

ZF

Lawrence C Paulson and others

October 10, 2011

Contents

1	ZF: Zermelo-Fraenkel Set Theory	13
1.1	Substitution	18
1.2	Bounded universal quantifier	18
1.3	Bounded existential quantifier	19
1.4	Rules for subsets	20
1.5	Rules for equality	20
1.6	Rules for Replace – the derived form of replacement	21
1.7	Rules for RepFun	22
1.8	Rules for Collect – forming a subset by separation	22
1.9	Rules for Unions	22
1.10	Rules for Unions of families	23
1.11	Rules for the empty set	23
1.12	Rules for Inter	24
1.13	Rules for Intersections of families	24
1.14	Rules for Powersets	24
1.15	Cantor’s Theorem: There is no surjection from a set to its powerset.	25
2	upair: Unordered Pairs	25
2.1	Unordered Pairs: constant <i>Upair</i>	25
2.2	Rules for Binary Union, Defined via <i>Upair</i>	25
2.3	Rules for Binary Intersection, Defined via <i>Upair</i>	26
2.4	Rules for Set Difference, Defined via <i>Upair</i>	26
2.5	Rules for <i>cons</i>	26
2.6	Singletons	27
2.7	Descriptions	27
2.8	Conditional Terms: <i>if-then-else</i>	28
2.9	Consequences of Foundation	29
2.10	Rules for Successor	30
2.11	Miniscoping of the Bounded Universal Quantifier	30
2.12	Miniscoping of the Bounded Existential Quantifier	31

2.13	Miniscoping of the Replacement Operator	32
2.14	Miniscoping of Unions	32
2.15	Miniscoping of Intersections	33
2.16	Other simprules	34
3	pair: Ordered Pairs	34
3.1	Sigma: Disjoint Union of a Family of Sets	35
3.2	Projections <i>fst</i> and <i>snd</i>	36
3.3	The Eliminator, <i>split</i>	36
3.4	A version of <i>split</i> for Formulae: Result Type <i>o</i>	36
4	equalities: Basic Equalities and Inclusions	37
4.1	Bounded Quantifiers	37
4.2	Converse of a Relation	38
4.3	Finite Set Constructions Using <i>cons</i>	38
4.4	Binary Intersection	40
4.5	Binary Union	41
4.6	Set Difference	42
4.7	Big Union and Intersection	44
4.8	Unions and Intersections of Families	45
4.9	Image of a Set under a Function or Relation	51
4.10	Inverse Image of a Set under a Function or Relation	52
4.11	Powerset Operator	54
4.12	RepFun	55
4.13	Collect	55
5	Fixedpt: Least and Greatest Fixed Points; the Knaster-Tarski Theorem	56
5.1	Monotone Operators	56
5.2	Proof of Knaster-Tarski Theorem using <i>lfp</i>	57
5.3	General Induction Rule for Least Fixedpoints	58
5.4	Proof of Knaster-Tarski Theorem using <i>gfp</i>	59
5.5	Coinduction Rules for Greatest Fixed Points	59
6	Bool: Booleans in Zermelo-Fraenkel Set Theory	60
6.1	Laws About 'not'	62
6.2	Laws About 'and'	63
6.3	Laws About 'or'	63
7	Sum: Disjoint Sums	64
7.1	Rules for the <i>Part</i> Primitive	64
7.2	Rules for Disjoint Sums	65
7.3	The Eliminator: <i>case</i>	66
7.4	More Rules for <i>Part(A, h)</i>	67

8	func: Functions, Function Spaces, Lambda-Abstraction	67
8.1	The Pi Operator: Dependent Function Space	67
8.2	Function Application	68
8.3	Lambda Abstraction	70
8.4	Extensionality	71
8.5	Images of Functions	72
8.6	Properties of $restrict(f, A)$	72
8.7	Unions of Functions	73
8.8	Domain and Range of a Function or Relation	74
8.9	Extensions of Functions	74
8.10	Function Updates	75
8.11	Monotonicity Theorems	75
	8.11.1 Replacement in its Various Forms	75
	8.11.2 Standard Products, Sums and Function Spaces	76
	8.11.3 Converse, Domain, Range, Field	76
	8.11.4 Images	77
9	QPair: Quine-Inspired Ordered Pairs and Disjoint Sums	78
9.1	Quine ordered pairing	79
	9.1.1 QSigma: Disjoint union of a family of sets Generalizes Cartesian product	79
	9.1.2 Projections: $qfst$, $qsnd$	80
	9.1.3 Eliminator: $qspl$ it	80
	9.1.4 $qspl$ it for predicates: result type o	81
	9.1.5 $qconverse$	81
9.2	The Quine-inspired notion of disjoint sum	81
	9.2.1 Eliminator – $qcase$	83
	9.2.2 Monotonicity	83
10	Perm: Injections, Surjections, Bijections, Composition	84
10.1	Surjective Function Space	84
10.2	Injective Function Space	85
10.3	Bijections	85
10.4	Identity Function	86
10.5	Converse of a Function	87
10.6	Converses of Injections, Surjections, Bijections	87
10.7	Composition of Two Relations	88
10.8	Domain and Range – see Suppes, Section 3.1	88
10.9	Other Results	89
10.10	Composition Preserves Functions, Injections, and Surjections	89
10.11	Dual Properties of inj and $surj$	90
	10.11.1 Inverses of Composition	90
	10.11.2 Proving that a Function is a Bijection	90
	10.11.3 Unions of Functions	91

10.11.4	Restrictions as Surjections and Bijections	91
10.11.5	Lemmas for Ramsey's Theorem	92
11	Trancl: Relations: Their General Properties and Transitive Closure	92
11.1	General properties of relations	93
11.1.1	irreflexivity	93
11.1.2	symmetry	93
11.1.3	antisymmetry	93
11.1.4	transitivity	93
11.2	Transitive closure of a relation	94
12	WF: 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	101
12.5	Unfolding <i>wftrec(r, a, H)</i>	102
12.5.1	Removal of the Premise <i>trans(r)</i>	102
13	Ordinal: Transitive Sets and Ordinals	102
13.1	Rules for Transset	103
13.1.1	Three Neat Characterisations of Transset	103
13.1.2	Consequences of Downwards Closure	103
13.1.3	Closure Properties	104
13.2	Lemmas for Ordinals	105
13.3	The Construction of Ordinals: 0, succ, Union	105
13.4	\jmath is 'less Than' for Ordinals	106
13.5	Natural Deduction Rules for Memrel	107
13.6	Transfinite Induction	108
13.6.1	Proving That \jmath is a Linear Ordering on the Ordinals	109
13.6.2	Some Rewrite Rules for \jmath , le	109
13.7	Results about Less-Than or Equals	110
13.7.1	Transitivity Laws	110
13.7.2	Union and Intersection	111
13.8	Results about Limits	112
13.9	Limit Ordinals – General Properties	113
13.9.1	Traditional 3-Way Case Analysis on Ordinals	114

14 OrdQuant: Special quantifiers	115
14.1 Quantifiers and union operator for ordinals	115
14.1.1 simplification of the new quantifiers	115
14.1.2 Union over ordinals	116
14.1.3 universal quantifier for ordinals	117
14.1.4 existential quantifier for ordinals	117
14.1.5 Rules for Ordinal-Indexed Unions	118
14.2 Quantification over a class	118
14.2.1 Relativized universal quantifier	118
14.2.2 Relativized existential quantifier	119
14.2.3 One-point rule for bounded quantifiers	121
14.2.4 Sets as Classes	121
15 Nat-ZF: The Natural numbers As a Least Fixed Point	122
15.1 Injectivity Properties and Induction	123
15.2 Variations on Mathematical Induction	124
15.3 <i>quasinat</i> : to allow a case-split rule for <i>nat-case</i>	125
15.4 Recursion on the Natural Numbers	126
16 Inductive-ZF: Inductive and Coinductive Definitions	126
17 Epsilon: Epsilon Induction and Recursion	127
17.1 Basic Closure Properties	128
17.2 Leastness of <i>eclose</i>	128
17.3 Epsilon Recursion	129
17.4 Rank	130
17.5 Corollaries of Leastness	131
18 Order: Partial and Total Orderings: Basic Definitions and Properties	132
18.1 Immediate Consequences of the Definitions	134
18.2 Restricting an Ordering's Domain	135
18.3 Empty and Unit Domains	136
18.3.1 Relations over the Empty Set	136
18.3.2 The Empty Relation Well-Orders the Unit Set	136
18.4 Order-Isomorphisms	136
18.5 Main results of Kunen, Chapter 1 section 6	138
18.6 Towards Kunen's Theorem 6.3: Linearity of the Similarity Relation	139
18.7 Miscellaneous Results by Krzysztof Grabczewski	141
18.8 Lemmas for the Reflexive Orders	141

19 OrderArith: Combining Orderings: Foundations of Ordinal Arithmetic	142
19.1 Addition of Relations – Disjoint Sum	143
19.1.1 Rewrite rules. Can be used to obtain introduction rules	143
19.1.2 Elimination Rule	143
19.1.3 Type checking	143
19.1.4 Linearity	143
19.1.5 Well-foundedness	144
19.1.6 An <i>ord-iso</i> congruence law	144
19.1.7 Associativity	144
19.2 Multiplication of Relations – Lexicographic Product	144
19.2.1 Rewrite rule. Can be used to obtain introduction rules	144
19.2.2 Type checking	145
19.2.3 Linearity	145
19.2.4 Well-foundedness	145
19.2.5 An <i>ord-iso</i> congruence law	145
19.2.6 Distributive law	146
19.2.7 Associativity	146
19.3 Inverse Image of a Relation	147
19.3.1 Rewrite rule	147
19.3.2 Type checking	147
19.3.3 Partial Ordering Properties	147
19.3.4 Linearity	147
19.3.5 Well-foundedness	147
19.4 Every well-founded relation is a subset of some inverse image of an ordinal	148
19.5 Other Results	149
19.5.1 The Empty Relation	149
19.5.2 The "measure" relation is useful with wfrec	149
19.5.3 Well-foundedness of Unions	150
19.5.4 Bijections involving Powersets	150
20 OrderType: Order Types and Ordinal Arithmetic	150
20.1 Proofs needing the combination of Ordinal.thy and Order.thy	151
20.2 Ordermap and ordertype	152
20.2.1 Unfolding of ordermap	152
20.2.2 Showing that ordermap, ordertype yield ordinals	152
20.2.3 ordermap preserves the orderings in both directions	152
20.2.4 Isomorphisms involving ordertype	153
20.2.5 Basic equalities for ordertype	153
20.2.6 A fundamental unfolding law for ordertype.	154
20.3 Alternative definition of ordinal	154
20.4 Ordinal Addition	154
20.4.1 Order Type calculations for radd	154

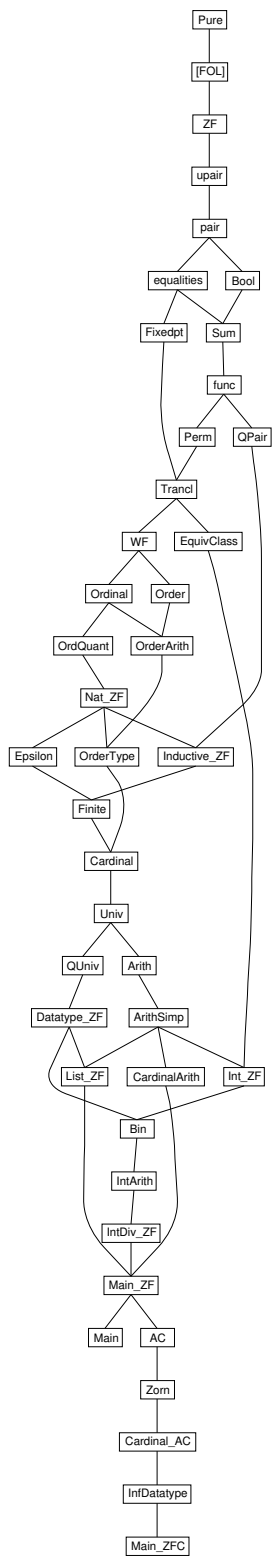
20.4.2	ordify: trivial coercion to an ordinal	155
20.4.3	Basic laws for ordinal addition	155
20.4.4	Ordinal addition with successor – via associativity!	157
20.5	Ordinal Subtraction	158
20.6	Ordinal Multiplication	159
20.6.1	A useful unfolding law	159
20.6.2	Basic laws for ordinal multiplication	160
20.6.3	Ordering/monotonicity properties of ordinal multiplication	161
20.7	The Relation Lt	161
21	Finite: Finite Powerset Operator and Finite Function Space	162
21.1	Finite Powerset Operator	162
21.2	Finite Function Space	164
21.3	The Contents of a Singleton Set	164
22	Cardinal: Cardinal Numbers Without the Axiom of Choice	165
22.1	The Schroeder-Bernstein Theorem	166
22.2	lesspoll: contributions by Krzysztof Grabczewski	167
22.3	The finite cardinals	171
22.4	The first infinite cardinal: Omega, or nat	172
22.5	Towards Cardinal Arithmetic	172
22.6	Lemmas by Krzysztof Grabczewski	173
22.7	Finite and infinite sets	174
23	Univ: The Cumulative Hierarchy and a Small Universe for Recursive Types	177
23.1	Immediate Consequences of the Definition of $Vfrom(A, i)$	178
23.1.1	Monotonicity	178
23.1.2	A fundamental equality: $Vfrom$ does not require ordinals!	178
23.2	Basic Closure Properties	178
23.2.1	Finite sets and ordered pairs	179
23.3	0, Successor and Limit Equations for $Vfrom$	179
23.4	$Vfrom$ applied to Limit Ordinals	179
23.4.1	Closure under Disjoint Union	180
23.5	Properties assuming $Transset(A)$	180
23.5.1	Products	181
23.5.2	Disjoint Sums, or Quine Ordered Pairs	181
23.5.3	Function Space!	182
23.6	The Set $Vset(i)$	182
23.6.1	Characterisation of the elements of $Vset(i)$	182
23.6.2	Reasoning about Sets in Terms of Their Elements' Ranks	183
23.6.3	Set Up an Environment for Simplification	183

23.6.4	Recursion over Vset Levels!	183
23.7	The Datatype Universe: $univ(A)$	184
23.7.1	The Set $univ(A)$ as a Limit	184
23.8	Closure Properties for $univ(A)$	184
23.8.1	Closure under Unordered and Ordered Pairs	184
23.8.2	The Natural Numbers	185
23.8.3	Instances for 1 and 2	185
23.8.4	Closure under Disjoint Union	185
23.9	Finite Branching Closure Properties	186
23.9.1	Closure under Finite Powerset	186
23.9.2	Closure under Finite Powers: Functions from a Natural Number	186
23.9.3	Closure under Finite Function Space	186
23.10*	For QUniv. Properties of Vfrom analogous to the "take-lemma" *	187
24	QUniv: A Small Universe for Lazy Recursive Types	187
24.1	Properties involving Transset and Sum	188
24.2	Introduction and Elimination Rules	188
24.3	Closure Properties	188
24.4	Quine Disjoint Sum	189
24.5	Closure for Quine-Inspired Products and Sums	189
24.6	Quine Disjoint Sum	190
24.7	The Natural Numbers	190
24.8	"Take-Lemma" Rules	190
25	Datatype-ZF: Datatype and CoDatatype Definitions	191
26	Arith: Arithmetic Operators and Their Definitions	191
26.1	<i>natify</i> , the Coercion to <i>nat</i>	192
26.2	Typing rules	194
26.3	Addition	195
26.4	Monotonicity of Addition	196
26.5	Multiplication	198
27	ArithSimp: Arithmetic with simplification	200
27.1	Difference	200
27.2	Remainder	200
27.3	Division	201
27.4	Further Facts about Remainder	202
27.5	Additional theorems about \leq	203
27.6	Cancellation Laws for Common Factors in Comparisons	203
27.7	More Lemmas about Remainder	204
27.7.1	More Lemmas About Difference	205

28 List-ZF: Lists in Zermelo-Fraenkel Set Theory	206
28.1 The function zip	220
29 EquivClass: Equivalence Relations	225
29.1 Suppes, Theorem 70: r is an equiv relation iff $\text{converse}(r) \circ r = r$	226
29.2 Defining Unary Operations upon Equivalence Classes	227
29.3 Defining Binary Operations upon Equivalence Classes	228
30 Int-ZF: The Integers as Equivalence Classes Over Pairs of Natural Numbers	229
30.1 Proving that intrel is an equivalence relation	231
30.2 Collapsing rules: to remove intify from arithmetic expressions	232
30.3 zminus : unary negation on int	233
30.4 znegative : the test for negative integers	234
30.5 nat-of : Coercion of an Integer to a Natural Number	234
30.6 zmagnitude : magnitude of an integer, as a natural number . .	235
30.7 $\text{op } \$+$: addition on int	236
30.8 $\text{op } \$\times$: Integer Multiplication	237
30.9 The "Less Than" Relation	240
30.10 Less Than or Equals	241
30.11 More subtraction laws (for zcompare-rls)	242
30.12 Monotonicity and Cancellation Results for Instantiation of the CancelNumerals Simprocs	242
30.13 Comparison laws	243
30.13.1 More inequality lemmas	244
30.13.2 The next several equations are permutative: watch out!	244
31 Bin: Arithmetic on Binary Integers	244
31.0.3 The Carry and Borrow Functions, bin-succ and bin-pred	247
31.0.4 bin-minus : Unary Negation of Binary Integers	247
31.0.5 bin-add : Binary Addition	247
31.0.6 bin-mult : Binary Multiplication	248
31.1 Computations	248
31.2 Simplification Rules for Comparison of Binary Numbers . . .	250
32 IntDiv-ZF: The Division Operators Div and Mod	256
32.1 Uniqueness and monotonicity of quotients and remainders . .	260
32.2 Correctness of posDivAlg , the Division Algorithm for $a \geq 0$ and $b > 0$	261
32.3 Some convenient biconditionals for products of signs	261
32.4 Correctness of negDivAlg , the division algorithm for $a \neq 0$ and $b \neq 0$	263
32.5 Existence shown by proving the division algorithm to be correct	264

32.6	division of a number by itself	267
32.7	Computation of division and remainder	268
32.8	Monotonicity in the first argument (divisor)	270
32.9	Monotonicity in the second argument (dividend)	270
32.10	More algebraic laws for zdiv and zmod	271
32.11	proving a zdiv (b*c) = (a zdiv b) zdiv c	273
32.12	Cancellation of common factors in "zdiv"	274
32.13	Distribution of factors over "zmod"	274
33	CardinalArith: Cardinal Arithmetic Without the Axiom of Choice	275
33.1	Cardinal addition	276
33.1.1	Cardinal addition is commutative	277
33.1.2	Cardinal addition is associative	277
33.1.3	0 is the identity for addition	277
33.1.4	Addition by another cardinal	277
33.1.5	Monotonicity of addition	277
33.1.6	Addition of finite cardinals is "ordinary" addition	278
33.2	Cardinal multiplication	278
33.2.1	Cardinal multiplication is commutative	278
33.2.2	Cardinal multiplication is associative	278
33.2.3	Cardinal multiplication distributes over addition	278
33.2.4	Multiplication by 0 yields 0	278
33.2.5	1 is the identity for multiplication	279
33.3	Some inequalities for multiplication	279
33.3.1	Multiplication by a non-zero cardinal	279
33.3.2	Monotonicity of multiplication	279
33.4	Multiplication of finite cardinals is "ordinary" multiplication	279
33.5	Infinite Cardinals are Limit Ordinals	280
33.5.1	Establishing the well-ordering	280
33.5.2	Characterising initial segments of the well-ordering	281
33.5.3	The cardinality of initial segments	281
33.5.4	Toward's Kunen's Corollary 10.13 (1)	282
33.6	For Every Cardinal Number There Exists A Greater One	282
33.7	Basic Properties of Successor Cardinals	283
33.7.1	Removing elements from a finite set decreases its cardinality	283
33.7.2	Theorems by Krzysztof Grabczewski, proofs by lcp	284
34	Main-ZF: Theory Main: Everything Except AC	284
34.1	Iteration of the function F	285
34.2	Transfinite Recursion	285
35	AC: The Axiom of Choice	286

36 Zorn: Zorn's Lemma	287
36.1 Mathematical Preamble	288
36.2 The Transfinite Construction	288
36.3 Some Properties of the Transfinite Construction	288
36.4 Hausdorff's Theorem: Every Set Contains a Maximal Chain .	289
36.5 Zorn's Lemma: If All Chains in S Have Upper Bounds In S, then S contains a Maximal Element	290
36.6 Zermelo's Theorem: Every Set can be Well-Ordered	291
36.7 Zorn's Lemma for Partial Orders	292
37 Cardinal-AC: Cardinal Arithmetic Using AC	292
37.1 Strengthened Forms of Existing Theorems on Cardinals . . .	292
37.2 The relationship between cardinality and le-pollence	293
37.3 Other Applications of AC	293
38 InfDatatype: Infinite-Branching Datatype Definitions	294



1 ZF: Zermelo-Fraenkel Set Theory

```
theory ZF
imports FOL
begin
```

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

```
typedecl i
arities i :: term
```

```
consts
```

```
zero      :: i          (0) — the empty set
Pow       :: i => i      — power sets
Inf       :: i          — infinite set
```

Bounded Quantifiers

```
consts
```

```
Ball     :: [i, i => o] => o
Bex      :: [i, i => o] => o
```

General Union and Intersection

```
consts
```

```
Union    :: i => i
Inter    :: i => i
```

Variations on Replacement

```
consts
```

```
PrimReplace :: [i, [i, i] => o] => i
Replace     :: [i, [i, i] => o] => i
RepFun      :: [i, i => i] => i
Collect     :: [i, i => o] => i
```

Definite descriptions – via Replace over the set "1"

```
consts
```

```
The       :: (i => o) => i    (binder THE 10)
If        :: [o, i, i] => i  ((if (-)/ then (-)/ else (-)) [10] 10)
```

```
abbreviation (input)
```

```
old-if    :: [o, i, i] => i  (if '(-,-,-')) where
if(P,a,b) == If(P,a,b)
```

Finite Sets

```
consts
```

```
Upair    :: [i, i] => i
cons     :: [i, i] => i
succ     :: i => i
```

Ordered Pairing

consts

Pair :: $[i, i] \Rightarrow i$
fst :: $i \Rightarrow i$
snd :: $i \Rightarrow i$
split :: $[[i, i] \Rightarrow 'a, i] \Rightarrow 'a::\{\}$ — for pattern-matching

Sigma and Pi Operators

consts

Sigma :: $[i, i \Rightarrow i] \Rightarrow i$
Pi :: $[i, i \Rightarrow i] \Rightarrow i$

Relations and Functions

consts

domain :: $i \Rightarrow i$
range :: $i \Rightarrow i$
field :: $i \Rightarrow i$
converse :: $i \Rightarrow i$
relation :: $i \Rightarrow o$ — recognizes sets of pairs
function :: $i \Rightarrow o$ — recognizes functions; can have non-pairs
Lambda :: $[i, i \Rightarrow i] \Rightarrow i$
restrict :: $[i, i] \Rightarrow i$

Infixes in order of decreasing precedence

consts

Image :: $[i, i] \Rightarrow i$ (**infixl** “ 90) — image
vimage :: $[i, i] \Rightarrow i$ (**infixl** -“ 90) — inverse image
apply :: $[i, i] \Rightarrow i$ (**infixl** ‘ 90) — function application
Int :: $[i, i] \Rightarrow i$ (**infixl** *Int* 70) — binary intersection
Un :: $[i, i] \Rightarrow i$ (**infixl** *Un* 65) — binary union
Diff :: $[i, i] \Rightarrow i$ (**infixl** - 65) — set difference
Subset :: $[i, i] \Rightarrow o$ (**infixl** <= 50) — subset relation
mem :: $[i, i] \Rightarrow o$ (**infixl** : 50) — membership relation

abbreviation

not-mem :: $[i, i] \Rightarrow o$ (**infixl** ~: 50) — negated membership relation
where $x \sim: y == \sim (x : y)$

abbreviation

cart-prod :: $[i, i] \Rightarrow i$ (**infixr** * 80) — Cartesian product
where $A * B == \text{Sigma}(A, \%-. B)$

abbreviation

function-space :: $[i, i] \Rightarrow i$ (**infixr** -> 60) — function space
where $A -> B == \text{Pi}(A, \%-. B)$

nonterminal *is* and *patterns*

syntax

```

:: i => is (-)
-Enum :: [i, is] => is (-, / -)

-Finset :: is => i ({(-)})
-Tuple :: [i, is] => i (<(-, / -)>)
-Collect :: [pttrn, i, o] => i ((1{- . / - -})
-Replace :: [pttrn, pttrn, i, o] => i ((1{- . / - -})
-RepFun :: [i, pttrn, i] => i ((1{- . / - -}) [51,0,51])
-INTER :: [pttrn, i, i] => i ((3INT -.- / -) 10)
-UNION :: [pttrn, i, i] => i ((3UN -.- / -) 10)
-PROD :: [pttrn, i, i] => i ((3PROD -.- / -) 10)
-SUM :: [pttrn, i, i] => i ((3SUM -.- / -) 10)
-lam :: [pttrn, i, i] => i ((3lam -.- / -) 10)
-Ball :: [pttrn, i, o] => o ((3ALL -.- / -) 10)
-Bex :: [pttrn, i, o] => o ((3EX -.- / -) 10)

-pattern :: patterns => pttrn (<->)
:: pttrn => patterns (-)
-patterns :: [pttrn, patterns] => patterns (-, / -)

```

translations

```

{x, xs} == CONST cons(x, {xs})
{x} == CONST cons(x, 0)
{x:A. P} == CONST Collect(A, %x. P)
{y. x:A. Q} == CONST Replace(A, %x y. Q)
{b. x:A} == CONST RepFun(A, %x. b)
INT x:A. B == CONST Inter({B. x:A})
UN x:A. B == CONST Union({B. x:A})
PROD x:A. B == CONST Pi(A, %x. B)
SUM x:A. B == CONST Sigma(A, %x. B)
lam x:A. f == CONST Lambda(A, %x. f)
ALL x:A. P == CONST Ball(A, %x. P)
EX x:A. P == CONST Bex(A, %x. P)

<x, y, z> == <x, <y, z>>
<x, y> == CONST Pair(x, y)
%<x,y,zs>.b == CONST split(%x <y,zs>.b)
%<x,y>.b == CONST split(%x y. b)

```

notation (*xsymbols*)

```

cart-prod (infixr × 80) and
Int (infixl ∩ 70) and
Un (infixl ∪ 65) and

```

function-space (**infixr** \rightarrow 60) and
Subset (**infixl** \subseteq 50) and
mem (**infixl** \in 50) and
not-mem (**infixl** \notin 50) and
Union (**U**- [90] 90) and
Inter (**∩**- [90] 90)

syntax (*xsymbols*)

-Collect :: [pttrn, i, o] => i ((1{- ∈ - ./ -})
-Replace :: [pttrn, pttrn, i, o] => i ((1{- ./ - ∈ -, -})
-RepFun :: [i, pttrn, i] => i ((1{- ./ - ∈ -}) [51,0,51])
-UNION :: [pttrn, i, i] => i ((3∪-∈-./ -) 10)
-INTER :: [pttrn, i, i] => i ((3∩-∈-./ -) 10)
-PROD :: [pttrn, i, i] => i ((3Π-∈-./ -) 10)
-SUM :: [pttrn, i, i] => i ((3Σ-∈-./ -) 10)
-lam :: [pttrn, i, i] => i ((3λ-∈-./ -) 10)
-Ball :: [pttrn, i, o] => o ((3∀-∈-./ -) 10)
-Bex :: [pttrn, i, o] => o ((3∃-∈-./ -) 10)
-Tuple :: [i, is] => i ((-./ -))
-pattern :: patterns => pttrn ((-))

notation (*HTML output*)

cart-prod (**infixr** \times 80) and
Int (**infixl** \cap 70) and
Un (**infixl** \cup 65) and
Subset (**infixl** \subseteq 50) and
mem (**infixl** \in 50) and
not-mem (**infixl** \notin 50) and
Union (**U**- [90] 90) and
Inter (**∩**- [90] 90)

syntax (*HTML output*)

-Collect :: [pttrn, i, o] => i ((1{- ∈ - ./ -})
-Replace :: [pttrn, pttrn, i, o] => i ((1{- ./ - ∈ -, -})
-RepFun :: [i, pttrn, i] => i ((1{- ./ - ∈ -}) [51,0,51])
-UNION :: [pttrn, i, i] => i ((3∪-∈-./ -) 10)
-INTER :: [pttrn, i, i] => i ((3∩-∈-./ -) 10)
-PROD :: [pttrn, i, i] => i ((3Π-∈-./ -) 10)
-SUM :: [pttrn, i, i] => i ((3Σ-∈-./ -) 10)
-lam :: [pttrn, i, i] => i ((3λ-∈-./ -) 10)
-Ball :: [pttrn, i, o] => o ((3∀-∈-./ -) 10)
-Bex :: [pttrn, i, o] => o ((3∃-∈-./ -) 10)
-Tuple :: [i, is] => i ((-./ -))
-pattern :: patterns => pttrn ((-))

finalconsts

0 Pow Inf Union PrimReplace mem

defs

Ball-def: $Ball(A, P) == \forall x. x \in A \rightarrow P(x)$

Bex-def: $Bex(A, P) == \exists x. x \in A \ \& \ P(x)$

subset-def: $A \leq B == \forall x \in A. x \in B$

axiomatization where

extension: $A = B \leftrightarrow A \leq B \ \& \ B \leq A$ **and**

Union-iff: $A \in Union(C) \leftrightarrow (\exists B \in C. A \in B)$ **and**

Pow-iff: $A \in Pow(B) \leftrightarrow A \leq B$ **and**

infinity: $0 \in Inf \ \& \ (\forall y \in Inf. succ(y) \in Inf)$ **and**

foundation: $A = 0 \mid (\exists x \in A. \forall y \in x. y \sim A)$ **and**

replacement: $(\forall x \in A. \forall y \ z. P(x, y) \ \& \ P(x, z) \rightarrow y = z) \implies$
 $b \in PrimReplace(A, P) \leftrightarrow (\exists x \in A. P(x, b))$

defs

Replace-def: $Replace(A, P) == PrimReplace(A, \lambda x y. (EX!z. P(x, z)) \ \& \ P(x, y))$

RepFun-def: $RepFun(A, f) == \{y \mid x \in A, y = f(x)\}$

Collect-def: $Collect(A, P) == \{y \mid x \in A, x = y \ \& \ P(x)\}$

Upair-def: $Upair(a, b) == \{y \mid x \in Pow(Pow(0)), (x = 0 \ \& \ y = a) \mid (x = Pow(0) \ \& \ y = b)\}$

cons-def: $cons(a, A) == Upair(a, a) \cup A$

succ-def: $succ(i) == cons(i, i)$

Diff-def: $A - B == \{ x \in A . \sim(x \in B) \}$
Inter-def: $Inter(A) == \{ x \in Union(A) . \forall y \in A. x \in y \}$
Un-def: $A Un B == Union(Upair(A,B))$
Int-def: $A Int B == Inter(Upair(A,B))$

the-def: $The(P) == Union(\{y . x \in \{0\}, P(y)\})$
if-def: $if(P,a,b) == THE z. P \& z=a \mid \sim P \& z=b$

