'

lemma *and-absorb* [*simp*]: $a:\text{bool} \implies a \text{ and } a = a$
by (*elim boolE, auto*)

lemma *and-commute*: $\llbracket a:\text{bool}; b:\text{bool} \rrbracket \implies a \text{ and } b = b \text{ and } a$
by (*elim boolE, auto*)

lemma *and-assoc*: $a:\text{bool} \implies (a \text{ and } b) \text{ and } c = a \text{ and } (b \text{ and } c)$
by (*elim boolE, auto*)

lemma *and-or-distrib*: $\llbracket a: \text{bool}; b: \text{bool}; c: \text{bool} \rrbracket \implies$
 $(a \text{ or } b) \text{ and } c = (a \text{ and } c) \text{ or } (b \text{ and } c)$
by (*elim boolE*, *auto*)

6.3 Laws About 'or'

lemma *or-absorb* [*simp*]: $a: \text{bool} \implies a \text{ or } a = a$
by (*elim boolE*, *auto*)

lemma *or-commute*: $\llbracket a: \text{bool}; b: \text{bool} \rrbracket \implies a \text{ or } b = b \text{ or } a$
by (*elim boolE*, *auto*)

lemma *or-assoc*: $a: \text{bool} \implies (a \text{ or } b) \text{ or } c = a \text{ or } (b \text{ or } c)$
by (*elim boolE*, *auto*)

lemma *or-and-distrib*: $\llbracket a: \text{bool}; b: \text{bool}; c: \text{bool} \rrbracket \implies$
 $(a \text{ and } b) \text{ or } c = (a \text{ or } c) \text{ and } (b \text{ or } c)$
by (*elim boolE*, *auto*)

definition

bool-of-o :: $o \Rightarrow i$ **where**
bool-of-o(P) \equiv (*if* P *then* 1 *else* 0)

lemma [*simp*]: $\text{bool-of-o}(\text{True}) = 1$
by (*simp add: bool-of-o-def*)

lemma [*simp*]: $\text{bool-of-o}(\text{False}) = 0$
by (*simp add: bool-of-o-def*)

lemma [*simp, TC*]: $\text{bool-of-o}(P) \in \text{bool}$
by (*simp add: bool-of-o-def*)

lemma [*simp*]: $(\text{bool-of-o}(P) = 1) \longleftrightarrow P$
by (*simp add: bool-of-o-def*)

lemma [*simp*]: $(\text{bool-of-o}(P) = 0) \longleftrightarrow \neg P$
by (*simp add: bool-of-o-def*)

end

7 Disjoint Sums

theory *Sum* **imports** *Bool equalities* **begin**

And the "Part" primitive for simultaneous recursive type definitions

definition *sum* :: $[i, i] \Rightarrow i$ (**infixr** $\langle + \rangle$ 65) **where**
 $A + B \equiv \{0\} * A \cup \{1\} * B$

definition $Inl :: i \Rightarrow i$ **where**

$$Inl(a) \equiv \langle 0, a \rangle$$

definition $Inr :: i \Rightarrow i$ **where**

$$Inr(b) \equiv \langle 1, b \rangle$$

definition $case :: [i \Rightarrow i, i \Rightarrow i, i] \Rightarrow i$ **where**

$$case(c, d) \equiv (\lambda \langle y, z \rangle. cond(y, d(z), c(z)))$$

definition $Part :: [i, i \Rightarrow i] \Rightarrow i$ **where**

$$Part(A, h) \equiv \{x \in A. \exists z. x = h(z)\}$$

7.1 Rules for the *Part* Primitive

lemma *Part-iff*:

$$a \in Part(A, h) \longleftrightarrow a \in A \wedge (\exists y. a = h(y))$$

unfolding *Part-def*

apply (*rule separation*)

done

lemma *Part-eqI* [*intro*]:

$$\llbracket a \in A; a = h(b) \rrbracket \Longrightarrow a \in Part(A, h)$$

by (*unfold Part-def, blast*)

lemmas *PartI* = *refl* [*THEN* [2] *Part-eqI*]

lemma *PartE* [*elim!*]:

$$\llbracket a \in Part(A, h); \bigwedge z. \llbracket a \in A; a = h(z) \rrbracket \Longrightarrow P$$

$$\rrbracket \Longrightarrow P$$

apply (*unfold Part-def, blast*)

done

lemma *Part-subset*: $Part(A, h) \subseteq A$

unfolding *Part-def*

apply (*rule Collect-subset*)

done

7.2 Rules for Disjoint Sums

lemmas *sum-defs* = *sum-def Inl-def Inr-def case-def*

lemma *Sigma-bool*: $Sigma(bool, C) = C(0) + C(1)$

by (*unfold bool-def sum-def, blast*)

lemma *InlI* [*intro!, simp, TC*]: $a \in A \Longrightarrow Inl(a) \in A+B$

by (*unfold sum-defs, blast*)

lemma *InrI* [*intro!*,*simp*,*TC*]: $b \in B \implies \text{Inr}(b) \in A+B$
by (*unfold sum-defs*, *blast*)

lemma *sumE* [*elim!*]:
 $\llbracket u \in A+B;$
 $\quad \bigwedge x. \llbracket x \in A; u=\text{Inl}(x) \rrbracket \implies P;$
 $\quad \bigwedge y. \llbracket y \in B; u=\text{Inr}(y) \rrbracket \implies P$
 $\rrbracket \implies P$
by (*unfold sum-defs*, *blast*)

lemma *Inl-iff* [*iff*]: $\text{Inl}(a)=\text{Inl}(b) \longleftrightarrow a=b$
by (*simp add: sum-defs*)

lemma *Inr-iff* [*iff*]: $\text{Inr}(a)=\text{Inr}(b) \longleftrightarrow a=b$
by (*simp add: sum-defs*)

lemma *Inl-Inr-iff* [*simp*]: $\text{Inl}(a)=\text{Inr}(b) \longleftrightarrow \text{False}$
by (*simp add: sum-defs*)

lemma *Inr-Inl-iff* [*simp*]: $\text{Inr}(b)=\text{Inl}(a) \longleftrightarrow \text{False}$
by (*simp add: sum-defs*)

lemma *sum-empty* [*simp*]: $0+0 = 0$
by (*simp add: sum-defs*)

lemmas *Inl-inject* = *Inl-iff* [*THEN iffD1*]
lemmas *Inr-inject* = *Inr-iff* [*THEN iffD1*]
lemmas *Inl-neq-Inr* = *Inl-Inr-iff* [*THEN iffD1*, *THEN FalseE*, *elim!*]
lemmas *Inr-neq-Inl* = *Inr-Inl-iff* [*THEN iffD1*, *THEN FalseE*, *elim!*]

lemma *InlD*: $\text{Inl}(a): A+B \implies a \in A$
by *blast*

lemma *InrD*: $\text{Inr}(b): A+B \implies b \in B$
by *blast*

lemma *sum-iff*: $u \in A+B \longleftrightarrow (\exists x. x \in A \wedge u=\text{Inl}(x)) \mid (\exists y. y \in B \wedge u=\text{Inr}(y))$
by *blast*

lemma *Inl-in-sum-iff* [*simp*]: $(\text{Inl}(x) \in A+B) \longleftrightarrow (x \in A)$
by *auto*

lemma *Inr-in-sum-iff* [simp]: $(\text{Inr}(y) \in A+B) \longleftrightarrow (y \in B)$
by *auto*

lemma *sum-subset-iff*: $A+B \subseteq C+D \longleftrightarrow A \leq C \wedge B \leq D$
by *blast*

lemma *sum-equal-iff*: $A+B = C+D \longleftrightarrow A=C \wedge B=D$
by (*simp add: extension sum-subset-iff, blast*)

lemma *sum-eq-2-times*: $A+A = 2*A$
by (*simp add: sum-def, blast*)

7.3 The Eliminator: *case*

lemma *case-Inl* [simp]: $\text{case}(c, d, \text{Inl}(a)) = c(a)$
by (*simp add: sum-defs*)

lemma *case-Inr* [simp]: $\text{case}(c, d, \text{Inr}(b)) = d(b)$
by (*simp add: sum-defs*)

lemma *case-type* [TC]:

$$\begin{aligned} & \llbracket u \in A+B; \\ & \quad \bigwedge x. x \in A \implies c(x): C(\text{Inl}(x)); \\ & \quad \bigwedge y. y \in B \implies d(y): C(\text{Inr}(y)) \\ & \rrbracket \implies \text{case}(c, d, u) \in C(u) \end{aligned}$$

by *auto*

lemma *expand-case*: $u \in A+B \implies$

$$\begin{aligned} & R(\text{case}(c, d, u)) \longleftrightarrow \\ & ((\forall x \in A. u = \text{Inl}(x) \longrightarrow R(c(x))) \wedge \\ & (\forall y \in B. u = \text{Inr}(y) \longrightarrow R(d(y)))) \end{aligned}$$

by *auto*

lemma *case-cong*:

$$\begin{aligned} & \llbracket z \in A+B; \\ & \quad \bigwedge x. x \in A \implies c(x)=c'(x); \\ & \quad \bigwedge y. y \in B \implies d(y)=d'(y) \\ & \rrbracket \implies \text{case}(c, d, z) = \text{case}(c', d', z) \end{aligned}$$

by *auto*

lemma *case-case*: $z \in A+B \implies$

$$\begin{aligned} & \text{case}(c, d, \text{case}(\lambda x. \text{Inl}(c'(x)), \lambda y. \text{Inr}(d'(y)), z)) = \\ & \text{case}(\lambda x. c'(x), \lambda y. d'(y), z) \end{aligned}$$

by *auto*

7.4 More Rules for $\text{Part}(A, h)$

lemma *Part-mono*: $A \leq B \implies \text{Part}(A, h) \leq \text{Part}(B, h)$
by *blast*

lemma *Part-Collect*: $Part(Collect(A,P), h) = Collect(Part(A,h), P)$
by *blast*

lemmas *Part-CollectE* =
 $Part-Collect [THEN equalityD1, THEN subsetD, THEN CollectE]$

lemma *Part-Inl*: $Part(A+B, Inl) = \{Inl(x). x \in A\}$
by *blast*

lemma *Part-Inr*: $Part(A+B, Inr) = \{Inr(y). y \in B\}$
by *blast*

lemma *PartD1*: $a \in Part(A,h) \implies a \in A$
by (*simp add: Part-def*)

lemma *Part-id*: $Part(A, \lambda x. x) = A$
by *blast*

lemma *Part-Inr2*: $Part(A+B, \lambda x. Inr(h(x))) = \{Inr(y). y \in Part(B,h)\}$
by *blast*

lemma *Part-sum-equality*: $C \subseteq A+B \implies Part(C, Inl) \cup Part(C, Inr) = C$
by *blast*

end

8 Functions, Function Spaces, Lambda-Abstraction

theory *func* **imports** *equalities Sum* **begin**

8.1 The Pi Operator: Dependent Function Space

lemma *subset-Sigma-imp-relation*: $r \subseteq Sigma(A,B) \implies relation(r)$
by (*simp add: relation-def, blast*)

lemma *relation-converse-converse* [*simp*]:
 $relation(r) \implies converse(converse(r)) = r$
by (*simp add: relation-def, blast*)

lemma *relation-restrict* [*simp*]: $relation(restrict(r,A))$
by (*simp add: restrict-def relation-def, blast*)

lemma *Pi-iff*:
 $f \in Pi(A,B) \iff function(f) \wedge f \leq Sigma(A,B) \wedge A \leq domain(f)$
by (*unfold Pi-def, blast*)

lemma *Pi-iff-old*:
 $f \in Pi(A,B) \iff f \leq Sigma(A,B) \wedge (\forall x \in A. \exists! y. \langle x, y \rangle: f)$

by (*unfold Pi-def function-def, blast*)

lemma *fun-is-function*: $f \in Pi(A,B) \implies function(f)$
by (*simp only: Pi-iff*)

lemma *function-imp-Pi*:
 $\llbracket function(f); relation(f) \rrbracket \implies f \in domain(f) \rightarrow range(f)$
by (*simp add: Pi-iff relation-def, blast*)

lemma *functionI*:
 $\llbracket \bigwedge x y y'. \llbracket \langle x,y \rangle : r; \langle x,y' \rangle : r \rrbracket \implies y=y' \rrbracket \implies function(r)$
by (*simp add: function-def, blast*)

lemma *fun-is-rel*: $f \in Pi(A,B) \implies f \subseteq Sigma(A,B)$
by (*unfold Pi-def, blast*)

lemma *Pi-cong*:
 $\llbracket A=A'; \bigwedge x. x \in A' \implies B(x)=B'(x) \rrbracket \implies Pi(A,B) = Pi(A',B')$
by (*simp add: Pi-def cong add: Sigma-cong*)

lemma *fun-weaken-type*: $\llbracket f \in A \rightarrow B; B \leq D \rrbracket \implies f \in A \rightarrow D$
by (*unfold Pi-def, best*)

8.2 Function Application

lemma *apply-equality2*: $\llbracket \langle a,b \rangle : f; \langle a,c \rangle : f; f \in Pi(A,B) \rrbracket \implies b=c$
by (*unfold Pi-def function-def, blast*)

lemma *function-apply-equality*: $\llbracket \langle a,b \rangle : f; function(f) \rrbracket \implies f'a = b$
by (*unfold apply-def function-def, blast*)

lemma *apply-equality*: $\llbracket \langle a,b \rangle : f; f \in Pi(A,B) \rrbracket \implies f'a = b$
unfolding *Pi-def*
apply (*blast intro: function-apply-equality*)
done

lemma *apply-0*: $a \notin domain(f) \implies f'a = 0$
by (*unfold apply-def, blast*)

lemma *Pi-memberD*: $\llbracket f \in Pi(A,B); c \in f \rrbracket \implies \exists x \in A. c = \langle x, f'x \rangle$
apply (*frule fun-is-rel*)
apply (*blast dest: apply-equality*)
done

lemma *function-apply-Pair*: $\llbracket \text{function}(f); a \in \text{domain}(f) \rrbracket \implies \langle a, f'a \rangle: f$
apply (*simp add: function-def, clarify*)
apply (*subgoal-tac f'a = y, blast*)
apply (*simp add: apply-def, blast*)
done

lemma *apply-Pair*: $\llbracket f \in \text{Pi}(A, B); a \in A \rrbracket \implies \langle a, f'a \rangle: f$
apply (*simp add: Pi-iff*)
apply (*blast intro: function-apply-Pair*)
done

lemma *apply-type* [TC]: $\llbracket f \in \text{Pi}(A, B); a \in A \rrbracket \implies f'a \in B(a)$
by (*blast intro: apply-Pair dest: fun-is-rel*)

lemma *apply-funtype*: $\llbracket f \in A \multimap B; a \in A \rrbracket \implies f'a \in B$
by (*blast dest: apply-type*)

lemma *apply-iff*: $f \in \text{Pi}(A, B) \implies \langle a, b \rangle: f \longleftrightarrow a \in A \wedge f'a = b$
apply (*frule fun-is-rel*)
apply (*blast intro!: apply-Pair apply-equality*)
done

lemma *Pi-type*: $\llbracket f \in \text{Pi}(A, C); \bigwedge x. x \in A \implies f'x \in B(x) \rrbracket \implies f \in \text{Pi}(A, B)$
apply (*simp only: Pi-iff*)
apply (*blast dest: function-apply-equality*)
done

lemma *Pi-Collect-iff*:
 $(f \in \text{Pi}(A, \lambda x. \{y \in B(x). P(x, y)\}))$
 $\longleftrightarrow f \in \text{Pi}(A, B) \wedge (\forall x \in A. P(x, f'x))$
by (*blast intro: Pi-type dest: apply-type*)

lemma *Pi-weaken-type*:
 $\llbracket f \in \text{Pi}(A, B); \bigwedge x. x \in A \implies B(x) \leq C(x) \rrbracket \implies f \in \text{Pi}(A, C)$
by (*blast intro: Pi-type dest: apply-type*)

lemma *domain-type*: $\llbracket \langle a, b \rangle \in f; f \in \text{Pi}(A, B) \rrbracket \implies a \in A$
by (*blast dest: fun-is-rel*)

lemma *range-type*: $\llbracket \langle a, b \rangle \in f; f \in \text{Pi}(A, B) \rrbracket \implies b \in B(a)$
by (*blast dest: fun-is-rel*)

lemma *Pair-mem-PiD*: $\llbracket \langle a, b \rangle : f; f \in \text{Pi}(A, B) \rrbracket \implies a \in A \wedge b \in B(a) \wedge f'a = b$
by (*blast intro: domain-type range-type apply-equality*)

8.3 Lambda Abstraction

lemma *lamI*: $a \in A \implies \langle a, b(a) \rangle \in (\lambda x \in A. b(x))$
unfolding *lam-def*
apply (*erule RepFunI*)
done

lemma *lamE*:
 $\llbracket p: (\lambda x \in A. b(x)); \bigwedge x. \llbracket x \in A; p = \langle x, b(x) \rangle \rrbracket \implies P$
 $\rrbracket \implies P$
by (*simp add: lam-def, blast*)

lemma *lamD*: $\llbracket \langle a, c \rangle : (\lambda x \in A. b(x)) \rrbracket \implies c = b(a)$
by (*simp add: lam-def*)

lemma *lam-type [TC]*:
 $\llbracket \bigwedge x. x \in A \implies b(x) : B(x) \rrbracket \implies (\lambda x \in A. b(x)) \in \text{Pi}(A, B)$
by (*simp add: lam-def Pi-def function-def, blast*)

lemma *lam-funtype*: $(\lambda x \in A. b(x)) \in A \multimap \{b(x). x \in A\}$
by (*blast intro: lam-type*)

lemma *function-lam*: *function* $(\lambda x \in A. b(x))$
by (*simp add: function-def lam-def*)

lemma *relation-lam*: *relation* $(\lambda x \in A. b(x))$
by (*simp add: relation-def lam-def*)

lemma *beta-if [simp]*: $(\lambda x \in A. b(x)) \text{ ' } a = (\text{if } a \in A \text{ then } b(a) \text{ else } 0)$
by (*simp add: apply-def lam-def, blast*)

lemma *beta*: $a \in A \implies (\lambda x \in A. b(x)) \text{ ' } a = b(a)$
by (*simp add: apply-def lam-def, blast*)

lemma *lam-empty [simp]*: $(\lambda x \in 0. b(x)) = 0$
by (*simp add: lam-def*)

lemma *domain-lam [simp]*: $\text{domain}(\text{Lambda}(A, b)) = A$
by (*simp add: lam-def, blast*)

lemma *lam-cong [cong]*:
 $\llbracket A = A'; \bigwedge x. x \in A' \implies b(x) = b'(x) \rrbracket \implies \text{Lambda}(A, b) = \text{Lambda}(A', b')$
by (*simp only: lam-def cong add: RepFun-cong*)

lemma *lam-theI*:

$(\bigwedge x. x \in A \implies \exists! y. Q(x,y)) \implies \exists f. \forall x \in A. Q(x, f'x)$
apply (*rule-tac* $x = \lambda x \in A. \text{THE } y. Q(x,y)$ **in** *exI*)
apply *simp*
apply (*blast intro: theI*)
done

lemma *lam-eqE*: $\llbracket (\lambda x \in A. f(x)) = (\lambda x \in A. g(x)); a \in A \rrbracket \implies f(a)=g(a)$
by (*fast intro!: lamI elim: equalityE lamE*)

lemma *Pi-empty1* [*simp*]: $Pi(0,A) = \{0\}$
by (*unfold Pi-def function-def, blast*)

lemma *singleton-fun* [*simp*]: $\{\langle a,b \rangle\} \in \{a\} \rightarrow \{b\}$
by (*unfold Pi-def function-def, blast*)

lemma *Pi-empty2* [*simp*]: $(A \rightarrow 0) = (\text{if } A=0 \text{ then } \{0\} \text{ else } 0)$
by (*unfold Pi-def function-def, force*)

lemma *fun-space-empty-iff* [*iff*]: $(A \rightarrow X)=0 \longleftrightarrow X=0 \wedge (A \neq 0)$
apply *auto*
apply (*fast intro!: equals0I intro: lam-type*)
done

8.4 Extensionality

lemma *fun-subset*:
 $\llbracket f \in Pi(A,B); g \in Pi(C,D); A \leq C; \bigwedge x. x \in A \implies f'x = g'x \rrbracket \implies f \leq g$
by (*force dest: Pi-memberD intro: apply-Pair*)

lemma *fun-extension*:
 $\llbracket f \in Pi(A,B); g \in Pi(A,D); \bigwedge x. x \in A \implies f'x = g'x \rrbracket \implies f=g$
by (*blast del: subsetI intro: subset-refl sym fun-subset*)

lemma *eta* [*simp*]: $f \in Pi(A,B) \implies (\lambda x \in A. f'x) = f$
apply (*rule fun-extension*)
apply (*auto simp add: lam-type apply-type beta*)
done

lemma *fun-extension-iff*:
 $\llbracket f \in Pi(A,B); g \in Pi(A,C) \rrbracket \implies (\forall a \in A. f'a = g'a) \longleftrightarrow f=g$
by (*blast intro: fun-extension*)

lemma *fun-subset-eq*: $\llbracket f \in Pi(A,B); g \in Pi(A,C) \rrbracket \implies f \subseteq g \longleftrightarrow (f = g)$

by (blast dest: apply-Pair
intro: fun-extension apply-equality [symmetric])

lemma Pi-lamE:
 assumes major: $f \in Pi(A,B)$
 and minor: $\bigwedge b. \llbracket \forall x \in A. b(x):B(x); f = (\lambda x \in A. b(x)) \rrbracket \implies P$
 shows P
 apply (rule minor)
 apply (rule-tac [2] eta [symmetric])
 apply (blast intro: major apply-type)+
 done

8.5 Images of Functions

lemma image-lam: $C \subseteq A \implies (\lambda x \in A. b(x)) \text{ `` } C = \{b(x). x \in C\}$
 by (unfold lam-def, blast)

lemma Repfun-function-if:
 $function(f)$
 $\implies \{f'x. x \in C\} = (if\ C \subseteq domain(f)\ then\ f''C\ else\ cons(0,f''C))$
 apply simp
 apply (intro conjI impI)
 apply (blast dest: function-apply-equality intro: function-apply-Pair)
 apply (rule equalityI)
 apply (blast intro!: function-apply-Pair apply-0)
 apply (blast dest: function-apply-equality intro: apply-0 [symmetric])
 done

lemma image-function:
 $\llbracket function(f); C \subseteq domain(f) \rrbracket \implies f''C = \{f'x. x \in C\}$
 by (simp add: Repfun-function-if)

lemma image-fun: $\llbracket f \in Pi(A,B); C \subseteq A \rrbracket \implies f''C = \{f'x. x \in C\}$
 apply (simp add: Pi-iff)
 apply (blast intro: image-function)
 done

lemma image-eq-UN:
 assumes $f: f \in Pi(A,B)$ $C \subseteq A$ shows $f''C = (\bigcup x \in C. \{f'x\})$
 by (auto simp add: image-fun [OF f])

lemma Pi-image-cons:
 $\llbracket f \in Pi(A,B); x \in A \rrbracket \implies f \text{ `` } cons(x,y) = cons(f'x, f''y)$
 by (blast dest: apply-equality apply-Pair)

8.6 Properties of $\text{restrict}(f, A)$

lemma *restrict-subset*: $\text{restrict}(f, A) \subseteq f$
by (*unfold restrict-def, blast*)

lemma *function-restrictI*:
 $\text{function}(f) \implies \text{function}(\text{restrict}(f, A))$
by (*unfold restrict-def function-def, blast*)

lemma *restrict-type2*: $\llbracket f \in \text{Pi}(C, B); A \leq C \rrbracket \implies \text{restrict}(f, A) \in \text{Pi}(A, B)$
by (*simp add: Pi-iff function-def restrict-def, blast*)

lemma *restrict*: $\text{restrict}(f, A) \text{ ' } a = (\text{if } a \in A \text{ then } f'a \text{ else } 0)$
by (*simp add: apply-def restrict-def, blast*)

lemma *restrict-empty* [*simp*]: $\text{restrict}(f, 0) = 0$
by (*unfold restrict-def, simp*)

lemma *restrict-iff*: $z \in \text{restrict}(r, A) \longleftrightarrow z \in r \wedge (\exists x \in A. \exists y. z = \langle x, y \rangle)$
by (*simp add: restrict-def*)

lemma *restrict-restrict* [*simp*]:
 $\text{restrict}(\text{restrict}(r, A), B) = \text{restrict}(r, A \cap B)$
by (*unfold restrict-def, blast*)

lemma *domain-restrict* [*simp*]: $\text{domain}(\text{restrict}(f, C)) = \text{domain}(f) \cap C$
unfolding *restrict-def*
apply (*auto simp add: domain-def*)
done

lemma *restrict-idem*: $f \subseteq \text{Sigma}(A, B) \implies \text{restrict}(f, A) = f$
by (*simp add: restrict-def, blast*)

lemma *domain-restrict-idem*:
 $\llbracket \text{domain}(r) \subseteq A; \text{relation}(r) \rrbracket \implies \text{restrict}(r, A) = r$
by (*simp add: restrict-def relation-def, blast*)

lemma *domain-restrict-lam* [*simp*]: $\text{domain}(\text{restrict}(\text{Lambda}(A, f), C)) = A \cap C$
unfolding *restrict-def lam-def*
apply (*rule equalityI*)
apply (*auto simp add: domain-iff*)
done

lemma *restrict-if* [*simp*]: $\text{restrict}(f, A) \text{ ' } a = (\text{if } a \in A \text{ then } f'a \text{ else } 0)$
by (*simp add: restrict apply-0*)

lemma *restrict-lam-eq*:
 $A \leq C \implies \text{restrict}(\lambda x \in C. b(x), A) = (\lambda x \in A. b(x))$

by (*unfold restrict-def lam-def, auto*)

lemma *fun-cons-restrict-eq*:

$f \in \text{cons}(a, b) \rightarrow B \implies f = \text{cons}(\langle a, f \cdot a \rangle, \text{restrict}(f, b))$

apply (*rule equalityI*)

prefer 2 **apply** (*blast intro: apply-Pair restrict-subset [THEN subsetD]*)

apply (*auto dest!: Pi-memberD simp add: restrict-def lam-def*)

done

8.7 Unions of Functions

lemma *function-Union*:

$\llbracket \forall x \in S. \text{function}(x);$
 $\forall x \in S. \forall y \in S. x \leq y \mid y \leq x \rrbracket$
 $\implies \text{function}(\bigcup(S))$

by (*unfold function-def, blast*)

lemma *fun-Union*:

$\llbracket \forall f \in S. \exists C D. f \in C \rightarrow D;$
 $\forall f \in S. \forall y \in S. f \leq y \mid y \leq f \rrbracket \implies$
 $\bigcup(S) \in \text{domain}(\bigcup(S)) \rightarrow \text{range}(\bigcup(S))$

unfolding *Pi-def*

apply (*blast intro!: rel-Union function-Union*)

done

lemma *gen-relation-Union*:

$(\bigwedge f. f \in F \implies \text{relation}(f)) \implies \text{relation}(\bigcup(F))$

by (*simp add: relation-def*)

lemmas *Un-rls = Un-subset-iff SUM-Un-distrib1 prod-Un-distrib2*

subset-trans [OF - Un-upper1]

subset-trans [OF - Un-upper2]

lemma *fun-disjoint-Un*:

$\llbracket f \in A \rightarrow B; g \in C \rightarrow D; A \cap C = \emptyset \rrbracket$
 $\implies (f \cup g) \in (A \cup C) \rightarrow (B \cup D)$

apply (*simp add: Pi-iff extension Un-rls*)

apply (*unfold function-def, blast*)

done

lemma *fun-disjoint-apply1*: $a \notin \text{domain}(g) \implies (f \cup g) \cdot a = f \cdot a$

by (*simp add: apply-def, blast*)

lemma *fun-disjoint-apply2*: $c \notin \text{domain}(f) \implies (f \cup g) \cdot c = g \cdot c$

by (*simp add: apply-def, blast*)

8.8 Domain and Range of a Function or Relation

lemma *domain-of-fun*: $f \in Pi(A,B) \implies domain(f)=A$
by (*unfold Pi-def*, *blast*)

lemma *apply-rangeI*: $\llbracket f \in Pi(A,B); a \in A \rrbracket \implies f'a \in range(f)$
by (*erule apply-Pair [THEN rangeI]*, *assumption*)

lemma *range-of-fun*: $f \in Pi(A,B) \implies f \in A \rightarrow range(f)$
by (*blast intro: Pi-type apply-rangeI*)

8.9 Extensions of Functions

lemma *fun-extend*:

$\llbracket f \in A \rightarrow B; c \notin A \rrbracket \implies cons(\langle c,b \rangle, f) \in cons(c,A) \rightarrow cons(b,B)$
apply (*frule singleton-fun [THEN fun-disjoint-Un]*, *blast*)
apply (*simp add: cons-eq*)
done

lemma *fun-extend3*:

$\llbracket f \in A \rightarrow B; c \notin A; b \in B \rrbracket \implies cons(\langle c,b \rangle, f) \in cons(c,A) \rightarrow B$
by (*blast intro: fun-extend [THEN fun-weaken-type]*)

lemma *extend-apply*:

$c \notin domain(f) \implies cons(\langle c,b \rangle, f)'a = (if\ a=c\ then\ b\ else\ f'a)$
by (*auto simp add: apply-def*)

lemma *fun-extend-apply [simp]*:

$\llbracket f \in A \rightarrow B; c \notin A \rrbracket \implies cons(\langle c,b \rangle, f)'a = (if\ a=c\ then\ b\ else\ f'a)$
apply (*rule extend-apply*)
apply (*simp add: Pi-def*, *blast*)
done

lemmas *singleton-apply = apply-equality [OF singletonI singleton-fun, simp]*

lemma *cons-fun-eq*:

$c \notin A \implies cons(c,A) \rightarrow B = (\bigcup f \in A \rightarrow B. \bigcup b \in B. \{cons(\langle c,b \rangle, f)\})$
apply (*rule equalityI*)
apply (*safe elim!: fun-extend3*)

apply (*subgoal-tac restrict (x, A) \in A \rightarrow B*)

prefer 2 **apply** (*blast intro: restrict-type2*)

apply (*rule UN-I, assumption*)

apply (*rule apply-funtype [THEN UN-I]*)

apply *assumption*

apply (*rule consI1*)

apply (*simp (no-asm)*)

apply (*rule fun-extension*)

apply *assumption*

```

  apply (blast intro: fun-extend)
apply (erule consE, simp-all)
done

```

```

lemma succ-fun-eq: succ(n) -> B = (∪ f ∈ n->B. ∪ b ∈ B. {cons(⟨n,b⟩, f)})
by (simp add: succ-def mem-not-refl cons-fun-eq)

```

8.10 Function Updates

definition

```

update :: [i,i,i] ⇒ i where
  update(f,a,b) ≡ λx ∈ cons(a, domain(f)). if(x=a, b, f x)

```

nonterminal updbinds and updbind

syntax

```

-updbind    :: [i, i] ⇒ updbind  (⟨⟨indent=2 notation=⟨infix update⟩⟩- := / -⟩⟩)
              :: updbind ⇒ updbinds (⟨-⟩)
-updbinds   :: [updbind, updbinds] ⇒ updbinds (⟨-, / -⟩)
-Update     :: [i, updbinds] ⇒ i  (⟨⟨open-block notation=⟨mixfix function up-
date⟩⟩-/'((-)')⟩ [900,0] 900)

```

syntax-consts

```

-Update ⇔ update

```

translations

```

-Update (f, -updbinds(b,bs)) == -Update (-Update(f,b), bs)
f(x:=y)                        == CONST update(f,x,y)

```

```

lemma update-apply [simp]: f(x:=y) ‘ z = (if z=x then y else f z)
apply (simp add: update-def)
apply (case-tac z ∈ domain(f))
apply (simp-all add: apply-0)
done

```

```

lemma update-idem: [f x = y; f ∈ Pi(A,B); x ∈ A] ⇒ f(x:=y) = f
  unfolding update-def
apply (simp add: domain-of-fun cons-absorb)
apply (rule fun-extension)
apply (best intro: apply-type if-type lam-type, assumption, simp)
done

```

```

declare refl [THEN update-idem, simp]

```

```

lemma domain-update [simp]: domain(f(x:=y)) = cons(x, domain(f))
by (unfold update-def, simp)

```

```

lemma update-type: [f ∈ Pi(A,B); x ∈ A; y ∈ B(x)] ⇒ f(x:=y) ∈ Pi(A, B)
  unfolding update-def

```


apply (*simp add: domain-of-fun cons-absorb apply-funtype lam-type*)
done

8.11 Monotonicity Theorems

8.11.1 Replacement in its Various Forms

lemma *Replace-mono*: $A \leq B \implies \text{Replace}(A, P) \subseteq \text{Replace}(B, P)$
by (*blast elim!: ReplaceE*)

lemma *RepFun-mono*: $A \leq B \implies \{f(x). x \in A\} \subseteq \{f(x). x \in B\}$
by *blast*

lemma *Pow-mono*: $A \leq B \implies \text{Pow}(A) \subseteq \text{Pow}(B)$
by *blast*

lemma *Union-mono*: $A \leq B \implies \bigcup(A) \subseteq \bigcup(B)$
by *blast*

lemma *UN-mono*:
 $\llbracket A \leq C; \bigwedge x. x \in A \implies B(x) \leq D(x) \rrbracket \implies (\bigcup_{x \in A} B(x)) \subseteq (\bigcup_{x \in C} D(x))$
by *blast*

lemma *Inter-anti-mono*: $\llbracket A \leq B; A \neq \emptyset \rrbracket \implies \bigcap(B) \subseteq \bigcap(A)$
by *blast*

lemma *cons-mono*: $C \leq D \implies \text{cons}(a, C) \subseteq \text{cons}(a, D)$
by *blast*

lemma *Un-mono*: $\llbracket A \leq C; B \leq D \rrbracket \implies A \cup B \subseteq C \cup D$
by *blast*

lemma *Int-mono*: $\llbracket A \leq C; B \leq D \rrbracket \implies A \cap B \subseteq C \cap D$
by *blast*

lemma *Diff-mono*: $\llbracket A \leq C; D \leq B \rrbracket \implies A - B \subseteq C - D$
by *blast*

8.11.2 Standard Products, Sums and Function Spaces

lemma *Sigma-mono* [*rule-format*]:
 $\llbracket A \leq C; \bigwedge x. x \in A \implies B(x) \subseteq D(x) \rrbracket \implies \text{Sigma}(A, B) \subseteq \text{Sigma}(C, D)$
by *blast*

lemma *sum-mono*: $\llbracket A \leq C; B \leq D \rrbracket \implies A + B \subseteq C + D$
by (*unfold sum-def, blast*)

lemma *Pi-mono*: $B \leq C \implies A \multimap B \subseteq A \multimap C$

by (*blast intro: lam-type elim: Pi-lamE*)

lemma *lam-mono*: $A \leq B \implies \text{Lambda}(A, c) \subseteq \text{Lambda}(B, c)$
unfolding *lam-def*
apply (*erule RepFun-mono*)
done

8.11.3 Converse, Domain, Range, Field

lemma *converse-mono*: $r \leq s \implies \text{converse}(r) \subseteq \text{converse}(s)$
by *blast*

lemma *domain-mono*: $r \leq s \implies \text{domain}(r) \leq \text{domain}(s)$
by *blast*

lemmas *domain-rel-subset* = *subset-trans* [*OF domain-mono domain-subset*]

lemma *range-mono*: $r \leq s \implies \text{range}(r) \leq \text{range}(s)$
by *blast*

lemmas *range-rel-subset* = *subset-trans* [*OF range-mono range-subset*]

lemma *field-mono*: $r \leq s \implies \text{field}(r) \leq \text{field}(s)$
by *blast*

lemma *field-rel-subset*: $r \subseteq A * A \implies \text{field}(r) \subseteq A$
by (*erule field-mono* [*THEN subset-trans*], *blast*)

8.11.4 Images

lemma *image-pair-mono*:
 $\llbracket \bigwedge x y. \langle x, y \rangle : r \implies \langle x, y \rangle : s; \ A \leq B \rrbracket \implies r^{``A} \subseteq s^{``B}$
by *blast*

lemma *vimage-pair-mono*:
 $\llbracket \bigwedge x y. \langle x, y \rangle : r \implies \langle x, y \rangle : s; \ A \leq B \rrbracket \implies r^{-``A} \subseteq s^{-``B}$
by *blast*

lemma *image-mono*: $\llbracket r \leq s; \ A \leq B \rrbracket \implies r^{``A} \subseteq s^{``B}$
by *blast*

lemma *vimage-mono*: $\llbracket r \leq s; \ A \leq B \rrbracket \implies r^{-``A} \subseteq s^{-``B}$
by *blast*

lemma *Collect-mono*:
 $\llbracket A \leq B; \ \bigwedge x. x \in A \implies P(x) \longrightarrow Q(x) \rrbracket \implies \text{Collect}(A, P) \subseteq \text{Collect}(B, Q)$
by *blast*

lemmas *basic-monos* = *subset-refl imp-refl disj-mono conj-mono ex-mono*

```

lemma bex-image-simp:
   $\llbracket f \in Pi(X, Y); A \subseteq X \rrbracket \implies (\exists x \in f^{-1} A. P(x)) \longleftrightarrow (\exists x \in A. P(f'x))$ 
  apply safe
  apply rule
  prefer 2 apply assumption
  apply (simp add: apply-equality)
  apply (blast intro: apply-Pair)
  done

lemma ball-image-simp:
   $\llbracket f \in Pi(X, Y); A \subseteq X \rrbracket \implies (\forall x \in f^{-1} A. P(x)) \longleftrightarrow (\forall x \in A. P(f'x))$ 
  apply safe
  apply (blast intro: apply-Pair)
  apply (drule bspec) apply assumption
  apply (simp add: apply-equality)
  done

end

```

9 Quine-Inspired Ordered Pairs and Disjoint Sums

theory *QPair* **imports** *Sum func* **begin**

For non-well-founded data structures in ZF. Does not precisely follow Quine's construction. Thanks to Thomas Forster for suggesting this approach!

W. V. Quine, On Ordered Pairs and Relations, in Selected Logic Papers, 1966.

definition

```

QPair    ::  $[i, i] \Rightarrow i$  ( $\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix Quine pair} \rangle \rangle \langle -; / - \rangle \rangle$ )
where  $\langle a; b \rangle \equiv a + b$ 

```

definition

```

qfst ::  $i \Rightarrow i$  where
  qfst( $p$ )  $\equiv$  THE  $a. \exists b. p = \langle a; b \rangle$ 

```

definition

```

qsnd ::  $i \Rightarrow i$  where
  qsnd( $p$ )  $\equiv$  THE  $b. \exists a. p = \langle a; b \rangle$ 

```

definition

```

qsplit ::  $[[i, i] \Rightarrow 'a, i] \Rightarrow 'a::\{\}$  where
  qsplit( $c, p$ )  $\equiv$   $c(\text{qfst}(p), \text{qsnd}(p))$ 

```

definition

qconverse :: $i \Rightarrow i$ **where**
 $qconverse(r) \equiv \{z. w \in r, \exists x y. w = \langle x; y \rangle \wedge z = \langle y; x \rangle\}$

definition

QSigma :: $[i, i \Rightarrow i] \Rightarrow i$ **where**
 $QSigma(A, B) \equiv \bigcup_{x \in A} \bigcup_{y \in B(x)} \{\langle x; y \rangle\}$

syntax

-*QSUM* :: $[idt, i, i] \Rightarrow i$ ($\langle \langle indent=3 \text{ notation}=\text{binder } QSUM \in \rangle \rangle QSUM - \in$
 $-. / - \rangle 10$)

syntax-consts

-*QSUM* \equiv *QSigma*

translations

QSUM $x \in A. B \Rightarrow$ *CONST* *QSigma*(*A*, $\lambda x. B$)

abbreviation

qprod (**infixr** $\langle \langle * \rangle \rangle 80$) **where**
 $A \langle * \rangle B \equiv QSigma(A, \lambda -. B)$

definition

qsum :: $[i, i] \Rightarrow i$ (**infixr** $\langle \langle + \rangle \rangle 65$) **where**
 $A \langle + \rangle B \equiv (\{0\} \langle * \rangle A) \cup (\{1\} \langle * \rangle B)$

definition

QInl :: $i \Rightarrow i$ **where**
 $QInl(a) \equiv \langle 0; a \rangle$

definition

QInr :: $i \Rightarrow i$ **where**
 $QInr(b) \equiv \langle 1; b \rangle$

definition

qcase :: $[i \Rightarrow i, i \Rightarrow i, i] \Rightarrow i$ **where**
 $qcase(c, d) \equiv qsplit(\lambda y z. cond(y, d(z), c(z)))$

9.1 Quine ordered pairing

lemma *QPair-empty* [*simp*]: $\langle 0; 0 \rangle = 0$

by (*simp add: QPair-def*)

lemma *QPair-iff* [*simp*]: $\langle a; b \rangle = \langle c; d \rangle \longleftrightarrow a = c \wedge b = d$

apply (*simp add: QPair-def*)

apply (*rule sum-equal-iff*)

done

lemmas *QPair-inject* = *QPair-iff* [*THEN iffD1, THEN conjE, elim!*]

lemma *QPair-inject1*: $\langle a; b \rangle = \langle c; d \rangle \implies a = c$

by *blast*

lemma *QPair-inject2*: $\langle a;b \rangle = \langle c;d \rangle \implies b=d$
by *blast*

9.1.1 QSigma: Disjoint union of a family of sets Generalizes Cartesian product

lemma *QSigmaI* [*intro!*]: $\llbracket a \in A; b \in B(a) \rrbracket \implies \langle a;b \rangle \in QSigma(A,B)$
by (*simp add: QSigma-def*)

lemma *QSigmaE* [*elim!*]:
 $\llbracket c \in QSigma(A,B); \bigwedge x y. \llbracket x \in A; y \in B(x); c = \langle x;y \rangle \rrbracket \implies P$
 $\rrbracket \implies P$
by (*simp add: QSigma-def, blast*)

lemma *QSigmaE2* [*elim!*]:
 $\llbracket \langle a;b \rangle \in QSigma(A,B); \llbracket a \in A; b \in B(a) \rrbracket \implies P \rrbracket \implies P$
by (*simp add: QSigma-def*)

lemma *QSigmaD1*: $\langle a;b \rangle \in QSigma(A,B) \implies a \in A$
by *blast*

lemma *QSigmaD2*: $\langle a;b \rangle \in QSigma(A,B) \implies b \in B(a)$
by *blast*

lemma *QSigma-cong*:
 $\llbracket A=A'; \bigwedge x. x \in A' \implies B(x)=B'(x) \rrbracket \implies$
 $QSigma(A,B) = QSigma(A',B')$
by (*simp add: QSigma-def*)

lemma *QSigma-empty1* [*simp*]: $QSigma(0,B) = 0$
by *blast*

lemma *QSigma-empty2* [*simp*]: $A \langle * \rangle 0 = 0$
by *blast*

9.1.2 Projections: qfst, qsnd

lemma *qfst-conv* [*simp*]: $qfst(\langle a;b \rangle) = a$
by (*simp add: qfst-def*)

lemma *qsnd-conv* [*simp*]: $qsnd(\langle a;b \rangle) = b$
by (*simp add: qsnd-def*)

lemma *qfst-type* [*TC*]: $p \in QSigma(A,B) \implies qfst(p) \in A$
by *auto*

lemma *qsnd-type* [TC]: $p \in QSigma(A,B) \implies qsnd(p) \in B(qfst(p))$
by *auto*

lemma *QPair-qfst-qsnd-eq*: $a \in QSigma(A,B) \implies \langle qfst(a); qsnd(a) \rangle = a$
by *auto*

9.1.3 Eliminator: qsplit

lemma *qsplit* [simp]: $qsplit(\lambda x y. c(x,y), \langle a;b \rangle) \equiv c(a,b)$
by (*simp add: qsplit-def*)

lemma *qsplit-type* [elim!]:

$$\llbracket p \in QSigma(A,B);$$

$$\bigwedge x y. \llbracket x \in A; y \in B(x) \rrbracket \implies c(x,y):C(\langle x;y \rangle)$$

$$\rrbracket \implies qsplit(\lambda x y. c(x,y), p) \in C(p)$$

by *auto*

lemma *expand-qsplit*:
 $u \in A \langle * \rangle B \implies R(qsplit(c,u)) \longleftrightarrow (\forall x \in A. \forall y \in B. u = \langle x;y \rangle \longrightarrow R(c(x,y)))$
apply (*simp add: qsplit-def, auto*)
done

9.1.4 qsplit for predicates: result type o

lemma *qsplitI*: $R(a,b) \implies qsplit(R, \langle a;b \rangle)$
by (*simp add: qsplit-def*)

lemma *qsplitE*:

$$\llbracket qsplit(R,z); z \in QSigma(A,B);$$

$$\bigwedge x y. \llbracket z = \langle x;y \rangle; R(x,y) \rrbracket \implies P$$

$$\rrbracket \implies P$$

by (*simp add: qsplit-def, auto*)

lemma *qsplitD*: $qsplit(R, \langle a;b \rangle) \implies R(a,b)$
by (*simp add: qsplit-def*)

9.1.5 qconverse

lemma *qconverseI* [intro!]: $\langle a;b \rangle : r \implies \langle b;a \rangle : qconverse(r)$
by (*simp add: qconverse-def, blast*)

lemma *qconverseD* [elim!]: $\langle a;b \rangle \in qconverse(r) \implies \langle b;a \rangle \in r$
by (*simp add: qconverse-def, blast*)

lemma *qconverseE* [elim!]:

$$\llbracket yx \in qconverse(r);$$

$$\bigwedge x y. \llbracket yx = \langle y;x \rangle; \langle x;y \rangle : r \rrbracket \implies P$$

$\llbracket \implies P$
by (*simp add: qconverse-def, blast*)

lemma *qconverse-qconverse*: $r \leq Q\text{Sigma}(A,B) \implies qconverse(qconverse(r)) = r$
by *blast*

lemma *qconverse-type*: $r \subseteq A <*> B \implies qconverse(r) \subseteq B <*> A$
by *blast*

lemma *qconverse-prod*: $qconverse(A <*> B) = B <*> A$
by *blast*

lemma *qconverse-empty*: $qconverse(0) = 0$
by *blast*

9.2 The Quine-inspired notion of disjoint sum

lemmas *qsum-defs = qsum-def QInl-def QInr-def qcase-def*

lemma *QInlI [intro!]*: $a \in A \implies QInl(a) \in A <+> B$
by (*simp add: qsum-defs, blast*)

lemma *QInrI [intro!]*: $b \in B \implies QInr(b) \in A <+> B$
by (*simp add: qsum-defs, blast*)

lemma *qsumE [elim!]*:

$$\begin{aligned} & \llbracket u \in A <+> B; \\ & \quad \bigwedge x. \llbracket x \in A; u = QInl(x) \rrbracket \implies P; \\ & \quad \bigwedge y. \llbracket y \in B; u = QInr(y) \rrbracket \implies P \\ & \rrbracket \implies P \end{aligned}$$
by (*simp add: qsum-defs, blast*)

lemma *QInl-iff [iff]*: $QInl(a) = QInl(b) \longleftrightarrow a = b$
by (*simp add: qsum-defs*)

lemma *QInr-iff [iff]*: $QInr(a) = QInr(b) \longleftrightarrow a = b$
by (*simp add: qsum-defs*)

lemma *QInl-QInr-iff [simp]*: $QInl(a) = QInr(b) \longleftrightarrow \text{False}$
by (*simp add: qsum-defs*)

lemma *QInr-QInl-iff [simp]*: $QInr(b) = QInl(a) \longleftrightarrow \text{False}$

by (*simp add: qsum-defs*)

lemma *qsum-empty* [*simp*]: $0 <+> 0 = 0$
by (*simp add: qsum-defs*)

lemmas *QInl-inject* = *QInl-iff* [*THEN iffD1*]
lemmas *QInr-inject* = *QInr-iff* [*THEN iffD1*]
lemmas *QInl-neq-QInr* = *QInl-QInr-iff* [*THEN iffD1, THEN FalseE, elim!*]
lemmas *QInr-neq-QInl* = *QInr-QInl-iff* [*THEN iffD1, THEN FalseE, elim!*]

lemma *QInlD*: $QInl(a): A <+> B \implies a \in A$
by *blast*

lemma *QInrD*: $QInr(b): A <+> B \implies b \in B$
by *blast*

lemma *qsum-iff*:
 $u \in A <+> B \longleftrightarrow (\exists x. x \in A \wedge u = QInl(x)) \mid (\exists y. y \in B \wedge u = QInr(y))$
by *blast*

lemma *qsum-subset-iff*: $A <+> B \subseteq C <+> D \longleftrightarrow A \leq C \wedge B \leq D$
by *blast*

lemma *qsum-equal-iff*: $A <+> B = C <+> D \longleftrightarrow A = C \wedge B = D$
apply (*simp (no-asm) add: extension qsum-subset-iff*)
apply *blast*
done

9.2.1 Eliminator – **qcase**

lemma *qcase-QInl* [*simp*]: $qcase(c, d, QInl(a)) = c(a)$
by (*simp add: qsum-defs*)

lemma *qcase-QInr* [*simp*]: $qcase(c, d, QInr(b)) = d(b)$
by (*simp add: qsum-defs*)

lemma *qcase-type*:

$$\llbracket u \in A <+> B;$$

$$\bigwedge x. x \in A \implies c(x): C(QInl(x));$$

$$\bigwedge y. y \in B \implies d(y): C(QInr(y))$$

$$\rrbracket \implies qcase(c, d, u) \in C(u)$$
by (*simp add: qsum-defs, auto*)

lemma *Part-QInl*: $Part(A <+> B, QInl) = \{QInl(x). x \in A\}$
by *blast*

lemma *Part-QInr*: $Part(A <+> B, QInr) = \{QInr(y). y \in B\}$
by *blast*

lemma *Part-QInr2*: $Part(A <+> B, \lambda x. QInr(h(x))) = \{QInr(y). y \in Part(B, h)\}$
by *blast*

lemma *Part-qsum-equality*: $C \subseteq A <+> B \implies Part(C, QInl) \cup Part(C, QInr) = C$
by *blast*

9.2.2 Monotonicity

lemma *QPair-mono*: $\llbracket a \leq c; b \leq d \rrbracket \implies \langle a; b \rangle \subseteq \langle c; d \rangle$
by (*simp add: QPair-def sum-mono*)

lemma *QSigma-mono* [*rule-format*]:
 $\llbracket A \leq C; \forall x \in A. B(x) \subseteq D(x) \rrbracket \implies QSigma(A, B) \subseteq QSigma(C, D)$
by *blast*

lemma *QInl-mono*: $a \leq b \implies QInl(a) \subseteq QInl(b)$
by (*simp add: QInl-def subset-refl [THEN QPair-mono]*)

lemma *QInr-mono*: $a \leq b \implies QInr(a) \subseteq QInr(b)$
by (*simp add: QInr-def subset-refl [THEN QPair-mono]*)

lemma *qsum-mono*: $\llbracket A \leq C; B \leq D \rrbracket \implies A <+> B \subseteq C <+> D$
by *blast*

end

10 Injections, Surjections, Bijections, Composition

theory *Perm* **imports** *func* **begin**

definition

comp $:: [i, i] \Rightarrow i$ (**infixr** $\langle O \rangle$ 60) **where**
 $r \ O \ s \equiv \{xz \in domain(s) * range(r) .$
 $\exists x \ y \ z. xz = \langle x, z \rangle \wedge \langle x, y \rangle : s \wedge \langle y, z \rangle : r\}$

definition

id $:: i \Rightarrow i$ **where**
 $id(A) \equiv (\lambda x \in A. x)$

definition

$inj :: [i,i] \Rightarrow i$ **where**
 $inj(A,B) \equiv \{ f \in A \multimap B. \forall w \in A. \forall x \in A. f'w = f'x \longrightarrow w = x \}$

definition

$surj :: [i,i] \Rightarrow i$ **where**
 $surj(A,B) \equiv \{ f \in A \multimap B. \forall y \in B. \exists x \in A. f'x = y \}$

definition

$bij :: [i,i] \Rightarrow i$ **where**
 $bij(A,B) \equiv inj(A,B) \cap surj(A,B)$

10.1 Surjective Function Space

lemma *surj-is-fun*: $f \in surj(A,B) \Longrightarrow f \in A \multimap B$
unfolding *surj-def*
apply (*erule CollectD1*)
done

lemma *fun-is-surj*: $f \in Pi(A,B) \Longrightarrow f \in surj(A, range(f))$
unfolding *surj-def*
apply (*blast intro: apply-equality range-of-fun domain-type*)
done

lemma *surj-range*: $f \in surj(A,B) \Longrightarrow range(f) = B$
unfolding *surj-def*
apply (*best intro: apply-Pair elim: range-type*)
done

A function with a right inverse is a surjection

lemma *f-imp-surjective*:
 $\llbracket f \in A \multimap B; \bigwedge y. y \in B \Longrightarrow d(y): A; \bigwedge y. y \in B \Longrightarrow f'd(y) = y \rrbracket$
 $\Longrightarrow f \in surj(A,B)$
by (*simp add: surj-def, blast*)

lemma *lam-surjective*:
 $\llbracket \bigwedge x. x \in A \Longrightarrow c(x): B; \bigwedge y. y \in B \Longrightarrow d(y): A; \bigwedge y. y \in B \Longrightarrow c(d(y)) = y \rrbracket$
 $\Longrightarrow (\lambda x \in A. c(x)) \in surj(A,B)$
apply (*rule-tac d = d in f-imp-surjective*)
apply (*simp-all add: lam-type*)
done

Cantor's theorem revisited

lemma *cantor-surj*: $f \notin surj(A, Pow(A))$

```

apply (unfold surj-def, safe)
apply (cut-tac cantor)
apply (best del: subsetI)
done

```

10.2 Injective Function Space

```

lemma inj-is-fun:  $f \in \text{inj}(A,B) \implies f \in A \multimap B$ 
  unfolding inj-def
apply (erule CollectD1)
done

```

Good for dealing with sets of pairs, but a bit ugly in use [used in AC]

```

lemma inj-equality:
   $\llbracket \langle a,b \rangle : f; \langle c,b \rangle : f; f \in \text{inj}(A,B) \rrbracket \implies a=c$ 
  unfolding inj-def
apply (blast dest: Pair-mem-PiD)
done

```

```

lemma inj-apply-equality:  $\llbracket f \in \text{inj}(A,B); f'a=f'b; a \in A; b \in A \rrbracket \implies a=b$ 
by (unfold inj-def, blast)

```

A function with a left inverse is an injection

```

lemma f-imp-injective:  $\llbracket f \in A \multimap B; \forall x \in A. d(f'x)=x \rrbracket \implies f \in \text{inj}(A,B)$ 
apply (simp (no-asm-simp) add: inj-def)
apply (blast intro: subst-context [THEN box-equals])
done

```

```

lemma lam-injective:
   $\llbracket \bigwedge x. x \in A \implies c(x) : B; \bigwedge x. x \in A \implies d(c(x)) = x \rrbracket$ 
   $\implies (\lambda x \in A. c(x)) \in \text{inj}(A,B)$ 
apply (rule-tac d = d in f-imp-injective)
apply (simp-all add: lam-type)
done

```

10.3 Bijections

```

lemma bij-is-inj:  $f \in \text{bij}(A,B) \implies f \in \text{inj}(A,B)$ 
  unfolding bij-def
apply (erule IntD1)
done

```

```

lemma bij-is-surj:  $f \in \text{bij}(A,B) \implies f \in \text{surj}(A,B)$ 
  unfolding bij-def
apply (erule IntD2)
done

```

```

lemma bij-is-fun:  $f \in \text{bij}(A,B) \implies f \in A \multimap B$ 

```

```

by (rule bij-is-inj [THEN inj-is-fun])

lemma lam-bijective:
  [[ $\bigwedge x. x \in A \implies c(x): B;$ 
     $\bigwedge y. y \in B \implies d(y): A;$ 
     $\bigwedge x. x \in A \implies d(c(x)) = x;$ 
     $\bigwedge y. y \in B \implies c(d(y)) = y$ 
  ]]  $\implies (\lambda x \in A. c(x)) \in \text{bij}(A, B)$ 
  unfolding bij-def
apply (blast intro!: lam-injective lam-surjective)
done

lemma RepFun-bijective: ( $\forall y \in x. \exists ! y'. f(y') = f(y)$ )
   $\implies (\lambda z \in \{f(y). y \in x\}. \text{THE } y. f(y) = z) \in \text{bij}(\{f(y). y \in x\}, x)$ 
apply (rule-tac d = f in lam-bijective)
apply (auto simp add: the-equality2)
done



### 10.4 Identity Function



lemma idI [intro!]:  $a \in A \implies \langle a, a \rangle \in \text{id}(A)$ 
  unfolding id-def
apply (erule lamI)
done

lemma idE [elim!]: [ $p \in \text{id}(A); \bigwedge x. [x \in A; p = \langle x, x \rangle] \implies P$ ]  $\implies P$ 
by (simp add: id-def lam-def, blast)

lemma id-type:  $\text{id}(A) \in A \multimap A$ 
  unfolding id-def
apply (rule lam-type, assumption)
done

lemma id-conv [simp]:  $x \in A \implies \text{id}(A) 'x = x$ 
  unfolding id-def
apply (simp (no-asm-simp))
done

lemma id-mono:  $A \leq B \implies \text{id}(A) \subseteq \text{id}(B)$ 
  unfolding id-def
apply (erule lam-mono)
done

lemma id-subset-inj:  $A \leq B \implies \text{id}(A): \text{inj}(A, B)$ 
apply (simp add: inj-def id-def)
apply (blast intro: lam-type)
done

lemmas id-inj = subset-refl [THEN id-subset-inj]

```

lemma *id-surj*: $id(A): surj(A,A)$
unfolding *id-def surj-def*
apply (*simp (no-asm-simp)*)
done

lemma *id-bij*: $id(A): bij(A,A)$
unfolding *bij-def*
apply (*blast intro: id-inj id-surj*)
done

lemma *subset-iff-id*: $A \subseteq B \longleftrightarrow id(A) \in A \multimap B$
unfolding *id-def*
apply (*force intro!: lam-type dest: apply-type*)
done

id as the identity relation

lemma *id-iff* [*simp*]: $\langle x,y \rangle \in id(A) \longleftrightarrow x=y \wedge y \in A$
by *auto*

10.5 Converse of a Function

lemma *inj-converse-fun*: $f \in inj(A,B) \implies converse(f) \in range(f) \multimap A$
unfolding *inj-def*
apply (*simp (no-asm-simp) add: Pi-iff function-def*)
apply (*erule CollectE*)
apply (*simp (no-asm-simp) add: apply-iff*)
apply (*blast dest: fun-is-rel*)
done

Equations for $converse(f)$

The premises are equivalent to saying that f is injective...

lemma *left-inverse-lemma*:
 $\llbracket f \in A \multimap B; converse(f): C \multimap A; a \in A \rrbracket \implies converse(f) (f'a) = a$
by (*blast intro: apply-Pair apply-equality converseI*)

lemma *left-inverse* [*simp*]: $\llbracket f \in inj(A,B); a \in A \rrbracket \implies converse(f) (f'a) = a$
by (*blast intro: left-inverse-lemma inj-converse-fun inj-is-fun*)

lemma *left-inverse-eq*:
 $\llbracket f \in inj(A,B); f'a = y; x \in A \rrbracket \implies converse(f) 'y = x$
by *auto*

lemmas *left-inverse-bij = bij-is-inj* [*THEN left-inverse*]

lemma *right-inverse-lemma*:
 $\llbracket f \in A \multimap B; converse(f): C \multimap A; b \in C \rrbracket \implies f(converse(f) 'b) = b$
by (*rule apply-Pair* [*THEN converseD* [*THEN apply-equality*]], *auto*)

lemma *right-inverse* [*simp*]:

$\llbracket f \in \text{inj}(A,B); \ b \in \text{range}(f) \rrbracket \implies f'(\text{converse}(f) \cdot b) = b$

by (*blast intro: right-inverse-lemma inj-converse-fun inj-is-fun*)

lemma *right-inverse-bij*: $\llbracket f \in \text{bij}(A,B); \ b \in B \rrbracket \implies f'(\text{converse}(f) \cdot b) = b$

by (*force simp add: bij-def surj-range*)

10.6 Converses of Injections, Surjections, Bijections

lemma *inj-converse-inj*: $f \in \text{inj}(A,B) \implies \text{converse}(f): \text{inj}(\text{range}(f), A)$

apply (*rule f-imp-injective*)

apply (*erule inj-converse-fun, clarify*)

apply (*rule right-inverse*)

apply *assumption*

apply *blast*

done

lemma *inj-converse-surj*: $f \in \text{inj}(A,B) \implies \text{converse}(f): \text{surj}(\text{range}(f), A)$

by (*blast intro: f-imp-surjective inj-converse-fun left-inverse inj-is-fun
range-of-fun [THEN apply-type]*)

Adding this as an intro! rule seems to cause looping

lemma *bij-converse-bij* [*TC*]: $f \in \text{bij}(A,B) \implies \text{converse}(f): \text{bij}(B,A)$

unfolding *bij-def*

apply (*fast elim: surj-range [THEN subst] inj-converse-inj inj-converse-surj*)

done

10.7 Composition of Two Relations

The inductive definition package could derive these theorems for $r \ O \ s$

lemma *compI* [*intro*]: $\llbracket \langle a,b \rangle : s; \ \langle b,c \rangle : r \rrbracket \implies \langle a,c \rangle \in r \ O \ s$

by (*unfold comp-def, blast*)

lemma *compE* [*elim!*]:

$\llbracket xz \in r \ O \ s;$

$\bigwedge x \ y \ z. \llbracket xz = \langle x,z \rangle; \ \langle x,y \rangle : s; \ \langle y,z \rangle : r \rrbracket \implies P \rrbracket$

$\implies P$

by (*unfold comp-def, blast*)

lemma *compEpair*:

$\llbracket \langle a,c \rangle \in r \ O \ s;$

$\bigwedge y. \llbracket \langle a,y \rangle : s; \ \langle y,c \rangle : r \rrbracket \implies P \rrbracket$

$\implies P$

by (*erule compE, simp*)

lemma *converse-comp*: $\text{converse}(R \ O \ S) = \text{converse}(S) \ O \ \text{converse}(R)$

by *blast*

10.8 Domain and Range – see Suppes, Section 3.1

Boyer et al., Set Theory in First-Order Logic, JAR 2 (1986), 287-327

lemma *range-comp*: $\text{range}(r \circ s) \subseteq \text{range}(r)$
by *blast*

lemma *range-comp-eq*: $\text{domain}(r) \subseteq \text{range}(s) \implies \text{range}(r \circ s) = \text{range}(r)$
by (*rule range-comp [THEN equalityI], blast*)

lemma *domain-comp*: $\text{domain}(r \circ s) \subseteq \text{domain}(s)$
by *blast*

lemma *domain-comp-eq*: $\text{range}(s) \subseteq \text{domain}(r) \implies \text{domain}(r \circ s) = \text{domain}(s)$
by (*rule domain-comp [THEN equalityI], blast*)

lemma *image-comp*: $(r \circ s)''A = r''(s''A)$
by *blast*

lemma *inj-inj-range*: $f \in \text{inj}(A, B) \implies f \in \text{inj}(A, \text{range}(f))$
by (*auto simp add: inj-def Pi-iff function-def*)

lemma *inj-bij-range*: $f \in \text{inj}(A, B) \implies f \in \text{bij}(A, \text{range}(f))$
by (*auto simp add: bij-def intro: inj-inj-range inj-is-fun fun-is-surj*)

10.9 Other Results

lemma *comp-mono*: $\llbracket r' \leq r; s' \leq s \rrbracket \implies (r' \circ s') \subseteq (r \circ s)$
by *blast*

composition preserves relations

lemma *comp-rel*: $\llbracket s \leq A * B; r \leq B * C \rrbracket \implies (r \circ s) \subseteq A * C$
by *blast*

associative law for composition

lemma *comp-assoc*: $(r \circ s) \circ t = r \circ (s \circ t)$
by *blast*

lemma *left-comp-id*: $r \leq A * B \implies \text{id}(B) \circ r = r$
by *blast*

lemma *right-comp-id*: $r \leq A * B \implies r \circ \text{id}(A) = r$
by *blast*

10.10 Composition Preserves Functions, Injections, and Surjections

lemma *comp-function*: $\llbracket \text{function}(g); \text{function}(f) \rrbracket \implies \text{function}(f \circ g)$

by (*unfold function-def*, *blast*)

Don't think the premises can be weakened much

```
lemma comp-fun:  $\llbracket g \in A \multimap B; f \in B \multimap C \rrbracket \implies (f \circ g) \in A \multimap C$ 
apply (auto simp add: Pi-def comp-function Pow-iff comp-rel)
apply (subst range-rel-subset [THEN domain-comp-eq], auto)
done
```

```
lemma comp-fun-apply [simp]:
   $\llbracket g \in A \multimap B; a \in A \rrbracket \implies (f \circ g) 'a = f '(g 'a)$ 
apply (frule apply-Pair, assumption)
apply (simp add: apply-def image-comp)
apply (blast dest: apply-equality)
done
```

Simplifies compositions of lambda-abstractions

```
lemma comp-lam:
   $\llbracket \bigwedge x. x \in A \implies b(x) : B \rrbracket$ 
   $\implies (\lambda y \in B. c(y)) \circ (\lambda x \in A. b(x)) = (\lambda x \in A. c(b(x)))$ 
apply (subgoal-tac  $(\lambda x \in A. b(x)) \in A \multimap B$ )
apply (rule fun-extension)
  apply (blast intro: comp-fun lam-funtype)
  apply (rule lam-funtype)
apply simp
apply (simp add: lam-type)
done
```

```
lemma comp-inj:
   $\llbracket g \in \text{inj}(A, B); f \in \text{inj}(B, C) \rrbracket \implies (f \circ g) \in \text{inj}(A, C)$ 
apply (frule inj-is-fun [of g])
apply (frule inj-is-fun [of f])
apply (rule-tac  $d = \lambda y. \text{converse } (g) ' (\text{converse } (f) ' y)$  in f-imp-injective)
  apply (blast intro: comp-fun, simp)
done
```

```
lemma comp-surj:
   $\llbracket g \in \text{surj}(A, B); f \in \text{surj}(B, C) \rrbracket \implies (f \circ g) \in \text{surj}(A, C)$ 
  unfolding surj-def
apply (blast intro!: comp-fun comp-fun-apply)
done
```

```
lemma comp-bij:
   $\llbracket g \in \text{bij}(A, B); f \in \text{bij}(B, C) \rrbracket \implies (f \circ g) \in \text{bij}(A, C)$ 
  unfolding bij-def
apply (blast intro: comp-inj comp-surj)
done
```


10.11 Dual Properties of *inj* and *surj*

Useful for proofs from D Pastre. Automatic theorem proving in set theory. Artificial Intelligence, 10:1–27, 1978.

lemma *comp-mem-injD1*:

$\llbracket (f \ O \ g): inj(A, C); \ g \in A \multimap B; \ f \in B \multimap C \rrbracket \implies g \in inj(A, B)$
by (*unfold inj-def, force*)

lemma *comp-mem-injD2*:

$\llbracket (f \ O \ g): inj(A, C); \ g \in surj(A, B); \ f \in B \multimap C \rrbracket \implies f \in inj(B, C)$
apply (*unfold inj-def surj-def, safe*)
apply (*rule-tac x1 = x in bspec [THEN bexE]*)
apply (*erule-tac [3] x1 = w in bspec [THEN bexE], assumption+, safe*)
apply (*rule-tac t = (') (g) in subst-context*)
apply (*erule asm-rl bspec [THEN bspec, THEN mp]+*)
apply (*simp (no-asm-simp)*)
done

lemma *comp-mem-surjD1*:

$\llbracket (f \ O \ g): surj(A, C); \ g \in A \multimap B; \ f \in B \multimap C \rrbracket \implies f \in surj(B, C)$
unfolding *surj-def*
apply (*blast intro!: comp-fun-apply [symmetric] apply-funtype*)
done

lemma *comp-mem-surjD2*:

$\llbracket (f \ O \ g): surj(A, C); \ g \in A \multimap B; \ f \in inj(B, C) \rrbracket \implies g \in surj(A, B)$
apply (*unfold inj-def surj-def, safe*)
apply (*drule-tac x = f'y in bspec, auto*)
apply (*blast intro: apply-funtype*)
done

10.11.1 Inverses of Composition

left inverse of composition; one inclusion is $f \in A \rightarrow B \implies id(A) \subseteq converse(f) \ O \ f$

lemma *left-comp-inverse*: $f \in inj(A, B) \implies converse(f) \ O \ f = id(A)$
apply (*unfold inj-def, clarify*)
apply (*rule equalityI*)
apply (*auto simp add: apply-iff, blast*)
done

right inverse of composition; one inclusion is $f \in A \rightarrow B \implies f \ O \ converse(f) \subseteq id(B)$

lemma *right-comp-inverse*:

$f \in surj(A, B) \implies f \ O \ converse(f) = id(B)$
apply (*simp add: surj-def, clarify*)
apply (*rule equalityI*)

```

apply (best elim: domain-type range-type dest: apply-equality2)
apply (blast intro: apply-Pair)
done

```

10.11.2 Proving that a Function is a Bijection

```

lemma comp-eq-id-iff:
   $\llbracket f \in A \multimap B; g \in B \multimap A \rrbracket \implies f \circ g = id(B) \longleftrightarrow (\forall y \in B. f(g'y) = y)$ 
apply (unfold id-def, safe)
apply (drule-tac t =  $\lambda h. h'y$  in subst-context)
apply simp
apply (rule fun-extension)
apply (blast intro: comp-fun lam-type)
apply auto
done

lemma fg-imp-bijection:
   $\llbracket f \in A \multimap B; g \in B \multimap A; f \circ g = id(B); g \circ f = id(A) \rrbracket \implies f \in bij(A, B)$ 
unfolding bij-def
apply (simp add: comp-eq-id-iff)
apply (blast intro: f-imp-injective f-imp-surjective apply-funtype)
done

```

```

lemma nilpotent-imp-bijection:  $\llbracket f \in A \multimap A; f \circ f = id(A) \rrbracket \implies f \in bij(A, A)$ 
by (blast intro: fg-imp-bijection)

```

```

lemma invertible-imp-bijection:
   $\llbracket converse(f): B \multimap A; f \in A \multimap B \rrbracket \implies f \in bij(A, B)$ 
by (simp add: fg-imp-bijection comp-eq-id-iff
  left-inverse-lemma right-inverse-lemma)

```

10.11.3 Unions of Functions

See similar theorems in func.thy

Theorem by KG, proof by LCP

```

lemma inj-disjoint-Un:
   $\llbracket f \in inj(A, B); g \in inj(C, D); B \cap D = \emptyset \rrbracket$ 
   $\implies (\lambda a \in A \cup C. \text{if } a \in A \text{ then } f'a \text{ else } g'a) \in inj(A \cup C, B \cup D)$ 
apply (rule-tac d =  $\lambda z. \text{if } z \in B \text{ then } converse(f) 'z \text{ else } converse(g) 'z$ 
in lam-injective)
apply (auto simp add: inj-is-fun [THEN apply-type])
done

```

```

lemma surj-disjoint-Un:
   $\llbracket f \in surj(A, B); g \in surj(C, D); A \cap C = \emptyset \rrbracket$ 
   $\implies (f \cup g) \in surj(A \cup C, B \cup D)$ 
apply (simp add: surj-def fun-disjoint-Un)
apply (blast dest!: domain-of-fun)

```

```

      intro!: fun-disjoint-apply1 fun-disjoint-apply2)
done

```

A simple, high-level proof; the version for injections follows from it, using $f \in \text{inj}(A, B) \longleftrightarrow f \in \text{bij}(A, \text{range}(f))$

```

lemma bij-disjoint-Un:
  ⌊f ∈ bij(A,B); g ∈ bij(C,D); A ∩ C = 0; B ∩ D = 0⌋
  ⇒ (f ∪ g) ∈ bij(A ∪ C, B ∪ D)
apply (rule invertible-imp-bijective)
apply (subst converse-Un)
apply (auto intro: fun-disjoint-Un bij-is-fun bij-converse-bij)
done

```

10.11.4 Restrictions as Surjections and Bijections

```

lemma surj-image:
  f ∈ Pi(A,B) ⇒ f ∈ surj(A, f``A)
apply (simp add: surj-def)
apply (blast intro: apply-equality apply-Pair Pi-type)
done

```

```

lemma surj-image-eq: f ∈ surj(A, B) ⇒ f``A = B
by (auto simp add: surj-def image-fun) (blast dest: apply-type)

```

```

lemma restrict-image [simp]: restrict(f,A) `` B = f `` (A ∩ B)
by (auto simp add: restrict-def)

```

```

lemma restrict-inj:
  ⌊f ∈ inj(A,B); C ≤ A⌋ ⇒ restrict(f,C): inj(C,B)
  unfolding inj-def
apply (safe elim!: restrict-type2, auto)
done

```

```

lemma restrict-surj: ⌊f ∈ Pi(A,B); C ≤ A⌋ ⇒ restrict(f,C): surj(C, f``C)
apply (insert restrict-type2 [THEN surj-image])
apply (simp add: restrict-image)
done

```

```

lemma restrict-bij:
  ⌊f ∈ inj(A,B); C ≤ A⌋ ⇒ restrict(f,C): bij(C, f``C)
apply (simp add: inj-def bij-def)
apply (blast intro: restrict-surj surj-is-fun)
done

```

10.11.5 Lemmas for Ramsey's Theorem

```

lemma inj-weaken-type: ⌊f ∈ inj(A,B); B ≤ D⌋ ⇒ f ∈ inj(A,D)
  unfolding inj-def
apply (blast intro: fun-weaken-type)

```

done

lemma *inj-succ-restrict*:

$\llbracket f \in \text{inj}(\text{succ}(m), A) \rrbracket \implies \text{restrict}(f, m) \in \text{inj}(m, A - \{f'm\})$

apply (*rule restrict-bij [THEN bij-is-inj, THEN inj-weaken-type], assumption, blast*)

unfolding *inj-def*

apply (*fast elim: range-type mem-irrefl dest: apply-equality*)

done

lemma *inj-extend*:

$\llbracket f \in \text{inj}(A, B); a \notin A; b \notin B \rrbracket$

$\implies \text{cons}(\langle a, b \rangle, f) \in \text{inj}(\text{cons}(a, A), \text{cons}(b, B))$

unfolding *inj-def*

apply (*force intro: apply-type simp add: fun-extend*)

done

end

11 Relations: Their General Properties and Transitive Closure

theory *Trancl* **imports** *Fixedpt Perm* **begin**

definition

refl $:: [i, i] \Rightarrow o$ **where**

$\text{refl}(A, r) \equiv (\forall x \in A. \langle x, x \rangle \in r)$

definition

irrefl $:: [i, i] \Rightarrow o$ **where**

$\text{irrefl}(A, r) \equiv \forall x \in A. \langle x, x \rangle \notin r$

definition

sym $:: i \Rightarrow o$ **where**

$\text{sym}(r) \equiv \forall x y. \langle x, y \rangle : r \longrightarrow \langle y, x \rangle : r$

definition

asym $:: i \Rightarrow o$ **where**

$\text{asym}(r) \equiv \forall x y. \langle x, y \rangle : r \longrightarrow \neg \langle y, x \rangle : r$

definition

antisym $:: i \Rightarrow o$ **where**

$\text{antisym}(r) \equiv \forall x y. \langle x, y \rangle : r \longrightarrow \langle y, x \rangle : r \longrightarrow x = y$

definition

trans $:: i \Rightarrow o$ **where**

$\text{trans}(r) \equiv \forall x y z. \langle x, y \rangle : r \longrightarrow \langle y, z \rangle : r \longrightarrow \langle x, z \rangle : r$

definition

$trans-on :: [i,i] \Rightarrow o \ (\langle \langle open-block\ notation = \langle mixfix\ trans-on \rangle \rangle trans[-]'(-') \rangle) \text{ where}$
 $trans[A](r) \equiv \forall x \in A. \forall y \in A. \forall z \in A.$
 $\langle x,y \rangle : r \longrightarrow \langle y,z \rangle : r \longrightarrow \langle x,z \rangle : r$

definition

$rtranc :: i \Rightarrow i \ (\langle \langle notation = \langle postfix\ \hat{*} \rangle - \hat{*} \rangle \rangle [100] 100) \text{ where}$
 $r^{\hat{*}} \equiv lfp(field(r) * field(r), \lambda s. id(field(r)) \cup (r \ O \ s))$

definition

$tranc :: i \Rightarrow i \ (\langle \langle notation = \langle postfix\ \hat{+} \rangle - \hat{+} \rangle \rangle [100] 100) \text{ where}$
 $r^{\hat{+}} \equiv r \ O \ r^{\hat{*}}$

definition

$equiv :: [i,i] \Rightarrow o \text{ where}$
 $equiv(A,r) \equiv r \subseteq A * A \wedge refl(A,r) \wedge sym(r) \wedge trans(r)$

11.1 General properties of relations

11.1.1 irreflexivity

lemma *irreflI*:

$\llbracket \bigwedge x. x \in A \implies \langle x,x \rangle \notin r \rrbracket \implies irrefl(A,r)$
by (*simp add: irrefl-def*)

lemma *irreflE*: $\llbracket irrefl(A,r); x \in A \rrbracket \implies \langle x,x \rangle \notin r$
by (*simp add: irrefl-def*)

11.1.2 symmetry

lemma *symI*:

$\llbracket \bigwedge x y. \langle x,y \rangle : r \implies \langle y,x \rangle : r \rrbracket \implies sym(r)$
by (*unfold sym-def, blast*)

lemma *symE*: $\llbracket sym(r); \langle x,y \rangle : r \rrbracket \implies \langle y,x \rangle : r$
by (*unfold sym-def, blast*)

11.1.3 antisymmetry

lemma *antisymI*:

$\llbracket \bigwedge x y. \llbracket \langle x,y \rangle : r; \langle y,x \rangle : r \rrbracket \implies x=y \rrbracket \implies antisym(r)$
by (*simp add: antisym-def, blast*)

lemma *antisymE*: $\llbracket antisym(r); \langle x,y \rangle : r; \langle y,x \rangle : r \rrbracket \implies x=y$
by (*simp add: antisym-def, blast*)

11.1.4 transitivity

lemma *transD*: $\llbracket trans(r); \langle a,b \rangle : r; \langle b,c \rangle : r \rrbracket \implies \langle a,c \rangle : r$
by (*unfold trans-def, blast*)

lemma *trans-onD*:

$\llbracket \text{trans}[A](r); \langle a,b \rangle:r; \langle b,c \rangle:r; a \in A; b \in A; c \in A \rrbracket \implies \langle a,c \rangle:r$
by (*unfold trans-on-def, blast*)

lemma *trans-imp-trans-on*: $\text{trans}(r) \implies \text{trans}[A](r)$

by (*unfold trans-def trans-on-def, blast*)

lemma *trans-on-imp-trans*: $\llbracket \text{trans}[A](r); r \subseteq A * A \rrbracket \implies \text{trans}(r)$

by (*simp add: trans-on-def trans-def, blast*)

11.2 Transitive closure of a relation

lemma *rtrancl-bnd-mono*:

$\text{bnd-mono}(\text{field}(r) * \text{field}(r), \lambda s. \text{id}(\text{field}(r)) \cup (r \circ s))$
by (*rule bnd-monoI, blast+*)

lemma *rtrancl-mono*: $r \leq s \implies r^{\hat{*}} \subseteq s^{\hat{*}}$

unfolding *rtrancl-def*
apply (*rule lfp-mono*)
apply (*rule rtrancl-bnd-mono*) +
apply *blast*
done

lemmas *rtrancl-unfold* =

rtrancl-bnd-mono [*THEN* *rtrancl-def* [*THEN* *def-lfp-unfold*]]

lemmas *rtrancl-type* = *rtrancl-def* [*THEN* *def-lfp-subset*]

lemma *relation-rtrancl*: $\text{relation}(r^{\hat{*}})$

apply (*simp add: relation-def*)
apply (*blast dest: rtrancl-type* [*THEN* *subsetD*])
done

lemma *rtrancl-refl*: $\llbracket a \in \text{field}(r) \rrbracket \implies \langle a,a \rangle \in r^{\hat{*}}$

apply (*rule rtrancl-unfold* [*THEN* *ssubst*])
apply (*erule idI* [*THEN* *UnI1*])
done

lemma *rtrancl-into-rtrancl*: $\llbracket \langle a,b \rangle \in r^{\hat{*}}; \langle b,c \rangle \in r \rrbracket \implies \langle a,c \rangle \in r^{\hat{*}}$

apply (*rule rtrancl-unfold* [*THEN* *ssubst*])
apply (*rule compI* [*THEN* *UnI2*], *assumption*, *assumption*)
done

lemma *r-into-rtrancl*: $\langle a, b \rangle \in r \implies \langle a, b \rangle \in r^*$
by (*rule rtrancl-refl* [*THEN rtrancl-into-rtrancl*], *blast+*)

lemma *r-subset-rtrancl*: $\text{relation}(r) \implies r \subseteq r^*$
by (*simp add: relation-def*, *blast intro: r-into-rtrancl*)

lemma *rtrancl-field*: $\text{field}(r^*) = \text{field}(r)$
by (*blast intro: r-into-rtrancl dest!: rtrancl-type* [*THEN subsetD*])

lemma *rtrancl-full-induct* [*case-names initial step*, *consumes 1*]:

$$\begin{aligned} & \llbracket \langle a, b \rangle \in r^*; \\ & \quad \bigwedge x. x \in \text{field}(r) \implies P(\langle x, x \rangle); \\ & \quad \bigwedge x y z. \llbracket P(\langle x, y \rangle); \langle x, y \rangle \in r^*; \langle y, z \rangle \in r \rrbracket \implies P(\langle x, z \rangle) \rrbracket \\ & \implies P(\langle a, b \rangle) \end{aligned}$$

by (*erule def-induct* [*OF rtrancl-def rtrancl-bnd-mono*], *blast*)

lemma *rtrancl-induct* [*case-names initial step*, *induct set: rtrancl*]:

$$\begin{aligned} & \llbracket \langle a, b \rangle \in r^*; \\ & \quad P(a); \\ & \quad \bigwedge y z. \llbracket \langle a, y \rangle \in r^*; \langle y, z \rangle \in r; P(y) \rrbracket \implies P(z) \rrbracket \\ & \implies P(b) \end{aligned}$$

apply (*subgoal-tac* $\forall y. \langle a, b \rangle = \langle a, y \rangle \longrightarrow P(y)$)

apply (*erule spec* [*THEN mp*], *rule refl*)

apply (*erule rtrancl-full-induct*, *blast+*)
done

lemma *trans-rtrancl*: $\text{trans}(r^*)$
unfolding *trans-def*
apply (*intro allI impI*)
apply (*erule-tac* $b = z$ **in** *rtrancl-induct*, *assumption*)
apply (*blast intro: rtrancl-into-rtrancl*)
done

lemmas *rtrancl-trans* = *trans-rtrancl* [*THEN transD*]

lemma *rtranclE*:

$$\llbracket \langle a, b \rangle \in r^*; (a=b) \rrbracket \implies P;$$

```

       $\bigwedge y. \llbracket \langle a, y \rangle \in r^{\wedge*}; \quad \langle y, b \rangle \in r \rrbracket \implies P$ 
 $\implies P$ 
apply (subgoal-tac  $a = b \mid (\exists y. \langle a, y \rangle \in r^{\wedge*} \wedge \langle y, b \rangle \in r)$  )

apply blast
apply (erule rtrancl-induct, blast+)
done

```

```

lemma trans-trancl:  $\text{trans}(r^{\wedge+})$ 
  unfolding trans-def trancl-def
apply (blast intro: rtrancl-into-rtrancl
      trans-rtrancl [THEN transD, THEN compI])
done

```

```

lemmas trans-on-trancl = trans-trancl [THEN trans-imp-trans-on]

```

```

lemmas trancl-trans = trans-trancl [THEN transD]

```

```

lemma trancl-into-rtrancl:  $\langle a, b \rangle \in r^{\wedge+} \implies \langle a, b \rangle \in r^{\wedge*}$ 
  unfolding trancl-def
apply (blast intro: rtrancl-into-rtrancl)
done

```

```

lemma r-into-trancl:  $\langle a, b \rangle \in r \implies \langle a, b \rangle \in r^{\wedge+}$ 
  unfolding trancl-def
apply (blast intro!: rtrancl-refl)
done

```

```

lemma r-subset-trancl:  $\text{relation}(r) \implies r \subseteq r^{\wedge+}$ 
by (simp add: relation-def, blast intro: r-into-trancl)

```

```

lemma rtrancl-into-trancl1:  $\llbracket \langle a, b \rangle \in r^{\wedge*}; \quad \langle b, c \rangle \in r \rrbracket \implies \langle a, c \rangle \in r^{\wedge+}$ 
by (unfold trancl-def, blast)

```

```

lemma rtrancl-into-trancl2:
   $\llbracket \langle a, b \rangle \in r; \quad \langle b, c \rangle \in r^{\wedge*} \rrbracket \implies \langle a, c \rangle \in r^{\wedge+}$ 
apply (erule rtrancl-induct)
apply (erule r-into-trancl)

```


apply (*blast intro: r-into-trancl trancl-trans*)
done

lemma *trancl-induct* [*case-names initial step, induct set: trancl*]:

$\llbracket \langle a, b \rangle \in r^{\hat{+}};$
 $\bigwedge y. \llbracket \langle a, y \rangle \in r \rrbracket \implies P(y);$
 $\bigwedge y z. \llbracket \langle a, y \rangle \in r^{\hat{+}}; \langle y, z \rangle \in r; P(y) \rrbracket \implies P(z)$
 $\rrbracket \implies P(b)$

apply (*rule compEpair*)

apply (*unfold trancl-def, assumption*)

apply (*subgoal-tac $\forall z. \langle y, z \rangle \in r \longrightarrow P(z)$*)

apply *blast*

apply (*erule rtrancl-induct*)

apply (*blast intro: rtrancl-into-trancl1*)+

done

lemma *tranclE*:

$\llbracket \langle a, b \rangle \in r^{\hat{+}};$
 $\langle a, b \rangle \in r \implies P;$
 $\bigwedge y. \llbracket \langle a, y \rangle \in r^{\hat{+}}; \langle y, b \rangle \in r \rrbracket \implies P$
 $\rrbracket \implies P$

apply (*subgoal-tac $\langle a, b \rangle \in r \mid (\exists y. \langle a, y \rangle \in r^{\hat{+}} \wedge \langle y, b \rangle \in r)$*)

apply *blast*

apply (*rule compEpair*)

apply (*unfold trancl-def, assumption*)

apply (*erule rtranclE*)

apply (*blast intro: rtrancl-into-trancl1*)+

done

lemma *trancl-type*: $r^{\hat{+}} \subseteq \text{field}(r) * \text{field}(r)$

unfolding *trancl-def*

apply (*blast elim: rtrancl-type [THEN subsetD, THEN SigmaE2]*)

done

lemma *relation-trancl*: $\text{relation}(r^{\hat{+}})$

apply (*simp add: relation-def*)

apply (*blast dest: trancl-type [THEN subsetD]*)

done

lemma *trancl-subset-times*: $r \subseteq A * A \implies r^{\hat{+}} \subseteq A * A$

by (*insert trancl-type [of r], blast*)

lemma *trancl-mono*: $r \leq s \implies r^{\hat{+}} \subseteq s^{\hat{+}}$

by (*unfold trancl-def, intro comp-mono rtrancl-mono*)

```

lemma trancl-eq-r:  $\llbracket \text{relation}(r); \text{trans}(r) \rrbracket \implies r^{\wedge+} = r$ 
apply (rule equalityI)
  prefer 2 apply (erule r-subset-trancl, clarify)
apply (frule trancl-type [THEN subsetD], clarify)
apply (erule trancl-induct, assumption)
apply (blast dest: transD)
done

```

```

lemma rtrancl-idemp [simp]:  $(r^{\wedge*})^{\wedge*} = r^{\wedge*}$ 
apply (rule equalityI, auto)
  prefer 2
  apply (frule rtrancl-type [THEN subsetD])
  apply (blast intro: r-into-rtrancl)

```

converse direction

```

apply (frule rtrancl-type [THEN subsetD], clarify)
apply (erule rtrancl-induct)
apply (simp add: rtrancl-refl rtrancl-field)
apply (blast intro: rtrancl-trans)
done

```

```

lemma rtrancl-subset:  $\llbracket R \subseteq S; S \subseteq R^{\wedge*} \rrbracket \implies S^{\wedge*} = R^{\wedge*}$ 
apply (drule rtrancl-mono)
apply (drule rtrancl-mono, simp-all, blast)
done

```

```

lemma rtrancl-Un-rtrancl:
   $\llbracket \text{relation}(r); \text{relation}(s) \rrbracket \implies (r^{\wedge*} \cup s^{\wedge*})^{\wedge*} = (r \cup s)^{\wedge*}$ 
apply (rule rtrancl-subset)
apply (blast dest: r-subset-rtrancl)
apply (blast intro: rtrancl-mono [THEN subsetD])
done

```

```

lemma rtrancl-converseD:  $\langle x, y \rangle : \text{converse}(r)^{\wedge*} \implies \langle x, y \rangle : \text{converse}(r^{\wedge*})$ 
apply (rule converseI)
apply (frule rtrancl-type [THEN subsetD])
apply (erule rtrancl-induct)
apply (blast intro: rtrancl-refl)
apply (blast intro: r-into-rtrancl rtrancl-trans)
done

```

```

lemma rtrancl-converseI:  $\langle x, y \rangle : \text{converse}(r^{\wedge*}) \implies \langle x, y \rangle : \text{converse}(r)^{\wedge*}$ 

```

```

apply (drule converseD)
apply (frule rtrancl-type [THEN subsetD])
apply (erule rtrancl-induct)
apply (blast intro: rtrancl-refl)
apply (blast intro: r-into-rtrancl rtrancl-trans)
done

```

```

lemma rtrancl-converse:  $\text{converse}(r)^{\wedge*} = \text{converse}(r^{\wedge*})$ 
apply (safe intro!: equalityI)
apply (frule rtrancl-type [THEN subsetD])
apply (safe dest!: rtrancl-converseD intro!: rtrancl-converseI)
done

```

```

lemma trancl-converseD:  $\langle a, b \rangle : \text{converse}(r)^{\wedge+} \implies \langle a, b \rangle : \text{converse}(r^{\wedge+})$ 
apply (erule trancl-induct)
apply (auto intro: r-into-trancl trancl-trans)
done

```

```

lemma trancl-converseI:  $\langle x, y \rangle : \text{converse}(r^{\wedge+}) \implies \langle x, y \rangle : \text{converse}(r)^{\wedge+}$ 
apply (drule converseD)
apply (erule trancl-induct)
apply (auto intro: r-into-trancl trancl-trans)
done

```

```

lemma trancl-converse:  $\text{converse}(r)^{\wedge+} = \text{converse}(r^{\wedge+})$ 
apply (safe intro!: equalityI)
apply (frule trancl-type [THEN subsetD])
apply (safe dest!: trancl-converseD intro!: trancl-converseI)
done

```

```

lemma converse-trancl-induct [case-names initial step, consumes 1]:

$$\begin{aligned} & \llbracket \langle a, b \rangle : r^{\wedge+}; \bigwedge y. \langle y, b \rangle : r \implies P(y); \\ & \quad \bigwedge y z. \llbracket \langle y, z \rangle \in r; \langle z, b \rangle \in r^{\wedge+}; P(z) \rrbracket \implies P(y) \rrbracket \\ & \implies P(a) \end{aligned}$$

apply (drule converseI)
apply (simp (no-asm-use) add: trancl-converse [symmetric])
apply (erule trancl-induct)
apply (auto simp add: trancl-converse)
done

```

end

12 Well-Founded Recursion

theory *WF* **imports** *Trancl* **begin**

definition

$wf \quad :: i \Rightarrow o \text{ where}$

$$wf(r) \equiv \forall Z. Z=0 \mid (\exists x \in Z. \forall y. \langle y, x \rangle : r \longrightarrow \neg y \in Z)$$

definition

$wf\text{-}on :: [i, i] \Rightarrow o \quad (\langle \langle \text{open-block notation} = \langle \text{mixfix } wf\text{-}on \rangle \rangle wf[-]'(-)) \rangle \text{ where}$

$$wf\text{-}on(A, r) \equiv wf(r \cap A * A)$$

definition

$is\text{-}recfun :: [i, i, [i, i] \Rightarrow i, i] \Rightarrow o \text{ where}$

$$is\text{-}recfun(r, a, H, f) \equiv (f = (\lambda x \in r - \{\{a\}\}. H(x, restrict(f, r - \{\{x\}\}))))$$

definition

$the\text{-}recfun :: [i, i, [i, i] \Rightarrow i] \Rightarrow i \text{ where}$

$$the\text{-}recfun(r, a, H) \equiv (THE f. is\text{-}recfun(r, a, H, f))$$

definition

$wftrec :: [i, i, [i, i] \Rightarrow i] \Rightarrow i \text{ where}$

$$wftrec(r, a, H) \equiv H(a, the\text{-}recfun(r, a, H))$$

definition

$wfrec :: [i, i, [i, i] \Rightarrow i] \Rightarrow i \text{ where}$

$$wfrec(r, a, H) \equiv wftrec(r^{\wedge}+, a, \lambda x f. H(x, restrict(f, r - \{\{x\}\})))$$

definition

$wfrec\text{-}on :: [i, i, i, [i, i] \Rightarrow i] \Rightarrow i \quad (\langle \langle \text{open-block notation} = \langle \text{mixfix } wfrec\text{-}on \rangle \rangle wfrec[-]'(-, -, -)) \rangle$

where $wfrec[A](r, a, H) \equiv wfrec(r \cap A * A, a, H)$

12.1 Well-Founded Relations

12.1.1 Equivalences between wf and $wf\text{-}on$

lemma $wf\text{-}imp\text{-}wf\text{-}on$: $wf(r) \implies wf[A](r)$

by ($unfold\ wf\text{-}def\ wf\text{-}on\text{-}def, force$)

lemma $wf\text{-}on\text{-}imp\text{-}wf$: $\llbracket wf[A](r); r \subseteq A * A \rrbracket \implies wf(r)$

by ($simp\ add: wf\text{-}on\text{-}def\ subset\text{-}Int\text{-}iff$)

lemma $wf\text{-}on\text{-}field\text{-}imp\text{-}wf$: $wf[field(r)](r) \implies wf(r)$

by ($unfold\ wf\text{-}def\ wf\text{-}on\text{-}def, fast$)

lemma $wf\text{-}iff\text{-}wf\text{-}on\text{-}field$: $wf(r) \longleftrightarrow wf[field(r)](r)$

by ($blast\ intro: wf\text{-}imp\text{-}wf\text{-}on\ wf\text{-}on\text{-}field\text{-}imp\text{-}wf$)

lemma $wf\text{-}on\text{-}subset\text{-}A$: $\llbracket wf[A](r); B \leq A \rrbracket \implies wf[B](r)$

by ($unfold\ wf\text{-}on\text{-}def\ wf\text{-}def, fast$)

lemma $wf\text{-}on\text{-}subset\text{-}r$: $\llbracket wf[A](r); s \leq r \rrbracket \implies wf[A](s)$

by (*unfold wf-on-def wf-def, fast*)

lemma *wf-subset*: $\llbracket wf(s); r \leq s \rrbracket \implies wf(r)$
by (*simp add: wf-def, fast*)

12.1.2 Introduction Rules for *wf-on*

If every non-empty subset of A has an r -minimal element then we have $wf[A](r)$.

lemma *wf-onI*:
assumes *prem*: $\bigwedge Z u. \llbracket Z \leq A; u \in Z; \forall x \in Z. \exists y \in Z. \langle y, x \rangle : r \rrbracket \implies False$
shows $wf[A](r)$
unfolding *wf-on-def wf-def*
apply (*rule equalsOI [THEN disjCI, THEN allI]*)
apply (*rule-tac Z = Z in prem, blast+*)
done

If r allows well-founded induction over A then we have $wf[A](r)$. Premise is equivalent to $\bigwedge B. \forall x \in A. (\forall y. \langle y, x \rangle \in r \longrightarrow y \in B) \longrightarrow x \in B \implies A \subseteq B$

lemma *wf-onI2*:
assumes *prem*: $\bigwedge y B. \llbracket \forall x \in A. (\forall y \in A. \langle y, x \rangle : r \longrightarrow y \in B) \longrightarrow x \in B; y \in A \rrbracket \implies y \in B$
shows $wf[A](r)$
apply (*rule wf-onI*)
apply (*rule-tac c=u in prem [THEN DiffE]*)
prefer 3 apply blast
apply fast+
done

12.1.3 Well-founded Induction

Consider the least z in $domain(r)$ such that $P(z)$ does not hold...

lemma *wf-induct-raw*:
 $\llbracket wf(r); \bigwedge x. \llbracket \forall y. \langle y, x \rangle : r \longrightarrow P(y) \rrbracket \implies P(x) \rrbracket \implies P(a)$
unfolding *wf-def*
apply (*erule-tac x = {z ∈ domain(r). ¬ P(z)} in allE*)
apply blast
done

lemmas *wf-induct* = *wf-induct-raw* [*rule-format, consumes 1, case-names step, induct set: wf*]

The form of this rule is designed to match *wfI*

lemma *wf-induct2*:

$\llbracket wf(r); a \in A; field(r) \leq A; \bigwedge x. \llbracket x \in A; \forall y. \langle y, x \rangle: r \longrightarrow P(y) \rrbracket \Longrightarrow P(x) \rrbracket$
 $\Longrightarrow P(a)$
apply (*erule-tac* $P=a \in A$ **in** *rev-mp*)
apply (*erule-tac* $a=a$ **in** *wf-induct, blast*)
done

lemma *field-Int-square*: $field(r \cap A * A) \subseteq A$
by *blast*

lemma *wf-on-induct-raw* [*consumes 2, induct set: wf-on*]:
 $\llbracket wf[A](r); a \in A; \bigwedge x. \llbracket x \in A; \forall y \in A. \langle y, x \rangle: r \longrightarrow P(y) \rrbracket \Longrightarrow P(x) \rrbracket$
 $\Longrightarrow P(a)$
unfolding *wf-on-def*
apply (*erule wf-induct2, assumption*)
apply (*rule field-Int-square, blast*)
done

lemma *wf-on-induct* [*consumes 2, case-names step, induct set: wf-on*]:
 $wf[A](r) \Longrightarrow a \in A \Longrightarrow (\bigwedge x. x \in A \Longrightarrow (\bigwedge y. y \in A \Longrightarrow \langle y, x \rangle \in r \Longrightarrow P(y)))$
 $\Longrightarrow P(x) \Longrightarrow P(a)$
using *wf-on-induct-raw* [*of A r a P*] **by** *simp*

If r allows well-founded induction then we have $wf(r)$.

lemma *wfI*:
 $\llbracket field(r) \leq A; \bigwedge y B. \llbracket \forall x \in A. (\forall y \in A. \langle y, x \rangle: r \longrightarrow y \in B) \longrightarrow x \in B; y \in A \rrbracket \Longrightarrow y \in B \rrbracket$
 $\Longrightarrow wf(r)$
apply (*rule wf-on-subset-A* [*THEN wf-on-field-imp-wf*])
apply (*rule wf-onI2*)
prefer 2 apply blast
apply blast
done

12.2 Basic Properties of Well-Founded Relations

lemma *wf-not-refl*: $wf(r) \Longrightarrow \langle a, a \rangle \notin r$
by (*erule-tac* $a=a$ **in** *wf-induct, blast*)

lemma *wf-not-sym* [*rule-format*]: $wf(r) \Longrightarrow \forall x. \langle a, x \rangle: r \longrightarrow \langle x, a \rangle \notin r$
by (*erule-tac* $a=a$ **in** *wf-induct, blast*)

lemmas $wf\text{-}asym = wf\text{-}not\text{-}sym$ [*THEN swap*]

lemma *wf-on-not-refl*: $\llbracket wf[A](r); a \in A \rrbracket \Longrightarrow \langle a, a \rangle \notin r$
by (*erule-tac* $a=a$ **in** *wf-on-induct, assumption, blast*)

lemma *wf-on-not-sym*:

$\llbracket wf[A](r); a \in A \rrbracket \implies (\bigwedge b. b \in A \implies \langle a, b \rangle : r \implies \langle b, a \rangle \notin r)$
apply (*atomize (full), intro impI*)
apply (*erule-tac a=a in wf-on-induct, assumption, blast*)
done

lemma *wf-on-asy*:

$\llbracket wf[A](r); \neg Z \implies \langle a, b \rangle \in r; \langle b, a \rangle \notin r \implies Z; \neg Z \implies a \in A; \neg Z \implies b \in A \rrbracket \implies Z$
by (*blast dest: wf-on-not-sym*)

lemma *wf-on-chain3*:

$\llbracket wf[A](r); \langle a, b \rangle : r; \langle b, c \rangle : r; \langle c, a \rangle : r; a \in A; b \in A; c \in A \rrbracket \implies P$
apply (*subgoal-tac $\forall y \in A. \forall z \in A. \langle a, y \rangle : r \longrightarrow \langle y, z \rangle : r \longrightarrow \langle z, a \rangle : r \longrightarrow P$, blast*)
apply (*erule-tac a=a in wf-on-induct, assumption, blast*)
done

transitive closure of a WF relation is WF provided A is downward closed

lemma *wf-on-trancl*:

$\llbracket wf[A](r); r - "A \subseteq A \rrbracket \implies wf[A](r^{\wedge+})$
apply (*rule wf-onI2*)
apply (*frule bspec [THEN mp], assumption+*)
apply (*erule-tac a = y in wf-on-induct, assumption*)
apply (*blast elim: tranclE, blast*)
done

lemma *wf-trancl*: $wf(r) \implies wf(r^{\wedge+})$

apply (*simp add: wf-iff-wf-on-field*)
apply (*rule wf-on-subset-A*)
apply (*erule wf-on-trancl*)
apply *blast*
apply (*rule trancl-type [THEN field-rel-subset]*)
done

$r - " \{a\}$ is the set of everything under a in r

lemmas *underI* = *vimage-singleton-iff* [*THEN iffD2*]

lemmas *underD* = *vimage-singleton-iff* [*THEN iffD1*]

12.3 The Predicate *is-recfun*

lemma *is-recfun-type*: $is-recfun(r, a, H, f) \implies f \in r - " \{a\} -> range(f)$

unfolding *is-recfun-def*
apply (*erule ssubst*)
apply (*rule lamI [THEN rangeI, THEN lam-type], assumption*)
done

lemmas *is-recfun-imp-function* = *is-recfun-type* [THEN *fun-is-function*]

lemma *apply-recfun*:

$\llbracket \text{is-recfun}(r, a, H, f); \langle x, a \rangle : r \rrbracket \implies f'x = H(x, \text{restrict}(f, r - \{\{x\}\}))$

unfolding *is-recfun-def*

replace f only on the left-hand side

apply (*erule-tac* $P = \lambda x. t(x) = u$ **for** $t\ u$ **in** *ssubst*)

apply (*simp add: underI*)

done

lemma *is-recfun-equal* [*rule-format*]:

$\llbracket wf(r); \text{trans}(r); \text{is-recfun}(r, a, H, f); \text{is-recfun}(r, b, H, g) \rrbracket$

$\implies \langle x, a \rangle : r \longrightarrow \langle x, b \rangle : r \longrightarrow f'x = g'x$

apply (*frule-tac* $f = f$ **in** *is-recfun-type*)

apply (*frule-tac* $f = g$ **in** *is-recfun-type*)

apply (*simp add: is-recfun-def*)

apply (*erule-tac* $a = x$ **in** *wf-induct*)

apply (*intro impI*)

apply (*elim ssubst*)

apply (*simp (no-asm-simp) add: vimage-singleton-iff restrict-def*)

apply (*rule-tac* $t = \lambda z. H(x, z)$ **for** x **in** *subst-context*)

apply (*subgoal-tac* $\forall y \in r - \{\{x\}. \forall z. \langle y, z \rangle : f \longleftrightarrow \langle y, z \rangle : g$)

apply (*blast dest: transD*)

apply (*simp add: apply-iff*)

apply (*blast dest: transD intro: sym*)

done

lemma *is-recfun-cut*:

$\llbracket wf(r); \text{trans}(r);$

$\text{is-recfun}(r, a, H, f); \text{is-recfun}(r, b, H, g); \langle b, a \rangle : r \rrbracket$

$\implies \text{restrict}(f, r - \{\{b\}\}) = g$

apply (*frule-tac* $f = f$ **in** *is-recfun-type*)

apply (*rule fun-extension*)

apply (*blast dest: transD intro: restrict-type2*)

apply (*erule is-recfun-type, simp*)

apply (*blast dest: transD intro: is-recfun-equal*)

done

12.4 Recursion: Main Existence Lemma

lemma *is-recfun-functional*:

$\llbracket wf(r); \text{trans}(r); \text{is-recfun}(r, a, H, f); \text{is-recfun}(r, a, H, g) \rrbracket \implies f = g$

by (*blast intro: fun-extension is-recfun-type is-recfun-equal*)

lemma *the-recfun-eq*:

$\llbracket \text{is-recfun}(r, a, H, f); wf(r); \text{trans}(r) \rrbracket \implies \text{the-recfun}(r, a, H) = f$

unfolding *the-recfun-def*

apply (*blast intro: is-recfun-functional*)

done

lemma *is-the-recfun*:

$\llbracket is-recfun(r, a, H, f); wf(r); trans(r) \rrbracket$
 $\implies is-recfun(r, a, H, the-recfun(r, a, H))$

by (*simp add: the-recfun-eq*)

lemma *unfold-the-recfun*:

$\llbracket wf(r); trans(r) \rrbracket \implies is-recfun(r, a, H, the-recfun(r, a, H))$

apply (*rule-tac a=a in wf-induct, assumption*)

apply (*rename-tac a1*)

apply (*rule-tac f = $\lambda y \in r - \{a1\}. wftrec(r, y, H)$ in is-the-recfun*)

apply *typecheck*

unfolding *is-recfun-def wftrec-def*

— Applying the substitution: must keep the quantified assumption!

apply (*rule lam-cong [OF refl]*)

apply (*drule underD*)

apply (*fold is-recfun-def*)

apply (*rule-tac t = $\lambda z. H(x, z)$ for x in subst-context*)

apply (*rule fun-extension*)

apply (*blast intro: is-recfun-type*)

apply (*rule lam-type [THEN restrict-type2]*)

apply *blast*

apply (*blast dest: transD*)

apply *atomize*

apply (*frule spec [THEN mp], assumption*)

apply (*subgoal-tac $\langle xa, a1 \rangle \in r$*)

apply (*drule-tac x1 = xa in spec [THEN mp], assumption*)

apply (*simp add: vimage-singleton-iff*
apply-recfun is-recfun-cut)

apply (*blast dest: transD*)

done

12.5 Unfolding $wftrec(r, a, H)$

lemma *the-recfun-cut*:

$\llbracket wf(r); trans(r); \langle b, a \rangle : r \rrbracket$
 $\implies restrict(the-recfun(r, a, H), r - \{b\}) = the-recfun(r, b, H)$

by (*blast intro: is-recfun-cut unfold-the-recfun*)

lemma *wftrec*:

$\llbracket wf(r); trans(r) \rrbracket \implies$
 $wftrec(r, a, H) = H(a, \lambda x \in r - \{a\}. wftrec(r, x, H))$

unfolding *wftrec-def*

apply (*subst unfold-the-recfun [unfolded is-recfun-def]*)

apply (*simp-all add: vimage-singleton-iff [THEN iff-sym] the-recfun-cut*)

done

12.5.1 Removal of the Premise $trans(r)$

lemma *wfrec*:

$wf(r) \implies wfrec(r, a, H) = H(a, \lambda x \in r - \{\{a\}. wfrec(r, x, H))$

unfolding *wfrec-def*

apply (*erule wf-trancl* [*THEN wftrec*, *THEN ssubst*])

apply (*rule trans-trancl*)

apply (*rule vimage-pair-mono* [*THEN restrict-lam-eq*, *THEN subst-context*])

apply (*erule r-into-trancl*)

apply (*rule subset-refl*)

done

lemma *def-wfrec*:

$\llbracket \bigwedge x. h(x) \equiv wfrec(r, x, H); wf(r) \rrbracket \implies$

$h(a) = H(a, \lambda x \in r - \{\{a\}. h(x))$

apply *simp*

apply (*elim wfrec*)

done

lemma *wfrec-type*:

$\llbracket wf(r); a \in A; field(r) \leq A;$

$\bigwedge x u. \llbracket x \in A; u \in Pi(r - \{\{x\}, B) \rrbracket \implies H(x, u) \in B(x)$

$\rrbracket \implies wfrec(r, a, H) \in B(a)$

apply (*rule-tac a = a in wf-induct2, assumption+*)

apply (*subst wfrec, assumption*)

apply (*simp add: lam-type underD*)

done

lemma *wfrec-on*:

$\llbracket wf[A](r); a \in A \rrbracket \implies$

$wfrec[A](r, a, H) = H(a, \lambda x \in (r - \{\{a\}\} \cap A. wfrec[A](r, x, H))$

unfolding *wf-on-def wfrec-on-def*

apply (*erule wfrec* [*THEN trans*])

apply (*simp add: vimage-Int-square*)

done

Minimal-element characterization of well-foundedness

lemma *wf-eq-minimal*: $wf(r) \longleftrightarrow (\forall Q. x \in Q \longrightarrow (\exists z \in Q. \forall y. \langle y, z \rangle : r \longrightarrow y \notin Q))$

unfolding *wf-def* **by** *blast*

end

13 Transitive Sets and Ordinals

theory *Ordinal* **imports** *WF Bool equalities begin*

definition

$Memrel \quad :: i \Rightarrow i \text{ where}$
 $Memrel(A) \equiv \{ z \in A * A . \exists x y. z = \langle x, y \rangle \wedge x \in y \}$

definition

$Transset \quad :: i \Rightarrow o \text{ where}$
 $Transset(i) \equiv \forall x \in i. x <= i$

definition

$Ord \quad :: i \Rightarrow o \text{ where}$
 $Ord(i) \equiv Transset(i) \wedge (\forall x \in i. Transset(x))$

definition

$lt \quad :: [i, i] \Rightarrow o \text{ (infixl } \langle < \rangle 50) \text{ where}$
 $i < j \quad \equiv i \in j \wedge Ord(j)$

definition

$Limit \quad :: i \Rightarrow o \text{ where}$
 $Limit(i) \equiv Ord(i) \wedge 0 < i \wedge (\forall y. y < i \longrightarrow succ(y) < i)$

abbreviation

$le \text{ (infixl } \langle \leq \rangle 50) \text{ where}$
 $x \leq y \equiv x < succ(y)$

13.1 Rules for Transset**13.1.1 Three Neat Characterisations of Transset**

lemma *Transset-iff-Pow*: $Transset(A) <-> A <= Pow(A)$
by (*unfold Transset-def, blast*)

lemma *Transset-iff-Union-succ*: $Transset(A) <-> \bigcup (succ(A)) = A$
unfolding *Transset-def*
apply (*blast elim!: equalityE*)
done

lemma *Transset-iff-Union-subset*: $Transset(A) <-> \bigcup (A) \subseteq A$
by (*unfold Transset-def, blast*)

13.1.2 Consequences of Downwards Closure

lemma *Transset-doubleton-D*:
 $\llbracket Transset(C); \{a, b\} : C \rrbracket \Longrightarrow a \in C \wedge b \in C$
by (*unfold Transset-def, blast*)

lemma *Transset-Pair-D*:
 $\llbracket Transset(C); \langle a, b \rangle \in C \rrbracket \Longrightarrow a \in C \wedge b \in C$
apply (*simp add: Pair-def*)
apply (*blast dest: Transset-doubleton-D*)
done

lemma *Transset-includes-domain*:

$\llbracket \text{Transset}(C); A*B \subseteq C; b \in B \rrbracket \implies A \subseteq C$
by (*blast dest: Transset-Pair-D*)

lemma *Transset-includes-range*:

$\llbracket \text{Transset}(C); A*B \subseteq C; a \in A \rrbracket \implies B \subseteq C$
by (*blast dest: Transset-Pair-D*)

13.1.3 Closure Properties

lemma *Transset-0*: $\text{Transset}(0)$

by (*unfold Transset-def, blast*)

lemma *Transset-Un*:

$\llbracket \text{Transset}(i); \text{Transset}(j) \rrbracket \implies \text{Transset}(i \cup j)$
by (*unfold Transset-def, blast*)

lemma *Transset-Int*:

$\llbracket \text{Transset}(i); \text{Transset}(j) \rrbracket \implies \text{Transset}(i \cap j)$
by (*unfold Transset-def, blast*)

lemma *Transset-succ*: $\text{Transset}(i) \implies \text{Transset}(\text{succ}(i))$

by (*unfold Transset-def, blast*)

lemma *Transset-Pow*: $\text{Transset}(i) \implies \text{Transset}(\text{Pow}(i))$

by (*unfold Transset-def, blast*)

lemma *Transset-Union*: $\text{Transset}(A) \implies \text{Transset}(\bigcup(A))$

by (*unfold Transset-def, blast*)

lemma *Transset-Union-family*:

$\llbracket \bigwedge i. i \in A \implies \text{Transset}(i) \rrbracket \implies \text{Transset}(\bigcup(A))$
by (*unfold Transset-def, blast*)

lemma *Transset-Inter-family*:

$\llbracket \bigwedge i. i \in A \implies \text{Transset}(i) \rrbracket \implies \text{Transset}(\bigcap(A))$
by (*unfold Inter-def Transset-def, blast*)

lemma *Transset-UN*:

$(\bigwedge x. x \in A \implies \text{Transset}(B(x))) \implies \text{Transset}(\bigcup_{x \in A} B(x))$
by (*rule Transset-Union-family, auto*)

lemma *Transset-INT*:

$(\bigwedge x. x \in A \implies \text{Transset}(B(x))) \implies \text{Transset}(\bigcap_{x \in A} B(x))$
by (*rule Transset-Inter-family, auto*)

13.2 Lemmas for Ordinals

lemma *OrdI*:

by (*simp add: Ord-def*)
 $\llbracket \text{Transset}(i); \bigwedge x. x \in i \implies \text{Transset}(x) \rrbracket \implies \text{Ord}(i)$

lemma *Ord-is-Transset*: $\text{Ord}(i) \implies \text{Transset}(i)$
by (*simp add: Ord-def*)

lemma *Ord-contains-Transset*:
 $\llbracket \text{Ord}(i); j \in i \rrbracket \implies \text{Transset}(j)$
by (*unfold Ord-def, blast*)

lemma *Ord-in-Ord*: $\llbracket \text{Ord}(i); j \in i \rrbracket \implies \text{Ord}(j)$
by (*unfold Ord-def Transset-def, blast*)

lemma *Ord-in-Ord'*: $\llbracket j \in i; \text{Ord}(i) \rrbracket \implies \text{Ord}(j)$
by (*blast intro: Ord-in-Ord*)

lemmas *Ord-succD* = *Ord-in-Ord* [*OF - succI1*]

lemma *Ord-subset-Ord*: $\llbracket \text{Ord}(i); \text{Transset}(j); j \leq i \rrbracket \implies \text{Ord}(j)$
by (*simp add: Ord-def Transset-def, blast*)

lemma *OrdmemD*: $\llbracket j \in i; \text{Ord}(i) \rrbracket \implies j \leq i$
by (*unfold Ord-def Transset-def, blast*)

lemma *Ord-trans*: $\llbracket i \in j; j \in k; \text{Ord}(k) \rrbracket \implies i \in k$
by (*blast dest: OrdmemD*)

lemma *Ord-succ-subsetI*: $\llbracket i \in j; \text{Ord}(j) \rrbracket \implies \text{succ}(i) \subseteq j$
by (*blast dest: OrdmemD*)

13.3 The Construction of Ordinals: 0, succ, Union

lemma *Ord-0* [*iff, TC*]: $\text{Ord}(0)$
by (*blast intro: OrdI Transset-0*)

lemma *Ord-succ* [*TC*]: $\text{Ord}(i) \implies \text{Ord}(\text{succ}(i))$
by (*blast intro: OrdI Transset-succ Ord-is-Transset Ord-contains-Transset*)

lemmas *Ord-1* = *Ord-0* [*THEN Ord-succ*]

lemma *Ord-succ-iff* [*iff*]: $\text{Ord}(\text{succ}(i)) \iff \text{Ord}(i)$
by (*blast intro: Ord-succ dest!: Ord-succD*)

lemma *Ord-Un* [*intro, simp, TC*]: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \text{Ord}(i \cup j)$
unfolding *Ord-def*
apply (*blast intro!: Transset-Un*)

done

lemma *Ord-Int* [TC]: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \text{Ord}(i \cap j)$
unfolding *Ord-def*
apply (*blast intro!*: *Transset-Int*)
done

There is no set of all ordinals, for then it would contain itself

lemma *ON-class*: $\neg (\forall i. i \in X \iff \text{Ord}(i))$
proof (*rule notI*)
assume $X: \forall i. i \in X \iff \text{Ord}(i)$
have $\forall x y. x \in X \longrightarrow y \in x \longrightarrow y \in X$
by (*simp add: X, blast intro: Ord-in-Ord*)
hence *Transset*(X)
by (*auto simp add: Transset-def*)
moreover have $\bigwedge x. x \in X \implies \text{Transset}(x)$
by (*simp add: X Ord-def*)
ultimately have *Ord*(X) **by** (*rule OrdI*)
hence $X \in X$ **by** (*simp add: X*)
thus False **by** (*rule mem-irrefl*)
qed

13.4 < is 'less Than' for Ordinals

lemma *ltI*: $\llbracket i \in j; \text{Ord}(j) \rrbracket \implies i < j$
by (*unfold lt-def, blast*)

lemma *ltE*:
 $\llbracket i < j; \llbracket i \in j; \text{Ord}(i); \text{Ord}(j) \rrbracket \implies P \rrbracket \implies P$
unfolding *lt-def*
apply (*blast intro: Ord-in-Ord*)
done

lemma *ltD*: $i < j \implies i \in j$
by (*erule ltE, assumption*)

lemma *not-lt0* [*simp*]: $\neg i < 0$
by (*unfold lt-def, blast*)

lemma *lt-Ord*: $j < i \implies \text{Ord}(j)$
by (*erule ltE, assumption*)

lemma *lt-Ord2*: $j < i \implies \text{Ord}(i)$
by (*erule ltE, assumption*)

lemmas *le-Ord2* = *lt-Ord2* [*THEN Ord-succD*]

lemmas $lt0E = not\text{-}lt0$ [*THEN* $notE$, $elim!$]

lemma $lt\text{-}trans$ [*trans*]: $\llbracket i < j; j < k \rrbracket \implies i < k$
by ($blast$ $intro!$: ltI $elim!$: ltE $intro$: $Ord\text{-}trans$)

lemma $lt\text{-}not\text{-}sym$: $i < j \implies \neg (j < i)$
unfolding $lt\text{-}def$
apply ($blast$ $elim$: $mem\text{-}asym$)
done

lemmas $lt\text{-}asym = lt\text{-}not\text{-}sym$ [*THEN* $swap$]

lemma $lt\text{-}irrefl$ [$elim!$]: $i < i \implies P$
by ($blast$ $intro$: $lt\text{-}asym$)

lemma $lt\text{-}not\text{-}refl$: $\neg i < i$
apply ($rule$ $notI$)
apply ($erule$ $lt\text{-}irrefl$)
done

Recall that $i \leq j$ abbreviates $i \leq j!$

lemma $le\text{-}iff$: $i \leq j \iff i < j \mid (i = j \wedge Ord(j))$
by ($unfold$ $lt\text{-}def$, $blast$)

lemma leI : $i < j \implies i \leq j$
by ($simp$ add : $le\text{-}iff$)

lemma $le\text{-}eqI$: $\llbracket i = j; Ord(j) \rrbracket \implies i \leq j$
by ($simp$ add : $le\text{-}iff$)

lemmas $le\text{-}refl = refl$ [*THEN* $le\text{-}eqI$]

lemma $le\text{-}refl\text{-}iff$ [iff]: $i \leq i \iff Ord(i)$
by ($simp$ ($no\text{-}asm\text{-}simp$) add : $lt\text{-}not\text{-}refl$ $le\text{-}iff$)

lemma $leCI$: $(\neg (i = j \wedge Ord(j)) \implies i < j) \implies i \leq j$
by ($simp$ add : $le\text{-}iff$, $blast$)

lemma leE :
 $\llbracket i \leq j; i < j \implies P; \llbracket i = j; Ord(j) \rrbracket \implies P \rrbracket \implies P$
by ($simp$ add : $le\text{-}iff$, $blast$)

lemma $le\text{-}anti\text{-}sym$: $\llbracket i \leq j; j \leq i \rrbracket \implies i = j$
apply ($simp$ add : $le\text{-}iff$)
apply ($blast$ $elim$: $lt\text{-}asym$)
done

lemma *le0-iff* [*simp*]: $i \leq 0 \iff i = 0$
by (*blast elim!*: *leE*)

lemmas $le0D = le0-iff$ [*THEN iffD1*, *dest!*]

13.5 Natural Deduction Rules for Memrel

lemma *Memrel-iff* [*simp*]: $\langle a, b \rangle \in \text{Memrel}(A) \iff a \in b \wedge a \in A \wedge b \in A$
by (*unfold Memrel-def*, *blast*)

lemma *MemrelI* [*intro!*]: $\llbracket a \in b; a \in A; b \in A \rrbracket \implies \langle a, b \rangle \in \text{Memrel}(A)$
by *auto*

lemma *MemrelE* [*elim!*]:
 $\llbracket \langle a, b \rangle \in \text{Memrel}(A);$
 $\llbracket a \in A; b \in A; a \in b \rrbracket \implies P$
 $\implies P$
by *auto*

lemma *Memrel-type*: $\text{Memrel}(A) \subseteq A * A$
by (*unfold Memrel-def*, *blast*)

lemma *Memrel-mono*: $A \leq B \implies \text{Memrel}(A) \subseteq \text{Memrel}(B)$
by (*unfold Memrel-def*, *blast*)

lemma *Memrel-0* [*simp*]: $\text{Memrel}(0) = 0$
by (*unfold Memrel-def*, *blast*)

lemma *Memrel-1* [*simp*]: $\text{Memrel}(1) = 0$
by (*unfold Memrel-def*, *blast*)

lemma *relation-Memrel*: $\text{relation}(\text{Memrel}(A))$
by (*simp add: relation-def Memrel-def*)

lemma *wf-Memrel*: $\text{wf}(\text{Memrel}(A))$
unfolding *wf-def*
apply (*rule foundation [THEN disjE, THEN allI]*, *erule disjI1*, *blast*)
done

The premise $\text{Ord}(i)$ does not suffice.

lemma *trans-Memrel*:
 $\text{Ord}(i) \implies \text{trans}(\text{Memrel}(i))$
by (*unfold Ord-def Transset-def trans-def*, *blast*)

However, the following premise is strong enough.

lemma *Transset-trans-Memrel*:
 $\forall j \in i. \text{Transset}(j) \implies \text{trans}(\text{Memrel}(i))$
by (*unfold Transset-def trans-def*, *blast*)

lemma *Transset-Memrel-iff*:
 $\text{Transset}(A) \implies \langle a, b \rangle \in \text{Memrel}(A) \iff a \in b \wedge b \in A$
by (*unfold Transset-def, blast*)

13.6 Transfinite Induction

lemma *Transset-induct*:

$$\begin{aligned} & \llbracket i \in k; \text{Transset}(k); \\ & \quad \bigwedge x. \llbracket x \in k; \forall y \in x. P(y) \rrbracket \implies P(x) \rrbracket \\ & \implies P(i) \end{aligned}$$

apply (*simp add: Transset-def*)
apply (*erule wf-Memrel [THEN wf-induct2], blast+*)
done

lemma *Ord-induct* [*consumes 2*]:
 $i \in k \implies \text{Ord}(k) \implies (\bigwedge x. x \in k \implies (\bigwedge y. y \in x \implies P(y)) \implies P(x)) \implies P(i)$
using *Transset-induct* [*OF - Ord-is-Transset, of i k P*] **by** *simp*

lemma *trans-induct* [*consumes 1, case-names step*]:
 $\text{Ord}(i) \implies (\bigwedge x. \text{Ord}(x) \implies (\bigwedge y. y \in x \implies P(y)) \implies P(x)) \implies P(i)$
apply (*rule Ord-succ [THEN succI1 [THEN Ord-induct]], assumption*)
apply (*blast intro: Ord-succ [THEN Ord-in-Ord]*)
done

14 Fundamental properties of the epsilon ordering ($<$ on ordinals)

14.0.1 Proving That $<$ is a Linear Ordering on the Ordinals

lemma *Ord-linear*:
 $\text{Ord}(i) \implies \text{Ord}(j) \implies i \in j \mid i = j \mid j \in i$
proof (*induct i arbitrary: j rule: trans-induct*)
case (*step i*)
note *step-i = step*
show *?case using* $\langle \text{Ord}(j) \rangle$
proof (*induct j rule: trans-induct*)
case (*step j*)
thus *?case using step-i*
by (*blast dest: Ord-trans*)
qed
qed

The trichotomy law for ordinals

lemma *Ord-linear-lt*:
assumes *o: Ord(i) Ord(j)*

```

  obtains (lt)  $i < j$  | (eq)  $i = j$  | (gt)  $j < i$ 
apply (simp add: lt-def)
apply (rule-tac  $i1 = i$  and  $j1 = j$  in Ord-linear [THEN disjE])
apply (blast intro: o)+
done

```

```

lemma Ord-linear2:
  assumes o: Ord(i) Ord(j)
  obtains (lt)  $i < j$  | (ge)  $j \leq i$ 
apply (rule-tac  $i = i$  and  $j = j$  in Ord-linear-lt)
apply (blast intro: leI le-eqI sym o) +
done

```

```

lemma Ord-linear-le:
  assumes o: Ord(i) Ord(j)
  obtains (le)  $i \leq j$  | (ge)  $j \leq i$ 
apply (rule-tac  $i = i$  and  $j = j$  in Ord-linear-lt)
apply (blast intro: leI le-eqI o) +
done

```

```

lemma le-imp-not-lt:  $j \leq i \implies \neg i < j$ 
by (blast elim!: leE elim: lt-asym)

```

```

lemma not-lt-imp-le:  $\llbracket \neg i < j; \text{Ord}(i); \text{Ord}(j) \rrbracket \implies j \leq i$ 
by (rule-tac  $i = i$  and  $j = j$  in Ord-linear2, auto)

```

14.0.2 Some Rewrite Rules for $<$, \leq

```

lemma Ord-mem-iff-lt:  $\text{Ord}(j) \implies i \in j \iff i < j$ 
by (unfold lt-def, blast)

```

```

lemma not-lt-iff-le:  $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \neg i < j \iff j \leq i$ 
by (blast dest: le-imp-not-lt not-lt-imp-le)

```

```

lemma not-le-iff-lt:  $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \neg i \leq j \iff j < i$ 
by (simp (no-asm-simp) add: not-lt-iff-le [THEN iff-sym])

```

```

lemma Ord-0-le:  $\text{Ord}(i) \implies 0 \leq i$ 
by (erule not-lt-iff-le [THEN iffD1], auto)

```

```

lemma Ord-0-lt:  $\llbracket \text{Ord}(i); i \neq 0 \rrbracket \implies 0 < i$ 
apply (erule not-le-iff-lt [THEN iffD1])
apply (rule Ord-0, blast)
done

```

```

lemma Ord-0-lt-iff:  $\text{Ord}(i) \implies i \neq 0 \iff 0 < i$ 
by (blast intro: Ord-0-lt)

```

14.1 Results about Less-Than or Equals

lemma *zero-le-succ-iff* [iff]: $0 \leq \text{succ}(x) \leftrightarrow \text{Ord}(x)$
by (*blast intro: Ord-0-le elim: ltE*)

lemma *subset-imp-le*: $\llbracket j <= i; \text{Ord}(i); \text{Ord}(j) \rrbracket \implies j \leq i$
apply (*rule not-lt-iff-le [THEN iffD1], assumption+*)
apply (*blast elim: ltE mem-irrefl*)
done

lemma *le-imp-subset*: $i \leq j \implies i <= j$
by (*blast dest: OrdmemD elim: ltE leE*)

lemma *le-subset-iff*: $j \leq i \leftrightarrow j <= i \wedge \text{Ord}(i) \wedge \text{Ord}(j)$
by (*blast dest: subset-imp-le le-imp-subset elim: ltE*)

lemma *le-succ-iff*: $i \leq \text{succ}(j) \leftrightarrow i \leq j \mid i = \text{succ}(j) \wedge \text{Ord}(i)$
apply (*simp (no-asm) add: le-iff*)
apply *blast*
done

lemma *all-lt-imp-le*: $\llbracket \text{Ord}(i); \text{Ord}(j); \bigwedge x. x < j \implies x < i \rrbracket \implies j \leq i$
by (*blast intro: not-lt-imp-le dest: lt-irrefl*)

14.1.1 Transitivity Laws

lemma *lt-trans1*: $\llbracket i \leq j; j < k \rrbracket \implies i < k$
by (*blast elim!: leE intro: lt-trans*)

lemma *lt-trans2*: $\llbracket i < j; j \leq k \rrbracket \implies i < k$
by (*blast elim!: leE intro: lt-trans*)

lemma *le-trans*: $\llbracket i \leq j; j \leq k \rrbracket \implies i \leq k$
by (*blast intro: lt-trans1*)

lemma *succ-leI*: $i < j \implies \text{succ}(i) \leq j$
apply (*rule not-lt-iff-le [THEN iffD1]*)
apply (*blast elim: ltE leE lt-asm*)
done

lemma *succ-leE*: $\text{succ}(i) \leq j \implies i < j$
apply (*rule not-le-iff-lt [THEN iffD1]*)
apply (*blast elim: ltE leE lt-asm*)
done

lemma *succ-le-iff* [iff]: $\text{succ}(i) \leq j \leftrightarrow i < j$
by (*blast intro: succ-leI succ-leE*)

lemma *succ-le-imp-le*: $\text{succ}(i) \leq \text{succ}(j) \implies i \leq j$
by (*blast dest!*: *succ-leE*)

lemma *lt-subset-trans*: $\llbracket i \subseteq j; j < k; \text{Ord}(i) \rrbracket \implies i < k$
apply (*rule subset-imp-le* [*THEN lt-trans1*])
apply (*blast intro*: *elim*: *ltE*) +
done

lemma *lt-imp-0-lt*: $j < i \implies 0 < i$
by (*blast intro*: *lt-trans1 Ord-0-le* [*OF lt-Ord*])

lemma *succ-lt-iff*: $\text{succ}(i) < j \iff i < j \wedge \text{succ}(i) \neq j$
apply *auto*
apply (*blast intro*: *lt-trans le-refl dest*: *lt-Ord*)
apply (*frule lt-Ord*)
apply (*rule not-le-iff-lt* [*THEN iffD1*])
apply (*blast intro*: *lt-Ord2*)
apply *blast*
apply (*simp add*: *lt-Ord lt-Ord2 le-iff*)
apply (*blast dest*: *lt-asym*)
done

lemma *Ord-succ-mem-iff*: $\text{Ord}(j) \implies \text{succ}(i) \in \text{succ}(j) \iff i \in j$
apply (*insert succ-le-iff* [*of i j*])
apply (*simp add*: *lt-def*)
done

14.1.2 Union and Intersection

lemma *Un-upper1-le*: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies i \leq i \cup j$
by (*rule Un-upper1* [*THEN subset-imp-le*], *auto*)

lemma *Un-upper2-le*: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies j \leq i \cup j$
by (*rule Un-upper2* [*THEN subset-imp-le*], *auto*)

lemma *Un-least-lt*: $\llbracket i < k; j < k \rrbracket \implies i \cup j < k$
apply (*rule-tac i = i and j = j in Ord-linear-le*)
apply (*auto simp add*: *Un-commute le-subset-iff subset-Un-iff lt-Ord*)
done

lemma *Un-least-lt-iff*: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies i \cup j < k \iff i < k \wedge j < k$
apply (*safe intro!*: *Un-least-lt*)
apply (*rule-tac* [2] *Un-upper2-le* [*THEN lt-trans1*])
apply (*rule Un-upper1-le* [*THEN lt-trans1*], *auto*)
done

lemma *Un-least-mem-iff*:
 $\llbracket \text{Ord}(i); \text{Ord}(j); \text{Ord}(k) \rrbracket \implies i \cup j \in k \iff i \in k \wedge j \in k$

apply (*insert Un-least-lt-iff* [*of i j k*])
apply (*simp add: lt-def*)
done

lemma *Int-greatest-lt*: $\llbracket i < k; j < k \rrbracket \implies i \cap j < k$
apply (*rule-tac i = i and j = j in Ord-linear-le*)
apply (*auto simp add: Int-commute le-subset-iff subset-Int-iff lt-Ord*)
done

lemma *Ord-Un-if*:
 $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies i \cup j = (\text{if } j < i \text{ then } i \text{ else } j)$
by (*simp add: not-lt-iff-le le-imp-subset leI*
subset-Un-iff [*symmetric*] *subset-Un-iff2* [*symmetric*])

lemma *succ-Un-distrib*:
 $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \text{succ}(i \cup j) = \text{succ}(i) \cup \text{succ}(j)$
by (*simp add: Ord-Un-if lt-Ord le-Ord2*)

lemma *lt-Un-iff*:
 $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies k < i \cup j \iff k < i \mid k < j$
apply (*simp add: Ord-Un-if not-lt-iff-le*)
apply (*blast intro: leI lt-trans2*)
done

lemma *le-Un-iff*:
 $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies k \leq i \cup j \iff k \leq i \mid k \leq j$
by (*simp add: succ-Un-distrib lt-Un-iff* [*symmetric*])

lemma *Un-upper1-lt*: $\llbracket k < i; \text{Ord}(j) \rrbracket \implies k < i \cup j$
by (*simp add: lt-Un-iff lt-Ord2*)

lemma *Un-upper2-lt*: $\llbracket k < j; \text{Ord}(i) \rrbracket \implies k < i \cup j$
by (*simp add: lt-Un-iff lt-Ord2*)

lemma *Ord-Union-succ-eq*: $\text{Ord}(i) \implies \bigcup (\text{succ}(i)) = i$
by (*blast intro: Ord-trans*)

14.2 Results about Limits

lemma *Ord-Union* [*intro, simp, TC*]: $\llbracket \bigwedge i. i \in A \implies \text{Ord}(i) \rrbracket \implies \text{Ord}(\bigcup (A))$
apply (*rule Ord-is-Transset* [*THEN Transset-Union-family, THEN OrdI*])
apply (*blast intro: Ord-contains-Transset*)
done

lemma *Ord-UN* [*intro, simp, TC*]:
 $\llbracket \bigwedge x. x \in A \implies \text{Ord}(B(x)) \rrbracket \implies \text{Ord}(\bigcup_{x \in A} B(x))$
by (*rule Ord-Union, blast*)

lemma *Ord-Inter* [*intro,simp,TC*]:
 $\llbracket \bigwedge i. i \in A \implies \text{Ord}(i) \rrbracket \implies \text{Ord}(\bigcap (A))$
apply (*rule Transset-Inter-family* [*THEN OrdI*])
apply (*blast intro: Ord-is-Transset*)
apply (*simp add: Inter-def*)
apply (*blast intro: Ord-contains-Transset*)
done

lemma *Ord-INT* [*intro,simp,TC*]:
 $\llbracket \bigwedge x. x \in A \implies \text{Ord}(B(x)) \rrbracket \implies \text{Ord}(\bigcap_{x \in A} B(x))$
by (*rule Ord-Inter, blast*)

lemma *UN-least-le*:
 $\llbracket \text{Ord}(i); \bigwedge x. x \in A \implies b(x) \leq i \rrbracket \implies (\bigcup_{x \in A} b(x)) \leq i$
apply (*rule le-imp-subset* [*THEN UN-least, THEN subset-imp-le*])
apply (*blast intro: Ord-UN elim: ltE*)
done

lemma *UN-succ-least-lt*:
 $\llbracket j < i; \bigwedge x. x \in A \implies b(x) < j \rrbracket \implies (\bigcup_{x \in A} \text{succ}(b(x))) < i$
apply (*rule ltE, assumption*)
apply (*rule UN-least-le* [*THEN lt-trans2*])
apply (*blast intro: succ-leI*)
done

lemma *UN-upper-lt*:
 $\llbracket a \in A; i < b(a); \text{Ord}(\bigcup_{x \in A} b(x)) \rrbracket \implies i < (\bigcup_{x \in A} b(x))$
by (*unfold lt-def, blast*)

lemma *UN-upper-le*:
 $\llbracket a \in A; i \leq b(a); \text{Ord}(\bigcup_{x \in A} b(x)) \rrbracket \implies i \leq (\bigcup_{x \in A} b(x))$
apply (*frule ltD*)
apply (*rule le-imp-subset* [*THEN subset-trans, THEN subset-imp-le*])
apply (*blast intro: lt-Ord UN-upper*)
done

lemma *lt-Union-iff*: $\forall i \in A. \text{Ord}(i) \implies (j < \bigcup (A)) \iff (\exists i \in A. j < i)$
by (*auto simp: lt-def Ord-Union*)

lemma *Union-upper-le*:
 $\llbracket j \in J; i \leq j; \text{Ord}(\bigcup (J)) \rrbracket \implies i \leq \bigcup J$
apply (*subst Union-eq-UN*)
apply (*rule UN-upper-le, auto*)
done

lemma *le-implies-UN-le-UN*:

$\llbracket \bigwedge x. x \in A \implies c(x) \leq d(x) \rrbracket \implies (\bigcup x \in A. c(x)) \leq (\bigcup x \in A. d(x))$
apply (*rule UN-least-le*)
apply (*rule-tac [2] UN-upper-le*)
apply (*blast intro: Ord-UN le-Ord2*) +
done

lemma *Ord-equality*: $\text{Ord}(i) \implies (\bigcup y \in i. \text{succ}(y)) = i$
by (*blast intro: Ord-trans*)

lemma *Ord-Union-subset*: $\text{Ord}(i) \implies \bigcup(i) \subseteq i$
by (*blast intro: Ord-trans*)

14.3 Limit Ordinals – General Properties

lemma *Limit-Union-eq*: $\text{Limit}(i) \implies \bigcup(i) = i$
unfolding *Limit-def*
apply (*fast intro!: ltI elim!: ltE elim: Ord-trans*)
done

lemma *Limit-is-Ord*: $\text{Limit}(i) \implies \text{Ord}(i)$
unfolding *Limit-def*
apply (*erule conjunct1*)
done

lemma *Limit-has-0*: $\text{Limit}(i) \implies 0 < i$
unfolding *Limit-def*
apply (*erule conjunct2 [THEN conjunct1]*)
done

lemma *Limit-nonzero*: $\text{Limit}(i) \implies i \neq 0$
by (*drule Limit-has-0, blast*)

lemma *Limit-has-succ*: $\llbracket \text{Limit}(i); j < i \rrbracket \implies \text{succ}(j) < i$
by (*unfold Limit-def, blast*)

lemma *Limit-succ-lt-iff* [*simp*]: $\text{Limit}(i) \implies \text{succ}(j) < i \iff (j < i)$
apply (*safe intro!: Limit-has-succ*)
apply (*frule lt-Ord*)
apply (*blast intro: lt-trans*)
done

lemma *zero-not-Limit* [*iff*]: $\neg \text{Limit}(0)$
by (*simp add: Limit-def*)

lemma *Limit-has-1*: $\text{Limit}(i) \implies 1 < i$
by (*blast intro: Limit-has-0 Limit-has-succ*)

lemma *increasing-LimitI*: $\llbracket 0 < l; \forall x \in l. \exists y \in l. x < y \rrbracket \implies \text{Limit}(l)$

```

apply (unfold Limit-def, simp add: lt-Ord2, clarify)
apply (drule-tac i=y in ltD)
apply (blast intro: lt-trans1 [OF - ltI] lt-Ord2)
done

```

```

lemma non-succ-LimitI:
  assumes i: 0 < i and nsucc:  $\bigwedge y. \text{succ}(y) \neq i$ 
  shows Limit(i)
proof -
  have Oi: Ord(i) using i by (simp add: lt-def)
  { fix y
    assume yi: y < i
    hence Osy: Ord(succ(y)) by (simp add: lt-Ord Ord-succ)
    have  $\neg i \leq y$  using yi by (blast dest: le-imp-not-lt)
    hence succ(y) < i using nsucc [of y]
    by (blast intro: Ord-linear-lt [OF Osy Oi]) }
  thus ?thesis using i Oi by (auto simp add: Limit-def)
qed

```

```

lemma succ-LimitE [elim!]: Limit(succ(i))  $\implies$  P
apply (rule lt-irrefl)
apply (rule Limit-has-succ, assumption)
apply (erule Limit-is-Ord [THEN Ord-succD, THEN le-refl])
done

```

```

lemma not-succ-Limit [simp]:  $\neg \text{Limit}(\text{succ}(i))$ 
by blast

```

```

lemma Limit-le-succD:  $\llbracket \text{Limit}(i); i \leq \text{succ}(j) \rrbracket \implies i \leq j$ 
by (blast elim!: leE)

```

14.3.1 Traditional 3-Way Case Analysis on Ordinals

```

lemma Ord-cases-disj: Ord(i)  $\implies i=0 \mid (\exists j. \text{Ord}(j) \wedge i=\text{succ}(j)) \mid \text{Limit}(i)$ 
by (blast intro!: non-succ-LimitI Ord-0-lt)

```

```

lemma Ord-cases:
  assumes i: Ord(i)
  obtains  $(0) \ i=0 \mid (\text{succ}) \ j \text{ where } \text{Ord}(j) \ i=\text{succ}(j) \mid (\text{limit}) \ \text{Limit}(i)$ 
by (insert Ord-cases-disj [OF i], auto)

```

```

lemma trans-induct3-raw:
   $\llbracket \text{Ord}(i);$ 
     $P(0);$ 
     $\bigwedge x. \llbracket \text{Ord}(x); P(x) \rrbracket \implies P(\text{succ}(x));$ 
     $\bigwedge x. \llbracket \text{Limit}(x); \forall y \in x. P(y) \rrbracket \implies P(x)$ 
   $\rrbracket \implies P(i)$ 
apply (erule trans-induct)
apply (erule Ord-cases, blast+)

```


done

lemma *trans-induct3* [*case-names 0 succ limit, consumes 1*]:

$Ord(i) \implies P(0) \implies (\bigwedge x. Ord(x) \implies P(x) \implies P(succ(x))) \implies (\bigwedge x. Limit(x) \implies (\bigwedge y. y \in x \implies P(y)) \implies P(x)) \implies P(i)$
using *trans-induct3-raw* [*of i P*] **by** *simp*

A set of ordinals is either empty, contains its own union, or its union is a limit ordinal.

lemma *Union-le*: $\llbracket \bigwedge x. x \in I \implies x \leq j; Ord(j) \rrbracket \implies \bigcup(I) \leq j$
by (*auto simp add: le-subset-iff Union-least*)

lemma *Ord-set-cases*:

assumes *I*: $\forall i \in I. Ord(i)$
shows $I = 0 \vee \bigcup(I) \in I \vee (\bigcup(I) \notin I \wedge Limit(\bigcup(I)))$

proof (*cases* $\bigcup(I)$ *rule: Ord-cases*)

show $Ord(\bigcup I)$ **using** *I* **by** (*blast intro: Ord-Union*)

next

assume $\bigcup I = 0$ **thus** *?thesis* **by** (*simp, blast intro: subst-elem*)

next

fix *j*

assume *j*: $Ord(j)$ **and** $UIj: \bigcup(I) = succ(j)$

{ **assume** $\forall i \in I. i \leq j$

hence $\bigcup(I) \leq j$

by (*simp add: Union-le j*)

hence *False*

by (*simp add: UIj lt-not-refl*) }

then obtain *i* **where** $i \in I$ $succ(j) \leq i$ **using** *I j*

by (*atomize, auto simp add: not-le-iff-lt*)

have $\bigcup(I) \leq succ(j)$ **using** *UIj j* **by** *auto*

hence $i \leq succ(j)$ **using** *i*

by (*simp add: le-subset-iff Union-subset-iff*)

hence $succ(j) = i$ **using** *i*

by (*blast intro: le-anti-sym*)

hence $succ(j) \in I$ **by** (*simp add: i*)

thus *?thesis* **by** (*simp add: UIj*)

next

assume $Limit(\bigcup I)$ **thus** *?thesis* **by** *auto*

qed

If the union of a set of ordinals is a successor, then it is an element of that set.

lemma *Ord-Union-eq-succD*: $\llbracket \forall x \in X. Ord(x); \bigcup X = succ(j) \rrbracket \implies succ(j) \in X$
by (*drule Ord-set-cases, auto*)

lemma *Limit-Union* [*rule-format*]: $\llbracket I \neq 0; (\bigwedge i. i \in I \implies Limit(i)) \rrbracket \implies Limit(\bigcup I)$

apply (*simp add: Limit-def lt-def*)

apply (*blast intro!: equalityI*)

done

end

15 Special quantifiers

theory *OrdQuant* imports *Ordinal* begin

15.1 Quantifiers and union operator for ordinals

definition

$oall :: [i, i \Rightarrow o] \Rightarrow o$ **where**
 $oall(A, P) \equiv \forall x. x < A \longrightarrow P(x)$

definition

$oex :: [i, i \Rightarrow o] \Rightarrow o$ **where**
 $oex(A, P) \equiv \exists x. x < A \wedge P(x)$

definition

$OUnion :: [i, i \Rightarrow i] \Rightarrow i$ **where**
 $OUnion(i, B) \equiv \{z: \bigcup_{x \in i}. B(x). Ord(i)\}$

syntax

-oall :: [idt, i, o] $\Rightarrow o$ ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \forall \langle \rangle \rangle \forall \text{ } \langle \text{ } \rangle \text{ } \rangle 10$)
-oex :: [idt, i, o] $\Rightarrow o$ ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \exists \langle \rangle \rangle \exists \text{ } \langle \text{ } \rangle \text{ } \rangle 10$)
-OUNION :: [idt, i, i] $\Rightarrow i$ ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \bigcup \langle \rangle \rangle \bigcup \text{ } \langle \text{ } \rangle \text{ } \rangle 10$)

syntax-consts

-oall $\Rightarrow oall$ **and**
-oex $\Rightarrow oex$ **and**
-OUNION $\Rightarrow OUnion$

translations

$\forall x < a. P \Rightarrow CONST\ oall(a, \lambda x. P)$
 $\exists x < a. P \Rightarrow CONST\ oex(a, \lambda x. P)$
 $\bigcup x < a. B \Rightarrow CONST\ OUnion(a, \lambda x. B)$

15.1.1 simplification of the new quantifiers

lemma [simp]: $(\forall x < 0. P(x))$
by (simp add: oall-def)

lemma [simp]: $\neg(\exists x < 0. P(x))$
by (simp add: oex-def)

lemma [simp]: $(\forall x < succ(i). P(x)) \longleftrightarrow (Ord(i) \longrightarrow P(i) \wedge (\forall x < i. P(x)))$
apply (simp add: oall-def le-iff)
apply (blast intro: lt-Ord2)
done

lemma [simp]: $(\exists x < \text{succ}(i). P(x)) <-> (\text{Ord}(i) \wedge (P(i) \mid (\exists x < i. P(x))))$
apply (simp add: oex-def le-iff)
apply (blast intro: lt-Ord2)
done

15.1.2 Union over ordinals

lemma Ord-OUN [intro,simp]:
 $\llbracket \bigwedge x. x < A \implies \text{Ord}(B(x)) \rrbracket \implies \text{Ord}(\bigcup x < A. B(x))$
by (simp add: OUnion-def ltI Ord-UN)

lemma OUN-upper-lt:
 $\llbracket a < A; i < b(a); \text{Ord}(\bigcup x < A. b(x)) \rrbracket \implies i < (\bigcup x < A. b(x))$
by (unfold OUnion-def lt-def, blast)

lemma OUN-upper-le:
 $\llbracket a < A; i \leq b(a); \text{Ord}(\bigcup x < A. b(x)) \rrbracket \implies i \leq (\bigcup x < A. b(x))$
apply (unfold OUnion-def, auto)
apply (rule UN-upper-le)
apply (auto simp add: lt-def)
done

lemma Limit-OUN-eq: $\text{Limit}(i) \implies (\bigcup x < i. x) = i$
by (simp add: OUnion-def Limit-Union-eq Limit-is-Ord)

lemma OUN-least:
 $(\bigwedge x. x < A \implies B(x) \subseteq C) \implies (\bigcup x < A. B(x)) \subseteq C$
by (simp add: OUnion-def UN-least ltI)

lemma OUN-least-le:
 $\llbracket \text{Ord}(i); \bigwedge x. x < A \implies b(x) \leq i \rrbracket \implies (\bigcup x < A. b(x)) \leq i$
by (simp add: OUnion-def UN-least-le ltI Ord-0-le)

lemma le-implies-OUN-le-OUN:
 $\llbracket \bigwedge x. x < A \implies c(x) \leq d(x) \rrbracket \implies (\bigcup x < A. c(x)) \leq (\bigcup x < A. d(x))$
by (blast intro: OUN-least-le OUN-upper-le le-Ord2 Ord-OUN)

lemma OUN-UN-eq:
 $(\bigwedge x. x \in A \implies \text{Ord}(B(x)))$
 $\implies (\bigcup z < (\bigcup x \in A. B(x)). C(z)) = (\bigcup x \in A. \bigcup z < B(x). C(z))$
by (simp add: OUnion-def)

lemma OUN-Union-eq:
 $(\bigwedge x. x \in X \implies \text{Ord}(x))$
 $\implies (\bigcup z < \bigcup (X). C(z)) = (\bigcup x \in X. \bigcup z < x. C(z))$
by (simp add: OUnion-def)

lemma *atomize-oall* [*symmetric, rulify*]:

$$(\bigwedge x. x < A \implies P(x)) \equiv \text{Trueprop } (\forall x < A. P(x))$$

by (*simp add: oall-def atomize-all atomize-imp*)

15.1.3 universal quantifier for ordinals

lemma *oallI* [*intro!*]:

$$\llbracket \bigwedge x. x < A \implies P(x) \rrbracket \implies \forall x < A. P(x)$$

by (*simp add: oall-def*)

lemma *ospec*: $\llbracket \forall x < A. P(x); x < A \rrbracket \implies P(x)$
by (*simp add: oall-def*)

lemma *oallE*:

$$\llbracket \forall x < A. P(x); P(x) \implies Q; \neg x < A \implies Q \rrbracket \implies Q$$

by (*simp add: oall-def, blast*)

lemma *rev-oallE* [*elim*]:

$$\llbracket \forall x < A. P(x); \neg x < A \implies Q; P(x) \implies Q \rrbracket \implies Q$$

by (*simp add: oall-def, blast*)

lemma *oall-simp* [*simp*]: $(\forall x < a. \text{True}) <-> \text{True}$
by *blast*

lemma *oall-cong* [*cong*]:

$$\llbracket a = a'; \bigwedge x. x < a' \implies P(x) <-> P'(x) \rrbracket$$

$$\implies \text{oall}(a, \lambda x. P(x)) <-> \text{oall}(a', \lambda x. P'(x))$$

by (*simp add: oall-def*)

15.1.4 existential quantifier for ordinals

lemma *oexI* [*intro*]:

$$\llbracket P(x); x < A \rrbracket \implies \exists x < A. P(x)$$

apply (*simp add: oex-def, blast*)
done

lemma *oexCI*:

$$\llbracket \forall x < A. \neg P(x) \implies P(a); a < A \rrbracket \implies \exists x < A. P(x)$$

apply (*simp add: oex-def, blast*)
done

lemma *oexE* [*elim!*]:

$$\llbracket \exists x < A. P(x); \bigwedge x. \llbracket x < A; P(x) \rrbracket \implies Q \rrbracket \implies Q$$

apply (*simp add: oex-def, blast*)
done

lemma *oex-cong* [*cong*]:

$$\llbracket a=a'; \bigwedge x. x < a' \implies P(x) <-> P'(x) \rrbracket$$

$$\implies \text{oex}(a, \lambda x. P(x)) <-> \text{oex}(a', \lambda x. P'(x))$$

apply (*simp add: oex-def cong add: conj-cong*)
done

15.1.5 Rules for Ordinal-Indexed Unions

lemma *OUN-I* [*intro*]: $\llbracket a < i; b \in B(a) \rrbracket \implies b: (\bigcup z < i. B(z))$
by (*unfold OUnion-def lt-def, blast*)

lemma *OUN-E* [*elim!*]:

$$\llbracket b \in (\bigcup z < i. B(z)); \bigwedge a. \llbracket b \in B(a); a < i \rrbracket \implies R \rrbracket \implies R$$

apply (*unfold OUnion-def lt-def, blast*)
done

lemma *OUN-iff*: $b \in (\bigcup x < i. B(x)) <-> (\exists x < i. b \in B(x))$
by (*unfold OUnion-def oex-def lt-def, blast*)

lemma *OUN-cong* [*cong*]:

$$\llbracket i=j; \bigwedge x. x < j \implies C(x)=D(x) \rrbracket \implies (\bigcup x < i. C(x)) = (\bigcup x < j. D(x))$$

by (*simp add: OUnion-def lt-def OUN-iff*)

lemma *lt-induct*:

$$\llbracket i < k; \bigwedge x. \llbracket x < k; \forall y < x. P(y) \rrbracket \implies P(x) \rrbracket \implies P(i)$$

apply (*simp add: lt-def oall-def*)
apply (*erule conjE*)
apply (*erule Ord-induct, assumption, blast*)
done

15.2 Quantification over a class

definition

rall :: $[i \Rightarrow o, i \Rightarrow o] \Rightarrow o$ **where**
 $rall(M, P) \equiv \forall x. M(x) \longrightarrow P(x)$

definition

rex :: $[i \Rightarrow o, i \Rightarrow o] \Rightarrow o$ **where**
 $rex(M, P) \equiv \exists x. M(x) \wedge P(x)$

syntax

-rall :: $[pttrn, i \Rightarrow o, o] \Rightarrow o$ ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \forall \rangle \rangle \forall \langle \cdot \rangle \cdot \rangle$)
10)

-rex :: $[pttrn, i \Rightarrow o, o] \Rightarrow o$ ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \exists \rangle \rangle \exists \langle \cdot \rangle \cdot \rangle$)
10)

syntax-consts

-rall \equiv *rall* **and**

-rex \equiv *rex*

translations

$\forall x[M]. P \equiv \text{CONST } \text{rall}(M, \lambda x. P)$
 $\exists x[M]. P \equiv \text{CONST } \text{rex}(M, \lambda x. P)$

15.2.1 Relativized universal quantifier

lemma *rallI* [*intro!*]: $\llbracket \bigwedge x. M(x) \implies P(x) \rrbracket \implies \forall x[M]. P(x)$
by (*simp add: rall-def*)

lemma *rspec*: $\llbracket \forall x[M]. P(x); M(x) \rrbracket \implies P(x)$
by (*simp add: rall-def*)

lemma *rev-rallE* [*elim*]:
 $\llbracket \forall x[M]. P(x); \neg M(x) \rrbracket \implies Q; P(x) \implies Q \rrbracket \implies Q$
by (*simp add: rall-def, blast*)

lemma *rallE*: $\llbracket \forall x[M]. P(x); P(x) \implies Q; \neg M(x) \implies Q \rrbracket \implies Q$
by *blast*

lemma *rall-triv* [*simp*]: $(\forall x[M]. P) \longleftrightarrow ((\exists x. M(x)) \longrightarrow P)$
by (*simp add: rall-def*)

lemma *rall-cong* [*cong*]:
 $(\bigwedge x. M(x) \implies P(x) <-> P'(x)) \implies (\forall x[M]. P(x)) <-> (\forall x[M]. P'(x))$
by (*simp add: rall-def*)

15.2.2 Relativized existential quantifier

lemma *rexI* [*intro*]: $\llbracket P(x); M(x) \rrbracket \implies \exists x[M]. P(x)$
by (*simp add: rex-def, blast*)

lemma *rev-rexI*: $\llbracket M(x); P(x) \rrbracket \implies \exists x[M]. P(x)$
by *blast*

lemma *rexCI*: $\llbracket \forall x[M]. \neg P(x) \implies P(a); M(a) \rrbracket \implies \exists x[M]. P(x)$
by *blast*

lemma *rexE* [*elim!*]: $\llbracket \exists x[M]. P(x); \bigwedge x. \llbracket M(x); P(x) \rrbracket \implies Q \rrbracket \implies Q$
by (*simp add: rex-def, blast*)

lemma *rex-triv* [*simp*]: $(\exists x[M]. P) \longleftrightarrow ((\exists x. M(x)) \wedge P)$
by (*simp add: rex-def*)

lemma *rex-cong* [*cong*]:
 $(\bigwedge x. M(x) \implies P(x) <-> P'(x)) \implies (\exists x[M]. P(x)) <-> (\exists x[M]. P'(x))$

by (*simp add: rex-def cong: conj-cong*)

lemma *rall-is-ball* [*simp*]: $(\forall x[\lambda z. z \in A]. P(x)) <-> (\forall x \in A. P(x))$
by *blast*

lemma *rex-is-bex* [*simp*]: $(\exists x[\lambda z. z \in A]. P(x)) <-> (\exists x \in A. P(x))$
by *blast*

lemma *atomize-rall*: $(\bigwedge x. M(x) \implies P(x)) \equiv \text{Trueprop } (\forall x[M]. P(x))$
by (*simp add: rall-def atomize-all atomize-imp*)

declare *atomize-rall* [*symmetric, rulify*]

lemma *rall-simps1*:

$(\forall x[M]. P(x) \wedge Q) <-> (\forall x[M]. P(x)) \wedge ((\forall x[M]. \text{False}) \mid Q)$
 $(\forall x[M]. P(x) \mid Q) <-> ((\forall x[M]. P(x)) \mid Q)$
 $(\forall x[M]. P(x) \longrightarrow Q) <-> ((\exists x[M]. P(x)) \longrightarrow Q)$
 $(\neg(\forall x[M]. P(x))) <-> (\exists x[M]. \neg P(x))$

by *blast+*

lemma *rall-simps2*:

$(\forall x[M]. P \wedge Q(x)) <-> ((\forall x[M]. \text{False}) \mid P) \wedge (\forall x[M]. Q(x))$
 $(\forall x[M]. P \mid Q(x)) <-> (P \mid (\forall x[M]. Q(x)))$
 $(\forall x[M]. P \longrightarrow Q(x)) <-> (P \longrightarrow (\forall x[M]. Q(x)))$

by *blast+*

lemmas *rall-simps* [*simp*] = *rall-simps1 rall-simps2*

lemma *rall-conj-distrib*:

$(\forall x[M]. P(x) \wedge Q(x)) <-> ((\forall x[M]. P(x)) \wedge (\forall x[M]. Q(x)))$

by *blast*

lemma *rex-simps1*:

$(\exists x[M]. P(x) \wedge Q) <-> ((\exists x[M]. P(x)) \wedge Q)$
 $(\exists x[M]. P(x) \mid Q) <-> (\exists x[M]. P(x)) \mid ((\exists x[M]. \text{True}) \wedge Q)$
 $(\exists x[M]. P(x) \longrightarrow Q) <-> ((\forall x[M]. P(x)) \longrightarrow ((\exists x[M]. \text{True}) \wedge Q))$
 $(\neg(\exists x[M]. P(x))) <-> (\forall x[M]. \neg P(x))$

by *blast+*

lemma *rex-simps2*:

$(\exists x[M]. P \wedge Q(x)) <-> (P \wedge (\exists x[M]. Q(x)))$
 $(\exists x[M]. P \mid Q(x)) <-> ((\exists x[M]. \text{True}) \wedge P) \mid (\exists x[M]. Q(x))$
 $(\exists x[M]. P \longrightarrow Q(x)) <-> (((\forall x[M]. \text{False}) \mid P) \longrightarrow (\exists x[M]. Q(x)))$

by *blast+*

lemmas *rex-simps* [*simp*] = *rex-simps1 rex-simps2*

lemma *rex-disj-distrib*:

$(\exists x[M]. P(x) \mid Q(x)) <-> ((\exists x[M]. P(x)) \mid (\exists x[M]. Q(x)))$

by *blast*

15.2.3 One-point rule for bounded quantifiers

lemma *rex-triv-one-point1* [simp]: $(\exists x[M]. x=a) \leftrightarrow (M(a))$
by *blast*

lemma *rex-triv-one-point2* [simp]: $(\exists x[M]. a=x) \leftrightarrow (M(a))$
by *blast*

lemma *rex-one-point1* [simp]: $(\exists x[M]. x=a \wedge P(x)) \leftrightarrow (M(a) \wedge P(a))$
by *blast*

lemma *rex-one-point2* [simp]: $(\exists x[M]. a=x \wedge P(x)) \leftrightarrow (M(a) \wedge P(a))$
by *blast*

lemma *rall-one-point1* [simp]: $(\forall x[M]. x=a \longrightarrow P(x)) \leftrightarrow (M(a) \longrightarrow P(a))$
by *blast*

lemma *rall-one-point2* [simp]: $(\forall x[M]. a=x \longrightarrow P(x)) \leftrightarrow (M(a) \longrightarrow P(a))$
by *blast*

15.2.4 Sets as Classes

definition

setclass :: $[i,i] \Rightarrow o$ ($\langle \langle \text{open-block notation} = \langle \text{prefix setclass} \rangle \rangle \#\# - \rangle$ [40] 40)

where

setclass(*A*) $\equiv \lambda x. x \in A$

lemma *setclass-iff* [simp]: *setclass*(*A*,*x*) $\leftrightarrow x \in A$
by (*simp add: setclass-def*)

lemma *rall-setclass-is-ball* [simp]: $(\forall x[\#\#A]. P(x)) \leftrightarrow (\forall x \in A. P(x))$
by *auto*

lemma *rex-setclass-is-bex* [simp]: $(\exists x[\#\#A]. P(x)) \leftrightarrow (\exists x \in A. P(x))$
by *auto*

ML

```
<
val Ord-atomize =
  atomize ([(const-name <oall>, @{thms ospec}), (const-name <rall>, @{thms rspec})]
  @
    ZF-conn-pairs, ZF-mem-pairs);
>
declaration <fn - =>
  Simplifier.map-ss (Simplifier.set-mksimps (fn ctxt =>
    map mk-eq o Ord-atomize o Variable.gen-all ctxt))
>
```


Setting up the one-point-rule simproc

```
simproc-setup defined-rer ( $\exists x[M]. P(x) \wedge Q(x)$ ) =  $\langle$   

   $K (Quantifier1.rearrange-Bex (fn\ ctxt => unfold-tac\ ctxt\ @\{thms\ rer-def\}))$   

 $\rangle$ 
```

```
simproc-setup defined-rall ( $\forall x[M]. P(x) \longrightarrow Q(x)$ ) =  $\langle$   

   $K (Quantifier1.rearrange-Ball (fn\ ctxt => unfold-tac\ ctxt\ @\{thms\ rall-def\}))$   

 $\rangle$ 
```

end

16 The Natural numbers As a Least Fixed Point

theory *Nat* **imports** *OrdQuant Bool* **begin**

definition

```
nat :: i where  

nat  $\equiv lfp(Inf, \lambda X. \{0\} \cup \{succ(i). i \in X\})$ 
```

definition

```
quasinat :: i  $\Rightarrow$  o where  

quasinat(n)  $\equiv n=0 \mid (\exists m. n = succ(m))$ 
```

definition

```
nat-case :: [i, i $\Rightarrow$ i, i $\Rightarrow$ i] where  

nat-case(a,b,k)  $\equiv THE\ y. k=0 \wedge y=a \mid (\exists x. k=succ(x) \wedge y=b(x))$ 
```

definition

```
nat-rec :: [i, i, [i,i] $\Rightarrow$ i] $\Rightarrow$ i where  

nat-rec(k,a,b)  $\equiv$   

  wfrec(Memrel(nat), k,  $\lambda n\ f. nat-case(a, \lambda m. b(m, f'm), n)$ )
```

definition

```
Le :: i where  

Le  $\equiv \{\langle x,y \rangle : nat*nat. x \leq y\}$ 
```

definition

```
Lt :: i where  

Lt  $\equiv \{\langle x, y \rangle : nat*nat. x < y\}$ 
```

definition

```
Ge :: i where  

Ge  $\equiv \{\langle x,y \rangle : nat*nat. y \leq x\}$ 
```

definition

$Gt :: i$ **where**
 $Gt \equiv \{\langle x, y \rangle : nat * nat. y < x\}$

definition

$greater-than :: i \Rightarrow i$ **where**
 $greater-than(n) \equiv \{i \in nat. n < i\}$

No need for a less-than operator: a natural number is its list of predecessors!

lemma *nat-bnd-mono*: $bnd-mono(Inf, \lambda X. \{0\} \cup \{succ(i). i \in X\})$
apply (*rule bnd-monoI*)
apply (*cut-tac infinity, blast, blast*)
done

lemmas *nat-unfold* = *nat-bnd-mono* [*THEN nat-def* [*THEN def-lfp-unfold*]]

lemma *nat-0I* [*iff, TC*]: $0 \in nat$
apply (*subst nat-unfold*)
apply (*rule singletonI* [*THEN UnI1*])
done

lemma *nat-succI* [*intro!, TC*]: $n \in nat \implies succ(n) \in nat$
apply (*subst nat-unfold*)
apply (*erule RepFunI* [*THEN UnI2*])
done

lemma *nat-1I* [*iff, TC*]: $1 \in nat$
by (*rule nat-0I* [*THEN nat-succI*])

lemma *nat-2I* [*iff, TC*]: $2 \in nat$
by (*rule nat-1I* [*THEN nat-succI*])

lemma *bool-subset-nat*: $bool \subseteq nat$
by (*blast elim!:* *boolE*)

lemmas *bool-into-nat* = *bool-subset-nat* [*THEN subsetD*]

16.1 Injectivity Properties and Induction

lemma *nat-induct* [*case-names 0 succ, induct set: nat*]:
 $\llbracket n \in nat; P(0); \bigwedge x. \llbracket x \in nat; P(x) \rrbracket \implies P(succ(x)) \rrbracket \implies P(n)$
by (*erule def-induct* [*OF nat-def nat-bnd-mono*], *blast*)

lemma *natE*:
assumes $n \in nat$
obtains $(0) n=0 \mid (succ) x$ **where** $x \in nat \ n=succ(x)$
using *assms*

by (rule nat-unfold [THEN equalityD1, THEN subsetD, THEN UnE]) auto

lemma nat-into-Ord [simp]: $n \in \text{nat} \implies \text{Ord}(n)$
by (erule nat-induct, auto)

lemmas nat-0-le = nat-into-Ord [THEN Ord-0-le]

lemmas nat-le-refl = nat-into-Ord [THEN le-refl]

lemma Ord-nat [iff]: $\text{Ord}(\text{nat})$
apply (rule OrdI)
apply (erule tac [2] nat-into-Ord [THEN Ord-is-Transset])
unfolding Transset-def
apply (rule ballI)
apply (erule nat-induct, auto)
done

lemma Limit-nat [iff]: $\text{Limit}(\text{nat})$
unfolding Limit-def
apply (safe intro!: ltI Ord-nat)
apply (erule ltD)
done

lemma naturals-not-limit: $a \in \text{nat} \implies \neg \text{Limit}(a)$
by (induct a rule: nat-induct, auto)

lemma succ-natD: $\text{succ}(i): \text{nat} \implies i \in \text{nat}$
by (rule Ord-trans [OF succI1], auto)

lemma nat-succ-iff [iff]: $\text{succ}(n): \text{nat} \longleftrightarrow n \in \text{nat}$
by (blast dest!: succ-natD)

lemma nat-le-Limit: $\text{Limit}(i) \implies \text{nat} \leq i$
apply (rule subset-imp-le)
apply (simp-all add: Limit-is-Ord)
apply (rule subsetI)
apply (erule nat-induct)
apply (erule Limit-has-0 [THEN ltD])
apply (blast intro: Limit-has-succ [THEN ltD] ltI Limit-is-Ord)
done

lemmas succ-in-naturalD = Ord-trans [OF succI1 - nat-into-Ord]

lemma lt-nat-in-nat: $\llbracket m < n; \ n \in \text{nat} \rrbracket \implies m \in \text{nat}$
apply (erule ltE)
apply (erule Ord-trans, assumption, simp)

done

lemma *le-in-nat*: $\llbracket m \leq n; n \in \text{nat} \rrbracket \implies m \in \text{nat}$
by (*blast dest!: lt-nat-in-nat*)

16.2 Variations on Mathematical Induction

lemmas *complete-induct* = *Ord-induct* [*OF* - *Ord-nat*, *case-names less*, *consumes 1*]

lemma *complete-induct-rule* [*case-names less*, *consumes 1*]:
 $i \in \text{nat} \implies (\bigwedge x. x \in \text{nat} \implies (\bigwedge y. y \in x \implies P(y)) \implies P(x)) \implies P(i)$
using *complete-induct* [*of i P*] **by** *simp*

lemma *nat-induct-from*:

assumes $m \leq n$ $m \in \text{nat}$ $n \in \text{nat}$
and $P(m)$
and $\bigwedge x. \llbracket x \in \text{nat}; m \leq x; P(x) \rrbracket \implies P(\text{succ}(x))$
shows $P(n)$
proof –
from *assms*(3) **have** $m \leq n \longrightarrow P(m) \longrightarrow P(n)$
by (*rule nat-induct*) (*use assms*(5) **in** $\langle \text{simp-all add: distrib-simps le-succ-iff} \rangle$)
with *assms*(1,2,4) **show** *?thesis* **by** *blast*
qed

lemma *diff-induct* [*case-names 0 0-succ succ-succ*, *consumes 2*]:

$\llbracket m \in \text{nat}; n \in \text{nat};$
 $\bigwedge x. x \in \text{nat} \implies P(x, 0);$
 $\bigwedge y. y \in \text{nat} \implies P(0, \text{succ}(y));$
 $\bigwedge x y. \llbracket x \in \text{nat}; y \in \text{nat}; P(x, y) \rrbracket \implies P(\text{succ}(x), \text{succ}(y)) \rrbracket$
 $\implies P(m, n)$
apply (*erule-tac* $x = m$ **in** *rev-bspec*)
apply (*erule nat-induct*, *simp*)
apply (*rule ballI*)
apply (*rename-tac i j*)
apply (*erule-tac* $n=j$ **in** *nat-induct*, *auto*)
done

lemma *succ-lt-induct-lemma* [*rule-format*]:

$m \in \text{nat} \implies P(m, \text{succ}(m)) \longrightarrow (\forall x \in \text{nat}. P(m, x) \longrightarrow P(m, \text{succ}(x))) \longrightarrow$
 $(\forall n \in \text{nat}. m < n \longrightarrow P(m, n))$
apply (*erule nat-induct*)
apply (*intro impI*, *rule nat-induct* [*THEN ballI*])
prefer 4 **apply** (*intro impI*, *rule nat-induct* [*THEN ballI*])

apply (*auto simp add: le-iff*)
done

lemma *succ-lt-induct*:

$$\begin{aligned} & \llbracket m < n; \ n \in \text{nat}; \\ & \quad P(m, \text{succ}(m)); \\ & \quad \bigwedge x. \llbracket x \in \text{nat}; \ P(m, x) \rrbracket \implies P(m, \text{succ}(x)) \rrbracket \\ & \implies P(m, n) \end{aligned}$$

by (*blast intro: succ-lt-induct-lemma lt-nat-in-nat*)

16.3 quasinat: to allow a case-split rule for *nat-case*

True if the argument is zero or any successor

lemma [*iff*]: *quasinat*(0)
by (*simp add: quasinat-def*)

lemma [*iff*]: *quasinat*(*succ*(*x*))
by (*simp add: quasinat-def*)

lemma *nat-imp-quasinat*: $n \in \text{nat} \implies \text{quasinat}(n)$
by (*erule natE, simp-all*)

lemma *non-nat-case*: $\neg \text{quasinat}(x) \implies \text{nat-case}(a, b, x) = 0$
by (*simp add: quasinat-def nat-case-def*)

lemma *nat-cases-disj*: $k=0 \mid (\exists y. k = \text{succ}(y)) \mid \neg \text{quasinat}(k)$
apply (*case-tac k=0, simp*)
apply (*case-tac $\exists m. k = \text{succ}(m)$*)
apply (*simp-all add: quasinat-def*)
done

lemma *nat-cases*:

$$\llbracket k=0 \implies P; \ \bigwedge y. k = \text{succ}(y) \implies P; \ \neg \text{quasinat}(k) \implies P \rrbracket \implies P$$

by (*insert nat-cases-disj [of k], blast*)

lemma *nat-case-0* [*simp*]: *nat-case*(*a*, *b*, 0) = *a*
by (*simp add: nat-case-def*)

lemma *nat-case-succ* [*simp*]: *nat-case*(*a*, *b*, *succ*(*n*)) = *b*(*n*)
by (*simp add: nat-case-def*)

lemma *nat-case-type* [*TC*]:

$$\begin{aligned} & \llbracket n \in \text{nat}; \ a \in C(0); \ \bigwedge m. m \in \text{nat} \implies b(m): C(\text{succ}(m)) \rrbracket \\ & \implies \text{nat-case}(a, b, n) \in C(n) \end{aligned}$$

by (*erule nat-induct, auto*)

lemma *split-nat-case*:

```

    P(nat-case(a,b,k))  $\longleftrightarrow$ 
    ((k=0  $\longrightarrow$  P(a))  $\wedge$  ( $\forall x. k=succ(x) \longrightarrow P(b(x))$ )  $\wedge$  ( $\neg quasinat(k) \longrightarrow P(0)$ ))
  apply (rule nat-cases [of k])
  apply (auto simp add: non-nat-case)
done

```

16.4 Recursion on the Natural Numbers

```

lemma nat-rec-0: nat-rec(0,a,b) = a
  apply (rule nat-rec-def [THEN def-wfrec, THEN trans])
  apply (rule wf-Memrel)
  apply (rule nat-case-0)
done

```

```

lemma nat-rec-succ: m  $\in$  nat  $\implies$  nat-rec(succ(m),a,b) = b(m, nat-rec(m,a,b))
  apply (rule nat-rec-def [THEN def-wfrec, THEN trans])
  apply (rule wf-Memrel)
  apply (simp add: vimage-singleton-iff)
done

```

```

lemma Un-nat-type [TC]:  $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies i \cup j \in \text{nat}$ 
  apply (rule Un-least-lt [THEN ltD])
  apply (simp-all add: lt-def)
done

```

```

lemma Int-nat-type [TC]:  $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies i \cap j \in \text{nat}$ 
  apply (rule Int-greatest-lt [THEN ltD])
  apply (simp-all add: lt-def)
done

```

```

lemma nat-nonempty [simp]: nat  $\neq$  0
by blast

```

A natural number is the set of its predecessors

```

lemma nat-eq-Collect-lt: i  $\in$  nat  $\implies \{j \in \text{nat}. j < i\} = i$ 
  apply (rule equalityI)
  apply (blast dest: ltD)
  apply (auto simp add: Ord-mem-iff-lt)
  apply (blast intro: lt-trans)
done

```

```

lemma Le-iff [iff]:  $\langle x,y \rangle \in Le \longleftrightarrow x \leq y \wedge x \in \text{nat} \wedge y \in \text{nat}$ 
by (force simp add: Le-def)

```

end

17 Inductive and Coinductive Definitions

```

theory Inductive
imports Fixedpt QPair Nat
keywords
  inductive coinductive inductive-cases rep-datatype primrec :: thy-decl and
  domains intros monos con-defs type-intros type-elim
  elimination induction case-eqns recursor-eqns :: quasi-command
begin

```

```

lemma def-swap-iff:  $a \equiv b \implies a = c \longleftrightarrow c = b$ 
by blast

```

```

lemma def-trans:  $f \equiv g \implies g(a) = b \implies f(a) = b$ 
by simp

```

```

lemma refl-thin:  $\bigwedge P. a = a \implies P \implies P$  .

```

```

ML-file  $\langle ind-syntax.ML \rangle$ 
ML-file  $\langle Tools/ind-cases.ML \rangle$ 
ML-file  $\langle Tools/cartprod.ML \rangle$ 
ML-file  $\langle Tools/inductive-package.ML \rangle$ 
ML-file  $\langle Tools/induct-tacs.ML \rangle$ 
ML-file  $\langle Tools/primrec-package.ML \rangle$ 

```

```

ML  $\langle$ 
  structure Lfp =
    struct
      val oper      = Const  $\langle lfp \rangle$ 
      val bnd-mono = Const  $\langle bnd-mono \rangle$ 
      val bnd-monoI = @ $\{thm\ bnd-monoI\}$ 
      val subs     = @ $\{thm\ def-lfp-subset\}$ 
      val Tarski   = @ $\{thm\ def-lfp-unfold\}$ 
      val induct   = @ $\{thm\ def-induct\}$ 
    end;

```

```

structure Standard-Prod =
  struct
    val sigma    = Const  $\langle Sigma \rangle$ 
    val pair     = Const  $\langle Pair \rangle$ 
    val split-name = const-name  $\langle split \rangle$ 
    val pair-iff  = @ $\{thm\ Pair-iff\}$ 
    val split-eq = @ $\{thm\ split\}$ 
    val fsplitI  = @ $\{thm\ splitI\}$ 
    val fsplitD  = @ $\{thm\ splitD\}$ 
    val fsplitE  = @ $\{thm\ splitE\}$ 
  end;

```

```

structure Standard-CP = CartProd-Fun (Standard-Prod);

```

```

structure Standard-Sum =
  struct
    val sum      = Const ⟨sum⟩
    val inl      = Const ⟨Inl⟩
    val inr      = Const ⟨Inr⟩
    val elim     = Const ⟨case⟩
    val case-inl = @ { thm case-Inl }
    val case-inr = @ { thm case-Inr }
    val inl-iff  = @ { thm Inl-iff }
    val inr-iff  = @ { thm Inr-iff }
    val distinct = @ { thm Inl-Inr-iff }
    val distinct' = @ { thm Inr-Inl-iff }
    val free-SEs = Ind-Syntax.mk-free-SEs
                      [distinct, distinct', inl-iff, inr-iff, Standard-Prod.pair-iff]
  end;

structure Ind-Package =
  Add-inductive-def-Fun
    (structure Fp=Lfp and Pr=Standard-Prod and CP=Standard-CP
     and Su=Standard-Sum val coind = false);

structure Gfp =
  struct
    val oper      = Const ⟨gfp⟩
    val bnd-mono  = Const ⟨bnd-mono⟩
    val bnd-monoI = @ { thm bnd-monoI }
    val subs      = @ { thm def-gfp-subset }
    val Tarski    = @ { thm def-gfp-unfold }
    val induct    = @ { thm def-Collect-coinduct }
  end;

structure Quine-Prod =
  struct
    val sigma     = Const ⟨QSigma⟩
    val pair      = Const ⟨QPair⟩
    val split-name = const-name ⟨qsplit⟩
    val pair-iff  = @ { thm QPair-iff }
    val split-eq  = @ { thm qsplit }
    val fsplitI   = @ { thm qsplitI }
    val fsplitD   = @ { thm qsplitD }
    val fsplitE   = @ { thm qsplitE }
  end;

structure Quine-CP = CartProd-Fun (Quine-Prod);

structure Quine-Sum =

```



```

struct
  val sum      = Const ⟨qsum⟩
  val inl      = Const ⟨QInl⟩
  val inr      = Const ⟨QInr⟩
  val elim     = Const ⟨qcase⟩
  val case-inl = @ {thm qcase-QInl}
  val case-inr = @ {thm qcase-QInr}
  val inl-iff  = @ {thm QInl-iff}
  val inr-iff  = @ {thm QInr-iff}
  val distinct = @ {thm QInl-QInr-iff}
  val distinct' = @ {thm QInr-QInl-iff}
  val free-SEs = Ind-Syntax.mk-free-SEs
                    [distinct, distinct', inl-iff, inr-iff, Quine-Prod.pair-iff]
end;

structure CoInd-Package =
  Add-inductive-def-Fun (structure Fp=Gfp and Pr=Quine-Prod and CP=Quine-CP
    and Su=Quine-Sum val coind = true);

>

end

```

18 Epsilon Induction and Recursion

theory Epsilon imports Nat begin

definition

eclose :: $i \Rightarrow i$ **where**
eclose(A) $\equiv \bigcup_{n \in \text{nat.}} \text{nat-rec}(n, A, \lambda m r. \bigcup(r))$

definition

transrec :: $[i, [i, i] \Rightarrow i] \Rightarrow i$ **where**
transrec(a, H) $\equiv \text{wfrec}(\text{Memrel}(\text{eclose}(\{a\})), a, H)$

definition

rank :: $i \Rightarrow i$ **where**
rank(a) $\equiv \text{transrec}(a, \lambda x f. \bigcup_{y \in x. \text{succ}(f'y))$

definition

transrec2 :: $[i, i, [i, i] \Rightarrow i] \Rightarrow i$ **where**
transrec2(k, a, b) \equiv
transrec($k,$
 $\lambda i r. \text{if}(i=0, a,$
 $\text{if}(\exists j. i=\text{succ}(j),$
 $b(\text{THE } j. i=\text{succ}(j), r'(\text{THE } j. i=\text{succ}(j))),$
 $\bigcup_{j < i. r'j)))$

definition

$recursor :: [i, [i, i] \Rightarrow i, i] \Rightarrow i$ **where**
 $recursor(a, b, k) \equiv transrec(k, \lambda n f. nat-case(a, \lambda m. b(m, f^m), n))$

definition

$rec :: [i, i, [i, i] \Rightarrow i] \Rightarrow i$ **where**
 $rec(k, a, b) \equiv recursor(a, b, k)$

18.1 Basic Closure Properties

lemma *arg-subset-eclose*: $A \subseteq eclose(A)$

unfolding *eclose-def*

apply (*rule nat-rec-0* [*THEN equalityD2*, *THEN subset-trans*])

apply (*rule nat-0I* [*THEN UN-upper*])

done

lemmas *arg-into-eclose* = *arg-subset-eclose* [*THEN subsetD*]

lemma *Transset-eclose*: $Transset(eclose(A))$

unfolding *eclose-def Transset-def*

apply (*rule subsetI* [*THEN ballI*])

apply (*erule UN-E*)

apply (*rule nat-succI* [*THEN UN-I*, *assumption*])

apply (*erule nat-rec-succ* [*THEN ssubst*])

apply (*erule UnionI*, *assumption*)

done

lemmas *eclose-subset* =

Transset-eclose [*unfolded Transset-def*, *THEN bspec*]

lemmas *ecloseD* = *eclose-subset* [*THEN subsetD*]

lemmas *arg-in-eclose-sing* = *arg-subset-eclose* [*THEN singleton-subsetD*]

lemmas *arg-into-eclose-sing* = *arg-in-eclose-sing* [*THEN ecloseD*]

lemmas *eclose-induct* =

Transset-induct [*OF - Transset-eclose*, *induct set: eclose*]

lemma *eps-induct*:

$\llbracket \bigwedge x. \forall y \in x. P(y) \implies P(x) \rrbracket \implies P(a)$

by (*rule arg-in-eclose-sing* [*THEN eclose-induct*], *blast*)

18.2 Leastness of *eclose*

lemma *eclose-least-lemma*:

```

     $\llbracket \text{Transset}(X); A \leq X; n \in \text{nat} \rrbracket \implies \text{nat-rec}(n, A, \lambda m r. \bigcup(r)) \subseteq X$ 
  unfolding Transset-def
  apply (erule nat-induct)
  apply (simp add: nat-rec-0)
  apply (simp add: nat-rec-succ, blast)
done

```

```

lemma eclose-least:
   $\llbracket \text{Transset}(X); A \leq X \rrbracket \implies \text{eclose}(A) \subseteq X$ 
  unfolding eclose-def
  apply (rule eclose-least-lemma [THEN UN-least], assumption+)
done

```

```

lemma eclose-induct-down [consumes 1]:
   $\llbracket a \in \text{eclose}(b);$ 
     $\bigwedge y. \llbracket y \in b \rrbracket \implies P(y);$ 
     $\bigwedge y z. \llbracket y \in \text{eclose}(b); P(y); z \in y \rrbracket \implies P(z)$ 
 $\rrbracket \implies P(a)$ 
  apply (rule eclose-least [THEN subsetD, THEN CollectD2, of eclose(b)])
  prefer 3 apply assumption
  unfolding Transset-def
  apply (blast intro: ecloseD)
  apply (blast intro: arg-subset-eclose [THEN subsetD])
done

```

```

lemma Transset-eclose-eq-arg:  $\text{Transset}(X) \implies \text{eclose}(X) = X$ 
  apply (erule equalityI [OF eclose-least arg-subset-eclose])
  apply (rule subset-refl)
done

```

A transitive set either is empty or contains the empty set.

```

lemma Transset-0-lemma [rule-format]:  $\text{Transset}(A) \implies x \in A \longrightarrow 0 \in A$ 
  apply (simp add: Transset-def)
  apply (rule-tac a=x in eps-induct, clarify)
  apply (drule bspec, assumption)
  apply (case-tac x=0, auto)
done

```

```

lemma Transset-0-disj:  $\text{Transset}(A) \implies A=0 \mid 0 \in A$ 
  by (blast dest: Transset-0-lemma)

```

18.3 Epsilon Recursion

```

lemma mem-eclose-trans:  $\llbracket A \in \text{eclose}(B); B \in \text{eclose}(C) \rrbracket \implies A \in \text{eclose}(C)$ 
  by (rule eclose-least [OF Transset-eclose eclose-subset, THEN subsetD],
    assumption+)

```

lemma *mem-eclose-sing-trans*:

$\llbracket A \in \text{eclose}(\{B\}); B \in \text{eclose}(\{C\}) \rrbracket \implies A \in \text{eclose}(\{C\})$
by (*rule* *eclose-least* [*OF Transset-eclose singleton-subsetI*, *THEN subsetD*],
assumption+))

lemma *under-Memrel*: $\llbracket \text{Transset}(i); j \in i \rrbracket \implies \text{Memrel}(i) - \{j\} = j$
by (*unfold Transset-def*, *blast*)

lemma *lt-Memrel*: $j < i \implies \text{Memrel}(i) - \{j\} = j$
by (*simp add: lt-def Ord-def under-Memrel*)

lemmas *under-Memrel-eclose* = *Transset-eclose* [*THEN under-Memrel*]

lemmas *wfrec-ssubst* = *wf-Memrel* [*THEN wfrec*, *THEN ssubst*]

lemma *wfrec-eclose-eq*:

$\llbracket k \in \text{eclose}(\{j\}); j \in \text{eclose}(\{i\}) \rrbracket \implies$
 $\text{wfrec}(\text{Memrel}(\text{eclose}(\{i\})), k, H) = \text{wfrec}(\text{Memrel}(\text{eclose}(\{j\})), k, H)$
apply (*erule* *eclose-induct*)
apply (*rule* *wfrec-ssubst*)
apply (*rule* *wfrec-ssubst*)
apply (*simp add: under-Memrel-eclose mem-eclose-sing-trans* [*of - j i*])
done

lemma *wfrec-eclose-eq2*:

$k \in i \implies \text{wfrec}(\text{Memrel}(\text{eclose}(\{i\})), k, H) = \text{wfrec}(\text{Memrel}(\text{eclose}(\{k\})), k, H)$
apply (*rule* *arg-in-eclose-sing* [*THEN wfrec-eclose-eq*])
apply (*erule* *arg-into-eclose-sing*)
done

lemma *transrec*: $\text{transrec}(a, H) = H(a, \lambda x \in a. \text{transrec}(x, H))$

unfolding *transrec-def*
apply (*rule* *wfrec-ssubst*)
apply (*simp add: wfrec-eclose-eq2 arg-in-eclose-sing under-Memrel-eclose*)
done

lemma *def-transrec*:

$\llbracket \bigwedge x. f(x) \equiv \text{transrec}(x, H) \rrbracket \implies f(a) = H(a, \lambda x \in a. f(x))$
apply *simp*
apply (*rule* *transrec*)
done

lemma *transrec-type*:

$\llbracket \bigwedge x u. \llbracket x \in \text{eclose}(\{a\}); u \in \text{Pi}(x, B) \rrbracket \implies H(x, u) \in B(x)$
 $\implies \text{transrec}(a, H) \in B(a)$
apply (*rule-tac* $i = a$ **in** *arg-in-eclose-sing* [*THEN* *eclose-induct*])
apply (*subst* *transrec*)

apply (*simp add: lam-type*)
done

lemma *eclose-sing-Ord*: $\text{Ord}(i) \implies \text{eclose}(\{i\}) \subseteq \text{succ}(i)$
apply (*erule Ord-is-Transset [THEN Transset-succ, THEN eclose-least]*)
apply (*rule succI1 [THEN singleton-subsetI]*)
done

lemma *succ-subset-eclose-sing*: $\text{succ}(i) \subseteq \text{eclose}(\{i\})$
apply (*insert arg-subset-eclose [of {i}], simp*)
apply (*frule eclose-subset, blast*)
done

lemma *eclose-sing-Ord-eq*: $\text{Ord}(i) \implies \text{eclose}(\{i\}) = \text{succ}(i)$
apply (*rule equalityI*)
apply (*erule eclose-sing-Ord*)
apply (*rule succ-subset-eclose-sing*)
done

lemma *Ord-transrec-type*:
assumes *jini*: $j \in i$
and *ordi*: $\text{Ord}(i)$
and *minor*: $\bigwedge x u. \llbracket x \in i; u \in \text{Pi}(x, B) \rrbracket \implies H(x, u) \in B(x)$
shows $\text{transrec}(j, H) \in B(j)$
apply (*rule transrec-type*)
apply (*insert jini ordi*)
apply (*blast intro!: minor*)
intro: *Ord-trans*
dest: *Ord-in-Ord [THEN eclose-sing-Ord, THEN subsetD]*)
done

18.4 Rank

lemma *rank*: $\text{rank}(a) = (\bigcup y \in a. \text{succ}(\text{rank}(y)))$
by (*subst rank-def [THEN def-transrec], simp*)

lemma *Ord-rank* [*simp*]: $\text{Ord}(\text{rank}(a))$
apply (*rule-tac a=a in eps-induct*)
apply (*subst rank*)
apply (*rule Ord-succ [THEN Ord-UN]*)
apply (*erule bspec, assumption*)
done

lemma *rank-of-Ord*: $\text{Ord}(i) \implies \text{rank}(i) = i$
apply (*erule trans-induct*)
apply (*subst rank*)
apply (*simp add: Ord-equality*)
done

```

lemma rank-lt:  $a \in b \implies \text{rank}(a) < \text{rank}(b)$ 
apply (rule-tac  $a1 = b$  in rank [THEN ssubst])
apply (erule UN-I [THEN ltI])
apply (rule-tac [2] Ord-UN, auto)
done

```

```

lemma eclose-rank-lt:  $a \in \text{eclose}(b) \implies \text{rank}(a) < \text{rank}(b)$ 
apply (erule eclose-induct-down)
apply (erule rank-lt)
apply (erule rank-lt [THEN lt-trans], assumption)
done

```

```

lemma rank-mono:  $a \leq b \implies \text{rank}(a) \leq \text{rank}(b)$ 
apply (rule subset-imp-le)
apply (auto simp add: rank [of a] rank [of b])
done

```

```

lemma rank-Pow:  $\text{rank}(\text{Pow}(a)) = \text{succ}(\text{rank}(a))$ 
apply (rule rank [THEN trans])
apply (rule le-anti-sym)
apply (rule-tac [2] UN-upper-le)
apply (rule UN-least-le)
apply (auto intro: rank-mono simp add: Ord-UN)
done

```

```

lemma rank-0 [simp]:  $\text{rank}(0) = 0$ 
by (rule rank [THEN trans], blast)

```

```

lemma rank-succ [simp]:  $\text{rank}(\text{succ}(x)) = \text{succ}(\text{rank}(x))$ 
apply (rule rank [THEN trans])
apply (rule equalityI [OF UN-least succI1 [THEN UN-upper]])
apply (erule succE, blast)
apply (erule rank-lt [THEN leI, THEN succ-leI, THEN le-imp-subset])
done

```

```

lemma rank-Union:  $\text{rank}(\bigcup(A)) = (\bigcup x \in A. \text{rank}(x))$ 
apply (rule equalityI)
apply (rule-tac [2] rank-mono [THEN le-imp-subset, THEN UN-least])
apply (erule-tac [2] Union-upper)
apply (subst rank)
apply (rule UN-least)
apply (erule UnionE)
apply (rule subset-trans)
apply (erule-tac [2] RepFunI [THEN Union-upper])
apply (erule rank-lt [THEN succ-leI, THEN le-imp-subset])
done

```

```

lemma rank-eclose:  $\text{rank}(\text{eclose}(a)) = \text{rank}(a)$ 
apply (rule le-anti-sym)

```

```

apply (rule-tac [2] arg-subset-eclose [THEN rank-mono])
apply (rule-tac a1 = eclose (a) in rank [THEN ssubst])
apply (rule Ord-rank [THEN UN-least-le])
apply (erule eclose-rank-lt [THEN succ-leI])
done

lemma rank-pair1: rank(a) < rank((a,b))
  unfolding Pair-def
apply (rule consI1 [THEN rank-lt, THEN lt-trans])
apply (rule consI1 [THEN consI2, THEN rank-lt])
done

lemma rank-pair2: rank(b) < rank((a,b))
  unfolding Pair-def
apply (rule consI1 [THEN consI2, THEN rank-lt, THEN lt-trans])
apply (rule consI1 [THEN consI2, THEN rank-lt])
done

lemma the-equality-if:
  
$$P(a) \implies (THE\ x.\ P(x)) = (if\ (\exists!x.\ P(x))\ then\ a\ else\ 0)$$

by (simp add: the-0 the-equality2)

```

```

lemma rank-apply:  $\llbracket i \in domain(f); function(f) \rrbracket \implies rank(f'i) < rank(f)$ 
apply clarify
apply (simp add: function-apply-equality)
apply (blast intro: lt-trans rank-lt rank-pair2)
done

```

18.5 Corollaries of Leastness

```

lemma mem-eclose-subset:  $A \in B \implies eclose(A) \leq eclose(B)$ 
apply (rule Transset-eclose [THEN eclose-least])
apply (erule arg-into-eclose [THEN eclose-subset])
done

```

```

lemma eclose-mono:  $A \leq B \implies eclose(A) \subseteq eclose(B)$ 
apply (rule Transset-eclose [THEN eclose-least])
apply (erule subset-trans)
apply (rule arg-subset-eclose)
done

```

```

lemma eclose-idem:  $eclose(eclose(A)) = eclose(A)$ 
apply (rule equalityI)
apply (rule eclose-least [OF Transset-eclose subset-refl])
apply (rule arg-subset-eclose)

```

done

lemma *transrec2-0* [*simp*]: $\text{transrec2}(0, a, b) = a$
by (rule *transrec2-def* [*THEN* *def-transrec*, *THEN* *trans*], *simp*)

lemma *transrec2-succ* [*simp*]: $\text{transrec2}(\text{succ}(i), a, b) = b(i, \text{transrec2}(i, a, b))$
apply (rule *transrec2-def* [*THEN* *def-transrec*, *THEN* *trans*])
apply (*simp* add: *the-equality if-P*)
done

lemma *transrec2-Limit*:
 $\text{Limit}(i) \implies \text{transrec2}(i, a, b) = (\bigcup j < i. \text{transrec2}(j, a, b))$
apply (rule *transrec2-def* [*THEN* *def-transrec*, *THEN* *trans*])
apply (*auto simp* add: *ONat-Union-def*)
done

lemma *def-transrec2*:
 $(\bigwedge x. f(x) \equiv \text{transrec2}(x, a, b))$
 $\implies f(0) = a \wedge$
 $f(\text{succ}(i)) = b(i, f(i)) \wedge$
 $(\text{Limit}(K) \longrightarrow f(K) = (\bigcup j < K. f(j)))$
by (*simp* add: *transrec2-Limit*)

lemmas *recursor-lemma* = *recursor-def* [*THEN* *def-transrec*, *THEN* *trans*]

lemma *recursor-0*: $\text{recursor}(a, b, 0) = a$
by (rule *nat-case-0* [*THEN* *recursor-lemma*])

lemma *recursor-succ*: $\text{recursor}(a, b, \text{succ}(m)) = b(m, \text{recursor}(a, b, m))$
by (rule *recursor-lemma*, *simp*)

lemma *rec-0* [*simp*]: $\text{rec}(0, a, b) = a$
unfolding *rec-def*
apply (rule *recursor-0*)
done

lemma *rec-succ* [*simp*]: $\text{rec}(\text{succ}(m), a, b) = b(m, \text{rec}(m, a, b))$
unfolding *rec-def*
apply (rule *recursor-succ*)
done

lemma *rec-type*:

$$\begin{aligned} & \llbracket n \in \text{nat}; \\ & \quad a \in C(0); \\ & \quad \bigwedge m z. \llbracket m \in \text{nat}; \quad z \in C(m) \rrbracket \implies b(m,z): C(\text{succ}(m)) \rrbracket \\ & \implies \text{rec}(n,a,b) \in C(n) \end{aligned}$$

by (*erule nat-induct, auto*)
end

19 Partial and Total Orderings: Basic Definitions and Properties

theory *Order* **imports** *WF Perm* **begin**

We adopt the following convention: *ord* is used for strict orders and *order* is used for their reflexive counterparts.

definition

$$\begin{aligned} \text{part-ord} &:: [i,i] \Rightarrow o & \textbf{where} \\ \text{part-ord}(A,r) &\equiv \text{irrefl}(A,r) \wedge \text{trans}[A](r) \end{aligned}$$

definition

$$\begin{aligned} \text{linear} &:: [i,i] \Rightarrow o & \textbf{where} \\ \text{linear}(A,r) &\equiv (\forall x \in A. \forall y \in A. \langle x,y \rangle : r \mid x=y \mid \langle y,x \rangle : r) \end{aligned}$$

definition

$$\begin{aligned} \text{tot-ord} &:: [i,i] \Rightarrow o & \textbf{where} \\ \text{tot-ord}(A,r) &\equiv \text{part-ord}(A,r) \wedge \text{linear}(A,r) \end{aligned}$$

definition

$$\text{preorder-on}(A, r) \equiv \text{refl}(A, r) \wedge \text{trans}[A](r)$$

definition

$$\text{partial-order-on}(A, r) \equiv \text{preorder-on}(A, r) \wedge \text{antisym}(r)$$

abbreviation

$$\text{Preorder}(r) \equiv \text{preorder-on}(\text{field}(r), r)$$

abbreviation

$$\text{Partial-order}(r) \equiv \text{partial-order-on}(\text{field}(r), r)$$

definition

$$\begin{aligned} \text{well-ord} &:: [i,i] \Rightarrow o & \textbf{where} \\ \text{well-ord}(A,r) &\equiv \text{tot-ord}(A,r) \wedge \text{wf}[A](r) \end{aligned}$$

definition

$$\begin{aligned} \text{mono-map} &:: [i,i,i,i] \Rightarrow i & \textbf{where} \\ \text{mono-map}(A,r,B,s) &\equiv \end{aligned}$$

$$\{f \in A \rightarrow B. \forall x \in A. \forall y \in A. \langle x, y \rangle : r \longrightarrow \langle f'x, f'y \rangle : s\}$$

definition

$ord\text{-}iso :: [i, i, i, i] \Rightarrow i$ ($\langle \langle notation = \langle infix\ ord\text{-}iso \rangle \langle -, - \rangle \cong / \langle -, - \rangle \rangle 51$) **where**
 $\langle A, r \rangle \cong \langle B, s \rangle \equiv$
 $\{f \in bij(A, B). \forall x \in A. \forall y \in A. \langle x, y \rangle : r \longleftrightarrow \langle f'x, f'y \rangle : s\}$

definition

$pred :: [i, i, i] \Rightarrow i$ **where**
 $pred(A, x, r) \equiv \{y \in A. \langle y, x \rangle : r\}$

definition

$ord\text{-}iso\text{-}map :: [i, i, i, i] \Rightarrow i$ **where**
 $ord\text{-}iso\text{-}map(A, r, B, s) \equiv$
 $\bigcup x \in A. \bigcup y \in B. \bigcup f \in ord\text{-}iso(pred(A, x, r), r, pred(B, y, s), s). \{\langle x, y \rangle\}$

definition

$first :: [i, i, i] \Rightarrow o$ **where**
 $first(u, X, R) \equiv u \in X \wedge (\forall v \in X. v \neq u \longrightarrow \langle u, v \rangle \in R)$

19.1 Immediate Consequences of the Definitions

lemma *part-ord-Imp-asym*:

$part\text{-}ord(A, r) \Longrightarrow asym(r \cap A * A)$

by (*unfold part-ord-def irrefl-def trans-on-def asym-def, blast*)

lemma *linearE*:

$\llbracket linear(A, r); x \in A; y \in A;$
 $\langle x, y \rangle : r \Longrightarrow P; x = y \Longrightarrow P; \langle y, x \rangle : r \Longrightarrow P$
 $\Longrightarrow P$

by (*simp add: linear-def, blast*)

lemma *well-ordI*:

$\llbracket wf[A](r); linear(A, r) \rrbracket \Longrightarrow well\text{-}ord(A, r)$

apply (*simp add: irrefl-def part-ord-def tot-ord-def*
trans-on-def well-ord-def wf-on-not-refl)

apply (*fast elim: linearE wf-on-asym wf-on-chain3*)
done

lemma *well-ord-is-wf*:

$well\text{-}ord(A, r) \Longrightarrow wf[A](r)$

by (*unfold well-ord-def, safe*)

lemma *well-ord-is-trans-on*:

$well\text{-}ord(A, r) \Longrightarrow trans[A](r)$

by (*unfold well-ord-def tot-ord-def part-ord-def, safe*)

lemma *well-ord-is-linear*: $\text{well-ord}(A,r) \implies \text{linear}(A,r)$
by (*unfold well-ord-def tot-ord-def, blast*)

lemma *pred-iff*: $y \in \text{pred}(A,x,r) \longleftrightarrow \langle y,x \rangle : r \wedge y \in A$
by (*unfold pred-def, blast*)

lemmas *predI* = *conjI* [*THEN pred-iff* [*THEN iffD2*]]

lemma *predE*: $\llbracket y \in \text{pred}(A,x,r); \llbracket y \in A; \langle y,x \rangle : r \rrbracket \implies P \rrbracket \implies P$
by (*simp add: pred-def*)

lemma *pred-subset-under*: $\text{pred}(A,x,r) \subseteq r - \{x\}$
by (*simp add: pred-def, blast*)

lemma *pred-subset*: $\text{pred}(A,x,r) \subseteq A$
by (*simp add: pred-def, blast*)

lemma *pred-pred-eq*:
 $\text{pred}(\text{pred}(A,x,r), y, r) = \text{pred}(A,x,r) \cap \text{pred}(A,y,r)$
by (*simp add: pred-def, blast*)

lemma *trans-pred-pred-eq*:
 $\llbracket \text{trans}[A](r); \langle y,x \rangle : r; x \in A; y \in A \rrbracket$
 $\implies \text{pred}(\text{pred}(A,x,r), y, r) = \text{pred}(A,y,r)$
by (*unfold trans-on-def pred-def, blast*)