Pair-def: $\langle a,b \rangle == \{\{a,a\}, \{a,b\}\}$
fst-def: $fst(p) == THE a. \exists b. p = \langle a,b \rangle$
snd-def: $snd(p) == THE b. \exists a. p = \langle a,b \rangle$
split-def: $split(c) == \%p. c(fst(p), snd(p))$
Sigma-def: $Sigma(A,B) == \bigcup x \in A. \bigcup y \in B(x). \{\langle x,y \rangle\}$

converse-def: $converse(r) == \{z. w \in r, \exists x y. w = \langle x,y \rangle \& z = \langle y,x \rangle\}$

domain-def: $domain(r) == \{x. w \in r, \exists y. w = \langle x,y \rangle\}$
range-def: $range(r) == domain(converse(r))$
field-def: $field(r) == domain(r) Un range(r)$
relation-def: $relation(r) == \forall z \in r. \exists x y. z = \langle x,y \rangle$
function-def: $function(r) ==$
 $\quad \forall x y. \langle x,y \rangle : r \dashrightarrow (\forall y'. \langle x,y' \rangle : r \dashrightarrow y=y')$
image-def: $r \text{ `` } A == \{y : range(r) . \exists x \in A. \langle x,y \rangle : r\}$
vimage-def: $r \text{ - `` } A == converse(r) \text{ `` } A$

lam-def: $Lambda(A,b) == \{\langle x,b(x) \rangle . x \in A\}$
apply-def: $f \text{ `` } a == Union(f \text{ `` } \{a\})$
Pi-def: $Pi(A,B) == \{f \in Pow(Sigma(A,B)). A \leq domain(f) \& function(f)\}$

restrict-def: $restrict(r,A) == \{z : r. \exists x \in A. \exists y. z = \langle x,y \rangle\}$

1.1 Substitution

lemma *subst-elim:* $[\![b \in A; a=b]\!] ==> a \in A$
<proof>

1.2 Bounded universal quantifier

lemma *ball* [*intro!*]: $[\![!!x. x \in A ==> P(x)]\!] ==> \forall x \in A. P(x)$
<proof>

lemmas *strip = impI allI ballI*

lemma *bspec* [*dest?*]: $[[\forall x \in A. P(x); x: A]] \implies P(x)$
<proof>

lemma *rev-ballE* [*elim*]:
 $[[\forall x \in A. P(x); x \sim : A \implies Q; P(x) \implies Q]] \implies Q$
<proof>

lemma *ballE*: $[[\forall x \in A. P(x); P(x) \implies Q; x \sim : A \implies Q]] \implies Q$
<proof>

lemma *rev-bspec*: $[[x: A; \forall x \in A. P(x)]] \implies P(x)$
<proof>

lemma *ball-triv* [*simp*]: $(\forall x \in A. P) \longleftrightarrow ((\exists x. x \in A) \dashrightarrow P)$
<proof>

lemma *ball-cong* [*cong*]:
 $[[A=A'; !!x. x \in A' \implies P(x) \longleftrightarrow P'(x)]] \implies (\forall x \in A. P(x)) \longleftrightarrow (\forall x \in A'. P'(x))$
<proof>

lemma *atomize-ball*:
 $(!!x. x \in A \implies P(x)) == \text{Trueprop } (\forall x \in A. P(x))$
<proof>

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

1.3 Bounded existential quantifier

lemma *bexI* [*intro*]: $[[P(x); x: A]] \implies \exists x \in A. P(x)$
<proof>

lemma *rev-bexI*: $[[x \in A; P(x)]] \implies \exists x \in A. P(x)$
<proof>

lemma *bexCI*: $[[\forall x \in A. \sim P(x) \implies P(a); a: A]] \implies \exists x \in A. P(x)$
<proof>

lemma *bexE* [*elim!*]: $[[\exists x \in A. P(x); !!x. [[x \in A; P(x)]] \implies Q]] \implies Q$
<proof>

lemma *bex-triv* [*simp*]: $(\exists x \in A. P) \leftrightarrow ((\exists x. x \in A) \& P)$
<proof>

lemma *bex-cong* [*cong*]:
[[$A=A'$; $\forall x. x \in A' \implies P(x) \leftrightarrow P'(x)$]]
 $\implies (\exists x \in A. P(x)) \leftrightarrow (\exists x \in A'. P'(x))$
<proof>

1.4 Rules for subsets

lemma *subsetI* [*intro!*]:
 $(\forall x. x \in A \implies x \in B) \implies A \leq B$
<proof>

lemma *subsetD* [*elim*]: [[$A \leq B$; $c \in A$]] $\implies c \in B$
<proof>

lemma *subsetCE* [*elim*]:
[[$A \leq B$; $c \sim A \implies P$; $c \in B \implies P$]] $\implies P$
<proof>

lemma *rev-subsetD*: [[$c \in A$; $A \leq B$]] $\implies c \in B$
<proof>

lemma *contra-subsetD*: [[$A \leq B$; $c \sim B$]] $\implies c \sim A$
<proof>

lemma *rev-contra-subsetD*: [[$c \sim B$; $A \leq B$]] $\implies c \sim A$
<proof>

lemma *subset-refl* [*simp*]: $A \leq A$
<proof>

lemma *subset-trans*: [[$A \leq B$; $B \leq C$]] $\implies A \leq C$
<proof>

lemma *subset-iff*:
 $A \leq B \leftrightarrow (\forall x. x \in A \implies x \in B)$
<proof>

1.5 Rules for equality

lemma *equalityI* [*intro*]: [[$A \leq B$; $B \leq A$]] $\implies A = B$
<proof>

lemma *equality-iffI*: $(!!x. x \in A \leftrightarrow x \in B) \implies A = B$
 $\langle proof \rangle$

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

lemma *equalityE*: $[A = B; [A \leq B; B \leq A]] \implies P \implies P$
 $\langle proof \rangle$

lemma *equalityCE*:
 $[A = B; [c \in A; c \in B]] \implies P; [c \sim A; c \sim B] \implies P \implies P$
 $\langle proof \rangle$

lemma *equality-iffD*:
 $A = B \implies (!!x. x : A \leftrightarrow x : B)$
 $\langle proof \rangle$

1.6 Rules for Replace – the derived form of replacement

lemma *Replace-iff*:
 $b : \{y. x \in A, P(x,y)\} \leftrightarrow (\exists x \in A. P(x,b) \ \& \ (\forall y. P(x,y) \dashrightarrow y=b))$
 $\langle proof \rangle$

lemma *ReplaceI* [*intro*]:
 $[P(x,b); x : A; !!y. P(x,y) \implies y=b] \implies$
 $b : \{y. x \in A, P(x,y)\}$
 $\langle proof \rangle$

lemma *ReplaceE*:
 $[b : \{y. x \in A, P(x,y)\};$
 $!!x. [x : A; P(x,b); \forall y. P(x,y) \dashrightarrow y=b] \implies R$
 $] \implies R$
 $\langle proof \rangle$

lemma *ReplaceE2* [*elim!*]:
 $[b : \{y. x \in A, P(x,y)\};$
 $!!x. [x : A; P(x,b)] \implies R$
 $] \implies R$
 $\langle proof \rangle$

lemma *Replace-cong* [*cong*]:
 $[A=B; !!x y. x \in B \implies P(x,y) \leftrightarrow Q(x,y)] \implies$
 $Replace(A,P) = Replace(B,Q)$
 $\langle proof \rangle$

1.7 Rules for RepFun

lemma *RepFunI*: $a \in A \implies f(a) : \{f(x). x \in A\}$
<proof>

lemma *RepFun-eqI* [*intro*]: $[\![b=f(a); a \in A \!]\!] \implies b : \{f(x). x \in A\}$
<proof>

lemma *RepFunE* [*elim!*]:
 $[\![b : \{f(x). x \in A\};$
 $!!x. [\![x \in A; b=f(x) \!]\!] \implies P \!]\!] \implies$
 P
<proof>

lemma *RepFun-cong* [*cong*]:
 $[\![A=B; !!x. x \in B \implies f(x)=g(x) \!]\!] \implies \text{RepFun}(A,f) = \text{RepFun}(B,g)$
<proof>

lemma *RepFun-iff* [*simp*]: $b : \{f(x). x \in A\} \iff (\exists x \in A. b=f(x))$
<proof>

lemma *triv-RepFun* [*simp*]: $\{x. x \in A\} = A$
<proof>

1.8 Rules for Collect – forming a subset by separation

lemma *separation* [*simp*]: $a : \{x \in A. P(x)\} \iff a \in A \ \& \ P(a)$
<proof>

lemma *CollectI* [*intro!*]: $[\![a \in A; P(a) \!]\!] \implies a : \{x \in A. P(x)\}$
<proof>

lemma *CollectE* [*elim!*]: $[\![a : \{x \in A. P(x)\}; [\![a \in A; P(a) \!]\!] \implies R \!]\!] \implies R$
<proof>

lemma *CollectD1*: $a : \{x \in A. P(x)\} \implies a \in A$
<proof>

lemma *CollectD2*: $a : \{x \in A. P(x)\} \implies P(a)$
<proof>

lemma *Collect-cong* [*cong*]:
 $[\![A=B; !!x. x \in B \implies P(x) \iff Q(x) \!]\!] \implies$
 $\text{Collect}(A, \%x. P(x)) = \text{Collect}(B, \%x. Q(x))$
<proof>

1.9 Rules for Unions

declare *Union-iff* [*simp*]

lemma *UnionI* [*intro*]: $[\![B : C; A : B]\!] \implies A : \text{Union}(C)$
<proof>

lemma *UnionE* [*elim!*]: $[\![A \in \text{Union}(C); !!B. [\![A : B; B : C]\!] \implies R]\!] \implies R$
<proof>

1.10 Rules for Unions of families

lemma *UN-iff* [*simp*]: $b : (\bigcup x \in A. B(x)) \iff (\exists x \in A. b \in B(x))$
<proof>

lemma *UN-I*: $[\![a : A; b : B(a)]\!] \implies b : (\bigcup x \in A. B(x))$
<proof>

lemma *UN-E* [*elim!*]:
 $[\![b : (\bigcup x \in A. B(x)); !!x. [\![x : A; b : B(x)]\!] \implies R]\!] \implies R$
<proof>

lemma *UN-cong*:
 $[\![A=B; !!x. x \in B \implies C(x)=D(x)]\!] \implies (\bigcup x \in A. C(x)) = (\bigcup x \in B. D(x))$
<proof>

1.11 Rules for the empty set

lemma *not-mem-empty* [*simp*]: $a \sim : 0$
<proof>

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

lemma *empty-subsetI* [*simp*]: $0 \leq A$
<proof>

lemma *equals0I*: $[\![!!y. y \in A \implies \text{False}]\!] \implies A=0$
<proof>

lemma *equals0D* [*dest*]: $A=0 \implies a \sim : A$
<proof>

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

lemma *not-emptyI*: $a \in A \implies A \sim = 0$
<proof>

lemma *not-emptyE*: $[\![A \sim = 0; !!x. x \in A \implies R]\!] \implies R$

$\langle proof \rangle$

1.12 Rules for Inter

lemma *Inter-iff*: $A \in Inter(C) \leftrightarrow (\forall x \in C. A: x) \ \& \ C \neq 0$
 $\langle proof \rangle$

lemma *InterI* [*intro!*]:
 $[\![\ !x. x: C \implies A: x; \ C \neq 0 \]\!] \implies A \in Inter(C)$
 $\langle proof \rangle$

lemma *InterD* [*elim, Pure.elim*]: $[\![A \in Inter(C); \ B \in C \]\!] \implies A \in B$
 $\langle proof \rangle$

lemma *InterE* [*elim*]:
 $[\![A \in Inter(C); \ B \sim C \implies R; \ A \in B \implies R \]\!] \implies R$
 $\langle proof \rangle$

1.13 Rules for Intersections of families

lemma *INT-iff*: $b : (\bigcap x \in A. B(x)) \leftrightarrow (\forall x \in A. b \in B(x)) \ \& \ A \neq 0$
 $\langle proof \rangle$

lemma *INT-I*: $[\![\ !x. x: A \implies b: B(x); \ A \neq 0 \]\!] \implies b : (\bigcap x \in A. B(x))$
 $\langle proof \rangle$

lemma *INT-E*: $[\![b : (\bigcap x \in A. B(x)); \ a: A \]\!] \implies b \in B(a)$
 $\langle proof \rangle$

lemma *INT-cong*:
 $[\![A=B; \ !x. x \in B \implies C(x)=D(x) \]\!] \implies (\bigcap x \in A. C(x)) = (\bigcap x \in B. D(x))$
 $\langle proof \rangle$

1.14 Rules for Powersets

lemma *PowI*: $A \leq B \implies A \in Pow(B)$
 $\langle proof \rangle$

lemma *PowD*: $A \in Pow(B) \implies A \leq B$
 $\langle proof \rangle$

declare *Pow-iff* [*iff*]

lemmas *Pow-bottom = empty-subsetI* [*THEN PowI*]

lemmas *Pow-top = subset-refl* [*THEN PowI*]

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

lemma *cantor*: $\exists S \in Pow(A). \forall x \in A. b(x) \sim S$
<proof>

<ML>

end

2 upair: Unordered Pairs

theory *upair* **imports** *ZF*
uses *Tools/typechk.ML* **begin**

<ML>

lemma *atomize-ball* [*symmetric, rulify*]:
 $(\forall x. x:A \implies P(x)) \implies Trueprop (ALL x:A. P(x))$
<proof>

2.1 Unordered Pairs: constant *Upair*

lemma *Upair-iff* [*simp*]: $c : Upair(a,b) \iff (c=a \mid c=b)$
<proof>

lemma *UpairI1*: $a : Upair(a,b)$
<proof>

lemma *UpairI2*: $b : Upair(a,b)$
<proof>

lemma *UpairE*: $[\mid a : Upair(b,c); a=b \implies P; a=c \implies P \mid] \implies P$
<proof>

2.2 Rules for Binary Union, Defined via *Upair*

lemma *Un-iff* [*simp*]: $c : A \cup B \iff (c:A \mid c:B)$
<proof>

lemma *UnI1*: $c : A \implies c : A \cup B$
<proof>

lemma *UnI2*: $c : B \implies c : A \cup B$
<proof>

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

lemma *UnE* [*elim!*]: $[[c : A \text{ Un } B; c:A \implies P; c:B \implies P]] \implies P$
<proof>

lemma *UnE'*: $[[c : A \text{ Un } B; c:A \implies P; [[c:B; c\sim:A]] \implies P]] \implies P$
<proof>

lemma *UnCI* [*intro!*]: $(c \sim: B \implies c : A) \implies c : A \text{ Un } B$
<proof>

2.3 Rules for Binary Intersection, Defined via *Upair*

lemma *Int-iff* [*simp*]: $c : A \text{ Int } B \iff (c:A \ \& \ c:B)$
<proof>

lemma *IntI* [*intro!*]: $[[c : A; c : B]] \implies c : A \text{ Int } B$
<proof>

lemma *IntD1*: $c : A \text{ Int } B \implies c : A$
<proof>

lemma *IntD2*: $c : A \text{ Int } B \implies c : B$
<proof>

lemma *IntE* [*elim!*]: $[[c : A \text{ Int } B; [[c:A; c:B]] \implies P]] \implies P$
<proof>

2.4 Rules for Set Difference, Defined via *Upair*

lemma *Diff-iff* [*simp*]: $c : A - B \iff (c:A \ \& \ c\sim:B)$
<proof>

lemma *DiffI* [*intro!*]: $[[c : A; c \sim: B]] \implies c : A - B$
<proof>

lemma *DiffD1*: $c : A - B \implies c : A$
<proof>

lemma *DiffD2*: $c : A - B \implies c \sim: B$
<proof>

lemma *DiffE* [*elim!*]: $[[c : A - B; [[c:A; c\sim:B]] \implies P]] \implies P$
<proof>

2.5 Rules for *cons*

lemma *cons-iff* [*simp*]: $a : \text{cons}(b,A) \iff (a=b \mid a:A)$

$\langle proof \rangle$

lemma *consI1* [*simp, TC*]: $a : cons(a, B)$
 $\langle proof \rangle$

lemma *consI2*: $a : B ==> a : cons(b, B)$
 $\langle proof \rangle$

lemma *consE* [*elim!*]: $[| a : cons(b, A); a=b ==> P; a:A ==> P |] ==> P$
 $\langle proof \rangle$

lemma *consE'*:
 $[| a : cons(b, A); a=b ==> P; [| a:A; a\sim=b |] ==> P |] ==> P$
 $\langle proof \rangle$

lemma *consCI* [*intro!*]: $(a\sim:B ==> a=b) ==> a : cons(b, B)$
 $\langle proof \rangle$

lemma *cons-not-0* [*simp*]: $cons(a, B) \sim = 0$
 $\langle proof \rangle$

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

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

2.6 Singletons

lemma *singleton-iff*: $a : \{b\} <-> a=b$
 $\langle proof \rangle$

lemma *singletonI* [*intro!*]: $a : \{a\}$
 $\langle proof \rangle$

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

2.7 Descriptions

lemma *the-equality* [*intro*]:
 $[| P(a); !!x. P(x) ==> x=a |] ==> (THE x. P(x)) = a$
 $\langle proof \rangle$

lemma *the-equality2*: $[| EX! x. P(x); P(a) |] ==> (THE x. P(x)) = a$
 $\langle proof \rangle$

lemma *theI*: $EX! x. P(x) ==> P(THE x. P(x))$

$\langle proof \rangle$

lemma *the-0*: $\sim (EX! x. P(x)) \implies (THE x. P(x))=0$
 $\langle proof \rangle$

lemma *theI2*:
 assumes *p1*: $\sim Q(0) \implies EX! x. P(x)$
 and *p2*: $!!x. P(x) \implies Q(x)$
 shows $Q(THE x. P(x))$
 $\langle proof \rangle$

lemma *the-eq-trivial* [*simp*]: $(THE x. x = a) = a$
 $\langle proof \rangle$

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

2.8 Conditional Terms: *if-then-else*

lemma *if-true* [*simp*]: $(if True then a else b) = a$
 $\langle proof \rangle$

lemma *if-false* [*simp*]: $(if False then a else b) = b$
 $\langle proof \rangle$

lemma *if-cong*:
 [[$P \leftrightarrow Q$; $Q \implies a=c$; $\sim Q \implies b=d$]]
 $\implies (if P then a else b) = (if Q then c else d)$
 $\langle proof \rangle$

lemma *if-weak-cong*: $P \leftrightarrow Q \implies (if P then x else y) = (if Q then x else y)$
 $\langle proof \rangle$

lemma *if-P*: $P \implies (if P then a else b) = a$
 $\langle proof \rangle$

lemma *if-not-P*: $\sim P \implies (if P then a else b) = b$
 $\langle proof \rangle$

lemma *split-if* [*split*]:
 $P(if Q then x else y) \leftrightarrow ((Q \longrightarrow P(x)) \& (\sim Q \longrightarrow P(y)))$

<proof>

lemmas *split-if-eq1* = *split-if* [of %x. x = b, standard]

lemmas *split-if-eq2* = *split-if* [of %x. a = x, standard]

lemmas *split-if-mem1* = *split-if* [of %x. x : b, standard]

lemmas *split-if-mem2* = *split-if* [of %x. a : x, standard]

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) <-> P & a:x | ~P & a:y

<proof>

lemma *if-type* [TC]:

[| P ==> a: A; ~P ==> b: A |] ==> (if P then a else b): A

<proof>

lemma *split-if-asm*: P(if Q then x else y) <-> (~((Q & ~P(x)) | (~Q & ~P(y))))

<proof>

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

2.9 Consequences of Foundation

lemma *mem-asym*: [| a:b; ~P ==> b:a |] ==> P

<proof>

lemma *mem-irrefl*: a:a ==> P

<proof>

lemma *mem-not-refl*: a ~: a

<proof>

lemma *mem-imp-not-eq*: a:A ==> a ~ = A

<proof>

lemma *eq-imp-not-mem*: a=A ==> a ~: A

<proof>

2.10 Rules for Successor

lemma *succ-iff*: $i : \text{succ}(j) \leftrightarrow i=j \mid i:j$
 $\langle \text{proof} \rangle$

lemma *succI1* [*simp*]: $i : \text{succ}(i)$
 $\langle \text{proof} \rangle$

lemma *succI2*: $i : j \implies i : \text{succ}(j)$
 $\langle \text{proof} \rangle$

lemma *succE* [*elim!*]:
 $\llbracket i : \text{succ}(j); i=j \implies P; i:j \implies P \rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *succCI* [*intro!*]: $(i \sim : j \implies i=j) \implies i : \text{succ}(j)$
 $\langle \text{proof} \rangle$

lemma *succ-not-0* [*simp*]: $\text{succ}(n) \sim = 0$
 $\langle \text{proof} \rangle$

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

lemma *succ-inject-iff* [*simp*]: $\text{succ}(m) = \text{succ}(n) \leftrightarrow m=n$
 $\langle \text{proof} \rangle$

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

2.11 Miniscoping of the Bounded Universal Quantifier

lemma *ball-simps1*:

$(\text{ALL } x:A. P(x) \ \& \ Q) \leftrightarrow (\text{ALL } x:A. P(x)) \ \& \ (A=0 \mid Q)$
 $(\text{ALL } x:A. P(x) \mid Q) \leftrightarrow ((\text{ALL } x:A. P(x)) \mid Q)$
 $(\text{ALL } x:A. P(x) \dashrightarrow Q) \leftrightarrow ((\text{EX } x:A. P(x)) \dashrightarrow Q)$
 $(\sim(\text{ALL } x:A. P(x))) \leftrightarrow (\text{EX } x:A. \sim P(x))$
 $(\text{ALL } x:0.P(x)) \leftrightarrow \text{True}$
 $(\text{ALL } x:\text{succ}(i).P(x)) \leftrightarrow P(i) \ \& \ (\text{ALL } x:i. P(x))$
 $(\text{ALL } x:\text{cons}(a,B).P(x)) \leftrightarrow P(a) \ \& \ (\text{ALL } x:B. P(x))$
 $(\text{ALL } x:\text{RepFun}(A,f).P(x)) \leftrightarrow (\text{ALL } y:A. P(f(y)))$

$(\text{ALL } x:\text{Union}(A).P(x)) \leftrightarrow (\text{ALL } y:A. \text{ALL } x:y. P(x))$
 ⟨proof⟩

lemma *ball-simps2*:

$(\text{ALL } x:A. P \ \& \ Q(x)) \leftrightarrow (A=0 \mid P) \ \& \ (\text{ALL } x:A. Q(x))$
 $(\text{ALL } x:A. P \mid Q(x)) \leftrightarrow (P \mid (\text{ALL } x:A. Q(x)))$
 $(\text{ALL } x:A. P \dashrightarrow Q(x)) \leftrightarrow (P \dashrightarrow (\text{ALL } x:A. Q(x)))$
 ⟨proof⟩

lemma *ball-simps3*:

$(\text{ALL } x:\text{Collect}(A,Q).P(x)) \leftrightarrow (\text{ALL } x:A. Q(x) \dashrightarrow P(x))$
 ⟨proof⟩

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

lemma *ball-conj-distrib*:

$(\text{ALL } x:A. P(x) \ \& \ Q(x)) \leftrightarrow ((\text{ALL } x:A. P(x)) \ \& \ (\text{ALL } x:A. Q(x)))$
 ⟨proof⟩

2.12 Miniscoping of the Bounded Existential Quantifier

lemma *bex-simps1*:

$(\text{EX } x:A. P(x) \ \& \ Q) \leftrightarrow ((\text{EX } x:A. P(x)) \ \& \ Q)$
 $(\text{EX } x:A. P(x) \mid Q) \leftrightarrow (\text{EX } x:A. P(x)) \mid (A^{\sim}=0 \ \& \ Q)$
 $(\text{EX } x:A. P(x) \dashrightarrow Q) \leftrightarrow ((\text{ALL } x:A. P(x)) \dashrightarrow (A^{\sim}=0 \ \& \ Q))$
 $(\text{EX } x:0.P(x)) \leftrightarrow \text{False}$
 $(\text{EX } x:\text{succ}(i).P(x)) \leftrightarrow P(i) \mid (\text{EX } x:i. P(x))$
 $(\text{EX } x:\text{cons}(a,B).P(x)) \leftrightarrow P(a) \mid (\text{EX } x:B. P(x))$
 $(\text{EX } x:\text{RepFun}(A,f). P(x)) \leftrightarrow (\text{EX } y:A. P(f(y)))$
 $(\text{EX } x:\text{Union}(A).P(x)) \leftrightarrow (\text{EX } y:A. \text{EX } x:y. P(x))$
 $(\sim(\text{EX } x:A. P(x))) \leftrightarrow (\text{ALL } x:A. \sim P(x))$
 ⟨proof⟩

lemma *bex-simps2*:

$(\text{EX } x:A. P \ \& \ Q(x)) \leftrightarrow (P \ \& \ (\text{EX } x:A. Q(x)))$
 $(\text{EX } x:A. P \mid Q(x)) \leftrightarrow (A^{\sim}=0 \ \& \ P) \mid (\text{EX } x:A. Q(x))$
 $(\text{EX } x:A. P \dashrightarrow Q(x)) \leftrightarrow ((A=0 \mid P) \dashrightarrow (\text{EX } x:A. Q(x)))$
 ⟨proof⟩

lemma *bex-simps3*:

$(\text{EX } x:\text{Collect}(A,Q).P(x)) \leftrightarrow (\text{EX } x:A. Q(x) \ \& \ P(x))$
 ⟨proof⟩

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

lemma *bex-disj-distrib*:

$(\text{EX } x:A. P(x) \mid Q(x)) \leftrightarrow ((\text{EX } x:A. P(x)) \mid (\text{EX } x:A. Q(x)))$
 ⟨proof⟩

lemma *bex-triv-one-point1* [*simp*]: $(\exists x:A. x=a) \leftrightarrow (a:A)$
 ⟨*proof*⟩

lemma *bex-triv-one-point2* [*simp*]: $(\exists x:A. a=x) \leftrightarrow (a:A)$
 ⟨*proof*⟩

lemma *bex-one-point1* [*simp*]: $(\exists x:A. x=a \ \& \ P(x)) \leftrightarrow (a:A \ \& \ P(a))$
 ⟨*proof*⟩

lemma *bex-one-point2* [*simp*]: $(\exists x:A. a=x \ \& \ P(x)) \leftrightarrow (a:A \ \& \ P(a))$
 ⟨*proof*⟩

lemma *ball-one-point1* [*simp*]: $(\forall x:A. x=a \ \rightarrow \ P(x)) \leftrightarrow (a:A \ \rightarrow \ P(a))$
 ⟨*proof*⟩

lemma *ball-one-point2* [*simp*]: $(\forall x:A. a=x \ \rightarrow \ P(x)) \leftrightarrow (a:A \ \rightarrow \ P(a))$
 ⟨*proof*⟩

2.13 Miniscoping of the Replacement Operator

These cover both *Replace* and *Collect*

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

2.14 Miniscoping of Unions

lemma *UN-simps1*:
 $(\text{UN } x:C. \text{cons}(a, B(x))) = (\text{if } C=0 \text{ then } 0 \text{ else } \text{cons}(a, \text{UN } x:C. B(x)))$
 $(\text{UN } x:C. A(x) \ \text{Un } B') = (\text{if } C=0 \text{ then } 0 \text{ else } (\text{UN } x:C. A(x)) \ \text{Un } B')$
 $(\text{UN } x:C. A' \ \text{Un } B(x)) = (\text{if } C=0 \text{ then } 0 \text{ else } A' \ \text{Un } (\text{UN } x:C. B(x)))$
 $(\text{UN } x:C. A(x) \ \text{Int } B') = ((\text{UN } x:C. A(x)) \ \text{Int } B')$
 $(\text{UN } x:C. A' \ \text{Int } B(x)) = (A' \ \text{Int } (\text{UN } x:C. B(x)))$
 $(\text{UN } x:C. A(x) - B') = ((\text{UN } x:C. A(x)) - B')$
 $(\text{UN } x:C. A' - B(x)) = (\text{if } C=0 \text{ then } 0 \text{ else } A' - (\text{INT } x:C. B(x)))$
 ⟨*proof*⟩

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

$\langle \text{proof} \rangle$

lemmas $UN\text{-simps} [simp] = UN\text{-simps1 } UN\text{-simps2}$

Opposite of miniscoping: pull the operator out

lemma $UN\text{-extend-simps1}$:

$$\begin{aligned} (UN\ x:C. A(x))\ Un\ B &= (if\ C=0\ then\ B\ else\ (UN\ x:C. A(x)\ Un\ B)) \\ ((UN\ x:C. A(x))\ Int\ B) &= (UN\ x:C. A(x)\ Int\ B) \\ ((UN\ x:C. A(x))\ -\ B) &= (UN\ x:C. A(x)\ -\ B) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma $UN\text{-extend-simps2}$:

$$\begin{aligned} cons(a, UN\ x:C. B(x)) &= (if\ C=0\ then\ \{a\}\ else\ (UN\ x:C. cons(a, B(x)))) \\ A\ Un\ (UN\ x:C. B(x)) &= (if\ C=0\ then\ A\ else\ (UN\ x:C. A\ Un\ B(x))) \\ (A\ Int\ (UN\ x:C. B(x))) &= (UN\ x:C. A\ Int\ B(x)) \\ A\ -\ (INT\ x:C. B(x)) &= (if\ C=0\ then\ A\ else\ (UN\ x:C. A\ -\ B(x))) \\ (UN\ y:A. UN\ x:y. B(x)) &= (UN\ x:\ Union(A). B(x)) \\ (UN\ a:A. B(f(a))) &= (UN\ x:\ RepFun(A,f). B(x)) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma $UN\text{-UN-extend}$:

$$(UN\ x:A. UN\ z: B(x). C(z)) = (UN\ z: (UN\ x:A. B(x)). C(z))$$

$\langle \text{proof} \rangle$

lemmas $UN\text{-extend-simps} = UN\text{-extend-simps1 } UN\text{-extend-simps2 } UN\text{-UN-extend}$

2.15 Miniscoping of Intersections

lemma $INT\text{-simps1}$:

$$\begin{aligned} (INT\ x:C. A(x)\ Int\ B) &= (INT\ x:C. A(x))\ Int\ B \\ (INT\ x:C. A(x)\ -\ B) &= (INT\ x:C. A(x))\ -\ B \\ (INT\ x:C. A(x)\ Un\ B) &= (if\ C=0\ then\ 0\ else\ (INT\ x:C. A(x))\ Un\ B) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma $INT\text{-simps2}$:

$$\begin{aligned} (INT\ x:C. A\ Int\ B(x)) &= A\ Int\ (INT\ x:C. B(x)) \\ (INT\ x:C. A\ -\ B(x)) &= (if\ C=0\ then\ 0\ else\ A\ -\ (UN\ x:C. B(x))) \\ (INT\ x:C. cons(a, B(x))) &= (if\ C=0\ then\ 0\ else\ cons(a, INT\ x:C. B(x))) \\ (INT\ x:C. A\ Un\ B(x)) &= (if\ C=0\ then\ 0\ else\ A\ Un\ (INT\ x:C. B(x))) \end{aligned}$$

$\langle \text{proof} \rangle$

lemmas $INT\text{-simps} [simp] = INT\text{-simps1 } INT\text{-simps2}$

Opposite of miniscoping: pull the operator out

lemma $INT\text{-extend-simps1}$:

$$\begin{aligned} (INT\ x:C. A(x))\ Int\ B &= (INT\ x:C. A(x)\ Int\ B) \\ (INT\ x:C. A(x))\ -\ B &= (INT\ x:C. A(x)\ -\ B) \\ (INT\ x:C. A(x))\ Un\ B &= (if\ C=0\ then\ B\ else\ (INT\ x:C. A(x)\ Un\ B)) \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *INT-extend-simps2*:

$A \text{ Int } (INT\ x:C. B(x)) = (INT\ x:C. A \text{ Int } B(x))$
 $A - (UN\ x:C. B(x)) = (\text{if } C=0 \text{ then } A \text{ else } (INT\ x:C. A - B(x)))$
 $\text{cons}(a, INT\ x:C. B(x)) = (\text{if } C=0 \text{ then } \{a\} \text{ else } (INT\ x:C. \text{cons}(a, B(x))))$
 $A \text{ Un } (INT\ x:C. B(x)) = (\text{if } C=0 \text{ then } A \text{ else } (INT\ x:C. A \text{ Un } B(x)))$

<proof>

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

2.16 Other simprules

lemma *misc-simps* [*simp*]:

$0 \text{ Un } A = A$
 $A \text{ Un } 0 = A$
 $0 \text{ Int } A = 0$
 $A \text{ Int } 0 = 0$
 $0 - A = 0$
 $A - 0 = A$
 $\text{Union}(0) = 0$
 $\text{Union}(\text{cons}(b,A)) = b \text{ Un } \text{Union}(A)$
 $\text{Inter}(\{b\}) = b$

<proof>

end

3 pair: Ordered Pairs

theory *pair* **imports** *upair*

uses *simpdata.ML*

begin

<ML>

lemma *singleton-eq-iff* [*iff*]: $\{a\} = \{b\} \langle - \rangle a=b$

<proof>

lemma *doubleton-eq-iff*: $\{a,b\} = \{c,d\} \langle - \rangle (a=c \ \& \ b=d) \mid (a=d \ \& \ b=c)$

<proof>

lemma *Pair-iff* [*simp*]: $\langle a,b \rangle = \langle c,d \rangle \langle - \rangle a=c \ \& \ b=d$

<proof>

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

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

lemma *Pair-not-0*: $\langle a, b \rangle \sim = 0$
 $\langle proof \rangle$

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

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

lemma *Pair-neq-fst*: $\langle a, b \rangle = a \implies P$
 $\langle proof \rangle$

lemma *Pair-neq-snd*: $\langle a, b \rangle = b \implies P$
 $\langle proof \rangle$

3.1 Sigma: Disjoint Union of a Family of Sets

Generalizes Cartesian product

lemma *Sigma-iff* [*simp*]: $\langle a, b \rangle : \text{Sigma}(A, B) \iff a:A \ \& \ b:B(a)$
 $\langle proof \rangle$

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

lemmas *SigmaD1* = *Sigma-iff* [*THEN iffD1*, *THEN conjunct1*, *standard*]
lemmas *SigmaD2* = *Sigma-iff* [*THEN iffD1*, *THEN conjunct2*, *standard*]

lemma *SigmaE* [*elim!*]:
 $\llbracket c : \text{Sigma}(A, B);$
 $\quad \exists x y. \llbracket x:A; y:B(x); c = \langle x, y \rangle \rrbracket \implies P$
 $\rrbracket \implies P$
 $\langle proof \rangle$

lemma *SigmaE2* [*elim!*]:
 $\llbracket \langle a, b \rangle : \text{Sigma}(A, B);$
 $\quad \llbracket a:A; b:B(a) \rrbracket \implies P$
 $\rrbracket \implies P$
 $\langle proof \rangle$

lemma *Sigma-cong*:
 $\llbracket A=A'; \exists x. x:A' \implies B(x)=B'(x) \rrbracket \implies$
 $\text{Sigma}(A, B) = \text{Sigma}(A', B')$
 $\langle proof \rangle$

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

$\langle proof \rangle$

lemma *Sigma-empty2* [simp]: $A * 0 = 0$
 $\langle proof \rangle$

lemma *Sigma-empty-iff*: $A * B = 0 \iff A = 0 \mid B = 0$
 $\langle proof \rangle$

3.2 Projections *fst* and *snd*

lemma *fst-conv* [simp]: $fst(\langle a, b \rangle) = a$
 $\langle proof \rangle$

lemma *snd-conv* [simp]: $snd(\langle a, b \rangle) = b$
 $\langle proof \rangle$

lemma *fst-type* [TC]: $p : Sigma(A, B) \implies fst(p) : A$
 $\langle proof \rangle$

lemma *snd-type* [TC]: $p : Sigma(A, B) \implies snd(p) : B(fst(p))$
 $\langle proof \rangle$

lemma *Pair-fst-snd-eq*: $a : Sigma(A, B) \implies \langle fst(a), snd(a) \rangle = a$
 $\langle proof \rangle$

3.3 The Eliminator, *split*

lemma *split* [simp]: $split(\%x y. c(x, y), \langle a, b \rangle) == c(a, b)$
 $\langle proof \rangle$

lemma *split-type* [TC]:
[[$p : Sigma(A, B)$;
!! $x y. [x : A; y : B(x)] \implies c(x, y) : C(\langle x, y \rangle)$
]] $\implies split(\%x y. c(x, y), p) : C(p)$
 $\langle proof \rangle$

lemma *expand-split*:
 $u : A * B \implies$
 $R(split(c, u)) \iff (ALL x:A. ALL y:B. u = \langle x, y \rangle \iff R(c(x, y)))$
 $\langle proof \rangle$

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

lemma *splitI*: $R(a, b) \implies split(R, \langle a, b \rangle)$
 $\langle proof \rangle$

lemma *splitE*:
[[$split(R, z)$; $z : Sigma(A, B)$;
!! $x y. [z = \langle x, y \rangle; R(x, y)] \implies P$
]] $\implies P$

<proof>

lemma *splitD*: $\text{split}(R, \langle a, b \rangle) \implies R(a, b)$
<proof>

Complex rules for Sigma.

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

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

end

4 equalities: 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*: $A \subseteq B \implies x \in A \implies x \in B$
<proof>

lemma *the-eq-0* [*simp*]: $(\text{THE } x. \text{False}) = 0$
<proof>

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*: $(\forall x \in A \cup B. P(x)) \iff (\forall x \in A. P(x)) \ \& \ (\forall x \in B. P(x))$
<proof>

lemma *bex-Un*: $(\exists x \in A \cup B. P(x)) \iff (\exists x \in A. P(x)) \ \mid \ (\exists x \in B. P(x))$
<proof>

lemma *ball-UN*: $(\forall z \in (\bigcup x \in A. B(x)). P(z)) \iff (\forall x \in A. \forall z \in B(x). P(z))$
<proof>

lemma *bex-UN*: $(\exists z \in (\bigcup x \in A. B(x)). P(z)) \iff (\exists x \in A. \exists z \in B(x). P(z))$
<proof>

4.2 Converse of a Relation

lemma *converse-iff* [simp]: $\langle a, b \rangle \in \text{converse}(r) \iff \langle b, a \rangle \in r$
 ⟨proof⟩

lemma *converseI* [intro!]: $\langle a, b \rangle \in r \implies \langle b, a \rangle \in \text{converse}(r)$
 ⟨proof⟩

lemma *converseD*: $\langle a, b \rangle \in \text{converse}(r) \implies \langle b, a \rangle \in r$
 ⟨proof⟩

lemma *converseE* [elim!]:

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

 ⟨proof⟩

lemma *converse-converse*: $r \subseteq \text{Sigma}(A, B) \implies \text{converse}(\text{converse}(r)) = r$
 ⟨proof⟩

lemma *converse-type*: $r \subseteq A * B \implies \text{converse}(r) \subseteq B * A$
 ⟨proof⟩

lemma *converse-prod* [simp]: $\text{converse}(A * B) = B * A$
 ⟨proof⟩

lemma *converse-empty* [simp]: $\text{converse}(0) = 0$
 ⟨proof⟩

lemma *converse-subset-iff*:
 $A \subseteq \text{Sigma}(X, Y) \implies \text{converse}(A) \subseteq \text{converse}(B) \iff A \subseteq B$
 ⟨proof⟩

4.3 Finite Set Constructions Using *cons*

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

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

lemma *cons-subset-iff* [iff]: $\text{cons}(a, B) \subseteq C \iff a \in C \ \& \ B \subseteq C$
 ⟨proof⟩

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

lemma *subset-empty-iff*: $A \subseteq 0 \iff A = 0$
 ⟨proof⟩

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

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

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

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

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

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

lemma *equal-singleton* [rule-format]: $[\mid a \in C; \forall y \in C. y = b \mid] \implies C = \{b\}$
(proof)

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

lemma *singleton-subsetI*: $a \in C \implies \{a\} \subseteq C$
(proof)

lemma *singleton-subsetD*: $\{a\} \subseteq C \implies a \in C$
(proof)

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

lemma *succ-subsetI*: $[\mid i \in j; i \subseteq j \mid] \implies \text{succ}(i) \subseteq j$
(proof)

lemma *succ-subsetE*:
 $[\mid \text{succ}(i) \subseteq j; \mid i \in j; i \subseteq j \mid] \implies P \mid] \implies P$
(proof)

lemma *succ-subset-iff*: $\text{succ}(a) \subseteq B \leftrightarrow (a \subseteq B \ \& \ a \in B)$
(proof)

4.4 Binary Intersection

lemma *Int-subset-iff*: $C \subseteq A \text{ Int } B \leftrightarrow C \subseteq A \ \& \ C \subseteq B$
<proof>

lemma *Int-lower1*: $A \text{ Int } B \subseteq A$
<proof>

lemma *Int-lower2*: $A \text{ Int } B \subseteq B$
<proof>

lemma *Int-greatest*: $[| C \subseteq A; C \subseteq B |] \implies C \subseteq A \text{ Int } B$
<proof>

lemma *Int-cons*: $\text{cons}(a, B) \text{ Int } C \subseteq \text{cons}(a, B \text{ Int } C)$
<proof>

lemma *Int-absorb [simp]*: $A \text{ Int } A = A$
<proof>

lemma *Int-left-absorb*: $A \text{ Int } (A \text{ Int } B) = A \text{ Int } B$
<proof>

lemma *Int-commute*: $A \text{ Int } B = B \text{ Int } A$
<proof>

lemma *Int-left-commute*: $A \text{ Int } (B \text{ Int } C) = B \text{ Int } (A \text{ Int } C)$
<proof>

lemma *Int-assoc*: $(A \text{ Int } B) \text{ Int } C = A \text{ Int } (B \text{ Int } C)$
<proof>

lemmas *Int-ac= Int-assoc Int-left-absorb Int-commute Int-left-commute*

lemma *Int-absorb1*: $B \subseteq A \implies A \cap B = B$
<proof>

lemma *Int-absorb2*: $A \subseteq B \implies A \cap B = A$
<proof>

lemma *Int-Un-distrib*: $A \text{ Int } (B \text{ Un } C) = (A \text{ Int } B) \text{ Un } (A \text{ Int } C)$
<proof>

lemma *Int-Un-distrib2*: $(B \text{ Un } C) \text{ Int } A = (B \text{ Int } A) \text{ Un } (C \text{ Int } A)$
<proof>

lemma *subset-Int-iff*: $A \subseteq B \leftrightarrow A \text{ Int } B = A$
<proof>

lemma *subset-Int-iff2*: $A \subseteq B \leftrightarrow B \text{ Int } A = A$
<proof>

lemma *Int-Diff-eq*: $C \subseteq A \implies (A - B) \text{ Int } C = C - B$
<proof>

lemma *Int-cons-left*:
 $\text{cons}(a, A) \text{ Int } B = (\text{if } a \in B \text{ then } \text{cons}(a, A \text{ Int } B) \text{ else } A \text{ Int } B)$
<proof>

lemma *Int-cons-right*:
 $A \text{ Int } \text{cons}(a, B) = (\text{if } a \in A \text{ then } \text{cons}(a, A \text{ Int } B) \text{ else } A \text{ Int } B)$
<proof>

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

4.5 Binary Union

lemma *Un-subset-iff*: $A \cup B \subseteq C \leftrightarrow A \subseteq C \ \& \ B \subseteq C$
<proof>

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

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

lemma *Un-least*: $[\![A \subseteq C; B \subseteq C]\!] \implies A \cup B \subseteq C$
<proof>

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

lemma *Un-absorb [simp]*: $A \cup A = A$
<proof>

lemma *Un-left-absorb*: $A \cup (A \cup B) = A \cup B$
<proof>

lemma *Un-commute*: $A \cup B = B \cup A$
<proof>

lemma *Un-left-commute*: $A \cup (B \cup C) = B \cup (A \cup C)$
<proof>

lemma *Un-assoc*: $(A \cup B) \cup C = A \cup (B \cup C)$
<proof>

lemmas *Un-ac = Un-assoc Un-left-absorb Un-commute Un-left-commute*

lemma *Un-absorb1*: $A \subseteq B \implies A \cup B = B$
 ⟨proof⟩

lemma *Un-absorb2*: $B \subseteq A \implies A \cup B = A$
 ⟨proof⟩

lemma *Un-Int-distrib*: $(A \text{ Int } B) \text{ Un } C = (A \text{ Un } C) \text{ Int } (B \text{ Un } C)$
 ⟨proof⟩

lemma *subset-Un-iff*: $A \subseteq B \iff A \text{ Un } B = B$
 ⟨proof⟩

lemma *subset-Un-iff2*: $A \subseteq B \iff B \text{ Un } A = B$
 ⟨proof⟩

lemma *Un-empty [iff]*: $(A \text{ Un } B = 0) \iff (A = 0 \ \& \ B = 0)$
 ⟨proof⟩

lemma *Un-eq-Union*: $A \text{ Un } B = \text{Union}(\{A, B\})$
 ⟨proof⟩

4.6 Set Difference

lemma *Diff-subset*: $A - B \subseteq A$
 ⟨proof⟩

lemma *Diff-contains*: $[\mid C \subseteq A; C \text{ Int } B = 0 \mid] \implies C \subseteq A - B$
 ⟨proof⟩

lemma *subset-Diff-cons-iff*: $B \subseteq A - \text{cons}(c, C) \iff B \subseteq A - C \ \& \ c \sim: B$
 ⟨proof⟩

lemma *Diff-cancel*: $A - A = 0$
 ⟨proof⟩

lemma *Diff-triv*: $A \text{ Int } B = 0 \implies A - B = A$
 ⟨proof⟩

lemma *empty-Diff [simp]*: $0 - A = 0$
 ⟨proof⟩

lemma *Diff-0 [simp]*: $A - 0 = A$
 ⟨proof⟩

lemma *Diff-eq-0-iff*: $A - B = 0 \iff A \subseteq B$
 ⟨proof⟩

lemma *Diff-cons*: $A - \text{cons}(a,B) = A - B - \{a\}$
<proof>

lemma *Diff-cons2*: $A - \text{cons}(a,B) = A - \{a\} - B$
<proof>

lemma *Diff-disjoint*: $A \text{ Int } (B-A) = 0$
<proof>

lemma *Diff-partition*: $A \subseteq B \implies A \text{ Un } (B-A) = B$
<proof>

lemma *subset-Un-Diff*: $A \subseteq B \text{ Un } (A - B)$
<proof>

lemma *double-complement*: $[| A \subseteq B; B \subseteq C |] \implies B - (C-A) = A$
<proof>

lemma *double-complement-Un*: $(A \text{ Un } B) - (B-A) = A$
<proof>

lemma *Un-Int-crazy*:
 $(A \text{ Int } B) \text{ Un } (B \text{ Int } C) \text{ Un } (C \text{ Int } A) = (A \text{ Un } B) \text{ Int } (B \text{ Un } C) \text{ Int } (C \text{ Un } A)$
<proof>

lemma *Diff-Un*: $A - (B \text{ Un } C) = (A-B) \text{ Int } (A-C)$
<proof>

lemma *Diff-Int*: $A - (B \text{ Int } C) = (A-B) \text{ Un } (A-C)$
<proof>

lemma *Un-Diff*: $(A \text{ Un } B) - C = (A - C) \text{ Un } (B - C)$
<proof>

lemma *Int-Diff*: $(A \text{ Int } B) - C = A \text{ Int } (B - C)$
<proof>

lemma *Diff-Int-distrib*: $C \text{ Int } (A-B) = (C \text{ Int } A) - (C \text{ Int } B)$
<proof>

lemma *Diff-Int-distrib2*: $(A-B) \text{ Int } C = (A \text{ Int } C) - (B \text{ Int } C)$
<proof>

lemma *Un-Int-assoc-iff*: $(A \text{ Int } B) \text{ Un } C = A \text{ Int } (B \text{ Un } C) \iff C \subseteq A$
<proof>

4.7 Big Union and Intersection

lemma *Union-subset-iff*: $Union(A) \subseteq C \leftrightarrow (\forall x \in A. x \subseteq C)$
<proof>

lemma *Union-upper*: $B \in A \implies B \subseteq Union(A)$
<proof>

lemma *Union-least*: $[\![\forall x. x \in A \implies x \subseteq C]\!] \implies Union(A) \subseteq C$
<proof>

lemma *Union-cons* [simp]: $Union(cons(a, B)) = a \ Un \ Union(B)$
<proof>

lemma *Union-Un-distrib*: $Union(A \ Un \ B) = Union(A) \ Un \ Union(B)$
<proof>

lemma *Union-Int-subset*: $Union(A \ Int \ B) \subseteq Union(A) \ Int \ Union(B)$
<proof>

lemma *Union-disjoint*: $Union(C) \ Int \ A = 0 \leftrightarrow (\forall B \in C. B \ Int \ A = 0)$
<proof>

lemma *Union-empty-iff*: $Union(A) = 0 \leftrightarrow (\forall B \in A. B = 0)$
<proof>

lemma *Int-Union2*: $Union(B) \ Int \ A = (\bigcup C \in B. C \ Int \ A)$
<proof>

lemma *Inter-subset-iff*: $A \neq 0 \implies C \subseteq Inter(A) \leftrightarrow (\forall x \in A. C \subseteq x)$
<proof>

lemma *Inter-lower*: $B \in A \implies Inter(A) \subseteq B$
<proof>

lemma *Inter-greatest*: $[\![A \neq 0; \forall x. x \in A \implies C \subseteq x]\!] \implies C \subseteq Inter(A)$
<proof>

lemma *INT-lower*: $x \in A \implies (\bigcap x \in A. B(x)) \subseteq B(x)$
<proof>

lemma *INT-greatest*: $[\![A \neq 0; \forall x. x \in A \implies C \subseteq B(x)]\!] \implies C \subseteq (\bigcap x \in A. B(x))$
<proof>

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

<proof>

lemma *Inter-Un-subset:*

$[[z \in A; z \in B]] \implies \text{Inter}(A) \text{ Un } \text{Inter}(B) \subseteq \text{Inter}(A \text{ Int } B)$
<proof>

lemma *Inter-Un-distrib:*

$[[A \neq 0; B \neq 0]] \implies \text{Inter}(A \text{ Un } B) = \text{Inter}(A) \text{ Int } \text{Inter}(B)$
<proof>

lemma *Union-singleton:* $\text{Union}(\{b\}) = b$

<proof>

lemma *Inter-singleton:* $\text{Inter}(\{b\}) = b$

<proof>

lemma *Inter-cons [simp]:*

$\text{Inter}(\text{cons}(a, B)) = (\text{if } B=0 \text{ then } a \text{ else } a \text{ Int } \text{Inter}(B))$
<proof>

4.8 Unions and Intersections of Families

lemma *subset-UN-iff-eq:* $A \subseteq (\bigcup i \in I. B(i)) \iff A = (\bigcup i \in I. A \text{ Int } B(i))$
<proof>

lemma *UN-subset-iff:* $(\bigcup x \in A. B(x)) \subseteq C \iff (\forall x \in A. B(x) \subseteq C)$
<proof>

lemma *UN-upper:* $x \in A \implies B(x) \subseteq (\bigcup x \in A. B(x))$
<proof>

lemma *UN-least:* $[[\exists x. x \in A \implies B(x) \subseteq C]] \implies (\bigcup x \in A. B(x)) \subseteq C$
<proof>

lemma *Union-eq-UN:* $\text{Union}(A) = (\bigcup x \in A. x)$
<proof>

lemma *Inter-eq-INT:* $\text{Inter}(A) = (\bigcap x \in A. x)$
<proof>

lemma *UN-0 [simp]:* $(\bigcup i \in 0. A(i)) = 0$
<proof>

lemma *UN-singleton:* $(\bigcup x \in A. \{x\}) = A$
<proof>

lemma *UN-Un:* $(\bigcup i \in A \text{ Un } B. C(i)) = (\bigcup i \in A. C(i)) \text{ Un } (\bigcup i \in B. C(i))$
<proof>

lemma *INT-Un*: $(\bigcap i \in I \text{ Un } J. A(i)) =$
 (if $I=0$ *then* $\bigcap j \in J. A(j)$
 else if $J=0$ *then* $\bigcap i \in I. A(i)$
 else $((\bigcap i \in I. A(i)) \text{ Int } (\bigcap j \in J. A(j)))$)

<proof>

lemma *UN-UN-flatten*: $(\bigcup x \in (\bigcup y \in A. B(y)). C(x)) = (\bigcup y \in A. \bigcup x \in B(y). C(x))$

<proof>

lemma *Int-UN-distrib*: $B \text{ Int } (\bigcup i \in I. A(i)) = (\bigcup i \in I. B \text{ Int } A(i))$

<proof>

lemma *Un-INT-distrib*: $I \neq 0 \implies B \text{ Un } (\bigcap i \in I. A(i)) = (\bigcap i \in I. B \text{ Un } A(i))$

<proof>

lemma *Int-UN-distrib2*:

$(\bigcup i \in I. A(i)) \text{ Int } (\bigcup j \in J. B(j)) = (\bigcup i \in I. \bigcup j \in J. A(i) \text{ Int } B(j))$

<proof>

lemma *Un-INT-distrib2*: $[I \neq 0; J \neq 0] \implies$

$(\bigcap i \in I. A(i)) \text{ Un } (\bigcap j \in J. B(j)) = (\bigcap i \in I. \bigcap j \in J. A(i) \text{ Un } B(j))$

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *INT-RepFun [simp]*: $(\bigcap x \in \text{RepFun}(A, f). B(x)) = (\bigcap a \in A. B(f(a)))$

<proof>

lemma *INT-Union-eq*:

$0 \sim: A \implies (\bigcap x \in \text{Union}(A). B(x)) = (\bigcap y \in A. \bigcap x \in y. B(x))$

<proof>

lemma *INT-UN-eq*:

$(\forall x \in A. B(x) \sim 0)$

$\implies (\bigcap z \in (\bigcup x \in A. B(x)). C(z)) = (\bigcap x \in A. \bigcap z \in B(x). C(z))$

<proof>

lemma *UN-Un-distrib*:

$$(\bigcup_{i \in I}. A(i) \text{ Un } B(i)) = (\bigcup_{i \in I}. A(i)) \text{ Un } (\bigcup_{i \in I}. B(i))$$

<proof>

lemma *INT-Int-distrib*:

$$I \neq 0 \implies (\bigcap_{i \in I}. A(i) \text{ Int } B(i)) = (\bigcap_{i \in I}. A(i)) \text{ Int } (\bigcap_{i \in I}. B(i))$$

<proof>

lemma *UN-Int-subset*:

$$(\bigcup_{z \in I} \text{Int } J. A(z)) \subseteq (\bigcup_{z \in I}. A(z)) \text{ Int } (\bigcup_{z \in J}. A(z))$$

<proof>

lemma *Diff-UN*: $I \neq 0 \implies B - (\bigcup_{i \in I}. A(i)) = (\bigcap_{i \in I}. B - A(i))$

<proof>

lemma *Diff-INT*: $I \neq 0 \implies B - (\bigcap_{i \in I}. A(i)) = (\bigcup_{i \in I}. B - A(i))$

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *SUM-UN-distrib1*:

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

<proof>

lemma *SUM-UN-distrib2*:

$$(\sum i \in I. \bigcup_{j \in J}. C(i, j)) = (\bigcup_{j \in J}. \sum i \in I. C(i, j))$$

<proof>

lemma *SUM-Un-distrib1*:

$$(\sum i \in I \text{ Un } J. C(i)) = (\sum i \in I. C(i)) \text{ Un } (\sum j \in J. C(j))$$

<proof>

lemma *SUM-Un-distrib2*:

$$(\sum_{i \in I}. A(i) \text{ Un } B(i)) = (\sum_{i \in I}. A(i)) \text{ Un } (\sum_{i \in I}. B(i))$$

<proof>

lemma *prod-Un-distrib2*: $I * (A \text{ Un } B) = I * A \text{ Un } I * B$

<proof>

lemma *SUM-Int-distrib1*:

$$(\sum_{i \in I} \text{Int } J. C(i)) = (\sum_{i \in I}. C(i)) \text{ Int } (\sum_{j \in J}. C(j))$$

<proof>

lemma *SUM-Int-distrib2*:

$$(\sum_{i \in I}. A(i) \text{ Int } B(i)) = (\sum_{i \in I}. A(i)) \text{ Int } (\sum_{i \in I}. B(i))$$

<proof>

lemma *prod-Int-distrib2*: $I * (A \text{ Int } B) = I * A \text{ Int } I * B$

<proof>

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

<proof>

lemma *times-subset-iff*:

$$(A' * B' \subseteq A * B) \leftrightarrow (A' = 0 \mid B' = 0 \mid (A' \subseteq A) \ \& \ (B' \subseteq B))$$

<proof>

lemma *Int-Sigma-eq*:

$$(\sum_{x \in A'}. B'(x)) \text{ Int } (\sum_{x \in A}. B(x)) = (\sum_{x \in A'} \text{Int } A. B'(x)) \text{ Int } B(x)$$

<proof>

lemma *domain-iff*: $a: \text{domain}(r) \leftrightarrow (\exists x y. \langle a, y \rangle \in r)$

<proof>

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

<proof>

lemma *domainE* [*elim!*]:

$$[\mid a \in \text{domain}(r); \ !y. \langle a, y \rangle \in r \implies P \mid] \implies P$$

<proof>

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

<proof>

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

<proof>

lemma *domain-0* [*simp*]: $\text{domain}(0) = 0$
<proof>

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

lemma *domain-Un-eq* [*simp*]: $\text{domain}(A \text{ Un } B) = \text{domain}(A) \text{ Un } \text{domain}(B)$
<proof>

lemma *domain-Int-subset*: $\text{domain}(A \text{ Int } B) \subseteq \text{domain}(A) \text{ Int } \text{domain}(B)$
<proof>

lemma *domain-Diff-subset*: $\text{domain}(A) - \text{domain}(B) \subseteq \text{domain}(A - B)$
<proof>

lemma *domain-UN*: $\text{domain}(\bigcup x \in A. B(x)) = (\bigcup x \in A. \text{domain}(B(x)))$
<proof>

lemma *domain-Union*: $\text{domain}(\text{Union}(A)) = (\bigcup x \in A. \text{domain}(x))$
<proof>

lemma *rangeI* [*intro*]: $\langle a, b \rangle \in r \implies b \in \text{range}(r)$
<proof>

lemma *rangeE* [*elim!*]: $[\![b \in \text{range}(r); \exists x. \langle x, b \rangle \in r \implies P \!]\!] \implies P$
<proof>

lemma *range-subset*: $\text{range}(A * B) \subseteq B$
<proof>

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

lemma *range-0* [*simp*]: $\text{range}(0) = 0$
<proof>

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

lemma *range-Un-eq* [*simp*]: $\text{range}(A \text{ Un } B) = \text{range}(A) \text{ Un } \text{range}(B)$
<proof>

lemma *range-Int-subset*: $\text{range}(A \text{ Int } B) \subseteq \text{range}(A) \text{ Int } \text{range}(B)$
<proof>

lemma *range-Diff-subset*: $\text{range}(A) - \text{range}(B) \subseteq \text{range}(A - B)$
(proof)

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

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

lemma *fieldI1*: $\langle a, b \rangle \in r \implies a \in \text{field}(r)$
(proof)

lemma *fieldI2*: $\langle a, b \rangle \in r \implies b \in \text{field}(r)$
(proof)

lemma *fieldCI* [intro]:
($\sim \langle c, a \rangle \in r \implies \langle a, b \rangle \in r \implies a \in \text{field}(r)$)
(proof)

lemma *fieldE* [elim!]:
[| $a \in \text{field}(r)$;
 $\forall x. \langle a, x \rangle \in r \implies P$;
 $\forall x. \langle x, a \rangle \in r \implies P$ |] $\implies P$
(proof)

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

lemma *domain-subset-field*: $\text{domain}(r) \subseteq \text{field}(r)$
(proof)

lemma *range-subset-field*: $\text{range}(r) \subseteq \text{field}(r)$
(proof)

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

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

lemma *relation-field-times-field*: $\text{relation}(r) \implies r \subseteq \text{field}(r) * \text{field}(r)$
(proof)

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

lemma *field-0* [*simp*]: $field(0) = 0$
<proof>

lemma *field-cons* [*simp*]: $field(cons(<a,b>,r)) = cons(a, cons(b, field(r)))$
<proof>

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

lemma *field-Int-subset*: $field(A \text{ Int } B) \subseteq field(A) \text{ Int } field(B)$
<proof>

lemma *field-Diff-subset*: $field(A) - field(B) \subseteq field(A - B)$
<proof>

lemma *field-converse* [*simp*]: $field(converse(r)) = field(r)$
<proof>

lemma *rel-Union*: $(\forall x \in S. \exists X A B. x \subseteq A * B) ==>$
 $Union(S) \subseteq domain(Union(S)) * range(Union(S))$
<proof>

lemma *rel-Un*: $[| r \subseteq A * B; s \subseteq C * D |] ==> (r \text{ Un } s) \subseteq (A \text{ Un } C) * (B \text{ Un } D)$
<proof>

lemma *domain-Diff-eq*: $[| <a,c> \in r; c \sim b |] ==> domain(r - \{<a,b>\}) = domain(r)$
<proof>

lemma *range-Diff-eq*: $[| <c,b> \in r; c \sim a |] ==> range(r - \{<a,b>\}) = range(r)$
<proof>

4.9 Image of a Set under a Function or Relation

lemma *image-iff*: $b \in r `` A <-> (\exists x \in A. <x,b> \in r)$
<proof>

lemma *image-singleton-iff*: $b \in r `` \{a\} <-> <a,b> \in r$
<proof>

lemma *imageI* [*intro*]: $[| <a,b> \in r; a \in A |] ==> b \in r `` A$
<proof>

lemma *imageE* [*elim!*]:
 $[| b \in r `` A; !!x. [| <x,b> \in r; x \in A |] ==> P |] ==> P$
<proof>

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

<proof>

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

<proof>

lemma *image-Un [simp]*: $r''(A \text{ Un } B) = (r''A) \text{ Un } (r''B)$

<proof>

lemma *image-UN*: $r''(\bigcup_{x \in A}. B(x)) = (\bigcup_{x \in A}. r''B(x))$

<proof>

lemma *Collect-image-eq*:

$\{z \in \text{Sigma}(A,B). P(z)\}''C = (\bigcup_{x \in A}. \{y \in B(x). x \in C \ \& \ P(\langle x,y \rangle)\})$

<proof>

lemma *image-Int-subset*: $r''(A \text{ Int } B) \subseteq (r''A) \text{ Int } (r''B)$

<proof>

lemma *image-Int-square-subset*: $(r \text{ Int } A*A)''B \subseteq (r''B) \text{ Int } A$

<proof>

lemma *image-Int-square*: $B \subseteq A \implies (r \text{ Int } A*A)''B = (r''B) \text{ Int } A$

<proof>

lemma *image-0-left [simp]*: $0''A = 0$

<proof>

lemma *image-Un-left*: $(r \text{ Un } s)''A = (r''A) \text{ Un } (s''A)$

<proof>

lemma *image-Int-subset-left*: $(r \text{ Int } s)''A \subseteq (r''A) \text{ Int } (s''A)$

<proof>

4.10 Inverse Image of a Set under a Function or Relation

lemma *vimage-iff*:

$a \in r^{-''B} \iff (\exists y \in B. \langle a,y \rangle \in r)$

<proof>

lemma *vimage-singleton-iff*: $a \in r^{-''\{b\}} \iff \langle a,b \rangle \in r$

<proof>

lemma *vimageI [intro]*: $[\langle a,b \rangle \in r; b \in B] \implies a \in r^{-''B}$

<proof>

lemma *vimageE* [*elim!*]:

$\llbracket a: r-{}^{\prime\prime}B; !!x. \llbracket \langle a, x \rangle \in r; x \in B \rrbracket \implies P \rrbracket \implies P$
<proof>

lemma *vimage-subset*: $r \subseteq A * B \implies r-{}^{\prime\prime}C \subseteq A$

<proof>

lemma *vimage-0* [*simp*]: $r-{}^{\prime\prime}0 = 0$

<proof>

lemma *vimage-Un* [*simp*]: $r-{}^{\prime\prime}(A \text{ Un } B) = (r-{}^{\prime\prime}A) \text{ Un } (r-{}^{\prime\prime}B)$

<proof>

lemma *vimage-Int-subset*: $r-{}^{\prime\prime}(A \text{ Int } B) \subseteq (r-{}^{\prime\prime}A) \text{ Int } (r-{}^{\prime\prime}B)$

<proof>

lemma *vimage-eq-UN*: $f-{}^{\prime\prime}B = (\bigcup_{y \in B}. f-{}^{\prime\prime}\{y\})$