19.2 Restricting an Ordering's Domain

lemma *part-ord-subset*:
 $\llbracket \text{part-ord}(A,r); B \leq A \rrbracket \implies \text{part-ord}(B,r)$
by (*unfold part-ord-def irreft-def trans-on-def, blast*)

lemma *linear-subset*:
 $\llbracket \text{linear}(A,r); B \leq A \rrbracket \implies \text{linear}(B,r)$
by (*unfold linear-def, blast*)

lemma *tot-ord-subset*:
 $\llbracket \text{tot-ord}(A,r); B \leq A \rrbracket \implies \text{tot-ord}(B,r)$
unfolding *tot-ord-def*
apply (*fast elim!: part-ord-subset linear-subset*)
done

lemma *well-ord-subset*:
 $\llbracket \text{well-ord}(A,r); B \leq A \rrbracket \implies \text{well-ord}(B,r)$
unfolding *well-ord-def*

apply (*fast elim!*: *tot-ord-subset wf-on-subset-A*)
done

lemma *irrefl-Int-iff*: $\text{irrefl}(A, r \cap A * A) \longleftrightarrow \text{irrefl}(A, r)$
by (*unfold irrefl-def, blast*)

lemma *trans-on-Int-iff*: $\text{trans}[A](r \cap A * A) \longleftrightarrow \text{trans}[A](r)$
by (*unfold trans-on-def, blast*)

lemma *part-ord-Int-iff*: $\text{part-ord}(A, r \cap A * A) \longleftrightarrow \text{part-ord}(A, r)$
unfolding *part-ord-def*
apply (*simp add: irrefl-Int-iff trans-on-Int-iff*)
done

lemma *linear-Int-iff*: $\text{linear}(A, r \cap A * A) \longleftrightarrow \text{linear}(A, r)$
by (*unfold linear-def, blast*)

lemma *tot-ord-Int-iff*: $\text{tot-ord}(A, r \cap A * A) \longleftrightarrow \text{tot-ord}(A, r)$
unfolding *tot-ord-def*
apply (*simp add: part-ord-Int-iff linear-Int-iff*)
done

lemma *wf-on-Int-iff*: $\text{wf}[A](r \cap A * A) \longleftrightarrow \text{wf}[A](r)$
apply (*unfold wf-on-def wf-def, fast*)
done

lemma *well-ord-Int-iff*: $\text{well-ord}(A, r \cap A * A) \longleftrightarrow \text{well-ord}(A, r)$
unfolding *well-ord-def*
apply (*simp add: tot-ord-Int-iff wf-on-Int-iff*)
done

19.3 Empty and Unit Domains

lemma *wf-on-any-0*: $\text{wf}[A](0)$
by (*simp add: wf-on-def wf-def, fast*)

19.3.1 Relations over the Empty Set

lemma *irrefl-0*: $\text{irrefl}(0, r)$
by (*unfold irrefl-def, blast*)

lemma *trans-on-0*: $\text{trans}[0](r)$
by (*unfold trans-on-def, blast*)

lemma *part-ord-0*: $\text{part-ord}(0, r)$
unfolding *part-ord-def*
apply (*simp add: irrefl-0 trans-on-0*)

done

lemma *linear-0*: *linear*(0,*r*)
by (*unfold linear-def*, *blast*)

lemma *tot-ord-0*: *tot-ord*(0,*r*)
unfolding *tot-ord-def*
apply (*simp add: part-ord-0 linear-0*)
done

lemma *wf-on-0*: *wf*[0](*r*)
by (*unfold wf-on-def wf-def*, *blast*)

lemma *well-ord-0*: *well-ord*(0,*r*)
unfolding *well-ord-def*
apply (*simp add: tot-ord-0 wf-on-0*)
done

19.3.2 The Empty Relation Well-Orders the Unit Set

by Grabczewski

lemma *tot-ord-unit*: *tot-ord*({*a*},0)
by (*simp add: irrefl-def trans-on-def part-ord-def linear-def tot-ord-def*)

lemma *well-ord-unit*: *well-ord*({*a*},0)
unfolding *well-ord-def*
apply (*simp add: tot-ord-unit wf-on-any-0*)
done

19.4 Order-Isomorphisms

Suppes calls them "similarities"

lemma *mono-map-is-fun*: $f \in \text{mono-map}(A,r,B,s) \implies f \in A \multimap B$
by (*simp add: mono-map-def*)

lemma *mono-map-is-inj*:
 $\llbracket \text{linear}(A,r); \text{wf}[B](s); f \in \text{mono-map}(A,r,B,s) \rrbracket \implies f \in \text{inj}(A,B)$
apply (*unfold mono-map-def inj-def*, *clarify*)
apply (*erule-tac x=w and y=x in linearE*, *assumption+*)
apply (*force intro: apply-type dest: wf-on-not-refl*) +
done

lemma *ord-isoI*:
 $\llbracket f \in \text{bij}(A, B);$
 $\bigwedge x y. \llbracket x \in A; y \in A \rrbracket \implies \langle x, y \rangle \in r \longleftrightarrow \langle f'x, f'y \rangle \in s \rrbracket$
 $\implies f \in \text{ord-iso}(A,r,B,s)$
by (*simp add: ord-iso-def*)

lemma *ord-iso-is-mono-map*:

$f \in \text{ord-iso}(A, r, B, s) \implies f \in \text{mono-map}(A, r, B, s)$

apply (*simp add: ord-iso-def mono-map-def*)

apply (*blast dest!: bij-is-fun*)

done

lemma *ord-iso-is-bij*:

$f \in \text{ord-iso}(A, r, B, s) \implies f \in \text{bij}(A, B)$

by (*simp add: ord-iso-def*)

lemma *ord-iso-apply*:

$\llbracket f \in \text{ord-iso}(A, r, B, s); \langle x, y \rangle: r; x \in A; y \in A \rrbracket \implies \langle f'x, f'y \rangle \in s$

by (*simp add: ord-iso-def*)

lemma *ord-iso-converse*:

$\llbracket f \in \text{ord-iso}(A, r, B, s); \langle x, y \rangle: s; x \in B; y \in B \rrbracket$

$\implies \langle \text{converse}(f) ' x, \text{converse}(f) ' y \rangle \in r$

apply (*simp add: ord-iso-def, clarify*)

apply (*erule bspec [THEN bspec, THEN iffD2]*)

apply (*erule asm-rl bij-converse-bij [THEN bij-is-fun, THEN apply-type]*)

apply (*auto simp add: right-inverse-bij*)

done

lemma *ord-iso-reft*: $\text{id}(A): \text{ord-iso}(A, r, A, r)$

by (*rule id-bij [THEN ord-isoI], simp*)

lemma *ord-iso-sym*: $f \in \text{ord-iso}(A, r, B, s) \implies \text{converse}(f): \text{ord-iso}(B, s, A, r)$

apply (*simp add: ord-iso-def*)

apply (*auto simp add: right-inverse-bij bij-converse-bij*

bij-is-fun [THEN apply-funtype])

done

lemma *mono-map-trans*:

$\llbracket g \in \text{mono-map}(A, r, B, s); f \in \text{mono-map}(B, s, C, t) \rrbracket$

$\implies (f \circ g): \text{mono-map}(A, r, C, t)$

unfolding *mono-map-def*

apply (*auto simp add: comp-fun*)

done

lemma *ord-iso-trans*:

$\llbracket g \in \text{ord-iso}(A, r, B, s); f \in \text{ord-iso}(B, s, C, t) \rrbracket$

```

     $\implies (f \circ g): \text{ord-iso}(A, r, C, t)$ 
  apply (unfold ord-iso-def, clarify)
  apply (frule bij-is-fun [of f])
  apply (frule bij-is-fun [of g])
  apply (auto simp add: comp-bij)
done

```

```

lemma mono-ord-isoI:
   $\llbracket f \in \text{mono-map}(A, r, B, s); \ g \in \text{mono-map}(B, s, A, r);$ 
   $f \circ g = \text{id}(B); \ g \circ f = \text{id}(A) \rrbracket \implies f \in \text{ord-iso}(A, r, B, s)$ 
  apply (simp add: ord-iso-def mono-map-def, safe)
  apply (intro fg-imp-bijective, auto)
  apply (subgoal-tac <g' (f'x), g' (f'y) >  $\in r$ )
  apply (simp add: comp-eq-id-iff [THEN iffD1])
  apply (blast intro: apply-funtype)
done

```

```

lemma well-ord-mono-ord-isoI:
   $\llbracket \text{well-ord}(A, r); \ \text{well-ord}(B, s);$ 
   $f \in \text{mono-map}(A, r, B, s); \ \text{converse}(f): \text{mono-map}(B, s, A, r) \rrbracket$ 
   $\implies f \in \text{ord-iso}(A, r, B, s)$ 
  apply (intro mono-ord-isoI, auto)
  apply (frule mono-map-is-fun [THEN fun-is-rel])
  apply (erule converse-converse [THEN subst], rule left-comp-inverse)
  apply (blast intro: left-comp-inverse mono-map-is-inj well-ord-is-linear
    well-ord-is-wf)+
done

```

```

lemma part-ord-ord-iso:
   $\llbracket \text{part-ord}(B, s); \ f \in \text{ord-iso}(A, r, B, s) \rrbracket \implies \text{part-ord}(A, r)$ 
  apply (simp add: part-ord-def irrefl-def trans-on-def ord-iso-def)
  apply (fast intro: bij-is-fun [THEN apply-type])
done

```

```

lemma linear-ord-iso:
   $\llbracket \text{linear}(B, s); \ f \in \text{ord-iso}(A, r, B, s) \rrbracket \implies \text{linear}(A, r)$ 
  apply (simp add: linear-def ord-iso-def, safe)
  apply (drule-tac x1 = f'x and x = f'y in bspec [THEN bspec])
  apply (safe elim!: bij-is-fun [THEN apply-type])
  apply (drule-tac t = (') (converse (f)) in subst-context)
  apply (simp add: left-inverse-bij)
done

```

```

lemma wf-on-ord-iso:

```

```

     $\llbracket wf[B](s); f \in ord\text{-}iso(A, r, B, s) \rrbracket \implies wf[A](r)$ 
apply (simp add: wf-on-def wf-def ord-iso-def, safe)
apply (drule-tac x = {f'z. z ∈ Z ∩ A} in spec)
apply (safe intro!: equalityI)
apply (blast dest!: equalityD1 intro: bij-is-fun [THEN apply-type])+
done

```

```

lemma well-ord-ord-iso:
     $\llbracket well\text{-}ord(B, s); f \in ord\text{-}iso(A, r, B, s) \rrbracket \implies well\text{-}ord(A, r)$ 
    unfolding well-ord-def tot-ord-def
apply (fast elim!: part-ord-ord-iso linear-ord-iso wf-on-ord-iso)
done

```

19.5 Main results of Kunen, Chapter 1 section 6

```

lemma well-ord-iso-subset-lemma:
     $\llbracket well\text{-}ord(A, r); f \in ord\text{-}iso(A, r, A', r); A' \leq A; y \in A \rrbracket$ 
     $\implies \neg \langle f'y, y \rangle: r$ 
apply (simp add: well-ord-def ord-iso-def)
apply (elim conjE CollectE)
apply (rule-tac a=y in wf-on-induct, assumption+)
apply (blast dest: bij-is-fun [THEN apply-type])
done

```

```

lemma well-ord-iso-predE:
     $\llbracket well\text{-}ord(A, r); f \in ord\text{-}iso(A, r, pred(A, x, r), r); x \in A \rrbracket \implies P$ 
apply (insert well-ord-iso-subset-lemma [of A r f pred(A, x, r) x])
apply (simp add: pred-subset)

apply (drule ord-iso-is-bij [THEN bij-is-fun, THEN apply-type], assumption)

apply (simp add: well-ord-def pred-def)
done

```

```

lemma well-ord-iso-pred-eq:
     $\llbracket well\text{-}ord(A, r); f \in ord\text{-}iso(pred(A, a, r), r, pred(A, c, r), r);$ 
     $a \in A; c \in A \rrbracket \implies a=c$ 
apply (frule well-ord-is-trans-on)
apply (frule well-ord-is-linear)
apply (erule-tac x=a and y=c in linearE, assumption+)
apply (drule ord-iso-sym)

apply (auto elim!: well-ord-subset [OF - pred-subset, THEN well-ord-iso-predE]
    intro!: predI
    simp add: trans-pred-pred-eq)
done

```


lemma *ord-iso-image-pred*:
 $\llbracket f \in \text{ord-iso}(A, r, B, s); a \in A \rrbracket \implies f \text{ “ } \text{pred}(A, a, r) = \text{pred}(B, f'a, s)$
unfolding *ord-iso-def pred-def*
apply (*erule CollectE*)
apply (*simp* (*no-asm-simp*) *add: image-fun [OF bij-is-fun Collect-subset]*)
apply (*rule equalityI*)
apply (*safe elim!: bij-is-fun [THEN apply-type]*)
apply (*rule RepFun-eqI*)
apply (*blast intro!: right-inverse-bij [symmetric]*)
apply (*auto simp add: right-inverse-bij bij-is-fun [THEN apply-funtype]*)
done

lemma *ord-iso-restrict-image*:
 $\llbracket f \in \text{ord-iso}(A, r, B, s); C \leq A \rrbracket$
 $\implies \text{restrict}(f, C) \in \text{ord-iso}(C, r, f''C, s)$
apply (*simp add: ord-iso-def*)
apply (*blast intro: bij-is-inj restrict-bij*)
done

lemma *ord-iso-restrict-pred*:
 $\llbracket f \in \text{ord-iso}(A, r, B, s); a \in A \rrbracket$
 $\implies \text{restrict}(f, \text{pred}(A, a, r)) \in \text{ord-iso}(\text{pred}(A, a, r), r, \text{pred}(B, f'a, s), s)$
apply (*simp add: ord-iso-image-pred [symmetric]*)
apply (*blast intro: ord-iso-restrict-image elim: predE*)
done

lemma *well-ord-iso-preserving*:
 $\llbracket \text{well-ord}(A, r); \text{well-ord}(B, s); \langle a, c \rangle: r;$
 $f \in \text{ord-iso}(\text{pred}(A, a, r), r, \text{pred}(B, b, s), s);$
 $g \in \text{ord-iso}(\text{pred}(A, c, r), r, \text{pred}(B, d, s), s);$
 $a \in A; c \in A; b \in B; d \in B \rrbracket \implies \langle b, d \rangle: s$
apply (*frule ord-iso-is-bij [THEN bij-is-fun, THEN apply-type], (erule asm-rl predI predE)+*)
apply (*subgoal-tac b = g'a*)
apply (*simp* (*no-asm-simp*))
apply (*rule well-ord-iso-pred-eq, auto*)
apply (*frule ord-iso-restrict-pred, (erule asm-rl predI)+*)
apply (*simp add: well-ord-is-trans-on trans-pred-pred-eq*)
apply (*erule ord-iso-sym [THEN ord-iso-trans], assumption*)
done

lemma *well-ord-iso-unique-lemma*:
 $\llbracket \text{well-ord}(A, r);$
 $f \in \text{ord-iso}(A, r, B, s); g \in \text{ord-iso}(A, r, B, s); y \in A \rrbracket$
 $\implies \neg \langle g'y, f'y \rangle \in s$

```

apply (frule well-ord-iso-subset-lemma)
apply (rule-tac f = converse (f) and g = g in ord-iso-trans)
apply auto
apply (blast intro: ord-iso-sym)
apply (frule ord-iso-is-bij [of f])
apply (frule ord-iso-is-bij [of g])
apply (frule ord-iso-converse)
apply (blast intro!: bij-converse-bij
      intro: bij-is-fun apply-funtype)+
apply (erule notE)
apply (simp add: left-inverse-bij bij-is-fun comp-fun-apply [of - A B])
done

```

```

lemma well-ord-iso-unique:  $\llbracket \text{well-ord}(A, r);$ 
       $f \in \text{ord-iso}(A, r, B, s); g \in \text{ord-iso}(A, r, B, s) \rrbracket \implies f = g$ 
apply (rule fun-extension)
apply (erule ord-iso-is-bij [THEN bij-is-fun])+
apply (subgoal-tac f'x  $\in B \wedge g'$ x  $\in B \wedge \text{linear}(B, s)$ )
  apply (simp add: linear-def)
  apply (blast dest: well-ord-iso-unique-lemma)
apply (blast intro: ord-iso-is-bij bij-is-fun apply-funtype
      well-ord-is-linear well-ord-ord-iso ord-iso-sym)
done

```

19.6 Towards Kunen's Theorem 6.3: Linearity of the Similarity Relation

```

lemma ord-iso-map-subset:  $\text{ord-iso-map}(A, r, B, s) \subseteq A * B$ 
by (unfold ord-iso-map-def, blast)

```

```

lemma domain-ord-iso-map:  $\text{domain}(\text{ord-iso-map}(A, r, B, s)) \subseteq A$ 
by (unfold ord-iso-map-def, blast)

```

```

lemma range-ord-iso-map:  $\text{range}(\text{ord-iso-map}(A, r, B, s)) \subseteq B$ 
by (unfold ord-iso-map-def, blast)

```

```

lemma converse-ord-iso-map:
   $\text{converse}(\text{ord-iso-map}(A, r, B, s)) = \text{ord-iso-map}(B, s, A, r)$ 
  unfolding ord-iso-map-def
apply (blast intro: ord-iso-sym)
done

```

```

lemma function-ord-iso-map:
   $\text{well-ord}(B, s) \implies \text{function}(\text{ord-iso-map}(A, r, B, s))$ 
  unfolding ord-iso-map-def function-def
apply (blast intro: well-ord-iso-pred-eq ord-iso-sym ord-iso-trans)
done

```

lemma *ord-iso-map-fun*: $well_ord(B,s) \implies ord_iso_map(A,r,B,s)$
 $\in domain(ord_iso_map(A,r,B,s)) \rightarrow range(ord_iso_map(A,r,B,s))$
by (*simp add: Pi-iff function-ord-iso-map*
ord-iso-map-subset [THEN domain-times-range])

lemma *ord-iso-map-mono-map*:
 $\llbracket well_ord(A,r); well_ord(B,s) \rrbracket$
 $\implies ord_iso_map(A,r,B,s)$
 $\in mono_map(domain(ord_iso_map(A,r,B,s)), r,$
 $range(ord_iso_map(A,r,B,s)), s)$
unfolding *mono-map-def*
apply (*simp (no-asm-simp) add: ord-iso-map-fun*)
apply *safe*
apply (*subgoal-tac x ∈ A ∧ ya:A ∧ y ∈ B ∧ yb:B*)
apply (*simp add: apply-equality [OF - ord-iso-map-fun]*)
unfolding *ord-iso-map-def*
apply (*blast intro: well-ord-iso-preserving, blast*)
done

lemma *ord-iso-map-ord-iso*:
 $\llbracket well_ord(A,r); well_ord(B,s) \rrbracket \implies ord_iso_map(A,r,B,s)$
 $\in ord_iso(domain(ord_iso_map(A,r,B,s)), r,$
 $range(ord_iso_map(A,r,B,s)), s)$
apply (*rule well-ord-mono-ord-isoI*)
prefer 4
apply (*rule converse-ord-iso-map [THEN subst]*)
apply (*simp add: ord-iso-map-mono-map*
ord-iso-map-subset [THEN converse-converse])
apply (*blast intro!: domain-ord-iso-map range-ord-iso-map*
intro: well-ord-subset ord-iso-map-mono-map)
done

lemma *domain-ord-iso-map-subset*:
 $\llbracket well_ord(A,r); well_ord(B,s);$
 $a \in A; a \notin domain(ord_iso_map(A,r,B,s)) \rrbracket$
 $\implies domain(ord_iso_map(A,r,B,s)) \subseteq pred(A, a, r)$
unfolding *ord-iso-map-def*
apply (*safe intro!: predI*)

apply (*simp (no-asm-simp)*)
apply (*frule-tac A = A in well-ord-is-linear*)
apply (*rename-tac b y f*)
apply (*erule-tac x=b and y=a in linearE, assumption+*)

apply *clarify*
apply *blast*

```

apply (frule ord-iso-is-bij [THEN bij-is-fun, THEN apply-type],
      (erule asm-rl predI predE)+)
apply (frule ord-iso-restrict-pred)
apply (simp add: pred-iff)
apply (simp split: split-if-asm
      add: well-ord-is-trans-on trans-pred-pred-eq domain-UN domain-Union,
      blast)
done

```

```

lemma domain-ord-iso-map-cases:
  
$$\llbracket \text{well-ord}(A, r); \text{well-ord}(B, s) \rrbracket$$


$$\implies \text{domain}(\text{ord-iso-map}(A, r, B, s)) = A \mid$$


$$(\exists x \in A. \text{domain}(\text{ord-iso-map}(A, r, B, s)) = \text{pred}(A, x, r))$$

apply (frule well-ord-is-wf)
unfolding wf-on-def wf-def
apply (drule-tac  $x = A - \text{domain}(\text{ord-iso-map}(A, r, B, s))$  in spec)
apply safe

```

```

apply (rule domain-ord-iso-map [THEN equalityI])
apply (erule Diff-eq-0-iff [THEN iffD1])

```

```

apply (blast del: domainI subsetI
      elim!: predE
      intro!: domain-ord-iso-map-subset
      intro: subsetI)+
done

```

```

lemma range-ord-iso-map-cases:
  
$$\llbracket \text{well-ord}(A, r); \text{well-ord}(B, s) \rrbracket$$


$$\implies \text{range}(\text{ord-iso-map}(A, r, B, s)) = B \mid$$


$$(\exists y \in B. \text{range}(\text{ord-iso-map}(A, r, B, s)) = \text{pred}(B, y, s))$$

apply (rule converse-ord-iso-map [THEN subst])
apply (simp add: domain-ord-iso-map-cases)
done

```

Kunen's Theorem 6.3: Fundamental Theorem for Well-Ordered Sets

```

theorem well-ord-trichotomy:
  
$$\llbracket \text{well-ord}(A, r); \text{well-ord}(B, s) \rrbracket$$


$$\implies \text{ord-iso-map}(A, r, B, s) \in \text{ord-iso}(A, r, B, s) \mid$$


$$(\exists x \in A. \text{ord-iso-map}(A, r, B, s) \in \text{ord-iso}(\text{pred}(A, x, r), r, B, s)) \mid$$


$$(\exists y \in B. \text{ord-iso-map}(A, r, B, s) \in \text{ord-iso}(A, r, \text{pred}(B, y, s), s))$$

apply (frule-tac  $B = B$  in domain-ord-iso-map-cases, assumption)
apply (frule-tac  $B = B$  in range-ord-iso-map-cases, assumption)
apply (drule ord-iso-map-ord-iso, assumption)
apply (elim disjE bexE)
apply (simp-all add: bexI)

```

```

apply (rule wf-on-not-refl [THEN notE])
  apply (erule well-ord-is-wf)
  apply assumption
apply (subgoal-tac ⟨x,y⟩: ord-iso-map (A,r,B,s) )
  apply (drule rangeI)
  apply (simp add: pred-def)
apply (unfold ord-iso-map-def, blast)
done

```

19.7 Miscellaneous Results by Krzysztof Grabczewski

```

lemma irrefl-converse: irrefl(A,r)  $\implies$  irrefl(A,converse(r))
by (unfold irrefl-def, blast)

```

```

lemma trans-on-converse: trans[A](r)  $\implies$  trans[A](converse(r))
by (unfold trans-on-def, blast)

```

```

lemma part-ord-converse: part-ord(A,r)  $\implies$  part-ord(A,converse(r))
  unfolding part-ord-def
apply (blast intro!: irrefl-converse trans-on-converse)
done

```

```

lemma linear-converse: linear(A,r)  $\implies$  linear(A,converse(r))
by (unfold linear-def, blast)

```

```

lemma tot-ord-converse: tot-ord(A,r)  $\implies$  tot-ord(A,converse(r))
  unfolding tot-ord-def
apply (blast intro!: part-ord-converse linear-converse)
done

```

```

lemma first-is-elem: first(b,B,r)  $\implies$  b  $\in$  B
by (unfold first-def, blast)

```

```

lemma well-ord-imp-ex1-first:
   $\llbracket \text{well-ord}(A,r); B \leq A; B \neq 0 \rrbracket \implies (\exists !b. \text{first}(b,B,r))$ 
  unfolding well-ord-def wf-on-def wf-def first-def
apply (elim conjE allE disjE, blast)
apply (erule bexE)
apply (rule-tac a = x in ex1I, auto)
apply (unfold tot-ord-def linear-def, blast)
done

```

```

lemma the-first-in:
   $\llbracket \text{well-ord}(A,r); B \leq A; B \neq 0 \rrbracket \implies (THE b. \text{first}(b,B,r)) \in B$ 
apply (drule well-ord-imp-ex1-first, assumption+)
apply (rule first-is-elem)

```

apply (erule theI)
done

19.8 Lemmas for the Reflexive Orders

lemma subset-vimage-vimage-iff:

$\llbracket \text{Preorder}(r); A \subseteq \text{field}(r); B \subseteq \text{field}(r) \rrbracket \implies$

$r - " A \subseteq r - " B \longleftrightarrow (\forall a \in A. \exists b \in B. \langle a, b \rangle \in r)$

apply (auto simp: subset-def preorder-on-def refl-def vimage-def image-def)

apply blast

unfolding trans-on-def

apply (erule-tac $P = (\lambda x. \forall y \in \text{field}(r).$

$\forall z \in \text{field}(r). \langle x, y \rangle \in r \longrightarrow \langle y, z \rangle \in r \longrightarrow \langle x, z \rangle \in r)$ for r in rev-ballE)

apply best

apply blast

done

lemma subset-vimage1-vimage1-iff:

$\llbracket \text{Preorder}(r); a \in \text{field}(r); b \in \text{field}(r) \rrbracket \implies$

$r - " \{a\} \subseteq r - " \{b\} \longleftrightarrow \langle a, b \rangle \in r$

by (simp add: subset-vimage-vimage-iff)

lemma Refl-antisym-eq-Image1-Image1-iff:

$\llbracket \text{refl}(\text{field}(r), r); \text{antisym}(r); a \in \text{field}(r); b \in \text{field}(r) \rrbracket \implies$

$r - " \{a\} = r - " \{b\} \longleftrightarrow a = b$

apply rule

apply (frule equality-iffD)

apply (drule equality-iffD)

apply (simp add: antisym-def refl-def)

apply best

apply (simp add: antisym-def refl-def)

done

lemma Partial-order-eq-Image1-Image1-iff:

$\llbracket \text{Partial-order}(r); a \in \text{field}(r); b \in \text{field}(r) \rrbracket \implies$

$r - " \{a\} = r - " \{b\} \longleftrightarrow a = b$

by (simp add: partial-order-on-def preorder-on-def

Refl-antisym-eq-Image1-Image1-iff)

lemma Refl-antisym-eq-vimage1-vimage1-iff:

$\llbracket \text{refl}(\text{field}(r), r); \text{antisym}(r); a \in \text{field}(r); b \in \text{field}(r) \rrbracket \implies$

$r - " \{a\} = r - " \{b\} \longleftrightarrow a = b$

apply rule

apply (frule equality-iffD)

apply (drule equality-iffD)

apply (simp add: antisym-def refl-def)

apply best

apply (simp add: antisym-def refl-def)

done

lemma *Partial-order-eq-vimage1-vimage1-iff*:
 $\llbracket \text{Partial-order}(r); a \in \text{field}(r); b \in \text{field}(r) \rrbracket \implies$
 $r -'' \{a\} = r -'' \{b\} \longleftrightarrow a = b$
by (*simp add: partial-order-on-def preorder-on-def*
Refl-antisym-eq-vimage1-vimage1-iff)

end

20 Combining Orderings: Foundations of Ordinal Arithmetic

theory *OrderArith* **imports** *Order Sum Ordinal* **begin**

definition

radd $:: [i, i, i, i] \Rightarrow i$ **where**
 $\text{radd}(A, r, B, s) \equiv$
 $\{z: (A+B) * (A+B).$
 $\quad (\exists x y. z = \langle \text{Inl}(x), \text{Inr}(y) \rangle) \mid$
 $\quad (\exists x' x. z = \langle \text{Inl}(x'), \text{Inl}(x) \rangle \wedge \langle x', x \rangle : r) \mid$
 $\quad (\exists y' y. z = \langle \text{Inr}(y'), \text{Inr}(y) \rangle \wedge \langle y', y \rangle : s)\}$

definition

rmult $:: [i, i, i, i] \Rightarrow i$ **where**
 $\text{rmult}(A, r, B, s) \equiv$
 $\{z: (A*B) * (A*B).$
 $\quad \exists x' y' x y. z = \langle \langle x', y' \rangle, \langle x, y \rangle \rangle \wedge$
 $\quad (\langle x', x \rangle : r \mid (x' = x \wedge \langle y', y \rangle : s))\}$

definition

rvimage $:: [i, i, i] \Rightarrow i$ **where**
 $\text{rvimage}(A, f, r) \equiv \{z \in A * A. \exists x y. z = \langle x, y \rangle \wedge \langle f'x, f'y \rangle : r\}$

definition

measure $:: [i, i] \Rightarrow i$ **where**
 $\text{measure}(A, f) \equiv \{\langle x, y \rangle : A * A. f(x) < f(y)\}$

20.1 Addition of Relations – Disjoint Sum

20.1.1 Rewrite rules. Can be used to obtain introduction rules

lemma *radd-Inl-Inr-iff [iff]*:
 $\langle \text{Inl}(a), \text{Inr}(b) \rangle \in \text{radd}(A, r, B, s) \longleftrightarrow a \in A \wedge b \in B$
by (*unfold radd-def, blast*)

lemma *radd-Inl-iff* [*iff*]:
 $\langle \text{Inl}(a'), \text{Inl}(a) \rangle \in \text{radd}(A, r, B, s) \longleftrightarrow a':A \wedge a \in A \wedge \langle a', a \rangle : r$
by (*unfold radd-def, blast*)

lemma *radd-Inr-iff* [*iff*]:
 $\langle \text{Inr}(b'), \text{Inr}(b) \rangle \in \text{radd}(A, r, B, s) \longleftrightarrow b':B \wedge b \in B \wedge \langle b', b \rangle : s$
by (*unfold radd-def, blast*)

lemma *radd-Inr-Inl-iff* [*simp*]:
 $\langle \text{Inr}(b), \text{Inl}(a) \rangle \in \text{radd}(A, r, B, s) \longleftrightarrow \text{False}$
by (*unfold radd-def, blast*)

declare *radd-Inr-Inl-iff* [*THEN iffD1, dest!*]

20.1.2 Elimination Rule

lemma *raddE*:
 $\llbracket \langle p', p \rangle \in \text{radd}(A, r, B, s);$
 $\bigwedge x y. \llbracket p' = \text{Inl}(x); x \in A; p = \text{Inr}(y); y \in B \rrbracket \implies Q;$
 $\bigwedge x' x. \llbracket p' = \text{Inl}(x'); p = \text{Inl}(x); \langle x', x \rangle : r; x':A; x \in A \rrbracket \implies Q;$
 $\bigwedge y' y. \llbracket p' = \text{Inr}(y'); p = \text{Inr}(y); \langle y', y \rangle : s; y':B; y \in B \rrbracket \implies Q$
 $\rrbracket \implies Q$
by (*unfold radd-def, blast*)

20.1.3 Type checking

lemma *radd-type*: $\text{radd}(A, r, B, s) \subseteq (A+B) * (A+B)$
unfolding *radd-def*
apply (*rule Collect-subset*)
done

lemmas *field-radd* = *radd-type* [*THEN field-rel-subset*]

20.1.4 Linearity

lemma *linear-radd*:
 $\llbracket \text{linear}(A, r); \text{linear}(B, s) \rrbracket \implies \text{linear}(A+B, \text{radd}(A, r, B, s))$
by (*unfold linear-def, blast*)

20.1.5 Well-foundedness

lemma *wf-on-radd*: $\llbracket \text{wf}[A](r); \text{wf}[B](s) \rrbracket \implies \text{wf}[A+B](\text{radd}(A, r, B, s))$
apply (*rule wf-onI2*)
apply (*subgoal-tac* $\forall x \in A. \text{Inl}(x) \in Ba$)
— Proving the lemma, which is needed twice!
prefer 2
apply (*erule-tac* $V = y \in A + B$ **in** *thin-rl*)
apply (*rule-tac ballI*)
apply (*erule-tac* $r = r$ **and** $a = x$ **in** *wf-on-induct, assumption*)
apply *blast*

Returning to main part of proof

```

apply safe
apply blast
apply (erule-tac  $r = s$  and  $a = ya$  in wf-on-induct, assumption, blast)
done

```

```

lemma wf-radd:  $\llbracket wf(r); wf(s) \rrbracket \implies wf(radd(field(r), r, field(s), s))$ 
apply (simp add: wf-iff-wf-on-field)
apply (rule wf-on-subset-A [OF - field-radd])
apply (blast intro: wf-on-radd)
done

```

```

lemma well-ord-radd:
   $\llbracket well-ord(A, r); well-ord(B, s) \rrbracket \implies well-ord(A+B, radd(A, r, B, s))$ 
apply (rule well-ordI)
apply (simp add: well-ord-def wf-on-radd)
apply (simp add: well-ord-def tot-ord-def linear-radd)
done

```

20.1.6 An *ord-iso* congruence law

```

lemma sum-bij:
   $\llbracket f \in bij(A, C); g \in bij(B, D) \rrbracket$ 
   $\implies (\lambda z \in A+B. case(\lambda x. Inl(f'x), \lambda y. Inr(g'y), z)) \in bij(A+B, C+D)$ 
apply (rule-tac  $d = case(\lambda x. Inl(converse(f)'x), \lambda y. Inr(converse(g)'y))$ 
  in lam-bijective)
apply (typecheck add: bij-is-inj inj-is-fun)
apply (auto simp add: left-inverse-bij right-inverse-bij)
done

```

```

lemma sum-ord-iso-cong:
   $\llbracket f \in ord-iso(A, r, A', r'); g \in ord-iso(B, s, B', s') \rrbracket \implies$ 
   $(\lambda z \in A+B. case(\lambda x. Inl(f'x), \lambda y. Inr(g'y), z))$ 
   $\in ord-iso(A+B, radd(A, r, B, s), A'+B', radd(A', r', B', s'))$ 
  unfolding ord-iso-def
apply (safe intro!: sum-bij)

```

```

apply (auto cong add: conj-cong simp add: bij-is-fun [THEN apply-type])
done

```

```

lemma sum-disjoint-bij:  $A \cap B = 0 \implies$ 
   $(\lambda z \in A+B. case(\lambda x. x, \lambda y. y, z)) \in bij(A+B, A \cup B)$ 
apply (rule-tac  $d = \lambda z. if z \in A then Inl(z) else Inr(z)$  in lam-bijective)
apply auto
done

```

20.1.7 Associativity

lemma *sum-assoc-bij*:

$(\lambda z \in (A+B)+C. \text{case}(\text{case}(\text{Inl}, \lambda y. \text{Inr}(\text{Inl}(y))), \lambda y. \text{Inr}(\text{Inr}(y)), z))$
 $\in \text{bij}((A+B)+C, A+(B+C))$

apply (*rule-tac* $d = \text{case}(\lambda x. \text{Inl}(\text{Inl}(x)), \text{case}(\lambda x. \text{Inl}(\text{Inr}(x)), \text{Inr}))$
in *lam-bijective*)

apply *auto*

done

lemma *sum-assoc-ord-iso*:

$(\lambda z \in (A+B)+C. \text{case}(\text{case}(\text{Inl}, \lambda y. \text{Inr}(\text{Inl}(y))), \lambda y. \text{Inr}(\text{Inr}(y)), z))$
 $\in \text{ord-iso}((A+B)+C, \text{radd}(A+B, \text{radd}(A, r, B, s), C, t),$
 $A+(B+C), \text{radd}(A, r, B+C, \text{radd}(B, s, C, t)))$

by (*rule sum-assoc-bij [THEN ord-isoI], auto*)

20.2 Multiplication of Relations – Lexicographic Product

20.2.1 Rewrite rule. Can be used to obtain introduction rules

lemma *rmult-iff* [*iff*]:

$\langle \langle a', b' \rangle, \langle a, b \rangle \rangle \in \text{rmult}(A, r, B, s) \longleftrightarrow$
 $(\langle a', a \rangle: r \wedge a': A \wedge a \in A \wedge b': B \wedge b \in B) \mid$
 $(\langle b', b \rangle: s \wedge a' = a \wedge a \in A \wedge b': B \wedge b \in B)$

by (*unfold rmult-def, blast*)

lemma *rmultE*:

$\llbracket \langle \langle a', b' \rangle, \langle a, b \rangle \rangle \in \text{rmult}(A, r, B, s);$
 $\llbracket \langle a', a \rangle: r; a': A; a \in A; b': B; b \in B \rrbracket \implies Q;$
 $\llbracket \langle b', b \rangle: s; a \in A; a' = a; b': B; b \in B \rrbracket \implies Q$

$\rrbracket \implies Q$

by *blast*

20.2.2 Type checking

lemma *rmult-type*: $\text{rmult}(A, r, B, s) \subseteq (A*B) * (A*B)$

by (*unfold rmult-def, rule Collect-subset*)

lemmas *field-rmult* = *rmult-type [THEN field-rel-subset]*

20.2.3 Linearity

lemma *linear-rmult*:

$\llbracket \text{linear}(A, r); \text{linear}(B, s) \rrbracket \implies \text{linear}(A*B, \text{rmult}(A, r, B, s))$

by (*simp add: linear-def, blast*)

20.2.4 Well-foundedness

lemma *wf-on-rmult*: $\llbracket \text{wf}[A](r); \text{wf}[B](s) \rrbracket \implies \text{wf}[A*B](\text{rmult}(A, r, B, s))$

apply (*rule wf-onI2*)

```

apply (erule SigmaE)
apply (erule ssubst)
apply (subgoal-tac  $\forall b \in B. \langle x, b \rangle: Ba, \text{blast}$ )
apply (erule-tac  $a = x$  in wf-on-induct, assumption)
apply (rule ballI)
apply (erule-tac  $a = b$  in wf-on-induct, assumption)
apply (best elim!: rmultE bspec [THEN mp])
done

```

```

lemma wf-rmult:  $\llbracket wf(r); wf(s) \rrbracket \implies wf(rmult(field(r), r, field(s), s))$ 
apply (simp add: wf-iff-wf-on-field)
apply (rule wf-on-subset-A [OF - field-rmult])
apply (blast intro: wf-on-rmult)
done

```

```

lemma well-ord-rmult:
   $\llbracket well-ord(A, r); well-ord(B, s) \rrbracket \implies well-ord(A*B, rmult(A, r, B, s))$ 
apply (rule well-ordI)
apply (simp add: well-ord-def wf-on-rmult)
apply (simp add: well-ord-def tot-ord-def linear-rmult)
done

```

20.2.5 An *ord-iso* congruence law

```

lemma prod-bij:
   $\llbracket f \in bij(A, C); g \in bij(B, D) \rrbracket$ 
   $\implies (lam \langle x, y \rangle: A*B. \langle f'x, g'y \rangle) \in bij(A*B, C*D)$ 
apply (rule-tac  $d = \lambda \langle x, y \rangle. \langle converse(f) 'x, converse(g) 'y \rangle$ 
  in lam-bijective)
apply (typecheck add: bij-is-inj inj-is-fun)
apply (auto simp add: left-inverse-bij right-inverse-bij)
done

```

```

lemma prod-ord-iso-cong:
   $\llbracket f \in ord-iso(A, r, A', r'); g \in ord-iso(B, s, B', s') \rrbracket$ 
   $\implies (lam \langle x, y \rangle: A*B. \langle f'x, g'y \rangle)$ 
   $\in ord-iso(A*B, rmult(A, r, B, s), A'*B', rmult(A', r', B', s'))$ 
  unfolding ord-iso-def
apply (safe intro!: prod-bij)
apply (simp-all add: bij-is-fun [THEN apply-type])
apply (blast intro: bij-is-inj [THEN inj-apply-equality])
done

```

```

lemma singleton-prod-bij:  $(\lambda z \in A. \langle x, z \rangle) \in bij(A, \{x\}*A)$ 
by (rule-tac  $d = snd$  in lam-bijective, auto)

```

```

lemma singleton-prod-ord-iso:

```

```

    well-ord( $\{x\}, xr$ )  $\implies$ 
      ( $\lambda z \in A. \langle x, z \rangle \in \text{ord-iso}(A, r, \{x\} * A, \text{rmult}(\{x\}, xr, A, r))$ )
  apply (rule singleton-prod-bij [THEN ord-isoI])
  apply (simp (no-asm-simp))
  apply (blast dest: well-ord-is-wf [THEN wf-on-not-refl])
done

```

```

lemma prod-sum-singleton-bij:
   $a \notin C \implies$ 
    ( $\lambda x \in C * B + D. \text{case}(\lambda x. x, \lambda y. \langle a, y \rangle, x)$ )
     $\in \text{bij}(C * B + D, C * B \cup \{a\} * D)$ 
  apply (rule subst-elem)
  apply (rule id-bij [THEN sum-bij, THEN comp-bij])
  apply (rule singleton-prod-bij)
  apply (rule sum-disjoint-bij, blast)
  apply (simp (no-asm-simp) cong add: case-cong)
  apply (rule comp-lam [THEN trans, symmetric])
  apply (fast elim!: case-type)
  apply (simp (no-asm-simp) add: case-case)
done

```

```

lemma prod-sum-singleton-ord-iso:
   $\llbracket a \in A; \text{well-ord}(A, r) \rrbracket \implies$ 
    ( $\lambda x \in \text{pred}(A, a, r) * B + \text{pred}(B, b, s). \text{case}(\lambda x. x, \lambda y. \langle a, y \rangle, x)$ )
     $\in \text{ord-iso}(\text{pred}(A, a, r) * B + \text{pred}(B, b, s),$ 
       $\text{radd}(A * B, \text{rmult}(A, r, B, s), B, s),$ 
       $\text{pred}(A, a, r) * B \cup \{a\} * \text{pred}(B, b, s), \text{rmult}(A, r, B, s))$ 
  apply (rule prod-sum-singleton-bij [THEN ord-isoI])
  apply (simp (no-asm-simp) add: pred-iff well-ord-is-wf [THEN wf-on-not-refl])
  apply (auto elim!: well-ord-is-wf [THEN wf-on-async] predE)
done

```

20.2.6 Distributive law

```

lemma sum-prod-distrib-bij:
  ( $\text{lam } \langle x, z \rangle: (A + B) * C. \text{case}(\lambda y. \text{Inl}(\langle y, z \rangle), \lambda y. \text{Inr}(\langle y, z \rangle), x)$ )
   $\in \text{bij}((A + B) * C, (A * C) + (B * C))$ 
  by (rule-tac d = case ( $\lambda \langle x, y \rangle. \text{Inl } (x), y$ ),  $\lambda \langle x, y \rangle. \text{Inr } (x), y$ )
    in lam-bijective, auto)

```

```

lemma sum-prod-distrib-ord-iso:
  ( $\text{lam } \langle x, z \rangle: (A + B) * C. \text{case}(\lambda y. \text{Inl}(\langle y, z \rangle), \lambda y. \text{Inr}(\langle y, z \rangle), x)$ )
   $\in \text{ord-iso}((A + B) * C, \text{rmult}(A + B, \text{radd}(A, r, B, s), C, t),$ 
    ( $A * C + B * C, \text{radd}(A * C, \text{rmult}(A, r, C, t), B * C, \text{rmult}(B, s, C, t))$ ))
  by (rule sum-prod-distrib-bij [THEN ord-isoI], auto)

```

20.2.7 Associativity

```

lemma prod-assoc-bij:

```

(*lam* $\langle \langle x, y \rangle, z \rangle : (A * B) * C. \langle x, \langle y, z \rangle \rangle$) $\in \text{bij}((A * B) * C, A * (B * C))$
by (*rule-tac* $d = \lambda \langle x, \langle y, z \rangle \rangle. \langle \langle x, y \rangle, z \rangle$ **in** *lam-bijective, auto*)

lemma *prod-assoc-ord-iso*:
(*lam* $\langle \langle x, y \rangle, z \rangle : (A * B) * C. \langle x, \langle y, z \rangle \rangle$)
 $\in \text{ord-iso}((A * B) * C, \text{rmult}(A * B, \text{rmult}(A, r, B, s), C, t),$
 $A * (B * C), \text{rmult}(A, r, B * C, \text{rmult}(B, s, C, t)))$
by (*rule prod-assoc-bij [THEN ord-isoI], auto*)

20.3 Inverse Image of a Relation

20.3.1 Rewrite rule

lemma *rvimage-iff*: $\langle a, b \rangle \in \text{rvimage}(A, f, r) \iff \langle f'a, f'b \rangle : r \wedge a \in A \wedge b \in A$
by (*unfold rvimage-def, blast*)

20.3.2 Type checking

lemma *rvimage-type*: $\text{rvimage}(A, f, r) \subseteq A * A$
by (*unfold rvimage-def, rule Collect-subset*)

lemmas *field-rvimage = rvimage-type [THEN field-rel-subset]*

lemma *rvimage-converse*: $\text{rvimage}(A, f, \text{converse}(r)) = \text{converse}(\text{rvimage}(A, f, r))$
by (*unfold rvimage-def, blast*)

20.3.3 Partial Ordering Properties

lemma *irrefl-rvimage*:
 $\llbracket f \in \text{inj}(A, B); \text{irrefl}(B, r) \rrbracket \implies \text{irrefl}(A, \text{rvimage}(A, f, r))$
unfolding *irrefl-def rvimage-def*
apply (*blast intro: inj-is-fun [THEN apply-type]*)
done

lemma *trans-on-rvimage*:
 $\llbracket f \in \text{inj}(A, B); \text{trans}[B](r) \rrbracket \implies \text{trans}[A](\text{rvimage}(A, f, r))$
unfolding *trans-on-def rvimage-def*
apply (*blast intro: inj-is-fun [THEN apply-type]*)
done

lemma *part-ord-rvimage*:
 $\llbracket f \in \text{inj}(A, B); \text{part-ord}(B, r) \rrbracket \implies \text{part-ord}(A, \text{rvimage}(A, f, r))$
unfolding *part-ord-def*
apply (*blast intro!: irrefl-rvimage trans-on-rvimage*)
done

20.3.4 Linearity

lemma *linear-rvimage*:
 $\llbracket f \in \text{inj}(A, B); \text{linear}(B, r) \rrbracket \implies \text{linear}(A, \text{rvimage}(A, f, r))$

```

apply (simp add: inj-def linear-def rimage-iff)
apply (blast intro: apply-funtype)
done

```

```

lemma tot-ord-rimage:
   $\llbracket f \in \text{inj}(A,B); \text{tot-ord}(B,r) \rrbracket \implies \text{tot-ord}(A, \text{rimage}(A,f,r))$ 
  unfolding tot-ord-def
apply (blast intro!: part-ord-rimage linear-rimage)
done

```

20.3.5 Well-foundedness

```

lemma wf-rimage [intro!]:  $\text{wf}(r) \implies \text{wf}(\text{rimage}(A,f,r))$ 
apply (simp (no-asm-use) add: rimage-def wf-eq-minimal)
apply clarify
apply (subgoal-tac  $\exists w. w \in \{w: \{f'x. x \in Q\}. \exists x. x \in Q \wedge (f'x = w) \}$ )
  apply (erule allE)
  apply (erule impE)
  apply assumption
  apply blast
apply blast
done

```

But note that the combination of *wf-imp-wf-on* and *wf-rimage* gives $\text{wf}(r) \implies \text{wf}[C](\text{rimage}(A, f, r))$

```

lemma wf-on-rimage:  $\llbracket f \in A \rightarrow B; \text{wf}[B](r) \rrbracket \implies \text{wf}[A](\text{rimage}(A,f,r))$ 
apply (rule wf-onI2)
apply (subgoal-tac  $\forall z \in A. f'z = f'y \longrightarrow z \in Ba$ )
  apply blast
apply (erule-tac  $a = f'y$  in wf-on-induct)
  apply (blast intro!: apply-funtype)
apply (blast intro!: apply-funtype dest!: rimage-iff [THEN iffD1])
done

```

```

lemma well-ord-rimage:
   $\llbracket f \in \text{inj}(A,B); \text{well-ord}(B,r) \rrbracket \implies \text{well-ord}(A, \text{rimage}(A,f,r))$ 
apply (rule well-ordI)
  unfolding well-ord-def tot-ord-def
apply (blast intro!: wf-on-rimage inj-is-fun)
apply (blast intro!: linear-rimage)
done

```

```

lemma ord-iso-rimage:
   $f \in \text{bij}(A,B) \implies f \in \text{ord-iso}(A, \text{rimage}(A,f,s), B, s)$ 
  unfolding ord-iso-def
apply (simp add: rimage-iff)
done

```

lemma *ord-iso-rvimage-eq*:
 $f \in \text{ord-iso}(A, r, B, s) \implies \text{rvimage}(A, f, s) = r \cap A * A$
by (*unfold ord-iso-def rvimage-def, blast*)

20.4 Every well-founded relation is a subset of some inverse image of an ordinal

lemma *wf-rvimage-Ord*: $\text{Ord}(i) \implies \text{wf}(\text{rvimage}(A, f, \text{Memrel}(i)))$
by (*blast intro: wf-rvimage wf-Memrel*)

definition

$\text{wfrank} :: [i, i] \Rightarrow i$ **where**
 $\text{wfrank}(r, a) \equiv \text{wfrec}(r, a, \lambda x f. \bigcup y \in r - \{x\}. \text{succ}(f'y))$

definition

$\text{wftype} :: i \Rightarrow i$ **where**
 $\text{wftype}(r) \equiv \bigcup y \in \text{range}(r). \text{succ}(\text{wfrank}(r, y))$

lemma *wfrank*: $\text{wf}(r) \implies \text{wfrank}(r, a) = (\bigcup y \in r - \{a\}. \text{succ}(\text{wfrank}(r, y)))$
by (*subst wfrank-def [THEN def-wfrec], simp-all*)

lemma *Ord-wfrank*: $\text{wf}(r) \implies \text{Ord}(\text{wfrank}(r, a))$
apply (*rule-tac a=a in wf-induct, assumption*)
apply (*subst wfrank, assumption*)
apply (*rule Ord-succ [THEN Ord-UN], blast*)
done

lemma *wfrank-lt*: $\llbracket \text{wf}(r); \langle a, b \rangle \in r \rrbracket \implies \text{wfrank}(r, a) < \text{wfrank}(r, b)$
apply (*rule-tac a1 = b in wfrank [THEN ssubst], assumption*)
apply (*rule UN-I [THEN ltI]*)
apply (*simp add: Ord-wfrank vimage-iff*)
done

lemma *Ord-wftype*: $\text{wf}(r) \implies \text{Ord}(\text{wftype}(r))$
by (*simp add: wftype-def Ord-wfrank*)

lemma *wftypeI*: $\llbracket \text{wf}(r); x \in \text{field}(r) \rrbracket \implies \text{wfrank}(r, x) \in \text{wftype}(r)$
apply (*simp add: wftype-def*)
apply (*blast intro: wfrank-lt [THEN ltD]*)
done

lemma *wf-imp-subset-rvimage*:

$\llbracket \text{wf}(r); r \subseteq A * A \rrbracket \implies \exists i f. \text{Ord}(i) \wedge r \subseteq \text{rvimage}(A, f, \text{Memrel}(i))$
apply (*rule-tac x=wftype(r) in exI*)
apply (*rule-tac x= $\lambda x \in A. \text{wfrank}(r, x)$ in exI*)
apply (*simp add: Ord-wftype, clarify*)
apply (*frule subsetD, assumption, clarify*)

apply (*simp add: rvimage-iff wfrank-lt [THEN ltD]*)
apply (*blast intro: wftypeI*)
done

theorem *wf-iff-subset-rvimage*:
 $\text{relation}(r) \implies \text{wf}(r) \longleftrightarrow (\exists i f A. \text{Ord}(i) \wedge r \subseteq \text{rvimage}(A, f, \text{Memrel}(i)))$
by (*blast dest!: relation-field-times-field wf-imp-subset-rvimage*
intro: wf-rvimage-Ord [THEN wf-subset])

20.5 Other Results

lemma *wf-times*: $A \cap B = 0 \implies \text{wf}(A * B)$
by (*simp add: wf-def, blast*)

Could also be used to prove *wf-radd*

lemma *wf-Un*:
 $\llbracket \text{range}(r) \cap \text{domain}(s) = 0; \text{wf}(r); \text{wf}(s) \rrbracket \implies \text{wf}(r \cup s)$
apply (*simp add: wf-def, clarify*)
apply (*rule equalityI*)
prefer 2 **apply** *blast*
apply *clarify*
apply (*drule-tac x=Z in spec*)
apply (*drule-tac x=Z \cap domain(s) in spec*)
apply *simp*
apply (*blast intro: elim: equalityE*)
done

20.5.1 The Empty Relation

lemma *wf0*: $\text{wf}(0)$
by (*simp add: wf-def, blast*)

lemma *linear0*: $\text{linear}(0, 0)$
by (*simp add: linear-def*)

lemma *well-ord0*: $\text{well-ord}(0, 0)$
by (*blast intro: wf-imp-wf-on well-ordI wf0 linear0*)

20.5.2 The "measure" relation is useful with wfrec

lemma *measure-eq-rvimage-Memrel*:
 $\text{measure}(A, f) = \text{rvimage}(A, \text{Lambda}(A, f), \text{Memrel}(\text{Collect}(\text{RepFun}(A, f), \text{Ord})))$
apply (*simp (no-asm) add: measure-def rvimage-def Memrel-iff*)
apply (*rule equalityI, auto*)
apply (*auto intro: Ord-in-Ord simp add: lt-def*)
done

lemma *wf-measure [iff]*: $\text{wf}(\text{measure}(A, f))$
by (*simp (no-asm) add: measure-eq-rvimage-Memrel wf-Memrel wf-rvimage*)

lemma *measure-iff* [iff]: $\langle x, y \rangle \in \text{measure}(A, f) \longleftrightarrow x \in A \wedge y \in A \wedge f(x) < f(y)$
by (*simp* (*no-asm*) *add*: *measure-def*)

lemma *linear-measure*:
assumes *Ord**f*: $\bigwedge x. x \in A \implies \text{Ord}(f(x))$
and *inj*: $\bigwedge x y. \llbracket x \in A; y \in A; f(x) = f(y) \rrbracket \implies x=y$
shows *linear*(*A*, *measure*(*A*, *f*))
apply (*auto simp add*: *linear-def*)
apply (*rule-tac* *i=f(x)* **and** *j=f(y)* **in** *Ord-linear-lt*)
apply (*simp-all add*: *Ord**f*)
apply (*blast intro*: *inj*)
done

lemma *wf-on-measure*: *wf*[*B*](*measure*(*A*, *f*))
by (*rule wf-imp-wf-on* [*OF wf-measure*])

lemma *well-ord-measure*:
assumes *Ord**f*: $\bigwedge x. x \in A \implies \text{Ord}(f(x))$
and *inj*: $\bigwedge x y. \llbracket x \in A; y \in A; f(x) = f(y) \rrbracket \implies x=y$
shows *well-ord*(*A*, *measure*(*A*, *f*))
apply (*rule well-ordI*)
apply (*rule wf-on-measure*)
apply (*blast intro*: *linear-measure Ord**f inj*)
done

lemma *measure-type*: *measure*(*A*, *f*) $\subseteq A * A$
by (*auto simp add*: *measure-def*)

20.5.3 Well-foundedness of Unions

lemma *wf-on-Union*:
assumes *wfA*: *wf*[*A*](*r*)
and *wfB*: $\bigwedge a. a \in A \implies \text{wf}[B(a)](s)$
and *ok*: $\bigwedge a u v. \llbracket \langle u, v \rangle \in s; v \in B(a); a \in A \rrbracket$
 $\implies (\exists a' \in A. \langle a', a \rangle \in r \wedge u \in B(a')) \mid u \in B(a)$
shows *wf*[$\bigcup a \in A. B(a)$](*s*)
apply (*rule wf-onI2*)
apply (*erule UN-E*)
apply (*subgoal-tac* $\forall z \in B(a). z \in Ba$, *blast*)
apply (*rule-tac* *a = a* **in** *wf-on-induct* [*OF wfA*], *assumption*)
apply (*rule ballI*)
apply (*rule-tac* *a = z* **in** *wf-on-induct* [*OF wfB*], *assumption*, *assumption*)
apply (*rename-tac* *u*)
apply (*erule-tac* *x=u* **in** *bspec*, *blast*)
apply (*erule mp*, *clarify*)
apply (*frule ok*, *assumption+*, *blast*)
done

20.5.4 Bijections involving Powersets

lemma *Pow-sum-bij*:

$(\lambda Z \in \text{Pow}(A+B). \langle \{x \in A. \text{Inl}(x) \in Z\}, \{y \in B. \text{Inr}(y) \in Z\} \rangle)$
 $\in \text{bij}(\text{Pow}(A+B), \text{Pow}(A) * \text{Pow}(B))$

apply (*rule-tac* $d = \lambda \langle X, Y \rangle. \{ \text{Inl } (x). x \in X \} \cup \{ \text{Inr } (y). y \in Y \}$
in *lam-bijective*)

apply *force+*

done

As a special case, we have $\text{bij}(\text{Pow}(A \times B), A \rightarrow \text{Pow}(B))$

lemma *Pow-Sigma-bij*:

$(\lambda r \in \text{Pow}(\text{Sigma}(A, B)). \lambda x \in A. r \text{ `` } \{x\})$
 $\in \text{bij}(\text{Pow}(\text{Sigma}(A, B)), \prod x \in A. \text{Pow}(B(x)))$

apply (*rule-tac* $d = \lambda f. \bigcup x \in A. \bigcup y \in f'x. \{ \langle x, y \rangle \}$ **in** *lam-bijective*)

apply (*blast intro: lam-type*)

apply (*blast dest: apply-type, simp-all*)

apply *fast*

apply (*rule fun-extension, auto*)

by *blast*

end

21 Order Types and Ordinal Arithmetic

theory *OrderType* **imports** *OrderArith OrdQuant Nat* **begin**

The order type of a well-ordering is the least ordinal isomorphic to it. Ordinal arithmetic is traditionally defined in terms of order types, as it is here. But a definition by transfinite recursion would be much simpler!

definition

ordermap $:: [i, i] \Rightarrow i$ **where**
ordermap(A, r) $\equiv \lambda x \in A. \text{wfrec}[A](r, x, \lambda x f. f \text{ `` } \text{pred}(A, x, r))$

definition

ordertype $:: [i, i] \Rightarrow i$ **where**
ordertype(A, r) $\equiv \text{ordermap}(A, r) \text{ `` } A$

definition

Ord-alt $:: i \Rightarrow o$ **where**
Ord-alt(X) $\equiv \text{well-ord}(X, \text{Memrel}(X)) \wedge (\forall u \in X. u = \text{pred}(X, u, \text{Memrel}(X)))$

definition

ordify $:: i \Rightarrow i$ **where**
ordify(x) $\equiv \text{if } \text{Ord}(x) \text{ then } x \text{ else } 0$

definition

$omult \quad :: [i,i] \Rightarrow i \quad (\text{infixl } \langle ** \rangle \ 70) \text{ where}$
 $i ** j \equiv ordertype(j*i, rmult(j, Memrel(j), i, Memrel(i)))$

definition

$raw-oadd \quad :: [i,i] \Rightarrow i \text{ where}$
 $raw-oadd(i,j) \equiv ordertype(i+j, radd(i, Memrel(i), j, Memrel(j)))$

definition

$oadd \quad :: [i,i] \Rightarrow i \quad (\text{infixl } \langle ++ \rangle \ 65) \text{ where}$
 $i ++ j \equiv raw-oadd(ordify(i), ordify(j))$

definition

$odiff \quad :: [i,i] \Rightarrow i \quad (\text{infixl } \langle -- \rangle \ 65) \text{ where}$
 $i -- j \equiv ordertype(i-j, Memrel(i))$

21.1 Proofs needing the combination of Ordinal.thy and Order.thy

lemma *le-well-ord-Memrel*: $j \leq i \implies well_ord(j, Memrel(i))$
apply (rule well-ordI)
apply (rule wf-Memrel [THEN wf-imp-wf-on])
apply (simp add: ltD lt-Ord linear-def
 $ltI [THEN lt-trans2 [of - j i]]$)
apply (intro ballI Ord-linear)
apply (blast intro: Ord-in-Ord lt-Ord)+
done

lemmas *well-ord-Memrel* = *le-refl* [THEN *le-well-ord-Memrel*]

lemma *lt-pred-Memrel*:

$j < i \implies pred(i, j, Memrel(i)) = j$
apply (simp add: pred-def lt-def)
apply (blast intro: Ord-trans)
done

lemma *pred-Memrel*:

$x \in A \implies pred(A, x, Memrel(A)) = A \cap x$
by (unfold pred-def Memrel-def, blast)

lemma *Ord-iso-implies-eq-lemma*:

$\llbracket j < i; f \in ord_iso(i, Memrel(i), j, Memrel(j)) \rrbracket \implies R$
apply (frule lt-pred-Memrel)
apply (erule ltE)
apply (rule well-ord-Memrel [THEN well-ord-iso-predE, of i f j], auto)

```

unfolding ord-iso-def

apply (simp (no-asm-simp))
apply (blast intro: bij-is-fun [THEN apply-type] Ord-trans)
done

lemma Ord-iso-implies-eq:
  
$$\llbracket \text{Ord}(i); \text{Ord}(j); f \in \text{ord-iso}(i, \text{Memrel}(i), j, \text{Memrel}(j)) \rrbracket$$


$$\implies i=j$$

apply (rule-tac  $i = i$  and  $j = j$  in Ord-linear-lt)
apply (blast intro: ord-iso-sym Ord-iso-implies-eq-lemma)+
done

```

21.2 Ordermap and ordertype

```

lemma ordermap-type:
  
$$\text{ordermap}(A, r) \in A \rightarrow \text{ordertype}(A, r)$$

unfolding ordermap-def ordertype-def
apply (rule lam-type)
apply (rule lamI [THEN imageI], assumption+)
done

```

21.2.1 Unfolding of ordermap

```

lemma ordermap-eq-image:
  
$$\llbracket \text{wf}[A](r); x \in A \rrbracket$$


$$\implies \text{ordermap}(A, r) \text{ ‘ } x = \text{ordermap}(A, r) \text{ ‘ ‘ } \text{pred}(A, x, r)$$

unfolding ordermap-def pred-def
apply (simp (no-asm-simp))
apply (erule wfrec-on [THEN trans], assumption)
apply (simp (no-asm-simp) add: subset-iff image-lam vimage-singleton-iff)
done

```

```

lemma ordermap-pred-unfold:
  
$$\llbracket \text{wf}[A](r); x \in A \rrbracket$$


$$\implies \text{ordermap}(A, r) \text{ ‘ } x = \{ \text{ordermap}(A, r) \text{ ‘ } y \mid y \in \text{pred}(A, x, r) \}$$

by (simp add: ordermap-eq-image pred-subset ordermap-type [THEN image-fun])

```

```

lemmas ordermap-unfold = ordermap-pred-unfold [simplified pred-def]

```

21.2.2 Showing that ordermap, ordertype yield ordinals

```

lemma Ord-ordermap:
  
$$\llbracket \text{well-ord}(A, r); x \in A \rrbracket \implies \text{Ord}(\text{ordermap}(A, r) \text{ ‘ } x)$$

apply (unfold well-ord-def tot-ord-def part-ord-def, safe)
apply (rule-tac  $a=x$  in wf-on-induct, assumption+)
apply (simp (no-asm-simp) add: ordermap-pred-unfold)

```

```

apply (rule OrdI [OF - Ord-is-Transset])
  unfolding pred-def Transset-def
apply (blast intro: trans-onD
      dest!: ordermap-unfold [THEN equalityD1]) +
done

```

```

lemma Ord-ordertype:
   $well\text{-}ord(A, r) \implies Ord(ordertype(A, r))$ 
  unfolding ordertype-def
apply (subst image-fun [OF ordermap-type subset-refl])
apply (rule OrdI [OF - Ord-is-Transset])
prefer 2 apply (blast intro: Ord-ordermap)
  unfolding Transset-def well-ord-def
apply (blast intro: trans-onD
      dest!: ordermap-unfold [THEN equalityD1])
done

```

21.2.3 ordermap preserves the orderings in both directions

```

lemma ordermap-mono:
   $\llbracket \langle w, x \rangle: r; \text{ wf}[A](r); w \in A; x \in A \rrbracket$ 
   $\implies ordermap(A, r) 'w \in ordermap(A, r) 'x$ 
apply (erule-tac  $x1 = x$  in ordermap-unfold [THEN ssubst], assumption, blast)
done

```

```

lemma converse-ordermap-mono:
   $\llbracket ordermap(A, r) 'w \in ordermap(A, r) 'x; well\text{-}ord(A, r); w \in A; x \in A \rrbracket$ 
   $\implies \langle w, x \rangle: r$ 
apply (unfold well-ord-def tot-ord-def, safe)
apply (erule-tac  $x=w$  and  $y=x$  in linearE, assumption+)
apply (blast elim!: mem-not-refl [THEN notE])
apply (blast dest: ordermap-mono intro: mem-asm)
done

```

```

lemma ordermap-surj:  $ordermap(A, r) \in surj(A, ordertype(A, r))$ 
  unfolding ordertype-def
  by (rule surj-image) (rule ordermap-type)

```

```

lemma ordermap-bij:
   $well\text{-}ord(A, r) \implies ordermap(A, r) \in bij(A, ordertype(A, r))$ 
  unfolding well-ord-def tot-ord-def bij-def inj-def
apply (force intro!: ordermap-type ordermap-surj
      elim: linearE dest: ordermap-mono
      simp add: mem-not-refl)
done

```

21.2.4 Isomorphisms involving ordertype

```

lemma ordertype-ord-iso:

```

```

well-ord(A,r)
 $\implies \text{ordemap}(A,r) \in \text{ord-iso}(A,r, \text{ordertype}(A,r), \text{Memrel}(\text{ordertype}(A,r)))$ 
unfolding ord-iso-def
apply (safe elim!: well-ord-is-wf
      intro!: ordemap-type [THEN apply-type] ordemap-mono ordemap-bij)
apply (blast dest!: converse-ordermap-mono)
done

```

```

lemma ordertype-eq:
   $\llbracket f \in \text{ord-iso}(A,r,B,s); \text{well-ord}(B,s) \rrbracket$ 
   $\implies \text{ordertype}(A,r) = \text{ordertype}(B,s)$ 
apply (frule well-ord-ord-iso, assumption)
apply (rule Ord-iso-implies-eq, (erule Ord-ordertype)+)
apply (blast intro: ord-iso-trans ord-iso-sym ordertype-ord-iso)
done

```

```

lemma ordertype-eq-imp-ord-iso:
   $\llbracket \text{ordertype}(A,r) = \text{ordertype}(B,s); \text{well-ord}(A,r); \text{well-ord}(B,s) \rrbracket$ 
   $\implies \exists f. f \in \text{ord-iso}(A,r,B,s)$ 
apply (rule exI)
apply (rule ordertype-ord-iso [THEN ord-iso-trans], assumption)
apply (erule ssubst)
apply (erule ordertype-ord-iso [THEN ord-iso-sym])
done

```

21.2.5 Basic equalities for ordertype

```

lemma le-ordertype-Memrel:  $j \leq i \implies \text{ordertype}(j, \text{Memrel}(i)) = j$ 
apply (rule Ord-iso-implies-eq [symmetric])
apply (erule ltE, assumption)
apply (blast intro: le-well-ord-Memrel Ord-ordertype)
apply (rule ord-iso-trans)
apply (erule-tac [2] le-well-ord-Memrel [THEN ordertype-ord-iso])
apply (rule id-bij [THEN ord-isoI])
apply (simp (no-asm-simp))
apply (fast elim: ltE Ord-in-Ord Ord-trans)
done

```

```

lemmas ordertype-Memrel = le-refl [THEN le-ordertype-Memrel]

```

```

lemma ordertype-0 [simp]:  $\text{ordertype}(0,r) = 0$ 
apply (rule id-bij [THEN ord-isoI, THEN ordertype-eq, THEN trans])
apply (erule emptyE)
apply (rule well-ord-0)
apply (rule Ord-0 [THEN ordertype-Memrel])
done

```

lemmas *bij-ordertype-vimage = ord-iso-rvimage* [THEN *ordertype-eq*]

21.2.6 A fundamental unfolding law for ordertype.

lemma *ordermap-pred-eq-ordermap*:

```

  [[well-ord(A,r); y ∈ A; z ∈ pred(A,y,r)]]
    ⇒ ordermap(pred(A,y,r), r) ‘ z = ordermap(A, r) ‘ z
apply (frule wf-on-subset-A [OF well-ord-is-wf pred-subset])
apply (rule-tac a=z in wf-on-induct, assumption+)
apply (safe elim!: predE)
apply (simp (no-asm-simp) add: ordermap-pred-unfold well-ord-is-wf pred-iff)

apply (simp (no-asm-simp) add: pred-pred-eq)
apply (simp add: pred-def)
apply (rule RepFun-cong [OF - refl])
apply (drule well-ord-is-trans-on)
apply (fast elim!: trans-onD)
done

```

lemma *ordertype-unfold*:

```

  ordertype(A,r) = {ordermap(A,r) ‘ y . y ∈ A}
unfolding ordertype-def
apply (rule image-fun [OF ordermap-type subset-refl])
done

```

Theorems by Krzysztof Grabczewski; proofs simplified by lcp

lemma *ordertype-pred-subset*: [[well-ord(A,r); x ∈ A]] ⇒

```

  ordertype(pred(A,x,r),r) ⊆ ordertype(A,r)
apply (simp add: ordertype-unfold well-ord-subset [OF - pred-subset])
apply (fast intro: ordermap-pred-eq-ordermap elim: predE)
done

```

lemma *ordertype-pred-lt*:

```

  [[well-ord(A,r); x ∈ A]]
    ⇒ ordertype(pred(A,x,r),r) < ordertype(A,r)
apply (rule ordertype-pred-subset [THEN subset-imp-le, THEN leE])
apply (simp-all add: Ord-ordertype well-ord-subset [OF - pred-subset])
apply (erule sym [THEN ordertype-eq-imp-ord-iso, THEN exE])
apply (erule-tac [3] well-ord-iso-predE)
apply (simp-all add: well-ord-subset [OF - pred-subset])
done

```

lemma *ordertype-pred-unfold*:

```

  well-ord(A,r)
    ⇒ ordertype(A,r) = {ordertype(pred(A,x,r),r). x ∈ A}
apply (rule equalityI)
apply (safe intro!: ordertype-pred-lt [THEN ltD])
apply (auto simp add: ordertype-def well-ord-is-wf [THEN ordermap-eq-image])

```

```

ordermap-type [THEN image-fun]
ordermap-pred-eq-ordermap pred-subset)
done

```

21.3 Alternative definition of ordinal

```

lemma Ord-is-Ord-alt: Ord(i)  $\implies$  Ord-alt(i)
  unfolding Ord-alt-def
  apply (rule conjI)
  apply (erule well-ord-Memrel)
  apply (unfold Ord-def Transset-def pred-def Memrel-def, blast)
done

```

```

lemma Ord-alt-is-Ord:
  Ord-alt(i)  $\implies$  Ord(i)
  apply (unfold Ord-alt-def Ord-def Transset-def well-ord-def
    tot-ord-def part-ord-def trans-on-def)
  apply (simp add: pred-Memrel)
  apply (blast elim!: equalityE)
done

```

21.4 Ordinal Addition

21.4.1 Order Type calculations for radd

Addition with 0

```

lemma bij-sum-0: ( $\lambda z \in A+0. \text{case}(\lambda x. x, \lambda y. y, z)$ )  $\in$  bij( $A+0, A$ )
  apply (rule-tac d = Inl in lam-bijective, safe)
  apply (simp-all (no-asm-simp))
done

```

```

lemma ordertype-sum-0-eq:
  well-ord( $A, r$ )  $\implies$  ordertype( $A+0, \text{radd}(A, r, 0, s)$ ) = ordertype( $A, r$ )
  apply (rule bij-sum-0 [THEN ord-isoI, THEN ordertype-eq])
  prefer 2 apply assumption
  apply force
done

```

```

lemma bij-0-sum: ( $\lambda z \in 0+A. \text{case}(\lambda x. x, \lambda y. y, z)$ )  $\in$  bij( $0+A, A$ )
  apply (rule-tac d = Inr in lam-bijective, safe)
  apply (simp-all (no-asm-simp))
done

```

```

lemma ordertype-0-sum-eq:
  well-ord( $A, r$ )  $\implies$  ordertype( $0+A, \text{radd}(0, s, A, r)$ ) = ordertype( $A, r$ )
  apply (rule bij-0-sum [THEN ord-isoI, THEN ordertype-eq])
  prefer 2 apply assumption
  apply force

```


done

Initial segments of radd. Statements by Grabczewski

lemma *pred-Inl-bij*:

$a \in A \implies (\lambda x \in \text{pred}(A, a, r). \text{Inl}(x))$
 $\in \text{bij}(\text{pred}(A, a, r), \text{pred}(A+B, \text{Inl}(a), \text{radd}(A, r, B, s)))$

unfolding *pred-def*

apply (*rule-tac* $d = \text{case } (\lambda x. x, \lambda y. y) \text{ in lam-bijective}$)

apply *auto*

done

lemma *ordertype-pred-Inl-eq*:

$\llbracket a \in A; \text{well-ord}(A, r) \rrbracket$
 $\implies \text{ordertype}(\text{pred}(A+B, \text{Inl}(a), \text{radd}(A, r, B, s)), \text{radd}(A, r, B, s)) =$
 $\text{ordertype}(\text{pred}(A, a, r), r)$

apply (*rule* *pred-Inl-bij* [*THEN* *ord-isoI*, *THEN* *ord-iso-sym*, *THEN* *ordertype-eq*])

apply (*simp-all* *add: well-ord-subset* [*OF* - *pred-subset*])

apply (*simp add: pred-def*)

done

lemma *pred-Inr-bij*:

$b \in B \implies$
 $\text{id}(A+\text{pred}(B, b, s))$
 $\in \text{bij}(A+\text{pred}(B, b, s), \text{pred}(A+B, \text{Inr}(b), \text{radd}(A, r, B, s)))$

unfolding *pred-def id-def*

apply (*rule-tac* $d = \lambda z. z \text{ in lam-bijective, auto}$)

done

lemma *ordertype-pred-Inr-eq*:

$\llbracket b \in B; \text{well-ord}(A, r); \text{well-ord}(B, s) \rrbracket$
 $\implies \text{ordertype}(\text{pred}(A+B, \text{Inr}(b), \text{radd}(A, r, B, s)), \text{radd}(A, r, B, s)) =$
 $\text{ordertype}(A+\text{pred}(B, b, s), \text{radd}(A, r, \text{pred}(B, b, s), s))$

apply (*rule* *pred-Inr-bij* [*THEN* *ord-isoI*, *THEN* *ord-iso-sym*, *THEN* *ordertype-eq*])

prefer 2 **apply** (*force simp add: pred-def id-def, assumption*)

apply (*blast intro: well-ord-radd well-ord-subset* [*OF* - *pred-subset*])

done

21.4.2 ordify: trivial coercion to an ordinal

lemma *Ord-ordify* [*iff*, *TC*]: $\text{Ord}(\text{ordify}(x))$

by (*simp add: ordify-def*)

lemma *ordify-idem* [*simp*]: $\text{ordify}(\text{ordify}(x)) = \text{ordify}(x)$

by (*simp add: ordify-def*)

21.4.3 Basic laws for ordinal addition

lemma *Ord-raw-oadd*: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \text{Ord}(\text{raw-oadd}(i, j))$

by (*simp add: raw-oadd-def ordify-def Ord-ordertype well-ord-radd*)

well-ord-Memrel)

lemma *Ord-oadd* [*iff*, *TC*]: *Ord*(*i*++*j*)
by (*simp add: oadd-def Ord-raw-oadd*)

Ordinal addition with zero

lemma *raw-oadd-0*: *Ord*(*i*) \implies *raw-oadd*(*i*, 0) = *i*
by (*simp add: raw-oadd-def ordify-def ordertype-sum-0-eq*
ordertype-Memrel well-ord-Memrel)

lemma *oadd-0* [*simp*]: *Ord*(*i*) \implies *i*++0 = *i*
apply (*simp (no-asm-simp) add: oadd-def raw-oadd-0 ordify-def*)
done

lemma *raw-oadd-0-left*: *Ord*(*i*) \implies *raw-oadd*(0, *i*) = *i*
by (*simp add: raw-oadd-def ordify-def ordertype-0-sum-eq ordertype-Memrel*
well-ord-Memrel)

lemma *oadd-0-left* [*simp*]: *Ord*(*i*) \implies 0++*i* = *i*
by (*simp add: oadd-def raw-oadd-0-left ordify-def*)

lemma *oadd-eq-if-raw-oadd*:
i++*j* = (*if* *Ord*(*i*) *then* (*if* *Ord*(*j*) *then* *raw-oadd*(*i*, *j*) *else* *i*)
else (*if* *Ord*(*j*) *then* *j* *else* 0))
by (*simp add: oadd-def ordify-def raw-oadd-0-left raw-oadd-0*)

lemma *raw-oadd-eq-oadd*: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \text{raw-oadd}(i, j) = i++j$
by (*simp add: oadd-def ordify-def*)

lemma *lt-oadd1*: $k < i \implies k < i++j$
apply (*simp add: oadd-def ordify-def lt-Ord2 raw-oadd-0, clarify*)
apply (*simp add: raw-oadd-def*)
apply (*rule ltE, assumption*)
apply (*rule ltI*)
apply (*force simp add: ordertype-pred-unfold well-ord-radd well-ord-Memrel*
ordertype-pred-Inl-eq lt-pred-Memrel leI [THEN le-ordertype-Memrel])
apply (*blast intro: Ord-ordertype well-ord-radd well-ord-Memrel*)
done

lemma *oadd-le-self*: *Ord*(*i*) \implies *i* \leq *i*++*j*
apply (*rule all-lt-imp-le*)
apply (*auto simp add: Ord-oadd lt-oadd1*)
done

Various other results

lemma *id-ord-iso-Memrel*: $A \leq B \implies id(A) \in ord\text{-}iso(A, Memrel(A), A, Memrel(B))$
apply (*rule id-bij* [*THEN ord-isoI*])
apply (*simp* (*no-asm-simp*))
apply *blast*
done

lemma *subset-ord-iso-Memrel*:
 $\llbracket f \in ord\text{-}iso(A, Memrel(B), C, r); A \leq B \rrbracket \implies f \in ord\text{-}iso(A, Memrel(A), C, r)$
apply (*frule ord-iso-is-bij* [*THEN bij-is-fun*, *THEN fun-is-rel*])
apply (*frule ord-iso-trans* [*OF id-ord-iso-Memrel*], *assumption*)
apply (*simp add: right-comp-id*)
done

lemma *restrict-ord-iso*:
 $\llbracket f \in ord\text{-}iso(i, Memrel(i), Order.pred(A, a, r), r); a \in A; j < i; trans[A](r) \rrbracket$
 $\implies restrict(f, j) \in ord\text{-}iso(j, Memrel(j), Order.pred(A, f'j, r), r)$
apply (*frule ltD*)
apply (*frule ord-iso-is-bij* [*THEN bij-is-fun*, *THEN apply-type*], *assumption*)
apply (*frule ord-iso-restrict-pred*, *assumption*)
apply (*simp add: pred-iff trans-pred-pred-eq lt-pred-Memrel*)
apply (*blast intro!: subset-ord-iso-Memrel le-imp-subset* [*OF leI*])
done

lemma *restrict-ord-iso2*:
 $\llbracket f \in ord\text{-}iso(Order.pred(A, a, r), r, i, Memrel(i)); a \in A; j < i; trans[A](r) \rrbracket$
 $\implies converse(restrict(converse(f), j)) \in ord\text{-}iso(Order.pred(A, converse(f)'j, r), r, j, Memrel(j))$
by (*blast intro: restrict-ord-iso ord-iso-sym ltI*)

lemma *ordertype-sum-Memrel*:
 $\llbracket well\text{-}ord(A, r); k < j \rrbracket$
 $\implies ordertype(A+k, radd(A, r, k, Memrel(j))) = ordertype(A+k, radd(A, r, k, Memrel(k)))$
apply (*erule ltE*)
apply (*rule ord-iso-reft* [*THEN sum-ord-iso-cong*, *THEN ordertype-eq*])
apply (*erule OrdmemD* [*THEN id-ord-iso-Memrel*, *THEN ord-iso-sym*])
apply (*simp-all add: well-ord-radd well-ord-Memrel*)
done

lemma *oadd-lt-mono2*: $k < j \implies i++k < i++j$
apply (*simp add: oadd-def ordify-def raw-oadd-0-left lt-Ord lt-Ord2*, *clarify*)
apply (*simp add: raw-oadd-def*)
apply (*rule ltE*, *assumption*)
apply (*rule ordertype-pred-unfold* [*THEN equalityD2*, *THEN subsetD*, *THEN ltI*])
apply (*simp-all add: Ord-ordertype well-ord-radd well-ord-Memrel*)
apply (*rule beI*)

```

apply (erule-tac [2] InrI)
apply (simp add: ordertype-pred-Inr-eq well-ord-Memrel lt-pred-Memrel
              leI [THEN le-ordertype-Memrel] ordertype-sum-Memrel)
done

lemma oadd-lt-cancel2:  $\llbracket i++j < i++k; \text{Ord}(j) \rrbracket \implies j < k$ 
apply (simp (asm-lr) add: oadd-eq-if-raw-oadd split: split-if-asm)
prefer 2
apply (frule-tac  $i = i$  and  $j = j$  in oadd-le-self)
apply (simp (asm-lr) add: oadd-def ordify-def lt-Ord not-lt-iff-le [THEN iff-sym])
apply (rule Ord-linear-lt, auto)
apply (simp-all add: raw-oadd-eq-oadd)
apply (blast dest: oadd-lt-mono2 elim: lt-irrefl lt-asymp)+
done

lemma oadd-lt-iff2:  $\text{Ord}(j) \implies i++j < i++k \longleftrightarrow j < k$ 
by (blast intro!: oadd-lt-mono2 dest!: oadd-lt-cancel2)

lemma oadd-inject:  $\llbracket i++j = i++k; \text{Ord}(j); \text{Ord}(k) \rrbracket \implies j = k$ 
apply (simp add: oadd-eq-if-raw-oadd split: split-if-asm)
apply (simp add: raw-oadd-eq-oadd)
apply (rule Ord-linear-lt, auto)
apply (force dest: oadd-lt-mono2 [of concl: i] simp add: lt-not-refl)+
done

lemma lt-oadd-disj:  $k < i++j \implies k < i \mid (\exists l \in j. k = i++l)$ 
apply (simp add: Ord-in-Ord' [of - j] oadd-eq-if-raw-oadd
              split: split-if-asm)
prefer 2
apply (simp add: Ord-in-Ord' [of - j] lt-def)
apply (simp add: ordertype-pred-unfold well-ord-radd well-ord-Memrel raw-oadd-def)
apply (erule ltD [THEN RepFunE])
apply (force simp add: ordertype-pred-Inl-eq well-ord-Memrel ltI
              lt-pred-Memrel le-ordertype-Memrel leI
              ordertype-pred-Inr-eq ordertype-sum-Memrel)
done

```

21.4.4 Ordinal addition with successor – via associativity!

```

lemma oadd-assoc:  $(i++j)++k = i++(j++k)$ 
apply (simp add: oadd-eq-if-raw-oadd Ord-raw-oadd raw-oadd-0 raw-oadd-0-left,
              clarify)
apply (simp add: raw-oadd-def)
apply (rule ordertype-eq [THEN trans])
apply (rule sum-ord-iso-cong [OF ordertype-ord-iso [THEN ord-iso-sym]
                              ord-iso-refl])
apply (simp-all add: Ord-ordertype well-ord-radd well-ord-Memrel)
apply (rule sum-assoc-ord-iso [THEN ordertype-eq, THEN trans])
apply (rule-tac [2] ordertype-eq)

```

apply (*rule-tac* [2] *sum-ord-iso-cong* [*OF ord-iso-refl ordertype-ord-iso*])
apply (*blast intro*: *Ord-ordertype well-ord-radd well-ord-Memrel*) +
done

lemma *oadd-unfold*: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies i++j = i \cup (\bigcup_{k \in j}. \{i++k\})$
apply (*rule subsetI* [*THEN equalityI*])
apply (*erule ltI* [*THEN lt-oadd-disj, THEN disjE*])
apply (*blast intro*: *Ord-oadd*)
apply (*blast elim!*: *ltE, blast*)
apply (*force intro*: *lt-oadd1 oadd-lt-mono2 simp add: Ord-mem-iff-lt*)
done

lemma *oadd-1*: $\text{Ord}(i) \implies i++1 = \text{succ}(i)$
apply (*simp* (*no-asm-simp*) *add: oadd-unfold Ord-1 oadd-0*)
apply *blast*
done

lemma *oadd-succ* [*simp*]: $\text{Ord}(j) \implies i++\text{succ}(j) = \text{succ}(i++j)$
apply (*simp add: oadd-eq-if-raw-oadd, clarify*)
apply (*simp add: raw-oadd-eq-oadd*)
apply (*simp add: oadd-1* [*of j, symmetric*] *oadd-1* [*of i++j, symmetric*]
oadd-assoc)
done

Ordinal addition with limit ordinals

lemma *oadd-UN*:
 $\llbracket \bigwedge x. x \in A \implies \text{Ord}(j(x)); a \in A \rrbracket$
 $\implies i++(\bigcup_{x \in A}. j(x)) = (\bigcup_{x \in A}. i++j(x))$
by (*blast intro*: *ltI Ord-UN Ord-oadd lt-oadd1* [*THEN ltD*]
oadd-lt-mono2 [*THEN ltD*]
elim!: *ltE dest!*: *ltI* [*THEN lt-oadd-disj*])

lemma *oadd-Limit*: $\text{Limit}(j) \implies i++j = (\bigcup_{k \in j}. i++k)$
apply (*frule Limit-has-0* [*THEN ltD*])
apply (*simp add: Limit-is-Ord* [*THEN Ord-in-Ord*] *oadd-UN* [*symmetric*]
Union-eq-UN [*symmetric*] *Limit-Union-eq*)
done

lemma *oadd-eq-0-iff*: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies (i++j) = 0 \longleftrightarrow i=0 \wedge j=0$
apply (*erule trans-induct3* [*of j*])
apply (*simp-all add: oadd-Limit*)
apply (*simp add: Union-empty-iff Limit-def lt-def, blast*)
done

lemma *oadd-eq-lt-iff*: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies 0 < (i++j) \longleftrightarrow 0 < i \mid 0 < j$
by (*simp add: Ord-0-lt-iff* [*symmetric*] *oadd-eq-0-iff*)

lemma *oadd-LimitI*: $\llbracket \text{Ord}(i); \text{Limit}(j) \rrbracket \implies \text{Limit}(i++j)$
apply (*simp add: oadd-Limit*)

```

apply (frule Limit-has-1 [THEN ltD])
apply (rule increasing-LimitI)
apply (rule Ord-0-lt)
  apply (blast intro: Ord-in-Ord [OF Limit-is-Ord])
apply (force simp add: Union-empty-iff oadd-eq-0-iff
          Limit-is-Ord [of j, THEN Ord-in-Ord], auto)
apply (rule-tac x=succ(y) in beaI)
apply (simp add: ltI Limit-is-Ord [of j, THEN Ord-in-Ord])
apply (simp add: Limit-def lt-def)
done

```

Order/monotonicity properties of ordinal addition

```

lemma oadd-le-self2:  $Ord(i) \implies i \leq j++i$ 
proof (induct i rule: trans-induct3)
  case 0 thus ?case by (simp add: Ord-0-le)
next
  case (succ i) thus ?case by (simp add: oadd-succ succ-leI)
next
  case (limit l)
  hence  $l = (\bigcup x \in l. x)$ 
  by (simp add: Union-eq-UN [symmetric] Limit-Union-eq)
  also have  $\dots \leq (\bigcup x \in l. j++x)$ 
  by (rule le-implies-UN-le-UN) (rule limit.hyps)
  finally have  $l \leq (\bigcup x \in l. j++x)$  .
  thus ?case using limit.hyps by (simp add: oadd-Limit)
qed

```

```

lemma oadd-le-mono1:  $k \leq j \implies k++i \leq j++i$ 
apply (frule lt-Ord)
apply (frule le-Ord2)
apply (simp add: oadd-eq-if-raw-oadd, clarify)
apply (simp add: raw-oadd-eq-oadd)
apply (erule-tac i = i in trans-induct3)
apply (simp (no-asm-simp))
apply (simp (no-asm-simp) add: oadd-succ succ-le-iff)
apply (simp (no-asm-simp) add: oadd-Limit)
apply (rule le-implies-UN-le-UN, blast)
done

```

```

lemma oadd-lt-mono:  $\llbracket i' \leq i; j' < j \rrbracket \implies i'++j' < i++j$ 
by (blast intro: lt-trans1 oadd-le-mono1 oadd-lt-mono2 Ord-succD elim: ltE)

```

```

lemma oadd-le-mono:  $\llbracket i' \leq i; j' \leq j \rrbracket \implies i'++j' \leq i++j$ 
by (simp del: oadd-succ add: oadd-succ [symmetric] le-Ord2 oadd-lt-mono)

```

```

lemma oadd-le-iff2:  $\llbracket Ord(j); Ord(k) \rrbracket \implies i++j \leq i++k \longleftrightarrow j \leq k$ 
by (simp del: oadd-succ add: oadd-lt-iff2 oadd-succ [symmetric] Ord-succ)

```

```

lemma oadd-lt-self:  $\llbracket Ord(i); 0 < j \rrbracket \implies i < i++j$ 

```

```

apply (rule lt-trans2)
apply (erule le-refl)
apply (simp only: lt-Ord2 oadd-1 [of i, symmetric])
apply (blast intro: succ-leI oadd-le-mono)
done

```

Every ordinal is exceeded by some limit ordinal.

```

lemma Ord-imp-greater-Limit:  $\text{Ord}(i) \implies \exists k. i < k \wedge \text{Limit}(k)$ 
apply (rule-tac  $x=i++\text{nat}$  in exI)
apply (blast intro: oadd-LimitI oadd-lt-self Limit-nat [THEN Limit-has-0])
done

```

```

lemma Ord2-imp-greater-Limit:  $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \exists k. i < k \wedge j < k \wedge \text{Limit}(k)$ 
apply (insert Ord-Un [of i j, THEN Ord-imp-greater-Limit])
apply (simp add: Un-least-lt-iff)
done

```

21.5 Ordinal Subtraction

The difference is $\text{ordertype}(j - i, \text{Memrel}(j))$. It's probably simpler to define the difference recursively!

```

lemma bij-sum-Diff:
   $A \leq B \implies (\lambda y \in B. \text{if}(y \in A, \text{Inl}(y), \text{Inr}(y))) \in \text{bij}(B, A + (B - A))$ 
apply (rule-tac  $d = \text{case } (\lambda x. x, \lambda y. y)$  in lam-bijective)
apply (blast intro!: if-type)
apply (fast intro!: case-type)
apply (erule-tac [2] sumE)
apply (simp-all (no-asm-simp))
done

lemma ordertype-sum-Diff:
   $i \leq j \implies$ 
     $\text{ordertype}(i + (j - i), \text{radd}(i, \text{Memrel}(j), j - i, \text{Memrel}(j))) =$ 
     $\text{ordertype}(j, \text{Memrel}(j))$ 
apply (safe dest!: le-subset-iff [THEN iffD1])
apply (rule bij-sum-Diff [THEN ord-isoI, THEN ord-iso-sym, THEN ordertype-eq])
apply (erule-tac [3] well-ord-Memrel, assumption)
apply (simp (no-asm-simp))
apply (frule-tac  $j = y$  in Ord-in-Ord, assumption)
apply (frule-tac  $j = x$  in Ord-in-Ord, assumption)
apply (simp (no-asm-simp) add: Ord-mem-iff-lt lt-Ord not-lt-iff-le)
apply (blast intro: lt-trans2 lt-trans)
done

```

```

lemma Ord-odiff [simp, TC]:
   $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \text{Ord}(i - j)$ 
  unfolding odiff-def
apply (blast intro: Ord-ordertype Diff-subset well-ord-subset well-ord-Memrel)

```

done

lemma *raw-oadd-ordertype-Diff*:

$i \leq j$
 $\implies \text{raw-oadd}(i, j - i) = \text{ordertype}(i + (j - i), \text{radd}(i, \text{Memrel}(j), j - i, \text{Memrel}(j)))$
apply (*simp add: raw-oadd-def odiff-def*)
apply (*safe dest!: le-subset-iff [THEN iffD1]*)
apply (*rule sum-ord-iso-cong [THEN ordertype-eq]*)
apply (*erule id-ord-iso-Memrel*)
apply (*rule ordertype-ord-iso [THEN ord-iso-sym]*)
apply (*blast intro: well-ord-radd Diff-subset well-ord-subset well-ord-Memrel*) +
done

lemma *oadd-odiff-inverse*: $i \leq j \implies i ++ (j - i) = j$

by (*simp add: lt-Ord le-Ord2 oadd-def ordify-def raw-oadd-ordertype-Diff*
ordertype-sum-Diff ordertype-Memrel lt-Ord2 [THEN Ord-succD])

lemma *odiff-oadd-inverse*: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies (i ++ j) - i = j$

apply (*rule oadd-inject*)
apply (*blast intro: oadd-odiff-inverse oadd-le-self*)
apply (*blast intro: Ord-ordertype Ord-oadd Ord-odiff*) +
done

lemma *odiff-lt-mono2*: $\llbracket i < j; k \leq i \rrbracket \implies i - k < j - k$

apply (*rule-tac i = k in oadd-lt-cancel2*)
apply (*simp add: oadd-odiff-inverse*)
apply (*subst oadd-odiff-inverse*)
apply (*blast intro: le-trans leI, assumption*)
apply (*simp (no-asm-simp) add: lt-Ord le-Ord2*)
done

21.6 Ordinal Multiplication

lemma *Ord-omult* [*simp, TC*]:

$\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \text{Ord}(i ** j)$

unfolding *omult-def*

apply (*blast intro: Ord-ordertype well-ord-rmult well-ord-Memrel*)
done

21.6.1 A useful unfolding law

lemma *pred-Pair-eq*:

$\llbracket a \in A; b \in B \rrbracket \implies \text{pred}(A * B, \langle a, b \rangle, \text{rmult}(A, r, B, s)) =$
 $\text{pred}(A, a, r) * B \cup (\{a\} * \text{pred}(B, b, s))$

apply (*unfold pred-def, blast*)

done

lemma *ordertype-pred-Pair-eq*:


```

    
$$\llbracket a \in A; b \in B; \text{well-ord}(A,r); \text{well-ord}(B,s) \rrbracket \implies$$


$$\text{ordertype}(\text{pred}(A*B, \langle a,b \rangle, \text{rmult}(A,r,B,s)), \text{rmult}(A,r,B,s)) =$$


$$\text{ordertype}(\text{pred}(A,a,r)*B + \text{pred}(B,b,s),$$


$$\text{radd}(A*B, \text{rmult}(A,r,B,s), B, s))$$

apply (simp (no-asm-simp) add: pred-Pair-eq)
apply (rule ordertype-eq [symmetric])
apply (rule prod-sum-singleton-ord-iso)
apply (simp-all add: pred-subset well-ord-rmult [THEN well-ord-subset])
apply (blast intro: pred-subset well-ord-rmult [THEN well-ord-subset]
    elim!: predE)
done

lemma ordertype-pred-Pair-lemma:

$$\llbracket i' < i; j' < j \rrbracket$$


$$\implies \text{ordertype}(\text{pred}(i*j, \langle i',j' \rangle, \text{rmult}(i, \text{Memrel}(i), j, \text{Memrel}(j))),$$


$$\text{rmult}(i, \text{Memrel}(i), j, \text{Memrel}(j))) =$$


$$\text{raw-oadd}(j**i', j')$$

unfolding raw-oadd-def omult-def
apply (simp add: ordertype-pred-Pair-eq lt-pred-Memrel ltD lt-Ord2
    well-ord-Memrel)
apply (rule trans)
apply (rule-tac [2] ordertype-ord-iso
    [THEN sum-ord-iso-cong, THEN ordertype-eq])
apply (rule-tac [3] ord-iso-refl)
apply (rule id-bij [THEN ord-isoI, THEN ordertype-eq])
apply (elim SigmaE sumE ltE ssubst)
apply (simp-all add: well-ord-rmult well-ord-radd well-ord-Memrel
    Ord-ordertype lt-Ord lt-Ord2)
apply (blast intro: Ord-trans) +
done

lemma lt-omult:

$$\llbracket \text{Ord}(i); \text{Ord}(j); k < j**i \rrbracket$$


$$\implies \exists j' i'. k = j**i' ++ j' \wedge j' < j \wedge i' < i$$

unfolding omult-def
apply (simp add: ordertype-pred-unfold well-ord-rmult well-ord-Memrel)
apply (safe elim!: ltE)
apply (simp add: ordertype-pred-Pair-lemma ltI raw-oadd-eq-oadd
    omult-def [symmetric] Ord-in-Ord' [of - i] Ord-in-Ord' [of - j])
apply (blast intro: ltI)
done

lemma omult-oadd-lt:

$$\llbracket j' < j; i' < i \rrbracket \implies j**i' ++ j' < j**i$$

unfolding omult-def
apply (rule ltI)
prefer 2
apply (simp add: Ord-ordertype well-ord-rmult well-ord-Memrel lt-Ord2)
apply (simp add: ordertype-pred-unfold well-ord-rmult well-ord-Memrel lt-Ord2)

```

```

apply (rule bexI [of - i'])
apply (rule bexI [of - j'])
apply (simp add: ordertype-pred-Pair-lemma ltI omult-def [symmetric])
apply (simp add: lt-Ord lt-Ord2 raw-oadd-eq-oadd)
apply (simp-all add: lt-def)
done

```

```

lemma omult-unfold:
   $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies j ** i = (\bigcup j' \in j. \bigcup i' \in i. \{j ** i' ++ j'\})$ 
apply (rule subsetI [THEN equalityI])
apply (rule lt-omult [THEN exE])
apply (erule-tac [3] ltI)
apply (simp-all add: Ord-omult)
apply (blast elim!: ltE)
apply (blast intro: omult-oadd-lt [THEN ltD] ltI)
done

```

21.6.2 Basic laws for ordinal multiplication

Ordinal multiplication by zero

```

lemma omult-0 [simp]:  $i ** 0 = 0$ 
  unfolding omult-def
apply (simp (no-asm-simp))
done

```

```

lemma omult-0-left [simp]:  $0 ** i = 0$ 
  unfolding omult-def
apply (simp (no-asm-simp))
done

```

Ordinal multiplication by 1

```

lemma omult-1 [simp]:  $\text{Ord}(i) \implies i ** 1 = i$ 
  unfolding omult-def
apply (rule-tac s1=Memrel(i)
    in ord-isoI [THEN ordertype-eq, THEN trans])
apply (rule-tac c = snd and d =  $\lambda z. \langle 0, z \rangle$  in lam-bijective)
apply (auto elim!: snd-type well-ord-Memrel ordertype-Memrel)
done

```

```

lemma omult-1-left [simp]:  $\text{Ord}(i) \implies 1 ** i = i$ 
  unfolding omult-def
apply (rule-tac s1=Memrel(i)
    in ord-isoI [THEN ordertype-eq, THEN trans])
apply (rule-tac c = fst and d =  $\lambda z. \langle z, 0 \rangle$  in lam-bijective)
apply (auto elim!: fst-type well-ord-Memrel ordertype-Memrel)
done

```

Distributive law for ordinal multiplication and addition

```

lemma oadd-omult-distrib:

```

$\llbracket \text{Ord}(i); \text{Ord}(j); \text{Ord}(k) \rrbracket \implies i ** (j ++ k) = (i ** j) ++ (i ** k)$
apply (*simp add: oadd-eq-if-raw-oadd*)
apply (*simp add: omult-def raw-oadd-def*)
apply (*rule ordertype-eq [THEN trans]*)
apply (*rule prod-ord-iso-cong [OF ordertype-ord-iso [THEN ord-iso-sym]*
ord-iso-refl])
apply (*simp-all add: well-ord-rmult well-ord-radd well-ord-Memrel*
Ord-ordertype)
apply (*rule sum-prod-distrib-ord-iso [THEN ordertype-eq, THEN trans]*)
apply (*rule-tac [2] ordertype-eq*)
apply (*rule-tac [2] sum-ord-iso-cong [OF ordertype-ord-iso ordertype-ord-iso]*)
apply (*simp-all add: well-ord-rmult well-ord-radd well-ord-Memrel*
Ord-ordertype)
done

lemma *omult-succ*: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies i ** \text{succ}(j) = (i ** j) ++ i$
by (*simp del: oadd-succ add: oadd-1 [of j, symmetric] oadd-omult-distrib*)