<proof>

lemma *function-vimage-Int*:

$\text{function}(f) \implies f-{}^{\prime\prime}(A \text{ Int } B) = (f-{}^{\prime\prime}A) \text{ Int } (f-{}^{\prime\prime}B)$
<proof>

lemma *function-vimage-Diff*: $\text{function}(f) \implies f-{}^{\prime\prime}(A - B) = (f-{}^{\prime\prime}A) - (f-{}^{\prime\prime}B)$

<proof>

lemma *function-image-vimage*: $\text{function}(f) \implies f-{}^{\prime\prime}(f-{}^{\prime\prime}A) \subseteq A$

<proof>

lemma *vimage-Int-square-subset*: $(r \text{ Int } A * A)-{}^{\prime\prime}B \subseteq (r-{}^{\prime\prime}B) \text{ Int } A$

<proof>

lemma *vimage-Int-square*: $B \subseteq A \implies (r \text{ Int } A * A)-{}^{\prime\prime}B = (r-{}^{\prime\prime}B) \text{ Int } A$

<proof>

lemma *vimage-0-left* [*simp*]: $0-{}^{\prime\prime}A = 0$

<proof>

lemma *vimage-Un-left*: $(r \text{ Un } s)-{}^{\prime\prime}A = (r-{}^{\prime\prime}A) \text{ Un } (s-{}^{\prime\prime}A)$

<proof>

lemma *vimage-Int-subset-left*: $(r \text{ Int } s)-{}^{\prime\prime}A \subseteq (r-{}^{\prime\prime}A) \text{ Int } (s-{}^{\prime\prime}A)$

<proof>

lemma *converse-Un* [*simp*]: $\text{converse}(A \text{ Un } B) = \text{converse}(A) \text{ Un } \text{converse}(B)$
 ⟨*proof*⟩

lemma *converse-Int* [*simp*]: $\text{converse}(A \text{ Int } B) = \text{converse}(A) \text{ Int } \text{converse}(B)$
 ⟨*proof*⟩

lemma *converse-Diff* [*simp*]: $\text{converse}(A - B) = \text{converse}(A) - \text{converse}(B)$
 ⟨*proof*⟩

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

lemma *converse-INT* [*simp*]:
 $\text{converse}(\bigcap x \in A. B(x)) = (\bigcap x \in A. \text{converse}(B(x)))$
 ⟨*proof*⟩

4.11 Powerset Operator

lemma *Pow-0* [*simp*]: $\text{Pow}(0) = \{0\}$
 ⟨*proof*⟩

lemma *Pow-insert*: $\text{Pow}(\text{cons}(a, A)) = \text{Pow}(A) \text{ Un } \{\text{cons}(a, X) \mid X: \text{Pow}(A)\}$
 ⟨*proof*⟩

lemma *Un-Pow-subset*: $\text{Pow}(A) \text{ Un } \text{Pow}(B) \subseteq \text{Pow}(A \text{ Un } B)$
 ⟨*proof*⟩

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

lemma *subset-Pow-Union*: $A \subseteq \text{Pow}(\text{Union}(A))$
 ⟨*proof*⟩

lemma *Union-Pow-eq* [*simp*]: $\text{Union}(\text{Pow}(A)) = A$
 ⟨*proof*⟩

lemma *Union-Pow-iff*: $\text{Union}(A) \in \text{Pow}(B) \iff A \in \text{Pow}(\text{Pow}(B))$
 ⟨*proof*⟩

lemma *Pow-Int-eq* [*simp*]: $\text{Pow}(A \text{ Int } B) = \text{Pow}(A) \text{ Int } \text{Pow}(B)$
 ⟨*proof*⟩

lemma *Pow-INT-eq*: $A \neq 0 \implies \text{Pow}(\bigcap x \in A. B(x)) = (\bigcap x \in A. \text{Pow}(B(x)))$
 ⟨*proof*⟩

4.12 RepFun

lemma *RepFun-subset*: $[\![\!|x. x \in A \implies f(x) \in B \!|\!] \implies \{f(x). x \in A\} \subseteq B$
<proof>

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

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

4.13 Collect

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

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

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

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

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

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

lemma *Collect-Collect-eq* [*simp*]:
 $\text{Collect}(\text{Collect}(A, P), Q) = \text{Collect}(A, \%x. P(x) \ \& \ Q(x))$
<proof>

lemma *Collect-Int-Collect-eq*:
 $\text{Collect}(A, P) \text{ Int } \text{Collect}(A, Q) = \text{Collect}(A, \%x. P(x) \ \& \ Q(x))$
<proof>

lemma *Collect-Union-eq* [*simp*]:
 $\text{Collect}(\bigcup x \in A. B(x), P) = (\bigcup x \in A. \text{Collect}(B(x), P))$
<proof>

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

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

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

lemma *Collect-conj-eq*: $\{x \in A. P(x) \ \& \ Q(x)\} = \text{Collect}(A, P) \text{ Int } \text{Collect}(A, Q)$
 <proof>

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>

end

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

theory *Fixedpt imports equalities begin*

definition

bnd-mono :: $[i, i \Rightarrow i] \Rightarrow o$ **where**
 $\text{bnd-mono}(D, h) == h(D) \leq D \ \& \ (\text{ALL } W \ X. W \leq X \ \longrightarrow \ X \leq D \ \longrightarrow \ h(W) \leq h(X))$

definition

lfp :: $[i, i \Rightarrow i] \Rightarrow i$ **where**
 $\text{lfp}(D, h) == \text{Inter}(\{X: \text{Pow}(D). h(X) \leq X\})$

definition

gfp :: $[i, i \Rightarrow i] \Rightarrow i$ **where**
 $\text{gfp}(D, h) == \text{Union}(\{X: \text{Pow}(D). X \leq h(X)\})$

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

5.1 Monotone Operators

lemma *bnd-monoI*:

$[\![\ h(D) \leq D; \ \& \ W \ X. [\![\ W \leq D; \ X \leq D; \ W \leq X \ \]\!] \implies h(W) \leq h(X) \ \]\!] \implies \text{bnd-mono}(D, h)$

<proof>

lemma *bnd-monoD1*: $\text{bnd-mono}(D, h) \implies h(D) \leq D$

<proof>

lemma *bnd-monoD2*: $[[\text{bnd-mono}(D,h); W \leq X; X \leq D]] \implies h(W) \leq h(X)$
<proof>

lemma *bnd-mono-subset*:
 $[[\text{bnd-mono}(D,h); X \leq D]] \implies h(X) \leq D$
<proof>

lemma *bnd-mono-Un*:
 $[[\text{bnd-mono}(D,h); A \leq D; B \leq D]] \implies h(A) \text{ Un } h(B) \leq h(A \text{ Un } B)$
<proof>

lemma *bnd-mono-UN*:
 $[[\text{bnd-mono}(D,h); \forall i \in I. A(i) \leq D]] \implies h(\bigcup_{i \in I} A(i)) \leq h(\bigcup_{i \in I} A(i))$
<proof>

lemma *bnd-mono-Int*:
 $[[\text{bnd-mono}(D,h); A \leq D; B \leq D]] \implies h(A \text{ Int } B) \leq h(A) \text{ Int } h(B)$
<proof>

5.2 Proof of Knaster-Tarski Theorem using *lfp*

lemma *lfp-lowerbound*:
 $[[h(A) \leq A; A \leq D]] \implies \text{lfp}(D,h) \leq A$
<proof>

lemma *lfp-subset*: $\text{lfp}(D,h) \leq D$
<proof>

lemma *def-lfp-subset*: $A == \text{lfp}(D,h) \implies A \leq D$
<proof>

lemma *lfp-greatest*:
 $[[h(D) \leq D; \forall X. [[h(X) \leq X; X \leq D]] \implies A \leq X]] \implies A \leq \text{lfp}(D,h)$
<proof>

lemma *lfp-lemma1*:
 $[[\text{bnd-mono}(D,h); h(A) \leq A; A \leq D]] \implies h(\text{lfp}(D,h)) \leq A$
<proof>

lemma *lfp-lemma2*: $\text{bnd-mono}(D,h) \implies h(\text{lfp}(D,h)) \leq \text{lfp}(D,h)$

$\langle proof \rangle$

lemma *lfp-lemma3*:

$bnd\text{-}mono(D,h) \implies lfp(D,h) \leq h(lfp(D,h))$

$\langle proof \rangle$

lemma *lfp-unfold*: $bnd\text{-}mono(D,h) \implies lfp(D,h) = h(lfp(D,h))$

$\langle proof \rangle$

lemma *def-lfp-unfold*:

$[[A == lfp(D,h); bnd\text{-}mono(D,h)]] \implies A = h(A)$

$\langle proof \rangle$

5.3 General Induction Rule for Least Fixedpoints

lemma *Collect-is-pre-fixedpt*:

$[[bnd\text{-}mono(D,h); \forall x. x : h(Collect(lfp(D,h),P)) \implies P(x)]] \implies h(Collect(lfp(D,h),P)) \leq Collect(lfp(D,h),P)$

$\langle proof \rangle$

lemma *induct*:

$[[bnd\text{-}mono(D,h); a : lfp(D,h); \forall x. x : h(Collect(lfp(D,h),P)) \implies P(x)]] \implies P(a)$

$\langle proof \rangle$

lemma *def-induct*:

$[[A == lfp(D,h); bnd\text{-}mono(D,h); a:A; \forall x. x : h(Collect(A,P)) \implies P(x)]] \implies P(a)$

$\langle proof \rangle$

lemma *lfp-Int-lowerbound*:

$[[h(D \text{ Int } A) \leq A; bnd\text{-}mono(D,h)]] \implies lfp(D,h) \leq A$

$\langle proof \rangle$

lemma *lfp-mono*:

assumes *hmono*: $bnd\text{-}mono(D,h)$

and *imono*: $bnd\text{-}mono(E,i)$

and *subhi*: $\forall X. X \leq D \implies h(X) \leq i(X)$

shows $lfp(D,h) \leq lfp(E,i)$

$\langle proof \rangle$

lemma *lfp-mono2*:

$\llbracket i(D) \leq D; \forall X. X \leq D \implies h(X) \leq i(X) \rrbracket \implies \text{lfp}(D, h) \leq \text{lfp}(D, i)$
<proof>

lemma *lfp-cong*:

$\llbracket D = D'; \forall X. X \leq D' \implies h(X) = h'(X) \rrbracket \implies \text{lfp}(D, h) = \text{lfp}(D', h')$
<proof>

5.4 Proof of Knaster-Tarski Theorem using *gfp*

lemma *gfp-upperbound*: $\llbracket A \leq h(A); A \leq D \rrbracket \implies A \leq \text{gfp}(D, h)$
<proof>

lemma *gfp-subset*: $\text{gfp}(D, h) \leq D$
<proof>

lemma *def-gfp-subset*: $A = \text{gfp}(D, h) \implies A \leq D$
<proof>

lemma *gfp-least*:

$\llbracket \text{bnd-mono}(D, h); \forall X. \llbracket X \leq h(X); X \leq D \rrbracket \implies X \leq A \rrbracket \implies$
 $\text{gfp}(D, h) \leq A$
<proof>

lemma *gfp-lemma1*:

$\llbracket \text{bnd-mono}(D, h); A \leq h(A); A \leq D \rrbracket \implies A \leq h(\text{gfp}(D, h))$
<proof>

lemma *gfp-lemma2*: $\text{bnd-mono}(D, h) \implies \text{gfp}(D, h) \leq h(\text{gfp}(D, h))$
<proof>

lemma *gfp-lemma3*:

$\text{bnd-mono}(D, h) \implies h(\text{gfp}(D, h)) \leq \text{gfp}(D, h)$
<proof>

lemma *gfp-unfold*: $\text{bnd-mono}(D, h) \implies \text{gfp}(D, h) = h(\text{gfp}(D, h))$
<proof>

lemma *def-gfp-unfold*:

$\llbracket A = \text{gfp}(D, h); \text{bnd-mono}(D, h) \rrbracket \implies A = h(A)$
<proof>

5.5 Coinduction Rules for Greatest Fixed Points

lemma *weak-coinduct*: $\llbracket a : X; X \leq h(X); X \leq D \rrbracket \implies a : \text{gfp}(D, h)$
<proof>

lemma *coinduct-lemma*:

$$\llbracket X \leq h(X \text{ Un } \text{gfp}(D,h)); X \leq D; \text{bnd-mono}(D,h) \rrbracket \implies$$

$$X \text{ Un } \text{gfp}(D,h) \leq h(X \text{ Un } \text{gfp}(D,h))$$
 <proof>

lemma *coinduct*:

$$\llbracket \text{bnd-mono}(D,h); a : X; X \leq h(X \text{ Un } \text{gfp}(D,h)); X \leq D \rrbracket$$

$$\implies a : \text{gfp}(D,h)$$
 <proof>

lemma *def-coinduct*:

$$\llbracket A == \text{gfp}(D,h); \text{bnd-mono}(D,h); a : X; X \leq h(X \text{ Un } A); X \leq D \rrbracket$$

$$\implies$$

$$a : A$$
 <proof>

lemma *def-Collect-coinduct*:

$$\llbracket A == \text{gfp}(D, \%w. \text{Collect}(D,P(w))); \text{bnd-mono}(D, \%w. \text{Collect}(D,P(w)));$$

$$a : X; X \leq D; !!z. z : X \implies P(X \text{ Un } A, z) \rrbracket \implies$$

$$a : A$$
 <proof>

lemma *gfp-mono*:

$$\llbracket \text{bnd-mono}(D,h); D \leq E;$$

$$!!X. X \leq D \implies h(X) \leq i(X) \rrbracket \implies \text{gfp}(D,h) \leq \text{gfp}(E,i)$$
 <proof>

end

6 Bool: Booleans in Zermelo-Fraenkel Set Theory

theory *Bool* imports *pair* begin

abbreviation

one (1) **where**
 $1 == \text{succ}(0)$

abbreviation

two (2) **where**
 $2 == \text{succ}(1)$

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

definition *bool* == {0,1}

definition $cond(b,c,d) == if(b=1,c,d)$

definition $not(b) == cond(b,0,1)$

definition

and $:: [i,i]==>i$ (**infixl and 70**) **where**
 a and $b == cond(a,b,0)$

definition

or $:: [i,i]==>i$ (**infixl or 65**) **where**
 a or $b == cond(a,1,b)$

definition

xor $:: [i,i]==>i$ (**infixl xor 65**) **where**
 a xor $b == cond(a,not(b),b)$

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

lemma $singleton-0: \{0\} = 1$
 $\langle proof \rangle$

lemma $bool-1I [simp,TC]: 1 : bool$
 $\langle proof \rangle$

lemma $bool-0I [simp,TC]: 0 : bool$
 $\langle proof \rangle$

lemma $one-not-0: 1 \sim 0$
 $\langle proof \rangle$

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

lemma $boolE:$

$[[c: bool; c=1 ==> P; c=0 ==> P]] ==> P$
 $\langle proof \rangle$

lemma $cond-1 [simp]: cond(1,c,d) = c$
 $\langle proof \rangle$

lemma $cond-0 [simp]: cond(0,c,d) = d$
 $\langle proof \rangle$

lemma *cond-type* [TC]: [| b: bool; c: A(1); d: A(0) |] ==> cond(b,c,d): A(b)
<proof>

lemma *cond-simple-type*: [| b: bool; c: A; d: A |] ==> cond(b,c,d): A
<proof>

lemma *def-cond-1*: [| !!b. j(b)==cond(b,c,d) |] ==> j(1) = c
<proof>

lemma *def-cond-0*: [| !!b. j(b)==cond(b,c,d) |] ==> j(0) = d
<proof>

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

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

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

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

lemma *not-type* [TC]: a:bool ==> not(a) : bool
<proof>

lemma *and-type* [TC]: [| a:bool; b:bool |] ==> a and b : bool
<proof>

lemma *or-type* [TC]: [| a:bool; b:bool |] ==> a or b : bool
<proof>

lemma *xor-type* [TC]: [| a:bool; b:bool |] ==> a xor b : bool
<proof>

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:bool ==> not(not(a)) = a
<proof>

lemma *not-and* [*simp*]: a:bool ==> not(a and b) = not(a) or not(b)
<proof>

lemma *not-or* [*simp*]: $a:\text{bool} \implies \text{not}(a \text{ or } b) = \text{not}(a) \text{ and } \text{not}(b)$
<proof>

6.2 Laws About 'and'

lemma *and-absorb* [*simp*]: $a:\text{bool} \implies a \text{ and } a = a$
<proof>

lemma *and-commute*: $[[a:\text{bool}; b:\text{bool}]] \implies a \text{ and } b = b \text{ and } a$
<proof>

lemma *and-assoc*: $a:\text{bool} \implies (a \text{ and } b) \text{ and } c = a \text{ and } (b \text{ and } c)$
<proof>

lemma *and-or-distrib*: $[[a:\text{bool}; b:\text{bool}; c:\text{bool}]] \implies$
 $(a \text{ or } b) \text{ and } c = (a \text{ and } c) \text{ or } (b \text{ and } c)$
<proof>

6.3 Laws About 'or'

lemma *or-absorb* [*simp*]: $a:\text{bool} \implies a \text{ or } a = a$
<proof>

lemma *or-commute*: $[[a:\text{bool}; b:\text{bool}]] \implies a \text{ or } b = b \text{ or } a$
<proof>

lemma *or-assoc*: $a:\text{bool} \implies (a \text{ or } b) \text{ or } c = a \text{ or } (b \text{ or } c)$
<proof>

lemma *or-and-distrib*: $[[a:\text{bool}; b:\text{bool}; c:\text{bool}]] \implies$
 $(a \text{ and } b) \text{ or } c = (a \text{ or } c) \text{ and } (b \text{ or } c)$
<proof>

definition

bool-of-o :: $o \implies i$ **where**
 $\text{bool-of-o}(P) == (\text{if } P \text{ then } 1 \text{ else } 0)$

lemma [*simp*]: $\text{bool-of-o}(\text{True}) = 1$
<proof>

lemma [*simp*]: $\text{bool-of-o}(\text{False}) = 0$
<proof>

lemma [*simp,TC*]: $\text{bool-of-o}(P) \in \text{bool}$
<proof>

lemma [*simp*]: $(\text{bool-of-o}(P) = 1) \iff P$
<proof>

lemma [simp]: (bool-of-o(P) = 0) <-> ~P
 <proof>

<ML>

end

7 Sum: Disjoint Sums

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

And the "Part" primitive for simultaneous recursive type definitions

definition sum :: [i,i]=>i (**infixr** + 65) **where**
 A+B == {0}*A Un {1}*B

definition Inl :: i=>i **where**
 Inl(a) == <0,a>

definition Inr :: i=>i **where**
 Inr(b) == <1,b>

definition case :: [i=>i, i=>i, i]=>i **where**
 case(c,d) == (%<y,z>. cond(y, d(z), c(z)))

definition Part :: [i,i=>i] => i **where**
 Part(A,h) == {x: A. EX z. x = h(z)}

7.1 Rules for the Part Primitive

lemma Part-iff:
 a : Part(A,h) <-> a:A & (EX y. a=h(y))
 <proof>

lemma Part-eqI [intro]:
 [| a : A; a=h(b) |] ==> a : Part(A,h)
 <proof>

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

lemma PartE [elim!]:
 [| a : Part(A,h); !!z. [| a : A; a=h(z) |] ==> P
 |] ==> P
 <proof>

lemma Part-subset: Part(A,h) <= A
 <proof>

7.2 Rules for Disjoint Sums

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

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

lemma *InlI* [*intro!, simp, TC*]: $a : A ==> Inl(a) : A+B$
<proof>

lemma *InrI* [*intro!, simp, TC*]: $b : B ==> Inr(b) : A+B$
<proof>

lemma *sumE* [*elim!*]:
 [[$u : A+B$;
 !! $x. [x:A; u=Inl(x)] ==> P$;
 !! $y. [y:B; u=Inr(y)] ==> P$
]] ==> P
<proof>

lemma *Inl-iff* [*iff*]: $Inl(a)=Inl(b) <-> a=b$
<proof>

lemma *Inr-iff* [*iff*]: $Inr(a)=Inr(b) <-> a=b$
<proof>

lemma *Inl-Inr-iff* [*simp*]: $Inl(a)=Inr(b) <-> False$
<proof>

lemma *Inr-Inl-iff* [*simp*]: $Inr(b)=Inl(a) <-> False$
<proof>

lemma *sum-empty* [*simp*]: $0+0 = 0$
<proof>

lemmas *Inl-inject* = *Inl-iff [THEN iffD1, standard]*

lemmas *Inr-inject* = *Inr-iff [THEN iffD1, standard]*

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*: $Inl(a) : A+B ==> a : A$

<proof>

lemma *InrD*: $Inr(b): A+B \implies b: B$
<proof>

lemma *sum-iff*: $u: A+B \iff (EX\ x. x:A \ \&\ u=Inl(x)) \mid (EX\ y. y:B \ \&\ u=Inr(y))$
<proof>

lemma *Inl-in-sum-iff* [*simp*]: $(Inl(x) \in A+B) \iff (x \in A)$
<proof>

lemma *Inr-in-sum-iff* [*simp*]: $(Inr(y) \in A+B) \iff (y \in B)$
<proof>

lemma *sum-subset-iff*: $A+B \leq C+D \iff A \leq C \ \&\ B \leq D$
<proof>

lemma *sum-equal-iff*: $A+B = C+D \iff A=C \ \&\ B=D$
<proof>

lemma *sum-eq-2-times*: $A+A = 2*A$
<proof>

7.3 The Eliminator: *case*

lemma *case-Inl* [*simp*]: $case(c, d, Inl(a)) = c(a)$
<proof>

lemma *case-Inr* [*simp*]: $case(c, d, Inr(b)) = d(b)$
<proof>

lemma *case-type* [*TC*]:
[[$u: A+B$;
 $!!x. x: A \implies c(x): C(Inl(x))$;
 $!!y. y: B \implies d(y): C(Inr(y))$
]] $\implies case(c,d,u) : C(u)$
<proof>

lemma *expand-case*: $u: A+B \implies$
 $R(case(c,d,u)) \iff$
 $((ALL\ x:A. u = Inl(x) \implies R(c(x))) \ \&$
 $(ALL\ y:B. u = Inr(y) \implies R(d(y))))$
<proof>

lemma *case-cong*:
[[$z: A+B$;
 $!!x. x:A \implies c(x)=c'(x)$;
 $!!y. y:B \implies d(y)=d'(y)$
]] $\implies case(c,d,z) = case(c',d',z)$

<proof>

lemma *case-case*: $z : A+B ==>$
 $case(c, d, case(\%x. Inl(c'(x)), \%y. Inr(d'(y)), z)) =$
 $case(\%x. c(c'(x)), \%y. d(d'(y)), z)$
<proof>

7.4 More Rules for $Part(A, h)$

lemma *Part-mono*: $A <= B ==> Part(A, h) <= Part(B, h)$
<proof>

lemma *Part-Collect*: $Part(Collect(A, P), h) = Collect(Part(A, h), P)$
<proof>

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

lemma *Part-Inl*: $Part(A+B, Inl) = \{Inl(x). x : A\}$
<proof>

lemma *Part-Inr*: $Part(A+B, Inr) = \{Inr(y). y : B\}$
<proof>

lemma *PartD1*: $a : Part(A, h) ==> a : A$
<proof>

lemma *Part-id*: $Part(A, \%x. x) = A$
<proof>

lemma *Part-Inr2*: $Part(A+B, \%x. Inr(h(x))) = \{Inr(y). y : Part(B, h)\}$
<proof>

lemma *Part-sum-equality*: $C <= A+B ==> Part(C, Inl) \text{ Un } Part(C, Inr) = C$
<proof>

end

8 func: Functions, Function Spaces, Lambda-Abstraction

theory *func* imports *equalities Sum* begin

8.1 The Pi Operator: Dependent Function Space

lemma *subset-Sigma-imp-relation*: $r <= Sigma(A, B) ==> relation(r)$
<proof>

lemma *relation-converse-converse* [*simp*]:

$relation(r) ==> converse(converse(r)) = r$
<proof>

lemma *relation-restrict [simp]*: $relation(restrict(r,A))$
<proof>

lemma *Pi-iff*:
 $f: Pi(A,B) <-> function(f) \& f \leq Sigma(A,B) \& A \leq domain(f)$
<proof>

lemma *Pi-iff-old*:
 $f: Pi(A,B) <-> f \leq Sigma(A,B) \& (ALL x:A. EX! y. <x,y>: f)$
<proof>

lemma *fun-is-function*: $f: Pi(A,B) ==> function(f)$
<proof>

lemma *function-imp-Pi*:
 $[| function(f); relation(f) |] ==> f \in domain(f) -> range(f)$
<proof>

lemma *functionI*:
 $[| !!x y y'. [| <x,y>:r; <x,y'>:r |] ==> y=y' |] ==> function(r)$
<proof>

lemma *fun-is-rel*: $f: Pi(A,B) ==> f \leq Sigma(A,B)$
<proof>

lemma *Pi-cong*:
 $[| A=A'; !!x. x:A' ==> B(x)=B'(x) |] ==> Pi(A,B) = Pi(A',B')$
<proof>

lemma *fun-weaken-type*: $[| f: A->B; B \leq D |] ==> f: A->D$
<proof>

8.2 Function Application

lemma *apply-equality2*: $[| <a,b>: f; <a,c>: f; f: Pi(A,B) |] ==> b=c$
<proof>

lemma *function-apply-equality*: $[| <a,b>: f; function(f) |] ==> f'a = b$
<proof>

lemma *apply-equality*: $[| <a,b>: f; f: Pi(A,B) |] ==> f'a = b$

$\langle proof \rangle$

lemma *apply-0*: $a \sim : domain(f) \implies f'a = 0$

$\langle proof \rangle$

lemma *Pi-memberD*: $[| f : Pi(A,B); c : f |] \implies EX x:A. c = \langle x, f'x \rangle$

$\langle proof \rangle$

lemma *function-apply-Pair*: $[| function(f); a : domain(f) |] \implies \langle a, f'a \rangle : f$

$\langle proof \rangle$

lemma *apply-Pair*: $[| f : Pi(A,B); a : A |] \implies \langle a, f'a \rangle : f$

$\langle proof \rangle$

lemma *apply-type [TC]*: $[| f : Pi(A,B); a : A |] \implies f'a : B(a)$

$\langle proof \rangle$

lemma *apply-funtype*: $[| f : A \rightarrow B; a : A |] \implies f'a : B$

$\langle proof \rangle$

lemma *apply-iff*: $f : Pi(A,B) \implies \langle a, b \rangle : f \leftrightarrow a : A \ \& \ f'a = b$

$\langle proof \rangle$

lemma *Pi-type*: $[| f : Pi(A,C); !!x. x:A \implies f'x : B(x) |] \implies f : Pi(A,B)$

$\langle proof \rangle$

lemma *Pi-Collect-iff*:

$(f : Pi(A, \%x. \{y:B(x). P(x,y)\}))$

$\leftrightarrow f : Pi(A,B) \ \& \ (ALL x: A. P(x, f'x))$

$\langle proof \rangle$

lemma *Pi-weaken-type*:

$[| f : Pi(A,B); !!x. x:A \implies B(x) \leq C(x) |] \implies f : Pi(A,C)$

$\langle proof \rangle$

lemma *domain-type*: $[| \langle a, b \rangle : f; f : Pi(A,B) |] \implies a : A$

$\langle proof \rangle$

lemma *range-type*: $[| \langle a, b \rangle : f; f : Pi(A,B) |] \implies b : B(a)$

$\langle proof \rangle$

lemma *Pair-mem-PiD*: $[[\langle a, b \rangle : f; f : \text{Pi}(A, B)]] \implies a : A \ \& \ b : B(a) \ \& \ f \ a = b$
 $\langle \text{proof} \rangle$

8.3 Lambda Abstraction

lemma *lamI*: $a : A \implies \langle a, b(a) \rangle : (\text{lam } x : A. b(x))$
 $\langle \text{proof} \rangle$

lemma *lamE*:
 $[[p : (\text{lam } x : A. b(x)); !!x. [x : A; p = \langle x, b(x) \rangle]]] \implies P$
 $[[]] \implies P$
 $\langle \text{proof} \rangle$

lemma *lamD*: $[[\langle a, c \rangle : (\text{lam } x : A. b(x))]] \implies c = b(a)$
 $\langle \text{proof} \rangle$

lemma *lam-type [TC]*:
 $[[!!x. x : A \implies b(x) : B(x)]] \implies (\text{lam } x : A. b(x)) : \text{Pi}(A, B)$
 $\langle \text{proof} \rangle$

lemma *lam-funtype*: $(\text{lam } x : A. b(x)) : A \rightarrow \{b(x). x : A\}$
 $\langle \text{proof} \rangle$

lemma *function-lam*: *function* $(\text{lam } x : A. b(x))$
 $\langle \text{proof} \rangle$

lemma *relation-lam*: *relation* $(\text{lam } x : A. b(x))$
 $\langle \text{proof} \rangle$

lemma *beta-if [simp]*: $(\text{lam } x : A. b(x)) \ 'a = (\text{if } a : A \text{ then } b(a) \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *beta*: $a : A \implies (\text{lam } x : A. b(x)) \ 'a = b(a)$
 $\langle \text{proof} \rangle$

lemma *lam-empty [simp]*: $(\text{lam } x : 0. b(x)) = 0$
 $\langle \text{proof} \rangle$

lemma *domain-lam [simp]*: $\text{domain}(\text{Lambda}(A, b)) = A$
 $\langle \text{proof} \rangle$

lemma *lam-cong [cong]*:
 $[[A = A'; !!x. x : A' \implies b(x) = b'(x)]] \implies \text{Lambda}(A, b) = \text{Lambda}(A', b')$
 $\langle \text{proof} \rangle$

lemma *lam-theI*:
 $(!!x. x : A \implies \text{EX! } y. Q(x, y)) \implies \text{EX } f. \text{ALL } x : A. Q(x, f \ x)$
 $\langle \text{proof} \rangle$

lemma *lam-eqE*: $[(\text{lam } x:A. f(x)) = (\text{lam } x:A. g(x)); a:A] \implies f(a)=g(a)$
 <proof>

lemma *Pi-empty1* [*simp*]: $Pi(0,A) = \{0\}$
 <proof>

lemma *singleton-fun* [*simp*]: $\{<a,b>\} : \{a\} \rightarrow \{b\}$
 <proof>

lemma *Pi-empty2* [*simp*]: $(A \rightarrow 0) = (\text{if } A=0 \text{ then } \{0\} \text{ else } 0)$
 <proof>

lemma *fun-space-empty-iff* [*iff*]: $(A \rightarrow X)=0 \iff X=0 \ \& \ (A \neq 0)$
 <proof>

8.4 Extensionality

lemma *fun-subset*:
 $[(f : Pi(A,B); g : Pi(C,D); A \leq C; \ \forall x. x:A \implies f'x = g'x)] \implies f \leq g$
 <proof>

lemma *fun-extension*:
 $[(f : Pi(A,B); g : Pi(A,D); \ \forall x. x:A \implies f'x = g'x)] \implies f=g$
 <proof>

lemma *eta* [*simp*]: $f : Pi(A,B) \implies (\text{lam } x:A. f'x) = f$
 <proof>

lemma *fun-extension-iff*:
 $[(f : Pi(A,B); g : Pi(A,C))] \implies (\forall a:A. f'a = g'a) \iff f=g$
 <proof>

lemma *fun-subset-eq*: $[(f : Pi(A,B); g : Pi(A,C))] \implies f \leq g \iff (f = g)$
 <proof>

lemma *Pi-lamE*:
assumes *major*: $f : Pi(A,B)$
and *minor*: $\forall b. [\forall x:A. b(x):B(x); f = (\text{lam } x:A. b(x))] \implies P$
shows P
 <proof>