Associative law

lemma *omult-assoc*:
 $\llbracket \text{Ord}(i); \text{Ord}(j); \text{Ord}(k) \rrbracket \implies (i ** j) ** k = i ** (j ** k)$
unfolding *omult-def*
apply (*rule ordertype-eq [THEN trans]*)
apply (*rule prod-ord-iso-cong [OF ord-iso-refl*
ordertype-ord-iso [THEN ord-iso-sym]])
apply (*blast intro: well-ord-rmult well-ord-Memrel*) +
apply (*rule prod-assoc-ord-iso*
[THEN ord-iso-sym, THEN ordertype-eq, THEN trans])
apply (*rule-tac [2] ordertype-eq*)
apply (*rule-tac [2] prod-ord-iso-cong [OF ordertype-ord-iso ord-iso-refl]*)
apply (*blast intro: well-ord-rmult well-ord-Memrel Ord-ordertype*) +
done

Ordinal multiplication with limit ordinals

lemma *omult-UN*:
 $\llbracket \text{Ord}(i); \bigwedge x. x \in A \implies \text{Ord}(j(x)) \rrbracket$
 $\implies i ** (\bigcup_{x \in A} j(x)) = (\bigcup_{x \in A} i ** j(x))$
by (*simp (no-asm-simp) add: Ord-UN omult-unfold, blast*)

lemma *omult-Limit*: $\llbracket \text{Ord}(i); \text{Limit}(j) \rrbracket \implies i ** j = (\bigcup_{k \in j} i ** k)$
by (*simp add: Limit-is-Ord [THEN Ord-in-Ord] omult-UN [symmetric]*
Union-eq-UN [symmetric] Limit-Union-eq)

21.6.3 Ordering/monotonicity properties of ordinal multiplication

lemma *lt-omult1*: $\llbracket k < i; 0 < j \rrbracket \implies k < i ** j$
apply (*safe elim!: ltE intro!: ltI Ord-omult*)
apply (*force simp add: omult-unfold*)

done

lemma *omult-le-self*: $\llbracket \text{Ord}(i); 0 < j \rrbracket \implies i \leq i**j$
by (*blast intro: all-lt-imp-le Ord-omult lt-omult1 lt-Ord2*)

lemma *omult-le-mono1*:
assumes *kj*: $k \leq j$ **and** *i*: *Ord*(*i*) **shows** $k**i \leq j**i$
proof –
have *o*: *Ord*(*k*) *Ord*(*j*) **by** (*rule lt-Ord [OF kj] le-Ord2 [OF kj]*) +
show ?thesis **using** *i*
proof (*induct i rule: trans-induct3*)
case 0 **thus** ?case
by *simp*
next
case (*succ i*) **thus** ?case
by (*simp add: o kj omult-succ oadd-le-mono*)
next
case (*limit l*)
thus ?case
by (*auto simp add: o kj omult-Limit le-implies-UN-le-UN*)
qed
qed

lemma *omult-lt-mono2*: $\llbracket k < j; 0 < i \rrbracket \implies i**k < i**j$
apply (*rule ltI*)
apply (*simp (no-asm-simp) add: omult-unfold lt-Ord2*)
apply (*safe elim!: ltE intro!: Ord-omult*)
apply (*force simp add: Ord-omult*)
done

lemma *omult-le-mono2*: $\llbracket k \leq j; \text{Ord}(i) \rrbracket \implies i**k \leq i**j$
apply (*rule subset-imp-le*)
apply (*safe elim!: ltE dest!: Ord-succD intro!: Ord-omult*)
apply (*simp add: omult-unfold*)
apply (*blast intro: Ord-trans*)
done

lemma *omult-le-mono*: $\llbracket i' \leq i; j' \leq j \rrbracket \implies i'**j' \leq i**j$
by (*blast intro: le-trans omult-le-mono1 omult-le-mono2 Ord-succD elim: ltE*)

lemma *omult-lt-mono*: $\llbracket i' \leq i; j' < j; 0 < i \rrbracket \implies i'**j' < i**j$
by (*blast intro: lt-trans1 omult-le-mono1 omult-lt-mono2 Ord-succD elim: ltE*)

lemma *omult-le-self2*:
assumes *i*: *Ord*(*i*) **and** *j*: $0 < j$ **shows** $i \leq j**i$
proof –
have *oj*: *Ord*(*j*) **by** (*rule lt-Ord2 [OF j]*)
show ?thesis **using** *i*
proof (*induct i rule: trans-induct3*)

```

    case 0 thus ?case
      by simp
  next
    case (succ i)
    have  $j ** i ++ 0 < j ** i ++ j$ 
      by (rule oadd-lt-mono2 [OF j])
    with succ.hyps show ?case
      by (simp add: oj j omult-succ) (rule lt-trans1)
  next
    case (limit l)
    hence  $l = (\bigcup x \in l. x)$ 
      by (simp add: Union-eq-UN [symmetric] Limit-Union-eq)
    also have  $\dots \leq (\bigcup x \in l. j ** x)$ 
      by (rule le-implies-UN-le-UN) (rule limit.hyps)
    finally have  $l \leq (\bigcup x \in l. j ** x)$  .
    thus ?case using limit.hyps by (simp add: oj omult-Limit)
  qed
qed

```

Further properties of ordinal multiplication

```

lemma omult-inject:  $\llbracket i ** j = i ** k; \ 0 < i; \text{Ord}(j); \text{Ord}(k) \rrbracket \implies j = k$ 
apply (rule Ord-linear-lt)
prefer 4 apply assumption
apply auto
apply (force dest: omult-lt-mono2 simp add: lt-not-refl)+
done

```

21.7 The Relation Lt

```

lemma wf-Lt: wf(Lt)
apply (rule wf-subset)
apply (rule wf-Memrel)
apply (auto simp add: Lt-def Memrel-def lt-def)
done

```

```

lemma irrefl-Lt: irrefl(A, Lt)
by (auto simp add: Lt-def irrefl-def)

```

```

lemma trans-Lt: trans[A](Lt)
apply (simp add: Lt-def trans-on-def)
apply (blast intro: lt-trans)
done

```

```

lemma part-ord-Lt: part-ord(A, Lt)
by (simp add: part-ord-def irrefl-Lt trans-Lt)

```

```

lemma linear-Lt: linear(nat, Lt)
apply (auto dest!: not-lt-imp-le simp add: Lt-def linear-def le-iff)
apply (drule lt-asym, auto)

```

done

lemma *tot-ord-Lt*: *tot-ord*(*nat*,*Lt*)
by (*simp add: tot-ord-def linear-Lt part-ord-Lt*)

lemma *well-ord-Lt*: *well-ord*(*nat*,*Lt*)
by (*simp add: well-ord-def wf-Lt wf-imp-wf-on tot-ord-Lt*)

end

22 Finite Powerset Operator and Finite Function Space

theory *Finite* **imports** *Inductive Epsilon Nat* **begin**

rep-datatype
elimination *natE*
induction *nat-induct*
case-eqns *nat-case-0 nat-case-succ*
recursor-eqns *recursor-0 recursor-succ*

consts

Fin :: $i \Rightarrow i$
FiniteFun :: $[i, i] \Rightarrow i$ ($\langle \langle \text{notation} = \langle \text{infix } -||> \rangle -||> / - \rangle [61, 60] 60 \rangle$)

inductive

domains *Fin*(*A*) \subseteq *Pow*(*A*)
intros
emptyI: $0 \in \text{Fin}(A)$
consI: $\llbracket a \in A; b \in \text{Fin}(A) \rrbracket \implies \text{cons}(a, b) \in \text{Fin}(A)$
type-intros *empty-subsetI cons-subsetI PowI*
type-elim *PowD* [*elim-format*]

inductive

domains *FiniteFun*(*A*,*B*) \subseteq *Fin*(*A***B*)
intros
emptyI: $0 \in A -||> B$
consI: $\llbracket a \in A; b \in B; h \in A -||> B; a \notin \text{domain}(h) \rrbracket$
 $\implies \text{cons}(\langle a, b \rangle, h) \in A -||> B$
type-intros *Fin.intros*

22.1 Finite Powerset Operator

lemma *Fin-mono*: $A \leq B \implies \text{Fin}(A) \subseteq \text{Fin}(B)$
unfolding *Fin.defs*
apply (*rule lfp-mono*)

```

apply (rule Fin.bnd-mono)+
apply blast
done

```

```

lemmas FinD = Fin.dom-subset [THEN subsetD, THEN PowD]

```

```

lemma Fin-induct [case-names 0 cons, induct set: Fin]:
   $\llbracket b \in \text{Fin}(A);$ 
     $P(0);$ 
     $\bigwedge x y. \llbracket x \in A; y \in \text{Fin}(A); x \notin y; P(y) \rrbracket \implies P(\text{cons}(x,y))$ 
 $\rrbracket \implies P(b)$ 
apply (erule Fin.induct, simp)
apply (case-tac  $a \in b$ )
apply (erule cons-absorb [THEN ssubst], assumption)
apply simp
done

```

```

declare Fin.intros [simp]

```

```

lemma Fin-0:  $\text{Fin}(0) = \{0\}$ 
by (blast intro: Fin.emptyI dest: FinD)

```

```

lemma Fin-UnI [simp]:  $\llbracket b \in \text{Fin}(A); c \in \text{Fin}(A) \rrbracket \implies b \cup c \in \text{Fin}(A)$ 
apply (erule Fin-induct)
apply (simp-all add: Un-cons)
done

```

```

lemma Fin-UnionI:  $C \in \text{Fin}(\text{Fin}(A)) \implies \bigcup(C) \in \text{Fin}(A)$ 
by (erule Fin-induct, simp-all)

```

```

lemma Fin-subset-lemma [rule-format]:  $b \in \text{Fin}(A) \implies \forall z. z \leq b \longrightarrow z \in \text{Fin}(A)$ 
apply (erule Fin-induct)
apply (simp add: subset-empty-iff)
apply (simp add: subset-cons-iff distrib-simps, safe)
apply (erule-tac  $b = z$  in cons-Diff [THEN subst], simp)
done

```

```

lemma Fin-subset:  $\llbracket c \leq b; b \in \text{Fin}(A) \rrbracket \implies c \in \text{Fin}(A)$ 
by (blast intro: Fin-subset-lemma)

```

lemma *Fin-IntI1* [*intro,simp*]: $b \in \text{Fin}(A) \implies b \cap c \in \text{Fin}(A)$
by (*blast intro: Fin-subset*)

lemma *Fin-IntI2* [*intro,simp*]: $c \in \text{Fin}(A) \implies b \cap c \in \text{Fin}(A)$
by (*blast intro: Fin-subset*)

lemma *Fin-0-induct-lemma* [*rule-format*]:
 $\llbracket c \in \text{Fin}(A); b \in \text{Fin}(A); P(b);$
 $\bigwedge x y. \llbracket x \in A; y \in \text{Fin}(A); x \in y; P(y) \rrbracket \implies P(y - \{x\})$
 $\rrbracket \implies c \leq b \longrightarrow P(b - c)$
apply (*erule Fin-induct, simp*)
apply (*subst Diff-cons*)
apply (*simp add: cons-subset-iff Diff-subset [THEN Fin-subset]*)
done

lemma *Fin-0-induct*:
 $\llbracket b \in \text{Fin}(A);$
 $P(b);$
 $\bigwedge x y. \llbracket x \in A; y \in \text{Fin}(A); x \in y; P(y) \rrbracket \implies P(y - \{x\})$
 $\rrbracket \implies P(0)$
apply (*rule Diff-cancel [THEN subst]*)
apply (*blast intro: Fin-0-induct-lemma*)
done

lemma *nat-fun-subset-Fin*: $n \in \text{nat} \implies n \rightarrow A \subseteq \text{Fin}(\text{nat} * A)$
apply (*induct-tac n*)
apply (*simp add: subset-iff*)
apply (*simp add: succ-def mem-not-refl [THEN cons-fun-eq]*)
apply (*fast intro!: Fin.consI*)
done

22.2 Finite Function Space

lemma *FiniteFun-mono*:
 $\llbracket A \leq C; B \leq D \rrbracket \implies A -||> B \subseteq C -||> D$
unfolding *FiniteFun.defs*
apply (*rule lfp-mono*)
apply (*rule FiniteFun.bnd-mono*)
apply (*intro Fin-mono Sigma-mono basic-monos, assumption+*)
done

lemma *FiniteFun-mono1*: $A \leq B \implies A -||> A \subseteq B -||> B$
by (*blast dest: FiniteFun-mono*)

lemma *FiniteFun-is-fun*: $h \in A -||> B \implies h \in \text{domain}(h) \rightarrow B$
apply (*erule FiniteFun.induct, simp*)
apply (*simp add: fun-extend3*)

done

lemma *FiniteFun-domain-Fin*: $h \in A -||> B \implies \text{domain}(h) \in \text{Fin}(A)$
by (*erule FiniteFun.induct, simp, simp*)

lemmas *FiniteFun-apply-type = FiniteFun-is-fun* [*THEN apply-type*]

lemma *FiniteFun-subset-lemma* [*rule-format*]:
 $b \in A -||> B \implies \forall z. z \leq b \longrightarrow z \in A -||> B$
apply (*erule FiniteFun.induct*)
apply (*simp add: subset-empty-iff FiniteFun.intros*)
apply (*simp add: subset-cons-iff distrib-simps, safe*)
apply (*erule-tac b = z in cons-Diff* [*THEN subst*])
apply (*drule spec* [*THEN mp*], *assumption*)
apply (*fast intro!*: *FiniteFun.intros*)
done

lemma *FiniteFun-subset*: $\llbracket c \leq b; b \in A -||> B \rrbracket \implies c \in A -||> B$
by (*blast intro: FiniteFun-subset-lemma*)

lemma *fun-FiniteFunI* [*rule-format*]: $A \in \text{Fin}(X) \implies \forall f. f \in A \rightarrow B \longrightarrow f \in A -||> B$
apply (*erule Fin.induct*)
apply (*simp add: FiniteFun.intros, clarify*)
apply (*case-tac a \in b*)
apply (*simp add: cons-absorb*)
apply (*subgoal-tac restrict (f,b) \in b -||> B*)
prefer 2 apply (*blast intro: restrict-type2*)
apply (*subst fun-cons-restrict-eq, assumption*)
apply (*simp add: restrict-def lam-def*)
apply (*blast intro: apply-funtype FiniteFun.intros*
FiniteFun-mono [*THEN* [*2*] *rev-subsetD*])
done

lemma *lam-FiniteFun*: $A \in \text{Fin}(X) \implies (\lambda x \in A. b(x)) \in A -||> \{b(x). x \in A\}$
by (*blast intro: fun-FiniteFunI lam-funtype*)

lemma *FiniteFun-Collect-iff*:
 $f \in \text{FiniteFun}(A, \{y \in B. P(y)\})$
 $\longleftrightarrow f \in \text{FiniteFun}(A, B) \wedge (\forall x \in \text{domain}(f). P(f'x))$
apply *auto*
apply (*blast intro: FiniteFun-mono* [*THEN* [*2*] *rev-subsetD*])
apply (*blast dest: Pair-mem-PiD FiniteFun-is-fun*)
apply (*rule-tac A1 = domain(f) in*
subset-refl [*THEN* [*2*] *FiniteFun-mono, THEN subsetD*])
apply (*fast dest: FiniteFun-domain-Fin Fin.dom-subset* [*THEN subsetD*])

```

apply (rule fun-FiniteFunI)
apply (erule FiniteFun-domain-Fin)
apply (rule-tac  $B = \text{range } f$ ) in fun-weaken-type)
  apply (blast dest: FiniteFun-is-fun range-of-fun range-type apply-equality)+
done

```

22.3 The Contents of a Singleton Set

definition

```

  contents ::  $i \Rightarrow i$  where
    contents( $X$ )  $\equiv$  THE  $x$ .  $X = \{x\}$ 

```

```

lemma contents-eq [simp]: contents ( $\{x\}$ ) =  $x$ 
by (simp add: contents-def)

```

end

23 Cardinal Numbers Without the Axiom of Choice

theory Cardinal **imports** OrderType Finite Nat Sum **begin**

definition

```

  Least ::  $(i \Rightarrow o) \Rightarrow i$  (binder  $\langle \mu \rangle$  10) where
    Least( $P$ )  $\equiv$  THE  $i$ . Ord( $i$ )  $\wedge$   $P(i) \wedge (\forall j. j < i \longrightarrow \neg P(j))$ 

```

definition

```

  eqpoll ::  $[i, i] \Rightarrow o$  (infixl  $\langle \approx \rangle$  50) where
     $A \approx B \equiv \exists f. f \in \text{bij}(A, B)$ 

```

definition

```

  lepoll ::  $[i, i] \Rightarrow o$  (infixl  $\langle \lesssim \rangle$  50) where
     $A \lesssim B \equiv \exists f. f \in \text{inj}(A, B)$ 

```

definition

```

  lesspoll ::  $[i, i] \Rightarrow o$  (infixl  $\langle \prec \rangle$  50) where
     $A \prec B \equiv A \lesssim B \wedge \neg(A \approx B)$ 

```

definition

```

  cardinal ::  $i \Rightarrow i$  ( $\langle \langle \text{open-block notation} = \langle \text{mixfix cardinal} \rangle \rangle | - | \rangle$ )
  where  $|A| \equiv (\mu i. i \approx A)$ 

```

definition

```

  Finite ::  $i \Rightarrow o$  where
    Finite( $A$ )  $\equiv \exists n \in \text{nat}. A \approx n$ 

```

definition

```

  Card ::  $i \Rightarrow o$  where
    Card( $i$ )  $\equiv (i = |i|)$ 

```

23.1 The Schroeder-Bernstein Theorem

See Davey and Priestly, page 106

lemma *decomp-bnd-mono*: $\text{bnd-mono}(X, \lambda W. X - g''(Y - f''W))$
by (*rule bnd-monoI, blast+*)

lemma *Banach-last-equation*:

$g \in Y \multimap X$
 $\implies g''(Y - f'' \text{ lfp}(X, \lambda W. X - g''(Y - f''W))) =$
 $X - \text{ lfp}(X, \lambda W. X - g''(Y - f''W))$

apply (*rule-tac* $P = \lambda u. v = X - u$ **for** v
in *decomp-bnd-mono* [*THEN lfp-unfold, THEN ssubst*])
apply (*simp add: double-complement fun-is-rel* [*THEN image-subset*])
done

lemma *decomposition*:

$\llbracket f \in X \multimap Y; g \in Y \multimap X \rrbracket \implies$
 $\exists XA XB YA YB. (XA \cap XB = 0) \wedge (XA \cup XB = X) \wedge$
 $(YA \cap YB = 0) \wedge (YA \cup YB = Y) \wedge$
 $f''XA = YA \wedge g''YB = XB$

apply (*intro exI conjI*)
apply (*rule-tac* [6] *Banach-last-equation*)
apply (*rule-tac* [5] *refl*)
apply (*assumption* |
rule Diff-disjoint Diff-partition fun-is-rel image-subset lfp-subset) +
done

lemma *schroeder-bernstein*:

$\llbracket f \in \text{inj}(X, Y); g \in \text{inj}(Y, X) \rrbracket \implies \exists h. h \in \text{bij}(X, Y)$
apply (*insert decomposition* [*of f X Y g*])
apply (*simp add: inj-is-fun*)
apply (*blast intro!: restrict-bij bij-disjoint-Un intro: bij-converse-bij*)
done

lemma *bij-imp-epoll*: $f \in \text{bij}(A, B) \implies A \approx B$
unfolding *epoll-def*
apply (*erule exI*)
done

lemmas *epoll-refl* = *id-bij* [*THEN bij-imp-epoll, simp*]

lemma *epoll-sym*: $X \approx Y \implies Y \approx X$
unfolding *epoll-def*
apply (*blast intro: bij-converse-bij*)

done

lemma *eqpoll-trans* [*trans*]:
 $\llbracket X \approx Y; Y \approx Z \rrbracket \implies X \approx Z$
unfolding *eqpoll-def*
apply (*blast intro: comp-bij*)
done

lemma *subset-imp-lepoll*: $X \leq Y \implies X \lesssim Y$
unfolding *lepoll-def*
apply (*rule exI*)
apply (*erule id-subset-inj*)
done

lemmas *lepoll-refl* = *subset-refl* [*THEN subset-imp-lepoll, simp*]

lemmas *le-imp-lepoll* = *le-imp-subset* [*THEN subset-imp-lepoll*]

lemma *eqpoll-imp-lepoll*: $X \approx Y \implies X \lesssim Y$
by (*unfold eqpoll-def bij-def lepoll-def, blast*)

lemma *lepoll-trans* [*trans*]: $\llbracket X \lesssim Y; Y \lesssim Z \rrbracket \implies X \lesssim Z$
unfolding *lepoll-def*
apply (*blast intro: comp-inj*)
done

lemma *eq-lepoll-trans* [*trans*]: $\llbracket X \approx Y; Y \lesssim Z \rrbracket \implies X \lesssim Z$
by (*blast intro: eqpoll-imp-lepoll lepoll-trans*)

lemma *lepoll-eq-trans* [*trans*]: $\llbracket X \lesssim Y; Y \approx Z \rrbracket \implies X \lesssim Z$
by (*blast intro: eqpoll-imp-lepoll lepoll-trans*)

lemma *eqpollI*: $\llbracket X \lesssim Y; Y \lesssim X \rrbracket \implies X \approx Y$
unfolding *lepoll-def eqpoll-def*
apply (*elim exE*)
apply (*rule schroeder-bernstein, assumption+*)
done

lemma *eqpollE*:
 $\llbracket X \approx Y; \llbracket X \lesssim Y; Y \lesssim X \rrbracket \implies P \rrbracket \implies P$
by (*blast intro: eqpoll-imp-lepoll eqpoll-sym*)

lemma *eqpoll-iff*: $X \approx Y \longleftrightarrow X \lesssim Y \wedge Y \lesssim X$
by (*blast intro: eqpollI elim!: eqpollE*)

lemma *lepoll-0-is-0*: $A \lesssim 0 \implies A = 0$

unfolding *lepoll-def inj-def*
apply (*blast dest: apply-type*)
done

lemmas *empty-lepollI = empty-subsetI [THEN subset-imp-lepoll]*

lemma *lepoll-0-iff*: $A \lesssim 0 \longleftrightarrow A=0$
by (*blast intro: lepoll-0-is-0 lepoll-refl*)

lemma *Un-lepoll-Un*:
 $\llbracket A \lesssim B; C \lesssim D; B \cap D = 0 \rrbracket \implies A \cup C \lesssim B \cup D$
unfolding *lepoll-def*
apply (*blast intro: inj-disjoint-Un*)
done

lemmas *eqpoll-0-is-0 = eqpoll-imp-lepoll [THEN lepoll-0-is-0]*

lemma *eqpoll-0-iff*: $A \approx 0 \longleftrightarrow A=0$
by (*blast intro: eqpoll-0-is-0 eqpoll-refl*)

lemma *eqpoll-disjoint-Un*:
 $\llbracket A \approx B; C \approx D; A \cap C = 0; B \cap D = 0 \rrbracket$
 $\implies A \cup C \approx B \cup D$
unfolding *eqpoll-def*
apply (*blast intro: bij-disjoint-Un*)
done

23.2 lesspoll: contributions by Krzysztof Grabczewski

lemma *lesspoll-not-refl*: $\neg (i \prec i)$
by (*simp add: lesspoll-def*)

lemma *lesspoll-irrefl [elim!]*: $i \prec i \implies P$
by (*simp add: lesspoll-def*)

lemma *lesspoll-imp-lepoll*: $A \prec B \implies A \lesssim B$
by (*unfold lesspoll-def, blast*)

lemma *lepoll-well-ord*: $\llbracket A \lesssim B; \text{well-ord}(B,r) \rrbracket \implies \exists s. \text{well-ord}(A,s)$
unfolding *lepoll-def*
apply (*blast intro: well-ord-rvimage*)
done

lemma *lepoll-iff-leqpoll*: $A \lesssim B \longleftrightarrow A \prec B \mid A \approx B$
unfolding *lesspoll-def*
apply (*blast intro!: eqpollI elim!: eqpollE*)
done

```

lemma inj-not-surj-succ:
  assumes fi:  $f \in \text{inj}(A, \text{succ}(m))$  and fns:  $f \notin \text{surj}(A, \text{succ}(m))$ 
  shows  $\exists f. f \in \text{inj}(A, m)$ 
proof -
  from fi [THEN inj-is-fun] fns
  obtain y where y:  $y \in \text{succ}(m) \wedge x. x \in A \implies f \cdot x \neq y$ 
  by (auto simp add: surj-def)
  show ?thesis
  proof
    show  $(\lambda z \in A. \text{if } f \cdot z = m \text{ then } y \text{ else } f \cdot z) \in \text{inj}(A, m)$  using y fi
    by (simp add: inj-def)
    (auto intro!: if-type [THEN lam-type] intro: Pi-type dest: apply-funtype)
  qed
qed

```

```

lemma lesspoll-trans [trans]:
   $\llbracket X \prec Y; Y \prec Z \rrbracket \implies X \prec Z$ 
  unfolding lesspoll-def
apply (blast elim!: eqpollE intro: eqpollI lepoll-trans)
done

```

```

lemma lesspoll-trans1 [trans]:
   $\llbracket X \lesssim Y; Y \prec Z \rrbracket \implies X \prec Z$ 
  unfolding lesspoll-def
apply (blast elim!: eqpollE intro: eqpollI lepoll-trans)
done

```

```

lemma lesspoll-trans2 [trans]:
   $\llbracket X \prec Y; Y \lesssim Z \rrbracket \implies X \prec Z$ 
  unfolding lesspoll-def
apply (blast elim!: eqpollE intro: eqpollI lepoll-trans)
done

```

```

lemma eq-lesspoll-trans [trans]:
   $\llbracket X \approx Y; Y \prec Z \rrbracket \implies X \prec Z$ 
  by (blast intro: eqpoll-imp-lepoll lesspoll-trans1)

```

```

lemma lesspoll-eq-trans [trans]:
   $\llbracket X \prec Y; Y \approx Z \rrbracket \implies X \prec Z$ 
  by (blast intro: eqpoll-imp-lepoll lesspoll-trans2)

```

```

lemma Least-equality:
   $\llbracket P(i); \text{Ord}(i); \bigwedge x. x < i \implies \neg P(x) \rrbracket \implies (\mu x. P(x)) = i$ 

```

```

    unfolding Least-def
  apply (rule the-equality, blast)
  apply (elim conjE)
  apply (erule Ord-linear-lt, assumption, blast+)
done

lemma LeastI:
  assumes P:  $P(i)$  and i:  $\text{Ord}(i)$  shows  $P(\mu x. P(x))$ 
proof -
  { from i have  $P(i) \implies P(\mu x. P(x))$ 
    proof (induct i rule: trans-induct)
      case (step i)
      show ?case
        proof (cases  $P(\mu a. P(a))$ )
          case True thus ?thesis .
        next
          case False
          hence  $\bigwedge x. x \in i \implies \neg P(x)$  using step
            by blast
          hence  $(\mu a. P(a)) = i$  using step
            by (blast intro: Least-equality ltD)
          thus ?thesis using step.prem by simp
        qed
      qed
    }
  thus ?thesis using P .
qed

```

The proof is almost identical to the one above!

```

lemma Least-le:
  assumes P:  $P(i)$  and i:  $\text{Ord}(i)$  shows  $(\mu x. P(x)) \leq i$ 
proof -
  { from i have  $P(i) \implies (\mu x. P(x)) \leq i$ 
    proof (induct i rule: trans-induct)
      case (step i)
      show ?case
        proof (cases  $(\mu a. P(a)) \leq i$ )
          case True thus ?thesis .
        next
          case False
          hence  $\bigwedge x. x \in i \implies \neg (\mu a. P(a)) \leq i$  using step
            by blast
          hence  $(\mu a. P(a)) = i$  using step
            by (blast elim: ltE intro: ltI Least-equality lt-trans1)
          thus ?thesis using step by simp
        qed
      qed
    }
  thus ?thesis by simp
qed

```

```

    }
    thus ?thesis using P .
qed

```

```

lemma less-LeastE:  $\llbracket P(i); i < (\mu x. P(x)) \rrbracket \implies Q$ 
apply (rule Least-le [THEN [2] lt-trans2, THEN lt-irrefl], assumption+)
apply (simp add: lt-Ord)
done

```

```

lemma LeastI2:
 $\llbracket P(i); \text{Ord}(i); \bigwedge j. P(j) \implies Q(j) \rrbracket \implies Q(\mu j. P(j))$ 
by (blast intro: LeastI)

```

```

lemma Least-0:
 $\llbracket \neg (\exists i. \text{Ord}(i) \wedge P(i)) \rrbracket \implies (\mu x. P(x)) = 0$ 
unfolding Least-def
apply (rule the-0, blast)
done

```

```

lemma Ord-Least [intro,simp,TC]:  $\text{Ord}(\mu x. P(x))$ 
proof (cases  $\exists i. \text{Ord}(i) \wedge P(i)$ )
case True
then obtain i where  $P(i) \wedge \text{Ord}(i)$  by auto
hence  $(\mu x. P(x)) \leq i$  by (rule Least-le)
thus ?thesis
by (elim ltE)
next
case False
hence  $(\mu x. P(x)) = 0$  by (rule Least-0)
thus ?thesis
by auto
qed

```

23.3 Basic Properties of Cardinals

```

lemma Least-cong:  $(\bigwedge y. P(y) \longleftrightarrow Q(y)) \implies (\mu x. P(x)) = (\mu x. Q(x))$ 
by simp

```

```

lemma cardinal-cong:  $X \approx Y \implies |X| = |Y|$ 
unfolding eqpoll-def cardinal-def
apply (rule Least-cong)
apply (blast intro: comp-bij bij-converse-bij)
done

```


lemma *well-ord-cardinal-epoll*:
assumes r : *well-ord*(A, r) **shows** $|A| \approx A$
proof (*unfold cardinal-def*)
show $(\mu i. i \approx A) \approx A$
by (*best intro: LeastI Ord-ordertype ordermap-bij bij-converse-bij bij-imp-epoll*
 r)
qed

lemmas *Ord-cardinal-epoll* = *well-ord-Memrel* [*THEN well-ord-cardinal-epoll*]

lemma *Ord-cardinal-idem*: *Ord*(A) $\implies ||A|| = |A|$
by (*rule Ord-cardinal-epoll* [*THEN cardinal-cong*])

lemma *well-ord-cardinal-epE*:
assumes woX : *well-ord*(X, r) **and** woY : *well-ord*(Y, s) **and** eq : $|X| = |Y|$
shows $X \approx Y$
proof –
have $X \approx |X|$ **by** (*blast intro: well-ord-cardinal-epoll* [*OF woX*] *epoll-sym*)
also have $\dots = |Y|$ **by** (*rule eq*)
also have $\dots \approx Y$ **by** (*rule well-ord-cardinal-epoll* [*OF woY*])
finally show *?thesis* .
qed

lemma *well-ord-cardinal-epoll-iff*:
 $\llbracket \text{well-ord}(X, r); \text{well-ord}(Y, s) \rrbracket \implies |X| = |Y| \longleftrightarrow X \approx Y$
by (*blast intro: cardinal-cong well-ord-cardinal-epE*)

lemma *Ord-cardinal-le*: *Ord*(i) $\implies |i| \leq i$
unfolding *cardinal-def*
apply (*erule epoll-refl* [*THEN Least-le*])
done

lemma *Card-cardinal-eq*: *Card*(K) $\implies |K| = K$
unfolding *Card-def*
apply (*erule sym*)
done

lemma *CardI*: $\llbracket \text{Ord}(i); \bigwedge j. j < i \implies \neg(j \approx i) \rrbracket \implies \text{Card}(i)$
unfolding *Card-def cardinal-def*
apply (*subst Least-equality*)
apply (*blast intro: epoll-refl*)
done

lemma *Card-is-Ord*: *Card*(i) $\implies \text{Ord}(i)$

```

  unfolding Card-def cardinal-def
  apply (erule ssubst)
  apply (rule Ord-Least)
done

```

```

lemma Card-cardinal-le:  $\text{Card}(K) \implies K \leq |K|$ 
  apply (simp (no-asm-simp) add: Card-is-Ord Card-cardinal-eq)
done

```

```

lemma Ord-cardinal [simp,intro!]:  $\text{Ord}(|A|)$ 
  unfolding cardinal-def
  apply (rule Ord-Least)
done

```

The cardinals are the initial ordinals.

```

lemma Card-iff-initial:  $\text{Card}(K) \longleftrightarrow \text{Ord}(K) \wedge (\forall j. j < K \longrightarrow \neg j \approx K)$ 
proof -
  { fix j
    assume K:  $\text{Card}(K)$   $j \approx K$ 
    assume  $j < K$ 
    also have ... =  $(\mu i. i \approx K)$  using K
    by (simp add: Card-def cardinal-def)
    finally have  $j < (\mu i. i \approx K)$  .
    hence False using K
    by (best dest: less-LeastE)
  }
  then show ?thesis
  by (blast intro: CardI Card-is-Ord)
qed

```

```

lemma lt-Card-imp-lesspoll:  $\llbracket \text{Card}(a); i < a \rrbracket \implies i < a$ 
  unfolding lesspoll-def
  apply (drule Card-iff-initial [THEN iffD1])
  apply (blast intro!: leI [THEN le-imp-lepoll])
done

```

```

lemma Card-0:  $\text{Card}(0)$ 
  apply (rule Ord-0 [THEN CardI])
  apply (blast elim!: ltE)
done

```

```

lemma Card-Un:  $\llbracket \text{Card}(K); \text{Card}(L) \rrbracket \implies \text{Card}(K \cup L)$ 
  apply (rule Ord-linear-le [of K L])
  apply (simp-all add: subset-Un-iff [THEN iffD1] Card-is-Ord le-imp-subset
    subset-Un-iff2 [THEN iffD1])
done

```

```

lemma Card-cardinal [iff]: Card( $|A|$ )
proof (unfold cardinal-def)
  show Card( $\mu i. i \approx A$ )
    proof (cases  $\exists i. \text{Ord}(i) \wedge i \approx A$ )
      case False thus ?thesis — degenerate case
      by (simp add: Least-0 Card-0)
    next
      case True — real case: A is isomorphic to some ordinal
      then obtain i where  $i: \text{Ord}(i) \ i \approx A$  by blast
      show ?thesis
        proof (rule CardI [OF Ord-Least], rule notI)
          fix j
          assume  $j: j < (\mu i. i \approx A)$ 
          assume  $j \approx (\mu i. i \approx A)$ 
          also have  $\dots \approx A$  using i by (auto intro: LeastI)
          finally have  $j \approx A$  .
          thus False
          by (rule less-LeastE [OF - j])
        qed
      qed
    qed

```

```

lemma cardinal-eq-lemma:
  assumes  $i: |i| \leq j$  and  $j: j \leq i$  shows  $|j| = |i|$ 
proof (rule eqpollI [THEN cardinal-cong])
  show  $j \lesssim i$  by (rule le-imp-lepoll [OF j])
next
  have  $Oi: \text{Ord}(i)$  using j by (rule le-Ord2)
  hence  $i \approx |i|$ 
  by (blast intro: Ord-cardinal-eqpoll eqpoll-sym)
  also have  $\dots \lesssim j$ 
  by (blast intro: le-imp-lepoll i)
  finally show  $i \lesssim j$  .
qed

```

```

lemma cardinal-mono:
  assumes  $ij: i \leq j$  shows  $|i| \leq |j|$ 
using Ord-cardinal [of i] Ord-cardinal [of j]
proof (cases rule: Ord-linear-le)
  case le thus ?thesis .
next
  case ge
  have  $i: \text{Ord}(i)$  using ij
  by (simp add: lt-Ord)
  have  $ci: |i| \leq j$ 
  by (blast intro: Ord-cardinal-le ij le-trans i)
  have  $|i| = ||i||$ 
  by (auto simp add: Ord-cardinal-idem i)

```

also have $\dots = |j|$
by (rule cardinal-eq-lemma [OF ge ci])
finally have $|i| = |j|$.
thus ?thesis **by** simp
qed

Since we have $|succ(nat)| \leq |nat|$, the converse of *cardinal-mono* fails!

lemma cardinal-lt-imp-lt: $\llbracket |i| < |j|; \text{Ord}(i); \text{Ord}(j) \rrbracket \implies i < j$
apply (rule Ord-linear2 [of i j], assumption+)
apply (erule lt-trans2 [THEN lt-irrefl])
apply (erule cardinal-mono)
done

lemma Card-lt-imp-lt: $\llbracket |i| < K; \text{Ord}(i); \text{Card}(K) \rrbracket \implies i < K$
by (simp (no-asm-simp) add: cardinal-lt-imp-lt Card-is-Ord Card-cardinal-eq)

lemma Card-lt-iff: $\llbracket \text{Ord}(i); \text{Card}(K) \rrbracket \implies (|i| < K) \longleftrightarrow (i < K)$
by (blast intro: Card-lt-imp-lt Ord-cardinal-le [THEN lt-trans1])

lemma Card-le-iff: $\llbracket \text{Ord}(i); \text{Card}(K) \rrbracket \implies (K \leq |i|) \longleftrightarrow (K \leq i)$
by (simp add: Card-lt-iff Card-is-Ord Ord-cardinal not-lt-iff-le [THEN iff-sym])

lemma well-ord-lepoll-imp-cardinal-le:
assumes $wB: \text{well-ord}(B, r)$ **and** $AB: A \lesssim B$
shows $|A| \leq |B|$
using Ord-cardinal [of A] Ord-cardinal [of B]
proof (cases rule: Ord-linear-le)
case le **thus** ?thesis .
next
case ge
from lepoll-well-ord [OF AB wB]
obtain s **where** $s: \text{well-ord}(A, s)$ **by** blast
have $B \approx |B|$ **by** (blast intro: wB eqpoll-sym well-ord-cardinal-epoll)
also have $\dots \lesssim |A|$ **by** (rule le-imp-lepoll [OF ge])
also have $\dots \approx A$ **by** (rule well-ord-cardinal-epoll [OF s])
finally have $B \lesssim A$.
hence $A \approx B$ **by** (blast intro: eqpollI AB)
hence $|A| = |B|$ **by** (rule cardinal-cong)
thus ?thesis **by** simp
qed

lemma lepoll-cardinal-le: $\llbracket A \lesssim i; \text{Ord}(i) \rrbracket \implies |A| \leq i$
apply (rule le-trans)
apply (erule well-ord-Memrel [THEN well-ord-lepoll-imp-cardinal-le], assumption)
apply (erule Ord-cardinal-le)
done

lemma lepoll-Ord-imp-epoll: $\llbracket A \lesssim i; \text{Ord}(i) \rrbracket \implies |A| \approx A$

by (*blast intro: lepoll-cardinal-le well-ord-Memrel well-ord-cardinal-epoll dest!: lepoll-well-ord*)

lemma *lesspoll-imp-epoll*: $\llbracket A \prec i; \text{Ord}(i) \rrbracket \implies |A| \approx A$
unfolding *lesspoll-def*
apply (*blast intro: lepoll-Ord-imp-epoll*)
done

lemma *cardinal-subset-Ord*: $\llbracket A \leq i; \text{Ord}(i) \rrbracket \implies |A| \subseteq i$
apply (*drule subset-imp-lepoll [THEN lepoll-cardinal-le]*)
apply (*auto simp add: lt-def*)
apply (*blast intro: Ord-trans*)
done

23.4 The finite cardinals

lemma *cons-lepoll-consD*:
 $\llbracket \text{cons}(u, A) \lesssim \text{cons}(v, B); u \notin A; v \notin B \rrbracket \implies A \lesssim B$
apply (*unfold lepoll-def inj-def, safe*)
apply (*rule-tac x = $\lambda x \in A. \text{if } f'x=v \text{ then } f'u \text{ else } f'x$ in exI*)
apply (*rule CollectI*)

apply (*rule if-type [THEN lam-type]*)
apply (*blast dest: apply-funtype*)
apply (*blast elim!: mem-irrefl dest: apply-funtype*)

apply (*simp (no-asm-simp)*)
apply *blast*
done

lemma *cons-epoll-consD*: $\llbracket \text{cons}(u, A) \approx \text{cons}(v, B); u \notin A; v \notin B \rrbracket \implies A \approx B$
apply (*simp add: epoll-iff*)
apply (*blast intro: cons-lepoll-consD*)
done

lemma *succ-lepoll-succD*: $\text{succ}(m) \lesssim \text{succ}(n) \implies m \lesssim n$
unfolding *succ-def*
apply (*erule cons-lepoll-consD*)
apply (*rule mem-not-refl*)
done

lemma *nat-lepoll-imp-le*:
 $m \in \text{nat} \implies n \in \text{nat} \implies m \lesssim n \implies m \leq n$
proof (*induct m arbitrary: n rule: nat-induct*)
case 0 **thus** ?case **by** (*blast intro!: nat-0-le*)
next
case (*succ m*)

```

show ?case using ⟨ $n \in \text{nat}$ ⟩
proof (cases rule: natE)
  case 0 thus ?thesis using succ
  by (simp add: lepoll-def inj-def)
next
  case (succ  $n'$ ) thus ?thesis using succ.hyps ⟨ $\text{succ}(m) \lesssim n$ ⟩
  by (blast intro!: succ-leI dest!: succ-lepoll-succD)
qed
qed

```

```

lemma nat-eqpoll-iff:  $\llbracket m \in \text{nat}; n \in \text{nat} \rrbracket \implies m \approx n \longleftrightarrow m = n$ 
apply (rule iffI)
apply (blast intro: nat-lepoll-imp-le le-anti-sym elim!: eqpollE)
apply (simp add: eqpoll-refl)
done

```

```

lemma nat-into-Card:
  assumes  $n: n \in \text{nat}$  shows Card( $n$ )
proof (unfold Card-def cardinal-def, rule sym)
  have Ord( $n$ ) using  $n$  by auto
  moreover
  { fix  $i$ 
    assume  $i < n$   $i \approx n$ 
    hence False using  $n$ 
    by (auto simp add: lt-nat-in-nat [THEN nat-eqpoll-iff])
  }
  ultimately show  $(\mu i. i \approx n) = n$  by (auto intro!: Least-equality)
qed

```

```

lemmas cardinal-0 = nat-0I [THEN nat-into-Card, THEN Card-cardinal-eq, iff]
lemmas cardinal-1 = nat-1I [THEN nat-into-Card, THEN Card-cardinal-eq, iff]

```

```

lemma succ-lepoll-natE:  $\llbracket \text{succ}(n) \lesssim n; n \in \text{nat} \rrbracket \implies P$ 
by (rule nat-lepoll-imp-le [THEN lt-irrefl], auto)

```

```

lemma nat-lepoll-imp-ex-epoll-n:
   $\llbracket n \in \text{nat}; \text{nat} \lesssim X \rrbracket \implies \exists Y. Y \subseteq X \wedge n \approx Y$ 
  unfolding lepoll-def eqpoll-def
apply (fast del: subsetI subsetCE
  intro!: subset-SIs
  dest!: Ord-nat [THEN [2] OrdmemD, THEN [2] restrict-inj]
  elim!: restrict-bij
  inj-is-fun [THEN fun-is-rel, THEN image-subset])
done

```

lemma *lepoll-succ*: $i \lesssim \text{succ}(i)$
by (*blast intro: subset-imp-lepoll*)

lemma *lepoll-imp-lesspoll-succ*:
assumes $A: A \lesssim m$ **and** $m: m \in \text{nat}$
shows $A \prec \text{succ}(m)$
proof –
{ **assume** $A \approx \text{succ}(m)$
hence $\text{succ}(m) \approx A$ **by** (*rule eqpoll-sym*)
also have $\dots \lesssim m$ **by** (*rule A*)
finally have $\text{succ}(m) \lesssim m$.
hence *False* **by** (*rule succ-lepoll-natE*) (*rule m*) }
moreover have $A \lesssim \text{succ}(m)$ **by** (*blast intro: lepoll-trans A lepoll-succ*)
ultimately show *?thesis* **by** (*auto simp add: lesspoll-def*)
qed

lemma *lesspoll-succ-imp-lepoll*:
 $\llbracket A \prec \text{succ}(m); m \in \text{nat} \rrbracket \implies A \lesssim m$
unfolding *lesspoll-def lepoll-def eqpoll-def bij-def*
apply (*auto dest: inj-not-surj-succ*)
done

lemma *lesspoll-succ-iff*: $m \in \text{nat} \implies A \prec \text{succ}(m) \longleftrightarrow A \lesssim m$
by (*blast intro!: lepoll-imp-lesspoll-succ lesspoll-succ-imp-lepoll*)

lemma *lepoll-succ-disj*: $\llbracket A \lesssim \text{succ}(m); m \in \text{nat} \rrbracket \implies A \lesssim m \mid A \approx \text{succ}(m)$
apply (*rule disjCI*)
apply (*rule lesspoll-succ-imp-lepoll*)
prefer 2 **apply** *assumption*
apply (*simp (no-asm-simp) add: lesspoll-def*)
done

lemma *lesspoll-cardinal-lt*: $\llbracket A \prec i; \text{Ord}(i) \rrbracket \implies |A| < i$
apply (*unfold lesspoll-def, clarify*)
apply (*frule lepoll-cardinal-le, assumption*)
apply (*blast intro: well-ord-Memrel well-ord-cardinal-epoll [THEN eqpoll-sym]*
dest: lepoll-well-ord elim!: leE)
done

23.5 The first infinite cardinal: Omega, or nat

lemma *lt-not-lepoll*:
assumes $n: n < i$ $n \in \text{nat}$ **shows** $\neg i \lesssim n$
proof –
{ **assume** $i: i \lesssim n$
have $\text{succ}(n) \lesssim i$ **using** n
by (*elim ltE, blast intro: Ord-succ-subsetI [THEN subset-imp-lepoll]*)

also have ... $\lesssim n$ by (rule i)
 finally have $\text{succ}(n) \lesssim n$.
 hence *False* by (rule *succ-lepoll-natE*) (rule n) }
 thus ?thesis by auto
 qed

A slightly weaker version of *nat-epoll-iff*

lemma *Ord-nat-epoll-iff*:
 assumes *i*: *Ord*(*i*) and *n*: $n \in \text{nat}$ shows $i \approx n \longleftrightarrow i = n$
 using *i* *nat-into-Ord* [OF *n*]
proof (cases rule: *Ord-linear-lt*)
 case *lt*
 hence $i \in \text{nat}$ by (rule *lt-nat-in-nat*) (rule *n*)
 thus ?thesis by (simp add: *nat-epoll-iff* *n*)
 next
 case *eq*
 thus ?thesis by (simp add: *eqpoll-refl*)
 next
 case *gt*
 hence $\neg i \lesssim n$ using *n* by (rule *lt-not-lepoll*)
 hence $\neg i \approx n$ using *n* by (blast intro: *eqpoll-imp-lepoll*)
 moreover have $i \neq n$ using $\langle n < i \rangle$ by auto
 ultimately show ?thesis by blast
 qed

lemma *Card-nat*: *Card*(*nat*)
proof –
 { fix *i*
 assume *i*: $i < \text{nat}$ $i \approx \text{nat}$
 hence $\neg \text{nat} \lesssim i$
 by (simp add: *lt-def* *lt-not-lepoll*)
 hence *False* using *i*
 by (simp add: *eqpoll-iff*)
 }
 hence $(\mu i. i \approx \text{nat}) = \text{nat}$ by (blast intro: *Least-equality* *eqpoll-refl*)
 thus ?thesis
 by (auto simp add: *Card-def* *cardinal-def*)
 qed

lemma *nat-le-cardinal*: $\text{nat} \leq i \implies \text{nat} \leq |i|$
apply (rule *Card-nat* [THEN *Card-cardinal-eq*, THEN *subst*])
apply (erule *cardinal-mono*)
done

lemma *n-lesspoll-nat*: $n \in \text{nat} \implies n < \text{nat}$
 by (blast intro: *Ord-nat* *Card-nat* *ltI* *lt-Card-imp-lesspoll*)

23.6 Towards Cardinal Arithmetic

lemma *cons-lepoll-cong*:

$\llbracket A \lesssim B; b \notin B \rrbracket \implies \text{cons}(a, A) \lesssim \text{cons}(b, B)$
apply (*unfold lepoll-def, safe*)
apply (*rule-tac* $x = \lambda y \in \text{cons}(a, A) . \text{if } y=a \text{ then } b \text{ else } f'y \text{ in } exI$)
apply (*rule-tac* $d = \lambda z. \text{if } z \in B \text{ then } \text{converse}(f) 'z \text{ else } a \text{ in lam-injective}$)
apply (*safe elim! consE*)
apply *simp-all*
apply (*blast intro: inj-is-fun [THEN apply-type]*)
done

lemma *cons-epoll-cong*:

$\llbracket A \approx B; a \notin A; b \notin B \rrbracket \implies \text{cons}(a, A) \approx \text{cons}(b, B)$
by (*simp add: eqpoll-iff cons-lepoll-cong*)

lemma *cons-lepoll-cons-iff*:

$\llbracket a \notin A; b \notin B \rrbracket \implies \text{cons}(a, A) \lesssim \text{cons}(b, B) \longleftrightarrow A \lesssim B$
by (*blast intro: cons-lepoll-cong cons-lepoll-consD*)

lemma *cons-epoll-cons-iff*:

$\llbracket a \notin A; b \notin B \rrbracket \implies \text{cons}(a, A) \approx \text{cons}(b, B) \longleftrightarrow A \approx B$
by (*blast intro: cons-epoll-cong cons-epoll-consD*)

lemma *singleton-epoll-1*: $\{a\} \approx 1$

unfolding *succ-def*
apply (*blast intro!: eqpoll-refl [THEN cons-epoll-cong]*)
done

lemma *cardinal-singleton*: $|\{a\}| = 1$

apply (*rule singleton-epoll-1 [THEN cardinal-cong, THEN trans]*)
apply (*simp (no-asm) add: nat-into-Card [THEN Card-cardinal-eq]*)
done

lemma *not-0-is-lepoll-1*: $A \neq 0 \implies 1 \lesssim A$

apply (*erule not-emptyE*)
apply (*rule-tac* $a = \text{cons}(x, A - \{x\}) \text{ in subst}$)
apply (*rule-tac* $[2] a = \text{cons}(0, 0) \text{ and } P = \lambda y. y \lesssim \text{cons}(x, A - \{x\}) \text{ in subst}$)
prefer 3 **apply** (*blast intro: cons-lepoll-cong subset-imp-lepoll, auto*)
done

lemma *succ-epoll-cong*: $A \approx B \implies \text{succ}(A) \approx \text{succ}(B)$

unfolding *succ-def*
apply (*simp add: cons-epoll-cong mem-not-refl*)
done

lemma *sum-epoll-cong*: $\llbracket A \approx C; B \approx D \rrbracket \implies A+B \approx C+D$

unfolding *epoll-def*

apply (*blast intro!*: *sum-bij*)
done

lemma *prod-epoll-cong*:
 $\llbracket A \approx C; B \approx D \rrbracket \implies A * B \approx C * D$
unfolding *epoll-def*
apply (*blast intro!*: *prod-bij*)
done

lemma *inj-disjoint-epoll*:
 $\llbracket f \in \text{inj}(A, B); A \cap B = 0 \rrbracket \implies A \cup (B - \text{range}(f)) \approx B$
unfolding *epoll-def*
apply (*rule exI*)
apply (*rule-tac* $c = \lambda x. \text{if } x \in A \text{ then } f'x \text{ else } x$
and $d = \lambda y. \text{if } y \in \text{range}(f) \text{ then } \text{converse}(f) 'y \text{ else } y$
in *lam-bijective*)
apply (*blast intro!*: *if-type inj-is-fun [THEN apply-type]*)
apply (*simp* (*no-asm-simp*) *add: inj-converse-fun [THEN apply-funtype]*)
apply (*safe elim!*: *UnE'*)
apply (*simp-all* *add: inj-is-fun [THEN apply-rangeI]*)
apply (*blast intro: inj-converse-fun [THEN apply-type]*) +
done

23.7 Lemmas by Krzysztof Grabczewski

If A has at most $n + 1$ elements and $a \in A$ then $A - \{a\}$ has at most n .

lemma *Diff-sing-lepoll*:
 $\llbracket a \in A; A \lesssim \text{succ}(n) \rrbracket \implies A - \{a\} \lesssim n$
unfolding *succ-def*
apply (*rule cons-lepoll-consD*)
apply (*rule-tac* [β] *mem-not-refl*)
apply (*erule cons-Diff [THEN ssubst]*, *safe*)
done

If A has at least $n + 1$ elements then $A - \{a\}$ has at least n .

lemma *lepoll-Diff-sing*:
assumes $A: \text{succ}(n) \lesssim A$ **shows** $n \lesssim A - \{a\}$
proof –
have $\text{cons}(n, n) \lesssim A$ **using** A
by (*unfold succ-def*)
also have $\dots \lesssim \text{cons}(a, A - \{a\})$
by (*blast intro: subset-imp-lepoll*)
finally have $\text{cons}(n, n) \lesssim \text{cons}(a, A - \{a\})$.
thus *?thesis*
by (*blast intro: cons-lepoll-consD mem-irrefl*)
qed

lemma *Diff-sing-epoll*: $\llbracket a \in A; A \approx \text{succ}(n) \rrbracket \implies A - \{a\} \approx n$

by (blast intro!: eqpollI
 elim!: eqpollE
 intro: Diff-sing-lepoll lepoll-Diff-sing)

lemma lepoll-1-is-sing: $\llbracket A \lesssim 1; a \in A \rrbracket \implies A = \{a\}$
 apply (frule Diff-sing-lepoll, assumption)
 apply (drule lepoll-0-is-0)
 apply (blast elim: equalityE)
 done

lemma Un-lepoll-sum: $A \cup B \lesssim A+B$
 unfolding lepoll-def
 apply (rule-tac $x = \lambda x \in A \cup B. \text{if } x \in A \text{ then } \text{Inl } (x) \text{ else } \text{Inr } (x) \text{ in } exI$)
 apply (rule-tac $d = \lambda z. \text{snd } (z) \text{ in } \text{lam-injective}$)
 apply force
 apply (simp add: Inl-def Inr-def)
 done

lemma well-ord-Un:
 $\llbracket \text{well-ord}(X, R); \text{well-ord}(Y, S) \rrbracket \implies \exists T. \text{well-ord}(X \cup Y, T)$
 by (erule well-ord-radd [THEN Un-lepoll-sum [THEN lepoll-well-ord]],
 assumption)

lemma disj-Un-epoll-sum: $A \cap B = 0 \implies A \cup B \approx A + B$
 unfolding epoll-def
 apply (rule-tac $x = \lambda a \in A \cup B. \text{if } a \in A \text{ then } \text{Inl } (a) \text{ else } \text{Inr } (a) \text{ in } exI$)
 apply (rule-tac $d = \lambda z. \text{case } (\lambda x. x, \lambda x. x, z) \text{ in } \text{lam-bijective}$)
 apply auto
 done

23.8 Finite and infinite sets

lemma eqpoll-imp-Finite-iff: $A \approx B \implies \text{Finite}(A) \longleftrightarrow \text{Finite}(B)$
 unfolding Finite-def
 apply (blast intro: eqpoll-trans eqpoll-sym)
 done

lemma Finite-0 [simp]: $\text{Finite}(0)$
 unfolding Finite-def
 apply (blast intro!: eqpoll-refl nat-0I)
 done

lemma Finite-cons: $\text{Finite}(x) \implies \text{Finite}(\text{cons}(y, x))$
 unfolding Finite-def
 apply (case-tac $y \in x$)
 apply (simp add: cons-absorb)
 apply (erule bexE)
 apply (rule bexI)

apply (*erule-tac* [2] *nat-succI*)
apply (*simp* (*no-asm-simp*) *add: succ-def cons-epoll-cong mem-not-refl*)
done

lemma *Finite-succ*: $Finite(x) \implies Finite(succ(x))$
unfolding *succ-def*
apply (*erule Finite-cons*)
done

lemma *lepoll-nat-imp-Finite*:
assumes *A*: $A \lesssim n$ **and** *n*: $n \in nat$ **shows** $Finite(A)$
proof –
have $A \lesssim n \implies Finite(A)$ **using** *n*
proof (*induct n*)
case 0
hence $A = 0$ **by** (*rule lepoll-0-is-0*)
thus ?*case* **by** *simp*
next
case (*succ n*)
hence $A \lesssim n \vee A \approx succ(n)$ **by** (*blast dest: lepoll-succ-disj*)
thus ?*case* **using** *succ* **by** (*auto simp add: Finite-def*)
qed
thus ?*thesis* **using** *A* .
qed

lemma *lesspoll-nat-is-Finite*:
 $A \prec nat \implies Finite(A)$
unfolding *Finite-def*
apply (*blast dest: ltD lesspoll-cardinal-lt*
lesspoll-imp-epoll [THEN epoll-sym])
done

lemma *lepoll-Finite*:
assumes *Y*: $Y \lesssim X$ **and** *X*: $Finite(X)$ **shows** $Finite(Y)$
proof –
obtain *n* **where** $n: n \in nat$ $X \approx n$ **using** *X*
by (*auto simp add: Finite-def*)
have $Y \lesssim X$ **by** (*rule Y*)
also have $\dots \approx n$ **by** (*rule n*)
finally have $Y \lesssim n$.
thus ?*thesis* **using** *n* **by** (*simp add: lepoll-nat-imp-Finite*)
qed

lemmas *subset-Finite* = *subset-imp-lepoll [THEN lepoll-Finite]*

lemma *Finite-cons-iff* [*iff*]: $Finite(cons(y,x)) \longleftrightarrow Finite(x)$
by (*blast intro: Finite-cons subset-Finite*)

lemma *Finite-succ-iff* [*iff*]: $Finite(succ(x)) \longleftrightarrow Finite(x)$

by (*simp add: succ-def*)

lemma *Finite-Int*: $Finite(A) \mid Finite(B) \implies Finite(A \cap B)$
by (*blast intro: subset-Finite*)

lemmas *Finite-Diff* = *Diff-subset* [*THEN subset-Finite*]

lemma *nat-le-infinite-Ord*:
 $\llbracket Ord(i); \neg Finite(i) \rrbracket \implies nat \leq i$
unfolding *Finite-def*
apply (*erule Ord-nat* [*THEN* [2] *Ord-linear2*])
prefer 2 **apply** *assumption*
apply (*blast intro!: eqpoll-refl elim!: ltE*)
done

lemma *Finite-imp-well-ord*:
 $Finite(A) \implies \exists r. well_ord(A, r)$
unfolding *Finite-def eqpoll-def*
apply (*blast intro: well-ord-rvimage bij-is-inj well-ord-Memrel nat-into-Ord*)
done

lemma *succ-lepoll-imp-not-empty*: $succ(x) \lesssim y \implies y \neq 0$
by (*fast dest!: lepoll-0-is-0*)

lemma *eqpoll-succ-imp-not-empty*: $x \approx succ(n) \implies x \neq 0$
by (*fast elim!: eqpoll-sym* [*THEN eqpoll-0-is-0*, *THEN succ-neq-0*])

lemma *Finite-Fin-lemma* [*rule-format*]:
 $n \in nat \implies \forall A. (A \approx n \wedge A \subseteq X) \longrightarrow A \in Fin(X)$
apply (*induct-tac n*)
apply (*rule allI*)
apply (*fast intro!: Fin.emptyI dest!: eqpoll-imp-lepoll* [*THEN lepoll-0-is-0*])
apply (*rule allI*)
apply (*rule impI*)
apply (*erule conjE*)
apply (*rule eqpoll-succ-imp-not-empty* [*THEN not-emptyE*], *assumption*)
apply (*frule Diff-sing-eqpoll*, *assumption*)
apply (*erule allE*)
apply (*erule impE*, *fast*)
apply (*drule subsetD*, *assumption*)
apply (*drule Fin.consI*, *assumption*)
apply (*simp add: cons-Diff*)
done

lemma *Finite-Fin*: $\llbracket Finite(A); A \subseteq X \rrbracket \implies A \in Fin(X)$
by (*unfold Finite-def*, *blast intro: Finite-Fin-lemma*)

lemma *Fin-lemma* [*rule-format*]: $n \in nat \implies \forall A. A \approx n \longrightarrow A \in Fin(A)$
apply (*induct-tac n*)

```

apply (simp add: eqpoll-0-iff, clarify)
apply (subgoal-tac  $\exists u. u \in A$ )
apply (erule exE)
apply (rule Diff-sing-eqpoll [elim-format])
prefer 2 apply assumption
apply assumption
apply (rule-tac  $b = A$  in cons-Diff [THEN subst], assumption)
apply (rule Fin.consI, blast)
apply (blast intro: subset-consI [THEN Fin-mono, THEN subsetD])

  unfolding eqpoll-def
apply (blast intro: bij-converse-bij [THEN bij-is-fun, THEN apply-type])
done

lemma Finite-into-Fin:  $\text{Finite}(A) \implies A \in \text{Fin}(A)$ 
  unfolding Finite-def
apply (blast intro: Fin-lemma)
done

lemma Fin-into-Finite:  $A \in \text{Fin}(U) \implies \text{Finite}(A)$ 
by (fast intro!: Finite-0 Finite-cons elim: Fin-induct)

lemma Finite-Fin-iff:  $\text{Finite}(A) \longleftrightarrow A \in \text{Fin}(A)$ 
by (blast intro: Finite-into-Fin Fin-into-Finite)

lemma Finite-Un:  $\llbracket \text{Finite}(A); \text{Finite}(B) \rrbracket \implies \text{Finite}(A \cup B)$ 
by (blast intro!: Fin-into-Finite Fin-UnI
  dest!: Finite-into-Fin
  intro: Un-upper1 [THEN Fin-mono, THEN subsetD]
  Un-upper2 [THEN Fin-mono, THEN subsetD])

lemma Finite-Un-iff [simp]:  $\text{Finite}(A \cup B) \longleftrightarrow (\text{Finite}(A) \wedge \text{Finite}(B))$ 
by (blast intro: subset-Finite Finite-Un)

The converse must hold too.

lemma Finite-Union:  $\llbracket \forall y \in X. \text{Finite}(y); \text{Finite}(X) \rrbracket \implies \text{Finite}(\bigcup(X))$ 
apply (simp add: Finite-Fin-iff)
apply (rule Fin-UnionI)
apply (erule Fin-induct, simp)
apply (blast intro: Fin.consI Fin-mono [THEN [2] rev-subsetD])
done

lemma Finite-induct [case-names 0 cons, induct set: Finite]:
 $\llbracket \text{Finite}(A); P(0);$ 
 $\bigwedge x B. \llbracket \text{Finite}(B); x \notin B; P(B) \rrbracket \implies P(\text{cons}(x, B)) \rrbracket$ 
 $\implies P(A)$ 
apply (erule Finite-into-Fin [THEN Fin-induct])
apply (blast intro: Fin-into-Finite) +

```

done

lemma *Diff-sing-Finite*: $Finite(A - \{a\}) \implies Finite(A)$
unfolding *Finite-def*
apply (*case-tac* $a \in A$)
apply (*subgoal-tac* [2] $A - \{a\} = A$, *auto*)
apply (*rule-tac* $x = succ\ (n)$ **in** *bexI*)
apply (*subgoal-tac* $cons\ (a, A - \{a\}) = A \wedge cons\ (n, n) = succ\ (n)$)
apply (*drule-tac* $a = a$ **and** $b = n$ **in** *cons-egpoll-cong*)
apply (*auto dest: mem-irrefl*)
done

lemma *Diff-Finite* [*rule-format*]: $Finite(B) \implies Finite(A-B) \longrightarrow Finite(A)$
apply (*erule Finite-induct, auto*)
apply (*case-tac* $x \in A$)
apply (*subgoal-tac* [2] $A - cons\ (x, B) = A - B$)
apply (*subgoal-tac* $A - cons\ (x, B) = (A - B) - \{x\}$, *simp*)
apply (*drule Diff-sing-Finite, auto*)
done

lemma *Finite-RepFun*: $Finite(A) \implies Finite(RepFun(A,f))$
by (*erule Finite-induct, simp-all*)

lemma *Finite-RepFun-iff-lemma* [*rule-format*]:

$$\llbracket Finite(x); \bigwedge x\ y. f(x)=f(y) \implies x=y \rrbracket$$

$$\implies \forall A. x = RepFun(A,f) \longrightarrow Finite(A)$$

apply (*erule Finite-induct*)
apply *clarify*
apply (*case-tac* $A=0$, *simp*)
apply (*blast del: allE, clarify*)
apply (*subgoal-tac* $\exists z \in A. x = f(z)$)
prefer 2 **apply** (*blast del: allE elim: equalityE, clarify*)
apply (*subgoal-tac* $B = \{f(u) . u \in A - \{z\}\}$)
apply (*blast intro: Diff-sing-Finite*)
apply (*thin-tac* $\forall A. P(A) \longrightarrow Finite(A)$ **for** P)
apply (*rule equalityI*)
apply (*blast intro: elim: equalityE*)
apply (*blast intro: elim: equalityCE*)
done

I don't know why, but if the premise is expressed using meta-connectives then the simplifier cannot prove it automatically in conditional rewriting.

lemma *Finite-RepFun-iff*:

$$(\forall x\ y. f(x)=f(y) \longrightarrow x=y) \implies Finite(RepFun(A,f)) \longleftrightarrow Finite(A)$$

by (*blast intro: Finite-RepFun Finite-RepFun-iff-lemma [of - f]*)

lemma *Finite-Pow*: $Finite(A) \implies Finite(Pow(A))$

```

apply (erule Finite-induct)
apply (simp-all add: Pow-insert Finite-Un Finite-RepFun)
done

```

```

lemma Finite-Pow-imp-Finite:  $Finite(Pow(A)) \implies Finite(A)$ 
apply (subgoal-tac Finite({{x} . x ∈ A}))
apply (simp add: Finite-RepFun-iff)
apply (blast intro: subset-Finite)
done

```

```

lemma Finite-Pow-iff [iff]:  $Finite(Pow(A)) \longleftrightarrow Finite(A)$ 
by (blast intro: Finite-Pow Finite-Pow-imp-Finite)

```

```

lemma Finite-cardinal-iff:
  assumes i:  $Ord(i)$  shows  $Finite(|i|) \longleftrightarrow Finite(i)$ 
  by (auto simp add: Finite-def) (blast intro: eqpoll-trans eqpoll-sym Ord-cardinal-eqpoll
  [OF i])+

```

```

lemma nat-wf-on-converse-Memrel:  $n \in nat \implies wf[n](converse(Memrel(n)))$ 
proof (induct n rule: nat-induct)
  case 0 thus ?case by (blast intro: wf-onI)
next
  case (succ x)
  hence wfx:  $\bigwedge Z. Z = 0 \vee (\exists z \in Z. \forall y. z \in y \wedge z \in x \wedge y \in x \wedge z \in x \longrightarrow y \notin Z)$ 
  by (simp add: wf-on-def wf-def) — not easy to erase the duplicate  $z \in x$ !
  show ?case
  proof (rule wf-onI)
    fix Z u
    assume Z:  $u \in Z \wedge \forall z \in Z. \exists y \in Z. \langle y, z \rangle \in converse(Memrel(succ(x)))$ 
    show False
    proof (cases x ∈ Z)
      case True thus False using Z
      by (blast elim: mem-irrefl mem-asym)
    next
      case False thus False using wfx [of Z] Z
      by blast
    qed
  qed
qed

```

```

lemma nat-well-ord-converse-Memrel:  $n \in nat \implies well\_ord(n, converse(Memrel(n)))$ 
apply (frule Ord-nat [THEN Ord-in-Ord, THEN well-ord-Memrel])
apply (simp add: well-ord-def tot-ord-converse nat-wf-on-converse-Memrel)
done

```



```

lemma well-ord-converse:
  [[well-ord(A,r);
    well-ord(ordertype(A,r), converse(Memrel(ordertype(A, r))))]]
   $\implies$  well-ord(A,converse(r))
apply (rule well-ord-Int-iff [THEN iffD1])
apply (frule ordermap-bij [THEN bij-is-inj, THEN well-ord-rvimage], assumption)
apply (simp add: rvimage-converse converse-Int converse-prod
        ordertype-ord-iso [THEN ord-iso-rvimage-eq])
done

lemma ordertype-eq-n:
  assumes r: well-ord(A,r) and A: A  $\approx$  n and n: n  $\in$  nat
  shows ordertype(A,r) = n
proof –
  have ordertype(A,r)  $\approx$  A
    by (blast intro: bij-imp-epoll bij-converse-bij ordermap-bij r)
  also have ...  $\approx$  n by (rule A)
  finally have ordertype(A,r)  $\approx$  n .
  thus ?thesis
    by (simp add: Ord-nat-epoll-iff Ord-ordertype n r)
qed

lemma Finite-well-ord-converse:
  [[Finite(A); well-ord(A,r)]]  $\implies$  well-ord(A,converse(r))
  unfolding Finite-def
apply (rule well-ord-converse, assumption)
apply (blast dest: ordertype-eq-n intro!: nat-well-ord-converse-Memrel)
done

lemma nat-into-Finite: n  $\in$  nat  $\implies$  Finite(n)
  by (auto simp add: Finite-def intro: eqpoll-refl)

lemma nat-not-Finite:  $\neg$  Finite(nat)
proof –
  { fix n
    assume n: n  $\in$  nat nat  $\approx$  n
    have n  $\in$  nat by (rule n)
    also have ... = n using n
      by (simp add: Ord-nat-epoll-iff Ord-nat)
    finally have n  $\in$  n .
    hence False
      by (blast elim: mem-irrefl)
  }
  thus ?thesis
    by (auto simp add: Finite-def)
qed

end

```

24 The Cumulative Hierarchy and a Small Universe for Recursive Types

theory *Univ* **imports** *Epsilon Cardinal* **begin**

definition

$Vfrom \quad :: [i, i] \Rightarrow i$ **where**
 $Vfrom(A, i) \equiv transrec(i, \lambda x f. A \cup (\bigcup y \in x. Pow(f'y)))$

abbreviation

$Vset :: i \Rightarrow i$ **where**
 $Vset(x) \equiv Vfrom(0, x)$

definition

$Vrec \quad :: [i, [i, i] \Rightarrow i] \Rightarrow i$ **where**
 $Vrec(a, H) \equiv transrec(rank(a), \lambda x g. \lambda z \in Vset(succ(x)).$
 $\quad H(z, \lambda w \in Vset(x). g'rank(w)'w)) \text{ ' } a$

definition

$Vrecursor :: [[i, i] \Rightarrow i, i] \Rightarrow i$ **where**
 $Vrecursor(H, a) \equiv transrec(rank(a), \lambda x g. \lambda z \in Vset(succ(x)).$
 $\quad H(\lambda w \in Vset(x). g'rank(w)'w, z)) \text{ ' } a$

definition

$univ \quad :: i \Rightarrow i$ **where**
 $univ(A) \equiv Vfrom(A, nat)$

24.1 Immediate Consequences of the Definition of $Vfrom(A, i)$

NOT SUITABLE FOR REWRITING – RECURSIVE!

lemma $Vfrom$: $Vfrom(A, i) = A \cup (\bigcup j \in i. Pow(Vfrom(A, j)))$
by (*subst* $Vfrom$ -def [*THEN* def-transrec], *simp*)

24.1.1 Monotonicity

lemma $Vfrom$ -mono [*rule-format*]:

$A \leq B \implies \forall j. i \leq j \longrightarrow Vfrom(A, i) \subseteq Vfrom(B, j)$

apply (*rule-tac* $a=i$ **in** *eps-induct*)

apply (*rule impI* [*THEN allI*])

apply (*subst* $Vfrom$ [*of* A])

apply (*subst* $Vfrom$ [*of* B])

apply (*erule* Un -mono)

apply (*erule* UN -mono, *blast*)

done

lemma $VfromI$: $\llbracket a \in Vfrom(A, j); j < i \rrbracket \implies a \in Vfrom(A, i)$

by (*blast* *dest*: $Vfrom$ -mono [*OF* subset-refl *le-imp-subset* [*OF leI*]])

24.1.2 A fundamental equality: Vfrom does not require ordinals!

lemma *Vfrom-rank-subset1*: $Vfrom(A, x) \subseteq Vfrom(A, rank(x))$

proof (*induct x rule: eps-induct*)

fix x

assume $\forall y \in x. Vfrom(A, y) \subseteq Vfrom(A, rank(y))$

thus $Vfrom(A, x) \subseteq Vfrom(A, rank(x))$

by (*simp add: Vfrom [of - x] Vfrom [of - rank(x)],*
blast intro!: rank-lt [THEN ltD])

qed

lemma *Vfrom-rank-subset2*: $Vfrom(A, rank(x)) \subseteq Vfrom(A, x)$

apply (*rule-tac a=x in eps-induct*)

apply (*subst Vfrom*)

apply (*subst Vfrom, rule subset-refl [THEN Un-mono]*)

apply (*rule UN-least*)

expand $rank(x1) = (\bigcup y \in x1. succ(rank(y)))$ in assumptions

apply (*erule rank [THEN equalityD1, THEN subsetD, THEN UN-E]*)

apply (*rule subset-trans*)

apply (*erule-tac [2] UN-upper*)

apply (*rule subset-refl [THEN Vfrom-mono, THEN subset-trans, THEN Pow-mono]*)

apply (*erule ltI [THEN le-imp-subset]*)

apply (*rule Ord-rank [THEN Ord-succ]*)

apply (*erule bspec, assumption*)

done

lemma *Vfrom-rank-eq*: $Vfrom(A, rank(x)) = Vfrom(A, x)$

apply (*rule equalityI*)

apply (*rule Vfrom-rank-subset2*)

apply (*rule Vfrom-rank-subset1*)

done

24.2 Basic Closure Properties

lemma *zero-in-Vfrom*: $y:x \implies 0 \in Vfrom(A, x)$

by (*subst Vfrom, blast*)

lemma *i-subset-Vfrom*: $i \subseteq Vfrom(A, i)$

apply (*rule-tac a=i in eps-induct*)

apply (*subst Vfrom, blast*)

done

lemma *A-subset-Vfrom*: $A \subseteq Vfrom(A, i)$

apply (*subst Vfrom*)

apply (*rule Un-upper1*)

done

lemmas *A-into-Vfrom* = *A-subset-Vfrom* [*THEN subsetD*]

lemma *subset-mem-Vfrom*: $a \subseteq Vfrom(A, i) \implies a \in Vfrom(A, succ(i))$
by (*subst Vfrom, blast*)

24.2.1 Finite sets and ordered pairs

lemma *singleton-in-Vfrom*: $a \in Vfrom(A, i) \implies \{a\} \in Vfrom(A, succ(i))$
by (*rule subset-mem-Vfrom, safe*)

lemma *doubleton-in-Vfrom*:
 $\llbracket a \in Vfrom(A, i); b \in Vfrom(A, i) \rrbracket \implies \{a, b\} \in Vfrom(A, succ(i))$
by (*rule subset-mem-Vfrom, safe*)

lemma *Pair-in-Vfrom*:
 $\llbracket a \in Vfrom(A, i); b \in Vfrom(A, i) \rrbracket \implies \langle a, b \rangle \in Vfrom(A, succ(succ(i)))$
unfolding *Pair-def*
apply (*blast intro: doubleton-in-Vfrom*)
done

lemma *succ-in-Vfrom*: $a \subseteq Vfrom(A, i) \implies succ(a) \in Vfrom(A, succ(succ(i)))$
apply (*intro subset-mem-Vfrom succ-subsetI, assumption*)
apply (*erule subset-trans*)
apply (*rule Vfrom-mono [OF subset-refl subset-succI]*)
done

24.3 0, Successor and Limit Equations for *Vfrom*

lemma *Vfrom-0*: $Vfrom(A, 0) = A$
by (*subst Vfrom, blast*)

lemma *Vfrom-succ-lemma*: $Ord(i) \implies Vfrom(A, succ(i)) = A \cup Pow(Vfrom(A, i))$
apply (*rule Vfrom [THEN trans]*)
apply (*rule equalityI [THEN subst-context,*
 $OF - succI1 [THEN RepFunI, THEN Union-upper]]$)
apply (*rule UN-least*)
apply (*rule subset-refl [THEN Vfrom-mono, THEN Pow-mono]*)
apply (*erule ltI [THEN le-imp-subset]*)
apply (*erule Ord-succ*)
done

lemma *Vfrom-succ*: $Vfrom(A, succ(i)) = A \cup Pow(Vfrom(A, i))$
apply (*rule-tac x1 = succ (i) in Vfrom-rank-eq [THEN subst]*)
apply (*rule-tac x1 = i in Vfrom-rank-eq [THEN subst]*)
apply (*subst rank-succ*)
apply (*rule Ord-rank [THEN Vfrom-succ-lemma]*)
done

lemma *Vfrom-Union*: $y:X \implies Vfrom(A, \bigcup(X)) = (\bigcup_{y \in X}. Vfrom(A, y))$
apply (*subst Vfrom*)
apply (*rule equalityI*)

first inclusion

```

apply (rule Un-least)
apply (rule A-subset-Vfrom [THEN subset-trans])
apply (rule UN-upper, assumption)
apply (rule UN-least)
apply (erule UnionE)
apply (rule subset-trans)
apply (erule-tac [2] UN-upper,
      subst Vfrom, erule subset-trans [OF UN-upper Un-upper2])

```

opposite inclusion

```

apply (rule UN-least)
apply (subst Vfrom, blast)
done

```

24.4 *Vfrom* applied to Limit Ordinals

lemma *Limit-Vfrom-eq*:

```

       $\text{Limit}(i) \implies \text{Vfrom}(A, i) = (\bigcup_{y \in i} \text{Vfrom}(A, y))$ 
apply (rule Limit-has-0 [THEN ltD, THEN Vfrom-Union, THEN subst], assumption)
apply (simp add: Limit-Union-eq)
done

```

lemma *Limit-VfromE*:

```

       $\llbracket a \in \text{Vfrom}(A, i); \neg R \implies \text{Limit}(i);$ 
       $\bigwedge x. \llbracket x < i; a \in \text{Vfrom}(A, x) \rrbracket \implies R$ 
]]  $\implies R$ 
apply (rule classical)
apply (rule Limit-Vfrom-eq [THEN equalityD1, THEN subsetD, THEN UN-E])
      prefer 2 apply assumption
apply blast
apply (blast intro: ltI Limit-is-Ord)
done

```

lemma *singleton-in-VLimit*:

```

       $\llbracket a \in \text{Vfrom}(A, i); \text{Limit}(i) \rrbracket \implies \{a\} \in \text{Vfrom}(A, i)$ 
apply (erule Limit-VfromE, assumption)
apply (erule singleton-in-Vfrom [THEN VfromI])
apply (blast intro: Limit-has-succ)
done

```

lemmas *Vfrom-UnI1* =

Un-upper1 [*THEN subset-refl* [*THEN Vfrom-mono*, *THEN subsetD*]]

lemmas *Vfrom-UnI2* =

Un-upper2 [*THEN subset-refl* [*THEN Vfrom-mono*, *THEN subsetD*]]

Hard work is finding a single $j:i$ such that $a, b \leq \text{Vfrom}(A, j)$

lemma *doubleton-in-VLimit*:

$\llbracket a \in Vfrom(A,i); b \in Vfrom(A,i); Limit(i) \rrbracket \implies \{a,b\} \in Vfrom(A,i)$
apply (*erule Limit-VfromE, assumption*)
apply (*erule Limit-VfromE, assumption*)
apply (*blast intro: VfromI [OF doubleton-in-Vfrom]*
 $Vfrom-UnI1 \ Vfrom-UnI2 \ Limit-has-succ \ Un-least-lt$)
done

lemma *Pair-in-VLimit:*

$\llbracket a \in Vfrom(A,i); b \in Vfrom(A,i); Limit(i) \rrbracket \implies \langle a,b \rangle \in Vfrom(A,i)$

Infer that a, b occur at ordinals $x, x_a < i$.

apply (*erule Limit-VfromE, assumption*)
apply (*erule Limit-VfromE, assumption*)

Infer that $succ(succ(x \cup x_a)) < i$

apply (*blast intro: VfromI [OF Pair-in-Vfrom]*
 $Vfrom-UnI1 \ Vfrom-UnI2 \ Limit-has-succ \ Un-least-lt$)
done

lemma *product-VLimit:* $Limit(i) \implies Vfrom(A,i) * Vfrom(A,i) \subseteq Vfrom(A,i)$
by (*blast intro: Pair-in-VLimit*)

lemmas *Sigma-subset-VLimit =*

subset-trans [OF Sigma-mono product-VLimit]

lemmas *nat-subset-VLimit =*

subset-trans [OF nat-le-Limit [THEN le-imp-subset] i-subset-Vfrom]

lemma *nat-into-VLimit:* $\llbracket n: nat; Limit(i) \rrbracket \implies n \in Vfrom(A,i)$
by (*blast intro: nat-subset-VLimit [THEN subsetD]*)

24.4.1 Closure under Disjoint Union

lemmas *zero-in-VLimit = Limit-has-0 [THEN ltD, THEN zero-in-Vfrom]*

lemma *one-in-VLimit:* $Limit(i) \implies 1 \in Vfrom(A,i)$
by (*blast intro: nat-into-VLimit*)

lemma *Inl-in-VLimit:*

$\llbracket a \in Vfrom(A,i); Limit(i) \rrbracket \implies Inl(a) \in Vfrom(A,i)$

unfolding *Inl-def*

apply (*blast intro: zero-in-VLimit Pair-in-VLimit*)
done

lemma *Inr-in-VLimit:*

$\llbracket b \in Vfrom(A,i); Limit(i) \rrbracket \implies Inr(b) \in Vfrom(A,i)$

unfolding *Inr-def*

apply (*blast intro: one-in-VLimit Pair-in-VLimit*)
done

lemma *sum-VLimit*: $\text{Limit}(i) \implies \text{Vfrom}(C, i) + \text{Vfrom}(C, i) \subseteq \text{Vfrom}(C, i)$
by (*blast intro!*: *Inl-in-VLimit Inr-in-VLimit*)

lemmas *sum-subset-VLimit* = *subset-trans* [*OF sum-mono sum-VLimit*]

24.5 Properties assuming $\text{Transset}(A)$

lemma *Transset-Vfrom*: $\text{Transset}(A) \implies \text{Transset}(\text{Vfrom}(A, i))$
apply (*rule-tac a=i in eps-induct*)
apply (*subst Vfrom*)
apply (*blast intro!*: *Transset-Union-family Transset-Un Transset-Pow*)
done

lemma *Transset-Vfrom-succ*:
 $\text{Transset}(A) \implies \text{Vfrom}(A, \text{succ}(i)) = \text{Pow}(\text{Vfrom}(A, i))$
apply (*rule Vfrom-succ [THEN trans]*)
apply (*rule equalityI [OF - Un-upper2]*)
apply (*rule Un-least [OF - subset-refl]*)
apply (*rule A-subset-Vfrom [THEN subset-trans]*)
apply (*erule Transset-Vfrom [THEN Transset-iff-Pow [THEN iffD1]]*)
done

lemma *Transset-Pair-subset*: $\llbracket \langle a, b \rangle \subseteq C; \text{Transset}(C) \rrbracket \implies a: C \wedge b: C$
by (*unfold Pair-def Transset-def, blast*)

lemma *Transset-Pair-subset-VLimit*:
 $\llbracket \langle a, b \rangle \subseteq \text{Vfrom}(A, i); \text{Transset}(A); \text{Limit}(i) \rrbracket$
 $\implies \langle a, b \rangle \in \text{Vfrom}(A, i)$
apply (*erule Transset-Pair-subset [THEN conjE]*)
apply (*erule Transset-Vfrom*)
apply (*blast intro: Pair-in-VLimit*)
done

lemma *Union-in-Vfrom*:
 $\llbracket X \in \text{Vfrom}(A, j); \text{Transset}(A) \rrbracket \implies \bigcup(X) \in \text{Vfrom}(A, \text{succ}(j))$
apply (*drule Transset-Vfrom*)
apply (*rule subset-mem-Vfrom*)
apply (*unfold Transset-def, blast*)
done

lemma *Union-in-VLimit*:
 $\llbracket X \in \text{Vfrom}(A, i); \text{Limit}(i); \text{Transset}(A) \rrbracket \implies \bigcup(X) \in \text{Vfrom}(A, i)$
apply (*rule Limit-VfromE, assumption+*)
apply (*blast intro: Limit-has-succ VfromI Union-in-Vfrom*)
done

General theorem for membership in $\text{Vfrom}(A, i)$ when i is a limit ordinal

lemma *in-VLimit*:

$$\begin{aligned}
& \llbracket a \in V_{\text{from}}(A, i); \ b \in V_{\text{from}}(A, i); \ \text{Limit}(i); \\
& \quad \wedge x \ y \ j. \ \llbracket j < i; \ 1:j; \ x \in V_{\text{from}}(A, j); \ y \in V_{\text{from}}(A, j) \rrbracket \\
& \quad \implies \exists k. \ h(x, y) \in V_{\text{from}}(A, k) \wedge k < i \rrbracket \\
& \implies h(a, b) \in V_{\text{from}}(A, i)
\end{aligned}$$

Infer that a, b occur at ordinals x, xa < i.

```

apply (erule Limit-VfromE, assumption)
apply (erule Limit-VfromE, assumption, atomize)
apply (drule-tac x=a in spec)
apply (drule-tac x=b in spec)
apply (drule-tac x=x  $\cup$  xa  $\cup$  2 in spec)
apply (simp add: Un-least-lt-iff lt-Ord Vfrom-UnI1 Vfrom-UnI2)
apply (blast intro: Limit-has-0 Limit-has-succ VfromI)
done

```

24.5.1 Products

```

lemma prod-in-Vfrom:
   $\llbracket a \in V_{\text{from}}(A, j); \ b \in V_{\text{from}}(A, j); \ \text{Transset}(A) \rrbracket$ 
   $\implies a * b \in V_{\text{from}}(A, \text{succ}(\text{succ}(\text{succ}(j))))$ 
apply (drule Transset-Vfrom)
apply (rule subset-mem-Vfrom)
  unfolding Transset-def
apply (blast intro: Pair-in-Vfrom)
done

```

```

lemma prod-in-VLimit:
   $\llbracket a \in V_{\text{from}}(A, i); \ b \in V_{\text{from}}(A, i); \ \text{Limit}(i); \ \text{Transset}(A) \rrbracket$ 
   $\implies a * b \in V_{\text{from}}(A, i)$ 
apply (erule in-VLimit, assumption+)
apply (blast intro: prod-in-Vfrom Limit-has-succ)
done

```

24.5.2 Disjoint Sums, or Quine Ordered Pairs

```

lemma sum-in-Vfrom:
   $\llbracket a \in V_{\text{from}}(A, j); \ b \in V_{\text{from}}(A, j); \ \text{Transset}(A); \ 1:j \rrbracket$ 
   $\implies a + b \in V_{\text{from}}(A, \text{succ}(\text{succ}(\text{succ}(j))))$ 
  unfolding sum-def
apply (drule Transset-Vfrom)
apply (rule subset-mem-Vfrom)
  unfolding Transset-def
apply (blast intro: zero-in-Vfrom Pair-in-Vfrom i-subset-Vfrom [THEN subsetD])
done

```

```

lemma sum-in-VLimit:
   $\llbracket a \in V_{\text{from}}(A, i); \ b \in V_{\text{from}}(A, i); \ \text{Limit}(i); \ \text{Transset}(A) \rrbracket$ 
   $\implies a + b \in V_{\text{from}}(A, i)$ 
apply (erule in-VLimit, assumption+)
apply (blast intro: sum-in-Vfrom Limit-has-succ)

```


done

24.5.3 Function Space!

lemma *fun-in-Vfrom*:
 $\llbracket a \in Vfrom(A, j); b \in Vfrom(A, j); Transset(A) \rrbracket \implies$
 $a \rightarrow b \in Vfrom(A, succ(succ(succ(succ(j)))))$
unfolding *Pi-def*
apply (*drule Transset-Vfrom*)
apply (*rule subset-mem-Vfrom*)
apply (*rule Collect-subset [THEN subset-trans]*)
apply (*subst Vfrom*)
apply (*rule subset-trans [THEN subset-trans]*)
apply (*rule-tac [3] Un-upper2*)
apply (*rule-tac [2] succI1 [THEN UN-upper]*)
apply (*rule Pow-mono*)
unfolding *Transset-def*
apply (*blast intro: Pair-in-Vfrom*)
done

lemma *fun-in-VLimit*:
 $\llbracket a \in Vfrom(A, i); b \in Vfrom(A, i); Limit(i); Transset(A) \rrbracket$
 $\implies a \rightarrow b \in Vfrom(A, i)$
apply (*erule in-VLimit, assumption+*)
apply (*blast intro: fun-in-Vfrom Limit-has-succ*)
done

lemma *Pow-in-Vfrom*:
 $\llbracket a \in Vfrom(A, j); Transset(A) \rrbracket \implies Pow(a) \in Vfrom(A, succ(succ(j)))$
apply (*drule Transset-Vfrom*)
apply (*rule subset-mem-Vfrom*)
unfolding *Transset-def*
apply (*subst Vfrom, blast*)
done

lemma *Pow-in-VLimit*:
 $\llbracket a \in Vfrom(A, i); Limit(i); Transset(A) \rrbracket \implies Pow(a) \in Vfrom(A, i)$
by (*blast elim: Limit-VfromE intro: Limit-has-succ Pow-in-Vfrom VfromI*)

24.6 The Set $Vset(i)$

lemma *Vset*: $Vset(i) = (\bigcup j \in i. Pow(Vset(j)))$
by (*subst Vfrom, blast*)

lemmas *Vset-succ* = *Transset-0 [THEN Transset-Vfrom-succ]*
lemmas *Transset-Vset* = *Transset-0 [THEN Transset-Vfrom]*

24.6.1 Characterisation of the elements of $Vset(i)$

lemma *VsetD* [*rule-format*]: $Ord(i) \implies \forall b. b \in Vset(i) \longrightarrow rank(b) < i$

```

apply (erule trans-induct)
apply (subst Vset, safe)
apply (subst rank)
apply (blast intro: ltI UN-succ-least-lt)
done

```

```

lemma VsetI-lemma [rule-format]:
   $\text{Ord}(i) \implies \forall b. \text{rank}(b) \in i \longrightarrow b \in \text{Vset}(i)$ 
apply (erule trans-induct)
apply (rule allI)
apply (subst Vset)
apply (blast intro!: rank-lt [THEN ltD])
done

```

```

lemma VsetI:  $\text{rank}(x) < i \implies x \in \text{Vset}(i)$ 
by (blast intro: VsetI-lemma elim: ltE)

```

Merely a lemma for the next result

```

lemma Vset-Ord-rank-iff:  $\text{Ord}(i) \implies b \in \text{Vset}(i) \longleftrightarrow \text{rank}(b) < i$ 
by (blast intro: VsetD VsetI)

```

```

lemma Vset-rank-iff [simp]:  $b \in \text{Vset}(a) \longleftrightarrow \text{rank}(b) < \text{rank}(a)$ 
apply (rule Vfrom-rank-eq [THEN subst])
apply (rule Ord-rank [THEN Vset-Ord-rank-iff])
done

```

This is $\text{rank}(\text{rank}(a)) = \text{rank}(a)$

```

declare Ord-rank [THEN rank-of-Ord, simp]

```

```

lemma rank-Vset:  $\text{Ord}(i) \implies \text{rank}(\text{Vset}(i)) = i$ 
apply (subst rank)
apply (rule equalityI, safe)
apply (blast intro: VsetD [THEN ltD])
apply (blast intro: VsetD [THEN ltD] Ord-trans)
apply (blast intro: i-subset-Vfrom [THEN subsetD])
      Ord-in-Ord [THEN rank-of-Ord, THEN ssubst]
done

```

```

lemma Finite-Vset:  $i \in \text{nat} \implies \text{Finite}(\text{Vset}(i))$ 
apply (erule nat-induct)
  apply (simp add: Vfrom-0)
apply (simp add: Vset-succ)
done

```

24.6.2 Reasoning about Sets in Terms of Their Elements' Ranks

```

lemma arg-subset-Vset-rank:  $a \subseteq \text{Vset}(\text{rank}(a))$ 
apply (rule subsetI)
apply (erule rank-lt [THEN VsetI])

```

done

lemma *Int-Vset-subset*:

$\llbracket \bigwedge i. \text{Ord}(i) \implies a \cap \text{Vset}(i) \subseteq b \rrbracket \implies a \subseteq b$

apply (*rule subset-trans*)

apply (*rule Int-greatest* [*OF subset-refl arg-subset-Vset-rank*])

apply (*blast intro: Ord-rank*)

done

24.6.3 Set Up an Environment for Simplification

lemma *rank-Inl*: $\text{rank}(a) < \text{rank}(\text{Inl}(a))$

unfolding *Inl-def*

apply (*rule rank-pair2*)

done

lemma *rank-Inr*: $\text{rank}(a) < \text{rank}(\text{Inr}(a))$

unfolding *Inr-def*

apply (*rule rank-pair2*)

done

lemmas *rank-rls* = *rank-Inl rank-Inr rank-pair1 rank-pair2*

24.6.4 Recursion over Vset Levels!

NOT SUITABLE FOR REWRITING: recursive!

lemma *Vrec*: $\text{Vrec}(a, H) = H(a, \lambda x \in \text{Vset}(\text{rank}(a)). \text{Vrec}(x, H))$

unfolding *Vrec-def*

apply (*subst transrec, simp*)

apply (*rule refl* [*THEN lam-cong, THEN subst-context*], *simp add: lt-def*)

done

This form avoids giant explosions in proofs. NOTE the form of the premise!

lemma *def-Vrec*:

$\llbracket \bigwedge x. h(x) \equiv \text{Vrec}(x, H) \rrbracket \implies$

$h(a) = H(a, \lambda x \in \text{Vset}(\text{rank}(a)). h(x))$

apply *simp*

apply (*rule Vrec*)

done

NOT SUITABLE FOR REWRITING: recursive!

lemma *Vrecursor*:

$\text{Vrecursor}(H, a) = H(\lambda x \in \text{Vset}(\text{rank}(a)). \text{Vrecursor}(H, x), a)$

unfolding *Vrecursor-def*

apply (*subst transrec, simp*)

apply (*rule refl* [*THEN lam-cong, THEN subst-context*], *simp add: lt-def*)

done

This form avoids giant explosions in proofs. NOTE the form of the premise!

```

lemma def-Vrecursor:
   $h \equiv Vrecursor(H) \implies h(a) = H(\lambda x \in Vset(rank(a)). h(x), a)$ 
apply simp
apply (rule Vrecursor)
done

```

24.7 The Datatype Universe: $univ(A)$

```

lemma univ-mono:  $A \leq B \implies univ(A) \subseteq univ(B)$ 
  unfolding univ-def
apply (erule Vfrom-mono)
apply (rule subset-refl)
done

```

```

lemma Transset-univ:  $Transset(A) \implies Transset(univ(A))$ 
  unfolding univ-def
apply (erule Transset-Vfrom)
done

```

24.7.1 The Set $univ(A)$ as a Limit

```

lemma univ-eq-UN:  $univ(A) = (\bigcup i \in nat. Vfrom(A, i))$ 
  unfolding univ-def
apply (rule Limit-nat [THEN Limit-Vfrom-eq])
done

```

```

lemma subset-univ-eq-Int:  $c \subseteq univ(A) \implies c = (\bigcup i \in nat. c \cap Vfrom(A, i))$ 
apply (rule subset-UN-iff-eq [THEN iffD1])
apply (erule univ-eq-UN [THEN subst])
done

```

```

lemma univ-Int-Vfrom-subset:
   $\llbracket a \subseteq univ(X);$ 
   $\bigwedge i. i : nat \implies a \cap Vfrom(X, i) \subseteq b \rrbracket$ 
   $\implies a \subseteq b$ 
apply (subst subset-univ-eq-Int, assumption)
apply (rule UN-least, simp)
done

```

```

lemma univ-Int-Vfrom-eq:
   $\llbracket a \subseteq univ(X); \quad b \subseteq univ(X);$ 
   $\bigwedge i. i : nat \implies a \cap Vfrom(X, i) = b \cap Vfrom(X, i) \rrbracket$ 
   $\implies a = b$ 
apply (rule equalityI)
apply (rule univ-Int-Vfrom-subset, assumption)
apply (blast elim: equalityCE)
apply (rule univ-Int-Vfrom-subset, assumption)
apply (blast elim: equalityCE)
done

```

24.8 Closure Properties for $univ(A)$

lemma *zero-in-univ*: $0 \in univ(A)$
 unfolding *univ-def*
apply (*rule nat-0I* [*THEN zero-in-Vfrom*])
done

lemma *zero-subset-univ*: $\{0\} \subseteq univ(A)$
by (*blast intro: zero-in-univ*)

lemma *A-subset-univ*: $A \subseteq univ(A)$
 unfolding *univ-def*
apply (*rule A-subset-Vfrom*)
done

lemmas *A-into-univ* = *A-subset-univ* [*THEN subsetD*]

24.8.1 Closure under Unordered and Ordered Pairs

lemma *singleton-in-univ*: $a: univ(A) \implies \{a\} \in univ(A)$
 unfolding *univ-def*
apply (*blast intro: singleton-in-VLimit Limit-nat*)
done

lemma *doubleton-in-univ*:
 $\llbracket a: univ(A); b: univ(A) \rrbracket \implies \{a,b\} \in univ(A)$
 unfolding *univ-def*
apply (*blast intro: doubleton-in-VLimit Limit-nat*)
done

lemma *Pair-in-univ*:
 $\llbracket a: univ(A); b: univ(A) \rrbracket \implies \langle a,b \rangle \in univ(A)$
 unfolding *univ-def*
apply (*blast intro: Pair-in-VLimit Limit-nat*)
done

lemma *Union-in-univ*:
 $\llbracket X: univ(A); Transset(A) \rrbracket \implies \bigcup(X) \in univ(A)$
 unfolding *univ-def*
apply (*blast intro: Union-in-VLimit Limit-nat*)
done

lemma *product-univ*: $univ(A) * univ(A) \subseteq univ(A)$
 unfolding *univ-def*
apply (*rule Limit-nat* [*THEN product-VLimit*])
done

24.8.2 The Natural Numbers

lemma *nat-subset-univ*: $nat \subseteq univ(A)$

```

  unfolding univ-def
apply (rule i-subset-Vfrom)
done

```

```

lemma nat-into-univ:  $n \in \text{nat} \implies n \in \text{univ}(A)$ 
  by (rule nat-subset-univ [THEN subsetD])

```

24.8.3 Instances for 1 and 2

```

lemma one-in-univ:  $1 \in \text{univ}(A)$ 
  unfolding univ-def
apply (rule Limit-nat [THEN one-in-VLimit])
done

```

unused!

```

lemma two-in-univ:  $2 \in \text{univ}(A)$ 
  by (blast intro: nat-into-univ)

```

```

lemma bool-subset-univ:  $\text{bool} \subseteq \text{univ}(A)$ 
  unfolding bool-def
apply (blast intro!: zero-in-univ one-in-univ)
done

```

```

lemmas bool-into-univ = bool-subset-univ [THEN subsetD]

```

24.8.4 Closure under Disjoint Union

```

lemma Inl-in-univ:  $a: \text{univ}(A) \implies \text{Inl}(a) \in \text{univ}(A)$ 
  unfolding univ-def
apply (erule Inl-in-VLimit [OF - Limit-nat])
done

```

```

lemma Inr-in-univ:  $b: \text{univ}(A) \implies \text{Inr}(b) \in \text{univ}(A)$ 
  unfolding univ-def
apply (erule Inr-in-VLimit [OF - Limit-nat])
done

```

```

lemma sum-univ:  $\text{univ}(C) + \text{univ}(C) \subseteq \text{univ}(C)$ 
  unfolding univ-def
apply (rule Limit-nat [THEN sum-VLimit])
done

```

```

lemmas sum-subset-univ = subset-trans [OF sum-mono sum-univ]

```

```

lemma Sigma-subset-univ:
   $\llbracket A \subseteq \text{univ}(D); \bigwedge x. x \in A \implies B(x) \subseteq \text{univ}(D) \rrbracket \implies \text{Sigma}(A, B) \subseteq \text{univ}(D)$ 
  apply (simp add: univ-def)
  apply (blast intro: Sigma-subset-VLimit del: subsetI)
done

```

24.9 Finite Branching Closure Properties

24.9.1 Closure under Finite Powerset

lemma *Fin-Vfrom-lemma*:

```

  [[b: Fin(Vfrom(A,i)); Limit(i)]] ==> ∃ j. b ⊆ Vfrom(A,j) ∧ j < i
apply (erule Fin-induct)
apply (blast dest!: Limit-has-0, safe)
apply (erule Limit-VfromE, assumption)
apply (blast intro!: Un-least-lt intro: Vfrom-UnI1 Vfrom-UnI2)
done

```

```

lemma Fin-VLimit: Limit(i) ==> Fin(Vfrom(A,i)) ⊆ Vfrom(A,i)
apply (rule subsetI)
apply (drule Fin-Vfrom-lemma, safe)
apply (rule Vfrom [THEN ssubst])
apply (blast dest!: ltD)
done

```

lemmas *Fin-subset-VLimit* = *subset-trans [OF Fin-mono Fin-VLimit]*

```

lemma Fin-univ: Fin(univ(A)) ⊆ univ(A)
  unfolding univ-def
apply (rule Limit-nat [THEN Fin-VLimit])
done