8.5 Images of Functions

lemma *image-lam*: $C \leq A \implies (\text{lam } x:A. b(x)) \text{ `` } C = \{b(x). x:C\}$
 <proof>

lemma *Repfun-function-if*:
 $\text{function}(f) \implies \{f'x. x:C\} = (\text{if } C \leq \text{domain}(f) \text{ then } f''C \text{ else } \text{cons}(0, f''C))$
 <proof>

lemma *image-function*:
 $[\text{function}(f); C \leq \text{domain}(f)] \implies f''C = \{f'x. x:C\}$
 <proof>

lemma *image-fun*: $[\text{f} : \text{Pi}(A,B); C \leq A] \implies f''C = \{f'x. x:C\}$
 <proof>

lemma *image-eq-UN*:
assumes $f: f \in \text{Pi}(A,B)$ $C \subseteq A$ **shows** $f''C = (\bigcup x \in C. \{f'x\})$
 <proof>

lemma *Pi-image-cons*:
 $[\text{f} : \text{Pi}(A,B); x: A] \implies f'' \text{cons}(x,y) = \text{cons}(f'x, f''y)$
 <proof>

8.6 Properties of *restrict(f, A)*

lemma *restrict-subset*: $\text{restrict}(f,A) \leq f$
 <proof>

lemma *function-restrictI*:
 $\text{function}(f) \implies \text{function}(\text{restrict}(f,A))$
 <proof>

lemma *restrict-type2*: $[\text{f} : \text{Pi}(C,B); A \leq C] \implies \text{restrict}(f,A) : \text{Pi}(A,B)$
 <proof>

lemma *restrict*: $\text{restrict}(f,A) \text{ ' } a = (\text{if } a : A \text{ then } f'a \text{ else } 0)$
 <proof>

lemma *restrict-empty* [*simp*]: $\text{restrict}(f,0) = 0$
 <proof>

lemma *restrict-iff*: $z \in \text{restrict}(r,A) \iff z \in r \ \& \ (\exists x \in A. \exists y. z = \langle x, y \rangle)$
 <proof>

lemma *restrict-restrict* [*simp*]:
 $\text{restrict}(\text{restrict}(r,A),B) = \text{restrict}(r, A \text{ Int } B)$
 <proof>

lemma *domain-restrict* [simp]: $\text{domain}(\text{restrict}(f, C)) = \text{domain}(f) \text{ Int } C$
 ⟨proof⟩

lemma *restrict-idem*: $f \leq \text{Sigma}(A, B) \implies \text{restrict}(f, A) = f$
 ⟨proof⟩

lemma *domain-restrict-idem*:
 $\llbracket \text{domain}(r) \leq A; \text{relation}(r) \rrbracket \implies \text{restrict}(r, A) = r$
 ⟨proof⟩

lemma *domain-restrict-lam* [simp]: $\text{domain}(\text{restrict}(\text{Lambda}(A, f), C)) = A \text{ Int } C$
 ⟨proof⟩

lemma *restrict-if* [simp]: $\text{restrict}(f, A) \text{ ' } a = (\text{if } a : A \text{ then } f \text{ ' } a \text{ else } 0)$
 ⟨proof⟩

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

lemma *fun-cons-restrict-eq*:
 $f : \text{cons}(a, b) \rightarrow B \implies f = \text{cons}(\langle a, f \text{ ' } a \rangle, \text{restrict}(f, b))$
 ⟨proof⟩

8.7 Unions of Functions

lemma *function-Union*:
 $\llbracket \text{ALL } x:S. \text{function}(x); \text{ALL } x:S. \text{ALL } y:S. x \leq y \mid y \leq x \rrbracket \implies \text{function}(\text{Union}(S))$
 ⟨proof⟩

lemma *fun-Union*:
 $\llbracket \text{ALL } f:S. \text{EX } C D. f:C \rightarrow D; \text{ALL } f:S. \text{ALL } y:S. f \leq y \mid y \leq f \rrbracket \implies \text{Union}(S) : \text{domain}(\text{Union}(S)) \rightarrow \text{range}(\text{Union}(S))$
 ⟨proof⟩

lemma *gen-relation-Union* [rule-format]:
 $\forall f \in F. \text{relation}(f) \implies \text{relation}(\text{Union}(F))$
 ⟨proof⟩

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*:

$[[f: A \rightarrow B; g: C \rightarrow D; A \text{ Int } C = 0 \]]$
 $\implies (f \text{ Un } g) : (A \text{ Un } C) \rightarrow (B \text{ Un } D)$

<proof>

lemma *fun-disjoint-apply1*: $a \notin \text{domain}(g) \implies (f \text{ Un } g)'a = f'a$

<proof>

lemma *fun-disjoint-apply2*: $c \notin \text{domain}(f) \implies (f \text{ Un } g)'c = g'c$

<proof>

8.8 Domain and Range of a Function or Relation

lemma *domain-of-fun*: $f : \text{Pi}(A,B) \implies \text{domain}(f) = A$

<proof>

lemma *apply-rangeI*: $[[f : \text{Pi}(A,B); a : A \]]$ $\implies f'a : \text{range}(f)$

<proof>

lemma *range-of-fun*: $f : \text{Pi}(A,B) \implies f : A \rightarrow \text{range}(f)$

<proof>

8.9 Extensions of Functions

lemma *fun-extend*:

$[[f: A \rightarrow B; c \sim : A \]]$ $\implies \text{cons}(\langle c, b \rangle, f) : \text{cons}(c, A) \rightarrow \text{cons}(b, B)$

<proof>

lemma *fun-extend3*:

$[[f: A \rightarrow B; c \sim : A; b : B \]]$ $\implies \text{cons}(\langle c, b \rangle, f) : \text{cons}(c, A) \rightarrow B$

<proof>

lemma *extend-apply*:

$c \sim : \text{domain}(f) \implies \text{cons}(\langle c, b \rangle, f)'a = (\text{if } a=c \text{ then } b \text{ else } f'a)$

<proof>

lemma *fun-extend-apply* [*simp*]:

$[[f: A \rightarrow B; c \sim : A \]]$ $\implies \text{cons}(\langle c, b \rangle, f)'a = (\text{if } a=c \text{ then } b \text{ else } f'a)$

<proof>

lemmas *singleton-apply = apply-equality* [OF *singletonI singleton-fun, simp*]

lemma *cons-fun-eq*:

$c \sim : A \implies \text{cons}(c, A) \rightarrow B = (\bigcup f \in A \rightarrow B. \bigcup b \in B. \{\text{cons}(\langle c, b \rangle, f)\})$

<proof>

<proof>

lemma *Pow-mono*: $A \leq B \implies \text{Pow}(A) \leq \text{Pow}(B)$

<proof>

lemma *Union-mono*: $A \leq B \implies \text{Union}(A) \leq \text{Union}(B)$

<proof>

lemma *UN-mono*:

$[\![A \leq C; \forall x. x:A \implies B(x) \leq D(x)]\!] \implies (\bigcup_{x \in A} B(x)) \leq (\bigcup_{x \in C} D(x))$

<proof>

lemma *Inter-anti-mono*: $[\![A \leq B; A \neq 0]\!] \implies \text{Inter}(B) \leq \text{Inter}(A)$

<proof>

lemma *cons-mono*: $C \leq D \implies \text{cons}(a, C) \leq \text{cons}(a, D)$

<proof>

lemma *Un-mono*: $[\![A \leq C; B \leq D]\!] \implies A \text{ Un } B \leq C \text{ Un } D$

<proof>

lemma *Int-mono*: $[\![A \leq C; B \leq D]\!] \implies A \text{ Int } B \leq C \text{ Int } D$

<proof>

lemma *Diff-mono*: $[\![A \leq C; D \leq B]\!] \implies A - B \leq C - D$

<proof>

8.11.2 Standard Products, Sums and Function Spaces

lemma *Sigma-mono* [*rule-format*]:

$[\![A \leq C; \forall x. x:A \implies B(x) \leq D(x)]\!] \implies \text{Sigma}(A, B) \leq \text{Sigma}(C, D)$

<proof>

lemma *sum-mono*: $[\![A \leq C; B \leq D]\!] \implies A + B \leq C + D$

<proof>

lemma *Pi-mono*: $B \leq C \implies A \rightarrow B \leq A \rightarrow C$

<proof>

lemma *lam-mono*: $A \leq B \implies \text{Lambda}(A, c) \leq \text{Lambda}(B, c)$

<proof>

8.11.3 Converse, Domain, Range, Field

lemma *converse-mono*: $r \leq s \implies \text{converse}(r) \leq \text{converse}(s)$

<proof>

lemma *domain-mono*: $r \leq s \implies \text{domain}(r) \leq \text{domain}(s)$
 ⟨proof⟩

lemmas *domain-rel-subset = subset-trans [OF domain-mono domain-subset]*

lemma *range-mono*: $r \leq s \implies \text{range}(r) \leq \text{range}(s)$
 ⟨proof⟩

lemmas *range-rel-subset = subset-trans [OF range-mono range-subset]*

lemma *field-mono*: $r \leq s \implies \text{field}(r) \leq \text{field}(s)$
 ⟨proof⟩

lemma *field-rel-subset*: $r \leq A * A \implies \text{field}(r) \leq A$
 ⟨proof⟩

8.11.4 Images

lemma *image-pair-mono*:

$\llbracket \! \! \! \forall x y. \langle x, y \rangle : r \implies \langle x, y \rangle : s; A \leq B \rrbracket \implies r \text{``} A \leq s \text{``} B$
 ⟨proof⟩

lemma *vimage-pair-mono*:

$\llbracket \! \! \! \forall x y. \langle x, y \rangle : r \implies \langle x, y \rangle : s; A \leq B \rrbracket \implies r \text{``} A \leq s \text{``} B$
 ⟨proof⟩

lemma *image-mono*: $\llbracket r \leq s; A \leq B \rrbracket \implies r \text{``} A \leq s \text{``} B$
 ⟨proof⟩

lemma *vimage-mono*: $\llbracket r \leq s; A \leq B \rrbracket \implies r \text{``} A \leq s \text{``} B$
 ⟨proof⟩

lemma *Collect-mono*:

$\llbracket A \leq B; \! \! \! \forall x. x : A \implies P(x) \dashrightarrow Q(x) \rrbracket \implies \text{Collect}(A, P) \leq \text{Collect}(B, Q)$
 ⟨proof⟩

lemmas *basic-monos = subset-refl imp-refl disj-mono conj-mono ex-mono
 Collect-mono Part-mono in-mono*

lemma *box-image-simp*:

$\llbracket f : \text{Pi}(X, Y); A \subseteq X \rrbracket \implies (\text{EX } x : f \text{``} A. P(x)) \leftrightarrow (\text{EX } x : A. P(f \text{``} x))$
 ⟨proof⟩

lemma *ball-image-simp*:

$\llbracket f : \text{Pi}(X, Y); A \subseteq X \rrbracket \implies (\text{ALL } x : f \text{``} A. P(x)) \leftrightarrow (\text{ALL } x : A. P(f \text{``} x))$

<proof>

end

9 QPair: 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

$QPair \quad :: [i, i] \Rightarrow i \quad (\langle -; - \rangle) \text{ where}$
 $\langle a; b \rangle == a + b$

definition

$qfst \quad :: i \Rightarrow i \text{ where}$
 $qfst(p) == THE a. EX b. p = \langle a; b \rangle$

definition

$qsnd \quad :: i \Rightarrow i \text{ where}$
 $qsnd(p) == THE b. EX a. p = \langle a; b \rangle$

definition

$qsplit \quad :: [[i, i] \Rightarrow 'a, i] \Rightarrow 'a::\{\} \text{ where}$
 $qsplit(c, p) == c(qfst(p), qsnd(p))$

definition

$qconverse \quad :: i \Rightarrow i \text{ where}$
 $qconverse(r) == \{z. w:r, EX x y. w = \langle x; y \rangle \ \& \ z = \langle y; x \rangle\}$

definition

$QSigma \quad :: [i, i \Rightarrow i] \Rightarrow i \text{ where}$
 $QSigma(A, B) == \bigcup x \in A. \bigcup y \in B(x). \{\langle x; y \rangle\}$

syntax

$-QSUM \quad :: [idt, i, i] \Rightarrow i \quad ((\exists QSUM \text{ :-./ -}) 10)$

translations

$QSUM \ x:A. B \Rightarrow CONST \ QSigma(A, \%x. B)$

abbreviation

$qprod \text{ (infixr } \langle * \rangle 80) \text{ where}$
 $A \langle * \rangle B == QSigma(A, \%-. B)$

definition

qsum :: $[i,i]=>i$ (infixr <+> 65) where
 $A <+> B == (\{0\} <*> A) \text{ Un } (\{1\} <*> B)$

definition

QInl :: $i=>i$ where
 $QInl(a) == <0;a>$

definition

QInr :: $i=>i$ where
 $QInr(b) == <1;b>$

definition

qcase :: $[i=>i, i=>i, i]=>i$ where
 $qcase(c,d) == qsplit(\%y z. cond(y, d(z), c(z)))$

9.1 Quine ordered pairing

lemma *QPair-empty* [simp]: $<0;0> = 0$
<proof>

lemma *QPair-iff* [simp]: $<a;b> = <c;d> <-> a=c \ \& \ b=d$
<proof>

lemmas *QPair-inject* = *QPair-iff* [THEN iffD1, THEN conjE, standard, elim!]

lemma *QPair-inject1*: $<a;b> = <c;d> ==> a=c$
<proof>

lemma *QPair-inject2*: $<a;b> = <c;d> ==> b=d$
<proof>

9.1.1 QSigma: Disjoint union of a family of sets Generalizes Cartesian product

lemma *QSigmaI* [intro!]: $[[a:A; b:B(a)]] ==> <a;b> : QSigma(A,B)$
<proof>

lemma *QSigmaE* [elim!]:
 $[[c: QSigma(A,B);$
 $!!x y. [[x:A; y:B(x); c=<x;y>]] ==> P$
 $]] ==> P$
<proof>

lemma *QSigmaE2* [elim!]:
 $[[<a;b>: QSigma(A,B); [[a:A; b:B(a)]] ==> P]] ==> P$
<proof>

lemma *QSigmaD1*: $\langle a;b \rangle : QSigma(A,B) \implies a : A$
 $\langle proof \rangle$

lemma *QSigmaD2*: $\langle a;b \rangle : QSigma(A,B) \implies b : B(a)$
 $\langle proof \rangle$

lemma *QSigma-cong*:
 $\llbracket A=A'; \ !x. x:A' \implies B(x)=B'(x) \rrbracket \implies$
 $QSigma(A,B) = QSigma(A',B')$
 $\langle proof \rangle$

lemma *QSigma-empty1* [*simp*]: $QSigma(0,B) = 0$
 $\langle proof \rangle$

lemma *QSigma-empty2* [*simp*]: $A \langle * \rangle 0 = 0$
 $\langle proof \rangle$

9.1.2 Projections: qfst, qsnd

lemma *qfst-conv* [*simp*]: $qfst(\langle a;b \rangle) = a$
 $\langle proof \rangle$

lemma *qsnd-conv* [*simp*]: $qsnd(\langle a;b \rangle) = b$
 $\langle proof \rangle$

lemma *qfst-type* [*TC*]: $p:QSigma(A,B) \implies qfst(p) : A$
 $\langle proof \rangle$

lemma *qsnd-type* [*TC*]: $p:QSigma(A,B) \implies qsnd(p) : B(qfst(p))$
 $\langle proof \rangle$

lemma *QPair-qfst-qsnd-eq*: $a:QSigma(A,B) \implies \langle qfst(a); qsnd(a) \rangle = a$
 $\langle proof \rangle$

9.1.3 Eliminator: qspllit

lemma *qspllit* [*simp*]: $qspllit(\%x y. c(x,y), \langle a;b \rangle) == c(a,b)$
 $\langle proof \rangle$

lemma *qspllit-type* [*elim!*]:
 $\llbracket p:QSigma(A,B);$
 $\ !x y. \llbracket x:A; y:B(x) \rrbracket \implies c(x,y):C(\langle x;y \rangle)$
 $\rrbracket \implies qspllit(\%x y. c(x,y), p) : C(p)$
 $\langle proof \rangle$

lemma *expand-qspllit*:
 $u: A \langle * \rangle B \implies R(qspllit(c,u)) \langle - \rangle (ALL x:A. ALL y:B. u = \langle x;y \rangle \dashrightarrow$
 $R(c(x,y)))$
 $\langle proof \rangle$

9.1.4 qsplit for predicates: result type o

lemma *qsplitI*: $R(a,b) \implies \text{qsplit}(R, \langle a;b \rangle)$
<proof>

lemma *qsplitE*:
[[*qsplit*(*R*,*z*); *z*:*QSigma*(*A*,*B*);
!!*x y*. [[*z* = $\langle x;y \rangle$; *R*(*x*,*y*)]] \implies *P*
]] \implies *P*
<proof>

lemma *qsplitD*: $\text{qsplit}(R, \langle a;b \rangle) \implies R(a,b)$
<proof>

9.1.5 qconverse

lemma *qconverseI* [*intro!*]: $\langle a;b \rangle : r \implies \langle b;a \rangle : \text{qconverse}(r)$
<proof>

lemma *qconverseD* [*elim!*]: $\langle a;b \rangle : \text{qconverse}(r) \implies \langle b;a \rangle : r$
<proof>

lemma *qconverseE* [*elim!*]:
[[*yx* : *qconverse*(*r*);
!!*x y*. [[*yx* = $\langle y;x \rangle$; $\langle x;y \rangle : r$]] \implies *P*
]] \implies *P*
<proof>

lemma *qconverse-qconverse*: $r \leq \text{QSigma}(A,B) \implies \text{qconverse}(\text{qconverse}(r)) = r$
<proof>

lemma *qconverse-type*: $r \leq A \langle * \rangle B \implies \text{qconverse}(r) \leq B \langle * \rangle A$
<proof>

lemma *qconverse-prod*: $\text{qconverse}(A \langle * \rangle B) = B \langle * \rangle A$
<proof>

lemma *qconverse-empty*: $\text{qconverse}(0) = 0$
<proof>

9.2 The Quine-inspired notion of disjoint sum

lemmas *qsum-defs* = *qsum-def* *QInl-def* *QInr-def* *qcase-def*

lemma *QInlI* [*intro!*]: $a : A \implies \text{QInl}(a) : A \langle + \rangle B$
<proof>

lemma *QInrI* [*intro!*]: $b : B \implies QInr(b) : A <+> B$
 ⟨*proof*⟩

lemma *qsumE* [*elim!*]:
 [! $u : A <+> B$;
 !! $x. [! x:A; u=QInl(x)] \implies P$;
 !! $y. [! y:B; u=QInr(y)] \implies P$
 !] $\implies P$
 ⟨*proof*⟩

lemma *QInl-iff* [*iff*]: $QInl(a)=QInl(b) <-> a=b$
 ⟨*proof*⟩

lemma *QInr-iff* [*iff*]: $QInr(a)=QInr(b) <-> a=b$
 ⟨*proof*⟩

lemma *QInl-QInr-iff* [*simp*]: $QInl(a)=QInr(b) <-> False$
 ⟨*proof*⟩

lemma *QInr-QInl-iff* [*simp*]: $QInr(b)=QInl(a) <-> False$
 ⟨*proof*⟩

lemma *qsum-empty* [*simp*]: $0 <+> 0 = 0$
 ⟨*proof*⟩

lemmas *QInl-inject* = *QInl-iff* [*THEN iffD1, standard*]
lemmas *QInr-inject* = *QInr-iff* [*THEN iffD1, standard*]
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*: $QInl(a): A <+> B \implies a : A$
 ⟨*proof*⟩

lemma *QInrD*: $QInr(b): A <+> B \implies b : B$
 ⟨*proof*⟩

lemma *qsum-iff*:
 $u : A <+> B <-> (EX x. x:A \ \& \ u=QInl(x)) \mid (EX y. y:B \ \& \ u=QInr(y))$
 ⟨*proof*⟩

lemma *qsum-subset-iff*: $A <+> B \leq C <+> D \leftrightarrow A \leq C \ \& \ B \leq D$
 ⟨proof⟩

lemma *qsum-equal-iff*: $A <+> B = C <+> D \leftrightarrow A = C \ \& \ B = D$
 ⟨proof⟩

9.2.1 Eliminator – qcase

lemma *qcase-QInl* [*simp*]: $qcase(c, d, QInl(a)) = c(a)$
 ⟨proof⟩

lemma *qcase-QInr* [*simp*]: $qcase(c, d, QInr(b)) = d(b)$
 ⟨proof⟩

lemma *qcase-type*:

$$\begin{aligned} & \llbracket u: A <+> B; \\ & \quad !!x. x: A \implies c(x): C(QInl(x)); \\ & \quad !!y. y: B \implies d(y): C(QInr(y)) \\ & \rrbracket \implies qcase(c,d,u) : C(u) \end{aligned}$$

 ⟨proof⟩

lemma *Part-QInl*: $Part(A <+> B, QInl) = \{QInl(x). x: A\}$
 ⟨proof⟩

lemma *Part-QInr*: $Part(A <+> B, QInr) = \{QInr(y). y: B\}$
 ⟨proof⟩

lemma *Part-QInr2*: $Part(A <+> B, \%x. QInr(h(x))) = \{QInr(y). y: Part(B,h)\}$
 ⟨proof⟩

lemma *Part-qsum-equality*: $C \leq A <+> B \implies Part(C, QInl) \cup Part(C, QInr) = C$
 ⟨proof⟩

9.2.2 Monotonicity

lemma *QPair-mono*: $\llbracket a \leq c; b \leq d \rrbracket \implies \langle a; b \rangle \leq \langle c; d \rangle$
 ⟨proof⟩

lemma *QSigma-mono* [*rule-format*]:

$$\llbracket A \leq C; \ ALL \ x:A. B(x) \leq D(x) \rrbracket \implies QSigma(A,B) \leq QSigma(C,D)$$

 ⟨proof⟩

lemma *QInl-mono*: $a \leq b \implies QInl(a) \leq QInl(b)$
 ⟨proof⟩

lemma *QInr-mono*: $a \leq b \implies \text{QInr}(a) \leq \text{QInr}(b)$
 ⟨proof⟩

lemma *qsum-mono*: $[[A \leq C; B \leq D]] \implies A <+> B \leq C <+> D$
 ⟨proof⟩

end

10 Perm: Injections, Surjections, Bijections, Composition

theory *Perm* imports *func* begin

definition

comp :: $[i, i] \Rightarrow i$ (**infixr** *O 60*) **where**
 $r \text{ O } s == \{xz : \text{domain}(s) * \text{range}(r) .$
 $\text{EX } x \ y \ z. xz = \langle x, z \rangle \ \& \ \langle x, y \rangle : s \ \& \ \langle y, z \rangle : r\}$

definition

id :: $i \Rightarrow i$ **where**
 $\text{id}(A) == (\text{lam } x:A. x)$

definition

inj :: $[i, i] \Rightarrow i$ **where**
 $\text{inj}(A, B) == \{ f : A \rightarrow B. \text{ALL } w:A. \text{ALL } x:A. f'w = f'x \implies w = x \}$

definition

surj :: $[i, i] \Rightarrow i$ **where**
 $\text{surj}(A, B) == \{ f : A \rightarrow B . \text{ALL } y:B. \text{EX } x:A. f'x = y \}$

definition

bij :: $[i, i] \Rightarrow i$ **where**
 $\text{bij}(A, B) == \text{inj}(A, B) \text{ Int } \text{surj}(A, B)$

10.1 Surjective Function Space

lemma *surj-is-fun*: $f : \text{surj}(A, B) \implies f : A \rightarrow B$
 ⟨proof⟩

lemma *fun-is-surj*: $f : \text{Pi}(A, B) \implies f : \text{surj}(A, \text{range}(f))$
 ⟨proof⟩

lemma *surj-range*: $f: \text{surj}(A,B) \implies \text{range}(f)=B$
 ⟨proof⟩

A function with a right inverse is a surjection

lemma *f-imp-surjective*:
 $[[f: A \multimap B; !!y. y:B \implies d(y): A; !!y. y:B \implies f \cdot d(y) = y]]$
 $\implies f: \text{surj}(A,B)$
 ⟨proof⟩

lemma *lam-surjective*:
 $[[!!x. x:A \implies c(x): B;$
 $!!y. y:B \implies d(y): A;$
 $!!y. y:B \implies c(d(y)) = y$
 $]] \implies (\text{lam } x:A. c(x)) : \text{surj}(A,B)$
 ⟨proof⟩

Cantor's theorem revisited

lemma *cantor-surj*: $f \sim: \text{surj}(A, \text{Pow}(A))$
 ⟨proof⟩

10.2 Injective Function Space

lemma *inj-is-fun*: $f: \text{inj}(A,B) \implies f: A \multimap B$
 ⟨proof⟩

Good for dealing with sets of pairs, but a bit ugly in use [used in AC]

lemma *inj-equality*:
 $[[\langle a, b \rangle : f; \langle c, b \rangle : f; f: \text{inj}(A,B)]]$ $\implies a=c$
 ⟨proof⟩

lemma *inj-apply-equality*: $[[f: \text{inj}(A,B); f \cdot a = f \cdot b; a:A; b:A]]$ $\implies a=b$
 ⟨proof⟩

A function with a left inverse is an injection

lemma *f-imp-injective*: $[[f: A \multimap B; \text{ALL } x:A. d(f \cdot x)=x]]$ $\implies f: \text{inj}(A,B)$
 ⟨proof⟩

lemma *lam-injective*:
 $[[!!x. x:A \implies c(x): B;$
 $!!x. x:A \implies d(c(x)) = x]]$
 $\implies (\text{lam } x:A. c(x)) : \text{inj}(A,B)$
 ⟨proof⟩

10.3 Bijections

lemma *bij-is-inj*: $f: \text{bij}(A,B) \implies f: \text{inj}(A,B)$
 ⟨proof⟩

lemma *bij-is-surj*: $f: \text{bij}(A,B) \implies f: \text{surj}(A,B)$

$\langle proof \rangle$

$f: \text{bij}(A,B) \iff f: A \rightarrow B$

lemmas *bij-is-fun = bij-is-inj* [THEN inj-is-fun, standard]

lemma *lam-bijective*:

$$\begin{aligned} & [\text{!!}x. x:A \implies c(x): B; \\ & \quad \text{!!}y. y:B \implies d(y): A; \\ & \quad \text{!!}x. x:A \implies d(c(x)) = x; \\ & \quad \text{!!}y. y:B \implies c(d(y)) = y \\ &] \implies (\text{lam } x:A. c(x)) : \text{bij}(A,B) \end{aligned}$$

$\langle proof \rangle$

lemma *RepFun-bijective*: (ALL $y : x. \text{EX! } y'. f(y') = f(y)$)

$\implies (\text{lam } z:\{f(y). y:x\}. \text{THE } y. f(y) = z) : \text{bij}(\{f(y). y:x\}, x)$

$\langle proof \rangle$

10.4 Identity Function

lemma *idI* [intro!]: $a:A \implies \langle a,a \rangle : \text{id}(A)$

$\langle proof \rangle$

lemma *idE* [elim!]: $[p: \text{id}(A); \text{!!}x. [x:A; p=\langle x,x \rangle] \implies P] \implies P$

$\langle proof \rangle$

lemma *id-type*: $\text{id}(A) : A \rightarrow A$

$\langle proof \rangle$

lemma *id-conv* [simp]: $x:A \implies \text{id}(A) 'x = x$

$\langle proof \rangle$

lemma *id-mono*: $A \leq B \implies \text{id}(A) \leq \text{id}(B)$

$\langle proof \rangle$

lemma *id-subset-inj*: $A \leq B \implies \text{id}(A) : \text{inj}(A,B)$

$\langle proof \rangle$

lemmas *id-inj = subset-refl* [THEN id-subset-inj, standard]

lemma *id-surj*: $\text{id}(A) : \text{surj}(A,A)$

$\langle proof \rangle$

lemma *id-bij*: $\text{id}(A) : \text{bij}(A,A)$

$\langle proof \rangle$

lemma *subset-iff-id*: $A \leq B \iff \text{id}(A) : A \rightarrow B$

$\langle proof \rangle$

id as the identity relation

lemma *id-iff* [simp]: $\langle x, y \rangle \in \text{id}(A) \iff x = y \ \& \ y \in A$
(proof)

10.5 Converse of a Function

lemma *inj-converse-fun*: $f: \text{inj}(A, B) \implies \text{converse}(f): \text{range}(f) \rightarrow A$
(proof)

Equations for converse(f)

The premises are equivalent to saying that f is injective...

lemma *left-inverse-lemma*:
[[$f: A \rightarrow B$; $\text{converse}(f): C \rightarrow A$; $a: A$]] $\implies \text{converse}(f) \text{ ` } (f \text{ ` } a) = a$
(proof)

lemma *left-inverse* [simp]: [[$f: \text{inj}(A, B)$; $a: A$]] $\implies \text{converse}(f) \text{ ` } (f \text{ ` } a) = a$
(proof)

lemma *left-inverse-eq*:
[[$f \in \text{inj}(A, B)$; $f \text{ ` } x = y$; $x \in A$]] $\implies \text{converse}(f) \text{ ` } y = x$
(proof)

lemmas *left-inverse-bij = bij-is-inj* [THEN left-inverse, standard]

lemma *right-inverse-lemma*:
[[$f: A \rightarrow B$; $\text{converse}(f): C \rightarrow A$; $b: C$]] $\implies f \text{ ` } (\text{converse}(f) \text{ ` } b) = b$
(proof)

lemma *right-inverse* [simp]:
[[$f: \text{inj}(A, B)$; $b: \text{range}(f)$]] $\implies f \text{ ` } (\text{converse}(f) \text{ ` } b) = b$
(proof)

lemma *right-inverse-bij*: [[$f: \text{bij}(A, B)$; $b: B$]] $\implies f \text{ ` } (\text{converse}(f) \text{ ` } b) = b$
(proof)

10.6 Converses of Injections, Surjections, Bijections

lemma *inj-converse-inj*: $f: \text{inj}(A, B) \implies \text{converse}(f): \text{inj}(\text{range}(f), A)$
(proof)

lemma *inj-converse-surj*: $f: \text{inj}(A, B) \implies \text{converse}(f): \text{surj}(\text{range}(f), A)$
(proof)

Adding this as an intro! rule seems to cause looping

lemma *bij-converse-bij* [TC]: $f: \text{bij}(A, B) \implies \text{converse}(f): \text{bij}(B, A)$
(proof)

10.7 Composition of Two Relations

The inductive definition package could derive these theorems for $\text{@termr } O$
 S

lemma *compI* [*intro*]: $[[\langle a,b \rangle : s; \langle b,c \rangle : r]] \implies \langle a,c \rangle : r \ O \ s$
 $\langle proof \rangle$

lemma *compE* [*elim!*]:
 $[[\langle xz \rangle : r \ O \ s;$
 $!!x \ y \ z. [[\langle xz \rangle = \langle x,z \rangle; \langle x,y \rangle : s; \langle y,z \rangle : r]] \implies P]]$
 $\implies P$
 $\langle proof \rangle$

lemma *compEpair*:
 $[[\langle a,c \rangle : r \ O \ s;$
 $!!y. [[\langle a,y \rangle : s; \langle y,c \rangle : r]] \implies P]]$
 $\implies P$
 $\langle proof \rangle$

lemma *converse-comp*: $\text{converse}(R \ O \ S) = \text{converse}(S) \ O \ \text{converse}(R)$
 $\langle proof \rangle$

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*: $\text{range}(r \ O \ s) \leq \text{range}(r)$
 $\langle proof \rangle$

lemma *range-comp-eq*: $\text{domain}(r) \leq \text{range}(s) \implies \text{range}(r \ O \ s) = \text{range}(r)$
 $\langle proof \rangle$

lemma *domain-comp*: $\text{domain}(r \ O \ s) \leq \text{domain}(s)$
 $\langle proof \rangle$

lemma *domain-comp-eq*: $\text{range}(s) \leq \text{domain}(r) \implies \text{domain}(r \ O \ s) = \text{domain}(s)$
 $\langle proof \rangle$

lemma *image-comp*: $(r \ O \ s)^{\text{``}A} = r^{\text{``}(s^{\text{``}A})$
 $\langle proof \rangle$

lemma *inj-inj-range*: $f : \text{inj}(A,B) \implies f : \text{inj}(A, \text{range}(f))$
 $\langle proof \rangle$

lemma *inj-bij-range*: $f : \text{inj}(A,B) \implies f : \text{bij}(A, \text{range}(f))$
 $\langle proof \rangle$