```

24.9.2 Closure under Finite Powers: Functions from a Natural Number

lemma *nat-fun-VLimit*:

```

  [[n: nat; Limit(i)]] ==> n -> Vfrom(A,i) ⊆ Vfrom(A,i)
apply (erule nat-fun-subset-Fin [THEN subset-trans])
apply (blast del: subsetI
  intro: subset-refl Fin-subset-VLimit Sigma-subset-VLimit nat-subset-VLimit)
done

```

lemmas *nat-fun-subset-VLimit* = *subset-trans [OF Pi-mono nat-fun-VLimit]*

```

lemma nat-fun-univ: n: nat ==> n -> univ(A) ⊆ univ(A)
  unfolding univ-def
apply (erule nat-fun-VLimit [OF - Limit-nat])
done

```

24.9.3 Closure under Finite Function Space

General but seldom-used version; normally the domain is fixed

lemma *FiniteFun-VLimit1*:

```

  Limit(i) ==> Vfrom(A,i) -||> Vfrom(A,i) ⊆ Vfrom(A,i)
apply (rule FiniteFun.dom-subset [THEN subset-trans])
apply (blast del: subsetI)

```

intro: Fin-subset-VLimit Sigma-subset-VLimit subset-refl)

done

lemma *FiniteFun-univ1*: $\text{univ}(A) -||> \text{univ}(A) \subseteq \text{univ}(A)$
unfolding *univ-def*
apply (*rule Limit-nat* [*THEN FiniteFun-VLimit1*])
done

Version for a fixed domain

lemma *FiniteFun-VLimit*:
 $\llbracket W \subseteq \text{Vfrom}(A, i); \text{Limit}(i) \rrbracket \implies W -||> \text{Vfrom}(A, i) \subseteq \text{Vfrom}(A, i)$
apply (*rule subset-trans*)
apply (*erule FiniteFun-mono* [*OF - subset-refl*])
apply (*erule FiniteFun-VLimit1*)
done

lemma *FiniteFun-univ*:
 $W \subseteq \text{univ}(A) \implies W -||> \text{univ}(A) \subseteq \text{univ}(A)$
unfolding *univ-def*
apply (*erule FiniteFun-VLimit* [*OF - Limit-nat*])
done

lemma *FiniteFun-in-univ*:
 $\llbracket f: W -||> \text{univ}(A); W \subseteq \text{univ}(A) \rrbracket \implies f \in \text{univ}(A)$
by (*erule FiniteFun-univ* [*THEN subsetD*], *assumption*)

Remove \subseteq from the rule above

lemmas *FiniteFun-in-univ' = FiniteFun-in-univ* [*OF - subsetI*]

24.10 * For QUniv. Properties of Vfrom analogous to the "take-lemma" *

Intersecting $a*b$ with $\text{Vfrom}...$

This version says a, b exist one level down, in the smaller set $\text{Vfrom}(X, i)$

lemma *doubleton-in-Vfrom-D*:
 $\llbracket \{a, b\} \in \text{Vfrom}(X, \text{succ}(i)); \text{Transset}(X) \rrbracket$
 $\implies a \in \text{Vfrom}(X, i) \wedge b \in \text{Vfrom}(X, i)$
by (*drule Transset-Vfrom-succ* [*THEN equalityD1*, *THEN subsetD*, *THEN PowD*],
assumption, *fast*)

This weaker version says a, b exist at the same level

lemmas *Vfrom-doubleton-D = Transset-Vfrom* [*THEN Transset-doubleton-D*]

lemma *Pair-in-Vfrom-D*:
 $\llbracket \langle a, b \rangle \in \text{Vfrom}(X, \text{succ}(i)); \text{Transset}(X) \rrbracket$


```

     $\implies a \in Vfrom(X, i) \wedge b \in Vfrom(X, i)$ 
  unfolding Pair-def
apply (blast dest!: doubleton-in-Vfrom-D Vfrom-doubleton-D)
done

lemma product-Int-Vfrom-subset:
  Transset(X)  $\implies$ 
   $(a * b) \cap Vfrom(X, succ(i)) \subseteq (a \cap Vfrom(X, i)) * (b \cap Vfrom(X, i))$ 
by (blast dest!: Pair-in-Vfrom-D)

```

ML

```

<
val rank-ss =
  simpset-of (context |> Simplifier.add-simp @{thm VsetI}
    |> Simplifier.add-simps (@{thms rank-rls} @ (@{thms rank-rls} RLN (2,
[@{thm lt-trans}])))));
>

end

```

25 A Small Universe for Lazy Recursive Types

theory *QUniv* **imports** *Univ QPair* **begin**

```

rep-datatype
  elimination sumE
  induction TrueI
  case-eqns case-Inl case-Inr

rep-datatype
  elimination qsumE
  induction TrueI
  case-eqns qcase-QInl qcase-QInr

```

```

definition
  quniv :: i  $\Rightarrow$  i where
    quniv(A)  $\equiv$  Pow(univ(eclose(A)))

```

25.1 Properties involving Transset and Sum

```

lemma Transset-includes-summands:
   $\llbracket Transset(C); A + B \subseteq C \rrbracket \implies A \subseteq C \wedge B \subseteq C$ 
apply (simp add: sum-def Un-subset-iff)
apply (blast dest: Transset-includes-range)
done

```

lemma *Transset-sum-Int-subset*:
 $\text{Transset}(C) \implies (A+B) \cap C \subseteq (A \cap C) + (B \cap C)$
apply (*simp add: sum-def Int-Un-distrib2*)
apply (*blast dest: Transset-Pair-D*)
done

25.2 Introduction and Elimination Rules

lemma *qunivI*: $X \subseteq \text{univ}(\text{eclose}(A)) \implies X \in \text{quniv}(A)$
by (*simp add: quniv-def*)

lemma *qunivD*: $X \in \text{quniv}(A) \implies X \subseteq \text{univ}(\text{eclose}(A))$
by (*simp add: quniv-def*)

lemma *quniv-mono*: $A \leq B \implies \text{quniv}(A) \subseteq \text{quniv}(B)$
unfolding *quniv-def*
apply (*erule eclose-mono [THEN univ-mono, THEN Pow-mono]*)
done

25.3 Closure Properties

lemma *univ-eclose-subset-quniv*: $\text{univ}(\text{eclose}(A)) \subseteq \text{quniv}(A)$
apply (*simp add: quniv-def Transset-iff-Pow [symmetric]*)
apply (*rule Transset-eclose [THEN Transset-univ]*)
done

lemma *univ-subset-quniv*: $\text{univ}(A) \subseteq \text{quniv}(A)$
apply (*rule arg-subset-eclose [THEN univ-mono, THEN subset-trans]*)
apply (*rule univ-eclose-subset-quniv*)
done

lemmas *univ-into-quniv* = *univ-subset-quniv* [*THEN subsetD*]

lemma *Pow-univ-subset-quniv*: $\text{Pow}(\text{univ}(A)) \subseteq \text{quniv}(A)$
unfolding *quniv-def*
apply (*rule arg-subset-eclose [THEN univ-mono, THEN Pow-mono]*)
done

lemmas *univ-subset-into-quniv* =
PowI [*THEN Pow-univ-subset-quniv* [*THEN subsetD*]]

lemmas *zero-in-quniv* = *zero-in-univ* [*THEN univ-into-quniv*]
lemmas *one-in-quniv* = *one-in-univ* [*THEN univ-into-quniv*]
lemmas *two-in-quniv* = *two-in-univ* [*THEN univ-into-quniv*]

lemmas *A-subset-quniv* = *subset-trans* [*OF A-subset-univ univ-subset-quniv*]

lemmas *A-into-quniv* = *A-subset-quniv* [*THEN subsetD*]

lemma *QPair-subset-univ*:
 $\llbracket a \subseteq \text{univ}(A); b \subseteq \text{univ}(A) \rrbracket \implies \langle a; b \rangle \subseteq \text{univ}(A)$
by (*simp add: QPair-def sum-subset-univ*)

25.4 Quine Disjoint Sum

lemma *QInl-subset-univ*: $a \subseteq \text{univ}(A) \implies \text{QInl}(a) \subseteq \text{univ}(A)$
unfolding *QInl-def*
apply (*erule empty-subsetI [THEN QPair-subset-univ]*)
done

lemmas *naturals-subset-nat* =
Ord-nat [THEN Ord-is-Transset, unfolded Transset-def, THEN bspec]

lemmas *naturals-subset-univ* =
subset-trans [OF naturals-subset-nat nat-subset-univ]

lemma *QInr-subset-univ*: $a \subseteq \text{univ}(A) \implies \text{QInr}(a) \subseteq \text{univ}(A)$
unfolding *QInr-def*
apply (*erule nat-1I [THEN naturals-subset-univ, THEN QPair-subset-univ]*)
done

25.5 Closure for Quine-Inspired Products and Sums

lemma *QPair-in-quniv*:
 $\llbracket a: \text{quniv}(A); b: \text{quniv}(A) \rrbracket \implies \langle a; b \rangle \in \text{quniv}(A)$
by (*simp add: quniv-def QPair-def sum-subset-univ*)

lemma *QSigma-quniv*: $\text{quniv}(A) \lt * \text{quniv}(A) \subseteq \text{quniv}(A)$
by (*blast intro: QPair-in-quniv*)

lemmas *QSigma-subset-quniv* = *subset-trans [OF QSigma-mono QSigma-quniv]*

lemma *quniv-QPair-D*:
 $\langle a; b \rangle \in \text{quniv}(A) \implies a: \text{quniv}(A) \wedge b: \text{quniv}(A)$
unfolding *quniv-def QPair-def*
apply (*rule Transset-includes-summands [THEN conjE]*)
apply (*rule Transset-eclose [THEN Transset-univ]*)
apply (*erule PowD, blast*)
done

lemmas *quniv-QPair-E* = *quniv-QPair-D [THEN conjE]*

lemma *quniv-QPair-iff*: $\langle a; b \rangle \in \text{quniv}(A) \longleftrightarrow a: \text{quniv}(A) \wedge b: \text{quniv}(A)$
by (*blast intro: QPair-in-quniv dest: quniv-QPair-D*)

25.6 Quine Disjoint Sum

lemma *QInl-in-quniv*: $a: \text{quniv}(A) \implies \text{QInl}(a) \in \text{quniv}(A)$
by (*simp add: QInl-def zero-in-quniv QPair-in-quniv*)

lemma *QInr-in-quniv*: $b: \text{quniv}(A) \implies \text{QInr}(b) \in \text{quniv}(A)$
by (*simp add: QInr-def one-in-quniv QPair-in-quniv*)

lemma *qsum-quniv*: $\text{quniv}(C) <+> \text{quniv}(C) \subseteq \text{quniv}(C)$
by (*blast intro: QInl-in-quniv QInr-in-quniv*)

lemmas *qsum-subset-quniv* = *subset-trans* [*OF qsum-mono qsum-quniv*]

25.7 The Natural Numbers

lemmas *nat-subset-quniv* = *subset-trans* [*OF nat-subset-univ univ-subset-quniv*]

lemmas *nat-into-quniv* = *nat-subset-quniv* [*THEN subsetD*]

lemmas *bool-subset-quniv* = *subset-trans* [*OF bool-subset-univ univ-subset-quniv*]

lemmas *bool-into-quniv* = *bool-subset-quniv* [*THEN subsetD*]

lemma *QPair-Int-Vfrom-succ-subset*:
 $\text{Transset}(X) \implies$
 $\langle a; b \rangle \cap \text{Vfrom}(X, \text{succ}(i)) \subseteq \langle a \cap \text{Vfrom}(X, i); b \cap \text{Vfrom}(X, i) \rangle$
by (*simp add: QPair-def sum-def Int-Un-distrib2 Un-mono*
product-Int-Vfrom-subset [*THEN subset-trans*]
Sigma-mono [*OF Int-lower1 subset-refl*])

25.8 "Take-Lemma" Rules

lemma *QPair-Int-Vfrom-subset*:
 $\text{Transset}(X) \implies$
 $\langle a; b \rangle \cap \text{Vfrom}(X, i) \subseteq \langle a \cap \text{Vfrom}(X, i); b \cap \text{Vfrom}(X, i) \rangle$
unfolding *QPair-def*
apply (*erule Transset-Vfrom* [*THEN Transset-sum-Int-subset*])
done

lemmas *QPair-Int-Vset-subset-trans* =
subset-trans [*OF Transset-0* [*THEN QPair-Int-Vfrom-subset*] *QPair-mono*]

lemma *QPair-Int-Vset-subset-UN*:
 $\text{Ord}(i) \implies \langle a; b \rangle \cap \text{Vset}(i) \subseteq (\bigcup_{j \in i}. \langle a \cap \text{Vset}(j); b \cap \text{Vset}(j) \rangle)$
apply (*erule Ord-cases*)

```

apply (simp add: Vfrom-0)

apply (erule ssubst)
apply (rule Transset-0 [THEN QPair-Int-Vfrom-succ-subset, THEN subset-trans])
apply (rule succI1 [THEN UN-upper])

apply (simp del: UN-simps
      add: Limit-Vfrom-eq Int-UN-distrib UN-mono QPair-Int-Vset-subset-trans)
done

end

```

26 Datatype and CoDatatype Definitions

```

theory Datatype
imports Inductive Univ QUniv
keywords datatype codatatype :: thy-decl
begin

ML-file <Tools/datatype-package.ML>

ML <
(*Typechecking rules for most datatypes involving univ*)
structure Data-Arg =
  struct
    val intrs =
      [@{thm SigmaI}], @{thm InlI}], @{thm InrI}],
      [@{thm Pair-in-univ}], @{thm Inl-in-univ}], @{thm Inr-in-univ}],
      [@{thm zero-in-univ}], @{thm A-into-univ}], @{thm nat-into-univ}], @{thm
UnCI}];

    val elims = [make-elim @{thm InlD}], make-elim @{thm InrD}], (*for mutual
recursion*)
      [@{thm SigmaE}], @{thm sumE}]; (*allows * and + in
spec*)
    end;

structure Data-Package =
  Add-datatype-def-Fun
    (structure Fp=Lfp and Pr=Standard-Prod and CP=Standard-CP
      and Su=Standard-Sum
      and Ind-Package = Ind-Package
      and Datatype-Arg = Data-Arg
      val coind = false);

```

```

(*Typechecking rules for most codatatypes involving quniv*)
structure CoData-Arg =
  struct
    val intrs =
      [ @{thm QSigmaI}, @{thm QInlI}, @{thm QInrI},
        @{thm QPair-in-quniv}, @{thm QInl-in-quniv}, @{thm QInr-in-quniv},
        @{thm zero-in-quniv}, @{thm A-into-quniv}, @{thm nat-into-quniv}, @{thm
UnCI}];

    val elims = [make-elim @{thm QInlD}, make-elim @{thm QInrD}, (*for mutual
recursion*)
      @{thm QSigmaE}, @{thm qsumE}];          (*allows * and +
in spec*)
    end;

structure CoData-Package =
  Add-datatype-def-Fun
  (structure Fp=Gfp and Pr=Quine-Prod and CP=Quine-CP
    and Su=Quine-Sum
    and Ind-Package = CoInd-Package
    and Datatype-Arg = CoData-Arg
    val coind = true);

(* Simproc for freeness reasoning: compare datatype constructors for equality *)

structure Data-Free:
sig
  val trace: bool Config.T
  val proc: Simplifier.proc
end =
  struct

    val trace = Attrib.setup-config-bool binding <data-free-trace> (K false);

    fun mk-new ([],[]) = Const <True>
      | mk-new (largs,rargs) =
          Balanced-Tree.make FOLogic.mk-conj
            (map FOLogic.mk-eq (ListPair.zip (largs,rargs)));

    val datatype-ss = simpset-of context;

    fun proc ctxt ct =
      let
        val old = Thm.term-of ct
        val thy = Proof-Context.theory-of ctxt
        val - =
          if Config.get ctxt trace then tracing (data-free: OLD = ^ Syntax.string-of-term

```

```

    ctxt old)
      else ()
      val (lhs,rhs) = FOLogic.dest-eq old
      val (lhead, largs) = strip-comb lhs
      and (rhead, rargs) = strip-comb rhs
      val lname = dest-Const-name lhead handle TERM - => raise Match;
      val rname = dest-Const-name rhead handle TERM - => raise Match;
      val lcon-info = the (Symtab.lookup (ConstructorsData.get thy) lname)
        handle Option.Option => raise Match;
      val rcon-info = the (Symtab.lookup (ConstructorsData.get thy) rname)
        handle Option.Option => raise Match;
      val new =
        if #big-rec-name lcon-info = #big-rec-name rcon-info
          andalso not (null (#free-iffs lcon-info)) then
            if lname = rname then mk-new (largs, rargs)
            else Const <False>
          else raise Match;
      val - =
        if Config.get ctxt trace then tracing (NEW = ^ Syntax.string-of-term ctxt
new)
        else ();
      val goal = Logic.mk-equals (old, new);
      val thm = Goal.prove ctxt [] [] goal
        (fn - => resolve-tac ctxt @ {thms iff-reflection} 1 THEN
          simp-tac (put-simpset datatype-ss ctxt
            |> Simplifier.add-simps (map (Thm.transfer thy) (#free-iffs lcon-info)))
        1)
        handle ERROR msg =>
          (warning (msg ^ \ndata-free simproc: \nfailed to prove ^ Syntax.string-of-term
ctxt goal);
            raise Match)
      in SOME thm end
      handle Match => NONE;

end;
>

simproc-setup data-free ((x::i) = y) = <fn - => Data-Free.proc>

end

```

27 Arithmetic Operators and Their Definitions

theory *Arith* **imports** *Univ* **begin**

Proofs about elementary arithmetic: addition, multiplication, etc.

definition

```

  pred :: i => i    where
    pred(y) ≡ nat-case(0, λx. x, y)

```

definition

$natify :: i \Rightarrow i$ **where**
 $natify \equiv Vrecursor(\lambda f a. \text{ if } a = succ(pred(a)) \text{ then } succ(f'pred(a))$
 $\text{ else } 0$)

consts

$raw-add :: [i,i] \Rightarrow i$
 $raw-diff :: [i,i] \Rightarrow i$
 $raw-mult :: [i,i] \Rightarrow i$

primrec

$raw-add \ 0, n = n$
 $raw-add \ (succ(m), n) = succ(raw-add(m, n))$

primrec

$raw-diff-0: \quad raw-diff(m, 0) = m$
 $raw-diff-succ: \quad raw-diff(m, succ(n)) =$
 $nat-case(0, \lambda x. x, raw-diff(m, n))$

primrec

$raw-mult \ 0, n = 0$
 $raw-mult \ (succ(m), n) = raw-add \ (n, raw-mult(m, n))$

definition

$add :: [i,i] \Rightarrow i$ **(infixl <#+> 65) where**
 $m \# + n \equiv raw-add \ (natify(m), natify(n))$

definition

$diff :: [i,i] \Rightarrow i$ **(infixl <#-> 65) where**
 $m \# - n \equiv raw-diff \ (natify(m), natify(n))$

definition

$mult :: [i,i] \Rightarrow i$ **(infixl <#*> 70) where**
 $m \# * n \equiv raw-mult \ (natify(m), natify(n))$

definition

$raw-div :: [i,i] \Rightarrow i$ **where**
 $raw-div \ (m, n) \equiv$
 $transrec(m, \lambda j f. \text{ if } j < n \mid n=0 \text{ then } 0 \text{ else } succ(f'(j\#-n)))$

definition

$raw-mod :: [i,i] \Rightarrow i$ **where**
 $raw-mod \ (m, n) \equiv$
 $transrec(m, \lambda j f. \text{ if } j < n \mid n=0 \text{ then } j \text{ else } f'(j\#-n))$

definition

$div :: [i,i] \Rightarrow i$ **(infixl <div> 70) where**
 $m \div n \equiv raw-div \ (natify(m), natify(n))$

definition

$mod :: [i,i] \Rightarrow i$ (**infixl** $\langle mod \rangle$ 70) **where**
 $m \text{ mod } n \equiv \text{raw-mod } (natify(m), natify(n))$

declare *rec-type* [simp]
nat-0-le [simp]

lemma *zero-lt-lemma*: $\llbracket 0 < k; k \in nat \rrbracket \Longrightarrow \exists j \in nat. k = succ(j)$
apply (*erule rev-mp*)
apply (*induct-tac k, auto*)
done

lemmas *zero-lt-natE* = *zero-lt-lemma* [THEN *bexE*]

27.1 *natify*, the Coercion to *nat*

lemma *pred-succ-eq* [simp]: $pred(succ(y)) = y$
by (*unfold pred-def, auto*)

lemma *natify-succ*: $natify(succ(x)) = succ(natify(x))$
by (*rule natify-def [THEN def-Vrecursor, THEN trans], auto*)

lemma *natify-0* [simp]: $natify(0) = 0$
by (*rule natify-def [THEN def-Vrecursor, THEN trans], auto*)

lemma *natify-non-succ*: $\forall z. x \neq succ(z) \Longrightarrow natify(x) = 0$
by (*rule natify-def [THEN def-Vrecursor, THEN trans], auto*)

lemma *natify-in-nat* [*iff, TC*]: $natify(x) \in nat$
apply (*rule-tac a=x in eps-induct*)
apply (*case-tac $\exists z. x = succ(z)$*)
apply (*auto simp add: natify-succ natify-non-succ*)
done

lemma *natify-ident* [simp]: $n \in nat \Longrightarrow natify(n) = n$
apply (*induct-tac n*)
apply (*auto simp add: natify-succ*)
done

lemma *natify-eqE*: $\llbracket natify(x) = y; x \in nat \rrbracket \Longrightarrow x=y$
by *auto*

lemma *natify-idem* [simp]: $natify(natify(x)) = natify(x)$

by *simp*

lemma *add-natify1* [*simp*]: $\text{natify}(m) \# + n = m \# + n$
by (*simp add: add-def*)

lemma *add-natify2* [*simp*]: $m \# + \text{natify}(n) = m \# + n$
by (*simp add: add-def*)

lemma *mult-natify1* [*simp*]: $\text{natify}(m) \# * n = m \# * n$
by (*simp add: mult-def*)

lemma *mult-natify2* [*simp*]: $m \# * \text{natify}(n) = m \# * n$
by (*simp add: mult-def*)

lemma *diff-natify1* [*simp*]: $\text{natify}(m) \# - n = m \# - n$
by (*simp add: diff-def*)

lemma *diff-natify2* [*simp*]: $m \# - \text{natify}(n) = m \# - n$
by (*simp add: diff-def*)

lemma *mod-natify1* [*simp*]: $\text{natify}(m) \bmod n = m \bmod n$
by (*simp add: mod-def*)

lemma *mod-natify2* [*simp*]: $m \bmod \text{natify}(n) = m \bmod n$
by (*simp add: mod-def*)

lemma *div-natify1* [*simp*]: $\text{natify}(m) \text{ div } n = m \text{ div } n$
by (*simp add: div-def*)

lemma *div-natify2* [*simp*]: $m \text{ div } \text{natify}(n) = m \text{ div } n$
by (*simp add: div-def*)

27.2 Typing rules

lemma *raw-add-type*: $\llbracket m \in \text{nat}; n \in \text{nat} \rrbracket \implies \text{raw-add } (m, n) \in \text{nat}$
by (*induct-tac m, auto*)

lemma *add-type* [*iff, TC*]: $m \# + n \in \text{nat}$

by (*simp add: add-def raw-add-type*)

lemma *raw-mult-type*: $\llbracket m \in \text{nat}; n \in \text{nat} \rrbracket \implies \text{raw-mult } (m, n) \in \text{nat}$
apply (*induct-tac m*)
apply (*simp-all add: raw-add-type*)
done

lemma *mult-type* [*iff, TC*]: $m \#* n \in \text{nat}$
by (*simp add: mult-def raw-mult-type*)

lemma *raw-diff-type*: $\llbracket m \in \text{nat}; n \in \text{nat} \rrbracket \implies \text{raw-diff } (m, n) \in \text{nat}$
by (*induct-tac n, auto*)

lemma *diff-type* [*iff, TC*]: $m \#- n \in \text{nat}$
by (*simp add: diff-def raw-diff-type*)

lemma *diff-0-eq-0* [*simp*]: $0 \#- n = 0$
unfolding *diff-def*
apply (*rule natify-in-nat [THEN nat-induct], auto*)
done

lemma *diff-succ-succ* [*simp*]: $\text{succ}(m) \#- \text{succ}(n) = m \#- n$
apply (*simp add: natify-succ diff-def*)
apply (*rule-tac x1 = n in natify-in-nat [THEN nat-induct], auto*)
done

declare *raw-diff-succ* [*simp del*]

lemma *diff-0* [*simp*]: $m \#- 0 = \text{natify}(m)$
by (*simp add: diff-def*)

lemma *diff-le-self*: $m \in \text{nat} \implies (m \#- n) \leq m$
apply (*subgoal-tac (m \#- natify (n)) \leq m*)
apply (*rule-tac [2] m = m and n = natify (n) in diff-induct*)
apply (*erule-tac [6] leE*)
apply (*simp-all add: le-iff*)
done

27.3 Addition

lemma *add-0-natify* [*simp*]: $0 \#+ m = \text{natify}(m)$

by (*simp add: add-def*)

lemma *add-succ [simp]: succ(m) #+ n = succ(m #+ n)*
by (*simp add: natify-succ add-def*)

lemma *add-0*: $m \in nat \implies 0 \# + m = m$
by *simp*

```

lemma add-assoc:  $(m \# + n) \# + k = m \# + (n \# + k)$ 
apply (subgoal-tac (natify( $m$ )  $\# +$  natify( $n$ ))  $\# +$  natify( $k$ ) =
      natify( $m$ )  $\# +$  (natify( $n$ )  $\# +$  natify( $k$ )))
apply (rule-tac [2]  $n = \text{natify}(m)$  in nat-induct)
apply auto
done

```

```

lemma add-0-right-natify [simp]:  $m \# + 0 = \text{natify}(m)$ 
apply (subgoal-tac natify( $m$ )  $\# + 0 = \text{natify}(m)$ )
apply (rule-tac [2]  $n = \text{natify}(m)$ ) in nat-induct
apply auto
done

```

```

lemma add-succ-right [simp]:  $m \# + \text{succ}(n) = \text{succ}(m \# + n)$ 
unfolding add-def
apply (rule-tac  $n = \text{natify}(m)$  in nat-induct)
apply (auto simp add: natify-succ)
done

```

lemma *add-0-right*: $m \in \text{nat} \implies m \# + 0 = m$
by *auto*

```

lemma add-commute:  $m \# + n = n \# + m$ 
apply (subgoal-tac natify( $m$ )  $\# +$  natify( $n$ ) = natify( $n$ )  $\# +$  natify( $m$ ) )
apply (rule-tac [2]  $n = \text{natify}(m)$ ) in nat-induct)
apply auto
done

```

```

lemma add-left-commute:  $m\#+(n\#+k)=n\#+(m\#+k)$ 
apply (rule add-commute [THEN trans])
apply (rule add-assoc [THEN trans])
apply (rule add-commute [THEN subst-context])
done

```

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

lemma *raw-add-left-cancel*:

$\llbracket \text{raw-add}(k, m) = \text{raw-add}(k, n); k \in \text{nat} \rrbracket \implies m = n$
apply (*erule rev-mp*)
apply (*induct-tac k, auto*)
done

lemma *add-left-cancel-natify*: $k \# + m = k \# + n \implies \text{natify}(m) = \text{natify}(n)$

unfolding *add-def*
apply (*drule raw-add-left-cancel, auto*)
done

lemma *add-left-cancel*:

$\llbracket i = j; i \# + m = j \# + n; m \in \text{nat}; n \in \text{nat} \rrbracket \implies m = n$
by (*force dest!: add-left-cancel-natify*)

lemma *add-le-elim1-natify*: $k \# + m \leq k \# + n \implies \text{natify}(m) \leq \text{natify}(n)$

apply (*rule-tac P = natify(k) \# + m \leq natify(k) \# + n in rev-mp*)
apply (*rule-tac [2] n = natify(k) in nat-induct*)
apply *auto*
done

lemma *add-le-elim1*: $\llbracket k \# + m \leq k \# + n; m \in \text{nat}; n \in \text{nat} \rrbracket \implies m \leq n$
by (*drule add-le-elim1-natify, auto*)

lemma *add-lt-elim1-natify*: $k \# + m < k \# + n \implies \text{natify}(m) < \text{natify}(n)$

apply (*rule-tac P = natify(k) \# + m < natify(k) \# + n in rev-mp*)
apply (*rule-tac [2] n = natify(k) in nat-induct*)
apply *auto*
done

lemma *add-lt-elim1*: $\llbracket k \# + m < k \# + n; m \in \text{nat}; n \in \text{nat} \rrbracket \implies m < n$
by (*drule add-lt-elim1-natify, auto*)

lemma *zero-less-add*: $\llbracket n \in \text{nat}; m \in \text{nat} \rrbracket \implies 0 < m \# + n \longleftrightarrow (0 < m \mid 0 < n)$
by (*induct-tac n, auto*)

27.4 Monotonicity of Addition

lemma *add-lt-mono1*: $\llbracket i < j; j \in \text{nat} \rrbracket \implies i \# + k < j \# + k$

apply (*frule lt-nat-in-nat, assumption*)
apply (*erule succ-lt-induct*)
apply (*simp-all add: leI*)
done

strict, in second argument

lemma *add-lt-mono2*: $\llbracket i < j; j \in \text{nat} \rrbracket \implies k \# + i < k \# + j$

by (*simp add: add-commute [of k] add-lt-mono1*)

A [clumsy] way of lifting $<$ monotonicity to \leq monotonicity

```

lemma Ord-lt-mono-imp-le-mono:
  assumes lt-mono:  $\bigwedge i j. \llbracket i < j; j : k \rrbracket \implies f(i) < f(j)$ 
    and ford:  $\bigwedge i. i : k \implies \text{Ord}(f(i))$ 
    and leij:  $i \leq j$ 
    and jink:  $j : k$ 
  shows  $f(i) \leq f(j)$ 
apply (insert leij jink)
apply (blast intro!: leCI lt-mono ford elim!: leE)
done

```

\leq monotonicity, 1st argument

```

lemma add-le-mono1:  $\llbracket i \leq j; j \in \text{nat} \rrbracket \implies i \# + k \leq j \# + k$ 
apply (rule-tac f =  $\lambda j. j \# + k$  in Ord-lt-mono-imp-le-mono, typecheck)
apply (blast intro: add-lt-mono1 add-type [THEN nat-into-Ord])
done

```

\leq monotonicity, both arguments

```

lemma add-le-mono:  $\llbracket i \leq j; k \leq l; j \in \text{nat}; l \in \text{nat} \rrbracket \implies i \# + k \leq j \# + l$ 
apply (rule add-le-mono1 [THEN le-trans], assumption+)
apply (subst add-commute, subst add-commute, rule add-le-mono1, assumption+)
done

```

Combinations of less-than and less-than-or-equals

```

lemma add-lt-le-mono:  $\llbracket i < j; k \leq l; j \in \text{nat}; l \in \text{nat} \rrbracket \implies i \# + k < j \# + l$ 
apply (rule add-lt-mono1 [THEN lt-trans2], assumption+)
apply (subst add-commute, subst add-commute, rule add-le-mono1, assumption+)
done

```

```

lemma add-le-lt-mono:  $\llbracket i \leq j; k < l; j \in \text{nat}; l \in \text{nat} \rrbracket \implies i \# + k < j \# + l$ 
by (subst add-commute, subst add-commute, erule add-lt-le-mono, assumption+)

```

Less-than: in other words, strict in both arguments

```

lemma add-lt-mono:  $\llbracket i < j; k < l; j \in \text{nat}; l \in \text{nat} \rrbracket \implies i \# + k < j \# + l$ 
apply (rule add-lt-le-mono)
apply (auto intro: leI)
done

```

```

lemma diff-add-inverse:  $(n \# + m) \# - n = \text{natify}(m)$ 
apply (subgoal-tac (natify(n) \# + m) \# - natify(n) = natify(m) ))
apply (rule-tac [2] n = natify(n) in nat-induct)
apply auto
done

```

```

lemma diff-add-inverse2:  $(m \# + n) \# - n = \text{natify}(m)$ 
by (simp add: add-commute [of m] diff-add-inverse)

```

lemma *diff-cancel*: $(k \# + m) \# - (k \# + n) = m \# - n$
apply (*subgoal-tac* ($\text{natify}(k) \# + \text{natify}(m)$) $\# - (\text{natify}(k) \# + \text{natify}(n)) =$
 $\text{natify}(m) \# - \text{natify}(n)$)
apply (*rule-tac* [2] $n = \text{natify}(k)$ **in** *nat-induct*)
apply *auto*
done

lemma *diff-cancel2*: $(m \# + k) \# - (n \# + k) = m \# - n$
by (*simp add: add-commute [of - k] diff-cancel*)

lemma *diff-add-0*: $n \# - (n \# + m) = 0$
apply (*subgoal-tac* $\text{natify}(n) \# - (\text{natify}(n) \# + \text{natify}(m)) = 0$)
apply (*rule-tac* [2] $n = \text{natify}(n)$ **in** *nat-induct*)
apply *auto*
done

lemma *pred-0 [simp]*: $\text{pred}(0) = 0$
by (*simp add: pred-def*)

lemma *eq-succ-imp-eq-m1*: $\llbracket i = \text{succ}(j); i \in \text{nat} \rrbracket \implies j = i \# - 1 \wedge j \in \text{nat}$
by *simp*

lemma *pred-Un-distrib*:
 $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{pred}(i \cup j) = \text{pred}(i) \cup \text{pred}(j)$
apply (*erule-tac* $n=i$ **in** *natE*, *simp*)
apply (*erule-tac* $n=j$ **in** *natE*, *simp*)
apply (*simp add: succ-Un-distrib [symmetric]*)
done

lemma *pred-type [TC,simp]*:
 $i \in \text{nat} \implies \text{pred}(i) \in \text{nat}$
by (*simp add: pred-def split: split-nat-case*)

lemma *nat-diff-pred*: $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies i \# - \text{succ}(j) = \text{pred}(i \# - j)$
apply (*rule-tac* $m=i$ **and** $n=j$ **in** *diff-induct*)
apply (*auto simp add: pred-def nat-imp-quasinat split: split-nat-case*)
done

lemma *diff-succ-eq-pred*: $i \# - \text{succ}(j) = \text{pred}(i \# - j)$
apply (*insert nat-diff-pred [of natify(i) natify(j)]*)
apply (*simp add: natify-succ [symmetric]*)
done

lemma *nat-diff-Un-distrib*:
 $\llbracket i \in \text{nat}; j \in \text{nat}; k \in \text{nat} \rrbracket \implies (i \cup j) \# - k = (i \# - k) \cup (j \# - k)$
apply (*rule-tac* $n=k$ **in** *nat-induct*)
apply (*simp-all add: diff-succ-eq-pred pred-Un-distrib*)
done

lemma *diff-Un-distrib*:

$\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies (i \cup j) \# - k = (i \# - k) \cup (j \# - k)$
by (*insert nat-diff-Un-distrib [of i j natify(k)], simp*)

We actually prove $i \# - j \# - k = i \# - (j \# + k)$

lemma *diff-diff-left [simplified]*:

$\text{natify}(i) \# - \text{natify}(j) \# - k = \text{natify}(i) \# - (\text{natify}(j) \# + k)$
by (*rule-tac m=natify(i) and n=natify(j) in diff-induct, auto*)

lemma *eq-add-iff*: $(u \# + m = u \# + n) \longleftrightarrow (0 \# + m = \text{natify}(n))$

apply *auto*

apply (*blast dest: add-left-cancel-natify*)

apply (*simp add: add-def*)

done

lemma *less-add-iff*: $(u \# + m < u \# + n) \longleftrightarrow (0 \# + m < \text{natify}(n))$

apply (*auto simp add: add-lt-elim1-natify*)

apply (*drule add-lt-mono1*)

apply (*auto simp add: add-commute [of u]*)

done

lemma *diff-add-eq*: $((u \# + m) \# - (u \# + n)) = ((0 \# + m) \# - n)$

by (*simp add: diff-cancel*)

lemma *eq-cong2*: $u = u' \implies (t \equiv u) \equiv (t \equiv u')$

by *auto*

lemma *iff-cong2*: $u \longleftrightarrow u' \implies (t \equiv u) \equiv (t \equiv u')$

by *auto*

27.5 Multiplication

lemma *mult-0 [simp]*: $0 \# * m = 0$

by (*simp add: mult-def*)

lemma *mult-succ [simp]*: $\text{succ}(m) \# * n = n \# + (m \# * n)$

by (*simp add: add-def mult-def natify-succ raw-mult-type*)

lemma *mult-0-right [simp]*: $m \# * 0 = 0$

unfolding *mult-def*

apply (*rule-tac n = natify(m) in nat-induct*)

apply *auto*

done


```

lemma mult-succ-right [simp]:  $m \#* \text{succ}(n) = m \#+ (m \#* n)$ 
apply (subgoal-tac natify( $m$ )  $\#* \text{succ}(\text{natify}(n)) =$ 
       $\text{natify}(m) \#+ (\text{natify}(m) \#* \text{natify}(n))$ )
apply (simp (no-asm-use) add: natify-succ add-def mult-def)
apply (rule-tac  $n = \text{natify}(m)$  in nat-induct)
apply (simp-all add: add-ac)
done

```

```

lemma mult-1-natify [simp]:  $1 \#* n = \text{natify}(n)$ 
by auto

```

```

lemma mult-1-right-natify [simp]:  $n \#* 1 = \text{natify}(n)$ 
by auto

```

```

lemma mult-1:  $n \in \text{nat} \implies 1 \#* n = n$ 
by simp

```

```

lemma mult-1-right:  $n \in \text{nat} \implies n \#* 1 = n$ 
by simp

```

```

lemma mult-commute:  $m \#* n = n \#* m$ 
apply (subgoal-tac  $\text{natify}(m) \#* \text{natify}(n) = \text{natify}(n) \#* \text{natify}(m)$  )
apply (rule-tac [2]  $n = \text{natify}(m)$  in nat-induct)
apply auto
done

```

```

lemma add-mult-distrib:  $(m \#+ n) \#* k = (m \#* k) \#+ (n \#* k)$ 
apply (subgoal-tac  $(\text{natify}(m) \#+ \text{natify}(n)) \#* \text{natify}(k) =$ 
       $(\text{natify}(m) \#* \text{natify}(k)) \#+ (\text{natify}(n) \#* \text{natify}(k))$ )
apply (rule-tac [2]  $n = \text{natify}(m)$  in nat-induct)
apply (simp-all add: add-assoc [symmetric])
done

```

```

lemma add-mult-distrib-left:  $k \#* (m \#+ n) = (k \#* m) \#+ (k \#* n)$ 
apply (subgoal-tac  $\text{natify}(k) \#* (\text{natify}(m) \#+ \text{natify}(n)) =$ 
       $(\text{natify}(k) \#* \text{natify}(m)) \#+ (\text{natify}(k) \#* \text{natify}(n))$ )
apply (rule-tac [2]  $n = \text{natify}(m)$  in nat-induct)
apply (simp-all add: add-ac)
done

```

```

lemma mult-assoc:  $(m \#* n) \#* k = m \#* (n \#* k)$ 
apply (subgoal-tac  $(\text{natify}(m) \#* \text{natify}(n)) \#* \text{natify}(k) =$ 
       $\text{natify}(m) \#* (\text{natify}(n) \#* \text{natify}(k))$ )

```

```

apply (rule-tac [2]  $n = \text{natty}(m)$  in nat-induct)
apply (simp-all add: add-mult-distrib)
done

```

```

lemma mult-left-commute:  $m \#* (n \#* k) = n \#* (m \#* k)$ 
apply (rule mult-commute [THEN trans])
apply (rule mult-assoc [THEN trans])
apply (rule mult-commute [THEN subst-context])
done

```

```

lemmas mult-ac = mult-assoc mult-commute mult-left-commute

```

```

lemma lt-succ-eq-0-disj:
   $\llbracket m \in \text{nat}; n \in \text{nat} \rrbracket \implies (m < \text{succ}(n)) \longleftrightarrow (m = 0 \mid (\exists j \in \text{nat}. m = \text{succ}(j) \wedge j < n))$ 
by (induct-tac m, auto)

```

```

lemma less-diff-conv [rule-format]:
   $\llbracket j \in \text{nat}; k \in \text{nat} \rrbracket \implies \forall i \in \text{nat}. (i < j \#- k) \longleftrightarrow (i \#+ k < j)$ 
by (erule-tac  $m = k$  in diff-induct, auto)

```

```

lemmas nat-typechecks = rec-type nat-0I nat-1I nat-succI Ord-nat

```

```

end

```

28 Arithmetic with simplification

```

theory ArithSimp
imports Arith
begin

```

28.1 Arithmetic simplification

```

ML-file <~~/src/Provers/Arith/cancel-numerals.ML>
ML-file <~~/src/Provers/Arith/combine-numerals.ML>
ML-file <arith-data.ML>

```

```

simproc-setup nateq-cancel-numerals
  ( $l \#+ m = n \mid l = m \#+ n \mid l \#* m = n \mid l = m \#* n \mid \text{succ}(m) = n \mid m = \text{succ}(n)$ ) =
  <K ArithData.nateq-cancel-numerals-proc>

```

```

simproc-setup natless-cancel-numerals
  ( $l \#+ m < n \mid l < m \#+ n \mid l \#* m < n \mid l < m \#* n \mid \text{succ}(m) < n \mid m < \text{succ}(n)$ ) =
  <K ArithData.natless-cancel-numerals-proc>

```

```

simproc-setup natdiff-cancel-numerals
  ((l #+ m) #- n | l #- (m #+ n) | (l #* m) #- n | l #- (m #* n) |
    succ(m) #- n | m #- succ(n)) =
  ⟨K ArithData.natdiff-cancel-numerals-proc⟩

```

28.1.1 Examples

```

lemma x #+ y = x #+ z apply simp oops
lemma y #+ x = x #+ z apply simp oops
lemma x #+ y #+ z = x #+ z apply simp oops
lemma y #+ (z #+ x) = z #+ x apply simp oops
lemma x #+ y #+ z = (z #+ y) #+ (x #+ w) apply simp oops
lemma x#*y #+ z = (z #+ y) #+ (y#*x #+ w) apply simp oops

lemma x #+ succ(y) = x #+ z apply simp oops
lemma x #+ succ(y) = succ(z #+ x) apply simp oops
lemma succ(x) #+ succ(y) #+ z = succ(z #+ y) #+ succ(x #+ w) apply simp
oops

lemma (x #+ y) #- (x #+ z) = w apply simp oops
lemma (y #+ x) #- (x #+ z) = dd apply simp oops
lemma (x #+ y #+ z) #- (x #+ z) = dd apply simp oops
lemma (y #+ (z #+ x)) #- (z #+ x) = dd apply simp oops
lemma (x #+ y #+ z) #- ((z #+ y) #+ (x #+ w)) = dd apply simp oops
lemma (x#*y #+ z) #- ((z #+ y) #+ (y#*x #+ w)) = dd apply simp oops

lemma (x #+ succ(y)) #- (x #+ z) = dd apply simp oops

lemma x #* y2 #+ y #* x2 = y #* x2 #+ x #* y2 apply simp oops

lemma (x #+ succ(y)) #- (succ(z #+ x)) = dd apply simp oops
lemma (succ(x) #+ succ(y) #+ z) #- (succ(z #+ y) #+ succ(x #+ w)) = dd
apply simp oops

lemma x : nat ==> x #+ y = x apply simp oops
lemma x : nat --> x #+ y = x apply simp oops
lemma x : nat ==> x #+ y < x apply simp oops
lemma x : nat ==> x < y#+x apply simp oops
lemma x : nat ==> x ≤ succ(x) apply simp oops

lemma x #+ y = x apply simp? oops

lemma x #+ y < x #+ z apply simp oops
lemma y #+ x < x #+ z apply simp oops
lemma x #+ y #+ z < x #+ z apply simp oops
lemma y #+ z #+ x < x #+ z apply simp oops

```

```

lemma y #+ (z #+ x) < z #+ x apply simp oops
lemma x #+ y #+ z < (z #+ y) #+ (x #+ w) apply simp oops
lemma x#*y #+ z < (z #+ y) #+ (y#*x #+ w) apply simp oops

lemma x #+ succ(y) < x #+ z apply simp oops
lemma x #+ succ(y) < succ(z #+ x) apply simp oops
lemma succ(x) #+ succ(y) #+ z < succ(z #+ y) #+ succ(x #+ w) apply simp
oops

lemma x #+ succ(y) ≤ succ(z #+ x) apply simp oops

```

28.2 Difference

```

lemma diff-self-eq-0 [simp]: m #- m = 0
apply (subgoal-tac natify (m) #- natify (m) = 0)
apply (rule-tac [2] natify-in-nat [THEN nat-induct], auto)
done

```

```

lemma add-diff-inverse: [n ≤ m; m:nat] ⇒ n #+ (m#-n) = m
apply (frule lt-nat-in-nat, erule nat-succI)
apply (erule rev-mp)
apply (rule-tac m = m and n = n in diff-induct, auto)
done

```

```

lemma add-diff-inverse2: [n ≤ m; m:nat] ⇒ (m#-n) #+ n = m
apply (frule lt-nat-in-nat, erule nat-succI)
apply (simp (no-asm-simp) add: add-commute add-diff-inverse)
done

```

```

lemma diff-succ: [n ≤ m; m:nat] ⇒ succ(m) #- n = succ(m#-n)
apply (frule lt-nat-in-nat, erule nat-succI)
apply (erule rev-mp)
apply (rule-tac m = m and n = n in diff-induct)
apply (simp-all (no-asm-simp))
done

```

```

lemma zero-less-diff [simp]:
  [m: nat; n: nat] ⇒ 0 < (n #- m) ↔ m < n
apply (rule-tac m = m and n = n in diff-induct)
apply (simp-all (no-asm-simp))
done

```

```

lemma diff-mult-distrib:  $(m \#- n) \#* k = (m \#* k) \#- (n \#* k)$ 
apply (subgoal-tac (natify ( $m$ )  $\#-$  natify ( $n$ ))  $\#*$  natify ( $k$ ) = (natify ( $m$ )  $\#*$ 
natify ( $k$ ))  $\#-$  (natify ( $n$ )  $\#*$  natify ( $k$ )))
apply (rule-tac [2]  $m = \text{natify } (m)$  and  $n = \text{natify } (n)$  in diff-induct)
apply (simp-all add: diff-cancel)
done

```

```

lemma diff-mult-distrib2:  $k \#* (m \#- n) = (k \#* m) \#- (k \#* n)$ 
apply (simp (no-asm) add: mult-commute [of  $k$ ] diff-mult-distrib)
done

```

28.3 Remainder

```

lemma div-termination:  $\llbracket 0 < n; \ n \leq m; \ m:\text{nat} \rrbracket \implies m \#- n < m$ 
apply (frule lt-nat-in-nat, erule nat-succI)
apply (erule rev-mp)
apply (erule rev-mp)
apply (rule-tac  $m = m$  and  $n = n$  in diff-induct)
apply (simp-all (no-asm-simp) add: diff-le-self)
done

```

```

lemmas div-rls =
  nat-typechecks Ord-transrec-type apply-funtype
  div-termination [THEN ltD]
  nat-into-Ord not-lt-iff-le [THEN iffD1]

```

```

lemma raw-mod-type:  $\llbracket m:\text{nat}; \ n:\text{nat} \rrbracket \implies \text{raw-mod } (m, n) \in \text{nat}$ 
  unfolding raw-mod-def
apply (rule Ord-transrec-type)
apply (auto simp add: nat-into-Ord [THEN Ord-0-lt-iff])
apply (blast intro: div-rls)
done

```

```

lemma mod-type [TC,iff]:  $m \text{ mod } n \in \text{nat}$ 
  unfolding mod-def
apply (simp (no-asm) add: mod-def raw-mod-type)
done

```

```

lemma DIVISION-BY-ZERO-DIV:  $a \text{ div } 0 = 0$ 
  unfolding div-def
apply (rule raw-div-def [THEN def-transrec, THEN trans])
apply (simp (no-asm-simp))
done

```

```

lemma DIVISION-BY-ZERO-MOD:  $a \text{ mod } 0 = \text{natify}(a)$ 

```

```

  unfolding mod-def
  apply (rule raw-mod-def [THEN def-transrec, THEN trans])
  apply (simp (no-asm-simp))
  done

```

```

lemma raw-mod-less:  $m < n \implies \text{raw-mod } (m, n) = m$ 
  apply (rule raw-mod-def [THEN def-transrec, THEN trans])
  apply (simp (no-asm-simp) add: div-termination [THEN ltD])
  done

```

```

lemma mod-less [simp]:  $\llbracket m < n; n \in \text{nat} \rrbracket \implies m \bmod n = m$ 
  apply (frule lt-nat-in-nat, assumption)
  apply (simp (no-asm-simp) add: mod-def raw-mod-less)
  done

```

```

lemma raw-mod-geq:
   $\llbracket 0 < n; n \leq m; m : \text{nat} \rrbracket \implies \text{raw-mod } (m, n) = \text{raw-mod } (m \# -n, n)$ 
  apply (frule lt-nat-in-nat, erule nat-succI)
  apply (rule raw-mod-def [THEN def-transrec, THEN trans])
  apply (simp (no-asm-simp) add: div-termination [THEN ltD] not-lt-iff-le [THEN
    iffD2], blast)
  done

```

```

lemma mod-geq:  $\llbracket n \leq m; m : \text{nat} \rrbracket \implies m \bmod n = (m \# -n) \bmod n$ 
  apply (frule lt-nat-in-nat, erule nat-succI)
  apply (case-tac n=0)
  apply (simp add: DIVISION-BY-ZERO-MOD)
  apply (simp add: mod-def raw-mod-geq nat-into-Ord [THEN Ord-0-lt-iff])
  done

```

28.4 Division

```

lemma raw-div-type:  $\llbracket m : \text{nat}; n : \text{nat} \rrbracket \implies \text{raw-div } (m, n) \in \text{nat}$ 
  unfolding raw-div-def
  apply (rule Ord-transrec-type)
  apply (auto simp add: nat-into-Ord [THEN Ord-0-lt-iff])
  apply (blast intro: div-rls)
  done

```

```

lemma div-type [TC, iff]:  $m \text{ div } n \in \text{nat}$ 
  unfolding div-def
  apply (simp (no-asm) add: div-def raw-div-type)
  done

```

```

lemma raw-div-less:  $m < n \implies \text{raw-div } (m, n) = 0$ 
  apply (rule raw-div-def [THEN def-transrec, THEN trans])
  apply (simp (no-asm-simp) add: div-termination [THEN ltD])
  done

```

```

lemma div-less [simp]:  $\llbracket m < n; n \in \text{nat} \rrbracket \implies m \text{ div } n = 0$ 
apply (frule lt-nat-in-nat, assumption)
apply (simp (no-asm-simp) add: div-def raw-div-less)
done

lemma raw-div-geq:  $\llbracket 0 < n; n \leq m; m:\text{nat} \rrbracket \implies \text{raw-div}(m,n) = \text{succ}(\text{raw-div}(m\#-n, n))$ 
apply (subgoal-tac  $n \neq 0$ )
prefer 2 apply blast
apply (frule lt-nat-in-nat, erule nat-succI)
apply (rule raw-div-def [THEN def-transrec, THEN trans])
apply (simp (no-asm-simp) add: div-termination [THEN ltD] not-lt-iff-le [THEN iffD2])
done

lemma div-geq [simp]:
   $\llbracket 0 < n; n \leq m; m:\text{nat} \rrbracket \implies m \text{ div } n = \text{succ}((m\#-n) \text{ div } n)$ 
apply (frule lt-nat-in-nat, erule nat-succI)
apply (simp (no-asm-simp) add: div-def raw-div-geq)
done

declare div-less [simp] div-geq [simp]

lemma mod-div-lemma:  $\llbracket m:\text{nat}; n:\text{nat} \rrbracket \implies (m \text{ div } n)\#*n \# + m \text{ mod } n = m$ 
apply (case-tac  $n=0$ )
  apply (simp add: DIVISION-BY-ZERO-MOD)
apply (simp add: nat-into-Ord [THEN Ord-0-lt-iff])
apply (erule complete-induct)
apply (case-tac  $x < n$ )

  case  $x < n$ 
apply (simp (no-asm-simp))

  case  $n \leq x$ 
apply (simp add: not-lt-iff-le add-assoc mod-geq div-termination [THEN ltD] add-diff-inverse)
done

lemma mod-div-equality-natify:  $(m \text{ div } n)\#*n \# + m \text{ mod } n = \text{natify}(m)$ 
apply (subgoal-tac ( $\text{natify}(m) \text{ div } \text{natify}(n)\#*\text{natify}(n) \# + \text{natify}(m) \text{ mod } \text{natify}(n) = \text{natify}(m)$ ))
apply force
apply (subst mod-div-lemma, auto)
done

lemma mod-div-equality:  $m:\text{nat} \implies (m \text{ div } n)\#*n \# + m \text{ mod } n = m$ 
apply (simp (no-asm-simp) add: mod-div-equality-natify)
done

```

28.5 Further Facts about Remainder

(mainly for mutilated chess board)

lemma *mod-succ-lemma*:

$\llbracket 0 < n; m:\text{nat}; n:\text{nat} \rrbracket$

$\implies \text{succ}(m) \bmod n = (\text{if } \text{succ}(m \bmod n) = n \text{ then } 0 \text{ else } \text{succ}(m \bmod n))$

apply (*erule complete-induct*)

apply (*case-tac succ (x) < n*)

case $\text{succ}(x) < n$

apply (*simp (no-asm-simp) add: nat-le-refl [THEN lt-trans] succ-neq-self*)

apply (*simp add: ltD [THEN mem-imp-not-eq]*)

case $n \leq \text{succ}(x)$

apply (*simp add: mod-geq not-lt-iff-le*)

apply (*erule leE*)

apply (*simp (no-asm-simp) add: mod-geq div-termination [THEN ltD] diff-succ*)

equality case

apply (*simp add: diff-self-eq-0*)

done

lemma *mod-succ*:

$n:\text{nat} \implies \text{succ}(m) \bmod n = (\text{if } \text{succ}(m \bmod n) = n \text{ then } 0 \text{ else } \text{succ}(m \bmod n))$

apply (*case-tac n=0*)

apply (*simp (no-asm-simp) add: natify-succ DIVISION-BY-ZERO-MOD*)

apply (*subgoal-tac natify (succ (m)) mod n = (if succ (natify (m)) mod n = n then 0 else succ (natify (m) mod n))*)

prefer 2

apply (*subst natify-succ*)

apply (*rule mod-succ-lemma*)

apply (*auto simp del: natify-succ simp add: nat-into-Ord [THEN Ord-0-lt-iff]*)

done

lemma *mod-less-divisor*: $\llbracket 0 < n; n:\text{nat} \rrbracket \implies m \bmod n < n$

apply (*subgoal-tac natify (m) mod n < n*)

apply (*rule-tac [2] i = natify (m) in complete-induct*)

apply (*case-tac [3] x < n, auto*)

case $n \leq x$

apply (*simp add: mod-geq not-lt-iff-le div-termination [THEN ltD]*)

done

lemma *mod-1-eq* [*simp*]: $m \bmod 1 = 0$

by (*cut-tac n = 1 in mod-less-divisor, auto*)

lemma *mod2-cases*: $b < 2 \implies k \bmod 2 = b \mid k \bmod 2 = (\text{if } b=1 \text{ then } 0 \text{ else } 1)$

apply (*subgoal-tac k mod 2: 2*)

prefer 2 **apply** (*simp add: mod-less-divisor [THEN ltD]*)


```

apply (drule ltD, auto)
done

```

```

lemma mod2-succ-succ [simp]: succ(succ(m)) mod 2 = m mod 2
apply (subgoal-tac m mod 2: 2)
prefer 2 apply (simp add: mod-less-divisor [THEN ltD])
apply (auto simp add: mod-succ)
done

```

```

lemma mod2-add-more [simp]: (m#+m#+n) mod 2 = n mod 2
apply (subgoal-tac (natify (m) #+natify (m) #+n) mod 2 = n mod 2)
apply (rule-tac [2] n = natify (m) in nat-induct)
apply auto
done

```

```

lemma mod2-add-self [simp]: (m#+m) mod 2 = 0
by (cut-tac n = 0 in mod2-add-more, auto)

```

28.6 Additional theorems about \leq

```

lemma add-le-self: m:nat  $\implies m \leq (m \#+ n)$ 
apply (simp (no-asm-simp))
done

```

```

lemma add-le-self2: m:nat  $\implies m \leq (n \#+ m)$ 
apply (simp (no-asm-simp))
done

```

```

lemma mult-le-mono1:  $\llbracket i \leq j; j:\text{nat} \rrbracket \implies (i\#*k) \leq (j\#*k)$ 
apply (subgoal-tac natify (i) #*natify (k)  $\leq j\#*natify (k)$ )
apply (frule-tac [2] lt-nat-in-nat)
apply (rule-tac [3] n = natify (k) in nat-induct)
apply (simp-all add: add-le-mono)
done

```

```

lemma mult-le-mono:  $\llbracket i \leq j; k \leq l; j:\text{nat}; l:\text{nat} \rrbracket \implies i\#*k \leq j\#*l$ 
apply (rule mult-le-mono1 [THEN le-trans], assumption+)
apply (subst mult-commute, subst mult-commute, rule mult-le-mono1, assumption+)
done

```

```

lemma mult-lt-mono2:  $\llbracket i < j; 0 < k; j:\text{nat}; k:\text{nat} \rrbracket \implies k\#*i < k\#*j$ 
apply (erule zero-lt-natE)
apply (frule-tac [2] lt-nat-in-nat)
apply (simp-all (no-asm-simp))

```

```

apply (induct-tac x)
apply (simp-all (no-asm-simp) add: add-lt-mono)
done

```

```

lemma mult-lt-mono1:  $\llbracket i < j; 0 < k; j:\text{nat}; k:\text{nat} \rrbracket \implies i \# * k < j \# * k$ 
apply (simp (no-asm-simp) add: mult-lt-mono2 mult-commute [of - k])
done

```

```

lemma add-eq-0-iff [iff]:  $m \# + n = 0 \longleftrightarrow \text{natify}(m) = 0 \wedge \text{natify}(n) = 0$ 
apply (subgoal-tac  $\text{natify } (m) \# + \text{natify } (n) = 0 \longleftrightarrow \text{natify } (m) = 0 \wedge \text{natify } (n) = 0$ )
apply (rule-tac [2]  $n = \text{natify } (m)$  in natE)
apply (rule-tac [4]  $n = \text{natify } (n)$  in natE)
apply auto
done

```

```

lemma zero-lt-mult-iff [iff]:  $0 < m \# * n \longleftrightarrow 0 < \text{natify}(m) \wedge 0 < \text{natify}(n)$ 
apply (subgoal-tac  $0 < \text{natify } (m) \# * \text{natify } (n) \longleftrightarrow 0 < \text{natify } (m) \wedge 0 < \text{natify } (n)$ )
apply (rule-tac [2]  $n = \text{natify } (m)$  in natE)
apply (rule-tac [4]  $n = \text{natify } (n)$  in natE)
apply (rule-tac [3]  $n = \text{natify } (n)$  in natE)
apply auto
done

```

```

lemma mult-eq-1-iff [iff]:  $m \# * n = 1 \longleftrightarrow \text{natify}(m) = 1 \wedge \text{natify}(n) = 1$ 
apply (subgoal-tac  $\text{natify } (m) \# * \text{natify } (n) = 1 \longleftrightarrow \text{natify } (m) = 1 \wedge \text{natify } (n) = 1$ )
apply (rule-tac [2]  $n = \text{natify } (m)$  in natE)
apply (rule-tac [4]  $n = \text{natify } (n)$  in natE)
apply auto
done

```

```

lemma mult-is-zero:  $\llbracket m:\text{nat}; n:\text{nat} \rrbracket \implies (m \# * n = 0) \longleftrightarrow (m = 0 \mid n = 0)$ 
apply auto
apply (erule natE)
apply (erule-tac [2] natE, auto)
done

```

```

lemma mult-is-zero-natify [iff]:
   $(m \# * n = 0) \longleftrightarrow (\text{natify}(m) = 0 \mid \text{natify}(n) = 0)$ 
apply (cut-tac  $m = \text{natify } (m)$  and  $n = \text{natify } (n)$  in mult-is-zero)
apply auto
done

```

28.7 Cancellation Laws for Common Factors in Comparisons

```

lemma mult-less-cancel-lemma:

```

```

     $\llbracket k: \text{nat}; m: \text{nat}; n: \text{nat} \rrbracket \implies (m \# * k < n \# * k) \longleftrightarrow (0 < k \wedge m < n)$ 
  apply (safe intro!: mult-lt-mono1)
  apply (erule natE, auto)
  apply (rule not-le-iff-lt [THEN iffD1])
  apply (drule-tac [3] not-le-iff-lt [THEN [2] rev-iffD2])
  prefer 5 apply (blast intro: mult-le-mono1, auto)
done

```

```

lemma mult-less-cancel2 [simp]:
   $(m \# * k < n \# * k) \longleftrightarrow (0 < \text{nativify}(k) \wedge \text{nativify}(m) < \text{nativify}(n))$ 
  apply (rule iff-trans)
  apply (rule-tac [2] mult-less-cancel-lemma, auto)
done

```

```

lemma mult-less-cancel1 [simp]:
   $(k \# * m < k \# * n) \longleftrightarrow (0 < \text{nativify}(k) \wedge \text{nativify}(m) < \text{nativify}(n))$ 
  apply (simp (no-asm) add: mult-less-cancel2 mult-commute [of k])
done

```

```

lemma mult-le-cancel2 [simp]:  $(m \# * k \leq n \# * k) \longleftrightarrow (0 < \text{nativify}(k) \longrightarrow \text{nativify}(m) \leq \text{nativify}(n))$ 
  apply (simp (no-asm-simp) add: not-lt-iff-le [THEN iff-sym])
  apply auto
done

```

```

lemma mult-le-cancel1 [simp]:  $(k \# * m \leq k \# * n) \longleftrightarrow (0 < \text{nativify}(k) \longrightarrow \text{nativify}(m) \leq \text{nativify}(n))$ 
  apply (simp (no-asm-simp) add: not-lt-iff-le [THEN iff-sym])
  apply auto
done

```

```

lemma mult-le-cancel-le1:  $k \in \text{nat} \implies k \# * m \leq k \longleftrightarrow (0 < k \longrightarrow \text{nativify}(m) \leq 1)$ 
  by (cut-tac k = k and m = m and n = 1 in mult-le-cancel1, auto)

```

```

lemma Ord-eq-iff-le:  $\llbracket \text{Ord}(m); \text{Ord}(n) \rrbracket \implies m = n \longleftrightarrow (m \leq n \wedge n \leq m)$ 
  by (blast intro: le-anti-sym)

```

```

lemma mult-cancel2-lemma:
   $\llbracket k: \text{nat}; m: \text{nat}; n: \text{nat} \rrbracket \implies (m \# * k = n \# * k) \longleftrightarrow (m = n \mid k = 0)$ 
  apply (simp (no-asm-simp) add: Ord-eq-iff-le [of m # * k] Ord-eq-iff-le [of m])
  apply (auto simp add: Ord-0-lt-iff)
done

```

```

lemma mult-cancel2 [simp]:
   $(m \# * k = n \# * k) \longleftrightarrow (\text{nativify}(m) = \text{nativify}(n) \mid \text{nativify}(k) = 0)$ 
  apply (rule iff-trans)
  apply (rule-tac [2] mult-cancel2-lemma, auto)
done

```

```

lemma mult-cancel1 [simp]:
   $(k \# * m = k \# * n) \longleftrightarrow (\text{natify}(m) = \text{natify}(n) \mid \text{natify}(k) = 0)$ 
apply (simp (no-asm) add: mult-cancel2 mult-commute [of k])
done

lemma div-cancel-raw:
   $\llbracket 0 < n; 0 < k; k:\text{nat}; m:\text{nat}; n:\text{nat} \rrbracket \implies (k \# * m) \text{ div } (k \# * n) = m \text{ div } n$ 
apply (erule-tac i = m in complete-induct)
apply (case-tac x < n)
  apply (simp add: div-less zero-lt-mult-iff mult-lt-mono2)
apply (simp add: not-lt-iff-le zero-lt-mult-iff le-refl [THEN mult-le-mono]
  div-geq diff-mult-distrib2 [symmetric] div-termination [THEN ltD])
done

lemma div-cancel:
   $\llbracket 0 < \text{natify}(n); 0 < \text{natify}(k) \rrbracket \implies (k \# * m) \text{ div } (k \# * n) = m \text{ div } n$ 
apply (cut-tac k = natify (k) and m = natify (m) and n = natify (n)
  in div-cancel-raw)
apply auto
done

```

28.8 More Lemmas about Remainder

```

lemma mult-mod-distrib-raw:
   $\llbracket k:\text{nat}; m:\text{nat}; n:\text{nat} \rrbracket \implies (k \# * m) \text{ mod } (k \# * n) = k \# * (m \text{ mod } n)$ 
apply (case-tac k=0)
  apply (simp add: DIVISION-BY-ZERO-MOD)
apply (case-tac n=0)
  apply (simp add: DIVISION-BY-ZERO-MOD)
apply (simp add: nat-into-Ord [THEN Ord-0-lt-iff])
apply (erule-tac i = m in complete-induct)
apply (case-tac x < n)
  apply (simp (no-asm-simp) add: mod-less zero-lt-mult-iff mult-lt-mono2)
apply (simp add: not-lt-iff-le zero-lt-mult-iff le-refl [THEN mult-le-mono]
  mod-geq diff-mult-distrib2 [symmetric] div-termination [THEN ltD])
done

lemma mod-mult-distrib2:  $k \# * (m \text{ mod } n) = (k \# * m) \text{ mod } (k \# * n)$ 
apply (cut-tac k = natify (k) and m = natify (m) and n = natify (n)
  in mult-mod-distrib-raw)
apply auto
done

lemma mult-mod-distrib:  $(m \text{ mod } n) \# * k = (m \# * k) \text{ mod } (n \# * k)$ 
apply (simp (no-asm) add: mult-commute mod-mult-distrib2)

```

done

lemma *mod-add-self2-raw*: $n \in \text{nat} \implies (m \# + n) \bmod n = m \bmod n$
apply (*subgoal-tac* ($n \# + m$) $\bmod n = (n \# + m \# - n) \bmod n$)
apply (*simp* *add*: *add-commute*)
apply (*subst* *mod-geq* [*symmetric*], *auto*)
done

lemma *mod-add-self2* [*simp*]: $(m \# + n) \bmod n = m \bmod n$
apply (*cut-tac* $n = \text{nativify } (n)$ **in** *mod-add-self2-raw*)
apply *auto*
done

lemma *mod-add-self1* [*simp*]: $(n \# + m) \bmod n = m \bmod n$
apply (*simp* (*no-asm-simp*) *add*: *add-commute* *mod-add-self2*)
done

lemma *mod-mult-self1-raw*: $k \in \text{nat} \implies (m \# + k \# * n) \bmod n = m \bmod n$
apply (*erule* *nat-induct*)
apply (*simp-all* (*no-asm-simp*) *add*: *add-left-commute* [*of - n*])
done

lemma *mod-mult-self1* [*simp*]: $(m \# + k \# * n) \bmod n = m \bmod n$
apply (*cut-tac* $k = \text{nativify } (k)$ **in** *mod-mult-self1-raw*)
apply *auto*
done

lemma *mod-mult-self2* [*simp*]: $(m \# + n \# * k) \bmod n = m \bmod n$
apply (*simp* (*no-asm*) *add*: *mult-commute* *mod-mult-self1*)
done

lemma *mult-eq-self-implies-10*: $m = m \# * n \implies \text{nativify}(n)=1 \mid m=0$
apply (*subgoal-tac* $m: \text{nat}$)
prefer 2
apply (*erule* *ssubst*)
apply *simp*
apply (*rule* *disjCI*)
apply (*drule* *sym*)
apply (*rule* *Ord-linear-lt* [*of natify(n) 1*])
apply *simp-all*
apply (*subgoal-tac* $m \# * n = 0$, *simp*)
apply (*subst* *mult-nativify2* [*symmetric*])
apply (*simp* *del*: *mult-nativify2*)
apply (*drule* *nat-into-Ord* [*THEN* *Ord-0-lt*, *THEN* [*2*] *mult-lt-mono2*], *auto*)
done

lemma *less-imp-succ-add* [*rule-format*]:
 $\llbracket m < n; n: \text{nat} \rrbracket \implies \exists k \in \text{nat}. n = \text{succ}(m \# + k)$

```

apply (frule lt-nat-in-nat, assumption)
apply (erule rev-mp)
apply (induct-tac n)
apply (simp-all (no-asm) add: le-iff)
apply (blast elim!: leE intro!: add-0-right [symmetric] add-succ-right [symmetric])
done

```

```

lemma less-iff-succ-add:
   $\llbracket m: \text{nat}; n: \text{nat} \rrbracket \implies (m < n) \longleftrightarrow (\exists k \in \text{nat}. n = \text{succ}(m \# + k))$ 
by (auto intro: less-imp-succ-add)

```

```

lemma add-lt-elim2:
   $\llbracket a \# + d = b \# + c; a < b; b \in \text{nat}; c \in \text{nat}; d \in \text{nat} \rrbracket \implies c < d$ 
by (drule less-imp-succ-add, auto)

```

```

lemma add-le-elim2:
   $\llbracket a \# + d = b \# + c; a \leq b; b \in \text{nat}; c \in \text{nat}; d \in \text{nat} \rrbracket \implies c \leq d$ 
by (drule less-imp-succ-add, auto)

```

28.8.1 More Lemmas About Difference

```

lemma diff-is-0-lemma:
   $\llbracket m: \text{nat}; n: \text{nat} \rrbracket \implies m \# - n = 0 \longleftrightarrow m \leq n$ 
apply (rule-tac m = m and n = n in diff-induct, simp-all)
done

```

```

lemma diff-is-0-iff:  $m \# - n = 0 \longleftrightarrow \text{natify}(m) \leq \text{natify}(n)$ 
by (simp add: diff-is-0-lemma [symmetric])

```

```

lemma nat-lt-imp-diff-eq-0:
   $\llbracket a: \text{nat}; b: \text{nat}; a < b \rrbracket \implies a \# - b = 0$ 
by (simp add: diff-is-0-iff le-iff)

```

```

lemma raw-nat-diff-split:
   $\llbracket a: \text{nat}; b: \text{nat} \rrbracket \implies$ 
   $(P(a \# - b)) \longleftrightarrow ((a < b \longrightarrow P(0)) \wedge (\forall d \in \text{nat}. a = b \# + d \longrightarrow P(d)))$ 
apply (case-tac a < b)
apply (force simp add: nat-lt-imp-diff-eq-0)
apply (rule iffI, force, simp)
apply (drule-tac x=a#-b in bspec)
apply (simp-all add: Ordinal.not-lt-iff-le add-diff-inverse)
done

```

```

lemma nat-diff-split:
   $(P(a \# - b)) \longleftrightarrow$ 
   $(\text{natify}(a) < \text{natify}(b) \longrightarrow P(0)) \wedge (\forall d \in \text{nat}. \text{natify}(a) = b \# + d \longrightarrow P(d))$ 
apply (cut-tac P=P and a=natify(a) and b=natify(b) in raw-nat-diff-split)
apply simp-all
done

```

Difference and less-than

```

lemma diff-lt-imp-lt:  $\llbracket (k\#-i) < (k\#-j); i \in \text{nat}; j \in \text{nat}; k \in \text{nat} \rrbracket \implies j < i$ 
apply (erule rev-mp)
apply (simp split: nat-diff-split, auto)
apply (blast intro: add-le-self lt-trans1)
apply (rule not-le-iff-lt [THEN iffD1], auto)
apply (subgoal-tac i #+ da < j #+ d, force)
apply (blast intro: add-le-lt-mono)
done

```

```

lemma lt-imp-diff-lt:  $\llbracket j < i; i \leq k; k \in \text{nat} \rrbracket \implies (k\#-i) < (k\#-j)$ 
apply (frule le-in-nat, assumption)
apply (frule lt-nat-in-nat, assumption)
apply (simp split: nat-diff-split, auto)
apply (blast intro: lt-asym lt-trans2)
apply (blast intro: lt-irrefl lt-trans2)
apply (rule not-le-iff-lt [THEN iffD1], auto)
apply (subgoal-tac j #+ d < i #+ da, force)
apply (blast intro: add-lt-le-mono)
done

```

```

lemma diff-lt-iff-lt:  $\llbracket i \leq k; j \in \text{nat}; k \in \text{nat} \rrbracket \implies (k\#-i) < (k\#-j) \longleftrightarrow j < i$ 
apply (frule le-in-nat, assumption)
apply (blast intro: lt-imp-diff-lt diff-lt-imp-lt)
done

```

end

29 Lists in Zermelo-Fraenkel Set Theory

theory *List* **imports** *Datatype ArithSimp* **begin**

consts

list :: $i \Rightarrow i$

datatype

list(A) = *Nil* | *Cons* ($a \in A, l \in \text{list}(A)$)

notation *Nil* ($\langle [] \rangle$)

syntax

-List :: $is \Rightarrow i$ ($\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix list enumeration} \rangle [-] \rangle \rangle$)

translations

$[x, xs]$ == *CONST Cons*($x, [xs]$)
 $[x]$ == *CONST Cons*($x, []$)

consts

$length :: i \Rightarrow i$
 $hd :: i \Rightarrow i$
 $tl :: i \Rightarrow i$

primrec

$length([]) = 0$
 $length(Cons(a,l)) = succ(length(l))$

primrec

$hd([]) = 0$
 $hd(Cons(a,l)) = a$

primrec

$tl([]) = []$
 $tl(Cons(a,l)) = l$

consts

$map :: [i \Rightarrow i, i] \Rightarrow i$
 $set-of-list :: i \Rightarrow i$
 $app :: [i, i] \Rightarrow i$ (**infixr** '@' 60)

primrec

$map(f, []) = []$
 $map(f, Cons(a,l)) = Cons(f(a), map(f,l))$

primrec

$set-of-list([]) = 0$
 $set-of-list(Cons(a,l)) = cons(a, set-of-list(l))$

primrec

$app-Nil: [] @ ys = ys$
 $app-Cons: (Cons(a,l)) @ ys = Cons(a, l @ ys)$

consts

$rev :: i \Rightarrow i$
 $flat :: i \Rightarrow i$
 $list-add :: i \Rightarrow i$

primrec

$rev([]) = []$
 $rev(Cons(a,l)) = rev(l) @ [a]$

primrec

$flat([]) = []$
 $flat(Cons(l,ls)) = l @ flat(ls)$

primrec

$list-add([]) = 0$
 $list-add(Cons(a,l)) = a \# + list-add(l)$

consts

$drop \quad :: [i,i] \Rightarrow i$

primrec

$drop-0: \quad drop(0,l) = l$
 $drop-succ: drop(succ(i), l) = tl \ (drop(i,l))$

definition

$take \quad :: [i,i] \Rightarrow i \text{ where}$
 $take(n, as) \equiv list-rec(\lambda n \in nat. [],$
 $\quad \lambda a \ l \ r. \lambda n \in nat. nat-case([], \lambda m. Cons(a, r'm), n), as) 'n$

definition

$nth :: [i, i] \Rightarrow i \text{ where}$
 $\text{— returns the (n+1)th element of a list, or 0 if the list is too short.}$
 $nth(n, as) \equiv list-rec(\lambda n \in nat. 0,$
 $\quad \lambda a \ l \ r. \lambda n \in nat. nat-case(a, \lambda m. r'm, n), as) 'n$

definition

$list-update :: [i, i, i] \Rightarrow i \text{ where}$
 $list-update(xs, i, v) \equiv list-rec(\lambda n \in nat. Nil,$
 $\quad \lambda u \ us \ vs. \lambda n \in nat. nat-case(Cons(v, us), \lambda m. Cons(u, vs'm), n), xs) 'i$

consts

$filter :: [i \Rightarrow o, i] \Rightarrow i$
 $upt :: [i, i] \Rightarrow i$

primrec

$filter(P, Nil) = Nil$
 $filter(P, Cons(x, xs)) =$
 $\quad (if \ P(x) \ then \ Cons(x, filter(P, xs)) \ else \ filter(P, xs))$

primrec

$upt(i, 0) = Nil$
 $upt(i, succ(j)) = (if \ i \leq j \ then \ upt(i, j)@[j] \ else \ Nil)$

definition

$min :: [i,i] \Rightarrow i \text{ where}$
 $min(x, y) \equiv (if \ x \leq y \ then \ x \ else \ y)$

definition

max :: $[i, i] \Rightarrow i$ **where**
max(x, y) \equiv (*if* $x \leq y$ *then* y *else* x)

declare *list.intros* [*simp*, *TC*]

inductive-cases *ConsE*: $\text{Cons}(a, l) \in \text{list}(A)$

lemma *Cons-type-iff* [*simp*]: $\text{Cons}(a, l) \in \text{list}(A) \longleftrightarrow a \in A \wedge l \in \text{list}(A)$
by (*blast elim: ConsE*)

lemma *Cons-iff*: $\text{Cons}(a, l) = \text{Cons}(a', l') \longleftrightarrow a = a' \wedge l = l'$
by *auto*

lemma *Nil-Cons-iff*: $\neg \text{Nil} = \text{Cons}(a, l)$
by *auto*

lemma *list-unfold*: $\text{list}(A) = \{0\} + (A * \text{list}(A))$
by (*blast intro!: list.intros [unfolded list.con-defs]*
elim: list.cases [unfolded list.con-defs])

lemma *list-mono*: $A \leq B \implies \text{list}(A) \subseteq \text{list}(B)$
unfolding *list.defs*
apply (*rule lfp-mono*)
apply (*simp-all add: list.bnd-mono*)
apply (*assumption | rule univ-mono basic-monos*)
done

lemma *list-univ*: $\text{list}(\text{univ}(A)) \subseteq \text{univ}(A)$
unfolding *list.defs list.con-defs*
apply (*rule lfp-lowerbound*)
apply (*rule-tac [2] A-subset-univ [THEN univ-mono]*)
apply (*blast intro!: zero-in-univ Inl-in-univ Inr-in-univ Pair-in-univ*)
done

lemmas *list-subset-univ* = *subset-trans* [*OF list-mono list-univ*]

lemma *list-into-univ*: $\llbracket l \in \text{list}(A); A \subseteq \text{univ}(B) \rrbracket \implies l \in \text{univ}(B)$
by (*blast intro: list-subset-univ [THEN subsetD]*)

lemma *list-case-type*:

```

     $\llbracket l \in \text{list}(A);$ 
       $c \in C(\text{Nil});$ 
       $\bigwedge x y. \llbracket x \in A; y \in \text{list}(A) \rrbracket \implies h(x,y): C(\text{Cons}(x,y))$ 
 $\rrbracket \implies \text{list-case}(c,h,l) \in C(l)$ 
by (erule list.induct, auto)

```

```

lemma list-0-triv:  $\text{list}(0) = \{\text{Nil}\}$ 
apply (rule equalityI, auto)
apply (induct-tac x, auto)
done

```

```

lemma tl-type:  $l \in \text{list}(A) \implies \text{tl}(l) \in \text{list}(A)$ 
apply (induct-tac l)
apply (simp-all (no-asm-simp) add: list.intros)
done

```

```

lemma drop-Nil [simp]:  $i \in \text{nat} \implies \text{drop}(i, \text{Nil}) = \text{Nil}$ 
apply (induct-tac i)
apply (simp-all (no-asm-simp))
done

```

```

lemma drop-succ-Cons [simp]:  $i \in \text{nat} \implies \text{drop}(\text{succ}(i), \text{Cons}(a,l)) = \text{drop}(i,l)$ 
apply (rule sym)
apply (induct-tac i)
apply (simp (no-asm))
apply (simp (no-asm-simp))
done

```

```

lemma drop-type [simp, TC]:  $\llbracket i \in \text{nat}; l \in \text{list}(A) \rrbracket \implies \text{drop}(i,l) \in \text{list}(A)$ 
apply (induct-tac i)
apply (simp-all (no-asm-simp) add: tl-type)
done

```

```

declare drop-succ [simp del]

```

```

lemma list-rec-type [TC]:
   $\llbracket l \in \text{list}(A);$ 
     $c \in C(\text{Nil});$ 
     $\bigwedge x y r. \llbracket x \in A; y \in \text{list}(A); r \in C(y) \rrbracket \implies h(x,y,r): C(\text{Cons}(x,y))$ 
 $\rrbracket \implies \text{list-rec}(c,h,l) \in C(l)$ 
by (induct-tac l, auto)

```

```

lemma map-type [TC]:
   $\llbracket l \in \text{list}(A); \bigwedge x. x \in A \implies h(x) \in B \rrbracket \implies \text{map}(h, l) \in \text{list}(B)$ 
apply (simp add: map-list-def)
apply (typecheck add: list.intros list-rec-type, blast)
done

lemma map-type2 [TC]:  $l \in \text{list}(A) \implies \text{map}(h, l) \in \text{list}(\{h(u). u \in A\})$ 
apply (erule map-type)
apply (erule RepFunI)
done

lemma length-type [TC]:  $l \in \text{list}(A) \implies \text{length}(l) \in \text{nat}$ 
by (simp add: length-list-def)

lemma lt-length-in-nat:
   $\llbracket x < \text{length}(xs); xs \in \text{list}(A) \rrbracket \implies x \in \text{nat}$ 
by (frule lt-nat-in-nat, typecheck)

lemma app-type [TC]:  $\llbracket xs: \text{list}(A); ys: \text{list}(A) \rrbracket \implies xs@ys \in \text{list}(A)$ 
by (simp add: app-list-def)

lemma rev-type [TC]:  $xs: \text{list}(A) \implies \text{rev}(xs) \in \text{list}(A)$ 
by (simp add: rev-list-def)

lemma flat-type [TC]:  $ls: \text{list}(\text{list}(A)) \implies \text{flat}(ls) \in \text{list}(A)$ 
by (simp add: flat-list-def)

lemma set-of-list-type [TC]:  $l \in \text{list}(A) \implies \text{set-of-list}(l) \in \text{Pow}(A)$ 
  unfolding set-of-list-list-def
apply (erule list-rec-type, auto)
done

lemma set-of-list-append:
   $xs: \text{list}(A) \implies \text{set-of-list}(xs@ys) = \text{set-of-list}(xs) \cup \text{set-of-list}(ys)$ 

```

```

apply (erule list.induct)
apply (simp-all (no-asm-simp) add: Un-cons)
done

```

```

lemma list-add-type [TC]:  $xs: \text{list}(\text{nat}) \implies \text{list-add}(xs) \in \text{nat}$ 
by (simp add: list-add-list-def)

```

```

lemma map-ident [simp]:  $l \in \text{list}(A) \implies \text{map}(\lambda u. u, l) = l$ 
apply (induct-tac l)
apply (simp-all (no-asm-simp))
done

```

```

lemma map-compose:  $l \in \text{list}(A) \implies \text{map}(h, \text{map}(j, l)) = \text{map}(\lambda u. h(j(u)), l)$ 
apply (induct-tac l)
apply (simp-all (no-asm-simp))
done

```

```

lemma map-app-distrib:  $xs: \text{list}(A) \implies \text{map}(h, xs @ ys) = \text{map}(h, xs) @ \text{map}(h, ys)$ 
apply (induct-tac xs)
apply (simp-all (no-asm-simp))
done

```

```

lemma map-flat:  $ls: \text{list}(\text{list}(A)) \implies \text{map}(h, \text{flat}(ls)) = \text{flat}(\text{map}(\text{map}(h), ls))$ 
apply (induct-tac ls)
apply (simp-all (no-asm-simp) add: map-app-distrib)
done

```

```

lemma list-rec-map:
   $l \in \text{list}(A) \implies$ 
     $\text{list-rec}(c, d, \text{map}(h, l)) =$ 
     $\text{list-rec}(c, \lambda x \ xs \ r. d(h(x), \text{map}(h, xs), r), l)$ 
apply (induct-tac l)
apply (simp-all (no-asm-simp))
done

```

```

lemmas list-CollectD = Collect-subset [THEN list-mono, THEN subsetD]

```

```

lemma map-list-Collect:  $l \in \text{list}(\{x \in A. h(x)=j(x)\}) \implies \text{map}(h, l) = \text{map}(j, l)$ 
apply (induct-tac l)
apply (simp-all (no-asm-simp))

```

done

lemma *length-map* [*simp*]: $xs: \text{list}(A) \implies \text{length}(\text{map}(h, xs)) = \text{length}(xs)$
by (*induct-tac xs, simp-all*)

lemma *length-app* [*simp*]:
 $\llbracket xs: \text{list}(A); ys: \text{list}(A) \rrbracket$
 $\implies \text{length}(xs@ys) = \text{length}(xs) \# + \text{length}(ys)$
by (*induct-tac xs, simp-all*)

lemma *length-rev* [*simp*]: $xs: \text{list}(A) \implies \text{length}(\text{rev}(xs)) = \text{length}(xs)$
apply (*induct-tac xs*)
apply (*simp-all (no-asm-simp) add: length-app*)
done

lemma *length-flat*:
 $ls: \text{list}(\text{list}(A)) \implies \text{length}(\text{flat}(ls)) = \text{list-add}(\text{map}(\text{length}, ls))$
apply (*induct-tac ls*)
apply (*simp-all (no-asm-simp) add: length-app*)
done

lemma *drop-length-Cons* [*rule-format*]:
 $xs: \text{list}(A) \implies$
 $\forall x. \exists z zs. \text{drop}(\text{length}(xs), \text{Cons}(x, xs)) = \text{Cons}(z, zs)$
by (*erule list.induct, simp-all*)

lemma *drop-length* [*rule-format*]:
 $l \in \text{list}(A) \implies \forall i \in \text{length}(l). (\exists z zs. \text{drop}(i, l) = \text{Cons}(z, zs))$
apply (*erule list.induct, simp-all, safe*)
apply (*erule drop-length-Cons*)
apply (*rule natE*)
apply (*erule Ord-trans [OF asm-rl length-type Ord-nat], assumption, simp-all*)
apply (*blast intro: succ-in-naturalD length-type*)
done

lemma *app-right-Nil* [*simp*]: $xs: \text{list}(A) \implies xs@Nil = xs$
by (*erule list.induct, simp-all*)

lemma *app-assoc*: $xs: \text{list}(A) \implies (xs@ys)@zs = xs@(ys@zs)$
by (*induct-tac xs, simp-all*)

```

lemma flat-app-distrib:  $ls: list(list(A)) \implies flat(ls@ms) = flat(ls)@flat(ms)$ 
apply (induct-tac ls)
apply (simp-all (no-asm-simp) add: app-assoc)
done

```

```

lemma rev-map-distrib:  $l \in list(A) \implies rev(map(h,l)) = map(h,rev(l))$ 
apply (induct-tac l)
apply (simp-all (no-asm-simp) add: map-app-distrib)
done

```

```

lemma rev-app-distrib:
   $\llbracket xs: list(A); ys: list(A) \rrbracket \implies rev(xs@ys) = rev(ys)@rev(xs)$ 
apply (erule list.induct)
apply (simp-all add: app-assoc)
done

```

```

lemma rev-rev-ident [simp]:  $l \in list(A) \implies rev(rev(l))=l$ 
apply (induct-tac l)
apply (simp-all (no-asm-simp) add: rev-app-distrib)
done

```

```

lemma rev-flat:  $ls: list(list(A)) \implies rev(flat(ls)) = flat(map(rev,rev(ls)))$ 
apply (induct-tac ls)
apply (simp-all add: map-app-distrib flat-app-distrib rev-app-distrib)
done

```

```

lemma list-add-app:
   $\llbracket xs: list(nat); ys: list(nat) \rrbracket$ 
   $\implies list-add(xs@ys) = list-add(ys) \# + list-add(xs)$ 
apply (induct-tac xs, simp-all)
done

```

```

lemma list-add-rev:  $l \in list(nat) \implies list-add(rev(l)) = list-add(l)$ 
apply (induct-tac l)
apply (simp-all (no-asm-simp) add: list-add-app)
done

```

```

lemma list-add-flat:
   $ls: list(list(nat)) \implies list-add(flat(ls)) = list-add(map(list-add,ls))$ 
apply (induct-tac ls)
apply (simp-all (no-asm-simp) add: list-add-app)
done

```

lemma *list-append-induct* [*case-names Nil snoc, consumes 1*]:

$\llbracket l \in \text{list}(A);$
 $P(\text{Nil});$
 $\bigwedge x y. \llbracket x \in A; y \in \text{list}(A); P(y) \rrbracket \implies P(y @ [x])$
 $\rrbracket \implies P(l)$
apply (*subgoal-tac* $P(\text{rev}(\text{rev}(l)))$, *simp*)
apply (*erule rev-type* [*THEN list.induct*], *simp-all*)
done

lemma *list-complete-induct-lemma* [*rule-format*]:

assumes *ih*:
 $\bigwedge l. \llbracket l \in \text{list}(A);$
 $\forall l' \in \text{list}(A). \text{length}(l') < \text{length}(l) \longrightarrow P(l') \rrbracket$
 $\implies P(l)$
shows $n \in \text{nat} \implies \forall l \in \text{list}(A). \text{length}(l) < n \longrightarrow P(l)$
apply (*induct-tac* *n*, *simp*)
apply (*blast intro: ih elim!: leE*)
done

theorem *list-complete-induct*:

$\llbracket l \in \text{list}(A);$
 $\bigwedge l. \llbracket l \in \text{list}(A);$
 $\forall l' \in \text{list}(A). \text{length}(l') < \text{length}(l) \longrightarrow P(l') \rrbracket$
 $\implies P(l)$
apply (*rule list-complete-induct-lemma* [*of A*])
prefer 4 **apply** (*rule le-refl, simp*)
apply *blast*
apply *simp*
apply *assumption*
done

lemma *min-sym*: $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{min}(i,j)=\text{min}(j,i)$

unfolding *min-def*
apply (*auto dest!: not-lt-imp-le dest: lt-not-sym intro: le-anti-sym*)
done

lemma *min-type* [*simp, TC*]: $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{min}(i,j):\text{nat}$
by (*unfold min-def, auto*)

lemma *min-0* [*simp*]: $i \in \text{nat} \implies \text{min}(0,i) = 0$
unfolding *min-def*

apply (*auto dest: not-lt-imp-le*)
done

lemma *min-02* [*simp*]: $i \in \text{nat} \implies \text{min}(i, 0) = 0$
unfolding *min-def*
apply (*auto dest: not-lt-imp-le*)
done

lemma *lt-min-iff*: $\llbracket i \in \text{nat}; j \in \text{nat}; k \in \text{nat} \rrbracket \implies i < \text{min}(j, k) \longleftrightarrow i < j \wedge i < k$
unfolding *min-def*
apply (*auto dest!: not-lt-imp-le intro: lt-trans2 lt-trans*)
done

lemma *min-succ-succ* [*simp*]:
 $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{min}(\text{succ}(i), \text{succ}(j)) = \text{succ}(\text{min}(i, j))$
apply (*unfold min-def, auto*)
done

lemma *filter-append* [*simp*]:
 $xs:\text{list}(A) \implies \text{filter}(P, xs @ ys) = \text{filter}(P, xs) @ \text{filter}(P, ys)$
by (*induct-tac xs, auto*)

lemma *filter-type* [*simp, TC*]: $xs:\text{list}(A) \implies \text{filter}(P, xs):\text{list}(A)$
by (*induct-tac xs, auto*)

lemma *length-filter*: $xs:\text{list}(A) \implies \text{length}(\text{filter}(P, xs)) \leq \text{length}(xs)$
apply (*induct-tac xs, auto*)
apply (*rule-tac j = length (l) in le-trans*)
apply (*auto simp add: le-iff*)
done

lemma *filter-is-subset*: $xs:\text{list}(A) \implies \text{set-of-list}(\text{filter}(P, xs)) \subseteq \text{set-of-list}(xs)$
by (*induct-tac xs, auto*)

lemma *filter-False* [*simp*]: $xs:\text{list}(A) \implies \text{filter}(\lambda p. \text{False}, xs) = \text{Nil}$
by (*induct-tac xs, auto*)

lemma *filter-True* [*simp*]: $xs:\text{list}(A) \implies \text{filter}(\lambda p. \text{True}, xs) = xs$
by (*induct-tac xs, auto*)

lemma *length-is-0-iff* [*simp*]: $xs:\text{list}(A) \implies \text{length}(xs) = 0 \longleftrightarrow xs = \text{Nil}$
by (*erule list.induct, auto*)

lemma *length-is-0-iff2* [simp]: $xs:list(A) \implies 0 = length(xs) \longleftrightarrow xs=Nil$
by (erule list.induct, auto)

lemma *length-tl* [simp]: $xs:list(A) \implies length(tl(xs)) = length(xs) \#- 1$
by (erule list.induct, auto)

lemma *length-greater-0-iff*: $xs:list(A) \implies 0 < length(xs) \longleftrightarrow xs \neq Nil$
by (erule list.induct, auto)

lemma *length-succ-iff*: $xs:list(A) \implies length(xs)=succ(n) \longleftrightarrow (\exists y\ ys. xs=Cons(y, ys) \wedge length(ys)=n)$
by (erule list.induct, auto)

lemma *append-is-Nil-iff* [simp]:
 $xs:list(A) \implies (xs@ys = Nil) \longleftrightarrow (xs=Nil \wedge ys = Nil)$
by (erule list.induct, auto)

lemma *append-is-Nil-iff2* [simp]:
 $xs:list(A) \implies (Nil = xs@ys) \longleftrightarrow (xs=Nil \wedge ys = Nil)$
by (erule list.induct, auto)

lemma *append-left-is-self-iff* [simp]:
 $xs:list(A) \implies (xs@ys = xs) \longleftrightarrow (ys = Nil)$
by (erule list.induct, auto)

lemma *append-left-is-self-iff2* [simp]:
 $xs:list(A) \implies (xs = xs@ys) \longleftrightarrow (ys = Nil)$
by (erule list.induct, auto)

lemma *append-left-is-Nil-iff* [rule-format]:
 $\llbracket xs:list(A); ys:list(A); zs:list(A) \rrbracket \implies$
 $length(ys)=length(zs) \longrightarrow (xs@ys=zs \longleftrightarrow (xs=Nil \wedge ys=zs))$
apply (erule list.induct)
apply (auto simp add: length-app)
done

lemma *append-left-is-Nil-iff2* [rule-format]:
 $\llbracket xs:list(A); ys:list(A); zs:list(A) \rrbracket \implies$
 $length(ys)=length(zs) \longrightarrow (zs=ys@xs \longleftrightarrow (xs=Nil \wedge ys=zs))$
apply (erule list.induct)
apply (auto simp add: length-app)
done

lemma *append-eq-append-iff* [rule-format]:
 $xs:list(A) \implies \forall ys \in list(A).$

```

      length(xs)=length(ys)  $\longrightarrow$  (xs@us = ys@vs)  $\longleftrightarrow$  (xs=ys  $\wedge$  us=vs)
apply (erule list.induct)
apply (simp (no-asm-simp))
apply clarify
apply (erule-tac a = ys in list.cases, auto)
done
declare append-eq-append-iff [simp]

```

```

lemma append-eq-append [rule-format]:
  xs:list(A)  $\implies$ 
     $\forall$  ys  $\in$  list(A).  $\forall$  us  $\in$  list(A).  $\forall$  vs  $\in$  list(A).
      length(us) = length(vs)  $\longrightarrow$  (xs@us = ys@vs)  $\longrightarrow$  (xs=ys  $\wedge$  us=vs)
apply (induct-tac xs)
apply (force simp add: length-app, clarify)
apply (erule-tac a = ys in list.cases, simp)
apply (subgoal-tac Cons (a, l) @ us = vs)
apply (drule rev-iffD1 [OF - append-left-is-Nil-iff], simp-all, blast)
done

```

```

lemma append-eq-append-iff2 [simp]:
   $\llbracket$ xs:list(A); ys:list(A); us:list(A); vs:list(A); length(us)=length(vs) $\rrbracket$ 
 $\implies$  xs@us = ys@vs  $\longleftrightarrow$  (xs=ys  $\wedge$  us=vs)
apply (rule iffI)
apply (rule append-eq-append, auto)
done

```

```

lemma append-self-iff [simp]:
   $\llbracket$ xs:list(A); ys:list(A); zs:list(A) $\rrbracket \implies$  xs@ys=xs@zs  $\longleftrightarrow$  ys=zs
by simp

```

```

lemma append-self-iff2 [simp]:
   $\llbracket$ xs:list(A); ys:list(A); zs:list(A) $\rrbracket \implies$  ys@xs=zs@xs  $\longleftrightarrow$  ys=zs
by simp

```

```

lemma append1-eq-iff [rule-format]:
  xs:list(A)  $\implies \forall$  ys  $\in$  list(A). xs@[x] = ys@[y]  $\longleftrightarrow$  (xs = ys  $\wedge$  x=y)
apply (erule list.induct)
apply clarify
apply (erule list.cases)
apply simp-all

```

Inductive step

```

apply clarify
apply (erule-tac a=ys in list.cases, simp-all)
done
declare append1-eq-iff [simp]

```

```

lemma append-right-is-self-iff [simp]:

```

$\llbracket xs:\text{list}(A); ys:\text{list}(A) \rrbracket \implies (xs@ys = ys) \longleftrightarrow (xs=\text{Nil})$
by (*simp* (*no-asm-simp*) *add: append-left-is-Nil-iff*)

lemma *append-right-is-self-iff2* [*simp*]:
 $\llbracket xs:\text{list}(A); ys:\text{list}(A) \rrbracket \implies (ys = xs@ys) \longleftrightarrow (xs=\text{Nil})$
apply (*rule iffI*)
apply (*drule sym, auto*)
done

lemma *hd-append* [*rule-format*]:
 $xs:\text{list}(A) \implies xs \neq \text{Nil} \longrightarrow \text{hd}(xs @ ys) = \text{hd}(xs)$
by (*induct-tac xs, auto*)
declare *hd-append* [*simp*]

lemma *tl-append* [*rule-format*]:
 $xs:\text{list}(A) \implies xs \neq \text{Nil} \longrightarrow \text{tl}(xs @ ys) = \text{tl}(xs)@ys$
by (*induct-tac xs, auto*)
declare *tl-append* [*simp*]

lemma *rev-is-Nil-iff* [*simp*]: $xs:\text{list}(A) \implies (\text{rev}(xs) = \text{Nil} \longleftrightarrow xs = \text{Nil})$
by (*erule list.induct, auto*)

lemma *Nil-is-rev-iff* [*simp*]: $xs:\text{list}(A) \implies (\text{Nil} = \text{rev}(xs) \longleftrightarrow xs = \text{Nil})$
by (*erule list.induct, auto*)

lemma *rev-is-rev-iff* [*rule-format*]:
 $xs:\text{list}(A) \implies \forall ys \in \text{list}(A). \text{rev}(xs)=\text{rev}(ys) \longleftrightarrow xs=ys$
apply (*erule list.induct, force, clarify*)
apply (*erule-tac a = ys in list.cases, auto*)
done
declare *rev-is-rev-iff* [*simp*]

lemma *rev-list-elim* [*rule-format*]:
 $xs:\text{list}(A) \implies$
 $(xs=\text{Nil} \longrightarrow P) \longrightarrow (\forall ys \in \text{list}(A). \forall y \in A. xs = ys@[y] \longrightarrow P) \longrightarrow P$
by (*erule list-append-induct, auto*)

lemma *length-drop* [*rule-format*]:
 $n \in \text{nat} \implies \forall xs \in \text{list}(A). \text{length}(\text{drop}(n, xs)) = \text{length}(xs) \# - n$
apply (*erule nat-induct*)
apply (*auto elim: list.cases*)
done
declare *length-drop* [*simp*]

lemma *drop-all* [*rule-format*]:

```

       $n \in \text{nat} \implies \forall xs \in \text{list}(A). \text{length}(xs) \leq n \longrightarrow \text{drop}(n, xs) = \text{Nil}$ 
apply (erule nat-induct)
apply (auto elim: list.cases)
done
declare drop-all [simp]

lemma drop-append [rule-format]:
   $n \in \text{nat} \implies$ 
   $\forall xs \in \text{list}(A). \text{drop}(n, xs @ ys) = \text{drop}(n, xs) @ \text{drop}(n \# - \text{length}(xs), ys)$ 
apply (induct-tac n)
apply (auto elim: list.cases)
done

lemma drop-drop:
   $m \in \text{nat} \implies \forall xs \in \text{list}(A). \forall n \in \text{nat}. \text{drop}(n, \text{drop}(m, xs)) = \text{drop}(n \# + m, xs)$ 
apply (induct-tac m)
apply (auto elim: list.cases)
done

lemma take-0 [simp]:  $xs: \text{list}(A) \implies \text{take}(0, xs) = \text{Nil}$ 
  unfolding take-def
apply (erule list.induct, auto)
done

lemma take-succ-Cons [simp]:
   $n \in \text{nat} \implies \text{take}(\text{succ}(n), \text{Cons}(a, xs)) = \text{Cons}(a, \text{take}(n, xs))$ 
by (simp add: take-def)

lemma take-Nil [simp]:  $n \in \text{nat} \implies \text{take}(n, \text{Nil}) = \text{Nil}$ 
by (unfold take-def, auto)

lemma take-all [rule-format]:
   $n \in \text{nat} \implies \forall xs \in \text{list}(A). \text{length}(xs) \leq n \longrightarrow \text{take}(n, xs) = xs$ 
apply (erule nat-induct)
apply (auto elim: list.cases)
done
declare take-all [simp]

lemma take-type [rule-format]:
   $xs: \text{list}(A) \implies \forall n \in \text{nat}. \text{take}(n, xs): \text{list}(A)$ 
apply (erule list.induct, simp, clarify)
apply (erule natE, auto)
done
declare take-type [simp, TC]

lemma take-append [rule-format]:

```

```

 $xs: \text{list}(A) \implies$ 
 $\forall ys \in \text{list}(A). \forall n \in \text{nat}. \text{take}(n, xs @ ys) =$ 
 $\text{take}(n, xs) @ \text{take}(n \# - \text{length}(xs), ys)$ 
apply (erule list.induct, simp, clarify)
apply (erule natE, auto)
done
declare take-append [simp]

lemma take-take [rule-format]:
 $m \in \text{nat} \implies$ 
 $\forall xs \in \text{list}(A). \forall n \in \text{nat}. \text{take}(n, \text{take}(m, xs)) = \text{take}(\min(n, m), xs)$ 
apply (induct-tac m, auto)
apply (erule-tac a = xs in list.cases)
apply (auto simp add: take-Nil)
apply (erule-tac n=n in natE)
apply (auto intro: take-0 take-type)
done

lemma nth-0 [simp]:  $\text{nth}(0, \text{Cons}(a, l)) = a$ 
by (simp add: nth-def)

lemma nth-Cons [simp]:  $n \in \text{nat} \implies \text{nth}(\text{succ}(n), \text{Cons}(a, l)) = \text{nth}(n, l)$ 
by (simp add: nth-def)

lemma nth-empty [simp]:  $\text{nth}(n, \text{Nil}) = 0$ 
by (simp add: nth-def)

lemma nth-type [rule-format]:
 $xs: \text{list}(A) \implies \forall n. n < \text{length}(xs) \longrightarrow \text{nth}(n, xs) \in A$ 
apply (erule list.induct, simp, clarify)
apply (subgoal-tac  $n \in \text{nat}$ )
apply (erule natE, auto dest!: le-in-nat)
done
declare nth-type [simp, TC]

lemma nth-eq-0 [rule-format]:
 $xs: \text{list}(A) \implies \forall n \in \text{nat}. \text{length}(xs) \leq n \longrightarrow \text{nth}(n, xs) = 0$ 
apply (erule list.induct, simp, clarify)
apply (erule natE, auto)
done

lemma nth-append [rule-format]:
 $xs: \text{list}(A) \implies$ 
 $\forall n \in \text{nat}. \text{nth}(n, xs @ ys) = (\text{if } n < \text{length}(xs) \text{ then } \text{nth}(n, xs)$ 
 $\text{else } \text{nth}(n \# - \text{length}(xs), ys))$ 
apply (induct-tac xs, simp, clarify)
apply (erule natE, auto)

```

done

lemma *set-of-list-conv-nth*:

$xs: \text{list}(A)$
 $\implies \text{set-of-list}(xs) = \{x \in A. \exists i \in \text{nat}. i < \text{length}(xs) \wedge x = \text{nth}(i, xs)\}$
apply (*induct-tac* xs , *simp-all*)
apply (*rule equalityI*, *auto*)
apply (*rule-tac* $x = 0$ **in** $\text{be}xI$, *auto*)
apply (*erule natE*, *auto*)
done

lemma *nth-take-lemma* [*rule-format*]:

$k \in \text{nat} \implies$
 $\forall xs \in \text{list}(A). (\forall ys \in \text{list}(A). k \leq \text{length}(xs) \longrightarrow k \leq \text{length}(ys) \longrightarrow$
 $(\forall i \in \text{nat}. i < k \longrightarrow \text{nth}(i, xs) = \text{nth}(i, ys)) \longrightarrow \text{take}(k, xs) = \text{take}(k, ys))$
apply (*induct-tac* k)
apply (*simp-all* (*no-asm-simp*) *add: lt-succ-eq-0-disj all-conj-distrib*)
apply *clarify*

apply (*erule-tac* $a=xs$ **in** list.cases , *simp*)
apply (*erule-tac* $a=ys$ **in** list.cases , *clarify*)
apply (*simp* (*no-asm-use*))
apply *clarify*
apply (*simp* (*no-asm-simp*))
apply (*rule conjI*, *force*)
apply (*rename-tac* y ys z zs)
apply (*drule-tac* $x = zs$ **and** $x1 = ys$ **in** bspec [*THEN* bspec], *auto*)
done

lemma *nth-equalityI* [*rule-format*]:

$\llbracket xs: \text{list}(A); ys: \text{list}(A); \text{length}(xs) = \text{length}(ys);$
 $\forall i \in \text{nat}. i < \text{length}(xs) \longrightarrow \text{nth}(i, xs) = \text{nth}(i, ys) \rrbracket$
 $\implies xs = ys$
apply (*subgoal-tac* $\text{length}(xs) \leq \text{length}(ys)$)
apply (*cut-tac* $k = \text{length}(xs)$ **and** $xs = xs$ **and** $ys = ys$ **in** *nth-take-lemma*)
apply (*simp-all* *add: take-all*)
done

lemma *take-equalityI* [*rule-format*]:

$\llbracket xs: \text{list}(A); ys: \text{list}(A); (\forall i \in \text{nat}. \text{take}(i, xs) = \text{take}(i, ys)) \rrbracket$
 $\implies xs = ys$
apply (*case-tac* $\text{length}(xs) \leq \text{length}(ys)$)
apply (*drule-tac* $x = \text{length}(ys)$ **in** bspec)
apply (*drule-tac* [3] *not-lt-imp-le*)
apply (*subgoal-tac* [5] $\text{length}(ys) \leq \text{length}(xs)$)

```

apply (rule-tac [6] j = succ (length (ys)) in le-trans)
apply (rule-tac [6] leI)
apply (drule-tac [5] x = length (xs) in bspec)
apply (simp-all add: take-all)
done

```

```

lemma nth-drop [rule-format]:
  n ∈ nat ⇒ ∀ i ∈ nat. ∀ xs ∈ list(A). nth(i, drop(n, xs)) = nth(n #+ i, xs)
apply (induct-tac n, simp-all, clarify)
apply (erule list.cases, auto)
done

```

```

lemma take-succ [rule-format]:
  xs ∈ list(A)
  ⇒ ∀ i. i < length(xs) → take(succ(i), xs) = take(i, xs) @ [nth(i, xs)]
apply (induct-tac xs, auto)
apply (subgoal-tac i ∈ nat)
apply (erule natE)
apply (auto simp add: le-in-nat)
done

```

```

lemma take-add [rule-format]:
  [xs ∈ list(A); j ∈ nat]
  ⇒ ∀ i ∈ nat. take(i #+ j, xs) = take(i, xs) @ take(j, drop(i, xs))
apply (induct-tac xs, simp-all, clarify)
apply (erule-tac n = i in natE, simp-all)
done

```

```

lemma length-take:
  l ∈ list(A) ⇒ ∀ n ∈ nat. length(take(n, l)) = min(n, length(l))
apply (induct-tac l, safe, simp-all)
apply (erule natE, simp-all)
done

```

29.1 The function zip

Crafty definition to eliminate a type argument

consts

zip-aux :: [i, i] ⇒ i

primrec

zip-aux(B, []) =
 (λys ∈ list(B). list-case([], λy l. [], ys))

zip-aux(B, Cons(x, l)) =
 (λys ∈ list(B).
 list-case(Nil, λy zs. Cons(⟨x, y⟩, zip-aux(B, l) 'zs), ys))

definition


```

zip :: [i, i] ⇒ i where
  zip(xs, ys) ≡ zip-aux(set-of-list(ys),xs) 'ys

lemma list-on-set-of-list: xs ∈ list(A) ⇒ xs ∈ list(set-of-list(xs))
apply (induct-tac xs, simp-all)
apply (blast intro: list-mono [THEN subsetD])
done

lemma zip-Nil [simp]: ys:list(A) ⇒ zip(Nil, ys)=Nil
apply (simp add: zip-def list-on-set-of-list [of - A])
apply (erule list.cases, simp-all)
done

lemma zip-Nil2 [simp]: xs:list(A) ⇒ zip(xs, Nil)=Nil
apply (simp add: zip-def list-on-set-of-list [of - A])
apply (erule list.cases, simp-all)
done

lemma zip-aux-unique [rule-format]:
  ⌊B ≤ C; xs ∈ list(A)⌋
  ⇒ ∀ ys ∈ list(B). zip-aux(C,xs) ' ys = zip-aux(B,xs) ' ys
apply (induct-tac xs)
apply simp-all
apply (blast intro: list-mono [THEN subsetD], clarify)
apply (erule-tac a=ys in list.cases, auto)
apply (blast intro: list-mono [THEN subsetD])
done

lemma zip-Cons-Cons [simp]:
  ⌊xs:list(A); ys:list(B); x ∈ A; y ∈ B⌋ ⇒
  zip(Cons(x,xs), Cons(y, ys)) = Cons(⟨x,y⟩, zip(xs, ys))
apply (simp add: zip-def, auto)
apply (rule zip-aux-unique, auto)
apply (simp add: list-on-set-of-list [of - B])
apply (blast intro: list-on-set-of-list list-mono [THEN subsetD])
done

lemma zip-type [rule-format]:
  xs:list(A) ⇒ ∀ ys ∈ list(B). zip(xs, ys):list(A*B)
apply (induct-tac xs)
apply (simp (no-asm))
apply clarify
apply (erule-tac a = ys in list.cases, auto)
done
declare zip-type [simp,TC]

```

```

lemma length-zip [rule-format]:
   $xs: \text{list}(A) \implies \forall ys \in \text{list}(B). \text{length}(\text{zip}(xs, ys)) =$ 
     $\min(\text{length}(xs), \text{length}(ys))$ 

  unfolding min-def
apply (induct-tac xs, simp-all, clarify)
apply (erule-tac a = ys in list.cases, auto)
done
declare length-zip [simp]

lemma zip-append1 [rule-format]:
   $\llbracket ys: \text{list}(A); zs: \text{list}(B) \rrbracket \implies$ 
     $\forall xs \in \text{list}(A). \text{zip}(xs @ ys, zs) =$ 
     $\text{zip}(xs, \text{take}(\text{length}(xs), zs)) @ \text{zip}(ys, \text{drop}(\text{length}(xs), zs))$ 
apply (induct-tac zs, force, clarify)
apply (erule-tac a = xs in list.cases, simp-all)
done

lemma zip-append2 [rule-format]:
   $\llbracket xs: \text{list}(A); zs: \text{list}(B) \rrbracket \implies \forall ys \in \text{list}(B). \text{zip}(xs, ys @ zs) =$ 
     $\text{zip}(\text{take}(\text{length}(ys), xs), ys) @ \text{zip}(\text{drop}(\text{length}(ys), xs), zs)$ 
apply (induct-tac xs, force, clarify)
apply (erule-tac a = ys in list.cases, auto)
done

lemma zip-append [simp]:
   $\llbracket \text{length}(xs) = \text{length}(us); \text{length}(ys) = \text{length}(vs);$ 
     $xs: \text{list}(A); us: \text{list}(B); ys: \text{list}(A); vs: \text{list}(B) \rrbracket$ 
     $\implies \text{zip}(xs @ ys, us @ vs) = \text{zip}(xs, us) @ \text{zip}(ys, vs)$ 
by (simp (no-asm-simp) add: zip-append1 drop-append diff-self-eq-0)

lemma zip-rev [rule-format]:
   $ys: \text{list}(B) \implies \forall xs \in \text{list}(A).$ 
     $\text{length}(xs) = \text{length}(ys) \longrightarrow \text{zip}(\text{rev}(xs), \text{rev}(ys)) = \text{rev}(\text{zip}(xs, ys))$ 
apply (induct-tac ys, force, clarify)
apply (erule-tac a = xs in list.cases)
apply (auto simp add: length-rev)
done
declare zip-rev [simp]

lemma nth-zip [rule-format]:
   $ys: \text{list}(B) \implies \forall i \in \text{nat}. \forall xs \in \text{list}(A).$ 
     $i < \text{length}(xs) \longrightarrow i < \text{length}(ys) \longrightarrow$ 
     $\text{nth}(i, \text{zip}(xs, ys)) = \langle \text{nth}(i, xs), \text{nth}(i, ys) \rangle$ 
apply (induct-tac ys, force, clarify)
apply (erule-tac a = xs in list.cases, simp)
apply (auto elim: natE)
done

```

declare *nth-zip* [*simp*]

lemma *set-of-list-zip* [*rule-format*]:

$\llbracket xs: \text{list}(A); ys: \text{list}(B); i \in \text{nat} \rrbracket$
 $\implies \text{set-of-list}(\text{zip}(xs, ys)) =$
 $\{ \langle x, y \rangle : A * B. \exists i \in \text{nat}. i < \min(\text{length}(xs), \text{length}(ys))$
 $\wedge x = \text{nth}(i, xs) \wedge y = \text{nth}(i, ys) \}$

by (*force intro!*: *Collect-cong simp add: lt-min-iff set-of-list-conv-nth*)

lemma *list-update-Nil* [*simp*]: $i \in \text{nat} \implies \text{list-update}(\text{Nil}, i, v) = \text{Nil}$

by (*unfold list-update-def, auto*)

lemma *list-update-Cons-0* [*simp*]: $\text{list-update}(\text{Cons}(x, xs), 0, v) = \text{Cons}(v, xs)$

by (*unfold list-update-def, auto*)

lemma *list-update-Cons-succ* [*simp*]:

$n \in \text{nat} \implies$

$\text{list-update}(\text{Cons}(x, xs), \text{succ}(n), v) = \text{Cons}(x, \text{list-update}(xs, n, v))$

apply (*unfold list-update-def, auto*)

done

lemma *list-update-type* [*rule-format*]:

$\llbracket xs: \text{list}(A); v \in A \rrbracket \implies \forall n \in \text{nat}. \text{list-update}(xs, n, v): \text{list}(A)$

apply (*induct-tac xs*)

apply (*simp (no-asm)*)

apply *clarify*

apply (*erule natE, auto*)

done

declare *list-update-type* [*simp, TC*]

lemma *length-list-update* [*rule-format*]:

$xs: \text{list}(A) \implies \forall i \in \text{nat}. \text{length}(\text{list-update}(xs, i, v)) = \text{length}(xs)$

apply (*induct-tac xs*)

apply (*simp (no-asm)*)

apply *clarify*

apply (*erule natE, auto*)

done

declare *length-list-update* [*simp*]

lemma *nth-list-update* [*rule-format*]:

$\llbracket xs: \text{list}(A) \rrbracket \implies \forall i \in \text{nat}. \forall j \in \text{nat}. i < \text{length}(xs) \longrightarrow$
 $\text{nth}(j, \text{list-update}(xs, i, x)) = (\text{if } i=j \text{ then } x \text{ else } \text{nth}(j, xs))$

apply (*induct-tac xs*)

apply *simp-all*

apply *clarify*

apply (*rename-tac i j*)

apply (*erule-tac n=i in natE*)

```

apply (erule-tac [2] n=j in natE)
apply (erule-tac n=j in natE, simp-all, force)
done

```

```

lemma nth-list-update-eq [simp]:
   $\llbracket i < \text{length}(xs); xs:\text{list}(A) \rrbracket \implies \text{nth}(i, \text{list-update}(xs, i, x)) = x$ 
by (simp (no-asm-simp) add: lt-length-in-nat nth-list-update)

```

```

lemma nth-list-update-neq [rule-format]:
   $xs:\text{list}(A) \implies$ 
   $\forall i \in \text{nat}. \forall j \in \text{nat}. i \neq j \longrightarrow \text{nth}(j, \text{list-update}(xs, i, x)) = \text{nth}(j, xs)$ 
apply (induct-tac xs)
apply (simp (no-asm))
apply clarify
apply (erule natE)
apply (erule-tac [2] natE, simp-all)
apply (erule natE, simp-all)
done
declare nth-list-update-neq [simp]

```

```

lemma list-update-overwrite [rule-format]:
   $xs:\text{list}(A) \implies \forall i \in \text{nat}. i < \text{length}(xs)$ 
   $\longrightarrow \text{list-update}(\text{list-update}(xs, i, x), i, y) = \text{list-update}(xs, i, y)$ 
apply (induct-tac xs)
apply (simp (no-asm))
apply clarify
apply (erule natE, auto)
done
declare list-update-overwrite [simp]

```

```

lemma list-update-same-conv [rule-format]:
   $xs:\text{list}(A) \implies$ 
   $\forall i \in \text{nat}. i < \text{length}(xs) \longrightarrow$ 
   $(\text{list-update}(xs, i, x) = xs) \longleftrightarrow (\text{nth}(i, xs) = x)$ 
apply (induct-tac xs)
apply (simp (no-asm))
apply clarify
apply (erule natE, auto)
done

```

```

lemma update-zip [rule-format]:
   $ys:\text{list}(B) \implies$ 
   $\forall i \in \text{nat}. \forall xy \in A*B. \forall xs \in \text{list}(A).$ 
   $\text{length}(xs) = \text{length}(ys) \longrightarrow$ 
   $\text{list-update}(\text{zip}(xs, ys), i, xy) = \text{zip}(\text{list-update}(xs, i, \text{fst}(xy)),$ 
   $\text{list-update}(ys, i, \text{snd}(xy)))$ 
apply (induct-tac ys)
apply auto

```

```

apply (erule-tac a = xs in list.cases)
apply (auto elim: natE)
done

```

```

lemma set-update-subset-cons [rule-format]:
  xs:list(A)  $\implies$ 
   $\forall i \in \text{nat}. \text{set-of-list}(\text{list-update}(xs, i, x)) \subseteq \text{cons}(x, \text{set-of-list}(xs))$ 
apply (induct-tac xs)
apply simp
apply (rule ballI)
apply (erule natE, simp-all, auto)
done

```

```

lemma set-of-list-update-subsetI:
   $\llbracket \text{set-of-list}(xs) \subseteq A; xs:\text{list}(A); x \in A; i \in \text{nat} \rrbracket$ 
 $\implies \text{set-of-list}(\text{list-update}(xs, i, x)) \subseteq A$ 
apply (rule subset-trans)
apply (rule set-update-subset-cons, auto)
done

```

```

lemma upt-rec:
   $j \in \text{nat} \implies \text{upt}(i, j) = (\text{if } i < j \text{ then } \text{Cons}(i, \text{upt}(\text{succ}(i), j)) \text{ else } \text{Nil})$ 
apply (induct-tac j, auto)
apply (drule not-lt-imp-le)
apply (auto simp: lt-Ord intro: le-anti-sym)
done

```

```

lemma upt-conv-Nil [simp]:  $\llbracket j \leq i; j \in \text{nat} \rrbracket \implies \text{upt}(i, j) = \text{Nil}$ 
apply (subst upt-rec, auto)
apply (auto simp add: le-iff)
apply (drule lt-asym [THEN notE], auto)
done

```

```

lemma upt-succ-append:
   $\llbracket i \leq j; j \in \text{nat} \rrbracket \implies \text{upt}(i, \text{succ}(j)) = \text{upt}(i, j) @ [j]$ 
by simp

```

```

lemma upt-conv-Cons:
   $\llbracket i < j; j \in \text{nat} \rrbracket \implies \text{upt}(i, j) = \text{Cons}(i, \text{upt}(\text{succ}(i), j))$ 
apply (rule trans)
apply (rule upt-rec, auto)
done

```

```

lemma upt-type [simp, TC]:  $j \in \text{nat} \implies \text{upt}(i, j) : \text{list}(\text{nat})$ 
by (induct-tac j, auto)

```

```

lemma upt-add-eq-append:
   $\llbracket i \leq j; j \in \text{nat}; k \in \text{nat} \rrbracket \implies \text{upt}(i, j \# + k) = \text{upt}(i, j) @ \text{upt}(j, j \# + k)$ 
apply (induct-tac k)
apply (auto simp add: app-assoc app-type)
apply (rule-tac j = j in le-trans, auto)
done

lemma length-upt [simp]:  $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{length}(\text{upt}(i, j)) = j \# - i$ 
apply (induct-tac j)
apply (rule-tac [?] sym)
apply (auto dest!: not-lt-imp-le simp add: diff-succ diff-is-0-iff)
done

lemma nth-upt [simp]:
   $\llbracket i \in \text{nat}; j \in \text{nat}; k \in \text{nat}; i \# + k < j \rrbracket \implies \text{nth}(k, \text{upt}(i, j)) = i \# + k$ 
apply (rotate-tac  $-1$ , erule rev-mp)
apply (induct-tac j, simp)
apply (auto dest!: not-lt-imp-le
  simp add: nth-append le-iff less-diff-conv add-commute)
done

lemma take-upt [rule-format]:
   $\llbracket m \in \text{nat}; n \in \text{nat} \rrbracket \implies$ 
   $\forall i \in \text{nat}. i \# + m \leq n \longrightarrow \text{take}(m, \text{upt}(i, n)) = \text{upt}(i, i \# + m)$ 
apply (induct-tac m)
apply (simp (no-asm-simp) add: take-0)
apply clarify
apply (subst upt-rec, simp)
apply (rule sym)
apply (subst upt-rec, simp)
apply (simp-all del: upt.simps)
apply (rule-tac j = succ (i \# + x) in lt-trans2)
apply auto
done
declare take-upt [simp]

lemma map-succ-upt:
   $\llbracket m \in \text{nat}; n \in \text{nat} \rrbracket \implies \text{map}(\text{succ}, \text{upt}(m, n)) = \text{upt}(\text{succ}(m), \text{succ}(n))$ 
apply (induct-tac n)
apply (auto simp add: map-app-distrib)
done

lemma nth-map [rule-format]:
   $xs:\text{list}(A) \implies$ 
   $\forall n \in \text{nat}. n < \text{length}(xs) \longrightarrow \text{nth}(n, \text{map}(f, xs)) = f(\text{nth}(n, xs))$ 
apply (induct-tac xs, simp)
apply (rule ballI)
apply (induct-tac n, auto)

```

```

done
declare nth-map [simp]

lemma nth-map-upt [rule-format]:
   $\llbracket m \in \text{nat}; n \in \text{nat} \rrbracket \implies$ 
   $\forall i \in \text{nat}. i < n \#- m \longrightarrow \text{nth}(i, \text{map}(f, \text{upt}(m, n))) = f(m \#+ i)$ 
  apply (rule-tac  $n = m$  and  $m = n$  in diff-induct, typecheck, simp, simp)
  apply (subst map-succ-upt [symmetric], simp-all, clarify)
  apply (subgoal-tac  $i < \text{length}(\text{upt}(0, x))$ )
  prefer 2
  apply (simp add: less-diff-conv)
  apply (rule-tac  $j = \text{succ}(i \#+ y)$  in lt-trans2)
  apply simp
  apply simp
  apply (subgoal-tac  $i < \text{length}(\text{upt}(y, x))$ )
  apply (simp-all add: add-commute less-diff-conv)
done

```

```

definition
  sublist ::  $[i, i] \Rightarrow i$  where
    sublist(xs, A)  $\equiv$ 
      map(fst, (filter( $\lambda p. \text{snd}(p): A$ , zip(xs, upt(0, length(xs)))))

```

```

lemma sublist-0 [simp]:  $xs:\text{list}(A) \implies \text{sublist}(xs, 0) = \text{Nil}$ 
by (unfold sublist-def, auto)

```

```

lemma sublist-Nil [simp]:  $\text{sublist}(\text{Nil}, A) = \text{Nil}$ 
by (unfold sublist-def, auto)

```

```

lemma sublist-shift-lemma:
   $\llbracket xs:\text{list}(B); i \in \text{nat} \rrbracket \implies$ 
   $\text{map}(\text{fst}, \text{filter}(\lambda p. \text{snd}(p): A, \text{zip}(xs, \text{upt}(i, i \#+ \text{length}(xs))))) =$ 
   $\text{map}(\text{fst}, \text{filter}(\lambda p. \text{snd}(p): \text{nat} \wedge \text{snd}(p) \#+ i \in A, \text{zip}(xs, \text{upt}(0, \text{length}(xs)))))$ 
  apply (erule list-append-induct)
  apply (simp (no-asm-simp))
  apply (auto simp add: add-commute length-app filter-append map-app-distrib)
done

```

```

lemma sublist-type [simp, TC]:
   $xs:\text{list}(B) \implies \text{sublist}(xs, A):\text{list}(B)$ 
  unfolding sublist-def
  apply (induct-tac xs)
  apply (auto simp add: filter-append map-app-distrib)
done

```

```

lemma upt-add-eq-append2:
   $\llbracket i \in \text{nat}; j \in \text{nat} \rrbracket \implies \text{upt}(0, i \#+ j) = \text{upt}(0, i) @ \text{upt}(i, i \#+ j)$ 

```

by (simp add: upt-add-eq-append [of 0] nat-0-le)

lemma *sublist-append*:

$\llbracket xs: \text{list}(B); ys: \text{list}(B) \rrbracket \implies$
 $\text{sublist}(xs @ ys, A) = \text{sublist}(xs, A) @ \text{sublist}(ys, \{j \in \text{nat}. j \# + \text{length}(xs): A\})$
 unfolding *sublist-def*
 apply (erule-tac l = ys in list-append-induct, simp)
 apply (simp (no-asm-simp) add: upt-add-eq-append2 app-assoc [symmetric])
 apply (auto simp add: sublist-shift-lemma length-type map-app-distrib app-assoc)
 apply (simp-all add: add-commute)
 done

lemma *sublist-Cons*:

$\llbracket xs: \text{list}(B); x \in B \rrbracket \implies$
 $\text{sublist}(\text{Cons}(x, xs), A) =$
 $(\text{if } 0 \in A \text{ then } [x] \text{ else } []) @ \text{sublist}(xs, \{j \in \text{nat}. \text{succ}(j) \in A\})$
 apply (erule-tac l = xs in list-append-induct)
 apply (simp (no-asm-simp) add: sublist-def)
 apply (simp del: app-Cons add: app-Cons [symmetric] sublist-append, simp)
 done

lemma *sublist-singleton* [simp]:

$\text{sublist}([x], A) = (\text{if } 0 \in A \text{ then } [x] \text{ else } [])$
 by (simp add: sublist-Cons)

lemma *sublist-upt-eq-take* [rule-format]:

$xs: \text{list}(A) \implies \forall n \in \text{nat}. \text{sublist}(xs, n) = \text{take}(n, xs)$
 apply (erule list.induct, simp)
 apply (clarify)
 apply (erule natE)
 apply (simp-all add: nat-eq-Collect-lt Ord-mem-iff-lt sublist-Cons)
 done
 declare *sublist-upt-eq-take* [simp]

lemma *sublist-Int-eq*:

$xs \in \text{list}(B) \implies \text{sublist}(xs, A \cap \text{nat}) = \text{sublist}(xs, A)$
 apply (erule list.induct)
 apply (simp-all add: sublist-Cons)
 done

Repetition of a List Element

consts *repeat* :: $[i, i] \Rightarrow i$

primrec

$\text{repeat}(a, 0) = []$

$\text{repeat}(a, \text{succ}(n)) = \text{Cons}(a, \text{repeat}(a, n))$

lemma *length-repeat*: $n \in \text{nat} \implies \text{length}(\text{repeat}(a, n)) = n$

by (*induct-tac* *n*, *auto*)

lemma *repeat-succ-app*: $n \in \text{nat} \implies \text{repeat}(a, \text{succ}(n)) = \text{repeat}(a, n) @ [a]$
apply (*induct-tac* *n*)
apply (*simp-all* *del*: *app-Cons* *add*: *app-Cons* [*symmetric*])
done

lemma *repeat-type* [*TC*]: $\llbracket a \in A; n \in \text{nat} \rrbracket \implies \text{repeat}(a, n) \in \text{list}(A)$
by (*induct-tac* *n*, *auto*)

end

30 Equivalence Relations

theory *EquivClass* **imports** *Trancl Perm* **begin**

definition

quotient :: $[i, i] \Rightarrow i$ (**infixl** $\langle ' / ' \rangle$ 90) **where**
 $A / r \equiv \{ r^{-1} \{ x \} \mid x \in A \}$

definition

congruent :: $[i, i \Rightarrow i] \Rightarrow o$ **where**
 $\text{congruent}(r, b) \equiv \forall y z. \langle y, z \rangle : r \longrightarrow b(y) = b(z)$

definition

congruent2 :: $[i, i, [i, i] \Rightarrow i] \Rightarrow o$ **where**
 $\text{congruent2}(r1, r2, b) \equiv \forall y1 z1 y2 z2. \\ \langle y1, z1 \rangle : r1 \longrightarrow \langle y2, z2 \rangle : r2 \longrightarrow b(y1, y2) = b(z1, z2)$

abbreviation

RESPECTS :: $[i \Rightarrow i, i] \Rightarrow o$ (**infixr** $\langle \text{respects} \rangle$ 80) **where**
 $f \text{ respects } r \equiv \text{congruent}(r, f)$

abbreviation

RESPECTS2 :: $[i \Rightarrow i \Rightarrow i, i] \Rightarrow o$ (**infixr** $\langle \text{respects2} \rangle$ 80) **where**
 $f \text{ respects2 } r \equiv \text{congruent2}(r, r, f)$
— Abbreviation for the common case where the relations are identical

30.1 Suppes, Theorem 70: r is an equiv relation iff $\text{converse}(r) \circ r = r$

lemma *sym-trans-comp-subset*:

$\llbracket \text{sym}(r); \text{trans}(r) \rrbracket \implies \text{converse}(r) \circ r \subseteq r$
by (*unfold trans-def sym-def*, *blast*)

lemma *refl-comp-subset*:

$\llbracket \text{refl}(A, r); r \subseteq A * A \rrbracket \implies r \subseteq \text{converse}(r) \circ r$
by (*unfold refl-def*, *blast*)

lemma *equiv-comp-eq*:
 $\text{equiv}(A, r) \implies \text{converse}(r) \text{ } O \text{ } r = r$
unfolding *equiv-def*
apply (*blast del: subsetI intro!: sym-trans-comp-subset refl-comp-subset*)
done

lemma *comp-equivI*:
 $\llbracket \text{converse}(r) \text{ } O \text{ } r = r; \text{ domain}(r) = A \rrbracket \implies \text{equiv}(A, r)$
unfolding *equiv-def refl-def sym-def trans-def*
apply (*erule equalityE*)
apply (*subgoal-tac $\forall x y. \langle x, y \rangle \in r \longrightarrow \langle y, x \rangle \in r$, blast+*)
done

lemma *equiv-class-subset*:
 $\llbracket \text{sym}(r); \text{ trans}(r); \langle a, b \rangle: r \rrbracket \implies r^{\{\{a\}\}} \subseteq r^{\{\{b\}\}}$
by (*unfold trans-def sym-def, blast*)

lemma *equiv-class-eq*:
 $\llbracket \text{equiv}(A, r); \langle a, b \rangle: r \rrbracket \implies r^{\{\{a\}\}} = r^{\{\{b\}\}}$
unfolding *equiv-def*
apply (*safe del: subsetI intro!: equalityI equiv-class-subset*)
apply (*unfold sym-def, blast*)
done

lemma *equiv-class-self*:
 $\llbracket \text{equiv}(A, r); a \in A \rrbracket \implies a \in r^{\{\{a\}\}}$
by (*unfold equiv-def refl-def, blast*)

lemma *subset-equiv-class*:
 $\llbracket \text{equiv}(A, r); r^{\{\{b\}\}} \subseteq r^{\{\{a\}\}}; b \in A \rrbracket \implies \langle a, b \rangle: r$
by (*unfold equiv-def refl-def, blast*)

lemma *eq-equiv-class*: $\llbracket r^{\{\{a\}\}} = r^{\{\{b\}\}}; \text{equiv}(A, r); b \in A \rrbracket \implies \langle a, b \rangle: r$
by (*assumption | rule equalityD2 subset-equiv-class*)
+

lemma *equiv-class-nondisjoint*:
 $\llbracket \text{equiv}(A, r); x: (r^{\{\{a\}\}} \cap r^{\{\{b\}\}}) \rrbracket \implies \langle a, b \rangle: r$
by (*unfold equiv-def trans-def sym-def, blast*)

lemma *equiv-type*: $\text{equiv}(A, r) \implies r \subseteq A * A$
by (*unfold equiv-def, blast*)

lemma *equiv-class-eq-iff*:

$equiv(A,r) \implies \langle x,y \rangle: r \longleftrightarrow r''\{x\} = r''\{y\} \wedge x \in A \wedge y \in A$
by (*blast intro: eq-equiv-class equiv-class-eq dest: equiv-type*)

lemma *eq-equiv-class-iff*:

$\llbracket equiv(A,r); x \in A; y \in A \rrbracket \implies r''\{x\} = r''\{y\} \longleftrightarrow \langle x,y \rangle: r$
by (*blast intro: eq-equiv-class equiv-class-eq dest: equiv-type*)

lemma *quotientI* [*TC*]: $x \in A \implies r''\{x\}: A//r$

unfolding *quotient-def*
apply (*erule RepFunI*)
done

lemma *quotientE*:

$\llbracket X \in A//r; \bigwedge x. \llbracket X = r''\{x\}; x \in A \rrbracket \implies P \rrbracket \implies P$
by (*unfold quotient-def, blast*)

lemma *Union-quotient*:

$equiv(A,r) \implies \bigcup (A//r) = A$
by (*unfold equiv-def refl-def quotient-def, blast*)

lemma *quotient-disj*:

$\llbracket equiv(A,r); X \in A//r; Y \in A//r \rrbracket \implies X=Y \mid (X \cap Y \subseteq \emptyset)$
unfolding *quotient-def*
apply (*safe intro!: equiv-class-eq, assumption*)
apply (*unfold equiv-def trans-def sym-def, blast*)
done

30.2 Defining Unary Operations upon Equivalence Classes

lemma *UN-equiv-class*:

$\llbracket equiv(A,r); b \text{ respects } r; a \in A \rrbracket \implies (\bigcup x \in r''\{a\}. b(x)) = b(a)$
apply (*subgoal-tac $\forall x \in r''\{a\}. b(x) = b(a)$*)
apply *simp*
apply (*blast intro: equiv-class-self*)
apply (*unfold equiv-def sym-def congruent-def, blast*)
done

lemma *UN-equiv-class-type*:

$\llbracket equiv(A,r); b \text{ respects } r; X \in A//r; \bigwedge x. x \in A \implies b(x) \in B \rrbracket$
 $\implies (\bigcup x \in X. b(x)) \in B$
apply (*unfold quotient-def, safe*)
apply (*simp (no-asm-simp) add: UN-equiv-class*)
done

lemma *UN-equiv-class-inject*:
 $\llbracket \text{equiv}(A,r); \quad b \text{ respects } r; \quad (\bigcup_{x \in X} b(x)) = (\bigcup_{y \in Y} b(y)); \quad X \in A//r; \quad Y \in A//r; \quad \bigwedge x y. \llbracket x \in A; y \in A; b(x) = b(y) \rrbracket \implies \langle x,y \rangle : r \rrbracket$
 $\implies X = Y$
apply (*unfold quotient-def, safe*)
apply (*rule equiv-class-eq, assumption*)
apply (*simp add: UN-equiv-class [of A r b]*)
done

30.3 Defining Binary Operations upon Equivalence Classes

lemma *congruent2-implies-congruent*:
 $\llbracket \text{equiv}(A,r1); \quad \text{congruent2}(r1,r2,b); \quad a \in A \rrbracket \implies \text{congruent}(r2,b(a))$
by (*unfold congruent-def congruent2-def equiv-def refl-def, blast*)

lemma *congruent2-implies-congruent-UN*:
 $\llbracket \text{equiv}(A1,r1); \quad \text{equiv}(A2,r2); \quad \text{congruent2}(r1,r2,b); \quad a \in A2 \rrbracket \implies$
 $\text{congruent}(r1, \lambda x1. \bigcup x2 \in r2. \{a\}. b(x1,x2))$
apply (*unfold congruent-def, safe*)
apply (*frule equiv-type [THEN subsetD], assumption*)
apply *clarify*
apply (*simp add: UN-equiv-class congruent2-implies-congruent*)
apply (*unfold congruent2-def equiv-def refl-def, blast*)
done

lemma *UN-equiv-class2*:
 $\llbracket \text{equiv}(A1,r1); \quad \text{equiv}(A2,r2); \quad \text{congruent2}(r1,r2,b); \quad a1: A1; \quad a2: A2 \rrbracket$
 $\implies (\bigcup x1 \in r1. \{a1\}. \bigcup x2 \in r2. \{a2\}. b(x1,x2)) = b(a1,a2)$
by (*simp add: UN-equiv-class congruent2-implies-congruent*
congruent2-implies-congruent-UN)

lemma *UN-equiv-class-type2*:
 $\llbracket \text{equiv}(A,r); \quad b \text{ respects2 } r; \quad X1: A//r; \quad X2: A//r; \quad \bigwedge x1 x2. \llbracket x1: A; x2: A \rrbracket \implies b(x1,x2) \in B \rrbracket$
 $\implies (\bigcup x1 \in X1. \bigcup x2 \in X2. b(x1,x2)) \in B$
apply (*unfold quotient-def, safe*)
apply (*blast intro: UN-equiv-class-type congruent2-implies-congruent-UN*
congruent2-implies-congruent quotientI)
done

lemma *congruent2I*:
 $\llbracket \text{equiv}(A1,r1); \quad \text{equiv}(A2,r2); \quad \bigwedge y z w. \llbracket w \in A2; \quad \langle y,z \rangle \in r1 \rrbracket \implies b(y,w) = b(z,w);$

```

       $\bigwedge y \ z \ w. \llbracket w \in A1; \langle y, z \rangle \in r2 \rrbracket \implies b(w, y) = b(w, z)$ 
 $\rrbracket \implies \text{congruent2}(r1, r2, b)$ 
apply (unfold congruent2-def equiv-def refl-def, safe)
apply (blast intro: trans)
done

```

```

lemma congruent2-commuteI:
  assumes equivA: equiv(A, r)
    and commute:  $\bigwedge y \ z. \llbracket y \in A; \ z \in A \rrbracket \implies b(y, z) = b(z, y)$ 
    and cong:  $\bigwedge y \ z \ w. \llbracket w \in A; \langle y, z \rangle: r \rrbracket \implies b(w, y) = b(w, z)$ 
  shows b respects2 r
apply (insert equivA [THEN equiv-type, THEN subsetD])
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 (blast intro: cong)+
done

```

```

lemma congruent-commuteI:
   $\llbracket \text{equiv}(A, r); \ Z \in A / r; \$ 
     $\bigwedge w. \llbracket w \in A \rrbracket \implies \text{congruent}(r, \lambda z. b(w, z));$ 
     $\bigwedge x \ y. \llbracket x \in A; \ y \in A \rrbracket \implies b(y, x) = b(x, y)$ 
 $\rrbracket \implies \text{congruent}(r, \lambda w. \bigcup_{z \in Z.} b(w, z))$ 
apply (simp (no-asm) add: congruent-def)
apply (safe elim!: quotientE)
apply (frule equiv-type [THEN subsetD], assumption)
apply (simp add: UN-equiv-class [of A r])
apply (simp add: congruent-def)
done

```

end

31 The Integers as Equivalence Classes Over Pairs of Natural Numbers

theory *Int* **imports** *EquivClass ArithSimp* **begin**

definition

```

  intrel :: i where
    intrel  $\equiv \{p \in (\text{nat} * \text{nat}) * (\text{nat} * \text{nat}).$ 
       $\exists x1 \ y1 \ x2 \ y2. p = \langle \langle x1, y1 \rangle, \langle x2, y2 \rangle \rangle \wedge x1 \# + y2 = x2 \# + y1 \}$ 

```

definition

```

  int :: i where
    int  $\equiv (\text{nat} * \text{nat}) / \text{intrel}$ 

```

definition

$int\text{-}of :: i \Rightarrow i$ — coercion from nat to int $(\langle \langle open\text{-}block\ notation = \langle prefix\ \$\# \rangle \rangle \$\# - \rangle \rangle [80] 80)$
where $\$ \# m \equiv intrel\ \langle \langle natify(m), 0 \rangle \rangle$

definition

$intify :: i \Rightarrow i$ — coercion from ANYTHING to int **where**
 $intify(m) \equiv if\ m \in int\ then\ m\ else\ \$\# 0$

definition

$raw\text{-}zminus :: i \Rightarrow i$ **where**
 $raw\text{-}zminus(z) \equiv \bigcup \langle x, y \rangle \in z. intrel\ \langle \langle y, x \rangle \rangle$

definition

$zminus :: i \Rightarrow i$ $(\langle \langle open\text{-}block\ notation = \langle prefix\ \$- \rangle \rangle \$- - \rangle \rangle [80] 80)$
where $\$ - z \equiv raw\text{-}zminus\ (intify(z))$

definition

$znegative :: i \Rightarrow o$ **where**
 $znegative(z) \equiv \exists x\ y. x < y \wedge y \in nat \wedge \langle x, y \rangle \in z$

definition

$iszero :: i \Rightarrow o$ **where**
 $iszero(z) \equiv z = \$\# 0$

definition

$raw\text{-}nat\text{-}of :: i \Rightarrow i$ **where**
 $raw\text{-}nat\text{-}of(z) \equiv natify\ (\bigcup \langle x, y \rangle \in z. x \# -y)$

definition

$nat\text{-}of :: i \Rightarrow i$ **where**
 $nat\text{-}of(z) \equiv raw\text{-}nat\text{-}of\ (intify(z))$

definition

$zmagnitude :: i \Rightarrow i$ **where**
 — could be replaced by an absolute value function from int to int?
 $zmagnitude(z) \equiv$
 $THE\ m. m \in nat \wedge ((\neg\ znegative(z) \wedge z = \$\# m) \mid$
 $(znegative(z) \wedge \$ - z = \$\# m))$

definition

$raw\text{-}zmult :: [i, i] \Rightarrow i$ **where**
 $raw\text{-}zmult(z1, z2) \equiv$
 $\bigcup p1 \in z1. \bigcup p2 \in z2. split(\lambda x1\ y1. split(\lambda x2\ y2.$
 $intrel\ \langle \langle x1 \# * x2 \# + y1 \# * y2, x1 \# * y2 \# + y1 \# * x2 \rangle \rangle, p2), p1)$

definition

$zmult :: [i, i] \Rightarrow i$ **(infixl $\langle \$* \rangle 70$) where**

$$z1 \$* z2 \equiv \text{raw-zmult } (\text{intify}(z1), \text{intify}(z2))$$

definition

$$\begin{aligned} \text{raw-zadd} &:: [i, i] \Rightarrow i \text{ where} \\ \text{raw-zadd } (z1, z2) &\equiv \\ &\bigcup z1 \in z1. \bigcup z2 \in z2. \text{ let } \langle x1, y1 \rangle = z1; \langle x2, y2 \rangle = z2 \\ &\text{ in } \text{intrel}''\{\langle x1 \# + x2, y1 \# + y2 \rangle\} \end{aligned}$$

definition

$$\begin{aligned} \text{zadd} &:: [i, i] \Rightarrow i \quad (\text{infixl } \langle \$+ \rangle 65) \text{ where} \\ z1 \$+ z2 &\equiv \text{raw-zadd } (\text{intify}(z1), \text{intify}(z2)) \end{aligned}$$

definition

$$\begin{aligned} \text{zdiff} &:: [i, i] \Rightarrow i \quad (\text{infixl } \langle \$- \rangle 65) \text{ where} \\ z1 \$- z2 &\equiv z1 \$+ \text{zminus}(z2) \end{aligned}$$

definition

$$\begin{aligned} \text{zless} &:: [i, i] \Rightarrow o \quad (\text{infixl } \langle \$< \rangle 50) \text{ where} \\ z1 \$< z2 &\equiv \text{znegative}(z1 \$- z2) \end{aligned}$$

definition

$$\begin{aligned} \text{zle} &:: [i, i] \Rightarrow o \quad (\text{infixl } \langle \$\leq \rangle 50) \text{ where} \\ z1 \$\leq z2 &\equiv z1 \$< z2 \mid \text{intify}(z1) = \text{intify}(z2) \end{aligned}$$

declare *quotientE* [elim!]

31.1 Proving that *intrel* is an equivalence relation

lemma *intrel-iff* [simp]:

$$\begin{aligned} &\langle \langle x1, y1 \rangle, \langle x2, y2 \rangle \rangle >: \text{intrel} \longleftrightarrow \\ &x1 \in \text{nat} \wedge y1 \in \text{nat} \wedge x2 \in \text{nat} \wedge y2 \in \text{nat} \wedge x1 \# + y2 = x2 \# + y1 \end{aligned}$$

by (simp add: *intrel-def*)

lemma *intrelI* [intro!]:

$$\begin{aligned} &\llbracket x1 \# + y2 = x2 \# + y1; x1 \in \text{nat}; y1 \in \text{nat}; x2 \in \text{nat}; y2 \in \text{nat} \rrbracket \\ &\implies \langle \langle x1, y1 \rangle, \langle x2, y2 \rangle \rangle >: \text{intrel} \end{aligned}$$

by (simp add: *intrel-def*)

lemma *intrelE* [elim!]:

$$\begin{aligned} &\llbracket p \in \text{intrel}; \\ &\quad \wedge x1 \ y1 \ x2 \ y2. \llbracket p = \langle \langle x1, y1 \rangle, \langle x2, y2 \rangle \rangle >; x1 \# + y2 = x2 \# + y1; \\ &\quad x1 \in \text{nat}; y1 \in \text{nat}; x2 \in \text{nat}; y2 \in \text{nat} \rrbracket \implies Q \rrbracket \end{aligned}$$

$$\implies Q$$

by (simp add: *intrel-def*, *blast*)

lemma *int-trans-lemma*:

$$\llbracket x1 \# + y2 = x2 \# + y1; x2 \# + y3 = x3 \# + y2 \rrbracket \implies x1 \# + y3 = x3 \# + y1$$

```

apply (rule sym)
apply (erule add-left-cancel)+
apply (simp-all (no-asm-simp))
done

```

```

lemma equiv-intrel: equiv(nat*nat, intrel)
apply (simp add: equiv-def refl-def sym-def trans-def)
apply (fast elim!: sym int-trans-lemma)
done

```

```

lemma image-intrel-int:  $\llbracket m \in \text{nat}; n \in \text{nat} \rrbracket \implies \text{intrel} \text{ `` } \{ \langle m, n \rangle \} \in \text{int}$ 
by (simp add: int-def)

```

```

declare equiv-intrel [THEN eq-equiv-class-iff, simp]
declare conj-cong [cong]

```

```

lemmas eq-intrelD = eq-equiv-class [OF - equiv-intrel]

```

```

lemma int-of-type [simp, TC]:  $\$ \# m \in \text{int}$ 
by (simp add: int-def quotient-def int-of-def, auto)

```

```

lemma int-of-eq [iff]:  $(\$ \# m = \$ \# n) \longleftrightarrow \text{nativify}(m) = \text{nativify}(n)$ 
by (simp add: int-of-def)

```

```

lemma int-of-inject:  $\llbracket \$ \# m = \$ \# n; m \in \text{nat}; n \in \text{nat} \rrbracket \implies m = n$ 
by (drule int-of-eq [THEN iffD1], auto)

```

```

lemma intify-in-int [iff, TC]:  $\text{intify}(x) \in \text{int}$ 
by (simp add: intify-def)

```

```

lemma intify-ident [simp]:  $n \in \text{int} \implies \text{intify}(n) = n$ 
by (simp add: intify-def)

```

31.2 Collapsing rules: to remove *intify* from arithmetic expressions

```

lemma intify-idem [simp]:  $\text{intify}(\text{intify}(x)) = \text{intify}(x)$ 
by simp

```

```

lemma int-of-nativify [simp]:  $\$ \# (\text{nativify}(m)) = \$ \# m$ 
by (simp add: int-of-def)

```

```

lemma zminus-intify [simp]:  $\$ - (\text{intify}(m)) = \$ - m$ 
by (simp add: zminus-def)

```


lemma *zadd-intify1* [*simp*]: *intify*(*x*) \$+ *y* = *x* \$+ *y*
by (*simp add: zadd-def*)

lemma *zadd-intify2* [*simp*]: *x* \$+ *intify*(*y*) = *x* \$+ *y*
by (*simp add: zadd-def*)

lemma *zdiff-intify1* [*simp*]: *intify*(*x*) \$- *y* = *x* \$- *y*
by (*simp add: zdiff-def*)

lemma *zdiff-intify2* [*simp*]: *x* \$- *intify*(*y*) = *x* \$- *y*
by (*simp add: zdiff-def*)

lemma *zmult-intify1* [*simp*]: *intify*(*x*) \$* *y* = *x* \$* *y*
by (*simp add: zmult-def*)

lemma *zmult-intify2* [*simp*]: *x* \$* *intify*(*y*) = *x* \$* *y*
by (*simp add: zmult-def*)

lemma *zless-intify1* [*simp*]: *intify*(*x*) \$< *y* \longleftrightarrow *x* \$< *y*
by (*simp add: zless-def*)

lemma *zless-intify2* [*simp*]: *x* \$< *intify*(*y*) \longleftrightarrow *x* \$< *y*
by (*simp add: zless-def*)

lemma *zle-intify1* [*simp*]: *intify*(*x*) \$≤ *y* \longleftrightarrow *x* \$≤ *y*
by (*simp add: zle-def*)

lemma *zle-intify2* [*simp*]: *x* \$≤ *intify*(*y*) \longleftrightarrow *x* \$≤ *y*
by (*simp add: zle-def*)

31.3 *zminus*: unary negation on *int*

lemma *zminus-congruent*: ($\lambda\langle x,y \rangle. \text{intrel} \langle \langle y,x \rangle \rangle$) respects *intrel*
by (*auto simp add: congruent-def add-ac*)

lemma *raw-zminus-type*: $z \in \text{int} \implies \text{raw-zminus}(z) \in \text{int}$
apply (*simp add: int-def raw-zminus-def*)
apply (*typecheck add: UN-equiv-class-type [OF equiv-intrel zminus-congruent]*)
done

lemma *zminus-type* [*TC,iff*]: $\$-z \in \text{int}$
by (*simp add: zminus-def raw-zminus-type*)

lemma *raw-zminus-inject*:
 $\llbracket \text{raw-zminus}(z) = \text{raw-zminus}(w); z \in \text{int}; w \in \text{int} \rrbracket \implies z=w$
apply (*simp add: int-def raw-zminus-def*)
apply (*erule UN-equiv-class-inject [OF equiv-intrel zminus-congruent], safe*)
apply (*auto dest: eq-intrelD simp add: add-ac*)
done

lemma *zminus-inject-intify* [*dest!*]: $\$-z = \$-w \implies \text{intify}(z) = \text{intify}(w)$
apply (*simp add: zminus-def*)
apply (*blast dest!: raw-zminus-inject*)
done

lemma *zminus-inject*: $\llbracket \$-z = \$-w; z \in \text{int}; w \in \text{int} \rrbracket \implies z=w$
by *auto*

lemma *raw-zminus*:
 $\llbracket x \in \text{nat}; y \in \text{nat} \rrbracket \implies \text{raw-zminus}(\text{intrel} \langle \langle x, y \rangle \rangle) = \text{intrel} \langle \langle y, x \rangle \rangle$
apply (*simp add: raw-zminus-def UN-equiv-class [OF equiv-intrel zminus-congruent]*)
done

lemma *zminus*:
 $\llbracket x \in \text{nat}; y \in \text{nat} \rrbracket \implies \$- (\text{intrel} \langle \langle x, y \rangle \rangle) = \text{intrel} \langle \langle y, x \rangle \rangle$
by (*simp add: zminus-def raw-zminus image-intrel-int*)

lemma *raw-zminus-zminus*: $z \in \text{int} \implies \text{raw-zminus} (\text{raw-zminus}(z)) = z$
by (*auto simp add: int-def raw-zminus*)

lemma *zminus-zminus-intify* [*simp*]: $\$- (\$- z) = \text{intify}(z)$
by (*simp add: zminus-def raw-zminus-type raw-zminus-zminus*)

lemma *zminus-int0* [*simp*]: $\$- (\$ \# 0) = \$ \# 0$
by (*simp add: int-of-def zminus*)

lemma *zminus-zminus*: $z \in \text{int} \implies \$- (\$- z) = z$
by *simp*

31.4 *znegative*: the test for negative integers

lemma *znegative*: $\llbracket x \in \text{nat}; y \in \text{nat} \rrbracket \implies \text{znegative}(\text{intrel} \langle \langle x, y \rangle \rangle) \longleftrightarrow x < y$
apply (*cases x < y*)
apply (*auto simp add: znegative-def not-lt-iff-le*)
apply (*subgoal-tac y #+ x2 < x #+ y2, force*)
apply (*rule add-le-lt-mono, auto*)
done

lemma *not-znegative-int-of [iff]*: $\neg \text{znegative}(\$ \# n)$

by (*simp add: znegative int-of-def*)

lemma *znegative-zminus-int-of [simp]*: $\text{znegative}(\$ - \$ \# \text{succ}(n))$

by (*simp add: znegative int-of-def zminus natify-succ*)

lemma *not-znegative-imp-zero*: $\neg \text{znegative}(\$ - \$ \# n) \implies \text{natify}(n)=0$

by (*simp add: znegative int-of-def zminus Ord-0-lt-iff [THEN iff-sym]*)

31.5 *nat-of*: Coercion of an Integer to a Natural Number

lemma *nat-of-intify [simp]*: $\text{nat-of}(\text{intify}(z)) = \text{nat-of}(z)$

by (*simp add: nat-of-def*)

lemma *nat-of-congruent*: $(\lambda x. (\lambda \langle x, y \rangle. x \# - y)(x))$ respects *intrel*

by (*auto simp add: congruent-def split: nat-diff-split*)

lemma *raw-nat-of*:

$\llbracket x \in \text{nat}; y \in \text{nat} \rrbracket \implies \text{raw-nat-of}(\text{intrel}''\{\langle x, y \rangle\}) = x \# - y$

by (*simp add: raw-nat-of-def UN-equiv-class [OF equiv-intrel nat-of-congruent]*)

lemma *raw-nat-of-int-of*: $\text{raw-nat-of}(\$ \# n) = \text{natify}(n)$

by (*simp add: int-of-def raw-nat-of*)

lemma *nat-of-int-of [simp]*: $\text{nat-of}(\$ \# n) = \text{natify}(n)$

by (*simp add: raw-nat-of-int-of nat-of-def*)

lemma *raw-nat-of-type*: $\text{raw-nat-of}(z) \in \text{nat}$

by (*simp add: raw-nat-of-def*)

lemma *nat-of-type [iff, TC]*: $\text{nat-of}(z) \in \text{nat}$

by (*simp add: nat-of-def raw-nat-of-type*)

31.6 *zmagnitude*: magnitide of an integer, as a natural number

lemma *zmagnitude-int-of [simp]*: $\text{zmagnitude}(\$ \# n) = \text{natify}(n)$

by (*auto simp add: zmagnitude-def int-of-eq*)

lemma *natify-int-of-eq*: $\text{natify}(x)=n \implies \$ \# x = \$ \# n$

apply (*drule sym*)

apply (*simp (no-asm-simp) add: int-of-eq*)

done

lemma *zmagnitude-zminus-int-of [simp]*: $\text{zmagnitude}(\$ - \$ \# n) = \text{natify}(n)$

apply (*simp add: zmagnitude-def*)

apply (*rule the-equality*)

apply (*auto dest!: not-znegative-imp-zero natify-int-of-eq*)

```

      iff del: int-of-eq, auto)
done

lemma zmagnitude-type [iff, TC]: zmagnitude(z) ∈ nat
apply (simp add: zmagnitude-def)
apply (rule theI2, auto)
done

lemma not-zneg-int-of:
   $\llbracket z \in \text{int}; \neg \text{znegative}(z) \rrbracket \implies \exists n \in \text{nat}. z = \$\# \ n$ 
apply (auto simp add: int-def znegative int-of-def not-lt-iff-le)
apply (rename-tac x y)
apply (rule-tac x = x#-y in bexI)
apply (auto simp add: add-diff-inverse2)
done

lemma not-zneg-mag [simp]:
   $\llbracket z \in \text{int}; \neg \text{znegative}(z) \rrbracket \implies \$\# \ (\text{zmagnitude}(z)) = z$ 
by (drule not-zneg-int-of, auto)

lemma zneg-int-of:
   $\llbracket \text{znegative}(z); z \in \text{int} \rrbracket \implies \exists n \in \text{nat}. z = \$- \ (\#\ \text{succ}(n))$ 
by (auto simp add: int-def znegative zminus int-of-def dest!: less-imp-succ-add)

lemma zneg-mag [simp]:
   $\llbracket \text{znegative}(z); z \in \text{int} \rrbracket \implies \$\# \ (\text{zmagnitude}(z)) = \$- \ z$ 
by (drule zneg-int-of, auto)

lemma int-cases:  $z \in \text{int} \implies \exists n \in \text{nat}. z = \$\# \ n \mid z = \$- \ (\#\ \text{succ}(n))$ 
apply (case-tac znegative (z) )
prefer 2 apply (blast dest: not-zneg-mag sym)
apply (blast dest: zneg-int-of)
done

lemma not-zneg-raw-nat-of:
   $\llbracket \neg \text{znegative}(z); z \in \text{int} \rrbracket \implies \$\# \ (\text{raw-nat-of}(z)) = z$ 
apply (drule not-zneg-int-of)
apply (auto simp add: raw-nat-of-type raw-nat-of-int-of)
done

lemma not-zneg-nat-of-intify:
   $\neg \text{znegative}(\text{intify}(z)) \implies \$\# \ (\text{nat-of}(z)) = \text{intify}(z)$ 
by (simp (no-asm-simp) add: nat-of-def not-zneg-raw-nat-of)

lemma not-zneg-nat-of:  $\llbracket \neg \text{znegative}(z); z \in \text{int} \rrbracket \implies \$\# \ (\text{nat-of}(z)) = z$ 
apply (simp (no-asm-simp) add: not-zneg-nat-of-intify)
done

lemma zneg-nat-of [simp]:  $\text{znegative}(\text{intify}(z)) \implies \text{nat-of}(z) = 0$ 

```

```

apply (subgoal-tac intify( $z \in \text{int}$ ))
apply (simp add: int-def)
apply (auto simp add: znegative nat-of-def raw-nat-of
      split: nat-diff-split)
done

```

31.7 ($\$+$): addition on int

Congruence Property for Addition

```

lemma zadd-congruent2:
  ( $\lambda z1\ z2. \text{let } \langle x1, y1 \rangle = z1; \langle x2, y2 \rangle = z2$ 
    in intrel “ $\{ \langle x1 \# + x2, y1 \# + y2 \rangle \}$ ”)
    respects2 intrel
apply (simp add: congruent2-def)

```

```

apply safe
apply (simp (no-asm-simp) add: add-assoc Let-def)

```

```

apply (rule-tac  $m1 = x1a$  in add-left-commute [THEN ssubst])
apply (rule-tac  $m1 = x2a$  in add-left-commute [THEN ssubst])
apply (simp (no-asm-simp) add: add-assoc [symmetric])
done

```

```

lemma raw-zadd-type:  $\llbracket z \in \text{int};\ w \in \text{int} \rrbracket \implies \text{raw-zadd}(z, w) \in \text{int}$ 
apply (simp add: int-def raw-zadd-def)
apply (rule UN-equiv-class-type2 [OF equiv-intrel zadd-congruent2], assumption+)
apply (simp add: Let-def)
done

```

```

lemma zadd-type [iff, TC]:  $z\ \$+\ w \in \text{int}$ 
by (simp add: zadd-def raw-zadd-type)

```

```

lemma raw-zadd:
   $\llbracket x1 \in \text{nat};\ y1 \in \text{nat};\ x2 \in \text{nat};\ y2 \in \text{nat} \rrbracket$ 
     $\implies \text{raw-zadd} (\text{intrel “}\{ \langle x1, y1 \rangle \}\text{”}, \text{intrel “}\{ \langle x2, y2 \rangle \}\text{”}) =$ 
      intrel “ $\{ \langle x1 \# + x2, y1 \# + y2 \rangle \}$ ”
apply (simp add: raw-zadd-def
      UN-equiv-class2 [OF equiv-intrel equiv-intrel zadd-congruent2])
apply (simp add: Let-def)
done

```

```

lemma zadd:
   $\llbracket x1 \in \text{nat};\ y1 \in \text{nat};\ x2 \in \text{nat};\ y2 \in \text{nat} \rrbracket$ 
     $\implies (\text{intrel “}\{ \langle x1, y1 \rangle \}\text{”})\ \$+\ (\text{intrel “}\{ \langle x2, y2 \rangle \}\text{”}) =$ 
      intrel “ $\{ \langle x1 \# + x2, y1 \# + y2 \rangle \}$ ”
by (simp add: zadd-def raw-zadd image-intrel-int)

```

```

lemma raw-zadd-int0:  $z \in \text{int} \implies \text{raw-zadd} (\$0, z) = z$ 
by (auto simp add: int-def int-of-def raw-zadd)

```

lemma *zadd-int0-intify* [*simp*]: $\$ \# 0 \ \$ + z = \text{intify}(z)$
by (*simp add: zadd-def raw-zadd-int0*)

lemma *zadd-int0*: $z \in \text{int} \implies \$ \# 0 \ \$ + z = z$
by *simp*

lemma *raw-zminus-zadd-distrib*:
 $\llbracket z \in \text{int}; \ w \in \text{int} \rrbracket \implies \$ - \text{raw-zadd}(z, w) = \text{raw-zadd}(\$ - z, \$ - w)$
by (*auto simp add: zminus raw-zadd int-def*)

lemma *zminus-zadd-distrib* [*simp*]: $\$ - (z \ \$ + w) = \$ - z \ \$ + \$ - w$
by (*simp add: zadd-def raw-zminus-zadd-distrib*)

lemma *raw-zadd-commute*:
 $\llbracket z \in \text{int}; \ w \in \text{int} \rrbracket \implies \text{raw-zadd}(z, w) = \text{raw-zadd}(w, z)$
by (*auto simp add: raw-zadd add-ac int-def*)

lemma *zadd-commute*: $z \ \$ + w = w \ \$ + z$
by (*simp add: zadd-def raw-zadd-commute*)

lemma *raw-zadd-assoc*:
 $\llbracket z1: \text{int}; \ z2: \text{int}; \ z3: \text{int} \rrbracket$
 $\implies \text{raw-zadd}(\text{raw-zadd}(z1, z2), z3) = \text{raw-zadd}(z1, \text{raw-zadd}(z2, z3))$
by (*auto simp add: int-def raw-zadd add-assoc*)

lemma *zadd-assoc*: $(z1 \ \$ + z2) \ \$ + z3 = z1 \ \$ + (z2 \ \$ + z3)$
by (*simp add: zadd-def raw-zadd-type raw-zadd-assoc*)

lemma *zadd-left-commute*: $z1 \ \$ + (z2 \ \$ + z3) = z2 \ \$ + (z1 \ \$ + z3)$
apply (*simp add: zadd-assoc [symmetric]*)
apply (*simp add: zadd-commute*)
done

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

lemma *int-of-add*: $\$ \# (m \ \$ + n) = (\$ \# m) \ \$ + (\$ \# n)$
by (*simp add: int-of-def zadd*)

lemma *int-succ-int-1*: $\$ \# \text{succ}(m) = \$ \# 1 \ \$ + (\$ \# m)$
by (*simp add: int-of-add [symmetric] natify-succ*)

lemma *int-of-diff*:
 $\llbracket m \in \text{nat}; \ n \leq m \rrbracket \implies \$ \# (m \ \$ - n) = (\$ \# m) \ \$ - (\$ \# n)$
apply (*simp add: int-of-def zdiff-def*)
apply (*frule lt-nat-in-nat*)
apply (*simp-all add: zadd zminus add-diff-inverse2*)

done

lemma *raw-zadd-zminus-inverse*: $z \in \text{int} \implies \text{raw-zadd } (z, \$- z) = \$\#0$
by (*auto simp add: int-def int-of-def zminus raw-zadd add-commute*)

lemma *zadd-zminus-inverse* [*simp*]: $z \$+ (\$- z) = \$\#0$
apply (*simp add: zadd-def*)
apply (*subst zminus-intify [symmetric]*)
apply (*rule intify-in-int [THEN raw-zadd-zminus-inverse]*)
done

lemma *zadd-zminus-inverse2* [*simp*]: $(\$- z) \$+ z = \$\#0$
by (*simp add: zadd-commute zadd-zminus-inverse*)

lemma *zadd-int0-right-intify* [*simp*]: $z \$+ \$\#0 = \text{intify}(z)$
by (*rule trans [OF zadd-commute zadd-int0-intify]*)

lemma *zadd-int0-right*: $z \in \text{int} \implies z \$+ \$\#0 = z$
by *simp*

31.8 $(\$*)$: Integer Multiplication

Congruence property for multiplication

lemma *zmult-congruent2*:
 $(\lambda p1\ p2. \text{split}(\lambda x1\ y1. \text{split}(\lambda x2\ y2. \text{intrel}\{\langle x1 \# * x2 \ \# + \ y1 \# * y2, x1 \# * y2 \ \# + \ y1 \# * x2 \rangle\}, p2), p1))$
respects2 intrel
apply (*rule equiv-intrel [THEN congruent2-commuteI], auto*)

apply (*rename-tac x y*)
apply (*frule-tac t = $\lambda u. x \# * u$ in sym [THEN subst-context]*)
apply (*drule-tac t = $\lambda u. y \# * u$ in subst-context*)
apply (*erule add-left-cancel*)
apply (*simp-all add: add-mult-distrib-left*)
done

lemma *raw-zmult-type*: $\llbracket z \in \text{int};\ w \in \text{int} \rrbracket \implies \text{raw-zmult}(z, w) \in \text{int}$
apply (*simp add: int-def raw-zmult-def*)
apply (*rule UN-equiv-class-type2 [OF equiv-intrel zmult-congruent2], assumption+*)
apply (*simp add: Let-def*)
done

lemma *zmult-type* [*iff, TC*]: $z \$* w \in \text{int}$
by (*simp add: zmult-def raw-zmult-type*)

lemma *raw-zmult*:
 $\llbracket x1 \in \text{nat};\ y1 \in \text{nat};\ x2 \in \text{nat};\ y2 \in \text{nat} \rrbracket$
 $\implies \text{raw-zmult}(\text{intrel}\{\langle x1, y1 \rangle\}, \text{intrel}\{\langle x2, y2 \rangle\}) =$

$\text{intrel } \{ \langle x1 \# * x2 \# + y1 \# * y2, x1 \# * y2 \# + y1 \# * x2 \rangle \}$
by (*simp add: raw-zmult-def*
 $\text{UN-equiv-class2 [OF equiv-intrel equiv-intrel zmult-congruent2]}$)

lemma *zmult:*

$\llbracket x1 \in \text{nat}; y1 \in \text{nat}; x2 \in \text{nat}; y2 \in \text{nat} \rrbracket$
 $\implies (\text{intrel } \{ \langle x1, y1 \rangle \}) \# * (\text{intrel } \{ \langle x2, y2 \rangle \}) =$
 $\text{intrel } \{ \langle x1 \# * x2 \# + y1 \# * y2, x1 \# * y2 \# + y1 \# * x2 \rangle \}$
by (*simp add: zmult-def raw-zmult image-intrel-int*)

lemma *raw-zmult-int0:* $z \in \text{int} \implies \text{raw-zmult } (\$ \# 0, z) = \$ \# 0$
by (*auto simp add: int-def int-of-def raw-zmult*)

lemma *zmult-int0* [*simp*]: $\$ \# 0 \# * z = \$ \# 0$
by (*simp add: zmult-def raw-zmult-int0*)

lemma *raw-zmult-int1:* $z \in \text{int} \implies \text{raw-zmult } (\$ \# 1, z) = z$
by (*auto simp add: int-def int-of-def raw-zmult*)

lemma *zmult-int1-intify* [*simp*]: $\$ \# 1 \# * z = \text{intify}(z)$
by (*simp add: zmult-def raw-zmult-int1*)

lemma *zmult-int1:* $z \in \text{int} \implies \$ \# 1 \# * z = z$
by *simp*

lemma *raw-zmult-commute:*

$\llbracket z \in \text{int}; w \in \text{int} \rrbracket \implies \text{raw-zmult}(z, w) = \text{raw-zmult}(w, z)$
by (*auto simp add: int-def raw-zmult add-ac mult-ac*)

lemma *zmult-commute:* $z \# * w = w \# * z$
by (*simp add: zmult-def raw-zmult-commute*)

lemma *raw-zmult-zminus:*

$\llbracket z \in \text{int}; w \in \text{int} \rrbracket \implies \text{raw-zmult}(\$ - z, w) = \$ - \text{raw-zmult}(z, w)$
by (*auto simp add: int-def zminus raw-zmult add-ac*)

lemma *zmult-zminus* [*simp*]: $(\$ - z) \# * w = \$ - (z \# * w)$
apply (*simp add: zmult-def raw-zmult-zminus*)
apply (*subst zminus-intify [symmetric], rule raw-zmult-zminus, auto*)
done

lemma *zmult-zminus-right* [*simp*]: $w \# * (\$ - z) = \$ - (w \# * z)$
by (*simp add: zmult-commute [of w]*)

lemma *raw-zmult-assoc:*

$\llbracket z1: \text{int}; z2: \text{int}; z3: \text{int} \rrbracket$
 $\implies \text{raw-zmult } (\text{raw-zmult}(z1, z2), z3) = \text{raw-zmult}(z1, \text{raw-zmult}(z2, z3))$
by (*auto simp add: int-def raw-zmult add-mult-distrib-left add-ac mult-ac*)

lemma *zmult-assoc*: $(z1 \ \$* \ z2) \ \$* \ z3 = z1 \ \$* \ (z2 \ \$* \ z3)$
by (*simp add: zmult-def raw-zmult-type raw-zmult-assoc*)

lemma *zmult-left-commute*: $z1 \ \$* \ (z2 \ \$* \ z3) = z2 \ \$* \ (z1 \ \$* \ z3)$
apply (*simp add: zmult-assoc [symmetric]*)
apply (*simp add: zmult-commute*)
done

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

lemma *raw-zadd-zmult-distrib*:
 $\llbracket z1: int; \ z2: int; \ w \in int \rrbracket$
 $\implies \text{raw-zmult}(\text{raw-zadd}(z1, z2), w) =$
 $\text{raw-zadd}(\text{raw-zmult}(z1, w), \text{raw-zmult}(z2, w))$
by (*auto simp add: int-def raw-zadd raw-zmult add-mult-distrib-left add-ac mult-ac*)

lemma *zadd-zmult-distrib*: $(z1 \ \$+ \ z2) \ \$* \ w = (z1 \ \$* \ w) \ \$+ \ (z2 \ \$* \ w)$
by (*simp add: zmult-def zadd-def raw-zadd-type raw-zmult-type raw-zadd-zmult-distrib*)

lemma *zadd-zmult-distrib2*: $w \ \$* \ (z1 \ \$+ \ z2) = (w \ \$* \ z1) \ \$+ \ (w \ \$* \ z2)$
by (*simp add: zmult-commute [of w] zadd-zmult-distrib*)

lemmas *int-typechecks =*
int-of-type zminus-type zmagnitude-type zadd-type zmult-type

lemma *zdiff-type [iff, TC]*: $z \ \$- \ w \in int$
by (*simp add: zdiff-def*)

lemma *zminus-zdiff-eq [simp]*: $\$- \ (z \ \$- \ y) = y \ \$- \ z$
by (*simp add: zdiff-def zadd-commute*)

lemma *zdiff-zmult-distrib*: $(z1 \ \$- \ z2) \ \$* \ w = (z1 \ \$* \ w) \ \$- \ (z2 \ \$* \ w)$
apply (*simp add: zdiff-def*)
apply (*subst zadd-zmult-distrib*)
apply (*simp add: zmult-zminus*)
done

lemma *zdiff-zmult-distrib2*: $w \ \$* \ (z1 \ \$- \ z2) = (w \ \$* \ z1) \ \$- \ (w \ \$* \ z2)$
by (*simp add: zmult-commute [of w] zdiff-zmult-distrib*)

lemma *zadd-zdiff-eq*: $x \ \$+ \ (y \ \$- \ z) = (x \ \$+ \ y) \ \$- \ z$
by (*simp add: zdiff-def zadd-ac*)

lemma *zdiff-zadd-eq*: $(x \$- y) \$+ z = (x \$+ z) \$- y$
by (*simp add: zdiff-def zadd-ac*)

31.9 The "Less Than" Relation

lemma *zless-linear-lemma*:
 $\llbracket z \in \text{int}; w \in \text{int} \rrbracket \implies z \$< w \mid z = w \mid w \$< z$
apply (*simp add: int-def zless-def znegative-def zdiff-def, auto*)
apply (*simp add: zadd zminus image-iff Bex-def*)
apply (*rule-tac i = xb# + ya and j = xc# + y in Ord-linear-lt*)
apply (*force dest!: spec simp add: add-ac*)
done

lemma *zless-linear*: $z \$< w \mid \text{intify}(z) = \text{intify}(w) \mid w \$< z$
apply (*cut-tac z = intify (z) and w = intify (w) in zless-linear-lemma*)
apply *auto*
done

lemma *zless-not-refl [iff]*: $\neg (z \$< z)$
by (*auto simp add: zless-def znegative-def int-of-def zdiff-def*)

lemma *neq-iff-zless*: $\llbracket x \in \text{int}; y \in \text{int} \rrbracket \implies (x \neq y) \longleftrightarrow (x \$< y \mid y \$< x)$
by (*cut-tac z = x and w = y in zless-linear, auto*)

lemma *zless-imp-intify-neq*: $w \$< z \implies \text{intify}(w) \neq \text{intify}(z)$
apply *auto*
apply (*subgoal-tac $\neg (\text{intify } (w) \$< \text{intify } (z))$*)
apply (*erule-tac [2] ssubst*)
apply (*simp (no-asm-use)*)
apply *auto*
done

lemma *zless-imp-succ-zadd-lemma*:
 $\llbracket w \$< z; w \in \text{int}; z \in \text{int} \rrbracket \implies (\exists n \in \text{nat}. z = w \$+ \$\#(\text{succ}(n)))$
apply (*simp add: zless-def znegative-def zdiff-def int-def*)
apply (*auto dest!: less-imp-succ-add simp add: zadd zminus int-of-def*)
apply (*rule-tac x = k in bexI*)
apply (*erule-tac i=succ (v) for v in add-left-cancel, auto*)
done

lemma *zless-imp-succ-zadd*:
 $w \$< z \implies (\exists n \in \text{nat}. w \$+ \$\#(\text{succ}(n)) = \text{intify}(z))$
apply (*subgoal-tac intify (w) \\$< intify (z)*)
apply (*drule-tac w = intify (w) in zless-imp-succ-zadd-lemma*)
apply *auto*
done

lemma *zless-succ-zadd-lemma*:

```

     $w \in \text{int} \implies w \$< w \$+ \$\# \text{succ}(n)$ 
  apply (simp add: zless-def znegative-def zdiff-def int-def)
  apply (auto simp add: zadd zminus int-of-def image-iff)
  apply (rule-tac  $x = 0$  in exI, auto)
done

```

```

lemma zless-succ-zadd:  $w \$< w \$+ \$\# \text{succ}(n)$ 
by (cut-tac intify-in-int [THEN zless-succ-zadd-lemma], auto)

```

```

lemma zless-iff-succ-zadd:
   $w \$< z \longleftrightarrow (\exists n \in \text{nat}. w \$+ \$\#(\text{succ}(n)) = \text{intify}(z))$ 
  apply (rule iffI)
  apply (erule zless-imp-succ-zadd, auto)
  apply (rename-tac  $n$ )
  apply (cut-tac  $w = w$  and  $n = n$  in zless-succ-zadd, auto)
done

```

```

lemma zless-int-of [simp]:  $\llbracket m \in \text{nat}; n \in \text{nat} \rrbracket \implies (\$ \# m \$< \$ \# n) \longleftrightarrow (m < n)$ 
  apply (simp add: less-iff-succ-add zless-iff-succ-zadd int-of-add [symmetric])
  apply (blast intro: sym)
done

```

```

lemma zless-trans-lemma:
   $\llbracket x \$< y; y \$< z; x \in \text{int}; y \in \text{int}; z \in \text{int} \rrbracket \implies x \$< z$ 
  apply (simp add: zless-def znegative-def zdiff-def int-def)
  apply (auto simp add: zadd zminus image-iff)
  apply (rename-tac  $x1\ x2\ y1\ y2$ )
  apply (rule-tac  $x = x1 \$+ x2$  in exI)
  apply (rule-tac  $x = y1 \$+ y2$  in exI)
  apply (auto simp add: add-lt-mono)
  apply (rule sym)
  apply hypsubst-thin
  apply (erule add-left-cancel)+
  apply auto
done

```

```

lemma zless-trans [trans]:  $\llbracket x \$< y; y \$< z \rrbracket \implies x \$< z$ 
  apply (subgoal-tac intify (x) $< intify (z) )
  apply (rule-tac [2]  $y = \text{intify}(y)$  in zless-trans-lemma)
  apply auto
done

```

```

lemma zless-not-sym:  $z \$< w \implies \neg (w \$< z)$ 
by (blast dest: zless-trans)

```

```

lemmas zless-asm = zless-not-sym [THEN swap]

```

```

lemma zless-imp-zle:  $z \$< w \implies z \$\leq w$ 

```

by (*simp add: zle-def*)

lemma *zle-linear*: $z \leq w \mid w \leq z$
apply (*simp add: zle-def*)
apply (*cut-tac zless-linear, blast*)
done

31.10 Less Than or Equals

lemma *zle-refl*: $z \leq z$
by (*simp add: zle-def*)

lemma *zle-eq-refl*: $x=y \implies x \leq y$
by (*simp add: zle-refl*)

lemma *zle-anti-sym-intify*: $\llbracket x \leq y; y \leq x \rrbracket \implies \text{intify}(x) = \text{intify}(y)$
apply (*simp add: zle-def, auto*)
apply (*blast dest: zless-trans*)
done

lemma *zle-anti-sym*: $\llbracket x \leq y; y \leq x; x \in \text{int}; y \in \text{int} \rrbracket \implies x=y$
by (*drule zle-anti-sym-intify, auto*)

lemma *zle-trans-lemma*:
 $\llbracket x \in \text{int}; y \in \text{int}; z \in \text{int}; x \leq y; y \leq z \rrbracket \implies x \leq z$
apply (*simp add: zle-def, auto*)
apply (*blast intro: zless-trans*)
done

lemma *zle-trans* [*trans*]: $\llbracket x \leq y; y \leq z \rrbracket \implies x \leq z$
apply (*subgoal-tac intify (x) \leq intify (z)*)
apply (*rule-tac [2] y = intify (y) in zle-trans-lemma*)
apply *auto*
done

lemma *zle-zless-trans* [*trans*]: $\llbracket i \leq j; j < k \rrbracket \implies i < k$
apply (*auto simp add: zle-def*)
apply (*blast intro: zless-trans*)
apply (*simp add: zless-def zdiff-def zadd-def*)
done

lemma *zless-zle-trans* [*trans*]: $\llbracket i < j; j \leq k \rrbracket \implies i < k$
apply (*auto simp add: zle-def*)
apply (*blast intro: zless-trans*)
apply (*simp add: zless-def zdiff-def zminus-def*)
done

lemma *not-zless-iff-zle*: $\neg (z < w) \longleftrightarrow (w \leq z)$
apply (*cut-tac z = z and w = w in zless-linear*)

```

apply (auto dest: zless-trans simp add: zle-def)
apply (auto dest!: zless-imp-intify-neq)
done

```

```

lemma not-zle-iff-zless:  $\neg (z \leq w) \longleftrightarrow (w < z)$ 
by (simp add: not-zless-iff-zle [THEN iff-sym])

```

31.11 More subtraction laws (for *zcompare-rls*)

```

lemma zdiff-zdiff-eq:  $(x \$- y) \$- z = x \$- (y \$+ z)$ 
by (simp add: zdiff-def zadd-ac)

```

```

lemma zdiff-zdiff-eq2:  $x \$- (y \$- z) = (x \$+ z) \$- y$ 
by (simp add: zdiff-def zadd-ac)

```

```

lemma zdiff-zless-iff:  $(x \$- y < z) \longleftrightarrow (x < z \$+ y)$ 
by (simp add: zless-def zdiff-def zadd-ac)

```

```

lemma zless-zdiff-iff:  $(x < z \$- y) \longleftrightarrow (x \$+ y < z)$ 
by (simp add: zless-def zdiff-def zadd-ac)

```

```

lemma zdiff-eq-iff:  $\llbracket x \in \text{int}; z \in \text{int} \rrbracket \implies (x \$- y = z) \longleftrightarrow (x = z \$+ y)$ 
by (auto simp add: zdiff-def zadd-assoc)

```

```

lemma eq-zdiff-iff:  $\llbracket x \in \text{int}; z \in \text{int} \rrbracket \implies (x = z \$- y) \longleftrightarrow (x \$+ y = z)$ 
by (auto simp add: zdiff-def zadd-assoc)

```

```

lemma zdiff-zle-iff-lemma:
   $\llbracket x \in \text{int}; z \in \text{int} \rrbracket \implies (x \$- y \leq z) \longleftrightarrow (x \leq z \$+ y)$ 
by (auto simp add: zle-def zdiff-eq-iff zdiff-zless-iff)

```

```

lemma zdiff-zle-iff:  $(x \$- y \leq z) \longleftrightarrow (x \leq z \$+ y)$ 
by (cut-tac zdiff-zle-iff-lemma [OF intify-in-int intify-in-int], simp)

```

```

lemma zle-zdiff-iff-lemma:
   $\llbracket x \in \text{int}; z \in \text{int} \rrbracket \implies (x \leq z \$- y) \longleftrightarrow (x \$+ y \leq z)$ 
apply (auto simp add: zle-def zdiff-eq-iff zless-zdiff-iff)
apply (auto simp add: zdiff-def zadd-assoc)
done

```

```

lemma zle-zdiff-iff:  $(x \leq z \$- y) \longleftrightarrow (x \$+ y \leq z)$ 
by (cut-tac zle-zdiff-iff-lemma [OF intify-in-int intify-in-int], simp)

```

This list of rewrites simplifies (in)equalities by bringing subtractions to the top and then moving negative terms to the other side. Use with *zadd-ac*

```

lemmas zcompare-rls =
  zdiff-def [symmetric]
  zadd-zdiff-eq zdiff-zadd-eq zdiff-zdiff-eq zdiff-zdiff-eq2
  zdiff-zless-iff zless-zdiff-iff zdiff-zle-iff zle-zdiff-iff

```

zdiff-eq-iff eq-zdiff-iff

31.12 Monotonicity and Cancellation Results for Instantiation of the CancelNumerals Simprocs

lemma *zadd-left-cancel*:

$\llbracket w \in \text{int}; w': \text{int} \rrbracket \implies (z \$+ w' = z \$+ w) \longleftrightarrow (w' = w)$

apply *safe*

apply (*drule-tac* $t = \lambda x. x \$+ (\$-z)$ **in** *subst-context*)

apply (*simp add: zadd-ac*)

done

lemma *zadd-left-cancel-intify [simp]*:

$(z \$+ w' = z \$+ w) \longleftrightarrow \text{intify}(w') = \text{intify}(w)$

apply (*rule iff-trans*)

apply (*rule-tac* [2] *zadd-left-cancel, auto*)

done

lemma *zadd-right-cancel*:

$\llbracket w \in \text{int}; w': \text{int} \rrbracket \implies (w' \$+ z = w \$+ z) \longleftrightarrow (w' = w)$

apply *safe*

apply (*drule-tac* $t = \lambda x. x \$+ (\$-z)$ **in** *subst-context*)

apply (*simp add: zadd-ac*)

done

lemma *zadd-right-cancel-intify [simp]*:

$(w' \$+ z = w \$+ z) \longleftrightarrow \text{intify}(w') = \text{intify}(w)$

apply (*rule iff-trans*)

apply (*rule-tac* [2] *zadd-right-cancel, auto*)

done

lemma *zadd-right-cancel-zless [simp]*: $(w' \$+ z \$< w \$+ z) \longleftrightarrow (w' \$< w)$

by (*simp add: zdiff-zless-iff [THEN iff-sym] zdiff-def zadd-assoc*)

lemma *zadd-left-cancel-zless [simp]*: $(z \$+ w' \$< z \$+ w) \longleftrightarrow (w' \$< w)$

by (*simp add: zadd-commute [of z] zadd-right-cancel-zless*)

lemma *zadd-right-cancel-zle [simp]*: $(w' \$+ z \$\leq w \$+ z) \longleftrightarrow w' \$\leq w$

by (*simp add: zle-def*)

lemma *zadd-left-cancel-zle [simp]*: $(z \$+ w' \$\leq z \$+ w) \longleftrightarrow w' \$\leq w$

by (*simp add: zadd-commute [of z] zadd-right-cancel-zle*)

lemmas *zadd-zless-mono1 = zadd-right-cancel-zless [THEN iffD2]*

lemmas *zadd-zless-mono2 = zadd-left-cancel-zless [THEN iffD2]*

lemmas *zadd-zle-mono1* = *zadd-right-cancel-zle* [*THEN iffD2*]

lemmas *zadd-zle-mono2* = *zadd-left-cancel-zle* [*THEN iffD2*]

lemma *zadd-zle-mono*: $\llbracket w' \$\leq w; z' \$\leq z \rrbracket \implies w' \$+ z' \$\leq w \$+ z$
by (*erule zadd-zle-mono1* [*THEN zle-trans*], *simp*)

lemma *zadd-zless-mono*: $\llbracket w' \$< w; z' \$\leq z \rrbracket \implies w' \$+ z' \$< w \$+ z$
by (*erule zadd-zless-mono1* [*THEN zless-zle-trans*], *simp*)

31.13 Comparison laws

lemma *zminus-zless-zminus* [*simp*]: $(\$- x \$< \$- y) \longleftrightarrow (y \$< x)$
by (*simp add: zless-def zdiff-def zadd-ac*)

lemma *zminus-zle-zminus* [*simp*]: $(\$- x \$\leq \$- y) \longleftrightarrow (y \$\leq x)$
by (*simp add: not-zless-iff-zle* [*THEN iff-sym*])

31.13.1 More inequality lemmas

lemma *equation-zminus*: $\llbracket x \in \text{int}; y \in \text{int} \rrbracket \implies (x = \$- y) \longleftrightarrow (y = \$- x)$
by *auto*

lemma *zminus-equation*: $\llbracket x \in \text{int}; y \in \text{int} \rrbracket \implies (\$- x = y) \longleftrightarrow (\$- y = x)$
by *auto*

lemma *equation-zminus-intify*: $(\text{intify}(x) = \$- y) \longleftrightarrow (\text{intify}(y) = \$- x)$
apply (*cut-tac x = intify (x) and y = intify (y) in equation-zminus*)
apply *auto*
done

lemma *zminus-equation-intify*: $(\$- x = \text{intify}(y)) \longleftrightarrow (\$- y = \text{intify}(x))$
apply (*cut-tac x = intify (x) and y = intify (y) in zminus-equation*)
apply *auto*
done

31.13.2 The next several equations are permutative: watch out!

lemma *zless-zminus*: $(x \$< \$- y) \longleftrightarrow (y \$< \$- x)$
by (*simp add: zless-def zdiff-def zadd-ac*)

lemma *zminus-zless*: $(\$- x \$< y) \longleftrightarrow (\$- y \$< x)$
by (*simp add: zless-def zdiff-def zadd-ac*)

lemma *zle-zminus*: $(x \$\leq \$- y) \longleftrightarrow (y \$\leq \$- x)$
by (*simp add: not-zless-iff-zle* [*THEN iff-sym*] *zminus-zless*)

```

lemma zminus-zle: ( $\$- x \$\leq y$ )  $\longleftrightarrow$  ( $\$- y \$\leq x$ )
by (simp add: not-zless-iff-zle [THEN iff-sym] zless-zminus)

end

```

32 Arithmetic on Binary Integers

```

theory Bin
imports Int Datatype
begin

```

```

consts bin :: i
datatype
  bin = Pls
      | Min
      | Bit (w ∈ bin, b ∈ bool)    (infixl  $\langle BIT \rangle$  90)

```

```

consts
  integ-of :: i  $\Rightarrow$  i
  NCons     :: [i,i]  $\Rightarrow$  i
  bin-succ  :: i  $\Rightarrow$  i
  bin-pred  :: i  $\Rightarrow$  i
  bin-minus :: i  $\Rightarrow$  i
  bin-adder :: i  $\Rightarrow$  i
  bin-mult  :: [i,i]  $\Rightarrow$  i

```

```

primrec
  integ-of-Pls: integ-of (Pls)    =  $\$ \# 0$ 
  integ-of-Min: integ-of (Min)     =  $\$ - (\$ \# 1)$ 
  integ-of-BIT: integ-of (w BIT b) =  $\$ \# b \$ + \text{integ-of}(w) \$ + \text{integ-of}(w)$ 

```

```

primrec
  NCons-Pls: NCons (Pls,b)    = cond(b,Pls BIT b,Pls)
  NCons-Min: NCons (Min,b)    = cond(b,Min,Min BIT b)
  NCons-BIT: NCons (w BIT c,b) = w BIT c BIT b

```

```

primrec
  bin-succ-Pls: bin-succ (Pls)    = Pls BIT 1
  bin-succ-Min: bin-succ (Min)     = Pls
  bin-succ-BIT: bin-succ (w BIT b) = cond(b, bin-succ(w) BIT 0, NCons(w,1))

```

```

primrec
  bin-pred-Pls: bin-pred (Pls)    = Min
  bin-pred-Min: bin-pred (Min)     = Min BIT 0
  bin-pred-BIT: bin-pred (w BIT b) = cond(b, NCons(w,0), bin-pred(w) BIT 1)

```

```

primrec

```


bin-minus-Pls:
 $\text{bin-minus } (Pls) = Pls$
bin-minus-Min:
 $\text{bin-minus } (Min) = Pls \text{ BIT } 1$
bin-minus-BIT:
 $\text{bin-minus } (w \text{ BIT } b) = \text{cond}(b, \text{bin-pred}(NCons(\text{bin-minus}(w), 0)), \text{bin-minus}(w) \text{ BIT } 0)$

primrec

bin-adder-Pls:
 $\text{bin-adder } (Pls) = (\lambda w \in bin. w)$
bin-adder-Min:
 $\text{bin-adder } (Min) = (\lambda w \in bin. \text{bin-pred}(w))$
bin-adder-BIT:
 $\text{bin-adder } (v \text{ BIT } x) =$
 $(\lambda w \in bin.$
 $\quad \text{bin-case } (v \text{ BIT } x, \text{bin-pred}(v \text{ BIT } x),$
 $\quad \lambda w y. NCons(\text{bin-adder } (v) \text{ ' cond}(x \text{ and } y, \text{bin-succ}(w), w),$
 $\quad \quad x \text{ xor } y),$
 $\quad w))$

definition

$\text{bin-add} :: [i, i] \Rightarrow i$ **where**
 $\text{bin-add}(v, w) \equiv \text{bin-adder}(v) \text{ ' } w$

primrec

bin-mult-Pls:
 $\text{bin-mult } (Pls, w) = Pls$
bin-mult-Min:
 $\text{bin-mult } (Min, w) = \text{bin-minus}(w)$
bin-mult-BIT:
 $\text{bin-mult } (v \text{ BIT } b, w) = \text{cond}(b, \text{bin-add}(NCons(\text{bin-mult}(v, w), 0), w), NCons(\text{bin-mult}(v, w), 0))$

syntax

$\text{-Int0} :: i \text{ (}\langle \#() 0 \rangle\text{)}$
 $\text{-Int1} :: i \text{ (}\langle \#() 1 \rangle\text{)}$
 $\text{-Int2} :: i \text{ (}\langle \#() 2 \rangle\text{)}$
 $\text{-Neg-Int1} :: i \text{ (}\langle \#-() 1 \rangle\text{)}$
 $\text{-Neg-Int2} :: i \text{ (}\langle \#-() 2 \rangle\text{)}$

translations

$\#0 \Rightarrow \text{CONST integ-of}(\text{CONST } Pls)$
 $\#1 \Rightarrow \text{CONST integ-of}(\text{CONST } Pls \text{ BIT } 1)$
 $\#2 \Rightarrow \text{CONST integ-of}(\text{CONST } Pls \text{ BIT } 1 \text{ BIT } 0)$
 $\#-1 \Rightarrow \text{CONST integ-of}(\text{CONST } Min)$
 $\#-2 \Rightarrow \text{CONST integ-of}(\text{CONST } Min \text{ BIT } 0)$

syntax

-Int :: num-token $\Rightarrow i$ ($\langle \langle \text{open-block notation} = \langle \text{literal number} \rangle \# - \rangle \rangle$ 1000)
-Neg-Int :: num-token $\Rightarrow i$ ($\langle \langle \text{open-block notation} = \langle \text{literal number} \rangle \# - - \rangle \rangle$ 1000)

syntax-consts

-Int0 -Int1 -Int2 -Int -Neg-Int1 -Neg-Int2 -Neg-Int \equiv integ-of

ML-file $\langle \text{Tools/numeral-syntax.ML} \rangle$

declare bin.intros [simp, TC]

lemma NCons-Pls-0: $NCons(Pls, 0) = Pls$
by simp

lemma NCons-Pls-1: $NCons(Pls, 1) = Pls \text{ BIT } 1$
by simp

lemma NCons-Min-0: $NCons(Min, 0) = Min \text{ BIT } 0$
by simp

lemma NCons-Min-1: $NCons(Min, 1) = Min$
by simp

lemma NCons-BIT: $NCons(w \text{ BIT } x, b) = w \text{ BIT } x \text{ BIT } b$
by (simp add: bin.case-eqns)

lemmas NCons-simps [simp] =
NCons-Pls-0 NCons-Pls-1 NCons-Min-0 NCons-Min-1 NCons-BIT

lemma integ-of-type [TC]: $w \in \text{bin} \implies \text{integ-of}(w) \in \text{int}$
apply (induct-tac w)
apply (simp-all add: bool-into-nat)
done

lemma NCons-type [TC]: $\llbracket w \in \text{bin}; b \in \text{bool} \rrbracket \implies NCons(w, b) \in \text{bin}$
by (induct-tac w, auto)

lemma bin-succ-type [TC]: $w \in \text{bin} \implies \text{bin-succ}(w) \in \text{bin}$
by (induct-tac w, auto)

lemma bin-pred-type [TC]: $w \in \text{bin} \implies \text{bin-pred}(w) \in \text{bin}$
by (induct-tac w, auto)

lemma *bin-minus-type* [TC]: $w \in \text{bin} \implies \text{bin-minus}(w) \in \text{bin}$
by (*induct-tac* *w*, *auto*)

lemma *bin-add-type* [rule-format]:
 $v \in \text{bin} \implies \forall w \in \text{bin}. \text{bin-add}(v, w) \in \text{bin}$
unfolding *bin-add-def*
apply (*induct-tac* *v*)
apply (*rule-tac* [3] *ballI*)
apply (*rename-tac* [3] w')
apply (*induct-tac* [3] w')
apply (*simp-all* *add: NCons-type*)
done

declare *bin-add-type* [TC]

lemma *bin-mult-type* [TC]: $\llbracket v \in \text{bin}; w \in \text{bin} \rrbracket \implies \text{bin-mult}(v, w) \in \text{bin}$
by (*induct-tac* *v*, *auto*)

32.0.1 The Carry and Borrow Functions, *bin-succ* and *bin-pred*

lemma *integ-of-NCons* [simp]:
 $\llbracket w \in \text{bin}; b \in \text{bool} \rrbracket \implies \text{integ-of}(\text{NCons}(w, b)) = \text{integ-of}(w \text{ BIT } b)$
apply (*erule bin.cases*)
apply (*auto elim!: boolE*)
done

lemma *integ-of-succ* [simp]:
 $w \in \text{bin} \implies \text{integ-of}(\text{bin-succ}(w)) = \$\#1 \$+ \text{integ-of}(w)$
apply (*erule bin.induct*)
apply (*auto simp add: zadd-ac elim!: boolE*)
done

lemma *integ-of-pred* [simp]:
 $w \in \text{bin} \implies \text{integ-of}(\text{bin-pred}(w)) = \$- (\$ \# 1) \$+ \text{integ-of}(w)$
apply (*erule bin.induct*)
apply (*auto simp add: zadd-ac elim!: boolE*)
done

32.0.2 *bin-minus*: Unary Negation of Binary Integers

lemma *integ-of-minus*: $w \in \text{bin} \implies \text{integ-of}(\text{bin-minus}(w)) = \$- \text{integ-of}(w)$
apply (*erule bin.induct*)
apply (*auto simp add: zadd-ac zminus-zadd-distrib elim!: boolE*)
done

32.0.3 *bin-add*: Binary Addition

lemma *bin-add-Pls* [simp]: $w \in \text{bin} \implies \text{bin-add}(\text{Pls}, w) = w$
by (*unfold bin-add-def*, *simp*)

```

lemma bin-add-Pls-right:  $w \in \text{bin} \implies \text{bin-add}(w, \text{Pls}) = w$ 
  unfolding bin-add-def
apply (erule bin.induct, auto)
done

lemma bin-add-Min [simp]:  $w \in \text{bin} \implies \text{bin-add}(\text{Min}, w) = \text{bin-pred}(w)$ 
by (unfold bin-add-def, simp)

lemma bin-add-Min-right:  $w \in \text{bin} \implies \text{bin-add}(w, \text{Min}) = \text{bin-pred}(w)$ 
  unfolding bin-add-def
apply (erule bin.induct, auto)
done

lemma bin-add-BIT-Pls [simp]:  $\text{bin-add}(v \text{ BIT } x, \text{Pls}) = v \text{ BIT } x$ 
by (unfold bin-add-def, simp)

lemma bin-add-BIT-Min [simp]:  $\text{bin-add}(v \text{ BIT } x, \text{Min}) = \text{bin-pred}(v \text{ BIT } x)$ 
by (unfold bin-add-def, simp)

lemma bin-add-BIT-BIT [simp]:
   $\llbracket w \in \text{bin}; y \in \text{bool} \rrbracket$ 
   $\implies \text{bin-add}(v \text{ BIT } x, w \text{ BIT } y) =$ 
     $\text{NCons}(\text{bin-add}(v, \text{cond}(x \text{ and } y, \text{bin-succ}(w), w)), x \text{ xor } y)$ 
by (unfold bin-add-def, simp)

lemma integ-of-add [rule-format]:
   $v \in \text{bin} \implies$ 
   $\forall w \in \text{bin}. \text{integ-of}(\text{bin-add}(v, w)) = \text{integ-of}(v) \$+ \text{integ-of}(w)$ 
apply (erule bin.induct, simp, simp)
apply (rule ballI)
apply (induct-tac wa)
apply (auto simp add: zadd-ac elim!: boolE)
done

lemma diff-integ-of-eq:
   $\llbracket v \in \text{bin}; w \in \text{bin} \rrbracket$ 
   $\implies \text{integ-of}(v) \$- \text{integ-of}(w) = \text{integ-of}(\text{bin-add}(v, \text{bin-minus}(w)))$ 
  unfolding zdiff-def
apply (simp add: integ-of-add integ-of-minus)
done

```

32.0.4 *bin-mult*: Binary Multiplication

```

lemma integ-of-mult:
   $\llbracket v \in \text{bin}; w \in \text{bin} \rrbracket$ 
   $\implies \text{integ-of}(\text{bin-mult}(v, w)) = \text{integ-of}(v) \$* \text{integ-of}(w)$ 
apply (induct-tac v, simp)

```

apply (*simp add: integ-of-minus*)
apply (*auto simp add: zadd-ac integ-of-add zadd-zmult-distrib elim!: boolE*)
done

32.1 Computations

lemma *bin-succ-1*: $\text{bin-succ}(w \text{ BIT } 1) = \text{bin-succ}(w) \text{ BIT } 0$
by *simp*

lemma *bin-succ-0*: $\text{bin-succ}(w \text{ BIT } 0) = \text{NCons}(w, 1)$
by *simp*

lemma *bin-pred-1*: $\text{bin-pred}(w \text{ BIT } 1) = \text{NCons}(w, 0)$
by *simp*

lemma *bin-pred-0*: $\text{bin-pred}(w \text{ BIT } 0) = \text{bin-pred}(w) \text{ BIT } 1$
by *simp*

lemma *bin-minus-1*: $\text{bin-minus}(w \text{ BIT } 1) = \text{bin-pred}(\text{NCons}(\text{bin-minus}(w), 0))$
by *simp*

lemma *bin-minus-0*: $\text{bin-minus}(w \text{ BIT } 0) = \text{bin-minus}(w) \text{ BIT } 0$
by *simp*

lemma *bin-add-BIT-11*: $w \in \text{bin} \implies \text{bin-add}(v \text{ BIT } 1, w \text{ BIT } 1) =$
 $\text{NCons}(\text{bin-add}(v, \text{bin-succ}(w)), 0)$
by *simp*

lemma *bin-add-BIT-10*: $w \in \text{bin} \implies \text{bin-add}(v \text{ BIT } 1, w \text{ BIT } 0) =$
 $\text{NCons}(\text{bin-add}(v, w), 1)$
by *simp*

lemma *bin-add-BIT-0*: $\llbracket w \in \text{bin}; y \in \text{bool} \rrbracket$
 $\implies \text{bin-add}(v \text{ BIT } 0, w \text{ BIT } y) = \text{NCons}(\text{bin-add}(v, w), y)$
by *simp*

lemma *bin-mult-1*: $\text{bin-mult}(v \text{ BIT } 1, w) = \text{bin-add}(\text{NCons}(\text{bin-mult}(v, w), 0), w)$
by *simp*

lemma *bin-mult-0*: $\text{bin-mult}(v \text{ BIT } 0, w) = \text{NCons}(\text{bin-mult}(v, w), 0)$
by *simp*

```

lemma int-of-0:  $\$ \# 0 = \# 0$ 
by simp

lemma int-of-succ:  $\$ \# \text{succ}(n) = \# 1 \$ + \$ \# n$ 
by (simp add: int-of-add [symmetric] natify-succ)

lemma zminus-0 [simp]:  $\$ - \# 0 = \# 0$ 
by simp

lemma zadd-0-intify [simp]:  $\# 0 \$ + z = \text{intify}(z)$ 
by simp

lemma zadd-0-right-intify [simp]:  $z \$ + \# 0 = \text{intify}(z)$ 
by simp

lemma zmult-1-intify [simp]:  $\# 1 \$ * z = \text{intify}(z)$ 
by simp

lemma zmult-1-right-intify [simp]:  $z \$ * \# 1 = \text{intify}(z)$ 
by (subst zmult-commute, simp)

lemma zmult-0 [simp]:  $\# 0 \$ * z = \# 0$ 
by simp

lemma zmult-0-right [simp]:  $z \$ * \# 0 = \# 0$ 
by (subst zmult-commute, simp)

lemma zmult-minus1 [simp]:  $\# -1 \$ * z = \$ - z$ 
by (simp add: zcompare-rls)

lemma zmult-minus1-right [simp]:  $z \$ * \# -1 = \$ - z$ 
apply (subst zmult-commute)
apply (rule zmult-minus1)
done

```

32.2 Simplification Rules for Comparison of Binary Numbers

Thanks to Norbert Voelker

```

lemma eq-integ-of-eq:
   $\llbracket v \in \text{bin}; w \in \text{bin} \rrbracket$ 
   $\implies ((\text{integ-of}(v)) = \text{integ-of}(w)) \longleftrightarrow$ 
   $\text{iszero}(\text{integ-of}(\text{bin-add}(v, \text{bin-minus}(w))))$ 
  unfolding iszero-def
apply (simp add: zcompare-rls integ-of-add integ-of-minus)
done

```

lemma *iszero-integ-of-Pls*: $iszero\ (integ-of(Pls))$
by (*unfold iszero-def, simp*)

lemma *nonzero-integ-of-Min*: $\neg iszero\ (integ-of(Min))$
unfolding *iszero-def*
apply (*simp add: zminus-equation*)
done

lemma *iszero-integ-of-BIT*:
 $\llbracket w \in bin; x \in bool \rrbracket$
 $\implies iszero\ (integ-of\ (w\ BIT\ x)) \longleftrightarrow (x=0 \wedge iszero\ (integ-of(w)))$
apply (*unfold iszero-def, simp*)
apply (*subgoal-tac integ-of (w) $\in int$*)
apply *typecheck*
apply (*drule int-cases*)
apply (*safe elim!: boolE*)
apply (*simp-all (asm-lr) add: zcompare-rls zminus-zadd-distrib [symmetric]*
 $int-of-add\ [symmetric]$)
done

lemma *iszero-integ-of-0*:
 $w \in bin \implies iszero\ (integ-of\ (w\ BIT\ 0)) \longleftrightarrow iszero\ (integ-of(w))$
by (*simp only: iszero-integ-of-BIT, blast*)

lemma *iszero-integ-of-1*: $w \in bin \implies \neg iszero\ (integ-of\ (w\ BIT\ 1))$
by (*simp only: iszero-integ-of-BIT, blast*)

lemma *less-integ-of-eq-neg*:
 $\llbracket v \in bin; w \in bin \rrbracket$
 $\implies integ-of(v) \$< integ-of(w)$
 $\longleftrightarrow znegative\ (integ-of\ (bin-add\ (v, bin-minus(w))))$
unfolding *zless-def zdiff-def*
apply (*simp add: integ-of-minus integ-of-add*)
done

lemma *not-neg-integ-of-Pls*: $\neg znegative\ (integ-of(Pls))$
by *simp*

lemma *neg-integ-of-Min*: $znegative\ (integ-of(Min))$
by *simp*

lemma *neg-integ-of-BIT*:
 $\llbracket w \in bin; x \in bool \rrbracket$
 $\implies znegative\ (integ-of\ (w\ BIT\ x)) \longleftrightarrow znegative\ (integ-of(w))$

```

apply simp
apply (subgoal-tac integ-of ( $w \in \text{int}$ )
apply typecheck
apply (drule int-cases)
apply (auto elim!: boolE simp add: integ-of-add [symmetric] zcompare-rls)
apply (simp-all add: zminus-zadd-distrib [symmetric] zdiff-def
      integ-of-add [symmetric])
apply (subgoal-tac  $\$ \# 1 \ \$ - \ \$ \# \text{ succ } (\text{succ } (n \ \# + \ n)) = \$ - \ \$ \# \text{ succ } (n \ \# + \ n) \ )$ 
  apply (simp add: zdiff-def)
apply (simp add: equation-zminus integ-of-diff [symmetric])
done

```

```

lemma le-integ-of-eq-not-less:
  (integ-of( $x$ )  $\$ \leq$  (integ-of( $w$ )))  $\longleftrightarrow \neg$  (integ-of( $w$ )  $\$ <$  (integ-of( $x$ )))
by (simp add: not-zless-iff-zle [THEN iff-sym])

```

```

declare bin-succ-BIT [simp del]
         bin-pred-BIT [simp del]
         bin-minus-BIT [simp del]
         NCons-Pls [simp del]
         NCons-Min [simp del]
         bin-adder-BIT [simp del]
         bin-mult-BIT [simp del]

```

```

declare integ-of-Pls [simp del] integ-of-Min [simp del] integ-of-BIT [simp del]

```

```

lemmas bin-arith-extra-simps =
  integ-of-add [symmetric]
  integ-of-minus [symmetric]
  integ-of-mult [symmetric]
  bin-succ-1 bin-succ-0
  bin-pred-1 bin-pred-0
  bin-minus-1 bin-minus-0
  bin-add-Pls-right bin-add-Min-right
  bin-add-BIT-0 bin-add-BIT-10 bin-add-BIT-11
  diff-integ-of-eq
  bin-mult-1 bin-mult-0 NCons-simps

```

```

lemmas bin-arith-simps =
  bin-pred-Pls bin-pred-Min
  bin-succ-Pls bin-succ-Min

```


bin-add-Pls bin-add-Min
bin-minus-Pls bin-minus-Min
bin-mult-Pls bin-mult-Min
bin-arith-extra-simps

lemmas *bin-rel-simps* =
eq-integ-of-eq iszero-integ-of-Pls nonzero-integ-of-Min
iszero-integ-of-0 iszero-integ-of-1
less-integ-of-eq-neg
not-neg-integ-of-Pls neg-integ-of-Min neg-integ-of-BIT
le-integ-of-eq-not-less

declare *bin-arith-simps* [*simp*]
declare *bin-rel-simps* [*simp*]

lemma *add-integ-of-left* [*simp*]:

$$\llbracket v \in \text{bin}; w \in \text{bin} \rrbracket$$

$$\implies \text{integ-of}(v) \$+ (\text{integ-of}(w) \$+ z) = (\text{integ-of}(\text{bin-add}(v,w)) \$+ z)$$
by (*simp add: zadd-assoc [symmetric]*)

lemma *mult-integ-of-left* [*simp*]:

$$\llbracket v \in \text{bin}; w \in \text{bin} \rrbracket$$

$$\implies \text{integ-of}(v) \$* (\text{integ-of}(w) \$* z) = (\text{integ-of}(\text{bin-mult}(v,w)) \$* z)$$
by (*simp add: zmult-assoc [symmetric]*)

lemma *add-integ-of-diff1* [*simp*]:

$$\llbracket v \in \text{bin}; w \in \text{bin} \rrbracket$$

$$\implies \text{integ-of}(v) \$+ (\text{integ-of}(w) \$- c) = \text{integ-of}(\text{bin-add}(v,w)) \$- (c)$$
unfolding *zdiff-def*
apply (*rule add-integ-of-left, auto*)
done

lemma *add-integ-of-diff2* [*simp*]:

$$\llbracket v \in \text{bin}; w \in \text{bin} \rrbracket$$

$$\implies \text{integ-of}(v) \$+ (c \$- \text{integ-of}(w)) =$$

$$\text{integ-of}(\text{bin-add}(v, \text{bin-minus}(w))) \$+ (c)$$
apply (*subst diff-integ-of-eq [symmetric]*)
apply (*simp-all add: zdiff-def zadd-ac*)
done

declare *int-of-0* [*simp*] *int-of-succ* [*simp*]

lemma *zdiff0* [*simp*]: $\#0 \ \$- \ x = \$-x$
by (*simp add: zdiff-def*)

lemma *zdiff0-right* [*simp*]: $x \ \$- \ \#0 = \text{intify}(x)$
by (*simp add: zdiff-def*)

lemma *zdiff-self* [*simp*]: $x \ \$- \ x = \#0$
by (*simp add: zdiff-def*)

lemma *znegative-iff-zless-0*: $k \in \text{int} \implies \text{znegative}(k) \longleftrightarrow k \ \$< \ \#0$
by (*simp add: zless-def*)

lemma *zero-zless-imp-znegative-zminus*: $\llbracket \#0 \ \$< \ k; k \in \text{int} \rrbracket \implies \text{znegative}(\$-k)$
by (*simp add: zless-def*)

lemma *zero-zle-int-of* [*simp*]: $\#0 \ \$\leq \ \$\# \ n$
by (*simp add: not-zless-iff-zle [THEN iff-sym] znegative-iff-zless-0 [THEN iff-sym]*)

lemma *nat-of-0* [*simp*]: $\text{nat-of}(\#0) = 0$
by (*simp only: natify-0 int-of-0 [symmetric] nat-of-int-of*)

lemma *nat-le-int0-lemma*: $\llbracket z \ \$\leq \ \$\#0; z \in \text{int} \rrbracket \implies \text{nat-of}(z) = 0$
by (*auto simp add: znegative-iff-zless-0 [THEN iff-sym] zle-def zneg-nat-of*)

lemma *nat-le-int0*: $z \ \$\leq \ \$\#0 \implies \text{nat-of}(z) = 0$
apply (*subgoal-tac nat-of (intify (z)) = 0*)
apply (*rule-tac [2] nat-le-int0-lemma, auto*)
done

lemma *int-of-eq-0-imp-natify-eq-0*: $\$ \# \ n = \#0 \implies \text{natify}(n) = 0$
by (*rule not-znegative-imp-zero, auto*)

lemma *nat-of-zminus-int-of*: $\text{nat-of}(\$- \$\# \ n) = 0$
by (*simp add: nat-of-def int-of-def raw-nat-of zminus image-intrel-int*)

lemma *int-of-nat-of*: $\#0 \ \$\leq \ z \implies \$\# \ \text{nat-of}(z) = \text{intify}(z)$
apply (*rule not-zneg-nat-of-intify*)
apply (*simp add: znegative-iff-zless-0 not-zless-iff-zle*)
done

declare *int-of-nat-of* [*simp*] *nat-of-zminus-int-of* [*simp*]

lemma *int-of-nat-of-if*: $\$ \# \ \text{nat-of}(z) = (\text{if } \#0 \ \$\leq \ z \text{ then } \text{intify}(z) \text{ else } \#0)$
by (*simp add: int-of-nat-of znegative-iff-zless-0 not-zle-iff-zless*)

lemma *zless-nat-iff-int-zless*: $\llbracket m \in \text{nat}; z \in \text{int} \rrbracket \implies (m < \text{nat-of}(z)) \longleftrightarrow (\$ \# m \ \$< \ z)$
apply (*case-tac znegative (z)*)
apply (*erule-tac [2] not-zneg-nat-of [THEN subst]*)

```

apply (auto dest: zless-trans dest!: zero-zle-int-of [THEN zle-zless-trans]
        simp add: znegative-iff-zless-0)
done

```

```

lemma zless-nat-conj-lemma:  $\$ \# 0 \ \$ < z \implies (\text{nat-of}(w) < \text{nat-of}(z)) \longleftrightarrow (w \ \$ < z)$ 
apply (rule iff-trans)
apply (rule zless-int-of [THEN iff-sym])
apply (auto simp add: int-of-nat-of-if simp del: zless-int-of)
apply (auto elim: zless-asym simp add: not-zle-iff-zless)
apply (blast intro: zless-zle-trans)
done

```

```

lemma zless-nat-conj:  $(\text{nat-of}(w) < \text{nat-of}(z)) \longleftrightarrow (\$ \# 0 \ \$ < z \wedge w \ \$ < z)$ 
apply (case-tac  $\$ \# 0 \ \$ < z$ )
apply (auto simp add: zless-nat-conj-lemma nat-le-int0 not-zless-iff-zle)
done

```

```

lemma integ-of-minus-reorient [simp]:
   $(\text{integ-of}(w) = \$ - x) \longleftrightarrow (\$ - x = \text{integ-of}(w))$ 
by auto

```

```

lemma integ-of-add-reorient [simp]:
   $(\text{integ-of}(w) = x \ \$ + y) \longleftrightarrow (x \ \$ + y = \text{integ-of}(w))$ 
by auto

```

```

lemma integ-of-diff-reorient [simp]:
   $(\text{integ-of}(w) = x \ \$ - y) \longleftrightarrow (x \ \$ - y = \text{integ-of}(w))$ 
by auto

```

```

lemma integ-of-mult-reorient [simp]:
   $(\text{integ-of}(w) = x \ \$ * y) \longleftrightarrow (x \ \$ * y = \text{integ-of}(w))$ 
by auto

```

```

lemmas [simp] =
  zminus-equation [where y = integ-of(w)]
  equation-zminus [where x = integ-of(w)]
  for w

```

```

lemmas [iff] =
  zminus-zless [where y = integ-of(w)]

```

```

    zless-zminus [where x = integ-of(w)]
  for w

lemmas [iff] =
  zminus-zle [where y = integ-of(w)]
  zle-zminus [where x = integ-of(w)]
  for w

lemmas [simp] =
  Let-def [where s = integ-of(w)] for w

lemma zless-iff-zdiff-zless-0: (x $< y)  $\longleftrightarrow$  (x$-y $< #0)
  by (simp add: zcompare-rls)

lemma eq-iff-zdiff-eq-0:  $\llbracket x \in \text{int}; y \in \text{int} \rrbracket \implies (x = y) \longleftrightarrow (x\$-y = \#0)$ 
  by (simp add: zcompare-rls)

lemma zle-iff-zdiff-zle-0: (x $\le$ y)  $\longleftrightarrow$  (x$-y $\le$ #0)
  by (simp add: zcompare-rls)

lemma left-zadd-zmult-distrib: i$*u $+ (j$*u $+ k) = (i$+j)$*u $+ k
  by (simp add: zadd-zmult-distrib zadd-ac)

lemma eq-add-iff1: (i$*u $+ m = j$*u $+ n)  $\longleftrightarrow$  ((i$-j)$*u $+ m = intify(n))
  apply (simp add: zdiff-def zadd-zmult-distrib)
  apply (simp add: zcompare-rls)
  apply (simp add: zadd-ac)
  done

lemma eq-add-iff2: (i$*u $+ m = j$*u $+ n)  $\longleftrightarrow$  (intify(m) = (j$-i)$*u $+ n)
  apply (simp add: zdiff-def zadd-zmult-distrib)
  apply (simp add: zcompare-rls)
  apply (simp add: zadd-ac)
  done

context fixes n :: i
begin

```

```

lemmas rel-iff-rel-0-rls =
  zless-iff-zdiff-zless-0 [where  $y = u \ \$+ \ v$ ]
  eq-iff-zdiff-eq-0 [where  $y = u \ \$+ \ v$ ]
  zle-iff-zdiff-zle-0 [where  $y = u \ \$+ \ v$ ]
  zless-iff-zdiff-zless-0 [where  $y = n$ ]
  eq-iff-zdiff-eq-0 [where  $y = n$ ]
  zle-iff-zdiff-zle-0 [where  $y = n$ ]
for  $u \ v$ 

lemma less-add-iff1:  $(i \ \$* \ u \ \$+ \ m \ \$< \ j \ \$* \ u \ \$+ \ n) \longleftrightarrow ((i \ \$- \ j) \ \$* \ u \ \$+ \ m \ \$< \ n)$ 
apply (simp add: zdiff-def zadd-zmult-distrib zadd-ac rel-iff-rel-0-rls)
done

lemma less-add-iff2:  $(i \ \$* \ u \ \$+ \ m \ \$< \ j \ \$* \ u \ \$+ \ n) \longleftrightarrow (m \ \$< \ (j \ \$- \ i) \ \$* \ u \ \$+ \ n)$ 
apply (simp add: zdiff-def zadd-zmult-distrib zadd-ac rel-iff-rel-0-rls)
done

end

lemma le-add-iff1:  $(i \ \$* \ u \ \$+ \ m \ \$\leq \ j \ \$* \ u \ \$+ \ n) \longleftrightarrow ((i \ \$- \ j) \ \$* \ u \ \$+ \ m \ \$\leq \ n)$ 
apply (simp add: zdiff-def zadd-zmult-distrib)
apply (simp add: zcompare-rls)
apply (simp add: zadd-ac)
done

lemma le-add-iff2:  $(i \ \$* \ u \ \$+ \ m \ \$\leq \ j \ \$* \ u \ \$+ \ n) \longleftrightarrow (m \ \$\leq \ (j \ \$- \ i) \ \$* \ u \ \$+ \ n)$ 
apply (simp add: zdiff-def zadd-zmult-distrib)
apply (simp add: zcompare-rls)
apply (simp add: zadd-ac)
done

ML-file  $\langle \text{int-arith.ML} \rangle$ 

simproc-setup inteq-cancel-numerals
 $(l \ \$+ \ m \ \$= \ n \mid l \ \$= \ m \ \$+ \ n \mid l \ \$- \ m \ \$= \ n \mid l \ \$= \ m \ \$- \ n \mid l \ \$* \ m \ \$= \ n \mid l \ \$= \ m \ \$* \ n) =$ 
 $\langle K \text{ Int-Numeral-Simprocs.inteq-cancel-numerals-proc} \rangle$ 

simproc-setup intless-cancel-numerals
 $(l \ \$+ \ m \ \$< \ n \mid l \ \$< \ m \ \$+ \ n \mid l \ \$- \ m \ \$< \ n \mid l \ \$< \ m \ \$- \ n \mid l \ \$* \ m \ \$< \ n \mid l \ \$< \ m \ \$* \ n) =$ 
 $\langle K \text{ Int-Numeral-Simprocs.intless-cancel-numerals-proc} \rangle$ 

simproc-setup intle-cancel-numerals
 $(l \ \$+ \ m \ \$\leq \ n \mid l \ \$\leq \ m \ \$+ \ n \mid l \ \$- \ m \ \$\leq \ n \mid l \ \$\leq \ m \ \$- \ n \mid l \ \$* \ m \ \$\leq \ n \mid l \ \$\leq \ m \ \$* \ n) =$ 
 $\langle K \text{ Int-Numeral-Simprocs.intle-cancel-numerals-proc} \rangle$ 

simproc-setup int-combine-numerals  $(i \ \$+ \ j \mid i \ \$- \ j) =$ 

```

$\langle K \text{ Int-Numeral-Simprocs.int-combine-numerals-proc} \rangle$

simproc-setup *int-combine-numerals-prod* ($i \ \$* \ j$) =
 $\langle K \text{ Int-Numeral-Simprocs.int-combine-numerals-prod-proc} \rangle$

32.2.1 Examples

combine-numerals-prod (products of separate literals)

lemma $\#5 \ \$* \ x \ \$* \ \#3 = y$ **apply simp oops**

schematic-goal $y2 \ \$+ \ ?x42 = y \ \$+ \ y2$ **apply simp oops**

lemma $oo : int \Longrightarrow l \ \$+ \ (l \ \$+ \ \#2) \ \$+ \ oo = oo$ **apply simp oops**

lemma $\#9 \$* x \ \$+ \ y = x \$* \#23 \ \$+ \ z$ **apply simp oops**

lemma $y \ \$+ \ x = x \ \$+ \ z$ **apply simp oops**

lemma $x : int \Longrightarrow x \ \$+ \ y \ \$+ \ z = x \ \$+ \ z$ **apply simp oops**

lemma $x : int \Longrightarrow y \ \$+ \ (z \ \$+ \ x) = z \ \$+ \ x$ **apply simp oops**

lemma $z : int \Longrightarrow x \ \$+ \ y \ \$+ \ z = (z \ \$+ \ y) \ \$+ \ (x \ \$+ \ w)$ **apply simp oops**

lemma $z : int \Longrightarrow x \$* y \ \$+ \ z = (z \ \$+ \ y) \ \$+ \ (y \$* x \ \$+ \ w)$ **apply simp oops**

lemma $\#-3 \ \$* \ x \ \$+ \ y \ \$\leq x \ \$* \ \#2 \ \$+ \ z$ **apply simp oops**

lemma $y \ \$+ \ x \ \$\leq x \ \$+ \ z$ **apply simp oops**

lemma $x \ \$+ \ y \ \$+ \ z \ \$\leq x \ \$+ \ z$ **apply simp oops**

lemma $y \ \$+ \ (z \ \$+ \ x) \ \$< z \ \$+ \ x$ **apply simp oops**

lemma $x \ \$+ \ y \ \$+ \ z \ \$< (z \ \$+ \ y) \ \$+ \ (x \ \$+ \ w)$ **apply simp oops**

lemma $x \$* y \ \$+ \ z \ \$< (z \ \$+ \ y) \ \$+ \ (y \$* x \ \$+ \ w)$ **apply simp oops**

lemma $l \ \$+ \ \#2 \ \$+ \ \#2 \ \$+ \ \#2 \ \$+ \ (l \ \$+ \ \#2) \ \$+ \ (oo \ \$+ \ \#2) = uu$ **apply simp oops**

lemma $u : int \Longrightarrow \#2 \ \$* \ u = u$ **apply simp oops**

lemma $(i \ \$+ \ j \ \$+ \ \#12 \ \$+ \ k) \ \$- \ \#15 = y$ **apply simp oops**

lemma $(i \ \$+ \ j \ \$+ \ \#12 \ \$+ \ k) \ \$- \ \#5 = y$ **apply simp oops**

lemma $y \ \$- \ b \ \$< b$ **apply simp oops**

lemma $y \ \$- \ (\#3 \ \$* \ b \ \$+ \ c) \ \$< b \ \$- \ \#2 \ \$* \ c$ **apply simp oops**

lemma $(\#2 \ \$* \ x \ \$- \ (u \ \$* \ v) \ \$+ \ y) \ \$- \ v \ \$* \ \#3 \ \$* \ u = w$ **apply simp oops**

lemma $(\#2 \ \$* \ x \ \$* \ u \ \$* \ v \ \$+ \ (u \ \$* \ v) \ \$* \ \#4 \ \$+ \ y) \ \$- \ v \ \$* \ u \ \$* \ \#4 = w$ **apply simp oops**

lemma $(\#2 \ \$* \ x \ \$* \ u \ \$* \ v \ \$+ \ (u \ \$* \ v) \ \$* \ \#4 \ \$+ \ y) \ \$- \ v \ \$* \ u = w$ **apply simp oops**

lemma $u \ \$* \ v \ \$- \ (x \ \$* \ u \ \$* \ v \ \$+ \ (u \ \$* \ v) \ \$* \ \#4 \ \$+ \ y) = w$ **apply simp oops**

lemma $(i \ \$+ \ j \ \$+ \ \#12 \ \$+ \ k) = u \ \$+ \ \#15 \ \$+ \ y$ **apply simp oops**

lemma $(i \ \$+ \ j \ \$* \ \#2 \ \$+ \ \#12 \ \$+ \ k) = j \ \$+ \ \#5 \ \$+ \ y$ **apply simp oops**

lemma $\#2 \ \$* \ y \ \$+ \ \#3 \ \$* \ z \ \$+ \ \#6 \ \$* \ w \ \$+ \ \#2 \ \$* \ y \ \$+ \ \#3 \ \$* \ z \ \$+ \ \#2 \ \$* \ u = \#2 \ \$* \ y' \ \$+ \ \#3 \ \$* \ z' \ \$+ \ \#6 \ \$* \ w' \ \$+ \ \#2 \ \$* \ y' \ \$+ \ \#3 \ \$* \ z' \ \$+ \ u \ \$+ \ vv$
apply simp oops

lemma $a \ \$+ \ \$-(b\$+c) \ \$+ \ b = d$ **apply simp oops**

lemma $a \ \$+ \ \$-(b\$+c) \ \$- \ b = d$ **apply simp oops**

negative numerals

lemma $(i \ \$+ \ j \ \$+ \ \#-2 \ \$+ \ k) \ \$- \ (u \ \$+ \ \#5 \ \$+ \ y) = zz$ **apply simp oops**

lemma $(i \ \$+ \ j \ \$+ \ \#-3 \ \$+ \ k) \ \$< \ u \ \$+ \ \#5 \ \$+ \ y$ **apply simp oops**

lemma $(i \ \$+ \ j \ \$+ \ \#3 \ \$+ \ k) \ \$< \ u \ \$+ \ \#-6 \ \$+ \ y$ **apply simp oops**

lemma $(i \ \$+ \ j \ \$+ \ \#-12 \ \$+ \ k) \ \$- \ \#15 = y$ **apply simp oops**

lemma $(i \ \$+ \ j \ \$+ \ \#12 \ \$+ \ k) \ \$- \ \#-15 = y$ **apply simp oops**

lemma $(i \ \$+ \ j \ \$+ \ \#-12 \ \$+ \ k) \ \$- \ \#-15 = y$ **apply simp oops**

Multiplying separated numerals

lemma $\#6 \ \$* \ (\$ \ x \ \$* \ \#2) = uu$ **apply simp oops**

lemma $\#4 \ \$* \ (\$ \ x \ \$* \ \$ \ x) \ \$* \ (\#2 \ \$* \ \$ \ x) = uu$ **apply simp oops**

end

33 The Division Operators Div and Mod

theory *IntDiv*

imports *Bin OrderArith*

begin

definition

$quorem :: [i,i] \Rightarrow o$ **where**

$quorem \equiv \lambda \langle a,b \rangle \langle q,r \rangle.$

$a = b\$*q \ \$+ \ r \wedge$

$(\#0\$<b \wedge \#0\$ \leq r \wedge r\$<b \mid \neg(\#0\$<b) \wedge b\$<r \wedge r \ \$ \leq \#0)$

definition

$adjust :: [i,i] \Rightarrow i$ **where**

$adjust(b) \equiv \lambda \langle q,r \rangle. \text{ if } \#0 \ \$ \leq r\$-b \text{ then } <\#2\$*q \ \$+ \ \#1, r\$-b>$

$\text{ else } <\#2\$*q, r>$

definition

$posDivAlg :: i \Rightarrow i$ **where**

$posDivAlg(ab) \equiv$

$wfrec(measure(int*int, \lambda \langle a,b \rangle. \text{ nat-of } (a \ \$- \ b \ \$+ \ \#1)),$

$ab,$

$\lambda\langle a,b\rangle f. \text{ if } (a\$<b \mid b\$ \leq \#0) \text{ then } <\#0,a>$
 $\text{ else adjust}(b, f \text{ ' } <a,\#2\$*b>))$

definition

$negDivAlg :: i \Rightarrow i$ **where**

$negDivAlg(ab) \equiv$
 $wfrec(measure(int*int, \lambda\langle a,b\rangle. nat-of (\$- a \$- b)),$
 $ab,$
 $\lambda\langle a,b\rangle f. \text{ if } (\#0 \$ \leq a\$+b \mid b\$ \leq \#0) \text{ then } <\#-1,a\$+b>$
 $\text{ else adjust}(b, f \text{ ' } <a,\#2\$*b>))$

definition

$negateSnd :: i \Rightarrow i$ **where**

$negateSnd \equiv \lambda\langle q,r\rangle. <q, \$-r>$

definition

$divAlg :: i \Rightarrow i$ **where**

$divAlg \equiv$
 $\lambda\langle a,b\rangle. \text{ if } \#0 \$ \leq a \text{ then}$
 $\text{ if } \#0 \$ \leq b \text{ then } posDivAlg (\langle a,b\rangle)$
 $\text{ else if } a=\#0 \text{ then } <\#0,\#0>$
 $\text{ else } negateSnd (negDivAlg (<\$-a,\$-b>))$
 else
 $\text{ if } \#0\$<b \text{ then } negDivAlg (\langle a,b\rangle)$
 $\text{ else } negateSnd (posDivAlg (<\$-a,\$-b>))$

definition

$zdiv :: [i,i] \Rightarrow i$ **(infixl <zdiv> 70) where**
 $a \text{ zdiv } b \equiv fst (divAlg (<intify(a), intify(b)>))$

definition

$zmod :: [i,i] \Rightarrow i$ **(infixl <zmod> 70) where**
 $a \text{ zmod } b \equiv snd (divAlg (<intify(a), intify(b)>))$

lemma $zpos\text{-}add\text{-}zpos\text{-}imp\text{-}zpos: \llbracket \#0 \$< x; \#0 \$< y \rrbracket \implies \#0 \$< x \$+ y$
apply $(rule\text{-}tac \ y = y \text{ in } zless\text{-}trans)$
apply $(rule\text{-}tac \ [2] \ zdiff\text{-}zless\text{-}iff \ [THEN \ iffD1])$
apply $auto$
done


```

lemma zpos-add-zpos-imp-zpos:  $\llbracket \#0 \leq x; \#0 \leq y \rrbracket \implies \#0 \leq x + y$ 
apply (rule-tac  $y = y$  in zle-trans)
apply (rule-tac [2] zdiff-zle-iff [THEN iffD1])
apply auto
done

```

```

lemma zneg-add-zneg-imp-zneg:  $\llbracket x < \#0; y < \#0 \rrbracket \implies x + y < \#0$ 
apply (rule-tac  $y = y$  in zless-trans)
apply (rule zless-zdiff-iff [THEN iffD1])
apply auto
done

```

```

lemma zneg-or-0-add-zneg-or-0-imp-zneg-or-0:
   $\llbracket x \leq \#0; y \leq \#0 \rrbracket \implies x + y \leq \#0$ 
apply (rule-tac  $y = y$  in zle-trans)
apply (rule zle-zdiff-iff [THEN iffD1])
apply auto
done

```

```

lemma zero-lt-zmagnitude:  $\llbracket \#0 < k; k \in \text{int} \rrbracket \implies 0 < \text{zmagnitude}(k)$ 
apply (drule zero-zless-imp-negative-zminus)
apply (drule-tac [2] zneg-int-of)
apply (auto simp add: zminus-equation [of k])
apply (subgoal-tac  $0 < \text{zmagnitude} (\# \text{succ } (n))$ )
  apply simp
apply (simp only: zmagnitude-int-of)
apply simp
done

```

```

lemma zless-add-succ-iff:
   $(w < z + \# \text{succ}(m)) \longleftrightarrow (w < z + \#m \mid \text{intify}(w) = z + \#m)$ 
apply (auto simp add: zless-iff-succ-zadd zadd-assoc int-of-add [symmetric])
apply (rule-tac [3]  $x = 0$  in bexI)
apply (cut-tac  $m = m$  in int-succ-int-1)
apply (cut-tac  $m = n$  in int-succ-int-1)
apply simp
apply (erule natE)
apply auto
apply (rule-tac  $x = \text{succ } (n)$  in bexI)
apply auto
done

```

```

lemma zadd-succ-lemma:
   $z \in \text{int} \implies (w + \# \text{succ}(m) \leq z) \longleftrightarrow (w + \#m < z)$ 
apply (simp only: not-zless-iff-zle [THEN iff-sym] zless-add-succ-iff)

```

```

apply (auto intro: zle-anti-sym elim: zless-asm
        simp add: zless-imp-zle not-zless-iff-zle)
done

lemma zadd-succ-zle-iff: (w $+ $# succ(m) $≤ z) ↔ (w $+ $#m $< z)
apply (cut-tac z = intify (z) in zadd-succ-lemma)
apply auto
done

lemma zless-add1-iff-zle: (w $< z $+ #1) ↔ (w $≤ z)
apply (subgoal-tac #1 = $# 1)
apply (simp only: zless-add-succ-iff zle-def)
apply auto
done

lemma add1-zle-iff: (w $+ #1 $≤ z) ↔ (w $< z)
apply (subgoal-tac #1 = $# 1)
apply (simp only: zadd-succ-zle-iff)
apply auto
done

lemma add1-left-zle-iff: (#1 $+ w $≤ z) ↔ (w $< z)
apply (subst zadd-commute)
apply (rule add1-zle-iff)
done

lemma zmult-mono-lemma: k ∈ nat ⇒ i $≤ j ⇒ i $* $#k $≤ j $* $#k
apply (induct-tac k)
  prefer 2 apply (subst int-succ-int-1)
apply (simp-all (no-asm-simp) add: zadd-zmult-distrib2 zadd-zle-mono)
done

lemma zmult-zle-mono1: [i $≤ j; #0 $≤ k] ⇒ i $*k $≤ j $*k
apply (subgoal-tac i $* intify (k) $≤ j $* intify (k) )
apply (simp (no-asm-use))
apply (rule-tac b = intify (k) in not-zneg-mag [THEN subst])
apply (rule-tac [3] zmult-mono-lemma)
apply auto
apply (simp add: znegative-iff-zless-0 not-zless-iff-zle [THEN iff-sym])
done

lemma zmult-zle-mono1-neg: [i $≤ j; k $≤ #0] ⇒ j $*k $≤ i $*k
apply (rule zminus-zle-zminus [THEN iffD1])
apply (simp del: zmult-zminus-right

```

```

      add: zmult-zminus-right [symmetric] zmult-zle-mono1 zle-zminus)
done

lemma zmult-zle-mono2:  $\llbracket i \leq j; \#0 \leq k \rrbracket \implies k * i \leq k * j$ 
apply (drule zmult-zle-mono1)
apply (simp-all add: zmult-commute)
done

lemma zmult-zle-mono2-neg:  $\llbracket i \leq j; k \leq \#0 \rrbracket \implies k * j \leq k * i$ 
apply (drule zmult-zle-mono1-neg)
apply (simp-all add: zmult-commute)
done

lemma zmult-zle-mono:
   $\llbracket i \leq j; k \leq l; \#0 \leq j; \#0 \leq k \rrbracket \implies i * k \leq j * l$ 
apply (erule zmult-zle-mono1 [THEN zle-trans])
apply assumption
apply (erule zmult-zle-mono2)
apply assumption
done

lemma zmult-zless-mono2-lemma [rule-format]:
   $\llbracket i < j; k \in \text{nat} \rrbracket \implies 0 < k \longrightarrow \#k * i < \#k * j$ 
apply (induct-tac k)
prefer 2
apply (subst int-succ-int-1)
apply (erule natE)
apply (simp-all add: zadd-zmult-distrib zadd-zless-mono zle-def)
apply (frule nat-0-le)
apply (subgoal-tac i $+ (i $+ $# xa $* i) $< j $+ (j $+ $# xa $* j) )
apply (simp (no-asm-use))
apply (rule zadd-zless-mono)
apply (simp-all (no-asm-simp) add: zle-def)
done

lemma zmult-zless-mono2:  $\llbracket i < j; \#0 < k \rrbracket \implies k * i < k * j$ 
apply (subgoal-tac intify (k) $* i $< intify (k) $* j)
apply (simp (no-asm-use))
apply (rule-tac b = intify (k) in not-zneg-mag [THEN subst])
apply (rule-tac [3] zmult-zless-mono2-lemma)
apply auto
apply (simp add: znegative-iff-zless-0)
apply (drule zless-trans, assumption)
apply (auto simp add: zero-lt-zmagnitude)
done

```

```

lemma zmult-zless-mono1:  $\llbracket i \$< j; \#0 \$< k \rrbracket \implies i \$*k \$< j \$*k$ 
apply (drule zmult-zless-mono2)
apply (simp-all add: zmult-commute)
done

```

```

lemma zmult-zless-mono:
   $\llbracket i \$< j; k \$< l; \#0 \$< j; \#0 \$< k \rrbracket \implies i \$*k \$< j \$*l$ 
apply (erule zmult-zless-mono1 [THEN zless-trans])
apply assumption
apply (erule zmult-zless-mono2)
apply assumption
done

```

```

lemma zmult-zless-mono1-neg:  $\llbracket i \$< j; k \$< \#0 \rrbracket \implies j \$*k \$< i \$*k$ 
apply (rule zminus-zless-zminus [THEN iffD1])
apply (simp del: zmult-zminus-right
  add: zmult-zminus-right [symmetric] zmult-zless-mono1 zless-zminus)
done

```

```

lemma zmult-zless-mono2-neg:  $\llbracket i \$< j; k \$< \#0 \rrbracket \implies k \$*j \$< k \$*i$ 
apply (rule zminus-zless-zminus [THEN iffD1])
apply (simp del: zmult-zminus
  add: zmult-zminus [symmetric] zmult-zless-mono2 zless-zminus)
done

```

```

lemma zmult-eq-lemma:
   $\llbracket m \in \text{int}; n \in \text{int} \rrbracket \implies (m = \#0 \mid n = \#0) \longleftrightarrow (m \$*n = \#0)$ 
apply (case-tac m  $\$< \#0$ )
apply (auto simp add: not-zless-iff-zle zle-def neq-iff-zless)
apply (force dest: zmult-zless-mono1-neg zmult-zless-mono1)
done

```

```

lemma zmult-eq-0-iff [iff]:  $(m \$*n = \#0) \longleftrightarrow (\text{intify}(m) = \#0 \mid \text{intify}(n) = \#0)$ 
apply (simp add: zmult-eq-lemma)
done

```

```

lemma zmult-zless-lemma:
   $\llbracket k \in \text{int}; m \in \text{int}; n \in \text{int} \rrbracket$ 
   $\implies (m \$*k \$< n \$*k) \longleftrightarrow ((\#0 \$< k \wedge m \$< n) \mid (k \$< \#0 \wedge n \$< m))$ 
apply (case-tac k  $= \#0$ )
apply (auto simp add: neq-iff-zless zmult-zless-mono1 zmult-zless-mono1-neg)

```

apply (*auto simp add: not-zless-iff-zle*
 not-zle-iff-zless [THEN iff-sym, of m\$k]
 not-zle-iff-zless [THEN iff-sym, of m])
apply (*auto simp add: zless-imp-zle zmult-zle-mono1 zmult-zle-mono1-neg*)
done

lemma *zmult-zless-cancel2*:
 $(m\$*k \$< n\$*k) \longleftrightarrow ((\#0 \$< k \wedge m\$<n) \mid (k \$< \#0 \wedge n\$<m))$
apply (*cut-tac k = intify (k) and m = intify (m) and n = intify (n)*
 in zmult-zless-lemma)
apply *auto*
done

lemma *zmult-zless-cancel1*:
 $(k\$*m \$< k\$*n) \longleftrightarrow ((\#0 \$< k \wedge m\$<n) \mid (k \$< \#0 \wedge n\$<m))$
by (*simp add: zmult-commute [of k] zmult-zless-cancel2*)

lemma *zmult-zle-cancel2*:
 $(m\$*k \$\leq n\$*k) \longleftrightarrow ((\#0 \$< k \longrightarrow m\$ \leq n) \wedge (k \$< \#0 \longrightarrow n\$ \leq m))$
by (*auto simp add: not-zless-iff-zle [THEN iff-sym] zmult-zless-cancel2*)

lemma *zmult-zle-cancel1*:
 $(k\$*m \$\leq k\$*n) \longleftrightarrow ((\#0 \$< k \longrightarrow m\$ \leq n) \wedge (k \$< \#0 \longrightarrow n\$ \leq m))$
by (*auto simp add: not-zless-iff-zle [THEN iff-sym] zmult-zless-cancel1*)

lemma *int-eq-iff-zle*: $\llbracket m \in \text{int}; n \in \text{int} \rrbracket \implies m=n \longleftrightarrow (m \$ \leq n \wedge n \$ \leq m)$
apply (*blast intro: zle-refl zle-anti-sym*)
done

lemma *zmult-cancel2-lemma*:
 $\llbracket k \in \text{int}; m \in \text{int}; n \in \text{int} \rrbracket \implies (m\$*k = n\$*k) \longleftrightarrow (k=\#0 \mid m=n)$
apply (*simp add: int-eq-iff-zle [of m\$k] int-eq-iff-zle [of m]*)
apply (*auto simp add: zmult-zle-cancel2 neq-iff-zless*)
done

lemma *zmult-cancel2 [simp]*:
 $(m\$*k = n\$*k) \longleftrightarrow (\text{intify}(k) = \#0 \mid \text{intify}(m) = \text{intify}(n))$
apply (*rule iff-trans*)
apply (*rule-tac [2] zmult-cancel2-lemma*)
apply *auto*
done

lemma *zmult-cancel1 [simp]*:
 $(k\$*m = k\$*n) \longleftrightarrow (\text{intify}(k) = \#0 \mid \text{intify}(m) = \text{intify}(n))$
by (*simp add: zmult-commute [of k] zmult-cancel2*)

33.1 Uniqueness and monotonicity of quotients and remainders

lemma *unique-quotient-lemma*:

```

  [[b*q' $+ r' $≤ b*q $+ r; #0 $≤ r'; #0 $< b; r $< b]]
    ⇒ q' $≤ q
  apply (subgoal-tac r' $+ b $* (q'$-q) $≤ r)
  prefer 2 apply (simp add: zdiff-zmult-distrib2 zadd-ac zcompare-rls)
  apply (subgoal-tac #0 $< b $* (#1 $+ q $- q') )
  prefer 2
  apply (erule zle-zless-trans)
  apply (simp add: zdiff-zmult-distrib2 zadd-zmult-distrib2 zadd-ac zcompare-rls)
  apply (erule zle-zless-trans)
  apply simp
  apply (subgoal-tac b $* q' $< b $* (#1 $+ q))
  prefer 2
  apply (simp add: zdiff-zmult-distrib2 zadd-zmult-distrib2 zadd-ac zcompare-rls)
  apply (auto elim: zless-asym
    simp add: zmult-zless-cancel1 zless-add1-iff-zle zadd-ac zcompare-rls)
done

```

lemma *unique-quotient-lemma-neg*:

```

  [[b*q' $+ r' $≤ b*q $+ r; r $≤ #0; b $< #0; b $< r]]
    ⇒ q $≤ q'
  apply (rule-tac b = $-b and r = $-r' and r' = $-r
    in unique-quotient-lemma)
  apply (auto simp del: zminus-zadd-distrib
    simp add: zminus-zadd-distrib [symmetric] zle-zminus zless-zminus)
done

```

lemma *unique-quotient*:

```

  [[quorem (<a,b>, <q,r>); quorem (<a,b>, <q',r'>); b ∈ int; b ≠ #0;
    q ∈ int; q' ∈ int]] ⇒ q = q'
  apply (simp add: split-ifs quorem-def neq-iff-zless)
  apply safe
  apply simp-all
  apply (blast intro: zle-anti-sym
    dest: zle-eq-refl [THEN unique-quotient-lemma]
    zle-eq-refl [THEN unique-quotient-lemma-neg] sym)+
done

```

lemma *unique-remainder*:

```

  [[quorem (<a,b>, <q,r>); quorem (<a,b>, <q',r'>); b ∈ int; b ≠ #0;
    q ∈ int; q' ∈ int;
    r ∈ int; r' ∈ int]] ⇒ r = r'
  apply (subgoal-tac q = q')
  prefer 2 apply (blast intro: unique-quotient)
  apply (simp add: quorem-def)
done

```

33.2 Correctness of posDivAlg, the Division Algorithm for $a \geq 0$ and $b > 0$

lemma *adjust-eq* [simp]:

adjust($b, \langle q, r \rangle$) = (let $\text{diff} = r \$ - b$ in
 if $\#0 \$ \leq \text{diff}$ then $\langle \#2 \$ * q \$ + \#1, \text{diff} \rangle$
 else $\langle \#2 \$ * q, r \rangle$)

by (*simp add: Let-def adjust-def*)

lemma *posDivAlg-termination*:

$\llbracket \#0 \$ < b; \neg a \$ < b \rrbracket$
 $\implies \text{nat-of}(a \$ - \#2 \$ * b \$ + \#1) < \text{nat-of}(a \$ - b \$ + \#1)$

apply (*simp (no-asm) add: zless-nat-conj*)

apply (*simp add: not-zless-iff-zle zless-add1-iff-zle zcompare-rls*)

done

lemmas *posDivAlg-unfold* = *def-wfrec* [*OF posDivAlg-def wf-measure*]

lemma *posDivAlg-eqn*:

$\llbracket \#0 \$ < b; a \in \text{int}; b \in \text{int} \rrbracket \implies$
 $\text{posDivAlg}(\langle a, b \rangle) =$
 (if $a \$ < b$ then $\langle \#0, a \rangle$ else *adjust*($b, \text{posDivAlg}(\langle a, \#2 \$ * b \rangle)$))

apply (*rule posDivAlg-unfold [THEN trans]*)

apply (*simp add: vimage-iff not-zless-iff-zle [THEN iff-sym]*)

apply (*blast intro: posDivAlg-termination*)

done

lemma *posDivAlg-induct-lemma* [*rule-format*]:

assumes *prem*:

$\bigwedge a \ b. \llbracket a \in \text{int}; b \in \text{int};$
 $\neg (a \$ < b \mid b \$ \leq \#0) \longrightarrow P(\langle a, \#2 \$ * b \rangle) \rrbracket \implies P(\langle a, b \rangle)$

shows $\langle u, v \rangle \in \text{int} * \text{int} \implies P(\langle u, v \rangle)$

using *wf-measure* [**where** $A = \text{int} * \text{int}$ **and** $f = \lambda \langle a, b \rangle. \text{nat-of}(a \$ - b \$ + \#1)$]

proof (*induct* $\langle u, v \rangle$ *arbitrary: u v rule: wf-induct*)

case (*step x*)

hence *uv*: $u \in \text{int} \ v \in \text{int}$ **by** *auto*

thus *?case*

apply (*rule prem*)

apply (*rule impI*)

apply (*rule step*)

apply (*auto simp add: step uv not-zle-iff-zless posDivAlg-termination*)

done

qed

lemma *posDivAlg-induct* [*consumes 2*]:

assumes *u-int*: $u \in \text{int}$

and *v-int*: $v \in \text{int}$

and *ih*: $\bigwedge a \ b. \llbracket a \in \text{int}; b \in \text{int};$

```

       $\neg (a \leq b \mid b \leq \#0) \longrightarrow P(a, \#2 \ast b) \Longrightarrow P(a, b)$ 
    shows  $P(u, v)$ 
  apply (subgoal-tac ( $\lambda(x, y). P(x, y)$ ) ( $\langle u, v \rangle$ ))
  apply simp
  apply (rule posDivAlg-induct-lemma)
  apply (simp (no-asm-use))
  apply (rule ih)
  apply (auto simp add: u-int v-int)
done

```

```

lemma intify-eq-0-iff-zle: intify( $m$ ) =  $\#0 \longleftrightarrow (m \leq \#0 \wedge \#0 \leq m)$ 
  by (simp add: int-eq-iff-zle)

```

33.3 Some convenient biconditionals for products of signs

```

lemma zmult-pos:  $\llbracket \#0 \leq i; \#0 \leq j \rrbracket \Longrightarrow \#0 \leq i \ast j$ 
  by (drule zmult-zless-mono1, auto)

```

```

lemma zmult-neg:  $\llbracket i \leq \#0; j \leq \#0 \rrbracket \Longrightarrow \#0 \leq i \ast j$ 
  by (drule zmult-zless-mono1-neg, auto)

```

```

lemma zmult-pos-neg:  $\llbracket \#0 \leq i; j \leq \#0 \rrbracket \Longrightarrow i \ast j \leq \#0$ 
  by (drule zmult-zless-mono1-neg, auto)

```

```

lemma int-0-less-lemma:
   $\llbracket x \in \text{int}; y \in \text{int} \rrbracket$ 
   $\Longrightarrow (\#0 \leq x \ast y) \longleftrightarrow (\#0 \leq x \wedge \#0 \leq y \mid x \leq \#0 \wedge y \leq \#0)$ 
  apply (auto simp add: zle-def not-zless-iff-zle zmult-pos zmult-neg)
  apply (rule ccontr)
  apply (rule-tac [2] ccontr)
  apply (auto simp add: zle-def not-zless-iff-zle)
  apply (erule-tac  $P = \#0 \leq x \ast y$  in rev-mp)
  apply (erule-tac [2]  $P = \#0 \leq x \ast y$  in rev-mp)
  apply (drule zmult-pos-neg, assumption)
  prefer 2
  apply (drule zmult-pos-neg, assumption)
  apply (auto dest: zless-not-sym simp add: zmult-commute)
done

```

```

lemma int-0-less-mult-iff:
   $(\#0 \leq x \ast y) \longleftrightarrow (\#0 \leq x \wedge \#0 \leq y \mid x \leq \#0 \wedge y \leq \#0)$ 
  apply (cut-tac  $x = \text{intify}(x)$  and  $y = \text{intify}(y)$  in int-0-less-lemma)
  apply auto
done

```


lemma *int-0-le-lemma*:

$$\llbracket x \in \text{int}; y \in \text{int} \rrbracket \implies (\#0 \leq x * y) \longleftrightarrow (\#0 \leq x \wedge \#0 \leq y \mid x \leq \#0 \wedge y \leq \#0)$$

by (*auto simp add: zle-def not-zless-iff-zle int-0-less-mult-iff*)

lemma *int-0-le-mult-iff*:

$$(\#0 \leq x * y) \longleftrightarrow ((\#0 \leq x \wedge \#0 \leq y) \mid (x \leq \#0 \wedge y \leq \#0))$$

apply (*cut-tac x = intify (x) and y = intify (y) in int-0-le-lemma*)
apply *auto*
done

lemma *zmult-less-0-iff*:

$$(x * y < \#0) \longleftrightarrow (\#0 < x \wedge y < \#0 \mid x < \#0 \wedge \#0 < y)$$

apply (*auto simp add: int-0-le-mult-iff not-zle-iff-zless [THEN iff-sym]*)
apply (*auto dest: zless-not-sym simp add: not-zle-iff-zless*)
done

lemma *zmult-le-0-iff*:

$$(x * y \leq \#0) \longleftrightarrow (\#0 \leq x \wedge y \leq \#0 \mid x \leq \#0 \wedge \#0 \leq y)$$

by (*auto dest: zless-not-sym simp add: int-0-less-mult-iff not-zless-iff-zle [THEN iff-sym]*)

lemma *posDivAlg-type* [*rule-format*]:

$$\llbracket a \in \text{int}; b \in \text{int} \rrbracket \implies \text{posDivAlg}(\langle a, b \rangle) \in \text{int} * \text{int}$$

apply (*rule-tac u = a and v = b in posDivAlg-induct*)
apply *assumption+*
apply (*case-tac #0 < ba*)
apply (*simp add: posDivAlg-eqn adjust-def integ-of-type split: split-if-asm*)
apply *clarify*
apply (*simp add: int-0-less-mult-iff not-zle-iff-zless*)
apply (*simp add: not-zless-iff-zle*)
apply (*subst posDivAlg-unfold*)
apply *simp*
done

lemma *posDivAlg-correct* [*rule-format*]:

$$\llbracket a \in \text{int}; b \in \text{int} \rrbracket \implies \#0 \leq a \longrightarrow \#0 < b \longrightarrow \text{quorem}(\langle a, b \rangle, \text{posDivAlg}(\langle a, b \rangle))$$

apply (*rule-tac u = a and v = b in posDivAlg-induct*)
apply *auto*
apply (*simp-all add: quorem-def*)

base case: $a < b$

apply (*simp add: posDivAlg-eqn*)
apply (*simp add: not-zless-iff-zle [THEN iff-sym]*)

```

apply (simp add: int-0-less-mult-iff)

main argument
apply (subst posDivAlg-eqn)
apply (simp-all (no-asm-simp))
apply (erule splitE)
apply (rule posDivAlg-type)
apply (simp-all add: int-0-less-mult-iff)
apply (auto simp add: zadd-zmult-distrib2 Let-def)

now just linear arithmetic
apply (simp add: not-zle-iff-zless zdiff-zless-iff)
done

```

33.4 Correctness of negDivAlg, the division algorithm for $a < 0$ and $b > 0$

```

lemma negDivAlg-termination:
   $\llbracket \#0 \leq b; a \leq b \leq \#0 \rrbracket$ 
   $\implies \text{nat-of}(\$- a \$- \#2 \$* b) < \text{nat-of}(\$- a \$- b)$ 
apply (simp (no-asm) add: zless-nat-conj)
apply (simp add: zcompare-rls not-zle-iff-zless zless-zdiff-iff [THEN iff-sym]
  zless-zminus)
done

lemmas negDivAlg-unfold = def-wfrec [OF negDivAlg-def wf-measure]

lemma negDivAlg-eqn:
   $\llbracket \#0 \leq b; a \in \text{int}; b \in \text{int} \rrbracket \implies$ 
   $\text{negDivAlg}(\langle a, b \rangle) =$ 
   $(\text{if } \#0 \leq a + b \text{ then } \langle \#-1, a + b \rangle$ 
   $\text{else } \text{adjust}(b, \text{negDivAlg}(\langle a, \#2 \$* b \rangle)))$ 
apply (rule negDivAlg-unfold [THEN trans])
apply (simp (no-asm-simp) add: vimage-iff not-zless-iff-zle [THEN iff-sym])
apply (blast intro: negDivAlg-termination)
done

lemma negDivAlg-induct-lemma [rule-format]:
  assumes prem:
     $\bigwedge a b. \llbracket a \in \text{int}; b \in \text{int};$ 
     $\neg (\#0 \leq a + b \mid b \leq \#0) \longrightarrow P(\langle a, \#2 \$* b \rangle)\rrbracket$ 
     $\implies P(\langle a, b \rangle)$ 
  shows  $\langle u, v \rangle \in \text{int} * \text{int} \implies P(\langle u, v \rangle)$ 
using wf-measure [where  $A = \text{int} * \text{int}$  and  $f = \lambda \langle a, b \rangle. \text{nat-of}(\$- a \$- b)$ ]
proof (induct  $\langle u, v \rangle$  arbitrary:  $u v$  rule: wf-induct)
  case (step x)
  hence  $uv: u \in \text{int} v \in \text{int}$  by auto
  thus ?case
  apply (rule prem)

```

```

    apply (rule impI)
    apply (rule step)
    apply (auto simp add: step uv not-zle-iff-zless negDivAlg-termination)
  done
qed

```

```

lemma negDivAlg-induct [consumes 2]:
  assumes u-int:  $u \in \text{int}$ 
    and v-int:  $v \in \text{int}$ 
    and ih:  $\bigwedge a b. \llbracket a \in \text{int}; b \in \text{int};$ 
       $\neg (\#0 \leq a + b \mid b \leq \#0) \longrightarrow P(a, \#2 * b) \rrbracket$ 
       $\implies P(a, b)$ 
  shows  $P(u, v)$ 
  apply (subgoal-tac ( $\lambda \langle x, y \rangle. P(x, y)$ ) ( $\langle u, v \rangle$ ))
  apply simp
  apply (rule negDivAlg-induct-lemma)
  apply (simp (no-asm-use))
  apply (rule ih)
  apply (auto simp add: u-int v-int)
  done

```

```

lemma negDivAlg-type:
   $\llbracket a \in \text{int}; b \in \text{int} \rrbracket \implies \text{negDivAlg}(\langle a, b \rangle) \in \text{int} * \text{int}$ 
  apply (rule-tac  $u = a$  and  $v = b$  in negDivAlg-induct)
  apply assumption+
  apply (case-tac  $\#0 \leq ba$ )
  apply (simp add: negDivAlg-eqn adjust-def integ-of-type
    split: split-if-asm)
  apply clarify
  apply (simp add: int-0-less-mult-iff not-zle-iff-zless)
  apply (simp add: not-zless-iff-zle)
  apply (subst negDivAlg-unfold)
  apply simp
  done

```

```

lemma negDivAlg-correct [rule-format]:
   $\llbracket a \in \text{int}; b \in \text{int} \rrbracket$ 
   $\implies a \leq \#0 \longrightarrow \#0 \leq b \longrightarrow \text{quorem}(\langle a, b \rangle, \text{negDivAlg}(\langle a, b \rangle))$ 
  apply (rule-tac  $u = a$  and  $v = b$  in negDivAlg-induct)
  apply auto
  apply (simp-all add: quorem-def)

```

base case: $0 \leq a + b$

```

  apply (simp add: negDivAlg-eqn)
  apply (simp add: not-zless-iff-zle [THEN iff-sym])

```

```

apply (simp add: int-0-less-mult-iff)

main argument
apply (subst negDivAlg-eqn)
apply (simp-all (no-asm-simp))
apply (erule splitE)
apply (rule negDivAlg-type)
apply (simp-all add: int-0-less-mult-iff)
apply (auto simp add: zadd-zmult-distrib2 Let-def)

now just linear arithmetic
apply (simp add: not-zle-iff-zless zdiff-zless-iff)
done

```

33.5 Existence shown by proving the division algorithm to be correct

```

lemma quorem-0:  $\llbracket b \neq \#0; \ b \in \text{int} \rrbracket \implies \text{quorem} (<\#0, b>, <\#0, \#0>)$ 
by (force simp add: quorem-def neq-iff-zless)

```

```

lemma posDivAlg-zero-divisor:  $\text{posDivAlg}(<a, \#0>) = <\#0, a>$ 
apply (subst posDivAlg-unfold)
apply simp
done

```

```

lemma posDivAlg-0 [simp]:  $\text{posDivAlg} (<\#0, b>) = <\#0, \#0>$ 
apply (subst posDivAlg-unfold)
apply (simp add: not-zle-iff-zless)
done

```

```

lemma linear-arith-lemma:  $\neg (\#0 \ \$ \leq \#-1 \ \$ + b) \implies (b \ \$ \leq \#0)$ 
apply (simp add: not-zle-iff-zless)
apply (drule zminus-zless-zminus [THEN iffD2])
apply (simp add: zadd-commute zless-add1-iff-zle zle-zminus)
done

```

```

lemma negDivAlg-minus1 [simp]:  $\text{negDivAlg} (<\#-1, b>) = <\#-1, b \ \$ - \#1>$ 
apply (subst negDivAlg-unfold)
apply (simp add: linear-arith-lemma integ-of-type vimage-iff)
done

```

```

lemma negateSnd-eq [simp]:  $\text{negateSnd} (\langle q, r \rangle) = \langle q, \$-r \rangle$ 
unfolding negateSnd-def
apply auto
done

```

```

lemma negateSnd-type:  $qr \in \text{int} * \text{int} \implies \text{negateSnd} (qr) \in \text{int} * \text{int}$ 

```

```

unfolding negateSnd-def
apply auto
done

lemma quorem-neg:
   $\llbracket \text{quorem } (\langle \$-a, \$-b \rangle, qr); a \in \text{int}; b \in \text{int}; qr \in \text{int} * \text{int} \rrbracket$ 
   $\implies \text{quorem } (\langle a, b \rangle, \text{negateSnd}(qr))$ 
apply clarify
apply (auto elim: zless-asym simp add: quorem-def zless-zminus)

```

linear arithmetic from here on

```

apply (simp-all add: zminus-equation [of a] zminus-zless)
apply (cut-tac [2] z = b and w = #0 in zless-linear)
apply (cut-tac [1] z = b and w = #0 in zless-linear)
apply auto
apply (blast dest: zle-zless-trans)+
done

```

```

lemma divAlg-correct:
   $\llbracket b \neq \#0; a \in \text{int}; b \in \text{int} \rrbracket \implies \text{quorem } (\langle a, b \rangle, \text{divAlg}(\langle a, b \rangle))$ 
apply (auto simp add: quorem-0 divAlg-def)
apply (safe intro!: quorem-neg posDivAlg-correct negDivAlg-correct
  posDivAlg-type negDivAlg-type)
apply (auto simp add: quorem-def neg-iff-zless)

```

linear arithmetic from here on

```

apply (auto simp add: zle-def)
done

```

```

lemma divAlg-type:  $\llbracket a \in \text{int}; b \in \text{int} \rrbracket \implies \text{divAlg}(\langle a, b \rangle) \in \text{int} * \text{int}$ 
apply (auto simp add: divAlg-def)
apply (auto simp add: posDivAlg-type negDivAlg-type negateSnd-type)
done

```

```

lemma zdiv-intify1 [simp]:  $\text{intify}(x) \text{ zdiv } y = x \text{ zdiv } y$ 
by (simp add: zdiv-def)

```

```

lemma zdiv-intify2 [simp]:  $x \text{ zdiv } \text{intify}(y) = x \text{ zdiv } y$ 
by (simp add: zdiv-def)

```

```

lemma zdiv-type [iff, TC]:  $z \text{ zdiv } w \in \text{int}$ 
unfolding zdiv-def
apply (blast intro: fst-type divAlg-type)
done

```

```

lemma zmod-intify1 [simp]:  $\text{intify}(x) \text{ zmod } y = x \text{ zmod } y$ 

```

```

by (simp add: zmod-def)

lemma zmod-intify2 [simp]:  $x \text{ zmod } \text{intify}(y) = x \text{ zmod } y$ 
  by (simp add: zmod-def)

lemma zmod-type [iff, TC]:  $z \text{ zmod } w \in \text{int}$ 
  unfolding zmod-def
  apply (rule snd-type)
  apply (blast intro: divAlg-type)
  done

lemma DIVISION-BY-ZERO-ZDIV:  $a \text{ zdiv } \#0 = \#0$ 
  by (simp add: zdiv-def divAlg-def posDivAlg-zero-divisor)

lemma DIVISION-BY-ZERO-ZMOD:  $a \text{ zmod } \#0 = \text{intify}(a)$ 
  by (simp add: zmod-def divAlg-def posDivAlg-zero-divisor)

lemma raw-zmod-zdiv-equality:
   $\llbracket a \in \text{int}; b \in \text{int} \rrbracket \implies a = b \$* (a \text{ zdiv } b) \$+ (a \text{ zmod } b)$ 
  apply (case-tac b = #0)
  apply (simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD)
  apply (cut-tac a = a and b = b in divAlg-correct)
  apply (auto simp add: quorem-def zdiv-def zmod-def split-def)
  done

lemma zmod-zdiv-equality:  $\text{intify}(a) = b \$* (a \text{ zdiv } b) \$+ (a \text{ zmod } b)$ 
  apply (rule trans)
  apply (rule-tac b = intify (b) in raw-zmod-zdiv-equality)
  apply auto
  done

lemma pos-mod:  $\#0 \$< b \implies \#0 \$\leq a \text{ zmod } b \wedge a \text{ zmod } b \$< b$ 
  apply (cut-tac a = intify (a) and b = intify (b) in divAlg-correct)
  apply (auto simp add: intify-eq-0-iff-zle quorem-def zmod-def split-def)
  apply (blast dest: zle-zless-trans)+
  done

lemmas pos-mod-sign = pos-mod [THEN conjunct1]
  and pos-mod-bound = pos-mod [THEN conjunct2]

lemma neg-mod:  $b \$< \#0 \implies a \text{ zmod } b \$\leq \#0 \wedge b \$< a \text{ zmod } b$ 
  apply (cut-tac a = intify (a) and b = intify (b) in divAlg-correct)
  apply (auto simp add: intify-eq-0-iff-zle quorem-def zmod-def split-def)

```

apply (*blast dest: zle-zless-trans*)
apply (*blast dest: zless-trans*) +
done

lemmas *neg-mod-sign* = *neg-mod* [*THEN conjunct1*]
and *neg-mod-bound* = *neg-mod* [*THEN conjunct2*]

lemma *quorem-div-mod*:
 $\llbracket b \neq \#0; a \in \text{int}; b \in \text{int} \rrbracket$
 $\implies \text{quorem}(\langle a, b \rangle, \langle a \text{ zdiv } b, a \text{ zmod } b \rangle)$
apply (*cut-tac a = a and b = b in zmod-zdiv-equality*)
apply (*auto simp add: quorem-def neg-iff-zless pos-mod-sign pos-mod-bound*
neg-mod-sign neg-mod-bound)
done

lemma *quorem-div*:
 $\llbracket \text{quorem}(\langle a, b \rangle, \langle q, r \rangle); b \neq \#0; a \in \text{int}; b \in \text{int}; q \in \text{int} \rrbracket$
 $\implies a \text{ zdiv } b = q$
by (*blast intro: quorem-div-mod [THEN unique-quotient]*)

lemma *quorem-mod*:
 $\llbracket \text{quorem}(\langle a, b \rangle, \langle q, r \rangle); b \neq \#0; a \in \text{int}; b \in \text{int}; q \in \text{int}; r \in \text{int} \rrbracket$
 $\implies a \text{ zmod } b = r$
by (*blast intro: quorem-div-mod [THEN unique-remainder]*)

lemma *zdiv-pos-pos-trivial-raw*:
 $\llbracket a \in \text{int}; b \in \text{int}; \#0 \leq a; a \leq b \rrbracket \implies a \text{ zdiv } b = \#0$
apply (*rule quorem-div*)
apply (*auto simp add: quorem-def*)

apply (*blast dest: zle-zless-trans*) +
done

lemma *zdiv-pos-pos-trivial*: $\llbracket \#0 \leq a; a \leq b \rrbracket \implies a \text{ zdiv } b = \#0$
apply (*cut-tac a = intify (a) and b = intify (b)*
in zdiv-pos-pos-trivial-raw)
apply *auto*
done

lemma *zdiv-neg-neg-trivial-raw*:
 $\llbracket a \in \text{int}; b \in \text{int}; a \leq \#0; b \leq a \rrbracket \implies a \text{ zdiv } b = \#0$
apply (*rule-tac r = a in quorem-div*)
apply (*auto simp add: quorem-def*)

apply (*blast dest: zle-zless-trans zless-trans*) +

done

lemma *zdiv-neg-neg-trivial*: $\llbracket a \leq \#0; b < a \rrbracket \implies a \text{ zdiv } b = \#0$
apply (*cut-tac* $a = \text{intify } (a)$ **and** $b = \text{intify } (b)$)
 in *zdiv-neg-neg-trivial-raw*)
apply *auto*
done

lemma *zadd-le-0-lemma*: $\llbracket a + b \leq \#0; \#0 < a; \#0 < b \rrbracket \implies \text{False}$
apply (*drule-tac* $z' = \#0$ **and** $z = b$ **in** *zadd-zless-mono*)
apply (*auto simp add: zle-def*)
apply (*blast dest: zless-trans*)
done

lemma *zdiv-pos-neg-trivial-raw*:
 $\llbracket a \in \text{int}; b \in \text{int}; \#0 < a; a + b \leq \#0 \rrbracket \implies a \text{ zdiv } b = \#-1$
apply (*rule-tac* $r = a + b$ **in** *quorem-div*)
apply (*auto simp add: quorem-def*)

apply (*blast dest: zadd-le-0-lemma zle-zless-trans*) +
done

lemma *zdiv-pos-neg-trivial*: $\llbracket \#0 < a; a + b \leq \#0 \rrbracket \implies a \text{ zdiv } b = \#-1$
apply (*cut-tac* $a = \text{intify } (a)$ **and** $b = \text{intify } (b)$)
 in *zdiv-pos-neg-trivial-raw*)
apply *auto*
done

lemma *zmod-pos-pos-trivial-raw*:
 $\llbracket a \in \text{int}; b \in \text{int}; \#0 \leq a; a < b \rrbracket \implies a \text{ zmod } b = a$
apply (*rule-tac* $q = \#0$ **in** *quorem-mod*)
apply (*auto simp add: quorem-def*)

apply (*blast dest: zle-zless-trans*) +
done

lemma *zmod-pos-pos-trivial*: $\llbracket \#0 \leq a; a < b \rrbracket \implies a \text{ zmod } b = \text{intify}(a)$
apply (*cut-tac* $a = \text{intify } (a)$ **and** $b = \text{intify } (b)$)
 in *zmod-pos-pos-trivial-raw*)
apply *auto*
done

lemma *zmod-neg-neg-trivial-raw*:
 $\llbracket a \in \text{int}; b \in \text{int}; a \leq \#0; b < a \rrbracket \implies a \text{ zmod } b = a$
apply (*rule-tac* $q = \#0$ **in** *quorem-mod*)
apply (*auto simp add: quorem-def*)

apply (*blast dest: zle-zless-trans zless-trans*) +
done

lemma *zmod-neg-neg-trivial*: $\llbracket a \leq \#0; b < a \rrbracket \implies a \text{ zmod } b = \text{intify}(a)$
apply (*cut-tac a = intify (a) and b = intify (b)*
 in *zmod-neg-neg-trivial-raw*)
apply *auto*
done

lemma *zmod-pos-neg-trivial-raw*:
 $\llbracket a \in \text{int}; b \in \text{int}; \#0 < a; a + b \leq \#0 \rrbracket \implies a \text{ zmod } b = a + b$
apply (*rule-tac q = #-1 in quorem-mod*)
apply (*auto simp add: quorem-def*)

apply (*blast dest: zadd-le-0-lemma zle-zless-trans*) +
done

lemma *zmod-pos-neg-trivial*: $\llbracket \#0 < a; a + b \leq \#0 \rrbracket \implies a \text{ zmod } b = a + b$
apply (*cut-tac a = intify (a) and b = intify (b)*
 in *zmod-pos-neg-trivial-raw*)
apply *auto*
done

lemma *zdiv-zminus-zminus-raw*:
 $\llbracket a \in \text{int}; b \in \text{int} \rrbracket \implies (\$-a) \text{ zdiv } (\$-b) = a \text{ zdiv } b$
apply (*case-tac b = \#0*)
apply (*simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*subst quorem-div-mod [THEN quorem-neg, simplified, THEN quorem-div]*)
apply *auto*
done

lemma *zdiv-zminus-zminus [simp]*: $(\$-a) \text{ zdiv } (\$-b) = a \text{ zdiv } b$
apply (*cut-tac a = intify (a) and b = intify (b) in zdiv-zminus-zminus-raw*)
apply *auto*
done

lemma *zmod-zminus-zminus-raw*:
 $\llbracket a \in \text{int}; b \in \text{int} \rrbracket \implies (\$-a) \text{ zmod } (\$-b) = \$- (a \text{ zmod } b)$
apply (*case-tac b = \#0*)
apply (*simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*subst quorem-div-mod [THEN quorem-neg, simplified, THEN quorem-mod]*)
apply *auto*

done

lemma *zmod-zminus-zminus* [*simp*]: $(\$-a) \text{ zmod } (\$-b) = \$- (a \text{ zmod } b)$
apply (*cut-tac* $a = \text{intify } (a)$ **and** $b = \text{intify } (b)$ **in** *zmod-zminus-zminus-raw*)
apply *auto*
done

33.6 division of a number by itself

lemma *self-quotient-aux1*: $\llbracket \#0 \$< a; a = r \$+ a\$*q; r \$< a \rrbracket \implies \#1 \$\leq q$
apply (*subgoal-tac* $\#0 \$< a\$*q$)
apply (*cut-tac* $w = \#0$ **and** $z = q$ **in** *add1-zle-iff*)
apply (*simp add: int-0-less-mult-iff*)
apply (*blast dest: zless-trans*)

apply (*drule-tac* $t = \lambda x. x \$- r$ **in** *subst-context*)
apply (*drule sym*)
apply (*simp add: zcompare-rls*)
done

lemma *self-quotient-aux2*: $\llbracket \#0 \$< a; a = r \$+ a\$*q; \#0 \$\leq r \rrbracket \implies q \$\leq \#1$
apply (*subgoal-tac* $\#0 \$\leq a\$* (\#1 \$-q)$)
apply (*simp add: int-0-le-mult-iff zcompare-rls*)
apply (*blast dest: zle-zless-trans*)
apply (*simp add: zdiff-zmult-distrib2*)
apply (*drule-tac* $t = \lambda x. x \$- a \$* q$ **in** *subst-context*)
apply (*simp add: zcompare-rls*)
done

lemma *self-quotient*:

$\llbracket \text{quorem}(\langle a, a \rangle, \langle q, r \rangle); a \in \text{int}; q \in \text{int}; a \neq \#0 \rrbracket \implies q = \#1$
apply (*simp add: split-ifs quorem-def neq-iff-zless*)
apply (*rule zle-anti-sym*)
apply *safe*
apply *auto*
prefer 4 **apply** (*blast dest: zless-trans*)
apply (*blast dest: zless-trans*)
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 (*rule-tac* [6] *zminus-equation* [*THEN iffD1*])
apply (*rule-tac* [2] *zminus-equation* [*THEN iffD1*])
apply (*force intro: self-quotient-aux1 self-quotient-aux2*
simp add: zadd-commute zmult-zminus)
done

lemma *self-remainder*:

$\llbracket \text{quorem}(\langle a, a \rangle, \langle q, r \rangle); a \in \text{int}; q \in \text{int}; r \in \text{int}; a \neq \#0 \rrbracket \implies r = \#0$
apply (*frule self-quotient*)
apply (*auto simp add: quorem-def*)

done

lemma *zdiv-self-raw*: $\llbracket a \neq \#0; a \in \text{int} \rrbracket \implies a \text{ zdiv } a = \#1$
apply (*blast intro: quorem-div-mod [THEN self-quotient]*)
done

lemma *zdiv-self [simp]*: $\text{intify}(a) \neq \#0 \implies a \text{ zdiv } a = \#1$
apply (*drule zdiv-self-raw*)
apply *auto*
done

lemma *zmod-self-raw*: $a \in \text{int} \implies a \text{ zmod } a = \#0$
apply (*case-tac a = #0*)
apply (*simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*blast intro: quorem-div-mod [THEN self-remainder]*)
done

lemma *zmod-self [simp]*: $a \text{ zmod } a = \#0$
apply (*cut-tac a = intify (a) in zmod-self-raw*)
apply *auto*
done

33.7 Computation of division and remainder

lemma *zdiv-zero [simp]*: $\#0 \text{ zdiv } b = \#0$
by (*simp add: zdiv-def divAlg-def*)

lemma *zdiv-eq-minus1*: $\#0 \text{ \$< } b \implies \#-1 \text{ zdiv } b = \#-1$
by (*simp (no-asm-simp) add: zdiv-def divAlg-def*)

lemma *zmod-zero [simp]*: $\#0 \text{ zmod } b = \#0$
by (*simp add: zmod-def divAlg-def*)

lemma *zdiv-minus1*: $\#0 \text{ \$< } b \implies \#-1 \text{ zdiv } b = \#-1$
by (*simp add: zdiv-def divAlg-def*)

lemma *zmod-minus1*: $\#0 \text{ \$< } b \implies \#-1 \text{ zmod } b = b \text{ \$- } \#1$
by (*simp add: zmod-def divAlg-def*)

lemma *zdiv-pos-pos*: $\llbracket \#0 \text{ \$< } a; \#0 \text{ \$}\leq b \rrbracket$
 $\implies a \text{ zdiv } b = \text{fst } (\text{posDivAlg}(<\text{intify}(a), \text{intify}(b)>))$
apply (*simp (no-asm-simp) add: zdiv-def divAlg-def*)
apply (*auto simp add: zle-def*)
done

lemma *zmod-pos-pos*:

```

    [[#0 $< a; #0 $≤ b]]
    ⇒ a zmod b = snd (posDivAlg(<intify(a), intify(b)>))
  apply (simp (no-asm-simp) add: zmod-def divAlg-def)
  apply (auto simp add: zle-def)
done

```

```

lemma zdiv-neg-pos:
  [[a $< #0; #0 $< b]]
  ⇒ a zdiv b = fst (negDivAlg(<intify(a), intify(b)>))
  apply (simp (no-asm-simp) add: zdiv-def divAlg-def)
  apply (blast dest: zle-zless-trans)
done

```

```

lemma zmod-neg-pos:
  [[a $< #0; #0 $< b]]
  ⇒ a zmod b = snd (negDivAlg(<intify(a), intify(b)>))
  apply (simp (no-asm-simp) add: zmod-def divAlg-def)
  apply (blast dest: zle-zless-trans)
done

```

```

lemma zdiv-pos-neg:
  [[#0 $< a; b $< #0]]
  ⇒ a zdiv b = fst (negateSnd(negDivAlg (<$-a, $-b>)))
  apply (simp (no-asm-simp) add: zdiv-def divAlg-def intify-eq-0-iff-zle)
  apply auto
  apply (blast dest: zle-zless-trans)+
  apply (blast dest: zless-trans)
  apply (blast intro: zless-imp-zle)
done

```

```

lemma zmod-pos-neg:
  [[#0 $< a; b $< #0]]
  ⇒ a zmod b = snd (negateSnd(negDivAlg (<$-a, $-b>)))
  apply (simp (no-asm-simp) add: zmod-def divAlg-def intify-eq-0-iff-zle)
  apply auto
  apply (blast dest: zle-zless-trans)+
  apply (blast dest: zless-trans)
  apply (blast intro: zless-imp-zle)
done

```

```

lemma zdiv-neg-neg:
  [[a $< #0; b $≤ #0]]
  ⇒ a zdiv b = fst (negateSnd(posDivAlg(<$-a, $-b>)))

```

```

apply (simp (no-asm-simp) add: zdiv-def divAlg-def)
apply auto
apply (blast dest!: zle-zless-trans) +
done

```

```

lemma zmod-neg-neg:
   $\llbracket a \leq \#0; b \leq \#0 \rrbracket$ 
   $\implies a \text{ zmod } b = \text{snd}(\text{negateSnd}(\text{posDivAlg}(\langle \$-a, \$-b \rangle)))$ 
apply (simp (no-asm-simp) add: zmod-def divAlg-def)
apply auto
apply (blast dest!: zle-zless-trans) +
done

```

```

declare zdiv-pos-pos [of integ-of (v) integ-of (w), simp] for v w
declare zdiv-neg-pos [of integ-of (v) integ-of (w), simp] for v w
declare zdiv-pos-neg [of integ-of (v) integ-of (w), simp] for v w
declare zdiv-neg-neg [of integ-of (v) integ-of (w), simp] for v w
declare zmod-pos-pos [of integ-of (v) integ-of (w), simp] for v w
declare zmod-neg-pos [of integ-of (v) integ-of (w), simp] for v w
declare zmod-pos-neg [of integ-of (v) integ-of (w), simp] for v w
declare zmod-neg-neg [of integ-of (v) integ-of (w), simp] for v w
declare posDivAlg-eqn [of concl: integ-of (v) integ-of (w), simp] for v w
declare negDivAlg-eqn [of concl: integ-of (v) integ-of (w), simp] for v w

```

```

lemma zmod-1 [simp]: a zmod #1 = #0
apply (cut-tac a = a and b = #1 in pos-mod-sign)
apply (cut-tac [2] a = a and b = #1 in pos-mod-bound)
apply auto

```

```

apply (drule add1-zle-iff [THEN iffD2])
apply (rule zle-anti-sym)
apply auto
done

```

```

lemma zdiv-1 [simp]: a zdiv #1 = intify(a)
apply (cut-tac a = a and b = #1 in zmod-zdiv-equality)
apply auto
done

```

```

lemma zmod-minus1-right [simp]: a zmod #-1 = #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

```

```

apply (drule add1-zle-iff [THEN iffD2])
apply (rule zle-anti-sym)

```

apply auto
done

lemma *zdiv-minus1-right-raw*: $a \in \text{int} \implies a \text{ zdiv } \#-1 = \$-a$
 apply (cut-tac $a = a$ and $b = \#-1$ in *zmod-zdiv-equality*)
 apply auto
 apply (rule *equation-zminus* [THEN *iffD2*])
 apply auto
 done

lemma *zdiv-minus1-right*: $a \text{ zdiv } \#-1 = \$-a$
 apply (cut-tac $a = \text{intify } (a)$ in *zdiv-minus1-right-raw*)
 apply auto
 done
 declare *zdiv-minus1-right* [simp]

33.8 Monotonicity in the first argument (divisor)

lemma *zdiv-mono1*: $\llbracket a \leq a'; \#0 \leq b \rrbracket \implies a \text{ zdiv } b \leq a' \text{ zdiv } b$
 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*)
 apply (simp-all (no-asm-simp) add: *pos-mod-sign pos-mod-bound*)
 done

lemma *zdiv-mono1-neg*: $\llbracket a \leq a'; b \leq \#0 \rrbracket \implies a' \text{ zdiv } b \leq a \text{ zdiv } b$
 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*)
 apply (simp-all (no-asm-simp) add: *neg-mod-sign neg-mod-bound*)
 done

33.9 Monotonicity in the second argument (dividend)

lemma *q-pos-lemma*:
 $\llbracket \#0 \leq b' * q' \leq r'; r' \leq b'; \#0 \leq b \rrbracket \implies \#0 \leq q'$
 apply (subgoal-tac $\#0 \leq b' * (q' \leq \#1)$)
 apply (simp add: *int-0-less-mult-iff*)
 apply (blast dest: *zless-trans intro: zless-add1-iff-zle* [THEN *iffD1*])
 apply (simp add: *zadd-zmult-distrib2*)
 apply (erule *zle-zless-trans*)
 apply (erule *zadd-zless-mono2*)
 done

lemma *zdiv-mono2-lemma*:
 $\llbracket b * q \leq r = b' * q' \leq r'; \#0 \leq b' * q' \leq r' \rrbracket$

```

       $r' \$< b'; \#0 \$\leq r; \#0 \$< b'; b' \$\leq b]$ 
 $\implies q \$\leq q'$ 
apply (frule q-pos-lemma, assumption+)
apply (subgoal-tac  $b\$*q \$< b\$*(q' \$+ \#1)$ )
  apply (simp add: zmult-zless-cancel1)
  apply (force dest: zless-add1-iff-zle [THEN iffD1] zless-trans zless-zle-trans)
apply (subgoal-tac  $b\$*q = r' \$- r \$+ b' \$*q'$ )
  prefer 2 apply (simp add: zcompare-rls)
apply (simp (no-asm-simp) add: zadd-zmult-distrib2)
apply (subst zadd-commute [of  $b \$* q'$ ], rule zadd-zless-mono)
  prefer 2 apply (blast intro: zmult-zle-mono1)
apply (subgoal-tac  $r' \$+ \#0 \$< b \$+ r$ )
  apply (simp add: zcompare-rls)
apply (rule zadd-zless-mono)
  apply auto
apply (blast dest: zless-zle-trans)
done

```

lemma *zdiv-mono2-raw*:

```

       $\llbracket \#0 \$\leq a; \#0 \$< b'; b' \$\leq b; a \in \text{int} \rrbracket$ 
 $\implies a \text{ zdiv } b \$\leq a \text{ zdiv } b'$ 
apply (subgoal-tac  $\#0 \$< b$ )
  prefer 2 apply (blast dest: zless-zle-trans)
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)
apply (simp-all add: pos-mod-sign pos-mod-bound)
done

```

lemma *zdiv-mono2*:

```

       $\llbracket \#0 \$\leq a; \#0 \$< b'; b' \$\leq b \rrbracket$ 
 $\implies a \text{ zdiv } b \$\leq a \text{ zdiv } b'$ 
apply (cut-tac  $a = \text{intify } (a)$  in zdiv-mono2-raw)
apply auto
done

```

lemma *q-neg-lemma*:

```

       $\llbracket b' \$*q' \$+ r' \$< \#0; \#0 \$\leq r'; \#0 \$< b \rrbracket \implies q' \$< \#0$ 
apply (subgoal-tac  $b' \$*q' \$< \#0$ )
  prefer 2 apply (force intro: zle-zless-trans)
apply (simp add: zmult-less-0-iff)
apply (blast dest: zless-trans)
done

```

lemma *zdiv-mono2-neg-lemma*:

$$\llbracket b * q + r = b' * q' + r'; \quad b' * q' + r' < \#0; \\ r < b; \quad \#0 \leq r'; \quad \#0 < b'; \quad b' \leq b \rrbracket \\ \implies q' \leq q$$

apply (*subgoal-tac* $\#0 < b$)
prefer 2 **apply** (*blast dest: zless-zle-trans*)
apply (*frule q-neg-lemma, assumption+*)
apply (*subgoal-tac* $b * q' < b * (q + \#1)$)
apply (*simp add: zmult-zless-cancel1*)
apply (*blast dest: zless-trans zless-add1-iff-zle [THEN iffD1]*)
apply (*simp (no-asm-simp) add: zadd-zmult-distrib2*)
apply (*subgoal-tac* $b * q' \leq b' * q'$)
prefer 2
apply (*simp add: zmult-zle-cancel2*)
apply (*blast dest: zless-trans*)
apply (*subgoal-tac* $b' * q' + r < b * (q + r)$)
prefer 2
apply (*erule ssubst*)
apply *simp*
apply (*drule-tac* $w' = r$ **and** $z' = \#0$ **in** *zadd-zless-mono*)
apply (*assumption*)
apply *simp*
apply (*simp (no-asm-use) add: zadd-commute*)
apply (*rule zle-zless-trans*)
prefer 2 **apply** (*assumption*)
apply (*simp (no-asm-simp) add: zmult-zle-cancel2*)
apply (*blast dest: zless-trans*)
done

lemma *zdiv-mono2-neg-raw*:

$$\llbracket a < \#0; \quad \#0 < b'; \quad b' \leq b; \quad a \in \text{int} \rrbracket \\ \implies a \text{ zdiv } b' \leq a \text{ zdiv } b$$

apply (*subgoal-tac* $\#0 < b$)
prefer 2 **apply** (*blast dest: zless-zle-trans*)
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*)
apply (*simp-all add: pos-mod-sign pos-mod-bound*)
done

lemma *zdiv-mono2-neg*: $\llbracket a < \#0; \quad \#0 < b'; \quad b' \leq b \rrbracket \\ \implies a \text{ zdiv } b' \leq a \text{ zdiv } b$
apply (*cut-tac* $a = \text{intify } (a)$ **in** *zdiv-mono2-neg-raw*)
apply *auto*
done

33.10 More algebraic laws for zdiv and zmod

lemma *zmult1-lemma*:

$\llbracket \text{quorem}(\langle b, c \rangle, \langle q, r \rangle); \ c \in \text{int}; \ c \neq \#0 \rrbracket$

$\implies \text{quorem}(\langle a\$*b, c \rangle, \langle a\$*q \$+ (a\$*r) \text{ zdiv } c, (a\$*r) \text{ zmod } c \rangle)$

apply (*auto simp add: split-ifs quorem-def neq-iff-zless zadd-zmult-distrib2*
pos-mod-sign pos-mod-bound neg-mod-sign neg-mod-bound)

apply (*auto intro: raw-zmod-zdiv-equality*)

done

lemma *zdiv-zmult1-eq-raw*:

$\llbracket b \in \text{int}; \ c \in \text{int} \rrbracket$

$\implies (a\$*b) \text{ zdiv } c = a\$*(b \text{ zdiv } c) \$+ a\$*(b \text{ zmod } c) \text{ zdiv } c$

apply (*case-tac c = #0*)

apply (*simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)

apply (*rule quorem-div-mod [THEN zmult1-lemma, THEN quorem-div]*)

apply *auto*

done

lemma *zdiv-zmult1-eq*: $(a\$*b) \text{ zdiv } c = a\$*(b \text{ zdiv } c) \$+ a\$*(b \text{ zmod } c) \text{ zdiv } c$

apply (*cut-tac b = intify (b) and c = intify (c) in zdiv-zmult1-eq-raw*)

apply *auto*

done

lemma *zmod-zmult1-eq-raw*:

$\llbracket b \in \text{int}; \ c \in \text{int} \rrbracket \implies (a\$*b) \text{ zmod } c = a\$*(b \text{ zmod } c) \text{ zmod } c$

apply (*case-tac c = #0*)

apply (*simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)

apply (*rule quorem-div-mod [THEN zmult1-lemma, THEN quorem-mod]*)

apply *auto*

done

lemma *zmod-zmult1-eq*: $(a\$*b) \text{ zmod } c = a\$*(b \text{ zmod } c) \text{ zmod } c$

apply (*cut-tac b = intify (b) and c = intify (c) in zmod-zmult1-eq-raw*)

apply *auto*

done

lemma *zmod-zmult1-eq'*: $(a\$*b) \text{ zmod } c = ((a \text{ zmod } c) \$* b) \text{ zmod } c$

apply (*rule trans*)

apply (*rule-tac b = (b \\$* a) zmod c in trans*)

apply (*rule-tac [2] zmod-zmult1-eq*)

apply (*simp-all (no-asm) add: zmult-commute*)

done

lemma *zmod-zmult-distrib*: $(a\$*b) \text{ zmod } c = ((a \text{ zmod } c) \$* (b \text{ zmod } c)) \text{ zmod } c$

apply (*rule zmod-zmult1-eq' [THEN trans]*)

apply (*rule zmod-zmult1-eq*)

done

lemma *zdiv-zmult-self1* [*simp*]: $\text{intify}(b) \neq \#0 \implies (a\$*b) \text{ zdiv } b = \text{intify}(a)$

```

by (simp add: zdiv-zmult1-eq)

lemma zdiv-zmult-self2 [simp]: intify(b) ≠ #0 ⇒ (b$a) zdiv b = intify(a)
  by (simp add: zmult-commute)

lemma zmod-zmult-self1 [simp]: (a$b) zmod b = #0
  by (simp add: zmod-zmult1-eq)

lemma zmod-zmult-self2 [simp]: (b$a) zmod b = #0
  by (simp add: zmult-commute zmod-zmult1-eq)

lemma zadd1-lemma:
  [[quorem(⟨a,c⟩, ⟨aq,ar⟩); quorem(⟨b,c⟩, ⟨bq,br⟩);
   c ∈ int; c ≠ #0]]
  ⇒ quorem(⟨a$b+c, c⟩, ⟨aq$b+ bq$b+ (ar$b+br) zdiv c, (ar$b+br) zmod c⟩)
apply (auto simp add: split-ifs quorem-def neg-iff-zless zadd-zmult-distrib2
  pos-mod-sign pos-mod-bound neg-mod-sign neg-mod-bound)
apply (auto intro: raw-zmod-zdiv-equality)
done

lemma zdiv-zadd1-eq-raw:
  [[a ∈ int; b ∈ int; c ∈ int]] ⇒
  (a$b) zdiv c = a zdiv c $+ b zdiv c $+ ((a zmod c $+ b zmod c) zdiv c)
apply (case-tac c = #0)
  apply (simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD)
  apply (blast intro: zadd1-lemma [OF quorem-div-mod quorem-div-mod,
    THEN quorem-div])
done

lemma zdiv-zadd1-eq:
  (a$b) zdiv c = a zdiv c $+ b zdiv c $+ ((a zmod c $+ b zmod c) zdiv c)
apply (cut-tac a = intify(a) and b = intify(b) and c = intify(c)
  in zdiv-zadd1-eq-raw)
apply auto
done

lemma zmod-zadd1-eq-raw:
  [[a ∈ int; b ∈ int; c ∈ int]]
  ⇒ (a$b) zmod c = (a zmod c $+ b zmod c) zmod c
apply (case-tac c = #0)
  apply (simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD)
  apply (blast intro: zadd1-lemma [OF quorem-div-mod quorem-div-mod,
    THEN quorem-mod])
done

```

lemma *zmod-zadd1-eq*: $(a\$+b) \text{ zmod } c = (a \text{ zmod } c \$+ b \text{ zmod } c) \text{ zmod } c$
apply (*cut-tac* $a = \text{intify } (a)$ **and** $b = \text{intify } (b)$ **and** $c = \text{intify } (c)$)
 in *zmod-zadd1-eq-raw*)
apply *auto*
done

lemma *zmod-div-trivial-raw*:
 $\llbracket a \in \text{int}; b \in \text{int} \rrbracket \implies (a \text{ zmod } b) \text{ zdiv } b = \#0$
apply (*case-tac* $b = \#0$)
 apply (*simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*auto simp add: neg-iff-zless pos-mod-sign pos-mod-bound*
 zdiv-pos-pos-trivial neg-mod-sign neg-mod-bound zdiv-neg-neg-trivial)
done

lemma *zmod-div-trivial* [*simp*]: $(a \text{ zmod } b) \text{ zdiv } b = \#0$
apply (*cut-tac* $a = \text{intify } (a)$ **and** $b = \text{intify } (b)$ **in** *zmod-div-trivial-raw*)
apply *auto*
done

lemma *zmod-mod-trivial-raw*:
 $\llbracket a \in \text{int}; b \in \text{int} \rrbracket \implies (a \text{ zmod } b) \text{ zmod } b = a \text{ zmod } b$
apply (*case-tac* $b = \#0$)
 apply (*simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*auto simp add: neg-iff-zless pos-mod-sign pos-mod-bound*
 zmod-pos-pos-trivial neg-mod-sign neg-mod-bound zmod-neg-neg-trivial)
done

lemma *zmod-mod-trivial* [*simp*]: $(a \text{ zmod } b) \text{ zmod } b = a \text{ zmod } b$
apply (*cut-tac* $a = \text{intify } (a)$ **and** $b = \text{intify } (b)$ **in** *zmod-mod-trivial-raw*)
apply *auto*
done

lemma *zmod-zadd-left-eq*: $(a\$+b) \text{ zmod } c = ((a \text{ zmod } c) \$+ b) \text{ zmod } c$
apply (*rule trans [symmetric]*)
apply (*rule zmod-zadd1-eq*)
apply (*simp (no-asm)*)
apply (*rule zmod-zadd1-eq [symmetric]*)
done

lemma *zmod-zadd-right-eq*: $(a\$+b) \text{ zmod } c = (a \$+ (b \text{ zmod } c)) \text{ zmod } c$
apply (*rule trans [symmetric]*)
apply (*rule zmod-zadd1-eq*)
apply (*simp (no-asm)*)
apply (*rule zmod-zadd1-eq [symmetric]*)
done

lemma *zdiv-zadd-self1* [*simp*]:
 $\text{intify}(a) \neq \#0 \implies (a\$+b) \text{ zdiv } a = b \text{ zdiv } a \$+ \#1$

by (*simp* (*no-asm-simp*) *add*: *zdiv-zadd1-eq*)

lemma *zdiv-zadd-self2* [*simp*]:

intify(a) ≠ #0 ⇒ (b\$+a) zdiv a = b zdiv a \$+ #1

by (*simp* (*no-asm-simp*) *add*: *zdiv-zadd1-eq*)

lemma *zmod-zadd-self1* [*simp*]: (*a\$+b*) *zmod a* = *b zmod a*

apply (*case-tac a = #0*)

apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)

apply (*simp* (*no-asm-simp*) *add*: *zmod-zadd1-eq*)

done

lemma *zmod-zadd-self2* [*simp*]: (*b\$+a*) *zmod a* = *b zmod a*

apply (*case-tac a = #0*)

apply (*simp add*: *DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)

apply (*simp* (*no-asm-simp*) *add*: *zmod-zadd1-eq*)

done

33.11 proving $a \text{ zdiv } (b * c) = (a \text{ zdiv } b) \text{ zdiv } c$

lemma *zdiv-zmult2-aux1*:

$\llbracket \#0 \$< c; \ b \$< r; \ r \$\leq \#0 \rrbracket \implies b\$*c \$< b\$*(q \text{ zmod } c) \$+ r$

apply (*subgoal-tac b \$* (c \$- q zmod c) \$< r \$* #1*)

apply (*simp add*: *zdiff-zmult-distrib2 zadd-commute zcompare-rls*)

apply (*rule zle-zless-trans*)

apply (*erule-tac [2] zmult-zless-mono1*)

apply (*rule zmult-zle-mono2-neg*)

apply (*auto simp add*: *zcompare-rls zadd-commute add1-zle-iff pos-mod-bound*)

apply (*blast intro*: *zless-imp-zle dest*: *zless-zle-trans*)

done

lemma *zdiv-zmult2-aux2*:

$\llbracket \#0 \$< c; \ b \$< r; \ r \$\leq \#0 \rrbracket \implies b \$* (q \text{ zmod } c) \$+ r \$\leq \#0$

apply (*subgoal-tac b \$* (q zmod c) \$≤ #0*)

prefer 2

apply (*simp add*: *zmult-le-0-iff pos-mod-sign*)

apply (*blast intro*: *zless-imp-zle dest*: *zless-zle-trans*)

apply (*drule zadd-zle-mono*)

apply *assumption*

apply (*simp add*: *zadd-commute*)

done

lemma *zdiv-zmult2-aux3*:

$\llbracket \#0 \$< c; \ \#0 \$\leq r; \ r \$< b \rrbracket \implies \#0 \$\leq b \$* (q \text{ zmod } c) \$+ r$

apply (*subgoal-tac #0 \$≤ b \$* (q zmod c)*)

prefer 2

apply (*simp add*: *int-0-le-mult-iff pos-mod-sign*)

apply (*blast intro*: *zless-imp-zle dest*: *zle-zless-trans*)

```

apply (drule zadd-zle-mono)
apply assumption
apply (simp add: zadd-commute)
done

```

```

lemma zdiv-zmult2-aux4:
  
$$\llbracket \#0 \ \$< \ c; \ \#0 \ \$\leq \ r; \ r \ \$< \ b \rrbracket \implies b \ \$* \ (q \ \text{zmod} \ c) \ \$+ \ r \ \$< \ b \ \$* \ c$$

apply (subgoal-tac r  $\$* \ \#1 \ \$< \ b \ \$* \ (c \ \$- \ q \ \text{zmod} \ c)$ )
apply (simp add: zdiff-zmult-distrib2 zadd-commute zcompare-rls)
apply (rule zless-zle-trans)
apply (erule zmult-zless-mono1)
apply (rule-tac [2] zmult-zle-mono2)
apply (auto simp add: zcompare-rls zadd-commute add1-zle-iff pos-mod-bound)
apply (blast intro: zless-imp-zle dest: zle-zless-trans)
done

```

```

lemma zdiv-zmult2-lemma:
  
$$\llbracket \text{quorem} \ (\langle a, b \rangle, \langle q, r \rangle); \ a \in \text{int}; \ b \in \text{int}; \ b \neq \#0; \ \#0 \ \$< \ c \rrbracket$$


$$\implies \text{quorem} \ (\langle a, b \ \$* \ c \rangle, \langle q \ \text{zdiv} \ c, \ b \ \$* \ (q \ \text{zmod} \ c) \ \$+ \ r \rangle)$$

apply (auto simp add: zmult-ac zmod-zdiv-equality [symmetric] quorem-def
  neq-iff-zless int-0-less-mult-iff
  zadd-zmult-distrib2 [symmetric] zdiv-zmult2-aux1 zdiv-zmult2-aux2
  zdiv-zmult2-aux3 zdiv-zmult2-aux4)
apply (blast dest: zless-trans)
done

```

```

lemma zdiv-zmult2-eq-raw:
  
$$\llbracket \#0 \ \$< \ c; \ a \in \text{int}; \ b \in \text{int} \rrbracket \implies a \ \text{zdiv} \ (b \ \$* \ c) = (a \ \text{zdiv} \ b) \ \text{zdiv} \ c$$

apply (case-tac b =  $\#0$ )
apply (simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD)
apply (rule quorem-div-mod [THEN zdiv-zmult2-lemma, THEN quorem-div])
apply (auto simp add: intify-eq-0-iff-zle)
apply (blast dest: zle-zless-trans)
done

```

```

lemma zdiv-zmult2-eq:  $\#0 \ \$< \ c \implies a \ \text{zdiv} \ (b \ \$* \ c) = (a \ \text{zdiv} \ b) \ \text{zdiv} \ c$ 
apply (cut-tac a = intify (a) and b = intify (b) in zdiv-zmult2-eq-raw)
apply auto
done

```

```

lemma zmod-zmult2-eq-raw:
  
$$\llbracket \#0 \ \$< \ c; \ a \in \text{int}; \ b \in \text{int} \rrbracket$$


$$\implies a \ \text{zmod} \ (b \ \$* \ c) = b \ \$* \ (a \ \text{zdiv} \ b \ \text{zmod} \ c) \ \$+ \ a \ \text{zmod} \ b$$

apply (case-tac b =  $\#0$ )
apply (simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD)
apply (rule quorem-div-mod [THEN zdiv-zmult2-lemma, THEN quorem-mod])
apply (auto simp add: intify-eq-0-iff-zle)
apply (blast dest: zle-zless-trans)

```

done

lemma *zmod-zmult2-eq*:

$\#0 \leq c \implies a \text{ zmod } (b * c) = b * (a \text{ zdiv } b \text{ zmod } c) + a \text{ zmod } b$
apply (*cut-tac* $a = \text{intify } (a)$ **and** $b = \text{intify } (b)$ **in** *zmod-zmult2-eq-raw*)
apply *auto*
done

33.12 Cancellation of common factors in "zdiv"

lemma *zdiv-zmult-zmult1-aux1*:

$\llbracket \#0 \leq b; \text{intify}(c) \neq \#0 \rrbracket \implies (c * a) \text{ zdiv } (c * b) = a \text{ zdiv } b$
apply (*subst* *zdiv-zmult2-eq*)
apply *auto*
done

lemma *zdiv-zmult-zmult1-aux2*:

$\llbracket b \leq \#0; \text{intify}(c) \neq \#0 \rrbracket \implies (c * a) \text{ zdiv } (c * b) = a \text{ zdiv } b$
apply (*subgoal-tac* $(c * (\$ - a)) \text{ zdiv } (c * (\$ - b)) = (\$ - a) \text{ zdiv } (\$ - b)$)
apply (*rule-tac* [2] *zdiv-zmult-zmult1-aux1*)
apply *auto*
done

lemma *zdiv-zmult-zmult1-raw*:

$\llbracket \text{intify}(c) \neq \#0; b \in \text{int} \rrbracket \implies (c * a) \text{ zdiv } (c * b) = a \text{ zdiv } b$
apply (*case-tac* $b = \#0$)
apply (*simp* *add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD*)
apply (*auto* *simp* *add: neq-iff-zless* [of b])
zdiv-zmult-zmult1-aux1 *zdiv-zmult-zmult1-aux2*
done

lemma *zdiv-zmult-zmult1*: $\text{intify}(c) \neq \#0 \implies (c * a) \text{ zdiv } (c * b) = a \text{ zdiv } b$

apply (*cut-tac* $b = \text{intify } (b)$ **in** *zdiv-zmult-zmult1-raw*)
apply *auto*
done

lemma *zdiv-zmult-zmult2*: $\text{intify}(c) \neq \#0 \implies (a * c) \text{ zdiv } (b * c) = a \text{ zdiv } b$

apply (*drule* *zdiv-zmult-zmult1*)
apply (*auto* *simp* *add: zmult-commute*)
done

33.13 Distribution of factors over "zmod"

lemma *zmod-zmult-zmult1-aux1*:

$\llbracket \#0 \leq b; \text{intify}(c) \neq \#0 \rrbracket$
 $\implies (c * a) \text{ zmod } (c * b) = c * (a \text{ zmod } b)$
apply (*subst* *zmod-zmult2-eq*)
apply *auto*
done

```

lemma zmod-zmult-zmult1-aux2:
   $\llbracket b \text{ \$} < \#0; \text{intify}(c) \neq \#0 \rrbracket$ 
   $\implies (c\$*a) \text{ zmod } (c\$*b) = c \$* (a \text{ zmod } b)$ 
apply (subgoal-tac (c $* ($-a)) zmod (c $* ($-b)) = c $* (($-a) zmod ($-b)))
apply (rule-tac [2] zmod-zmult-zmult1-aux1)
apply auto
done

```

```

lemma zmod-zmult-zmult1-raw:
   $\llbracket b \in \text{int}; c \in \text{int} \rrbracket \implies (c\$*a) \text{ zmod } (c\$*b) = c \$* (a \text{ zmod } b)$ 
apply (case-tac b = #0)
apply (simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD)
apply (case-tac c = #0)
apply (simp add: DIVISION-BY-ZERO-ZDIV DIVISION-BY-ZERO-ZMOD)
apply (auto simp add: neg-iff-zless [of b]
  zmod-zmult-zmult1-aux1 zmod-zmult-zmult1-aux2)
done

```

```

lemma zmod-zmult-zmult1: (c$*a) zmod (c$*b) = c $* (a zmod b)
apply (cut-tac b = intify (b) and c = intify (c) in zmod-zmult-zmult1-raw)
apply auto
done

```

```

lemma zmod-zmult-zmult2: (a$*c) zmod (b$*c) = (a zmod b) $* c
apply (cut-tac c = c in zmod-zmult-zmult1)
apply (auto simp add: zmult-commute)
done

```

```

lemma zdiv-neg-pos-less0:  $\llbracket a \text{ \$} < \#0; \#0 \text{ \$} < b \rrbracket \implies a \text{ zdiv } b \text{ \$} < \#0$ 
apply (subgoal-tac a zdiv b $≤ #-1)
apply (erule zle-zless-trans)
apply (simp (no-asm))
apply (rule zle-trans)
apply (rule-tac a' = #-1 in zdiv-mono1)
apply (rule zless-add1-iff-zle [THEN iffD1])
apply (simp (no-asm))
apply (auto simp add: zdiv-minus1)
done

```

```

lemma zdiv-nonneg-neg-le0:  $\llbracket \#0 \text{ \$} \leq a; b \text{ \$} < \#0 \rrbracket \implies a \text{ zdiv } b \text{ \$} \leq \#0$ 
apply (drule zdiv-mono1-neg)
apply auto
done

```

```

lemma pos-imp-zdiv-nonneg-iff:  $\#0 \text{ \$} < b \implies (\#0 \text{ \$} \leq a \text{ zdiv } b) \longleftrightarrow (\#0 \text{ \$} \leq a)$ 
apply auto

```

```

apply (drule-tac [2] zdiv-mono1)
apply (auto simp add: neg-iff-zless)
apply (simp (no-asm-use) add: not-zless-iff-zle [THEN iff-sym])
apply (blast intro: zdiv-neg-pos-less0)
done

lemma neg-imp-zdiv-nonneg-iff:  $b \leq \#0 \implies (\#0 \leq a \text{ zdiv } b) \longleftrightarrow (a \leq \#0)$ 
apply (subst zdiv-zminus-zminus [symmetric])
apply (rule iff-trans)
apply (rule pos-imp-zdiv-nonneg-iff)
apply auto
done

lemma pos-imp-zdiv-neg-iff:  $\#0 < b \implies (a \text{ zdiv } b < \#0) \longleftrightarrow (a < \#0)$ 
apply (simp (no-asm-simp) add: not-zle-iff-zless [THEN iff-sym])
apply (erule pos-imp-zdiv-nonneg-iff)
done

lemma neg-imp-zdiv-neg-iff:  $b < \#0 \implies (a \text{ zdiv } b < \#0) \longleftrightarrow (\#0 < a)$ 
apply (simp (no-asm-simp) add: not-zle-iff-zless [THEN iff-sym])
apply (erule neg-imp-zdiv-nonneg-iff)
done

end

```

34 Cardinal Arithmetic Without the Axiom of Choice

theory *CardinalArith* **imports** *Cardinal OrderArith ArithSimp Finite* **begin**

definition

InfCard :: $i \Rightarrow o$ **where**
 $\text{InfCard}(i) \equiv \text{Card}(i) \wedge \text{nat} \leq i$

definition

cmult :: $[i, i] \Rightarrow i$ (**infixl** $\langle \otimes \rangle$ 70) **where**
 $i \otimes j \equiv |i * j|$

definition

cadd :: $[i, i] \Rightarrow i$ (**infixl** $\langle \oplus \rangle$ 65) **where**
 $i \oplus j \equiv |i + j|$

definition

csquare-rel :: $i \Rightarrow i$ **where**
 $\text{csquare-rel}(K) \equiv$
 $\text{rvimage}(K * K,$

$\text{lam } \langle x, y \rangle : K * K. < x \cup y, x, y \rangle,$
 $\text{rmult}(K, \text{Memrel}(K), K * K, \text{rmult}(K, \text{Memrel}(K), K, \text{Memrel}(K)))$

definition

$\text{jump-cardinal} :: i \Rightarrow i$ **where**

— This definition is more complex than Kunen's but it more easily proved to be a cardinal

$\text{jump-cardinal}(K) \equiv$
 $\bigcup X \in \text{Pow}(K). \{z. r \in \text{Pow}(K * K), \text{well-ord}(X, r) \wedge z = \text{ordertype}(X, r)\}$

definition

$\text{csucc} :: i \Rightarrow i$ **where**

— needed because $\text{jump-cardinal}(K)$ might not be the successor of K

$\text{csucc}(K) \equiv \mu L. \text{Card}(L) \wedge K < L$

lemma *Card-Union* [*simp,intro,TC*]:

assumes $A: \bigwedge x. x \in A \implies \text{Card}(x)$ **shows** $\text{Card}(\bigcup(A))$

proof (*rule CardI*)

show $\text{Ord}(\bigcup A)$ **using** A

by (*simp add: Card-is-Ord*)

next

fix j

assume $j: j < \bigcup A$

hence $\exists c \in A. j < c \wedge \text{Card}(c)$ **using** A

by (*auto simp add: lt-def intro: Card-is-Ord*)

then obtain c **where** $c: c \in A \ j < c \ \text{Card}(c)$

by *blast*

hence $jls: j \prec c$

by (*simp add: lt-Card-imp-lesspoll*)

{ assume $\text{eqp}: j \approx \bigcup A$

have $c \lesssim \bigcup A$ **using** c

by (*blast intro: subset-imp-lepoll*)

also have $\dots \approx j$ **by** (*rule eqpoll-sym [OF eqp]*)

also have $\dots \prec c$ **by** (*rule jls*)

finally have $c \prec c$.

hence *False*

by *auto*

} thus $\neg j \approx \bigcup A$ **by** *blast*

qed

lemma *Card-UN*: $(\bigwedge x. x \in A \implies \text{Card}(K(x))) \implies \text{Card}(\bigcup_{x \in A} K(x))$

by *blast*

lemma *Card-OUN* [*simp,intro,TC*]:

$(\bigwedge x. x \in A \implies \text{Card}(K(x))) \implies \text{Card}(\bigcup_{x < A} K(x))$

by (*auto simp add: OUnion-def Card-0*)

lemma *in-Card-imp-lesspoll*: $\llbracket \text{Card}(K); b \in K \rrbracket \implies b \prec K$

```

  unfolding lesspoll-def
apply (simp add: Card-iff-initial)
apply (fast intro!: le-imp-lepoll ltI leI)
done

```

34.1 Cardinal addition

Note: Could omit proving the algebraic laws for cardinal addition and multiplication. On finite cardinals these operations coincide with addition and multiplication of natural numbers; on infinite cardinals they coincide with union (maximum). Either way we get most laws for free.