10.9 Other Results

lemma *comp-mono*: $[[r' \leq r; s' \leq s]] \implies (r' \ O \ s') \leq (r \ O \ s)$
<proof>

composition preserves relations

lemma *comp-rel*: $[[s \leq A * B; r \leq B * C]] \implies (r \ O \ s) \leq A * C$
<proof>

associative law for composition

lemma *comp-assoc*: $(r \ O \ s) \ O \ t = r \ O \ (s \ O \ t)$
<proof>

lemma *left-comp-id*: $r \leq A * B \implies id(B) \ O \ r = r$
<proof>

lemma *right-comp-id*: $r \leq A * B \implies r \ O \ id(A) = r$
<proof>

10.10 Composition Preserves Functions, Injections, and Surjections

lemma *comp-function*: $[[function(g); function(f)]] \implies function(f \ O \ g)$
<proof>

Don't think the premises can be weakened much

lemma *comp-fun*: $[[g: A \rightarrow B; f: B \rightarrow C]] \implies (f \ O \ g) : A \rightarrow C$
<proof>

lemma *comp-fun-apply* [*simp*]:
 $[[g: A \rightarrow B; a:A]] \implies (f \ O \ g)'a = f'(g'a)$
<proof>

Simplifies compositions of lambda-abstractions

lemma *comp-lam*:
 $[[!!x. x:A \implies b(x): B]] \implies (lam y:B. c(y)) \ O \ (lam x:A. b(x)) = (lam x:A. c(b(x)))$
<proof>

lemma *comp-inj*:
 $[[g: inj(A,B); f: inj(B,C)]] \implies (f \ O \ g) : inj(A,C)$
<proof>

lemma *comp-surj*:
 $[[g: surj(A,B); f: surj(B,C)]] \implies (f \ O \ g) : surj(A,C)$
<proof>

lemma *comp-bij*:

$\llbracket g: \text{bij}(A,B); f: \text{bij}(B,C) \rrbracket \implies (f \circ g) : \text{bij}(A,C)$
 $\langle \text{proof} \rangle$

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*:

$\llbracket (f \circ g): \text{inj}(A,C); g: A \rightarrow B; f: B \rightarrow C \rrbracket \implies g: \text{inj}(A,B)$
 $\langle \text{proof} \rangle$

lemma *comp-mem-injD2*:

$\llbracket (f \circ g): \text{inj}(A,C); g: \text{surj}(A,B); f: B \rightarrow C \rrbracket \implies f: \text{inj}(B,C)$
 $\langle \text{proof} \rangle$

lemma *comp-mem-surjD1*:

$\llbracket (f \circ g): \text{surj}(A,C); g: A \rightarrow B; f: B \rightarrow C \rrbracket \implies f: \text{surj}(B,C)$
 $\langle \text{proof} \rangle$

lemma *comp-mem-surjD2*:

$\llbracket (f \circ g): \text{surj}(A,C); g: A \rightarrow B; f: \text{inj}(B,C) \rrbracket \implies g: \text{surj}(A,B)$
 $\langle \text{proof} \rangle$

10.11.1 Inverses of Composition

left inverse of composition; one inclusion is $\text{@termf: } A \rightarrow B \implies \text{id}(A) \text{ ;= } \text{converse}(f) \circ f$

lemma *left-comp-inverse*: $f: \text{inj}(A,B) \implies \text{converse}(f) \circ f = \text{id}(A)$
 $\langle \text{proof} \rangle$

right inverse of composition; one inclusion is $\text{@termf: } A \rightarrow B \implies f \circ \text{converse}(f) \text{ ;= } \text{id}(B)$

lemma *right-comp-inverse*:

$f: \text{surj}(A,B) \implies f \circ \text{converse}(f) = \text{id}(B)$
 $\langle \text{proof} \rangle$

10.11.2 Proving that a Function is a Bijection

lemma *comp-eq-id-iff*:

$\llbracket f: A \rightarrow B; g: B \rightarrow A \rrbracket \implies f \circ g = \text{id}(B) \iff (\text{ALL } y: B. f'(g'y)=y)$
 $\langle \text{proof} \rangle$

lemma *fg-imp-bijective*:

$\llbracket f: A \rightarrow B; g: B \rightarrow A; f \circ g = \text{id}(B); g \circ f = \text{id}(A) \rrbracket \implies f : \text{bij}(A,B)$
 $\langle \text{proof} \rangle$

lemma *nilpotent-imp-bijective*: $[[f: A \rightarrow A; f \circ f = id(A)]] \implies f : bij(A,A)$
 $\langle proof \rangle$

lemma *invertible-imp-bijective*:
 $[[converse(f): B \rightarrow A; f: A \rightarrow B]] \implies f : bij(A,B)$
 $\langle proof \rangle$

10.11.3 Unions of Functions

See similar theorems in `func.thy`

Theorem by KG, proof by LCP

lemma *inj-disjoint-Un*:
 $[[f: inj(A,B); g: inj(C,D); B \text{ Int } D = 0]]$
 $\implies (lam a: A \text{ Un } C. if a:A then f'a else g'a) : inj(A \text{ Un } C, B \text{ Un } D)$
 $\langle proof \rangle$

lemma *surj-disjoint-Un*:
 $[[f: surj(A,B); g: surj(C,D); A \text{ Int } C = 0]]$
 $\implies (f \text{ Un } g) : surj(A \text{ Un } C, B \text{ Un } D)$
 $\langle proof \rangle$

A simple, high-level proof; the version for injections follows from it, using
`@termf:inj(A,B) |-> f:bij(A,range(f))`

lemma *bij-disjoint-Un*:
 $[[f: bij(A,B); g: bij(C,D); A \text{ Int } C = 0; B \text{ Int } D = 0]]$
 $\implies (f \text{ Un } g) : bij(A \text{ Un } C, B \text{ Un } D)$
 $\langle proof \rangle$

10.11.4 Restrictions as Surjections and Bijections

lemma *surj-image*:
 $f: Pi(A,B) \implies f: surj(A, f''A)$
 $\langle proof \rangle$

lemma *restrict-image [simp]*: $restrict(f,A) '' B = f '' (A \text{ Int } B)$
 $\langle proof \rangle$

lemma *restrict-inj*:
 $[[f: inj(A,B); C \leq A]] \implies restrict(f,C): inj(C,B)$
 $\langle proof \rangle$

lemma *restrict-surj*: $[[f: Pi(A,B); C \leq A]] \implies restrict(f,C): surj(C, f''C)$
 $\langle proof \rangle$

lemma *restrict-bij*:
 $[[f: inj(A,B); C \leq A]] \implies restrict(f,C): bij(C, f''C)$
 $\langle proof \rangle$

10.11.5 Lemmas for Ramsey's Theorem

lemma *inj-weaken-type*: $[| f: inj(A,B); B \leq D |] \implies f: inj(A,D)$
<proof>

lemma *inj-succ-restrict*:
 $[| f: inj(succ(m), A) |] \implies restrict(f,m) : inj(m, A - \{f^m\})$
<proof>

lemma *inj-extend*:
 $[| f: inj(A,B); a \sim A; b \sim B |]$
 $\implies cons(\langle a, b \rangle, f) : inj(cons(a,A), cons(b,B))$
<proof>

end

11 Trancl: Relations: Their General Properties and Transitive Closure

theory *Trancl* imports *Fixedpt Perm* **begin**

definition
refl :: $[i, i] \implies o$ **where**
 $refl(A, r) == (ALL x: A. \langle x, x \rangle : r)$

definition
irrefl :: $[i, i] \implies o$ **where**
 $irrefl(A, r) == ALL x: A. \langle x, x \rangle \sim : r$

definition
sym :: $i \implies o$ **where**
 $sym(r) == ALL x y. \langle x, y \rangle : r \dashrightarrow \langle y, x \rangle : r$

definition
asym :: $i \implies o$ **where**
 $asym(r) == ALL x y. \langle x, y \rangle : r \dashrightarrow \sim \langle y, x \rangle : r$

definition
antisym :: $i \implies o$ **where**
 $antisym(r) == ALL x y. \langle x, y \rangle : r \dashrightarrow \langle y, x \rangle : r \dashrightarrow x = y$

definition
trans :: $i \implies o$ **where**
 $trans(r) == ALL x y z. \langle x, y \rangle : r \dashrightarrow \langle y, z \rangle : r \dashrightarrow \langle x, z \rangle : r$

definition
trans-on :: $[i, i] \implies o$ (*trans[-]'(-')*) **where**

$trans[A](r) == ALL\ x:A.\ ALL\ y:A.\ ALL\ z:A.\ \langle x,y \rangle: r \dashrightarrow \langle y,z \rangle: r \dashrightarrow \langle x,z \rangle: r$

definition

$rtranc\ ::\ i=>i\ ((-\hat{*})\ [100]\ 100)\ \mathbf{where}$
 $r\hat{*} == lfp(field(r)*field(r), \%s.\ id(field(r))\ Un\ (r\ O\ s))$

definition

$tranc\ ::\ i=>i\ ((-\hat{+})\ [100]\ 100)\ \mathbf{where}$
 $r\hat{+} == r\ O\ r\hat{*}$

definition

$equiv\ ::\ [i,i]=>o\ \mathbf{where}$
 $equiv(A,r) == r\ <= A*A\ \&\ refl(A,r)\ \&\ sym(r)\ \&\ trans(r)$

11.1 General properties of relations

11.1.1 irreflexivity

lemma *irreflI*:

$[[\ !\!x.\ x:A ==> \langle x,x \rangle \sim: r\]] ==> irrefl(A,r)$
 $\langle proof \rangle$

lemma *irreflE*: $[[\ irrefl(A,r);\ x:A\]] ==> \langle x,x \rangle \sim: r$

$\langle proof \rangle$

11.1.2 symmetry

lemma *symI*:

$[[\ !\!x\ y.\ \langle x,y \rangle: r ==> \langle y,x \rangle: r\]] ==> sym(r)$
 $\langle proof \rangle$

lemma *symE*: $[[\ sym(r);\ \langle x,y \rangle: r\]] ==> \langle y,x \rangle: r$

$\langle proof \rangle$

11.1.3 antisymmetry

lemma *antisymI*:

$[[\ !\!x\ y.\ [\ \langle x,y \rangle: r;\ \langle y,x \rangle: r\] ==> x=y\]] ==> antisym(r)$
 $\langle proof \rangle$

lemma *antisymE*: $[[\ antisym(r);\ \langle x,y \rangle: r;\ \langle y,x \rangle: r\]] ==> x=y$

$\langle proof \rangle$

11.1.4 transitivity

lemma *transD*: $[[\ trans(r);\ \langle a,b \rangle: r;\ \langle b,c \rangle: r\]] ==> \langle a,c \rangle: r$

$\langle proof \rangle$

lemma *trans-onD*:

$\llbracket \text{trans}[A](r); \langle a,b \rangle : r; \langle b,c \rangle : r; a:A; b:A; c:A \rrbracket \implies \langle a,c \rangle : r$
 $\langle \text{proof} \rangle$

lemma *trans-imp-trans-on*: $\text{trans}(r) \implies \text{trans}[A](r)$
 $\langle \text{proof} \rangle$

lemma *trans-on-imp-trans*: $\llbracket \text{trans}[A](r); r \leq A * A \rrbracket \implies \text{trans}(r)$
 $\langle \text{proof} \rangle$

11.2 Transitive closure of a relation

lemma *rtrancl-bnd-mono*:
 $\text{bnd-mono}(\text{field}(r) * \text{field}(r), \%s. \text{id}(\text{field}(r)) \text{Un } (r \text{ O } s))$
 $\langle \text{proof} \rangle$

lemma *rtrancl-mono*: $r \leq s \implies r^{\wedge *} \leq s^{\wedge *}$
 $\langle \text{proof} \rangle$

lemmas *rtrancl-unfold* =
 $\text{rtrancl-bnd-mono} \text{ [THEN rtrancl-def [THEN def-lfp-unfold], standard]}$

lemmas *rtrancl-type* = $\text{rtrancl-def} \text{ [THEN def-lfp-subset, standard]}$

lemma *relation-rtrancl*: $\text{relation}(r^{\wedge *})$
 $\langle \text{proof} \rangle$

lemma *rtrancl-refl*: $\llbracket a : \text{field}(r) \rrbracket \implies \langle a,a \rangle : r^{\wedge *}$
 $\langle \text{proof} \rangle$

lemma *rtrancl-into-rtrancl*: $\llbracket \langle a,b \rangle : r^{\wedge *}; \langle b,c \rangle : r \rrbracket \implies \langle a,c \rangle : r^{\wedge *}$
 $\langle \text{proof} \rangle$

lemma *r-into-rtrancl*: $\langle a,b \rangle : r \implies \langle a,b \rangle : r^{\wedge *}$
 $\langle \text{proof} \rangle$

lemma *r-subset-rtrancl*: $\text{relation}(r) \implies r \leq r^{\wedge *}$
 $\langle \text{proof} \rangle$

lemma *rtrancl-field*: $\text{field}(r^{\wedge *}) = \text{field}(r)$
 $\langle \text{proof} \rangle$

lemma *rtrancl-full-induct* [*case-names initial step, consumes 1*]:

[[$\langle a, b \rangle : r^*$;
 !! $x. x: \text{field}(r) \implies P(\langle x, x \rangle)$;
 !! $x y z. [P(\langle x, y \rangle); \langle x, y \rangle : r^*; \langle y, z \rangle : r]$ $\implies P(\langle x, z \rangle)$]]
 $\implies P(\langle a, b \rangle)$
 <proof>

lemma *rtrancl-induct* [*case-names initial step, induct set: rtrancl*]:

[[$\langle a, b \rangle : r^*$;
 $P(a)$;
 !! $y z. [\langle a, y \rangle : r^*; \langle y, z \rangle : r; P(y)] \implies P(z)$
]]
 $\implies P(b)$

<proof>

lemma *trans-rtrancl*: $\text{trans}(r^*)$

<proof>

lemmas *rtrancl-trans = trans-rtrancl* [*THEN transD, standard*]

lemma *rtranclE*:

[[$\langle a, b \rangle : r^*; (a=b) \implies P$;
 !! $y. [\langle a, y \rangle : r^*; \langle y, b \rangle : r] \implies P$]]
 $\implies P$
 <proof>

lemma *trans-trancl*: $\text{trans}(r^+)$

<proof>

lemmas *trans-on-trancl = trans-trancl* [*THEN trans-imp-trans-on*]

lemmas *trancl-trans = trans-trancl* [*THEN transD, standard*]

lemma *trancl-into-rtrancl*: $\langle a, b \rangle : r^+ \implies \langle a, b \rangle : r^*$

<proof>

lemma *r-into-trancl*: $\langle a,b \rangle : r \implies \langle a,b \rangle : r^+$
 $\langle proof \rangle$

lemma *r-subset-trancl*: $relation(r) \implies r \leq r^+$
 $\langle proof \rangle$

lemma *rtrancl-into-trancl1*: $[\langle a,b \rangle : r^*; \langle b,c \rangle : r] \implies \langle a,c \rangle : r^+$
 $\langle proof \rangle$

lemma *rtrancl-into-trancl2*:
 $[\langle a,b \rangle : r; \langle b,c \rangle : r^*] \implies \langle a,c \rangle : r^+$
 $\langle proof \rangle$

lemma *trancl-induct* [*case-names initial step, induct set: trancl*]:
 $[\langle a,b \rangle : r^+;$
 $\quad !!y. [\langle a,y \rangle : r] \implies P(y);$
 $\quad !!y z. [\langle a,y \rangle : r^+; \langle y,z \rangle : r; P(y)] \implies P(z)$
 $]\implies P(b)$
 $\langle proof \rangle$

lemma *tranclE*:
 $[\langle a,b \rangle : r^+;$
 $\quad \langle a,b \rangle : r \implies P;$
 $\quad !!y. [\langle a,y \rangle : r^+; \langle y,b \rangle : r] \implies P$
 $]\implies P$
 $\langle proof \rangle$

lemma *trancl-type*: $r^+ \leq field(r)*field(r)$
 $\langle proof \rangle$

lemma *relation-trancl*: $relation(r^+)$
 $\langle proof \rangle$

lemma *trancl-subset-times*: $r \subseteq A * A \implies r^+ \subseteq A * A$
 $\langle proof \rangle$

lemma *trancl-mono*: $r \leq s \implies r^+ \leq s^+$
 $\langle proof \rangle$

lemma *trancl-eq-r*: $[relation(r); trans(r)] \implies r^+ = r$
 $\langle proof \rangle$

lemma *rtrancl-idemp* [*simp*]: $(r^{\wedge*})^{\wedge*} = r^{\wedge*}$
<proof>

lemma *rtrancl-subset*: $[[R \leq S; S \leq R^{\wedge*}]] \implies S^{\wedge*} = R^{\wedge*}$
<proof>

lemma *rtrancl-Un-rtrancl*:
 $[[\text{relation}(r); \text{relation}(s)]] \implies (r^{\wedge*} \cup s^{\wedge*})^{\wedge*} = (r \cup s)^{\wedge*}$
<proof>

lemma *rtrancl-converseD*: $\langle x, y \rangle : \text{converse}(r)^{\wedge*} \implies \langle x, y \rangle : \text{converse}(r^{\wedge*})$
<proof>

lemma *rtrancl-converseI*: $\langle x, y \rangle : \text{converse}(r^{\wedge*}) \implies \langle x, y \rangle : \text{converse}(r)^{\wedge*}$
<proof>

lemma *rtrancl-converse*: $\text{converse}(r)^{\wedge*} = \text{converse}(r^{\wedge*})$
<proof>

lemma *trancl-converseD*: $\langle a, b \rangle : \text{converse}(r)^{\wedge+} \implies \langle a, b \rangle : \text{converse}(r^{\wedge+})$
<proof>

lemma *trancl-converseI*: $\langle x, y \rangle : \text{converse}(r^{\wedge+}) \implies \langle x, y \rangle : \text{converse}(r)^{\wedge+}$
<proof>

lemma *trancl-converse*: $\text{converse}(r)^{\wedge+} = \text{converse}(r^{\wedge+})$
<proof>

lemma *converse-trancl-induct* [*case-names initial step, consumes 1*]:
 $[[\langle a, b \rangle : r^{\wedge+}; !!y. \langle y, b \rangle : r \implies P(y);$
 $!!y z. [[\langle y, z \rangle : r; \langle z, b \rangle : r^{\wedge+}; P(z)]] \implies P(y)]]$
 $\implies P(a)$
<proof>

end

12 WF: Well-Founded Recursion

theory *WF* imports *Trancl* begin

definition

$wf \quad :: i \Rightarrow o$ **where**

$$wf(r) == ALL Z. Z=0 \mid (EX x:Z. ALL y. \langle y, x \rangle : r \dashrightarrow \sim y : Z)$$

definition

$wf\text{-on} \quad :: [i, i] \Rightarrow o \quad (wf[-]'(-'))$ **where**

$$wf\text{-on}(A, r) == wf(r \text{ Int } A * A)$$

definition

$is\text{-recfun} \quad :: [i, i, [i, i] \Rightarrow i, i] \Rightarrow o$ **where**

$$is\text{-recfun}(r, a, H, f) == (f = (lam x: r - \{a\}. H(x, restrict(f, r - \{x\}))))$$

definition

$the\text{-recfun} \quad :: [i, i, [i, i] \Rightarrow i] \Rightarrow i$ **where**

$$the\text{-recfun}(r, a, H) == (THE f. is\text{-recfun}(r, a, H, f))$$

definition

$wftrec \quad :: [i, i, [i, i] \Rightarrow i] \Rightarrow i$ **where**

$$wftrec(r, a, H) == H(a, the\text{-recfun}(r, a, H))$$

definition

$wfrec \quad :: [i, i, [i, i] \Rightarrow i] \Rightarrow i$ **where**

$$wfrec(r, a, H) == wftrec(r \hat{+}, a, \%x f. H(x, restrict(f, r - \{x\})))$$

definition

$wfrec\text{-on} \quad :: [i, i, i, [i, i] \Rightarrow i] \Rightarrow i \quad (wfrec[-]'(-, -, -))$ **where**

$$wfrec[A](r, a, H) == wfrec(r \text{ Int } A * A, a, H)$$

12.1 Well-Founded Relations

12.1.1 Equivalences between wf and $wf\text{-on}$

lemma $wf\text{-imp-}wf\text{-on}$: $wf(r) \Rightarrow wf[A](r)$

$\langle proof \rangle$

lemma $wf\text{-on-imp-}wf$: $[wf[A](r); r \leq A * A] \Rightarrow wf(r)$

$\langle proof \rangle$

lemma $wf\text{-on-field-imp-}wf$: $wf[field(r)](r) \Rightarrow wf(r)$

$\langle proof \rangle$

lemma $wf\text{-iff-}wf\text{-on-field}$: $wf(r) \Leftrightarrow wf[field(r)](r)$

$\langle proof \rangle$

lemma $wf\text{-on-subset-}A$: $[wf[A](r); B \leq A] \Rightarrow wf[B](r)$

$\langle proof \rangle$

lemma *wf-on-subset-r*: $[[\text{wf}[A](r); s \leq r]] \implies \text{wf}[A](s)$
 $\langle \text{proof} \rangle$

lemma *wf-subset*: $[[\text{wf}(s); r \leq s]] \implies \text{wf}(r)$
 $\langle \text{proof} \rangle$

12.1.2 Introduction Rules for *wf-on*

If every non-empty subset of A has an r -minimal element then we have $\text{wf}[A](r)$.

lemma *wf-onI*:
assumes *prem*: $!!Z u. [[Z \leq A; u:Z; \text{ALL } x:Z. \text{EX } y:Z. \langle y, x \rangle : r]] \implies \text{False}$
shows $\text{wf}[A](r)$
 $\langle \text{proof} \rangle$

If r allows well-founded induction over A then we have $\text{wf}[A](r)$. 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*: $!!y B. [[\text{ALL } x:A. (\text{ALL } y:A. \langle y, x \rangle : r \dashrightarrow y:B) \dashrightarrow x:B; y:A]]$
 $\implies y:B$
shows $\text{wf}[A](r)$
 $\langle \text{proof} \rangle$

12.1.3 Well-founded Induction

Consider the least z in $\text{domain}(r)$ such that $P(z)$ does not hold...

lemma *wf-induct* [*induct set*: *wf*]:
 $[[\text{wf}(r);$
 $!!x. [[\text{ALL } y. \langle y, x \rangle : r \dashrightarrow P(y)]] \implies P(x)]]$
 $\implies P(a)$
 $\langle \text{proof} \rangle$

lemmas *wf-induct-rule* = *wf-induct* [*rule-format*, *induct set*: *wf*]

The form of this rule is designed to match *wfI*

lemma *wf-induct2*:
 $[[\text{wf}(r); a:A; \text{field}(r) \leq A;$
 $!!x. [[x:A; \text{ALL } y. \langle y, x \rangle : r \dashrightarrow P(y)]] \implies P(x)]]$
 $\implies P(a)$
 $\langle \text{proof} \rangle$

lemma *field-Int-square*: $\text{field}(r \text{ Int } A * A) \leq A$
 $\langle \text{proof} \rangle$

lemma *wf-on-induct* [*consumes 2*, *induct set*: *wf-on*]:

$$\begin{aligned} & \llbracket wf[A](r); a:A; \\ & \quad !!x.\llbracket x:A; ALL y:A. \langle y,x \rangle:r \dashrightarrow P(y) \rrbracket \implies P(x) \\ & \rrbracket \implies P(a) \end{aligned}$$
 <proof>

lemmas *wf-on-induct-rule* =
wf-on-induct [rule-format, consumes 2, induct set: wf-on]

If r allows well-founded induction then we have $wf(r)$.

lemma *wfI*:

$$\begin{aligned} & \llbracket field(r) \leq A; \\ & \quad !!y B. \llbracket ALL x:A. (ALL y:A. \langle y,x \rangle:r \dashrightarrow y:B) \dashrightarrow x:B; y:A \rrbracket \\ & \quad \implies y:B \rrbracket \\ & \implies wf(r) \end{aligned}$$
 <proof>

12.2 Basic Properties of Well-Founded Relations

lemma *wf-not-refl*: $wf(r) \implies \langle a,a \rangle \sim : r$
 <proof>

lemma *wf-not-sym* [rule-format]: $wf(r) \implies ALL x. \langle a,x \rangle:r \dashrightarrow \langle x,a \rangle \sim : r$
 <proof>

lemmas *wf-asy*m = *wf-not-sym [THEN swap, standard]*

lemma *wf-on-not-refl*: $\llbracket wf[A](r); a:A \rrbracket \implies \langle a,a \rangle \sim : r$
 <proof>

lemma *wf-on-not-sym* [rule-format]:

$$\llbracket wf[A](r); a:A \rrbracket \implies ALL b:A. \langle a,b \rangle:r \dashrightarrow \langle b,a \rangle \sim : r$$
 <proof>

lemma *wf-on-asy*m:

$$\begin{aligned} & \llbracket wf[A](r); \sim Z \implies \langle a,b \rangle : r; \\ & \quad \langle b,a \rangle \sim : r \implies Z; \sim Z \implies a : A; \sim Z \implies b : A \rrbracket \implies Z \end{aligned}$$
 <proof>

lemma *wf-on-chain3*:

$$\llbracket wf[A](r); \langle a,b \rangle:r; \langle b,c \rangle:r; \langle c,a \rangle:r; a:A; b:A; c:A \rrbracket \implies P$$
 <proof>

transitive closure of a WF relation is WF provided A is downward closed

lemma *wf-on-trancl*:

$$\llbracket wf[A](r); r - "A \leq A \rrbracket \implies wf[A](r^+)$$
 <proof>

lemma *wf-trancl*: $wf(r) \implies wf(r^+)$

<proof>

r – “ $\{a\}$ is the set of everything under a in r ”

lemmas *underI* = *vimage-singleton-iff* [THEN *iffD2*, *standard*]

lemmas *underD* = *vimage-singleton-iff* [THEN *iffD1*, *standard*]

12.3 The Predicate *is-recfun*

lemma *is-recfun-type*: $is-recfun(r,a,H,f) \implies f: r - \{a\} \rightarrow range(f)$

<proof>

lemmas *is-recfun-imp-function* = *is-recfun-type* [THEN *fun-is-function*]

lemma *apply-recfun*:

$[[is-recfun(r,a,H,f); \langle x,a \rangle : r]] \implies f'x = H(x, restrict(f, r - \{x\}))$

<proof>

lemma *is-recfun-equal* [rule-format]:

$[[wf(r); trans(r); is-recfun(r,a,H,f); is-recfun(r,b,H,g)]]$

$\implies \langle x,a \rangle : r \dashrightarrow \langle x,b \rangle : r \dashrightarrow f'x = g'x$

<proof>

lemma *is-recfun-cut*:

$[[wf(r); trans(r);$

$is-recfun(r,a,H,f); is-recfun(r,b,H,g); \langle b,a \rangle : r]]$

$\implies restrict(f, r - \{b\}) = g$

<proof>

12.4 Recursion: Main Existence Lemma

lemma *is-recfun-functional*:

$[[wf(r); trans(r); is-recfun(r,a,H,f); is-recfun(r,a,H,g)]] \implies f = g$

<proof>

lemma *the-recfun-eq*:

$[[is-recfun(r,a,H,f); wf(r); trans(r)]] \implies the-recfun(r,a,H) = f$

<proof>

lemma *is-the-recfun*:

$[[is-recfun(r,a,H,f); wf(r); trans(r)]]$

$\implies is-recfun(r, a, H, the-recfun(r,a,H))$

<proof>

lemma *unfold-the-recfun*:

$[[wf(r); trans(r)]] \implies is-recfun(r, a, H, the-recfun(r,a,H))$

<proof>

12.5 Unfolding $wftrec(r, a, H)$

lemma *the-recfun-cut*:

$$\llbracket wf(r); trans(r); \langle b, a \rangle : r \rrbracket$$
$$\implies restrict(the-recfun(r, a, H), r - \{\!-\!\} \{b\}) = the-recfun(r, b, H)$$

<proof>

lemma *wftrec*:

$$\llbracket wf(r); trans(r) \rrbracket \implies$$
$$wftrec(r, a, H) = H(a, lam\ x: r - \{\!-\!\} \{a\}. wftrec(r, x, H))$$

<proof>

12.5.1 Removal of the Premise $trans(r)$

lemma *wfrec*:

$$wf(r) \implies wfrec(r, a, H) = H(a, lam\ x: r - \{\!-\!\} \{a\}. wfrec(r, x, H))$$

<proof>

lemma *def-wfrec*:

$$\llbracket !!x. h(x) == wfrec(r, x, H); wf(r) \rrbracket \implies$$
$$h(a) = H(a, lam\ x: r - \{\!-\!\} \{a\}. h(x))$$

<proof>

lemma *wfrec-type*:

$$\llbracket wf(r); a:A; field(r) \leq A;$$
$$!!x\ u. \llbracket x:A; u: Pi(r - \{\!-\!\} \{x\}, B) \rrbracket \implies H(x, u) : B(x)$$
$$\rrbracket \implies wfrec(r, a, H) : B(a)$$

<proof>

lemma *wfrec-on*:

$$\llbracket wf[A](r); a:A \rrbracket \implies$$
$$wfrec[A](r, a, H) = H(a, lam\ x: (r - \{\!-\!\} \{a\})\ Int\ A. wfrec[A](r, x, H))$$

<proof>

Minimal-element characterization of well-foundedness

lemma *wf-eq-minimal*:

$$wf(r) \iff (ALL\ Q\ x. x:Q \implies (EX\ z:Q. ALL\ y. \langle y, z \rangle : r \implies y \sim Q))$$

<proof>

end

13 Ordinal: Transitive Sets and Ordinals

theory *Ordinal imports WF Bool equalities begin*

definition

$Memrel \quad :: i=>i \text{ where}$
 $Memrel(A) \quad == \{z: A*A . EX x y. z=<x,y> \ \& \ x:y \}$

definition

$Transset \quad :: i=>o \text{ where}$
 $Transset(i) \quad == ALL x:i. x<=i$

definition

$Ord \quad :: i=>o \text{ where}$
 $Ord(i) \quad == Transset(i) \ \& \ (ALL x:i. Transset(x))$

definition

$lt \quad :: [i,i] => o \text{ (infixl } < 50) \quad \text{where}$
 $i<j \quad == i:j \ \& \ Ord(j)$

definition

$Limit \quad :: i=>o \text{ where}$
 $Limit(i) \quad == Ord(i) \ \& \ 0<i \ \& \ (ALL y. y<i \ \longrightarrow \ succ(y)<i)$

abbreviation

$le \text{ (infixl } le \ 50) \text{ where}$
 $x \ le \ y \ == x < succ(y)$

notation (*xsymbols*)

$le \text{ (infixl } \leq 50)$

notation (*HTML output*)

$le \text{ (infixl } \leq 50)$

13.1 Rules for Transset

13.1.1 Three Neat Characterisations of Transset

lemma *Transset-iff-Pow*: $Transset(A) \longleftrightarrow A \leq Pow(A)$
<proof>

lemma *Transset-iff-Union-succ*: $Transset(A) \longleftrightarrow Union(succ(A)) = A$
<proof>

lemma *Transset-iff-Union-subset*: $Transset(A) \longleftrightarrow Union(A) \leq A$
<proof>

13.1.2 Consequences of Downwards Closure

lemma *Transset-doubleton-D*:
 $[[Transset(C); \{a,b\}: C]] \implies a:C \ \& \ b: C$
<proof>

lemma *Transset-Pair-D*:

$\llbracket \text{Transset}(C); \langle a, b \rangle : C \rrbracket \implies a : C \ \& \ b : C$
<proof>

lemma *Transset-includes-domain:*

$\llbracket \text{Transset}(C); A * B \leq C; b : B \rrbracket \implies A \leq C$
<proof>

lemma *Transset-includes-range:*

$\llbracket \text{Transset}(C); A * B \leq C; a : A \rrbracket \implies B \leq C$
<proof>

13.1.3 Closure Properties

lemma *Transset-0: Transset(0)*

<proof>

lemma *Transset-Un:*

$\llbracket \text{Transset}(i); \text{Transset}(j) \rrbracket \implies \text{Transset}(i \text{ Un } j)$
<proof>

lemma *Transset-Int:*

$\llbracket \text{Transset}(i); \text{Transset}(j) \rrbracket \implies \text{Transset}(i \text{ Int } j)$
<proof>

lemma *Transset-succ: Transset(i) ==> Transset(succ(i))*

<proof>

lemma *Transset-Pow: Transset(i) ==> Transset(Pow(i))*

<proof>

lemma *Transset-Union: Transset(A) ==> Transset(Union(A))*

<proof>

lemma *Transset-Union-family:*

$\llbracket \forall i. i : A \implies \text{Transset}(i) \rrbracket \implies \text{Transset}(\text{Union}(A))$
<proof>

lemma *Transset-Inter-family:*

$\llbracket \forall i. i : A \implies \text{Transset}(i) \rrbracket \implies \text{Transset}(\text{Inter}(A))$
<proof>

lemma *Transset-UN:*

$(\forall x. x \in A \implies \text{Transset}(B(x))) \implies \text{Transset}(\bigcup_{x \in A} B(x))$
<proof>

lemma *Transset-INT:*

$(\forall x. x \in A \implies \text{Transset}(B(x))) \implies \text{Transset}(\bigcap_{x \in A} B(x))$
<proof>

13.2 Lemmas for Ordinals

lemma *OrdI*:

$\llbracket \text{Transset}(i); \forall x. x:i \implies \text{Transset}(x) \rrbracket \implies \text{Ord}(i)$
<proof>

lemma *Ord-is-Transset*: $\text{Ord}(i) \implies \text{Transset}(i)$

<proof>

lemma *Ord-contains-Transset*:

$\llbracket \text{Ord}(i); j:i \rrbracket \implies \text{Transset}(j)$
<proof>

lemma *Ord-in-Ord*: $\llbracket \text{Ord}(i); j:i \rrbracket \implies \text{Ord}(j)$

<proof>

lemma *Ord-in-Ord'*: $\llbracket j:i; \text{Ord}(i) \rrbracket \implies \text{Ord}(j)$

<proof>

lemmas *Ord-succD = Ord-in-Ord* [*OF - succI1*]

lemma *Ord-subset-Ord*: $\llbracket \text{Ord}(i); \text{Transset}(j); j \leq i \rrbracket \implies \text{Ord}(j)$

<proof>

lemma *OrdmemD*: $\llbracket j:i; \text{Ord}(i) \rrbracket \implies j \leq i$

<proof>

lemma *Ord-trans*: $\llbracket i:j; j:k; \text{Ord}(k) \rrbracket \implies i:k$

<proof>

lemma *Ord-succ-subsetI*: $\llbracket i:j; \text{Ord}(j) \rrbracket \implies \text{succ}(i) \leq j$

<proof>

13.3 The Construction of Ordinals: 0, succ, Union

lemma *Ord-0* [*iff,TC*]: $\text{Ord}(0)$

<proof>

lemma *Ord-succ* [*TC*]: $\text{Ord}(i) \implies \text{Ord}(\text{succ}(i))$

<proof>

lemmas *Ord-1 = Ord-0* [*THEN Ord-succ*]

lemma *Ord-succ-iff* [*iff*]: $\text{Ord}(\text{succ}(i)) \iff \text{Ord}(i)$

<proof>

lemma *Ord-Un* [*intro,simp,TC*]: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \text{Ord}(i \cup j)$

<proof>

lemma *Ord-Int* [TC]: $[[\text{Ord}(i); \text{Ord}(j)]] \implies \text{Ord}(i \text{ Int } j)$
<proof>

lemma *ON-class*: $\sim (\text{ALL } i. i:X \iff \text{Ord}(i))$
<proof>

13.4 \lessdot is 'less Than' for Ordinals

lemma *ltI*: $[[i:j; \text{Ord}(j)]] \implies i < j$
<proof>

lemma *ltE*:
 $[[i < j; [[i:j; \text{Ord}(i); \text{Ord}(j)]] \implies P]] \implies P$
<proof>

lemma *ltD*: $i < j \implies i:j$
<proof>

lemma *not-lt0* [simp]: $\sim i < 0$
<proof>

lemma *lt-Ord*: $j < i \implies \text{Ord}(j)$
<proof>

lemma *lt-Ord2*: $j < i \implies \text{Ord}(i)$
<proof>

lemmas *le-Ord2* = *lt-Ord2* [THEN *Ord-succD*]

lemmas *lt0E* = *not-lt0* [THEN *notE*, *elim!*]

lemma *lt-trans*: $[[i < j; j < k]] \implies i < k$
<proof>

lemma *lt-not-sym*: $i < j \implies \sim (j < i)$
<proof>

lemmas *lt-asym* = *lt-not-sym* [THEN *swap*]

lemma *lt-irrefl* [*elim!*]: $i < i \implies P$
<proof>

lemma *lt-not-refl*: $\sim i < i$

$\langle proof \rangle$

lemma *le-iff*: $i \text{ le } j \leftrightarrow i < j \mid (i=j \ \& \ \text{Ord}(j))$
 $\langle proof \rangle$

lemma *leI*: $i < j \implies i \text{ le } j$
 $\langle proof \rangle$

lemma *le-eqI*: $[\mid i=j; \ \text{Ord}(j) \mid] \implies i \text{ le } j$
 $\langle proof \rangle$

lemmas *le-refl = refl* [THEN *le-eqI*]

lemma *le-refl-iff* [*iff*]: $i \text{ le } i \leftrightarrow \text{Ord}(i)$
 $\langle proof \rangle$

lemma *leCI*: $(\sim (i=j \ \& \ \text{Ord}(j)) \implies i < j) \implies i \text{ le } j$
 $\langle proof \rangle$

lemma *leE*:
 $[\mid i \text{ le } j; \ i < j \implies P; \ \mid i=j; \ \text{Ord}(j) \mid] \implies P \mid] \implies P$
 $\langle proof \rangle$

lemma *le-anti-sym*: $[\mid i \text{ le } j; \ j \text{ le } i \mid] \implies i=j$
 $\langle proof \rangle$

lemma *le0-iff* [*simp*]: $i \text{ le } 0 \leftrightarrow i=0$
 $\langle proof \rangle$

lemmas *le0D = le0-iff* [THEN *iffD1*, *dest!*]

13.5 Natural Deduction Rules for Memrel

lemma *Memrel-iff* [*simp*]: $\langle a,b \rangle : \text{Memrel}(A) \leftrightarrow a:b \ \& \ a:A \ \& \ b:A$
 $\langle proof \rangle$

lemma *MemrelI* [*intro!*]: $[\mid a: b; \ a: A; \ b: A \mid] \implies \langle a,b \rangle : \text{Memrel}(A)$
 $\langle proof \rangle$

lemma *MemrelE* [*elim!*]:
 $[\mid \langle a,b \rangle : \text{Memrel}(A);$
 $\quad [\mid a: A; \ b: A; \ a:b \mid] \implies P \mid]$
 $\implies P$
 $\langle proof \rangle$

lemma *Memrel-type*: $\text{Memrel}(A) \leq A * A$
<proof>

lemma *Memrel-mono*: $A \leq B \implies \text{Memrel}(A) \leq \text{Memrel}(B)$
<proof>

lemma *Memrel-0* [*simp*]: $\text{Memrel}(0) = 0$
<proof>

lemma *Memrel-1* [*simp*]: $\text{Memrel}(1) = 0$
<proof>

lemma *relation-Memrel*: $\text{relation}(\text{Memrel}(A))$
<proof>

lemma *wf-Memrel*: $\text{wf}(\text{Memrel}(A))$
<proof>

The premise $\text{Ord}(i)$ does not suffice.

lemma *trans-Memrel*:
 $\text{Ord}(i) \implies \text{trans}(\text{Memrel}(i))$
<proof>

However, the following premise is strong enough.

lemma *Transset-trans-Memrel*:
 $\forall j \in i. \text{Transset}(j) \implies \text{trans}(\text{Memrel}(i))$
<proof>

lemma *Transset-Memrel-iff*:
 $\text{Transset}(A) \implies \langle a, b \rangle : \text{Memrel}(A) \iff a : b \ \& \ b : A$
<proof>

13.6 Transfinite Induction

lemma *Transset-induct*:
 $\llbracket i : k; \text{Transset}(k);$
 $\quad \llbracket x : k; \text{ALL } y : x. P(y) \rrbracket \implies P(x) \rrbracket$
 $\implies P(i)$
<proof>

lemmas *Ord-induct* [*consumes 2*] = *Transset-induct* [*OF - Ord-is-Transset*]

lemmas *Ord-induct-rule* = *Ord-induct* [*rule-format, consumes 2*]

lemma *trans-induct* [*consumes 1*]:

$$\begin{aligned} & \llbracket \text{Ord}(i); \\ & \quad !!x. \llbracket \text{Ord}(x); \text{ALL } y:x. P(y) \rrbracket \implies P(x) \rrbracket \\ & \implies P(i) \\ \langle \text{proof} \rangle \end{aligned}$$

lemmas *trans-induct-rule* = *trans-induct* [*rule-format*, *consumes 1*]

13.6.1 Proving That \mathfrak{i} is a Linear Ordering on the Ordinals

lemma *Ord-linear* [*rule-format*]:

$$\text{Ord}(i) \implies (\text{ALL } j. \text{Ord}(j) \longrightarrow i:j \mid i=j \mid j:i)$$
 $\langle \text{proof} \rangle$

lemma *Ord-linear-lt*:

$$\llbracket \text{Ord}(i); \text{Ord}(j); i < j \implies P; i = j \implies P; j < i \implies P \rrbracket \implies P$$
 $\langle \text{proof} \rangle$

lemma *Ord-linear2*:

$$\llbracket \text{Ord}(i); \text{Ord}(j); i < j \implies P; j \text{ le } i \implies P \rrbracket \implies P$$
 $\langle \text{proof} \rangle$

lemma *Ord-linear-le*:

$$\llbracket \text{Ord}(i); \text{Ord}(j); i \text{ le } j \implies P; j \text{ le } i \implies P \rrbracket \implies P$$
 $\langle \text{proof} \rangle$

lemma *le-imp-not-lt*: $j \text{ le } i \implies \sim i < j$

$\langle \text{proof} \rangle$

lemma *not-lt-imp-le*: $\llbracket \sim i < j; \text{Ord}(i); \text{Ord}(j) \rrbracket \implies j \text{ le } i$

$\langle \text{proof} \rangle$

13.6.2 Some Rewrite Rules for \mathfrak{i} , le

lemma *Ord-mem-iff-lt*: $\text{Ord}(j) \implies i:j \longleftrightarrow i < j$

$\langle \text{proof} \rangle$

lemma *not-lt-iff-le*: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \sim i < j \longleftrightarrow j \text{ le } i$

$\langle \text{proof} \rangle$

lemma *not-le-iff-lt*: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \sim i \text{ le } j \longleftrightarrow j < i$

$\langle \text{proof} \rangle$

lemma *Ord-0-le*: $\text{Ord}(i) \implies 0 \text{ le } i$

$\langle \text{proof} \rangle$

lemma *Ord-0-lt*: $\llbracket \text{Ord}(i); i \sim 0 \rrbracket \implies 0 < i$

$\langle \text{proof} \rangle$

lemma *Ord-0-lt-iff*: $Ord(i) \implies i \sim 0 \iff 0 < i$
<proof>

13.7 Results about Less-Than or Equals

lemma *zero-le-succ-iff* [*iff*]: $0 \text{ le } succ(x) \iff Ord(x)$
<proof>

lemma *subset-imp-le*: $[[j <= i; Ord(i); Ord(j)]] \implies j \text{ le } i$
<proof>

lemma *le-imp-subset*: $i \text{ le } j \implies i <= j$
<proof>

lemma *le-subset-iff*: $j \text{ le } i \iff j <= i \ \& \ Ord(i) \ \& \ Ord(j)$
<proof>

lemma *le-succ-iff*: $i \text{ le } succ(j) \iff i \text{ le } j \mid i = succ(j) \ \& \ Ord(i)$
<proof>

lemma *all-lt-imp-le*: $[[Ord(i); Ord(j); \forall x. x < j \implies x < i]] \implies j \text{ le } i$
<proof>

13.7.1 Transitivity Laws

lemma *lt-trans1*: $[[i \text{ le } j; j < k]] \implies i < k$
<proof>

lemma *lt-trans2*: $[[i < j; j \text{ le } k]] \implies i < k$
<proof>

lemma *le-trans*: $[[i \text{ le } j; j \text{ le } k]] \implies i \text{ le } k$
<proof>

lemma *succ-leI*: $i < j \implies succ(i) \text{ le } j$
<proof>

lemma *succ-leE*: $succ(i) \text{ le } j \implies i < j$
<proof>

lemma *succ-le-iff* [*iff*]: $succ(i) \text{ le } j \iff i < j$
<proof>

lemma *succ-le-imp-le*: $succ(i) \text{ le } succ(j) \implies i \text{ le } j$
<proof>

lemma *lt-subset-trans*: $[[i <= j; j < k; Ord(i)]] \implies i < k$
<proof>

lemma *lt-imp-0-lt*: $j < i \implies 0 < i$

<proof>

lemma *succ-lt-iff*: $\text{succ}(i) < j \iff i < j \ \& \ \text{succ}(i) \neq j$

<proof>

lemma *Ord-succ-mem-iff*: $\text{Ord}(j) \implies \text{succ}(i) \in \text{succ}(j) \iff i \in j$

<proof>

13.7.2 Union and Intersection

lemma *Un-upper1-le*: $[\text{Ord}(i); \text{Ord}(j)] \implies i \text{ le } i \text{ Un } j$

<proof>

lemma *Un-upper2-le*: $[\text{Ord}(i); \text{Ord}(j)] \implies j \text{ le } i \text{ Un } j$

<proof>

lemma *Un-least-lt*: $[i < k; j < k] \implies i \text{ Un } j < k$

<proof>

lemma *Un-least-lt-iff*: $[\text{Ord}(i); \text{Ord}(j)] \implies i \text{ Un } j < k \iff i < k \ \& \ j < k$

<proof>

lemma *Un-least-mem-iff*:

$[\text{Ord}(i); \text{Ord}(j); \text{Ord}(k)] \implies i \text{ Un } j : k \iff i : k \ \& \ j : k$

<proof>

lemma *Int-greatest-lt*: $[i < k; j < k] \implies i \text{ Int } j < k$

<proof>

lemma *Ord-Un-if*:

$[\text{Ord}(i); \text{Ord}(j)] \implies i \cup j = (\text{if } j < i \text{ then } i \text{ else } j)$

<proof>

lemma *succ-Un-distrib*:

$[\text{Ord}(i); \text{Ord}(j)] \implies \text{succ}(i \cup j) = \text{succ}(i) \cup \text{succ}(j)$

<proof>

lemma *lt-Un-iff*:

$[\text{Ord}(i); \text{Ord}(j)] \implies k < i \cup j \iff k < i \mid k < j$

<proof>

lemma *le-Un-iff*:

$[\text{Ord}(i); \text{Ord}(j)] \implies k \leq i \cup j \iff k \leq i \mid k \leq j$

<proof>

lemma *Un-upper1-lt*: $[[k < i; \text{Ord}(j)]] \implies k < i \text{ Un } j$
 $\langle \text{proof} \rangle$

lemma *Un-upper2-lt*: $[[k < j; \text{Ord}(i)]] \implies k < i \text{ Un } j$
 $\langle \text{proof} \rangle$

lemma *Ord-Union-succ-eq*: $\text{Ord}(i) \implies \bigcup(\text{succ}(i)) = i$
 $\langle \text{proof} \rangle$

13.8 Results about Limits

lemma *Ord-Union* [*intro,simp,TC*]: $[[!!i. i:A \implies \text{Ord}(i)]] \implies \text{Ord}(\text{Union}(A))$
 $\langle \text{proof} \rangle$

lemma *Ord-UN* [*intro,simp,TC*]:
 $[[!!x. x:A \implies \text{Ord}(B(x))]] \implies \text{Ord}(\bigcup_{x \in A} B(x))$
 $\langle \text{proof} \rangle$

lemma *Ord-Inter* [*intro,simp,TC*]:
 $[[!!i. i:A \implies \text{Ord}(i)]] \implies \text{Ord}(\text{Inter}(A))$
 $\langle \text{proof} \rangle$

lemma *Ord-INT* [*intro,simp,TC*]:
 $[[!!x. x:A \implies \text{Ord}(B(x))]] \implies \text{Ord}(\bigcap_{x \in A} B(x))$
 $\langle \text{proof} \rangle$

lemma *UN-least-le*:
 $[[\text{Ord}(i); !!x. x:A \implies b(x) \text{ le } i]] \implies (\bigcup_{x \in A} b(x)) \text{ le } i$
 $\langle \text{proof} \rangle$

lemma *UN-succ-least-lt*:
 $[[j < i; !!x. x:A \implies b(x) < j]] \implies (\bigcup_{x \in A} \text{succ}(b(x))) < i$
 $\langle \text{proof} \rangle$

lemma *UN-upper-lt*:
 $[[a \in A; i < b(a); \text{Ord}(\bigcup_{x \in A} b(x))]] \implies i < (\bigcup_{x \in A} b(x))$
 $\langle \text{proof} \rangle$

lemma *UN-upper-le*:
 $[[a: A; i \text{ le } b(a); \text{Ord}(\bigcup_{x \in A} b(x))]] \implies i \text{ le } (\bigcup_{x \in A} b(x))$
 $\langle \text{proof} \rangle$

lemma *lt-Union-iff*: $\forall i \in A. \text{Ord}(i) \implies (j < \bigcup(A)) \iff (\exists i \in A. j < i)$
 $\langle \text{proof} \rangle$

lemma *Union-upper-le*:

$\llbracket j: J; i \leq j; \text{Ord}(\bigcup(J)) \rrbracket \implies i \leq \bigcup J$
 ⟨proof⟩

lemma *le-implies-UN-le-UN*:

$\llbracket \forall x. x:A \implies c(x) \text{ le } d(x) \rrbracket \implies (\bigcup_{x \in A} c(x)) \text{ le } (\bigcup_{x \in A} d(x))$
 ⟨proof⟩

lemma *Ord-equality*: $\text{Ord}(i) \implies (\bigcup_{y \in i} \text{succ}(y)) = i$
 ⟨proof⟩

lemma *Ord-Union-subset*: $\text{Ord}(i) \implies \text{Union}(i) \leq i$
 ⟨proof⟩

13.9 Limit Ordinals – General Properties

lemma *Limit-Union-eq*: $\text{Limit}(i) \implies \text{Union}(i) = i$
 ⟨proof⟩

lemma *Limit-is-Ord*: $\text{Limit}(i) \implies \text{Ord}(i)$
 ⟨proof⟩

lemma *Limit-has-0*: $\text{Limit}(i) \implies 0 < i$
 ⟨proof⟩

lemma *Limit-nonzero*: $\text{Limit}(i) \implies i \sim 0$
 ⟨proof⟩

lemma *Limit-has-succ*: $\llbracket \text{Limit}(i); j < i \rrbracket \implies \text{succ}(j) < i$
 ⟨proof⟩

lemma *Limit-succ-lt-iff* [*simp*]: $\text{Limit}(i) \implies \text{succ}(j) < i \iff (j < i)$
 ⟨proof⟩

lemma *zero-not-Limit* [*iff*]: $\sim \text{Limit}(0)$
 ⟨proof⟩

lemma *Limit-has-1*: $\text{Limit}(i) \implies 1 < i$
 ⟨proof⟩

lemma *increasing-LimitI*: $\llbracket 0 < l; \forall x \in l. \exists y \in l. x < y \rrbracket \implies \text{Limit}(l)$
 ⟨proof⟩

lemma *non-succ-LimitI*:

$\llbracket 0 < i; \text{ALL } y. \text{succ}(y) \sim i \rrbracket \implies \text{Limit}(i)$
 ⟨proof⟩

lemma *succ-LimitE* [*elim!*]: $\text{Limit}(\text{succ}(i)) \implies P$
 ⟨proof⟩

lemma *not-succ-Limit* [*simp*]: $\sim \text{Limit}(\text{succ}(i))$
 ⟨*proof*⟩

lemma *Limit-le-succD*: $[[\text{Limit}(i); i \text{ le } \text{succ}(j)]] \implies i \text{ le } j$
 ⟨*proof*⟩

13.9.1 Traditional 3-Way Case Analysis on Ordinals

lemma *Ord-cases-disj*: $\text{Ord}(i) \implies i=0 \mid (\exists x. \text{Ord}(x) \ \& \ i=\text{succ}(x)) \mid \text{Limit}(i)$
 ⟨*proof*⟩

lemma *Ord-cases*:

$[[\text{Ord}(i);$
 $i=0 \implies P;$
 $!!j. [[\text{Ord}(j); i=\text{succ}(j)]] \implies P;$
 $\text{Limit}(i) \implies P$
 $]] \implies P$
 ⟨*proof*⟩

lemma *trans-induct3* [*case-names 0 succ limit, consumes 1*]:

$[[\text{Ord}(i);$
 $P(0);$
 $!!x. [[\text{Ord}(x); P(x)]] \implies P(\text{succ}(x));$
 $!!x. [[\text{Limit}(x); \text{ALL } y:x. P(y)]] \implies P(x)$
 $]] \implies P(i)$
 ⟨*proof*⟩

lemmas *trans-induct3-rule* = *trans-induct3* [*rule-format, case-names 0 succ limit, consumes 1*]

A set of ordinals is either empty, contains its own union, or its union is a limit ordinal.

lemma *Ord-set-cases*:

$\forall i \in I. \text{Ord}(i) \implies I=0 \vee \bigcup(I) \in I \vee (\bigcup(I) \notin I \wedge \text{Limit}(\bigcup(I)))$
 ⟨*proof*⟩

If the union of a set of ordinals is a successor, then it is an element of that set.

lemma *Ord-Union-eq-succD*: $[[\forall x \in X. \text{Ord}(x); \bigcup X = \text{succ}(j)]] \implies \text{succ}(j) \in X$
 ⟨*proof*⟩

lemma *Limit-Union* [*rule-format*]: $[[I \neq 0; \forall i \in I. \text{Limit}(i)]] \implies \text{Limit}(\bigcup I)$
 ⟨*proof*⟩

end

14 OrdQuant: Special quantifiers

theory *OrdQuant* imports *Ordinal* begin

14.1 Quantifiers and union operator for ordinals

definition

oall :: [*i*, *i* => *o*] => *o* **where**
oall(*A*, *P*) == *ALL* *x*. *x*<*A* --> *P*(*x*)

definition

oex :: [*i*, *i* => *o*] => *o* **where**
oex(*A*, *P*) == *EX* *x*. *x*<*A* & *P*(*x*)

definition

OUnion :: [*i*, *i* => *i*] => *i* **where**
OUnion(*i*,*B*) == {*z*: $\bigcup x \in i. B(x). \text{Ord}(i)$ }

syntax

-*oall* :: [*idt*, *i*, *o*] => *o* ((*3ALL* -<-./ -) 10)
-*oex* :: [*idt*, *i*, *o*] => *o* ((*3EX* -<-./ -) 10)
-*OUNION* :: [*idt*, *i*, *i*] => *i* ((*3UN* -<-./ -) 10)

translations

ALL *x*<*a*. *P* == *CONST* *oall*(*a*, %*x*. *P*)
EX *x*<*a*. *P* == *CONST* *oex*(*a*, %*x*. *P*)
UN *x*<*a*. *B* == *CONST* *OUnion*(*a*, %*x*. *B*)

syntax (*xsymbols*)

-*oall* :: [*idt*, *i*, *o*] => *o* ((*3V* -<-./ -) 10)
-*oex* :: [*idt*, *i*, *o*] => *o* ((*3E* -<-./ -) 10)
-*OUNION* :: [*idt*, *i*, *i*] => *i* ((*3U* -<-./ -) 10)

syntax (*HTML output*)

-*oall* :: [*idt*, *i*, *o*] => *o* ((*3V* -<-./ -) 10)
-*oex* :: [*idt*, *i*, *o*] => *o* ((*3E* -<-./ -) 10)
-*OUNION* :: [*idt*, *i*, *i*] => *i* ((*3U* -<-./ -) 10)

14.1.1 simplification of the new quantifiers

lemma [*simp*]: (*ALL* *x*<*0*. *P*(*x*))
<*proof*>

lemma [*simp*]: \sim (*EX* *x*<*0*. *P*(*x*))
<*proof*>

lemma [*simp*]: (*ALL* *x*<*succ*(*i*). *P*(*x*)) <-> (*Ord*(*i*) --> *P*(*i*) & (*ALL* *x*<*i*.
P(*x*)))
<*proof*>

lemma *[simp]*: $(\exists x < \text{succ}(i). P(x)) \leftrightarrow (\text{Ord}(i) \ \& \ (P(i) \mid (\exists x < i. P(x))))$
 $\langle \text{proof} \rangle$

14.1.2 Union over ordinals

lemma *Ord-OUN [intro,simp]*:
 $[\![\ \! \! x. x < A \implies \text{Ord}(B(x)) \]\!] \implies \text{Ord}(\bigcup x < A. B(x))$
 $\langle \text{proof} \rangle$

lemma *OUN-upper-lt*:
 $[\![\ a < A; \ i < b(a); \ \text{Ord}(\bigcup x < A. b(x)) \]\!] \implies i < (\bigcup x < A. b(x))$
 $\langle \text{proof} \rangle$

lemma *OUN-upper-le*:
 $[\![\ a < A; \ i \leq b(a); \ \text{Ord}(\bigcup x < A. b(x)) \]\!] \implies i \leq (\bigcup x < A. b(x))$
 $\langle \text{proof} \rangle$

lemma *Limit-OUN-eq*: $\text{Limit}(i) \implies (\bigcup x < i. x) = i$
 $\langle \text{proof} \rangle$

lemma *OUN-least*:
 $(\! \! x. x < A \implies B(x) \subseteq C) \implies (\bigcup x < A. B(x)) \subseteq C$
 $\langle \text{proof} \rangle$

lemma *OUN-least-le*:
 $[\![\ \text{Ord}(i); \ \! \! x. x < A \implies b(x) \leq i \]\!] \implies (\bigcup x < A. b(x)) \leq i$
 $\langle \text{proof} \rangle$

lemma *le-implies-OUN-le-OUN*:
 $[\![\ \! \! x. x < A \implies c(x) \leq d(x) \]\!] \implies (\bigcup x < A. c(x)) \leq (\bigcup x < A. d(x))$
 $\langle \text{proof} \rangle$

lemma *OUN-UN-eq*:
 $(\! \! x. x : A \implies \text{Ord}(B(x)))$
 $\implies (\bigcup z < (\bigcup x \in A. B(x)). C(z)) = (\bigcup x \in A. \bigcup z < B(x). C(z))$
 $\langle \text{proof} \rangle$

lemma *OUN-Union-eq*:
 $(\! \! x. x : X \implies \text{Ord}(x))$
 $\implies (\bigcup z < \text{Union}(X). C(z)) = (\bigcup x \in X. \bigcup z < x. C(z))$
 $\langle \text{proof} \rangle$

lemma *atomize-oall [symmetric, rulify]*:
 $(\! \! x. x < A \implies P(x)) \implies \text{Trueprop}(\text{ALL } x < A. P(x))$
 $\langle \text{proof} \rangle$

14.1.3 universal quantifier for ordinals

lemma *oall* [*intro!*]:

$\llbracket \forall x. x < A \implies P(x) \rrbracket \implies \text{ALL } x < A. P(x)$
<proof>

lemma *ospec*: $\llbracket \text{ALL } x < A. P(x); x < A \rrbracket \implies P(x)$

<proof>

lemma *oallE*:

$\llbracket \text{ALL } x < A. P(x); P(x) \implies Q; \sim x < A \implies Q \rrbracket \implies Q$
<proof>

lemma *rev-oallE* [*elim*]:

$\llbracket \text{ALL } x < A. P(x); \sim x < A \implies Q; P(x) \implies Q \rrbracket \implies Q$
<proof>

lemma *oall-simp* [*simp*]: $(\text{ALL } x < a. \text{True}) <-> \text{True}$

<proof>

lemma *oall-cong* [*cong*]:

$\llbracket a = a'; \forall x. x < a' \implies P(x) <-> P'(x) \rrbracket$
 $\implies \text{oall}(a, \%x. P(x)) <-> \text{oall}(a', \%x. P'(x))$
<proof>

14.1.4 existential quantifier for ordinals

lemma *oexI* [*intro*]:

$\llbracket P(x); x < A \rrbracket \implies \text{EX } x < A. P(x)$
<proof>

lemma *oexCI*:

$\llbracket \text{ALL } x < A. \sim P(x) \implies P(a); a < A \rrbracket \implies \text{EX } x < A. P(x)$
<proof>

lemma *oexE* [*elim!*]:

$\llbracket \text{EX } x < A. P(x); \forall x. \llbracket x < A; P(x) \rrbracket \implies Q \rrbracket \implies Q$
<proof>

lemma *oex-cong* [*cong*]:

$\llbracket a = a'; \forall x. x < a' \implies P(x) <-> P'(x) \rrbracket$
 $\implies \text{oex}(a, \%x. P(x)) <-> \text{oex}(a', \%x. P'(x))$
<proof>

14.1.5 Rules for Ordinal-Indexed Unions

lemma *OUN-I* [*intro*]: $[[a < i; b : B(a)]] \implies b : (\bigcup_{z < i}. B(z))$
<proof>

lemma *OUN-E* [*elim!*]:
 $[[b : (\bigcup_{z < i}. B(z)); !!a. [[b : B(a); a < i]] \implies R]] \implies R$
<proof>

lemma *OUN-iff*: $b : (\bigcup_{x < i}. B(x)) \iff (EX x < i. b : B(x))$
<proof>

lemma *OUN-cong* [*cong*]:
 $[[i = j; !!x. x < j \implies C(x) = D(x)]] \implies (\bigcup_{x < i}. C(x)) = (\bigcup_{x < j}. D(x))$
<proof>

lemma *lt-induct*:
 $[[i < k; !!x. [[x < k; ALL y < x. P(y)]] \implies P(x)]] \implies P(i)$
<proof>

14.2 Quantification over a class

definition

rall $:: [i=>o, i=>o] => o$ **where**
 $rall(M, P) == ALL x. M(x) \implies P(x)$

definition

rex $:: [i=>o, i=>o] => o$ **where**
 $rex(M, P) == EX x. M(x) \& P(x)$

syntax

-rall $:: [pttrn, i=>o, o] => o$ $((\exists ALL \text{-}[\cdot]/ \text{-}) 10)$
-rex $:: [pttrn, i=>o, o] => o$ $((\exists EX \text{-}[\cdot]/ \text{-}) 10)$

syntax (*xsymbols*)

-rall $:: [pttrn, i=>o, o] => o$ $((\exists \forall \text{-}[\cdot]/ \text{-}) 10)$
-rex $:: [pttrn, i=>o, o] => o$ $((\exists \exists \text{-}[\cdot]/ \text{-}) 10)$

syntax (*HTML output*)

-rall $:: [pttrn, i=>o, o] => o$ $((\exists \forall \text{-}[\cdot]/ \text{-}) 10)$
-rex $:: [pttrn, i=>o, o] => o$ $((\exists \exists \text{-}[\cdot]/ \text{-}) 10)$

translations

$ALL x[M]. P == CONST rall(M, \%x. P)$
 $EX x[M]. P == CONST rex(M, \%x. P)$

14.2.1 Relativized universal quantifier

lemma *rallI* [*intro!*]: $[[!!x. M(x) \implies P(x)]] \implies ALL x[M]. P(x)$
<proof>

lemma *rspec*: $\llbracket \text{ALL } x[M]. P(x); M(x) \rrbracket \implies P(x)$
 $\langle \text{proof} \rangle$

lemma *rev-rallE* [*elim*]:
 $\llbracket \text{ALL } x[M]. P(x); \sim M(x) \implies Q; P(x) \implies Q \rrbracket \implies Q$
 $\langle \text{proof} \rangle$

lemma *rallE*: $\llbracket \text{ALL } x[M]. P(x); P(x) \implies Q; \sim M(x) \implies Q \rrbracket \implies Q$
 $\langle \text{proof} \rangle$

lemma *rall-triv* [*simp*]: $(\text{ALL } x[M]. P) \longleftrightarrow ((\text{EX } x. M(x)) \dashrightarrow P)$
 $\langle \text{proof} \rangle$

lemma *rall-cong* [*cong*]:
 $(!!x. M(x) \implies P(x) \longleftrightarrow P'(x)) \implies (\text{ALL } x[M]. P(x)) \longleftrightarrow (\text{ALL } x[M]. P'(x))$
 $\langle \text{proof} \rangle$