34.1.1 Cardinal addition is commutative

```

lemma sum-commute-epoll:  $A+B \approx B+A$ 
proof (unfold epoll-def, rule exI)
  show  $(\lambda z \in A+B. \text{case}(Inr, Inl, z)) \in \text{bij}(A+B, B+A)$ 
  by (auto intro: lam-bijective [where d = case(Inr, Inl)])
qed

```

```

lemma cadd-commute:  $i \oplus j = j \oplus i$ 
  unfolding cadd-def
apply (rule sum-commute-epoll [THEN cardinal-cong])
done

```

34.1.2 Cardinal addition is associative

```

lemma sum-assoc-epoll:  $(A+B)+C \approx A+(B+C)$ 
  unfolding epoll-def
apply (rule exI)
apply (rule sum-assoc-bij)
done

```

Unconditional version requires AC

```

lemma well-ord-cadd-assoc:
  assumes  $i: \text{well-ord}(i, ri)$  and  $j: \text{well-ord}(j, rj)$  and  $k: \text{well-ord}(k, rk)$ 
  shows  $(i \oplus j) \oplus k = i \oplus (j \oplus k)$ 
proof (unfold cadd-def, rule cardinal-cong)
  have  $|i + j| + k \approx (i + j) + k$ 
  by (blast intro: sum-epoll-cong well-ord-cardinal-epoll epoll-refl well-ord-radd
    i j)
  also have  $\dots \approx i + (j + k)$ 
  by (rule sum-assoc-epoll)
  also have  $\dots \approx i + |j + k|$ 
  by (blast intro: sum-epoll-cong well-ord-cardinal-epoll epoll-refl well-ord-radd
    j k epoll-sym)
  finally show  $|i + j| + k \approx i + |j + k|$  .
qed

```

34.1.3 0 is the identity for addition

lemma *sum-0-epoll*: $0 + A \approx A$
unfolding *epoll-def*
apply (*rule exI*)
apply (*rule bij-0-sum*)
done

lemma *cadd-0* [*simp*]: $\text{Card}(K) \implies 0 \oplus K = K$
unfolding *cadd-def*
apply (*simp add: sum-0-epoll [THEN cardinal-cong] Card-cardinal-eq*)
done

34.1.4 Addition by another cardinal

lemma *sum-lepoll-self*: $A \lesssim A + B$
proof (*unfold lepoll-def, rule exI*)
show $(\lambda x \in A. \text{Inl } (x)) \in \text{inj}(A, A + B)$
by (*simp add: inj-def*)
qed

lemma *cadd-le-self*:
assumes $K: \text{Card}(K)$ **and** $L: \text{Ord}(L)$ **shows** $K \leq (K \oplus L)$
proof (*unfold cadd-def*)
have $K \leq |K|$
by (*rule Card-cardinal-le [OF K]*)
moreover have $|K| \leq |K + L|$ **using** $K L$
by (*blast intro: well-ord-lepoll-imp-cardinal-le sum-lepoll-self well-ord-radd well-ord-Memrel Card-is-Ord*)
ultimately show $K \leq |K + L|$
by (*blast intro: le-trans*)
qed

34.1.5 Monotonicity of addition

lemma *sum-lepoll-mono*:
 $\llbracket A \lesssim C; B \lesssim D \rrbracket \implies A + B \lesssim C + D$
unfolding *lepoll-def*
apply (*elim exE*)
apply (*rule-tac x = \lambda z \in A + B. case (\lambda w. Inl(f'w), \lambda y. Inr(fa'y), z) in exI*)
apply (*rule-tac d = case (\lambda w. Inl(converse(f) 'w), \lambda y. Inr(converse(fa) 'y))*
in *lam-injective*)
apply (*typecheck add: inj-is-fun, auto*)
done

lemma *cadd-le-mono*:
 $\llbracket K' \leq K; L' \leq L \rrbracket \implies (K' \oplus L') \leq (K \oplus L)$
unfolding *cadd-def*

```

apply (safe dest!: le-subset-iff [THEN iffD1])
apply (rule well-ord-lepoll-imp-cardinal-le)
apply (blast intro: well-ord-radd well-ord-Memrel)
apply (blast intro: sum-lepoll-mono subset-imp-lepoll)
done

```

34.1.6 Addition of finite cardinals is "ordinary" addition

```

lemma sum-succ-epoll:  $\text{succ}(A)+B \approx \text{succ}(A+B)$ 
  unfolding epoll-def
apply (rule exI)
apply (rule-tac c =  $\lambda z.$  if  $z=\text{Inl } (A)$  then  $A+B$  else  $z$ 
  and  $d = \lambda z.$  if  $z=A+B$  then  $\text{Inl } (A)$  else  $z$  in lam-bijective)
  apply simp-all
apply (blast dest: sym [THEN eq-imp-not-mem] elim: mem-irrefl)+
done

```

```

lemma cadd-succ-lemma:
  assumes  $\text{Ord}(m)$   $\text{Ord}(n)$  shows  $\text{succ}(m) \oplus n = |\text{succ}(m \oplus n)|$ 
proof (unfold cadd-def)
  have [intro]:  $m + n \approx |m + n|$  using assms
  by (blast intro: epoll-sym well-ord-cardinal-epoll well-ord-radd well-ord-Memrel)

  have  $|\text{succ}(m) + n| = |\text{succ}(m + n)|$ 
  by (rule sum-succ-epoll [THEN cardinal-cong])
  also have  $\dots = |\text{succ}(|m + n|)|$ 
  by (blast intro: succ-epoll-cong cardinal-cong)
  finally show  $|\text{succ}(m) + n| = |\text{succ}(|m + n|)|$  .
qed

```

```

lemma nat-cadd-eq-add:
  assumes  $m: m \in \text{nat}$  and [simp]:  $n \in \text{nat}$  shows  $m \oplus n = m \# + n$ 
using m
proof (induct m)
  case 0 thus ?case by (simp add: nat-into-Card cadd-0)
next
  case (succ m) thus ?case by (simp add: cadd-succ-lemma nat-into-Card Card-cardinal-eq)
qed

```

34.2 Cardinal multiplication

34.2.1 Cardinal multiplication is commutative

```

lemma prod-commute-epoll:  $A*B \approx B*A$ 
  unfolding epoll-def
apply (rule exI)
apply (rule-tac c =  $\lambda \langle x,y \rangle. \langle y,x \rangle$  and  $d = \lambda \langle x,y \rangle. \langle y,x \rangle$  in lam-bijective,
  auto)

```

done

lemma *cmult-commute*: $i \otimes j = j \otimes i$
unfolding *cmult-def*
apply (*rule prod-commute-epoll* [*THEN cardinal-cong*])
done

34.2.2 Cardinal multiplication is associative

lemma *prod-assoc-epoll*: $(A*B)*C \approx A*(B*C)$
unfolding *epoll-def*
apply (*rule exI*)
apply (*rule prod-assoc-bij*)
done

Unconditional version requires AC

lemma *well-ord-cmult-assoc*:
assumes *i*: *well-ord*(*i*,*ri*) **and** *j*: *well-ord*(*j*,*rj*) **and** *k*: *well-ord*(*k*,*rk*)
shows $(i \otimes j) \otimes k = i \otimes (j \otimes k)$
proof (*unfold cmult-def*, *rule cardinal-cong*)
have $|i * j| * k \approx (i * j) * k$
by (*blast intro: prod-epoll-cong well-ord-cardinal-epoll epoll-refl well-ord-rmult*
i j)
also have $\dots \approx i * (j * k)$
by (*rule prod-assoc-epoll*)
also have $\dots \approx i * |j * k|$
by (*blast intro: prod-epoll-cong well-ord-cardinal-epoll epoll-refl well-ord-rmult*
j k epoll-sym)
finally show $|i * j| * k \approx i * |j * k|$.
qed

34.2.3 Cardinal multiplication distributes over addition

lemma *sum-prod-distrib-epoll*: $(A+B)*C \approx (A*C)+(B*C)$
unfolding *epoll-def*
apply (*rule exI*)
apply (*rule sum-prod-distrib-bij*)
done

lemma *well-ord-cadd-cmult-distrib*:
assumes *i*: *well-ord*(*i*,*ri*) **and** *j*: *well-ord*(*j*,*rj*) **and** *k*: *well-ord*(*k*,*rk*)
shows $(i \oplus j) \otimes k = (i \otimes k) \oplus (j \otimes k)$
proof (*unfold cadd-def cmult-def*, *rule cardinal-cong*)
have $|i + j| * k \approx (i + j) * k$
by (*blast intro: prod-epoll-cong well-ord-cardinal-epoll epoll-refl well-ord-radd*
i j)
also have $\dots \approx i * k + j * k$
by (*rule sum-prod-distrib-epoll*)
also have $\dots \approx |i * k| + |j * k|$

by (blast intro: sum-epoll-cong well-ord-cardinal-epoll well-ord-rmult i j k
 epoll-sym)
 finally show $|i + j| * k \approx |i * k| + |j * k|$.
 qed

34.2.4 Multiplication by 0 yields 0

lemma prod-0-epoll: $0 * A \approx 0$
 unfolding epoll-def
 apply (rule exI)
 apply (rule lam-bijective, safe)
 done

lemma cmult-0 [simp]: $0 \otimes i = 0$
 by (simp add: cmult-def prod-0-epoll [THEN cardinal-cong])

34.2.5 1 is the identity for multiplication

lemma prod-singleton-epoll: $\{x\} * A \approx A$
 unfolding epoll-def
 apply (rule exI)
 apply (rule singleton-prod-bij [THEN bij-converse-bij])
 done

lemma cmult-1 [simp]: $\text{Card}(K) \implies 1 \otimes K = K$
 unfolding cmult-def succ-def
 apply (simp add: prod-singleton-epoll [THEN cardinal-cong] Card-cardinal-eq)
 done

34.3 Some inequalities for multiplication

lemma prod-square-lepoll: $A \lesssim A * A$
 unfolding lepoll-def inj-def
 apply (rule-tac $x = \lambda x \in A. \langle x, x \rangle$ in exI, simp)
 done

lemma cmult-square-le: $\text{Card}(K) \implies K \leq K \otimes K$
 unfolding cmult-def
 apply (rule le-trans)
 apply (rule-tac [2] well-ord-lepoll-imp-cardinal-le)
 apply (rule-tac [3] prod-square-lepoll)
 apply (simp add: le-refl Card-is-Ord Card-cardinal-eq)
 apply (blast intro: well-ord-rmult well-ord-Memrel Card-is-Ord)
 done

34.3.1 Multiplication by a non-zero cardinal

lemma prod-lepoll-self: $b \in B \implies A \lesssim A * B$
 unfolding lepoll-def inj-def

```

apply (rule-tac  $x = \lambda x \in A. \langle x, b \rangle$  in  $exI$ ,  $simp$ )
done

```

```

lemma cmult-le-self:
   $\llbracket Card(K); Ord(L); 0 < L \rrbracket \implies K \leq (K \otimes L)$ 
  unfolding cmult-def
apply (rule le-trans [OF Card-cardinal-le well-ord-lepoll-imp-cardinal-le])
  apply assumption
  apply (blast intro: well-ord-rmult well-ord-Memrel Card-is-Ord)
apply (blast intro: prod-lepoll-self ltD)
done

```

34.3.2 Monotonicity of multiplication

```

lemma prod-lepoll-mono:
   $\llbracket A \lesssim C; B \lesssim D \rrbracket \implies A * B \lesssim C * D$ 
  unfolding lepoll-def
apply (elim exE)
apply (rule-tac  $x = \text{lam } \langle w, y \rangle : A * B. <f'w, fa'y>$  in  $exI$ )
apply (rule-tac  $d = \lambda \langle w, y \rangle. <\text{converse } f \rangle 'w, \text{converse } (fa) 'y>$ 
  in lam-injective)
apply (typecheck add: inj-is-fun, auto)
done

```

```

lemma cmult-le-mono:
   $\llbracket K' \leq K; L' \leq L \rrbracket \implies (K' \otimes L') \leq (K \otimes L)$ 
  unfolding cmult-def
apply (safe dest!: le-subset-iff [THEN iffD1])
apply (rule well-ord-lepoll-imp-cardinal-le)
  apply (blast intro: well-ord-rmult well-ord-Memrel)
apply (blast intro: prod-lepoll-mono subset-imp-lepoll)
done

```

34.4 Multiplication of finite cardinals is "ordinary" multiplication

```

lemma prod-succ-epoll:  $\text{succ}(A) * B \approx B + A * B$ 
  unfolding epoll-def
apply (rule  $exI$ )
apply (rule-tac  $c = \lambda \langle x, y \rangle. \text{if } x=A \text{ then } Inl(y) \text{ else } Inr(\langle x, y \rangle)$ 
  and  $d = \text{case } (\lambda y. \langle A, y \rangle, \lambda z. z)$  in lam-bijective)
apply safe
apply (simp-all add: succI2 if-type mem-imp-not-eq)
done

```

```

lemma cmult-succ-lemma:
   $\llbracket Ord(m); Ord(n) \rrbracket \implies \text{succ}(m) \otimes n = n \oplus (m \otimes n)$ 
  unfolding cmult-def cadd-def

```

```

apply (rule prod-succ-epoll [THEN cardinal-cong, THEN trans])
apply (rule cardinal-cong [symmetric])
apply (rule sum-epoll-cong [OF epoll-refl well-ord-cardinal-epoll])
apply (blast intro: well-ord-rmult well-ord-Memrel)
done

```

```

lemma nat-cmult-eg-mult:  $\llbracket m \in \text{nat}; n \in \text{nat} \rrbracket \implies m \otimes n = m \# n$ 
apply (induct-tac m)
apply (simp-all add: cmult-succ-lemma nat-cadd-eg-add)
done

```

```

lemma cmult-2:  $\text{Card}(n) \implies 2 \otimes n = n \oplus n$ 
by (simp add: cmult-succ-lemma Card-is-Ord cadd-commute [of - 0])

```

```

lemma sum-lepoll-prod:
  assumes C:  $2 \lesssim C$  shows  $B+B \lesssim C*B$ 
proof -
  have  $B+B \lesssim 2*B$ 
    by (simp add: sum-eg-2-times)
  also have  $\dots \lesssim C*B$ 
    by (blast intro: prod-lepoll-mono lepoll-refl C)
  finally show  $B+B \lesssim C*B$  .
qed

```

```

lemma lepoll-imp-sum-lepoll-prod:  $\llbracket A \lesssim B; 2 \lesssim A \rrbracket \implies A+B \lesssim A*B$ 
by (blast intro: sum-lepoll-mono sum-lepoll-prod lepoll-trans lepoll-refl)

```

34.5 Infinite Cardinals are Limit Ordinals

```

lemma nat-cons-lepoll:  $\text{nat} \lesssim A \implies \text{cons}(u, A) \lesssim A$ 
  unfolding lepoll-def
apply (erule exE)
apply (rule-tac x =
   $\lambda z \in \text{cons}(u, A).$ 
     $\text{if } z=u \text{ then } f'0$ 
     $\text{else if } z \in \text{range}(f) \text{ then } f' \text{succ}(\text{converse}(f) \text{' } z) \text{ else } z$ 
  in exI)
apply (rule-tac d =
   $\lambda y. \text{if } y \in \text{range}(f) \text{ then } \text{nat-case}(u, \lambda z. f'z, \text{converse}(f) \text{' } y)$ 
     $\text{else } y$ 
  in lam-injective)
apply (fast intro!: if-type apply-type intro: inj-is-fun inj-converse-fun)
apply (simp add: inj-is-fun [THEN apply-rangeI]
   $\text{inj-converse-fun [THEN apply-rangeI]}$ 
   $\text{inj-converse-fun [THEN apply-funtype]}$ )
done

```

```

lemma nat-cons-epoll:  $\text{nat} \lesssim A \implies \text{cons}(u, A) \approx A$ 
apply (erule nat-cons-lepoll [THEN epollI])

```


apply (*rule subset-consI* [*THEN subset-imp-lepoll*])
done

lemma *nat-succ-epoll*: $\text{nat} \subseteq A \implies \text{succ}(A) \approx A$
unfolding *succ-def*
apply (*erule subset-imp-lepoll* [*THEN nat-cons-epoll*])
done

lemma *InfCard-nat*: *InfCard*(*nat*)
unfolding *InfCard-def*
apply (*blast intro: Card-nat le-refl Card-is-Ord*)
done

lemma *InfCard-is-Card*: *InfCard*(*K*) \implies *Card*(*K*)
unfolding *InfCard-def*
apply (*erule conjunct1*)
done

lemma *InfCard-Un*:
 $\llbracket \text{InfCard}(K); \text{Card}(L) \rrbracket \implies \text{InfCard}(K \cup L)$
unfolding *InfCard-def*
apply (*simp add: Card-Un Un-upper1-le* [*THEN* [*2*] *le-trans*] *Card-is-Ord*)
done

lemma *InfCard-is-Limit*: *InfCard*(*K*) \implies *Limit*(*K*)
unfolding *InfCard-def*
apply (*erule conjE*)
apply (*frule Card-is-Ord*)
apply (*rule ltI* [*THEN non-succ-LimitI*])
apply (*erule le-imp-subset* [*THEN subsetD*])
apply (*safe dest!*: *Limit-nat* [*THEN Limit-le-succD*])
unfolding *Card-def*
apply (*drule trans*)
apply (*erule le-imp-subset* [*THEN nat-succ-epoll*, *THEN cardinal-cong*])
apply (*erule Ord-cardinal-le* [*THEN lt-trans2*, *THEN lt-irrefl*])
apply (*rule le-eqI*, *assumption*)
apply (*rule Ord-cardinal*)
done

lemma *ordermap-epoll-pred*:
 $\llbracket \text{well-ord}(A, r); x \in A \rrbracket \implies \text{ordermap}(A, r) 'x \approx \text{Order.pred}(A, x, r)$
unfolding *epoll-def*
apply (*rule exI*)

```

apply (simp add: ordermap-eq-image well-ord-is-wf)
apply (erule ordermap-bij [THEN bij-is-inj, THEN restrict-bij,
                           THEN bij-converse-bij])
apply (rule pred-subset)
done

```

34.5.1 Establishing the well-ordering

```

lemma well-ord-csquare:
  assumes K: Ord(K) shows well-ord(K*K, csquare-rel(K))
proof (unfold csquare-rel-def, rule well-ord-rvimage)
  show  $\langle \lambda \langle x, y \rangle \in K \times K. \langle x \cup y, x, y \rangle \rangle \in \text{inj}(K \times K, K \times K \times K)$  using K
  by (force simp add: inj-def intro: lam-type Un-least-lt [THEN ltD] ltI)
next
  show well-ord( $K \times K \times K$ ,  $\text{rmult}(K, \text{Memrel}(K), K \times K, \text{rmult}(K, \text{Memrel}(K),$ 
   $K, \text{Memrel}(K)))$ )
  using K by (blast intro: well-ord-rmult well-ord-Memrel)
qed

```

34.5.2 Characterising initial segments of the well-ordering

```

lemma csquareD:
   $\llbracket \langle \langle x, y \rangle, \langle z, z \rangle \rangle \in \text{csquare-rel}(K); x < K; y < K; z < K \rrbracket \implies x \leq z \wedge y \leq z$ 
  unfolding csquare-rel-def
apply (erule rev-mp)
apply (elim ltE)
apply (simp add: rvimage-iff Un-absorb Un-least-mem-iff ltD)
apply (safe elim!: mem-irrefl intro!: Un-upper1-le Un-upper2-le)
apply (simp-all add: lt-def succI2)
done

```

```

lemma pred-csquare-subset:
   $z < K \implies \text{Order.pred}(K*K, \langle z, z \rangle, \text{csquare-rel}(K)) \subseteq \text{succ}(z)*\text{succ}(z)$ 
  unfolding Order.pred-def
apply (safe del: SigmaI dest!: csquareD)
apply (unfold lt-def, auto)
done

```

```

lemma csquare-ltI:
   $\llbracket x < z; y < z; z < K \rrbracket \implies \langle \langle x, y \rangle, \langle z, z \rangle \rangle \in \text{csquare-rel}(K)$ 
  unfolding csquare-rel-def
apply (subgoal-tac  $x < K \wedge y < K$ )
  prefer 2 apply (blast intro: lt-trans)
apply (elim ltE)
apply (simp add: rvimage-iff Un-absorb Un-least-mem-iff ltD)
done

```

```

lemma csquare-or-eqI:
   $\llbracket x \leq z; y \leq z; z < K \rrbracket \implies \langle \langle x, y \rangle, \langle z, z \rangle \rangle \in \text{csquare-rel}(K) \mid x = z \wedge y = z$ 

```

```

unfolding csquare-rel-def
apply (subgoal-tac  $x < K \wedge y < K$ )
prefer 2 apply (blast intro: lt-trans1)
apply (elim ltE)
apply (simp add: rvimage-iff Un-absorb Un-least-mem-iff ltD)
apply (elim succE)
apply (simp-all add: subset-Un-iff [THEN iff-sym]
          subset-Un-iff2 [THEN iff-sym] OrdmemD)
done

```

34.5.3 The cardinality of initial segments

```

lemma ordermap-z-lt:
   $\llbracket \text{Limit}(K); x < K; y < K; z = \text{succ}(x \cup y) \rrbracket \implies$ 
    ordermap( $K * K$ , csquare-rel( $K$ )) ' $\langle x, y \rangle <$ 
    ordermap( $K * K$ , csquare-rel( $K$ )) ' $\langle z, z \rangle$ 
apply (subgoal-tac  $z < K \wedge \text{well-ord}(K * K, \text{csquare-rel}(K))$ )
prefer 2 apply (blast intro!: Un-least-lt Limit-has-succ
          Limit-is-Ord [THEN well-ord-csquare], clarify)
apply (rule csquare-ltI [THEN ordermap-mono, THEN ltI])
apply (erule-tac [4] well-ord-is-wf)
apply (blast intro!: Un-upper1-le Un-upper2-le Ord-ordermap elim!: ltE)+
done

```

Kunen: "each $\langle x, y \rangle \in K \times K$ has no more than $z \times z$ predecessors..." (page 29)

```

lemma ordermap-csquare-le:
  assumes  $K: \text{Limit}(K)$  and  $x: x < K$  and  $y: y < K$ 
  defines  $z \equiv \text{succ}(x \cup y)$ 
  shows  $|\text{ordermap}(K \times K, \text{csquare-rel}(K)) \text{ ' $\langle x, y \rangle$ ' }| \leq |\text{succ}(z)| \otimes |\text{succ}(z)|$ 
proof (unfold cmult-def, rule well-ord-lepoll-imp-cardinal-le)
  show  $\text{well-ord}(|\text{succ}(z)| \times |\text{succ}(z)|,$ 
     $\text{rmult}(|\text{succ}(z)|, \text{Memrel}(|\text{succ}(z)|), |\text{succ}(z)|, \text{Memrel}(|\text{succ}(z)|)))$ 
    by (blast intro: Ord-cardinal well-ord-Memrel well-ord-rmult)
next
  have  $zK: z < K$  using  $x \ y \ K \ z\text{-def}$ 
  by (blast intro: Un-least-lt Limit-has-succ)
  hence  $oz: \text{Ord}(z)$  by (elim ltE)
  have  $\text{ordermap}(K \times K, \text{csquare-rel}(K)) \text{ ' $\langle x, y \rangle$ ' } \lesssim \text{ordermap}(K \times K, \text{csquare-rel}(K))$ 
     $\text{ ' $\langle z, z \rangle$ ' }$ 
    using  $z\text{-def}$ 
  by (blast intro: ordermap-z-lt leI le-imp-lepoll  $K \ x \ y$ )
  also have  $\dots \approx \text{Order.pred}(K \times K, \langle z, z \rangle, \text{csquare-rel}(K))$ 
  proof (rule ordermap-epoll-pred)
  show  $\text{well-ord}(K \times K, \text{csquare-rel}(K))$  using  $K$ 
  by (rule Limit-is-Ord [THEN well-ord-csquare])
next
  show  $\langle z, z \rangle \in K \times K$  using  $zK$ 
  by (blast intro: ltD)

```

qed
 also have ... $\lesssim \text{succ}(z) \times \text{succ}(z)$ using zK
 by (rule pred-csquare-subset [THEN subset-imp-lepoll])
 also have ... $\approx |\text{succ}(z)| \times |\text{succ}(z)|$ using oz
 by (blast intro: prod-epoll-cong Ord-succ Ord-cardinal-epoll eqpoll-sym)
 finally show $\text{ordermap}(K \times K, \text{csquare-rel}(K)) \text{ ‘ } \langle x, y \rangle \lesssim |\text{succ}(z)| \times |\text{succ}(z)| \text{ .}$
 qed

Kunen: "... so the order type is $\leq K$ "

lemma ordertype-csquare-le:

assumes $IK: \text{InfCard}(K)$ and $eq: \bigwedge y. y \in K \implies \text{InfCard}(y) \implies y \otimes y = y$

shows $\text{ordertype}(K * K, \text{csquare-rel}(K)) \leq K$

proof –

have $CK: \text{Card}(K)$ using IK by (rule InfCard-is-Card)

hence $OK: \text{Ord}(K)$ by (rule Card-is-Ord)

moreover have $\text{Ord}(\text{ordertype}(K \times K, \text{csquare-rel}(K)))$ using OK

by (rule well-ord-csquare [THEN Ord-ordertype])

ultimately show ?thesis

proof (rule all-lt-imp-le)

fix i

assume $i: i < \text{ordertype}(K \times K, \text{csquare-rel}(K))$

hence $Oi: \text{Ord}(i)$ by (elim ltE)

obtain $x\ y$ where $x: x \in K$ and $y: y \in K$

and $ieq: i = \text{ordermap}(K \times K, \text{csquare-rel}(K)) \text{ ‘ } \langle x, y \rangle$

using i by (auto simp add: ordertype-unfold elim: ltE)

hence $xy: \text{Ord}(x) \text{ Ord}(y) \ x < K \ y < K$ using OK

by (blast intro: Ord-in-Ord ltI)+

hence $ou: \text{Ord}(x \cup y)$

by (simp add: Ord-Un)

show $i < K$

proof (rule Card-lt-imp-lt [OF - Oi CK])

have $|i| \leq |\text{succ}(\text{succ}(x \cup y))| \otimes |\text{succ}(\text{succ}(x \cup y))|$ using $IK\ xy$

by (auto simp add: ieq intro: InfCard-is-Limit [THEN ordermap-csquare-le])

moreover have $|\text{succ}(\text{succ}(x \cup y))| \otimes |\text{succ}(\text{succ}(x \cup y))| < K$

proof (cases rule: Ord-linear2 [OF ou Ord-nat])

assume $x \cup y < \text{nat}$

hence $|\text{succ}(\text{succ}(x \cup y))| \otimes |\text{succ}(\text{succ}(x \cup y))| \in \text{nat}$

by (simp add: lt-def nat-cmult-eq-mult nat-succI mult-type
nat-into-Card [THEN Card-cardinal-eq] Ord-nat)

also have ... $\subseteq K$ using IK

by (simp add: InfCard-def le-imp-subset)

finally show $|\text{succ}(\text{succ}(x \cup y))| \otimes |\text{succ}(\text{succ}(x \cup y))| < K$

by (simp add: ltI OK)

next

assume $\text{nat}xy: \text{nat} \leq x \cup y$

hence $seg: |\text{succ}(\text{succ}(x \cup y))| = |x \cup y|$ using xy

by (simp add: le-imp-subset nat-succ-epoll [THEN cardinal-cong])

le-succ-iff)

also have ... $< K$ using xy

```

      by (simp add: Un-least-lt Ord-cardinal-le [THEN lt-trans1])
    finally have  $|succ(succ(x \cup y))| < K$  .
    moreover have  $InfCard(|succ(succ(x \cup y))|)$  using  $xy$  natxy
      by (simp add: seq InfCard-def Card-cardinal nat-le-cardinal)
    ultimately show ?thesis by (simp add: eq ltD)
  qed
  ultimately show  $|i| < K$  by (blast intro: lt-trans1)
qed
qed
qed

```

lemma *InfCard-csquare-eq*:

```

  assumes  $IK: InfCard(K)$  shows  $K \otimes K = K$ 
proof -
  have  $OK: Ord(K)$  using  $IK$  by (simp add: Card-is-Ord InfCard-is-Card)
  show  $K \otimes K = K$  using  $OK$   $IK$ 
  proof (induct rule: trans-induct)
    case (step i)
    show  $i \otimes i = i$ 
    proof (rule le-anti-sym)
      have  $|i \times i| = |ordertype(i \times i, csquare-rel(i))|$ 
      by (rule cardinal-cong,
        simp add: step.hyps well-ord-csquare [THEN ordermap-bij, THEN bij-imp-epoll])
      hence  $i \otimes i \leq ordertype(i \times i, csquare-rel(i))$ 
      by (simp add: step.hyps cmult-def Ord-cardinal-le well-ord-csquare [THEN
        Ord-ordertype])
      moreover
      have  $ordertype(i \times i, csquare-rel(i)) \leq i$  using step
      by (simp add: ordertype-csquare-le)
      ultimately show  $i \otimes i \leq i$  by (rule le-trans)
    next
      show  $i \leq i \otimes i$  using step
      by (blast intro: cmult-square-le InfCard-is-Card)
    qed
  qed
qed

```

lemma *well-ord-InfCard-square-eq*:

```

  assumes  $r: well\_ord(A, r)$  and  $I: InfCard(|A|)$  shows  $A \times A \approx A$ 
proof -
  have  $A \times A \approx |A| \times |A|$ 
  by (blast intro: prod-epoll-cong well-ord-cardinal-epoll eqpoll-sym r)
  also have  $\dots \approx A$ 
  proof (rule well-ord-cardinal-eqE [OF - r])
    show  $well\_ord(|A| \times |A|, rmult(|A|, Memrel(|A|), |A|, Memrel(|A|)))$ 
    by (blast intro: Ord-cardinal well-ord-rmult well-ord-Memrel r)
  next

```

```

    show  $||A| \times |A| = |A|$  using InfCard-csquare-eq I
    by (simp add: cmult-def)
  qed
  finally show ?thesis .
qed

```

```

lemma InfCard-square-eqpoll:  $\text{InfCard}(K) \implies K \times K \approx K$ 
apply (rule well-ord-InfCard-square-eq)
  apply (erule InfCard-is-Card [THEN Card-is-Ord, THEN well-ord-Memrel])
  apply (simp add: InfCard-is-Card [THEN Card-cardinal-eq])
done

```

```

lemma Inf-Card-is-InfCard:  $\llbracket \text{Card}(i); \neg \text{Finite}(i) \rrbracket \implies \text{InfCard}(i)$ 
by (simp add: InfCard-def Card-is-Ord [THEN nat-le-infinite-Ord])

```

34.5.4 Toward's Kunen's Corollary 10.13 (1)

```

lemma InfCard-le-cmult-eq:  $\llbracket \text{InfCard}(K); L \leq K; 0 < L \rrbracket \implies K \otimes L = K$ 
apply (rule le-anti-sym)
  prefer 2
  apply (erule ltE, blast intro: cmult-le-self InfCard-is-Card)
  apply (frule InfCard-is-Card [THEN Card-is-Ord, THEN le-refl])
  apply (rule cmult-le-mono [THEN le-trans], assumption+)
  apply (simp add: InfCard-csquare-eq)
done

```

```

lemma InfCard-cmult-eq:  $\llbracket \text{InfCard}(K); \text{InfCard}(L) \rrbracket \implies K \otimes L = K \cup L$ 
apply (rule-tac  $i = K$  and  $j = L$  in Ord-linear-le)
  apply (typecheck add: InfCard-is-Card Card-is-Ord)
  apply (rule cmult-commute [THEN ssubst])
  apply (rule Un-commute [THEN ssubst])
  apply (simp-all add: InfCard-is-Limit [THEN Limit-has-0] InfCard-le-cmult-eq
    subset-Un-iff2 [THEN iffD1] le-imp-subset)
done

```

```

lemma InfCard-cdouble-eq:  $\text{InfCard}(K) \implies K \oplus K = K$ 
apply (simp add: cmult-2 [symmetric] InfCard-is-Card cmult-commute)
  apply (simp add: InfCard-le-cmult-eq InfCard-is-Limit Limit-has-0 Limit-has-succ)
done

```

```

lemma InfCard-le-cadd-eq:  $\llbracket \text{InfCard}(K); L \leq K \rrbracket \implies K \oplus L = K$ 
apply (rule le-anti-sym)
  prefer 2
  apply (erule ltE, blast intro: cadd-le-self InfCard-is-Card)
  apply (frule InfCard-is-Card [THEN Card-is-Ord, THEN le-refl])
  apply (rule cadd-le-mono [THEN le-trans], assumption+)
  apply (simp add: InfCard-cdouble-eq)

```

done

lemma *InfCard-cadd-eq*: $\llbracket \text{InfCard}(K); \text{InfCard}(L) \rrbracket \implies K \oplus L = K \cup L$
apply (rule-tac $i = K$ **and** $j = L$ **in** *Ord-linear-le*)
apply (typecheck add: *InfCard-is-Card Card-is-Ord*)
apply (rule *cadd-commute* [THEN *ssubst*])
apply (rule *Un-commute* [THEN *ssubst*])
apply (simp-all add: *InfCard-le-cadd-eq subset-Un-iff2* [THEN *iffD1*] *le-imp-subset*)
done

34.6 For Every Cardinal Number There Exists A Greater One

This result is Kunen's Theorem 10.16, which would be trivial using AC

lemma *Ord-jump-cardinal*: $\text{Ord}(\text{jump-cardinal}(K))$
unfolding *jump-cardinal-def*
apply (rule *Ord-is-Transset* [THEN [2] *OrdI*])
prefer 2 **apply** (blast intro!: *Ord-ordertype*)
unfolding *Transset-def*
apply (safe del: *subsetI*)
apply (simp add: *ordertype-pred-unfold, safe*)
apply (rule *UN-I*)
apply (rule-tac [2] *ReplaceI*)
prefer 4 **apply** (blast intro: *well-ord-subset elim!*: *predE*)
done

lemma *jump-cardinal-iff*:
 $i \in \text{jump-cardinal}(K) \longleftrightarrow$
 $(\exists r \ X. \ r \subseteq K * K \wedge X \subseteq K \wedge \text{well-ord}(X, r) \wedge i = \text{ordertype}(X, r))$
unfolding *jump-cardinal-def*
apply (blast del: *subsetI*)
done

lemma *K-lt-jump-cardinal*: $\text{Ord}(K) \implies K < \text{jump-cardinal}(K)$
apply (rule *Ord-jump-cardinal* [THEN [2] *ltI*])
apply (rule *jump-cardinal-iff* [THEN *iffD2*])
apply (rule-tac $x = \text{Memrel}(K)$ **in** *exI*)
apply (rule-tac $x = K$ **in** *exI*)
apply (simp add: *ordertype-Memrel well-ord-Memrel*)
apply (simp add: *Memrel-def subset-iff*)
done

lemma *Card-jump-cardinal-lemma*:
 $\llbracket \text{well-ord}(X, r); \ r \subseteq K * K; \ X \subseteq K; \ f \in \text{bij}(\text{ordertype}(X, r), \text{jump-cardinal}(K)) \rrbracket$
 $\implies \text{jump-cardinal}(K) \in \text{jump-cardinal}(K)$

```

apply (subgoal-tac f O ordermap (X,r)  $\in$  bij (X, jump-cardinal (K)))
  prefer 2 apply (blast intro: comp-bij ordermap-bij)
apply (rule jump-cardinal-iff [THEN iffD2])
apply (intro exI conjI)
apply (rule subset-trans [OF rvimage-type Sigma-mono], assumption+)
apply (erule bij-is-inj [THEN well-ord-rvimage])
apply (rule Ord-jump-cardinal [THEN well-ord-Memrel])
apply (simp add: well-ord-Memrel [THEN [2] bij-ordertype-vimage]
  ordertype-Memrel Ord-jump-cardinal)
done

```

```

lemma Card-jump-cardinal: Card(jump-cardinal(K))
apply (rule Ord-jump-cardinal [THEN CardI])
  unfolding eqpoll-def
apply (safe dest!: ltD jump-cardinal-iff [THEN iffD1])
apply (blast intro: Card-jump-cardinal-lemma [THEN mem-irrefl])
done

```

34.7 Basic Properties of Successor Cardinals

```

lemma csucc-basic: Ord(K)  $\implies$  Card(csucc(K))  $\wedge$  K < csucc(K)
  unfolding csucc-def
apply (rule LeastI)
apply (blast intro: Card-jump-cardinal K-lt-jump-cardinal Ord-jump-cardinal)+
done

```

```

lemmas Card-csucc = csucc-basic [THEN conjunct1]

```

```

lemmas lt-csucc = csucc-basic [THEN conjunct2]

```

```

lemma Ord-0-lt-csucc: Ord(K)  $\implies$  0 < csucc(K)
by (blast intro: Ord-0-le lt-csucc lt-trans1)

```

```

lemma csucc-le:  $\llbracket \text{Card}(L); K < L \rrbracket \implies \text{csucc}(K) \leq L$ 
  unfolding csucc-def
apply (rule Least-le)
apply (blast intro: Card-is-Ord)+
done

```

```

lemma lt-csucc-iff:  $\llbracket \text{Ord}(i); \text{Card}(K) \rrbracket \implies i < \text{csucc}(K) \longleftrightarrow |i| \leq K$ 
apply (rule iffI)
apply (rule-tac [2] Card-lt-imp-lt)
apply (erule-tac [2] lt-trans1)
apply (simp-all add: lt-csucc Card-csucc Card-is-Ord)
apply (rule notI [THEN not-lt-imp-le])
apply (rule Card-cardinal [THEN csucc-le, THEN lt-trans1, THEN lt-irrefl], as-
  sumption)
apply (rule Ord-cardinal-le [THEN lt-trans1])

```


apply (*simp-all add: Ord-cardinal Card-is-Ord*)
done

lemma *Card-lt-csucc-iff*:
 $\llbracket \text{Card}(K'); \text{Card}(K) \rrbracket \implies K' < \text{csucc}(K) \longleftrightarrow K' \leq K$
by (*simp add: lt-csucc-iff Card-cardinal-eq Card-is-Ord*)

lemma *InfCard-csucc*: $\text{InfCard}(K) \implies \text{InfCard}(\text{csucc}(K))$
by (*simp add: InfCard-def Card-csucc Card-is-Ord*
 $\text{lt-csucc [THEN leI, THEN [2] le-trans]}$)

34.7.1 Removing elements from a finite set decreases its cardinality

lemma *Finite-imp-cardinal-cons [simp]*:
assumes *FA: Finite(A)* **and** *a: a ∉ A* **shows** $|\text{cons}(a, A)| = \text{succ}(|A|)$
proof –
{ **fix** *X*
have $\text{Finite}(X) \implies a \notin X \implies \text{cons}(a, X) \lesssim X \implies \text{False}$
proof (*induct X rule: Finite-induct*)
case 0 **thus** *False* **by** (*simp add: lepoll-0-iff*)
next
case (*cons x Y*)
hence $\text{cons}(x, \text{cons}(a, Y)) \lesssim \text{cons}(x, Y)$ **by** (*simp add: cons-commute*)
hence $\text{cons}(a, Y) \lesssim Y$ **using** *cons* **by** (*blast dest: cons-lepoll-consD*)
thus *False* **using** *cons* **by** *auto*
qed
}
hence [*simp*]: $\neg \text{cons}(a, A) \lesssim A$ **using** *a FA* **by** *auto*
have [*simp*]: $|A| \approx A$ **using** *Finite-imp-well-ord [OF FA]*
by (*blast intro: well-ord-cardinal-epoll*)
have ($\mu i. i \approx \text{cons}(a, A) = \text{succ}(|A|)$)
proof (*rule Least-equality [OF - - notI]*)
show $\text{succ}(|A|) \approx \text{cons}(a, A)$
by (*simp add: succ-def cons-epoll-cong mem-not-refl a*)
next
show $\text{Ord}(\text{succ}(|A|))$ **by** *simp*
next
fix *i*
assume $i \leq |A|$ $i \approx \text{cons}(a, A)$
have $\text{cons}(a, A) \approx i$ **by** (*rule eqpoll-sym*) (*rule i*)
also have $\dots \lesssim |A|$ **by** (*rule le-imp-lepoll*) (*rule i*)
also have $\dots \approx A$ **by** *simp*
finally have $\text{cons}(a, A) \lesssim A$.
thus *False* **by** *simp*
qed
thus *?thesis* **by** (*simp add: cardinal-def*)
qed

lemma *Finite-imp-succ-cardinal-Diff*:
 $\llbracket \text{Finite}(A); a \in A \rrbracket \implies \text{succ}(|A - \{a\}|) = |A|$
apply (*rule-tac* $b = A$ **in** *cons-Diff* [*THEN subst*], *assumption*)
apply (*simp add*: *Finite-imp-cardinal-cons Diff-subset* [*THEN subset-Finite*])
apply (*simp add*: *cons-Diff*)
done

lemma *Finite-imp-cardinal-Diff*: $\llbracket \text{Finite}(A); a \in A \rrbracket \implies |A - \{a\}| < |A|$
apply (*rule succ-leE*)
apply (*simp add*: *Finite-imp-succ-cardinal-Diff*)
done

lemma *Finite-cardinal-in-nat* [*simp*]: $\text{Finite}(A) \implies |A| \in \text{nat}$
proof (*induct rule*: *Finite-induct*)
 case 0 **thus** ?*case* **by** (*simp add*: *cardinal-0*)
next
 case (*cons x A*) **thus** ?*case* **by** (*simp add*: *Finite-imp-cardinal-cons*)
qed

lemma *card-Un-Int*:
 $\llbracket \text{Finite}(A); \text{Finite}(B) \rrbracket \implies |A| \# + |B| = |A \cup B| \# + |A \cap B|$
apply (*erule Finite-induct, simp*)
apply (*simp add*: *Finite-Int cons-absorb Un-cons Int-cons-left*)
done

lemma *card-Un-disjoint*:
 $\llbracket \text{Finite}(A); \text{Finite}(B); A \cap B = 0 \rrbracket \implies |A \cup B| = |A| \# + |B|$
by (*simp add*: *Finite-Un card-Un-Int*)

lemma *card-partition*:
assumes *FC*: $\text{Finite}(C)$
shows
 $\text{Finite}(\bigcup C) \implies$
 $(\forall c \in C. |c| = k) \implies$
 $(\forall c1 \in C. \forall c2 \in C. c1 \neq c2 \longrightarrow c1 \cap c2 = 0) \implies$
 $k \# * |C| = |\bigcup C|$

using *FC*
proof (*induct rule*: *Finite-induct*)
 case 0 **thus** ?*case* **by** *simp*
next
 case (*cons x B*)
 hence $x \cap \bigcup B = 0$ **by** *auto*
 thus ?*case* **using** *cons*
 by (*auto simp add*: *card-Un-disjoint*)
qed

34.7.2 Theorems by Krzysztof Grabczewski, proofs by lcp

lemmas *nat-implies-well-ord = nat-into-Ord* [*THEN well-ord-Memrel*]

```

lemma nat-sum-epoll-sum:
  assumes m:  $m \in \text{nat}$  and n:  $n \in \text{nat}$  shows  $m + n \approx m \# + n$ 
proof -
  have  $m + n \approx |m+n|$  using m n
  by (blast intro: nat-implies-well-ord well-ord-radd well-ord-cardinal-epoll eqpoll-sym)
  also have  $\dots = m \# + n$  using m n
  by (simp add: nat-cadd-eq-add [symmetric] cadd-def)
  finally show ?thesis .
qed

lemma Ord-subset-natD [rule-format]:  $\text{Ord}(i) \implies i \subseteq \text{nat} \implies i \in \text{nat} \mid i = \text{nat}$ 
proof (induct i rule: trans-induct3)
  case 0 thus ?case by auto
next
  case (succ i) thus ?case by auto
next
  case (limit l) thus ?case
  by (blast dest: nat-le-Limit le-imp-subset)
qed

lemma Ord-nat-subset-into-Card:  $\llbracket \text{Ord}(i); i \subseteq \text{nat} \rrbracket \implies \text{Card}(i)$ 
  by (blast dest: Ord-subset-natD intro: Card-nat nat-into-Card)

end

```

35 Main ZF Theory: Everything Except AC

theory *ZF* **imports** *List IntDiv CardinalArith* **begin**

35.1 Iteration of the function F

```

consts iterates ::  $[i \Rightarrow i, i, i] \Rightarrow i$  ( $\langle \langle \text{notation} = \langle \text{mixfix iterates} \rangle \rangle \hat{\sim} '(-)' \rangle$ ) [60,1000,1000]
60)

```

primrec

$$F^{\frown 0} (x) = x$$

$$F^{\frown (\text{succ}(n))} (x) = F(F^{\frown n} (x))$$

definition

```

iterates-omega ::  $[i \Rightarrow i, i] \Rightarrow i$  ( $\langle \langle \text{notation} = \langle \text{mixfix iterates-omega} \rangle \rangle \hat{\sim} \omega '(-)' \rangle$ )
[60,1000] 60) where
   $F^{\frown \omega} (x) \equiv \bigcup_{n \in \text{nat}} F^{\frown n} (x)$ 

```

lemma *iterates-triv*:

```

 $\llbracket n \in \text{nat}; F(x) = x \rrbracket \implies F^{\frown n} (x) = x$ 
by (induct n rule: nat-induct, simp-all)

```

lemma *iterates-type* [*TC*]:

$$\llbracket n \in \text{nat}; a \in A; \bigwedge x. x \in A \implies F(x) \in A \rrbracket$$

$$\implies F^{\wedge n}(a) \in A$$
by (*induct n rule: nat-induct, simp-all*)

lemma *iterates-omega-triv*:

$$F(x) = x \implies F^{\wedge \omega}(x) = x$$
by (*simp add: iterates-omega-def iterates-triv*)

lemma *Ord-iterates [simp]*:

$$\llbracket n \in \text{nat}; \bigwedge i. \text{Ord}(i) \implies \text{Ord}(F(i)); \text{Ord}(x) \rrbracket$$

$$\implies \text{Ord}(F^{\wedge n}(x))$$
by (*induct n rule: nat-induct, simp-all*)

lemma *iterates-commute*: $n \in \text{nat} \implies F(F^{\wedge n}(x)) = F^{\wedge n}(F(x))$
by (*induct-tac n, simp-all*)

35.2 Transfinite Recursion

Transfinite recursion for definitions based on the three cases of ordinals

definition

$\text{transrec3} :: [i, i, [i, i] \Rightarrow i, [i, i] \Rightarrow i] \Rightarrow i$ **where**
 $\text{transrec3}(k, a, b, c) \equiv$
 $\text{transrec}(k, \lambda x r.$
 $\quad \text{if } x=0 \text{ then } a$
 $\quad \text{else if } \text{Limit}(x) \text{ then } c(x, \lambda y \in x. r'y)$
 $\quad \text{else } b(\text{Arith.pred}(x), r \text{ ` Arith.pred}(x)))$

lemma *transrec3-0 [simp]*: $\text{transrec3}(0, a, b, c) = a$
by (*rule transrec3-def [THEN def-transrec, THEN trans], simp*)

lemma *transrec3-succ [simp]*:
 $\text{transrec3}(\text{succ}(i), a, b, c) = b(i, \text{transrec3}(i, a, b, c))$
by (*rule transrec3-def [THEN def-transrec, THEN trans], simp*)

lemma *transrec3-Limit*:
 $\text{Limit}(i) \implies$
 $\text{transrec3}(i, a, b, c) = c(i, \lambda j \in i. \text{transrec3}(j, a, b, c))$
by (*rule transrec3-def [THEN def-transrec, THEN trans], force*)

declaration $\langle \text{fn } - \Rightarrow$
 $\text{Simplifier.map-ss } (\text{Simplifier.set-mksimps } (\text{fn ctxt} \Rightarrow$
 $\text{map mk-eq o Ord-atomize o Variable.gen-all ctxt}))$
 \rangle

end

36 The Axiom of Choice

theory *AC* **imports** *ZF* **begin**

This definition comes from Halmos (1960), page 59.

axiomatization where

AC: $\llbracket a \in A; \bigwedge x. x \in A \implies (\exists y. y \in B(x)) \rrbracket \implies \exists z. z \in Pi(A, B)$

lemma *AC-Pi*: $\llbracket \bigwedge x. x \in A \implies (\exists y. y \in B(x)) \rrbracket \implies \exists z. z \in Pi(A, B)$

apply (*case-tac* *A=0*)

apply (*simp add: Pi-empty1*)

apply (*blast intro: AC*)

done

lemma *AC-ball-Pi*: $\forall x \in A. \exists y. y \in B(x) \implies \exists y. y \in Pi(A, B)$

apply (*rule AC-Pi*)

apply (*erule bspec, assumption*)

done

lemma *AC-Pi-Pow*: $\exists f. f \in (\prod X \in Pow(C) - \{0\}. X)$

apply (*rule-tac* *B1 = $\lambda x. x$ in AC-Pi [THEN exE]*)

apply (*erule-tac [2] exI, blast*)

done

lemma *AC-func*:

$\llbracket \bigwedge x. x \in A \implies (\exists y. y \in x) \rrbracket \implies \exists f \in A \multimap \bigcup (A). \forall x \in A. f'x \in x$

apply (*rule-tac* *B1 = $\lambda x. x$ in AC-Pi [THEN exE]*)

prefer 2 apply (*blast dest: apply-type intro: Pi-type, blast*)

done

lemma *non-empty-family*: $\llbracket 0 \notin A; x \in A \rrbracket \implies \exists y. y \in x$

by (*subgoal-tac* *x \neq 0, blast+*)

lemma *AC-func0*: $0 \notin A \implies \exists f \in A \multimap \bigcup (A). \forall x \in A. f'x \in x$

apply (*rule AC-func*)

apply (*simp-all add: non-empty-family*)

done

lemma *AC-func-Pow*: $\exists f \in (Pow(C) - \{0\}) \multimap C. \forall x \in Pow(C) - \{0\}. f'x \in x$

apply (*rule AC-func0 [THEN bexE]*)

apply (*rule-tac [2] bexI*)

prefer 2 apply *assumption*

apply (*erule-tac [2] fun-weaken-type, blast+*)

done

lemma *AC-Pi0*: $0 \notin A \implies \exists f. f \in (\prod x \in A. x)$

```

apply (rule AC-Pi)
apply (simp-all add: non-empty-family)
done

end

```

37 Zorn's Lemma

theory *Zorn* **imports** *OrderArith AC Inductive* **begin**

Based upon the unpublished article “Towards the Mechanization of the Proofs of Some Classical Theorems of Set Theory,” by Abrial and Laffitte.

definition

Subset-rel :: $i \Rightarrow i$ **where**
 $\text{Subset-rel}(A) \equiv \{z \in A * A . \exists x y. z = \langle x, y \rangle \wedge x <= y \wedge x \neq y\}$

definition

chain :: $i \Rightarrow i$ **where**
 $\text{chain}(A) \equiv \{F \in \text{Pow}(A). \forall X \in F. \forall Y \in F. X <= Y \mid Y <= X\}$

definition

super :: $[i, i] \Rightarrow i$ **where**
 $\text{super}(A, c) \equiv \{d \in \text{chain}(A). c <= d \wedge c \neq d\}$

definition

maxchain :: $i \Rightarrow i$ **where**
 $\text{maxchain}(A) \equiv \{c \in \text{chain}(A). \text{super}(A, c) = \emptyset\}$

definition

increasing :: $i \Rightarrow i$ **where**
 $\text{increasing}(A) \equiv \{f \in \text{Pow}(A) \rightarrow \text{Pow}(A). \forall x. x <= A \longrightarrow x <= f'x\}$

Lemma for the inductive definition below

lemma *Union-in-Pow*: $Y \in \text{Pow}(\text{Pow}(A)) \implies \bigcup(Y) \in \text{Pow}(A)$
by *blast*

We could make the inductive definition conditional on $\text{next} \in \text{increasing}(S)$ but instead we make this a side-condition of an introduction rule. Thus the induction rule lets us assume that condition! Many inductive proofs are therefore unconditional.

consts

TFin :: $[i, i] \Rightarrow i$

inductive

domains $\text{TFin}(S, \text{next}) \subseteq \text{Pow}(S)$

intros

nextI: $\llbracket x \in \text{TFin}(S, \text{next}); \text{next} \in \text{increasing}(S) \rrbracket$
 $\implies \text{next}'x \in \text{TFin}(S, \text{next})$

Pow-UnionI: $Y \in \text{Pow}(\text{TFin}(S, \text{next})) \implies \bigcup(Y) \in \text{TFin}(S, \text{next})$

monos *Pow-mono*
con-defs *increasing-def*
type-intros *CollectD1 [THEN apply-funtype] Union-in-Pow*

37.1 Mathematical Preamble

lemma *Union-lemma0*: $(\forall x \in C. x \leq A \mid B \leq x) \implies \bigcup(C) \leq A \mid B \leq \bigcup(C)$
by *blast*

lemma *Inter-lemma0*:
 $\llbracket c \in C; \forall x \in C. A \leq x \mid x \leq B \rrbracket \implies A \subseteq \bigcap(C) \mid \bigcap(C) \subseteq B$
by *blast*

37.2 The Transfinite Construction

lemma *increasingD1*: $f \in \text{increasing}(A) \implies f \in \text{Pow}(A) \multimap \text{Pow}(A)$
unfolding *increasing-def*
apply (*erule CollectD1*)
done

lemma *increasingD2*: $\llbracket f \in \text{increasing}(A); x \leq A \rrbracket \implies x \subseteq f'x$
by (*unfold increasing-def, blast*)

lemmas *TFin-UnionI = PowI [THEN TFin.Pow-UnionI]*

lemmas *TFin-is-subset = TFin.dom-subset [THEN subsetD, THEN PowD]*

Structural induction on $\text{TFin}(S, \text{next})$

lemma *TFin-induct*:
 $\llbracket n \in \text{TFin}(S, \text{next});$
 $\bigwedge x. \llbracket x \in \text{TFin}(S, \text{next}); P(x); \text{next} \in \text{increasing}(S) \rrbracket \implies P(\text{next}'x);$
 $\bigwedge Y. \llbracket Y \subseteq \text{TFin}(S, \text{next}); \forall y \in Y. P(y) \rrbracket \implies P(\bigcup(Y))$
 $\rrbracket \implies P(n)$
by (*erule TFin.induct, blast+*)

37.3 Some Properties of the Transfinite Construction

lemmas *increasing-trans = subset-trans [OF - increasingD2,*
OF - - TFin-is-subset]

Lemma 1 of section 3.1

lemma *TFin-linear-lemma1*:
 $\llbracket n \in \text{TFin}(S, \text{next}); m \in \text{TFin}(S, \text{next});$
 $\forall x \in \text{TFin}(S, \text{next}). x \leq m \longrightarrow x = m \mid \text{next}'x \leq m \rrbracket$
 $\implies n \leq m \mid \text{next}'m \leq n$
apply (*erule TFin-induct*)

apply (*erule-tac* [2] *Union-lemma0*)

apply (*blast dest: increasing-trans*)
done

Lemma 2 of section 3.2. Interesting in its own right! Requires $next \in increasing(S)$ in the second induction step.

lemma *TFin-linear-lemma2*:

$\llbracket m \in TFin(S, next); next \in increasing(S) \rrbracket$
 $\implies \forall n \in TFin(S, next). n \leq m \longrightarrow n = m \mid next'n \subseteq m$

apply (*erule TFin-induct*)

apply (*rule impI* [*THEN ballI*])

case split using *TFin-linear-lemma1*

apply (*rule-tac* $n1 = n$ **and** $m1 = x$ **in** *TFin-linear-lemma1* [*THEN disjE*],
assumption+)

apply (*blast del: subsetI*
intro: increasing-trans subsetI, blast)

second induction step

apply (*rule impI* [*THEN ballI*])

apply (*rule Union-lemma0* [*THEN disjE*])

apply (*erule-tac* [3] *disjI2*)

prefer 2 **apply** *blast*

apply (*rule ballI*)

apply (*erule bspec, assumption*)

apply (*erule subsetD, assumption*)

apply (*rule-tac* $n1 = n$ **and** $m1 = x$ **in** *TFin-linear-lemma1* [*THEN disjE*],
assumption+, blast)

apply (*erule increasingD2* [*THEN subset-trans, THEN disjI1*])

apply (*blast dest: TFin-is-subset*)
done

a more convenient form for Lemma 2

lemma *TFin-subsetD*:

$\llbracket n \leq m; m \in TFin(S, next); n \in TFin(S, next); next \in increasing(S) \rrbracket$
 $\implies n = m \mid next'n \subseteq m$

by (*blast dest: TFin-linear-lemma2* [*rule-format*])

Consequences from section 3.3 – Property 3.2, the ordering is total

lemma *TFin-subset-linear*:

$\llbracket m \in TFin(S, next); n \in TFin(S, next); next \in increasing(S) \rrbracket$
 $\implies n \subseteq m \mid m \leq n$

apply (*rule disjE*)

apply (*rule TFin-linear-lemma1* [*OF - TFin-linear-lemma2*])

apply (*assumption+, erule disjI2*)

apply (*blast del: subsetI*
intro: subsetI increasingD2 [*THEN subset-trans*] *TFin-is-subset*)

done

Lemma 3 of section 3.3

lemma *equal-next-upper*:

$\llbracket n \in TFin(S, next); m \in TFin(S, next); m = next'm \rrbracket \implies n \subseteq m$
apply (*erule* *TFin-induct*)
apply (*drule* *TFin-subsetD*)
apply (*assumption+*, *force*, *blast*)
done

Property 3.3 of section 3.3

lemma *equal-next-Union*:

$\llbracket m \in TFin(S, next); next \in increasing(S) \rrbracket$
 $\implies m = next'm <-> m = \bigcup (TFin(S, next))$
apply (*rule iffI*)
apply (*rule Union-upper* [*THEN equalityI*])
apply (*rule-tac* [2] *equal-next-upper* [*THEN Union-least*])
apply (*assumption+*)
apply (*erule ssubst*)
apply (*rule increasingD2* [*THEN equalityI*], *assumption*)
apply (*blast del: subsetI*
intro: subsetI TFin-UnionI TFin.nextI TFin-is-subset)
done

37.4 Hausdorff's Theorem: Every Set Contains a Maximal Chain

NOTE: We assume the partial ordering is \subseteq , the subset relation!

* Defining the "next" operation for Hausdorff's Theorem *

lemma *chain-subset-Pow*: $chain(A) \subseteq Pow(A)$
unfolding *chain-def*
apply (*rule Collect-subset*)
done

lemma *super-subset-chain*: $super(A, c) \subseteq chain(A)$
unfolding *super-def*
apply (*rule Collect-subset*)
done

lemma *maxchain-subset-chain*: $maxchain(A) \subseteq chain(A)$
unfolding *maxchain-def*
apply (*rule Collect-subset*)
done

lemma *choice-super*:

$\llbracket ch \in (\prod X \in Pow(chain(S)) - \{0\}. X); X \in chain(S); X \notin maxchain(S) \rrbracket$
 $\implies ch \text{ ' } super(S, X) \in super(S, X)$

apply (*erule apply-type*)
apply (*unfold super-def maxchain-def, blast*)
done

lemma *choice-not-equals:*

$$\llbracket ch \in (\prod X \in Pow(chain(S)) - \{0\}. X); X \in chain(S); X \notin maxchain(S) \rrbracket$$

$$\implies ch \text{ ' } super(S, X) \neq X$$

apply (*rule notI*)
apply (*drule choice-super, assumption, assumption*)
apply (*simp add: super-def*)
done

This justifies Definition 4.4

lemma *Hausdorff-next-exists:*

$$ch \in (\prod X \in Pow(chain(S)) - \{0\}. X) \implies$$

$$\exists next \in increasing(S). \forall X \in Pow(S).$$

$$next \text{ ' } X = if(X \in chain(S) - maxchain(S), ch \text{ ' } super(S, X), X)$$

apply (*rule-tac x= $\lambda X \in Pow(S).$*
if $X \in chain(S) - maxchain(S)$ then $ch \text{ ' } super(S, X)$ else X
in bxI)

apply *force*
unfolding *increasing-def*
apply (*rule CollectI*)
apply (*rule lam-type*)
apply (*simp (no-asm-simp)*)
apply (*blast dest: super-subset-chain [THEN subsetD]*
chain-subset-Pow [THEN subsetD] choice-super)

Now, verify that it increases

apply (*simp (no-asm-simp) add: Pow-iff subset-refl*)
apply *safe*
apply (*drule choice-super*)
apply (*assumption+*)
apply (*simp add: super-def, blast*)
done

Lemma 4

lemma *TFin-chain-lemma4:*

$$\llbracket c \in TFin(S, next);$$

$$ch \in (\prod X \in Pow(chain(S)) - \{0\}. X);$$

$$next \in increasing(S);$$

$$\forall X \in Pow(S). next \text{ ' } X =$$

$$if(X \in chain(S) - maxchain(S), ch \text{ ' } super(S, X), X) \rrbracket$$

$$\implies c \in chain(S)$$

apply (*erule TFin-induct*)
apply (*simp (no-asm-simp) add: chain-subset-Pow [THEN subsetD, THEN PowD]*
choice-super [THEN super-subset-chain [THEN subsetD]])
unfolding *chain-def*
apply (*rule CollectI, blast, safe*)

```

apply (rule-tac  $m1=B$  and  $n1=Ba$  in TFin-subset-linear [THEN disjE], fast+)

Blast-tac's slow

done

theorem Hausdorff:  $\exists c. c \in \text{maxchain}(S)$ 
apply (rule AC-Pi-Pow [THEN exE])
apply (rule Hausdorff-next-exists [THEN bexE], assumption)
apply (rename-tac ch next)
apply (subgoal-tac  $\bigcup (TFin (S, next)) \in \text{chain} (S)$  )
prefer 2
  apply (blast intro!: TFin-chain-lemma4 subset-refl [THEN TFin-UnionI])
apply (rule-tac  $x = \bigcup (TFin (S, next))$  in exI)
apply (rule classical)
apply (subgoal-tac next ‘  $\text{Union}(TFin (S, next)) = \bigcup (TFin (S, next))$  )
apply (rule-tac [2] equal-next-Union [THEN iffD2, symmetric])
apply (rule-tac [2] subset-refl [THEN TFin-UnionI])
prefer 2 apply assumption
apply (rule-tac [2] refl)
apply (simp add: subset-refl [THEN TFin-UnionI,
  THEN TFin.dom-subset [THEN subsetD, THEN PowD]])
apply (erule choice-not-equals [THEN notE])
apply (assumption+)
done

```

37.5 Zorn's Lemma: If All Chains in S Have Upper Bounds In S, then S contains a Maximal Element

Used in the proof of Zorn's Lemma

lemma *chain-extend*:

$\llbracket c \in \text{chain}(A); z \in A; \forall x \in c. x \leq z \rrbracket \implies \text{cons}(z, c) \in \text{chain}(A)$
by (*unfold chain-def, blast*)

lemma *Zorn*: $\forall c \in \text{chain}(S). \bigcup(c) \in S \implies \exists y \in S. \forall z \in S. y \leq z \longrightarrow y = z$
apply (rule *Hausdorff* [THEN *exE*])
apply (simp add: *maxchain-def*)
apply (rename-tac *c*)
apply (rule-tac $x = \bigcup(c)$ in *bexI*)
prefer 2 **apply** *blast*
apply *safe*
apply (rename-tac *z*)
apply (rule *classical*)
apply (subgoal-tac $\text{cons}(z, c) \in \text{super}(S, c)$)
apply (blast elim: *equalityE*)
apply (*unfold super-def, safe*)
apply (*fast elim: chain-extend*)
apply (*fast elim: equalityE*)
done

Alternative version of Zorn's Lemma

theorem *Zorn2*:

$\forall c \in \text{chain}(S). \exists y \in S. \forall x \in c. x \subseteq y \implies \exists y \in S. \forall z \in S. y \leq z \longrightarrow y = z$
apply (*cut-tac Hausdorff maxchain-subset-chain*)
apply (*erule exE*)
apply (*drule subsetD, assumption*)
apply (*drule bspec, assumption, erule bexE*)
apply (*rule-tac x = y in bexI*)
prefer 2 **apply** *assumption*
apply *clarify*
apply *rule* **apply** *assumption*
apply *rule*
apply (*rule ccontr*)
apply (*frule-tac z=z in chain-extend*)
apply (*assumption, blast*)
unfolding *maxchain-def super-def*
apply (*blast elim!: equalityCE*)
done

37.6 Zermelo's Theorem: Every Set can be Well-Ordered

Lemma 5

lemma *TFin-well-lemma5*:

$\llbracket n \in \text{TFin}(S, \text{next}); Z \subseteq \text{TFin}(S, \text{next}); z:Z; \neg \bigcap (Z) \in Z \rrbracket$
 $\implies \forall m \in Z. n \subseteq m$
apply (*erule TFin-induct*)
prefer 2 **apply** *blast*

second induction step is easy

apply (*rule ballI*)
apply (*rule bspec [THEN TFin-subsetD, THEN disjE], auto*)
apply (*subgoal-tac m = $\bigcap (Z)$)*)
apply *blast+*
done

Well-ordering of $\text{TFin}(S, \text{next})$

lemma *well-ord-TFin-lemma*: $\llbracket Z \subseteq \text{TFin}(S, \text{next}); z \in Z \rrbracket \implies \bigcap (Z) \in Z$
apply (*rule classical*)
apply (*subgoal-tac Z = $\{\bigcup (\text{TFin}(S, \text{next}))\}$*)
apply (*simp (no-asm-simp) add: Inter-singleton*)
apply (*erule equal-singleton*)
apply (*rule Union-upper [THEN equalityI]*)
apply (*rule-tac [2] subset-refl [THEN TFin-UnionI, THEN TFin-well-lemma5, THEN bspec], blast+*)
done

This theorem just packages the previous result

lemma *well-ord-TFin*:

```

      next ∈ increasing(S)
      ⇒ well-ord(TFin(S,next), Subset-rel(TFin(S,next)))
apply (rule well-ordI)
      unfolding Subset-rel-def linear-def

```

Prove the well-foundedness goal

```

apply (rule wf-onI)
apply (frule well-ord-TFin-lemma, assumption)
apply (drule-tac x = ⋂(Z) in bspec, assumption)
apply blast

```

Now prove the linearity goal

```

apply (intro ballI)
apply (case-tac x=y)
apply blast

```

The $x \neq y$ case remains

```

apply (rule-tac n1=x and m1=y in TFin-subset-linear [THEN disjE],
      assumption+, blast+)
done

```

* Defining the "next" operation for Zermelo's Theorem *

```

lemma choice-Diff:
  ⌊ch ∈ (⋂ X ∈ Pow(S) - {0}. X); X ⊆ S; X ≠ S⌋ ⇒ ch ‘ (S-X) ∈ S-X
apply (erule apply-type)
apply (blast elim!: equalityE)
done

```

This justifies Definition 6.1

```

lemma Zermelo-next-exists:
  ch ∈ (⋂ X ∈ Pow(S) - {0}. X) ⇒
    ∃ next ∈ increasing(S). ∀ X ∈ Pow(S).
      next‘X = (if X=S then S else cons(ch‘(S-X), X))
apply (rule-tac x=λX∈Pow(S). if X=S then S else cons(ch‘(S-X), X)
      in bexI)
apply force
      unfolding increasing-def
apply (rule CollectI)
apply (rule lam-type)

```

Type checking is surprisingly hard!

```

apply (simp (no-asm-simp) add: Pow-iff cons-subset-iff subset-refl)
apply (blast intro!: choice-Diff [THEN DiffD1])

```

Verify that it increases

```

apply (intro allI impI)
apply (simp add: Pow-iff subset-consI subset-refl)
done

```

The construction of the injection

lemma *choice-imp-injection*:

```

  [[ch ∈ (∏ X ∈ Pow(S) - {0}. X);
    next ∈ increasing(S);
    ∀ X ∈ Pow(S). next'X = if(X=S, S, cons(ch'(S-X), X))]]
  ⇒ (λ x ∈ S. ∪({y ∈ TFin(S,next). x ∉ y}))
    ∈ inj(S, TFin(S,next) - {S})
apply (rule-tac d = λy. ch' (S-y) in lam-injective)
apply (rule DiffI)
apply (rule Collect-subset [THEN TFin-UnionI])
apply (blast intro!: Collect-subset [THEN TFin-UnionI] elim: equalityE)
apply (subgoal-tac x ∉ ∪({y ∈ TFin (S,next) . x ∉ y}) )
prefer 2 apply (blast elim: equalityE)
apply (subgoal-tac ∪({y ∈ TFin (S,next) . x ∉ y}) ≠ S)
prefer 2 apply (blast elim: equalityE)

```

For proving $x \in \text{next}' \bigcup(\dots)$. Abrial and Laffitte's justification appears to be faulty.

```

apply (subgoal-tac ¬ next' Union({y ∈ TFin (S,next) . x ∉ y})
  ⊆ ∪({y ∈ TFin (S,next) . x ∉ y}) )
prefer 2
apply (simp del: Union-iff
  add: Collect-subset [THEN TFin-UnionI, THEN TFin-is-subset]
  Pow-iff cons-subset-iff subset-refl choice-Diff [THEN DiffD2])
apply (subgoal-tac x ∈ next' Union({y ∈ TFin (S,next) . x ∉ y}) )
prefer 2
apply (blast intro!: Collect-subset [THEN TFin-UnionI] TFin.nextI)

```

End of the lemmas!

```

apply (simp add: Collect-subset [THEN TFin-UnionI, THEN TFin-is-subset])
done

```

The wellordering theorem

```

theorem AC-well-ord: ∃ r. well-ord(S,r)
apply (rule AC-Pi-Pow [THEN exE])
apply (rule Zermelo-next-exists [THEN bexE], assumption)
apply (rule exI)
apply (rule well-ord-rvimage)
apply (erule-tac [2] well-ord-TFin)
apply (rule choice-imp-injection [THEN inj-weaken-type], blast+)
done

```

37.7 Zorn's Lemma for Partial Orders

Reimported from HOL by Clemens Ballarin.

definition *Chain* :: $i \Rightarrow i$ **where**

$$\text{Chain}(r) = \{A \in \text{Pow}(\text{field}(r)). \forall a \in A. \forall b \in A. \langle a, b \rangle \in r \mid \langle b, a \rangle \in r\}$$

lemma *mono-Chain*:

$r \subseteq s \implies \text{Chain}(r) \subseteq \text{Chain}(s)$
unfolding *Chain-def*
by *blast*

theorem *Zorn-po*:

assumes *po*: *Partial-order*(*r*)
and *u*: $\forall C \in \text{Chain}(r). \exists u \in \text{field}(r). \forall a \in C. \langle a, u \rangle \in r$
shows $\exists m \in \text{field}(r). \forall a \in \text{field}(r). \langle m, a \rangle \in r \longrightarrow a = m$

proof –

have *Preorder*(*r*) **using** *po* **by** (*simp add: partial-order-on-def*)
 — Mirror *r* in the set of subsets below (wrt *r*) elements of *A* (?).

let $?B = \lambda x \in \text{field}(r). r - \{x\}$ **let** $?S = ?B \text{ `` } \text{field}(r)$

have $\forall C \in \text{chain}(?S). \exists U \in ?S. \forall A \in C. A \subseteq U$

proof (*clarsimp simp: chain-def Subset-rel-def bex-image-simp*)

fix *C*

assume 1: $C \subseteq ?S$ **and** 2: $\forall A \in C. \forall B \in C. A \subseteq B \mid B \subseteq A$

let $?A = \{x \in \text{field}(r). \exists M \in C. M = ?B \{x\}$

have $C = ?B \text{ `` } ?A$ **using** 1

apply (*auto simp: image-def*)

apply *rule*

apply *rule*

apply (*drule subsetD*) **apply** *assumption*

apply (*erule CollectE*)

apply *rule* **apply** *assumption*

apply (*erule bexE*)

apply *rule* **prefer** 2 **apply** *assumption*

apply *rule*

apply (*erule lamE*) **apply** *simp*

apply *assumption*

apply (*thin-tac* $C \subseteq X$ **for** *X*)

apply (*fast elim: lamE*)

done

have $?A \in \text{Chain}(r)$

proof (*simp add: Chain-def subsetI, intro conjI ballI impI*)

fix *a b*

assume $a \in \text{field}(r)$ $r - \{a\} \in C$ $b \in \text{field}(r)$ $r - \{b\} \in C$

hence $r - \{a\} \subseteq r - \{b\} \mid r - \{b\} \subseteq r - \{a\}$ **using** 2 **by** *auto*

then show $\langle a, b \rangle \in r \mid \langle b, a \rangle \in r$

using $\langle \text{Preorder}(r) \rangle \langle a \in \text{field}(r) \rangle \langle b \in \text{field}(r) \rangle$

by (*simp add: subset-vimage1-vimage1-iff*)

qed

then obtain *u* **where** $uA: u \in \text{field}(r) \forall a \in ?A. \langle a, u \rangle \in r$

using *u*

apply *auto*

apply (*drule bspec*) **apply** *assumption*

apply *auto*

done

have $\forall A \in C. A \subseteq r - \{u\}$

```

proof (auto intro!: vimageI)
  fix a B
  assume aB:  $B \in C$   $a \in B$ 
  with 1 obtain x where  $x \in \text{field}(r)$   $B = r - \{x\}$ 
    apply -
    apply (drule subsetD) apply assumption
    apply (erule imageE)
    apply (erule lamE)
    apply simp
    done
  then show  $\langle a, u \rangle \in r$  using uA aB  $\langle \text{Preorder}(r) \rangle$ 
    by (auto simp: preorder-on-def refl-def) (blast dest: trans-onD)+
  qed
  then show  $\exists U \in \text{field}(r). \forall A \in C. A \subseteq r - \{U\}$ 
    using  $\langle u \in \text{field}(r) \rangle$  ..
  qed
from Zorn2 [OF this]
obtain m B where  $m \in \text{field}(r)$   $B = r - \{m\}$ 
   $\forall x \in \text{field}(r). B \subseteq r - \{x\} \longrightarrow B = r - \{x\}$ 
  by (auto elim!: lamE simp: ball-image-simp)
then have  $\forall a \in \text{field}(r). \langle m, a \rangle \in r \longrightarrow a = m$ 
  using po  $\langle \text{Preorder}(r) \rangle$   $\langle m \in \text{field}(r) \rangle$ 
  by (auto simp: subset-vimage1-vimage1-iff Partial-order-eq-vimage1-vimage1-iff)
then show ?thesis using  $\langle m \in \text{field}(r) \rangle$  by blast
qed

end

```

38 Cardinal Arithmetic Using AC

theory Cardinal-AC **imports** CardinalArith Zorn **begin**

38.1 Strengthened Forms of Existing Theorems on Cardinals

```

lemma cardinal-epoll:  $|A| \approx A$ 
apply (rule AC-well-ord [THEN exE])
apply (erule well-ord-cardinal-epoll)
done

```

The theorem $||A|| = |A|$

```

lemmas cardinal-idem = cardinal-epoll [THEN cardinal-cong, simp]

```

```

lemma cardinal-eqE:  $|X| = |Y| \implies X \approx Y$ 
apply (rule AC-well-ord [THEN exE])
apply (rule AC-well-ord [THEN exE])
apply (rule well-ord-cardinal-eqE, assumption+)
done

```

```

lemma cardinal-epoll-iff:  $|X| = |Y| \longleftrightarrow X \approx Y$ 

```


by (blast intro: cardinal-cong cardinal-eqE)

lemma cardinal-disjoint-Un:

$\llbracket |A|=|B|; |C|=|D|; A \cap C = 0; B \cap D = 0 \rrbracket$
 $\implies |A \cup C| = |B \cup D|$

by (simp add: cardinal-epoll-iff epoll-disjoint-Un)

lemma lepoll-imp-cardinal-le: $A \lesssim B \implies |A| \leq |B|$

apply (rule AC-well-ord [THEN exE])

apply (erule well-ord-lepoll-imp-cardinal-le, assumption)

done

lemma cadd-assoc: $(i \oplus j) \oplus k = i \oplus (j \oplus k)$

apply (rule AC-well-ord [THEN exE])

apply (rule AC-well-ord [THEN exE])

apply (rule AC-well-ord [THEN exE])

apply (rule well-ord-cadd-assoc, assumption+)

done

lemma cmult-assoc: $(i \otimes j) \otimes k = i \otimes (j \otimes k)$

apply (rule AC-well-ord [THEN exE])

apply (rule AC-well-ord [THEN exE])

apply (rule AC-well-ord [THEN exE])

apply (rule well-ord-cmult-assoc, assumption+)

done

lemma cadd-cmult-distrib: $(i \oplus j) \otimes k = (i \otimes k) \oplus (j \otimes k)$

apply (rule AC-well-ord [THEN exE])

apply (rule AC-well-ord [THEN exE])

apply (rule AC-well-ord [THEN exE])

apply (rule well-ord-cadd-cmult-distrib, assumption+)

done

lemma InfCard-square-eq: $\text{InfCard}(|A|) \implies A * A \approx A$

apply (rule AC-well-ord [THEN exE])

apply (erule well-ord-InfCard-square-eq, assumption)

done

38.2 The relationship between cardinality and le-pollence

lemma Card-le-imp-lepoll:

assumes $|A| \leq |B|$ shows $A \lesssim B$

proof –

have $A \approx |A|$

by (rule cardinal-epoll [THEN epoll-sym])

also have $\dots \lesssim |B|$

by (rule le-imp-subset [THEN subset-imp-lepoll]) (rule assms)

also have $\dots \approx B$

by (rule cardinal-epoll)

finally show *?thesis* .
qed

lemma *le-Card-iff*: $\text{Card}(K) \implies |A| \leq K \longleftrightarrow A \lesssim K$
apply (*erule Card-cardinal-eq* [THEN *subst*], *rule iffI*,
erule Card-le-imp-lepoll)
apply (*erule lepoll-imp-cardinal-le*)
done

lemma *cardinal-0-iff-0* [*simp*]: $|A| = 0 \longleftrightarrow A = 0$
apply *auto*
apply (*drule cardinal-0* [THEN *ssubst*])
apply (*blast intro: eqpoll-0-iff* [THEN *iffD1*] *cardinal-eqpoll-iff* [THEN *iffD1*])
done

lemma *cardinal-lt-iff-lesspoll*:
assumes *i*: $\text{Ord}(i)$ **shows** $i < |A| \longleftrightarrow i \prec A$
proof
assume $i < |A|$
hence $i \prec |A|$
by (*blast intro: lt-Card-imp-lesspoll Card-cardinal*)
also have $\dots \approx A$
by (*rule cardinal-eqpoll*)
finally show $i \prec A$.
next
assume $i \prec A$
also have $\dots \approx |A|$
by (*blast intro: cardinal-eqpoll eqpoll-sym*)
finally have $i \prec |A|$.
thus $i < |A|$ **using** *i*
by (*force intro: cardinal-lt-imp-lt lesspoll-cardinal-lt*)
qed

lemma *cardinal-le-imp-lepoll*: $i \leq |A| \implies i \lesssim A$
by (*blast intro: lt-Ord Card-le-imp-lepoll Ord-cardinal-le le-trans*)

38.3 Other Applications of AC

lemma *surj-implies-inj*:
assumes *f*: $f \in \text{surj}(X, Y)$ **shows** $\exists g. g \in \text{inj}(Y, X)$
proof –
from *f AC-Pi* [*of Y λy. f – “{y}”*]
obtain *z* **where** $z: z \in (\prod_{y \in Y}. f – “\{y\})$
by (*auto simp add: surj-def*) (*fast dest: apply-Pair*)
show *?thesis*
proof
show $z \in \text{inj}(Y, X)$ **using** *z surj-is-fun* [*OF f*]
by (*blast dest: apply-type Pi-memberD*
intro: apply-equality Pi-type f-imp-injective)

qed
qed

Kunen's Lemma 10.20

lemma *surj-implies-cardinal-le*:

assumes $f: f \in \text{surj}(X, Y)$ **shows** $|Y| \leq |X|$

proof (*rule lepoll-imp-cardinal-le*)

from f [*THEN surj-implies-inj*] **obtain** g **where** $g \in \text{inj}(Y, X)$..

thus $Y \lesssim X$

by (*auto simp add: lepoll-def*)

qed

Kunen's Lemma 10.21

lemma *cardinal-UN-le*:

assumes $K: \text{InfCard}(K)$

shows $(\bigwedge i. i \in K \implies |X(i)| \leq K) \implies |\bigcup i \in K. X(i)| \leq K$

proof (*simp add: K InfCard-is-Card le-Card-iff*)

have [*intro*]: $\text{Ord}(K)$ **by** (*blast intro: InfCard-is-Card Card-is-Ord K*)

assume $\bigwedge i. i \in K \implies X(i) \lesssim K$

hence $\bigwedge i. i \in K \implies \exists f. f \in \text{inj}(X(i), K)$ **by** (*simp add: lepoll-def*)

with *AC-Pi* **obtain** f **where** $f: f \in (\prod i \in K. \text{inj}(X(i), K))$

by *force*

{ **fix** z

assume $z: z \in (\bigcup i \in K. X(i))$

then obtain i **where** $i: i \in K \text{ Ord}(i) z \in X(i)$

by (*blast intro: Ord-in-Ord [of K]*)

hence $(\mu i. z \in X(i)) \leq i$ **by** (*fast intro: Least-le*)

hence $(\mu i. z \in X(i)) < K$ **by** (*best intro: lt-trans1 ltI i*)

hence $(\mu i. z \in X(i)) \in K$ **and** $z \in X(\mu i. z \in X(i))$

by (*auto intro: LeastI ltD i*)

} **note** *mems = this*

have $(\bigcup i \in K. X(i)) \lesssim K \times K$

proof (*unfold lepoll-def*)

show $\exists f. f \in \text{inj}(\bigcup \text{RepFun}(K, X), K \times K)$

apply (*rule exI*)

apply (*rule-tac c = $\lambda z. \langle \mu i. z \in X(i), f ' (\mu i. z \in X(i)) ' z \rangle$*

and $d = \lambda \langle i, j \rangle. \text{converse } (f' i) ' j$ **in** *lam-injective*)

apply (*force intro: f inj-is-fun mems apply-type Perm.left-inverse*) +

done

qed

also have $\dots \approx K$

by (*simp add: K InfCard-square-eq InfCard-is-Card Card-cardinal-eq*)

finally show $(\bigcup i \in K. X(i)) \lesssim K$.

qed

The same again, using *csucc*

lemma *cardinal-UN-lt-csucc*:

$\llbracket \text{InfCard}(K); \bigwedge i. i \in K \implies |X(i)| < \text{csucc}(K) \rrbracket$

$\implies |\bigcup i \in K. X(i)| < \text{csucc}(K)$

by (*simp add: Card-lt-csucc-iff cardinal-UN-le InfCard-is-Card Card-cardinal*)

The same again, for a union of ordinals. In use, $j(i)$ is a bit like $\text{rank}(i)$, the least ordinal j such that $i:V_{\text{from}}(A,j)$.

lemma *cardinal-UN-Ord-lt-csucc*:

$\llbracket \text{InfCard}(K); \bigwedge i. i \in K \implies j(i) < \text{csucc}(K) \rrbracket$
 $\implies (\bigcup i \in K. j(i)) < \text{csucc}(K)$

apply (*rule cardinal-UN-lt-csucc [THEN Card-lt-imp-lt], assumption*)

apply (*blast intro: Ord-cardinal-le [THEN lt-trans1] elim: ltE*)

apply (*blast intro!: Ord-UN elim: ltE*)

apply (*erule InfCard-is-Card [THEN Card-is-Ord, THEN Card-csucc]*)

done

38.4 The Main Result for Infinite-Branching Datatypes

As above, but the index set need not be a cardinal. Work backwards along the injection from W into K , given that $W \neq 0$.

lemma *inj-UN-subset*:

assumes $f: f \in \text{inj}(A,B)$ **and** $a: a \in A$

shows $(\bigcup x \in A. C(x)) \subseteq (\bigcup y \in B. C(\text{if } y \in \text{range}(f) \text{ then } \text{converse}(f) 'y \text{ else } a))$

proof (*rule UN-least*)

fix x

assume $x: x \in A$

hence $fx: f ' x \in B$ **by** (*blast intro: f inj-is-fun [THEN apply-type]*)

have $C(x) \subseteq C(\text{if } f ' x \in \text{range}(f) \text{ then } \text{converse}(f) ' (f ' x) \text{ else } a)$

using fx **by** (*simp add: inj-is-fun [THEN apply-rangeI]*)

also have $\dots \subseteq (\bigcup y \in B. C(\text{if } y \in \text{range}(f) \text{ then } \text{converse}(f) ' y \text{ else } a))$

by (*rule UN-upper [OF fx]*)

finally show $C(x) \subseteq (\bigcup y \in B. C(\text{if } y \in \text{range}(f) \text{ then } \text{converse}(f) ' y \text{ else } a))$.

qed

theorem *le-UN-Ord-lt-csucc*:

assumes $IK: \text{InfCard}(K)$ **and** $WK: |W| \leq K$ **and** $j: \bigwedge w. w \in W \implies j(w) < \text{csucc}(K)$

shows $(\bigcup w \in W. j(w)) < \text{csucc}(K)$

proof –

have $CK: \text{Card}(K)$

by (*simp add: InfCard-is-Card IK*)

then obtain f **where** $f: f \in \text{inj}(W, K)$ **using** WK

by (*auto simp add: le-Card-iff lepoll-def*)

have $OU: \text{Ord}(\bigcup w \in W. j(w))$ **using** j

by (*blast elim: ltE*)

note *lt-subset-trans* [*OF - - OU, trans*]

show *?thesis*

proof (*cases W=0*)

case *True* — solve the easy 0 case

thus *?thesis* **by** (*simp add: CK Card-is-Ord Card-csucc Ord-0-lt-csucc*)

next

```

case False
then obtain x where x: x ∈ W by blast
have (⋃ x ∈ W. j(x)) ⊆ (⋃ k ∈ K. j(if k ∈ range(f) then converse(f) ‘ k else
x))
  by (rule inj-UN-subset [OF f x])
also have ... < csucc(K) using IK
proof (rule cardinal-UN-Ord-lt-csucc)
  fix k
  assume k ∈ K
  thus j(if k ∈ range(f) then converse(f) ‘ k else x) < csucc(K) using f x j
    by (simp add: inj-converse-fun [THEN apply-type])
qed
finally show ?thesis .
qed
qed
end

```

39 Infinite-Branching Datatype Definitions

theory InfDatatype **imports** Datatype Univ Finite Cardinal-AC **begin**

lemmas fun-Limit-VfromE =

Limit-VfromE [OF apply-funtype InfCard-csucc [THEN InfCard-is-Limit]]

lemma fun-Vcsucc-lemma:

assumes f: f ∈ D → Vfrom(A, csucc(K)) **and** DK: |D| ≤ K **and** ICK: Inf-Card(K)

shows ∃ j. f ∈ D → Vfrom(A, j) ∧ j < csucc(K)

proof (rule exI, rule conjI)

show f ∈ D → Vfrom(A, ⋃ z ∈ D. μ i. f ‘ z ∈ Vfrom(A, i))

proof (rule Pi-type [OF f])

fix d

assume d: d ∈ D

show f ‘ d ∈ Vfrom(A, ⋃ z ∈ D. μ i. f ‘ z ∈ Vfrom(A, i))

proof (rule fun-Limit-VfromE [OF f d ICK])

fix x

assume x < csucc(K) f ‘ d ∈ Vfrom(A, x)

hence f ‘ d ∈ Vfrom(A, μ i. f ‘ d ∈ Vfrom(A, i)) **using** d

by (fast elim: LeastI ltE)

also have ... ⊆ Vfrom(A, ⋃ z ∈ D. μ i. f ‘ z ∈ Vfrom(A, i))

by (rule Vfrom-mono) (auto intro: d)

finally show f ‘ d ∈ Vfrom(A, ⋃ z ∈ D. μ i. f ‘ z ∈ Vfrom(A, i)) .

qed

qed

next

show (⋃ d ∈ D. μ i. f ‘ d ∈ Vfrom(A, i)) < csucc(K)

proof (rule le-UN-Ord-lt-csucc [OF ICK DK])

fix d

```

assume  $d: d \in D$ 
show  $(\mu i. f \text{ ' } d \in Vfrom(A, i)) < csucc(K)$ 
  proof (rule fun-Limit-VfromE [OF f d ICK])
    fix  $x$ 
    assume  $x < csucc(K) \text{ ' } f \text{ ' } d \in Vfrom(A, x)$ 
    thus  $(\mu i. f \text{ ' } d \in Vfrom(A, i)) < csucc(K)$ 
      by (blast intro: Least-le lt-trans1 lt-Ord)
    qed
  qed
qed

lemma subset-Vsucc:
   $\llbracket D \subseteq Vfrom(A, csucc(K)); |D| \leq K; InfCard(K) \rrbracket$ 
   $\implies \exists j. D \subseteq Vfrom(A, j) \wedge j < csucc(K)$ 
by (simp add: subset-iff-id fun-Vsucc-lemma)

lemma fun-Vsucc:
   $\llbracket |D| \leq K; InfCard(K); D \subseteq Vfrom(A, csucc(K)) \rrbracket \implies$ 
   $D \rightarrow Vfrom(A, csucc(K)) \subseteq Vfrom(A, csucc(K))$ 
apply (safe dest!: fun-Vsucc-lemma subset-Vsucc)
apply (rule Vfrom [THEN ssubst])
apply (drule fun-is-rel)

apply (rule-tac a1 = succ (succ (j  $\cup$  ja)) in UN-I [THEN UnI2])
apply (blast intro: ltD InfCard-csucc InfCard-is-Limit Limit-has-succ
  Un-least-lt)
apply (erule subset-trans [THEN PowI])
apply (fast intro: Pair-in-Vfrom Vfrom-UnI1 Vfrom-UnI2)
done

lemma fun-in-Vsucc:
   $\llbracket f: D \rightarrow Vfrom(A, csucc(K)); |D| \leq K; InfCard(K);$ 
   $D \subseteq Vfrom(A, csucc(K)) \rrbracket$ 
   $\implies f: Vfrom(A, csucc(K))$ 
by (blast intro: fun-Vsucc [THEN subsetD])

Remove  $\subseteq$  from the rule above

lemmas fun-in-Vsucc' = fun-in-Vsucc [OF - - - subsetI]

lemma Card-fun-Vsucc:
   $InfCard(K) \implies K \rightarrow Vfrom(A, csucc(K)) \subseteq Vfrom(A, csucc(K))$ 
apply (frule InfCard-is-Card [THEN Card-is-Ord])
apply (blast del: subsetI
  intro: fun-Vsucc Ord-cardinal-le i-subset-Vfrom
  lt-csucc [THEN leI, THEN le-imp-subset, THEN subset-trans])
done

```

```

lemma Card-fun-in-Vsucc:
   $\llbracket f: K \rightarrow V_{\text{from}}(A, \text{csucc}(K)); \text{InfCard}(K) \rrbracket \implies f: V_{\text{from}}(A, \text{csucc}(K))$ 
by (blast intro: Card-fun-Vsucc [THEN subsetD])

lemma Limit-csucc:  $\text{InfCard}(K) \implies \text{Limit}(\text{csucc}(K))$ 
by (erule InfCard-csucc [THEN InfCard-is-Limit])

lemmas Pair-in-Vsucc = Pair-in-VLimit [OF - - Limit-csucc]
lemmas Inl-in-Vsucc = Inl-in-VLimit [OF - Limit-csucc]
lemmas Inr-in-Vsucc = Inr-in-VLimit [OF - Limit-csucc]
lemmas zero-in-Vsucc = Limit-csucc [THEN zero-in-VLimit]
lemmas nat-into-Vsucc = nat-into-VLimit [OF - Limit-csucc]

lemmas InfCard-nat-Un-cardinal = InfCard-Un [OF InfCard-nat Card-cardinal]

lemmas le-nat-Un-cardinal =
  Un-upper2-le [OF Ord-nat Card-cardinal [THEN Card-is-Ord]]

lemmas UN-upper-cardinal = UN-upper [THEN subset-imp-lepoll, THEN lepoll-imp-cardinal-le]

lemmas Data-Arg-intros =
  SigmaI InlI InrI
  Pair-in-univ Inl-in-univ Inr-in-univ
  zero-in-univ A-into-univ nat-into-univ UnCI

lemmas inf-datatype-intros =
  InfCard-nat InfCard-nat-Un-cardinal
  Pair-in-Vsucc Inl-in-Vsucc Inr-in-Vsucc
  zero-in-Vsucc A-into-Vfrom nat-into-Vsucc
  Card-fun-in-Vsucc fun-in-Vsucc' UN-I

end
theory ZFC imports ZF InfDatatype
begin

end

```