14.2.2 Relativized existential quantifier

lemma *rexI* [*intro*]: $\llbracket P(x); M(x) \rrbracket \implies \text{EX } x[M]. P(x)$
 $\langle \text{proof} \rangle$

lemma *rev-rexI*: $\llbracket M(x); P(x) \rrbracket \implies \text{EX } x[M]. P(x)$
 $\langle \text{proof} \rangle$

lemma *rexCI*: $\llbracket \text{ALL } x[M]. \sim P(x) \implies P(a); M(a) \rrbracket \implies \text{EX } x[M]. P(x)$
 $\langle \text{proof} \rangle$

lemma *rexE* [*elim!*]: $\llbracket \text{EX } x[M]. P(x); !!x. \llbracket M(x); P(x) \rrbracket \implies Q \rrbracket \implies Q$
 $\langle \text{proof} \rangle$

lemma *rex-triv* [*simp*]: $(\text{EX } x[M]. P) \longleftrightarrow ((\text{EX } x. M(x)) \& P)$
 $\langle \text{proof} \rangle$

lemma *rex-cong* [*cong*]:
 $(!!x. M(x) \implies P(x) \longleftrightarrow P'(x)) \implies (\text{EX } x[M]. P(x)) \longleftrightarrow (\text{EX } x[M]. P'(x))$
 $\langle \text{proof} \rangle$

lemma *rall-is-ball* [*simp*]: $(\forall x[\%z. z \in A]. P(x)) \longleftrightarrow (\forall x \in A. P(x))$
 $\langle \text{proof} \rangle$

lemma *rex-is-bex* [*simp*]: $(\exists x[\%z. z \in A]. P(x)) \leftrightarrow (\exists x \in A. P(x))$
 ⟨*proof*⟩

lemma *atomize-rall*: $(!\!x. M(x) \implies P(x)) \implies \text{Trueprop } (ALL\ x[M]. P(x))$
 ⟨*proof*⟩

declare *atomize-rall* [*symmetric, rulify*]

lemma *rall-simps1*:

$(ALL\ x[M]. P(x) \ \&\ Q) \leftrightarrow (ALL\ x[M]. P(x)) \ \&\ ((ALL\ x[M]. False) \ | \ Q)$
 $(ALL\ x[M]. P(x) \ | \ Q) \leftrightarrow ((ALL\ x[M]. P(x)) \ | \ Q)$
 $(ALL\ x[M]. P(x) \ \dashrightarrow \ Q) \leftrightarrow ((EX\ x[M]. P(x)) \ \dashrightarrow \ Q)$
 $(\sim(ALL\ x[M]. P(x))) \leftrightarrow (EX\ x[M]. \sim P(x))$

⟨*proof*⟩

lemma *rall-simps2*:

$(ALL\ x[M]. P \ \&\ Q(x)) \leftrightarrow ((ALL\ x[M]. False) \ | \ P) \ \&\ (ALL\ x[M]. Q(x))$
 $(ALL\ x[M]. P \ | \ Q(x)) \leftrightarrow (P \ | \ (ALL\ x[M]. Q(x)))$
 $(ALL\ x[M]. P \ \dashrightarrow \ Q(x)) \leftrightarrow (P \ \dashrightarrow \ (ALL\ x[M]. Q(x)))$

⟨*proof*⟩

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

lemma *rall-conj-distrib*:

$(ALL\ x[M]. P(x) \ \&\ Q(x)) \leftrightarrow ((ALL\ x[M]. P(x)) \ \&\ (ALL\ x[M]. Q(x)))$

⟨*proof*⟩

lemma *rex-simps1*:

$(EX\ x[M]. P(x) \ \&\ Q) \leftrightarrow ((EX\ x[M]. P(x)) \ \&\ Q)$
 $(EX\ x[M]. P(x) \ | \ Q) \leftrightarrow (EX\ x[M]. P(x)) \ | \ ((EX\ x[M]. True) \ \&\ Q)$
 $(EX\ x[M]. P(x) \ \dashrightarrow \ Q) \leftrightarrow ((ALL\ x[M]. P(x)) \ \dashrightarrow \ ((EX\ x[M]. True) \ \&\ Q))$
 $(\sim(EX\ x[M]. P(x))) \leftrightarrow (ALL\ x[M]. \sim P(x))$

⟨*proof*⟩

lemma *rex-simps2*:

$(EX\ x[M]. P \ \&\ Q(x)) \leftrightarrow (P \ \&\ (EX\ x[M]. Q(x)))$
 $(EX\ x[M]. P \ | \ Q(x)) \leftrightarrow ((EX\ x[M]. True) \ \&\ P) \ | \ (EX\ x[M]. Q(x))$
 $(EX\ x[M]. P \ \dashrightarrow \ Q(x)) \leftrightarrow (((ALL\ x[M]. False) \ | \ P) \ \dashrightarrow \ (EX\ x[M]. Q(x)))$

⟨*proof*⟩

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

lemma *rex-disj-distrib*:

$(EX\ x[M]. P(x) \ | \ Q(x)) \leftrightarrow ((EX\ x[M]. P(x)) \ | \ (EX\ x[M]. Q(x)))$

⟨*proof*⟩

14.2.3 One-point rule for bounded quantifiers

lemma *rex-triv-one-point1* [simp]: $(EX\ x[M].\ x=a) \leftrightarrow (M(a))$
<proof>

lemma *rex-triv-one-point2* [simp]: $(EX\ x[M].\ a=x) \leftrightarrow (M(a))$
<proof>

lemma *rex-one-point1* [simp]: $(EX\ x[M].\ x=a \ \&\ P(x)) \leftrightarrow (M(a) \ \&\ P(a))$
<proof>

lemma *rex-one-point2* [simp]: $(EX\ x[M].\ a=x \ \&\ P(x)) \leftrightarrow (M(a) \ \&\ P(a))$
<proof>

lemma *rall-one-point1* [simp]: $(ALL\ x[M].\ x=a \ \dashrightarrow P(x)) \leftrightarrow (M(a) \ \dashrightarrow P(a))$
<proof>

lemma *rall-one-point2* [simp]: $(ALL\ x[M].\ a=x \ \dashrightarrow P(x)) \leftrightarrow (M(a) \ \dashrightarrow P(a))$
<proof>

14.2.4 Sets as Classes

definition

setclass :: $[i,i] \Rightarrow o$ (*##- [40] 40*) **where**
setclass(A) == $\%x.\ x : A$

lemma *setclass-iff* [simp]: $setclass(A,x) \leftrightarrow x : A$
<proof>

lemma *rall-setclass-is-ball* [simp]: $(\forall\ x[##A].\ P(x)) \leftrightarrow (\forall\ x \in A.\ P(x))$
<proof>

lemma *rex-setclass-is-bex* [simp]: $(\exists\ x[##A].\ P(x)) \leftrightarrow (\exists\ x \in A.\ P(x))$
<proof>

<ML>

Setting up the one-point-rule simproc

<ML>

end

15 Nat-ZF: The Natural numbers As a Least Fixed Point

theory *Nat-ZF* **imports** *OrdQuant Bool* **begin**

definition

nat :: *i* **where**
nat == *lfp*(*Inf*, %*X*. {0} *Un* {*succ*(*i*). *i*:*X*})

definition

*quasi**nat* :: *i* => *o* **where**
*quasi**nat*(*n*) == *n*=0 | (\exists *m*. *n* = *succ*(*m*))

definition

nat-*case* :: [*i*, *i*=>*i*, *i*]=>*i* **where**
nat-*case*(*a*,*b*,*k*) == *THE* *y*. *k*=0 & *y*=*a* | (*EX* *x*. *k*=*succ*(*x*) & *y*=*b*(*x*))

definition

nat-*rec* :: [*i*, *i*, [*i*,*i*]=>*i*]=>*i* **where**
nat-*rec*(*k*,*a*,*b*) ==
wfrec(*Memrel*(*nat*), *k*, %*n* *f*. *nat*-*case*(*a*, %*m*. *b*(*m*, *f**m*), *n*))

definition

Le :: *i* **where**
Le == {<*x*,*y*>:*nat***nat*. *x le y*}

definition

Lt :: *i* **where**
Lt == {<*x*, *y*>:*nat***nat*. *x < y*}

definition

Ge :: *i* **where**
Ge == {<*x*,*y*>:*nat***nat*. *y le x*}

definition

Gt :: *i* **where**
Gt == {<*x*,*y*>:*nat***nat*. *y < x*}

definition

greater-*than* :: *i*=>*i* **where**
greater-*than*(*n*) == {*i*:*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*, %*X*. {0} *Un* {*succ*(*i*). *i*:*X*})
 <*proof*>

lemmas *nat-unfold* = *nat-bnd-mono* [*THEN nat-def* [*THEN def-lfp-unfold*], *standard*]

lemma *nat-0I* [*iff,TC*]: $0 : \text{nat}$
<proof>

lemma *nat-succI* [*intro!,TC*]: $n : \text{nat} \implies \text{succ}(n) : \text{nat}$
<proof>

lemma *nat-1I* [*iff,TC*]: $1 : \text{nat}$
<proof>

lemma *nat-2I* [*iff,TC*]: $2 : \text{nat}$
<proof>

lemma *bool-subset-nat*: $\text{bool} \leq \text{nat}$
<proof>

lemmas *bool-into-nat* = *bool-subset-nat* [*THEN subsetD, standard*]

15.1 Injectivity Properties and Induction

lemma *nat-induct* [*case-names 0 succ, induct set: nat*]:
 $\llbracket n : \text{nat}; P(0); \forall x. \llbracket x : \text{nat}; P(x) \rrbracket \implies P(\text{succ}(x)) \rrbracket \implies P(n)$
<proof>

lemma *natE*:
 $\llbracket n : \text{nat}; n=0 \implies P; \forall x. \llbracket x : \text{nat}; n=\text{succ}(x) \rrbracket \implies P \rrbracket \implies P$
<proof>

lemma *nat-into-Ord* [*simp*]: $n : \text{nat} \implies \text{Ord}(n)$
<proof>

lemmas *nat-0-le* = *nat-into-Ord* [*THEN Ord-0-le, standard*]

lemmas *nat-le-refl* = *nat-into-Ord* [*THEN le-refl, standard*]

lemma *Ord-nat* [*iff*]: $\text{Ord}(\text{nat})$
<proof>

lemma *Limit-nat* [*iff*]: $\text{Limit}(\text{nat})$
<proof>

lemma *naturals-not-limit*: $a \in \text{nat} \implies \sim \text{Limit}(a)$

<proof>

lemma *succ-natD*: $\text{succ}(i): \text{nat} \implies i: \text{nat}$
<proof>

lemma *nat-succ-iff* [*iff*]: $\text{succ}(n): \text{nat} \longleftrightarrow n: \text{nat}$
<proof>

lemma *nat-le-Limit*: $\text{Limit}(i) \implies \text{nat le } i$
<proof>

lemmas *succ-in-naturalD* = *Ord-trans* [*OF succI1 - nat-into-Ord*]

lemma *lt-nat-in-nat*: $[[m < n; n: \text{nat}]] \implies m: \text{nat}$
<proof>

lemma *le-in-nat*: $[[m \text{ le } n; n: \text{nat}]] \implies m: \text{nat}$
<proof>

15.2 Variations on Mathematical Induction

lemmas *complete-induct* = *Ord-induct* [*OF - Ord-nat, case-names less, consumes 1*]

lemmas *complete-induct-rule* =
complete-induct [*rule-format, case-names less, consumes 1*]

lemma *nat-induct-from-lemma* [*rule-format*]:
 $[[n: \text{nat}; m: \text{nat};$
 $!!x. [[x: \text{nat}; m \text{ le } x; P(x)]] \implies P(\text{succ}(x))]]$
 $\implies m \text{ le } n \dashrightarrow P(m) \dashrightarrow P(n)$
<proof>

lemma *nat-induct-from*:
 $[[m \text{ le } n; m: \text{nat}; n: \text{nat};$
 $P(m);$
 $!!x. [[x: \text{nat}; m \text{ le } x; P(x)]] \implies P(\text{succ}(x))]]$
 $\implies P(n)$
<proof>

lemma *diff-induct* [*case-names 0 0-succ succ-succ, consumes 2*]:
 $[[m: \text{nat}; n: \text{nat};$
 $!!x. x: \text{nat} \implies P(x, 0);$
 $!!y. y: \text{nat} \implies P(0, \text{succ}(y));$
 $!!x y. [[x: \text{nat}; y: \text{nat}; P(x, y)]] \implies P(\text{succ}(x), \text{succ}(y))]]$

$\implies P(m,n)$
 $\langle \text{proof} \rangle$

lemma *succ-lt-induct-lemma* [rule-format]:
 $m: \text{nat} \implies P(m, \text{succ}(m)) \dashrightarrow (\text{ALL } x: \text{nat}. P(m,x) \dashrightarrow P(m, \text{succ}(x)))$
 \dashrightarrow
 $(\text{ALL } n: \text{nat}. m < n \dashrightarrow P(m,n))$
 $\langle \text{proof} \rangle$

lemma *succ-lt-induct*:
 $[[m < n; n: \text{nat};$
 $P(m, \text{succ}(m));$
 $!!x. [[x: \text{nat}; P(m,x)]] \implies P(m, \text{succ}(x))]]$
 $\implies P(m,n)$
 $\langle \text{proof} \rangle$

15.3 quasinat: to allow a case-split rule for *nat-case*

True if the argument is zero or any successor

lemma [iff]: *quasinat(0)*
 $\langle \text{proof} \rangle$

lemma [iff]: *quasinat(succ(x))*
 $\langle \text{proof} \rangle$

lemma *nat-imp-quasinat*: $n \in \text{nat} \implies \text{quasinat}(n)$
 $\langle \text{proof} \rangle$

lemma *non-nat-case*: $\sim \text{quasinat}(x) \implies \text{nat-case}(a,b,x) = 0$
 $\langle \text{proof} \rangle$

lemma *nat-cases-disj*: $k=0 \mid (\exists y. k = \text{succ}(y)) \mid \sim \text{quasinat}(k)$
 $\langle \text{proof} \rangle$

lemma *nat-cases*:
 $[[k=0 \implies P; !!y. k = \text{succ}(y) \implies P; \sim \text{quasinat}(k) \implies P]] \implies P$
 $\langle \text{proof} \rangle$

lemma *nat-case-0* [simp]: $\text{nat-case}(a,b,0) = a$
 $\langle \text{proof} \rangle$

lemma *nat-case-succ* [simp]: $\text{nat-case}(a,b, \text{succ}(n)) = b(n)$
 $\langle \text{proof} \rangle$

lemma *nat-case-type* [TC]:

$$\llbracket n: \text{nat}; a: C(0); \forall m. m: \text{nat} \implies b(m): C(\text{succ}(m)) \rrbracket$$

$$\implies \text{nat-case}(a,b,n) : C(n)$$
 <proof>

lemma *split-nat-case*:

$$P(\text{nat-case}(a,b,k)) \iff$$

$$((k=0 \implies P(a)) \ \& \ (\forall x. k=\text{succ}(x) \implies P(b(x))) \ \& \ (\sim \text{quasinat}(k) \implies P(0)))$$
 <proof>

15.4 Recursion on the Natural Numbers

lemma *nat-rec-0*: $\text{nat-rec}(0,a,b) = a$
 <proof>

lemma *nat-rec-succ*: $m: \text{nat} \implies \text{nat-rec}(\text{succ}(m),a,b) = b(m, \text{nat-rec}(m,a,b))$
 <proof>

lemma *Un-nat-type* [TC]: $\llbracket i: \text{nat}; j: \text{nat} \rrbracket \implies i \text{ Un } j: \text{nat}$
 <proof>

lemma *Int-nat-type* [TC]: $\llbracket i: \text{nat}; j: \text{nat} \rrbracket \implies i \text{ Int } j: \text{nat}$
 <proof>

lemma *nat-nonempty* [simp]: $\text{nat} \sim= 0$
 <proof>

A natural number is the set of its predecessors

lemma *nat-eq-Collect-lt*: $i \in \text{nat} \implies \{j \in \text{nat}. j < i\} = i$
 <proof>

lemma *Le-iff* [iff]: $\langle x,y \rangle : \text{Le} \iff x \text{ le } y \ \& \ x : \text{nat} \ \& \ y : \text{nat}$
 <proof>

end

16 Inductive-ZF: Inductive and Coinductive Definitions

theory *Inductive-ZF*
imports *Fixedpt QPair Nat-ZF*
uses
 (*ind-syntax.ML*)

```

(Tools/cartprod.ML)
(Tools/ind-cases.ML)
(Tools/inductive-package.ML)
(Tools/induct-tacs.ML)
(Tools/primrec-package.ML)
begin

lemma def-swap-iff:  $a == b ==> a = c <-> c = b$ 
  <proof>

lemma def-trans:  $f == g ==> g(a) = b ==> f(a) = b$ 
  <proof>

lemma refl-thin:  $!!P. a = a ==> P ==> P$  <proof>

<ML>

end

```

17 Epsilon: Epsilon Induction and Recursion

```

theory Epsilon imports Nat-ZF begin

```

definition

```

eclose  ::  $i=>i$  where
  eclose( $A$ ) ==  $\bigcup_{n \in \text{nat.}} \text{nat-rec}(n, A, \%m r. \text{Union}(r))$ 

```

definition

```

transrec ::  $[i, [i,i]=>i] =>i$  where
  transrec( $a, H$ ) == wfrec(Memrel(eclose( $\{a\}$ )),  $a, H$ )

```

definition

```

rank    ::  $i=>i$  where
  rank( $a$ ) == transrec( $a, \%x f. \bigcup_{y \in x. \text{succ}(f'y)}$ )

```

definition

```

transrec2 ::  $[i, i, [i,i]=>i] =>i$  where
  transrec2( $k, a, b$ ) ==
    transrec( $k,$ 
       $\%i r. \text{if}(i=0, a,$ 
         $\text{if}(EX j. i=\text{succ}(j),$ 
           $b(\text{THE } j. i=\text{succ}(j), r'(\text{THE } j. i=\text{succ}(j))),$ 
           $\bigcup_{j < i. r'j}))$ )

```

definition

```

recursor ::  $[i, [i,i]=>i, i]=>i$  where
  recursor( $a, b, k$ ) == transrec( $k, \%n f. \text{nat-case}(a, \%m. b(m, f'm), n)$ )

```

definition

$rec :: [i, i, [i,i]=>i]=>i$ **where**
 $rec(k,a,b) == recursor(a,b,k)$

17.1 Basic Closure Properties

lemma *arg-subset-eclose*: $A \leq eclose(A)$
 $\langle proof \rangle$

lemmas *arg-into-eclose* = *arg-subset-eclose* [*THEN subsetD, standard*]

lemma *Transset-eclose*: $Transset(eclose(A))$
 $\langle proof \rangle$

lemmas *eclose-subset* =
Transset-eclose [*unfolded Transset-def, THEN bspec, standard*]

lemmas *ecloseD* = *eclose-subset* [*THEN subsetD, standard*]

lemmas *arg-in-eclose-sing* = *arg-subset-eclose* [*THEN singleton-subsetD*]
lemmas *arg-into-eclose-sing* = *arg-in-eclose-sing* [*THEN ecloseD, standard*]

lemmas *eclose-induct* =
Transset-induct [*OF - Transset-eclose, induct set: eclose*]

lemma *eps-induct*:
 $[[\forall x. \forall y. x. P(y) ==> P(x)]] ==> P(a)$
 $\langle proof \rangle$

17.2 Leastness of *eclose*

lemma *eclose-least-lemma*:
 $[[Transset(X); A \leq X; n: nat]] ==> nat-rec(n, A, \%m r. Union(r)) \leq X$
 $\langle proof \rangle$

lemma *eclose-least*:
 $[[Transset(X); A \leq X]] ==> eclose(A) \leq X$
 $\langle proof \rangle$

lemma *eclose-induct-down* [*consumes 1*]:
 $[[a: eclose(b);$
 $!!y. [[y: b]] ==> P(y);$
 $!!y z. [[y: eclose(b); P(y); z: y]] ==> P(z)$
 $]] ==> P(a)$

<proof>

lemma *Transset-eclose-eq-arg*: $\text{Transset}(X) \implies \text{eclose}(X) = X$
<proof>

A transitive set either is empty or contains the empty set.

lemma *Transset-0-lemma* [rule-format]: $\text{Transset}(A) \implies x \in A \implies 0 \in A$
<proof>

lemma *Transset-0-disj*: $\text{Transset}(A) \implies A = 0 \mid 0 \in A$
<proof>

17.3 Epsilon Recursion

lemma *mem-eclose-trans*: $[[A: \text{eclose}(B); B: \text{eclose}(C)]] \implies A: \text{eclose}(C)$
<proof>

lemma *mem-eclose-sing-trans*:
 $[[A: \text{eclose}(\{B\}); B: \text{eclose}(\{C\})]] \implies A: \text{eclose}(\{C\})$
<proof>

lemma *under-Memrel*: $[[\text{Transset}(i); j:i]] \implies \text{Memrel}(i) - \{j\} = j$
<proof>

lemma *lt-Memrel*: $j < i \implies \text{Memrel}(i) - \{j\} = j$
<proof>

lemmas *under-Memrel-eclose* = *Transset-eclose* [THEN *under-Memrel*, *standard*]

lemmas *wfrec-ssubst* = *wf-Memrel* [THEN *wfrec*, THEN *ssubst*]

lemma *wfrec-eclose-eq*:
 $[[k: \text{eclose}(\{j\}); j: \text{eclose}(\{i\})]] \implies$
 $\text{wfrec}(\text{Memrel}(\text{eclose}(\{i\})), k, H) = \text{wfrec}(\text{Memrel}(\text{eclose}(\{j\})), k, H)$
<proof>

lemma *wfrec-eclose-eq2*:
 $k: i \implies \text{wfrec}(\text{Memrel}(\text{eclose}(\{i\})), k, H) = \text{wfrec}(\text{Memrel}(\text{eclose}(\{k\})), k, H)$
<proof>

lemma *transrec*: $\text{transrec}(a, H) = H(a, \text{lam } x: a. \text{transrec}(x, H))$
<proof>

lemma *def-transrec*:
 $[[!!x. f(x) == \text{transrec}(x, H)]] \implies f(a) = H(a, \text{lam } x: a. f(x))$
<proof>

lemma *transrec-type*:

$$\llbracket \! \! \! \llbracket x : \text{eclose}(\{a\}); u : Pi(x, B) \rrbracket \implies H(x, u) : B(x) \rrbracket$$
$$\implies \text{transrec}(a, H) : B(a)$$

<proof>

lemma *eclose-sing-Ord*: $\text{Ord}(i) \implies \text{eclose}(\{i\}) \leq \text{succ}(i)$

<proof>

lemma *succ-subset-eclose-sing*: $\text{succ}(i) \leq \text{eclose}(\{i\})$

<proof>

lemma *eclose-sing-Ord-eq*: $\text{Ord}(i) \implies \text{eclose}(\{i\}) = \text{succ}(i)$

<proof>

lemma *Ord-transrec-type*:

assumes *jini*: $j : i$
and *ordi*: $\text{Ord}(i)$
and *minor*: $\llbracket x : i; u : Pi(x, B) \rrbracket \implies H(x, u) : B(x)$
shows $\text{transrec}(j, H) : B(j)$

<proof>

17.4 Rank

lemma *rank*: $\text{rank}(a) = (\bigcup y \in a. \text{succ}(\text{rank}(y)))$

<proof>

lemma *Ord-rank* [*simp*]: $\text{Ord}(\text{rank}(a))$

<proof>

lemma *rank-of-Ord*: $\text{Ord}(i) \implies \text{rank}(i) = i$

<proof>

lemma *rank-lt*: $a < b \implies \text{rank}(a) < \text{rank}(b)$

<proof>

lemma *eclose-rank-lt*: $a : \text{eclose}(b) \implies \text{rank}(a) < \text{rank}(b)$

<proof>

lemma *rank-mono*: $a \leq b \implies \text{rank}(a) \leq \text{rank}(b)$

<proof>

lemma *rank-Pow*: $\text{rank}(\text{Pow}(a)) = \text{succ}(\text{rank}(a))$

<proof>

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

<proof>

lemma *rank-succ* [*simp*]: $\text{rank}(\text{succ}(x)) = \text{succ}(\text{rank}(x))$

<proof>

lemma *rank-Union*: $\text{rank}(\text{Union}(A)) = (\bigcup x \in A. \text{rank}(x))$
<proof>

lemma *rank-eclose*: $\text{rank}(\text{eclose}(a)) = \text{rank}(a)$
<proof>

lemma *rank-pair1*: $\text{rank}(a) < \text{rank}(\langle a, b \rangle)$
<proof>

lemma *rank-pair2*: $\text{rank}(b) < \text{rank}(\langle a, b \rangle)$
<proof>

lemma *the-equality-if*:
 $P(a) \implies (\text{THE } x. P(x)) = (\text{if } (\text{EX } !x. P(x)) \text{ then } a \text{ else } 0)$
<proof>

lemma *rank-apply*: $[[i : \text{domain}(f); \text{function}(f)]] \implies \text{rank}(f'i) < \text{rank}(f)$
<proof>

17.5 Corollaries of Leastness

lemma *mem-eclose-subset*: $A:B \implies \text{eclose}(A) \leq \text{eclose}(B)$
<proof>

lemma *eclose-mono*: $A \leq B \implies \text{eclose}(A) \leq \text{eclose}(B)$
<proof>

lemma *eclose-idem*: $\text{eclose}(\text{eclose}(A)) = \text{eclose}(A)$
<proof>

lemma *transrec2-0* [*simp*]: $\text{transrec2}(0, a, b) = a$
<proof>

lemma *transrec2-succ* [*simp*]: $\text{transrec2}(\text{succ}(i), a, b) = b(i, \text{transrec2}(i, a, b))$
<proof>

lemma *transrec2-Limit*:
 $\text{Limit}(i) \implies \text{transrec2}(i, a, b) = (\bigcup j < i. \text{transrec2}(j, a, b))$
<proof>

lemma *def-transrec2*:

$(!!x. f(x) == \text{transrec2}(x, a, b))$
 $==> f(0) = a \ \&$
 $f(\text{succ}(i)) = b(i, f(i)) \ \&$
 $(\text{Limit}(K) \dashrightarrow f(K) = (\bigcup_{j < K}. f(j)))$
 <proof>

lemmas *recursor-lemma* = *recursor-def* [THEN *def-transrec*, THEN *trans*]

lemma *recursor-0*: $\text{recursor}(a, b, 0) = a$
 <proof>

lemma *recursor-succ*: $\text{recursor}(a, b, \text{succ}(m)) = b(m, \text{recursor}(a, b, m))$
 <proof>

lemma *rec-0* [*simp*]: $\text{rec}(0, a, b) = a$
 <proof>

lemma *rec-succ* [*simp*]: $\text{rec}(\text{succ}(m), a, b) = b(m, \text{rec}(m, a, b))$
 <proof>

lemma *rec-type*:
 $[[n: \text{nat};$
 $a: C(0);$
 $!!m\ z. [[m: \text{nat}; z: C(m)]] ==> b(m, z): C(\text{succ}(m))]]$
 $==> \text{rec}(n, a, b) : C(n)$
 <proof>

<ML>

end

18 Order: 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

part-ord :: $[i, i] ==> o$ **where**

$part-ord(A,r) == irrefl(A,r) \ \& \ trans[A](r)$

definition

$linear :: [i,i] => o$ **where**
 $linear(A,r) == (ALL\ x:A.\ ALL\ y:A.\ \langle x,y \rangle : r \mid x=y \mid \langle y,x \rangle : r)$

definition

$tot-ord :: [i,i] => o$ **where**
 $tot-ord(A,r) == part-ord(A,r) \ \& \ linear(A,r)$

definition

$preorder-on(A, r) \equiv refl(A, r) \ \wedge \ trans[A](r)$

definition

$partial-order-on(A, r) \equiv preorder-on(A, r) \ \wedge \ antisym(r)$

abbreviation

$Preorder(r) \equiv preorder-on(field(r), r)$

abbreviation

$Partial-order(r) \equiv partial-order-on(field(r), r)$

definition

$well-ord :: [i,i] => o$ **where**
 $well-ord(A,r) == tot-ord(A,r) \ \& \ wf[A](r)$

definition

$mono-map :: [i,i,i,i] => i$ **where**
 $mono-map(A,r,B,s) ==$
 $\{f: A \rightarrow B.\ ALL\ x:A.\ ALL\ y:A.\ \langle x,y \rangle : r \rightarrow \langle f\ x, f\ y \rangle : s\}$

definition

$ord-iso :: [i,i,i,i] => i$ **where**
 $ord-iso(A,r,B,s) ==$
 $\{f: bij(A,B).\ ALL\ x:A.\ ALL\ y:A.\ \langle x,y \rangle : r \leftrightarrow \langle f\ x, f\ y \rangle : s\}$

definition

$pred :: [i,i,i] => i$ **where**
 $pred(A,x,r) == \{y:A.\ \langle y,x \rangle : r\}$

definition

$ord-iso-map :: [i,i,i,i] => i$ **where**
 $ord-iso-map(A,r,B,s) ==$
 $\bigcup x \in A.\ \bigcup y \in B.\ \bigcup f \in ord-iso(pred(A,x,r), r, pred(B,y,s), s). \{\langle x,y \rangle\}$

definition

$first :: [i, i, i] => o$ **where**
 $first(u, X, R) == u:X \ \& \ (ALL\ v:X.\ v \sim u \rightarrow \langle u,v \rangle : R)$

notation (*xsymbols*)

ord-iso ($(\langle -, - \rangle \cong / \langle -, - \rangle)$ 51)

18.1 Immediate Consequences of the Definitions

lemma *part-ord-Imp-asy*:

$part-ord(A,r) \implies asym(r \text{ Int } A*A)$

<proof>

lemma *linearE*:

$$\begin{aligned} & [linear(A,r); x:A; y:A; \\ & \quad \langle x,y \rangle:r \implies P; x=y \implies P; \langle y,x \rangle:r \implies P] \\ & \implies P \end{aligned}$$

<proof>

lemma *well-ordI*:

$[wf[A](r); linear(A,r)] \implies well-ord(A,r)$

<proof>

lemma *well-ord-is-wf*:

$well-ord(A,r) \implies wf[A](r)$

<proof>

lemma *well-ord-is-trans-on*:

$well-ord(A,r) \implies trans[A](r)$

<proof>

lemma *well-ord-is-linear*: $well-ord(A,r) \implies linear(A,r)$

<proof>

lemma *pred-iff*: $y : pred(A,x,r) \iff \langle y,x \rangle:r \ \& \ y:A$

<proof>

lemmas *predI = conjI [THEN pred-iff [THEN iffD2]]*

lemma *predE*: $[y : pred(A,x,r); [y:A; \langle y,x \rangle:r] \implies P] \implies P$

<proof>

lemma *pred-subset-under*: $pred(A,x,r) \leq r - \{x\}$

<proof>

lemma *pred-subset*: $pred(A,x,r) \leq A$

<proof>

lemma *pred-pred-eq*:

$\text{pred}(\text{pred}(A,x,r), y, r) = \text{pred}(A,x,r) \text{ Int } \text{pred}(A,y,r)$

<proof>

lemma *trans-pred-pred-eq*:

$\llbracket \text{trans}[A](r); \langle y,x \rangle : r; x:A; y:A \rrbracket$
 $\implies \text{pred}(\text{pred}(A,x,r), y, r) = \text{pred}(A,y,r)$

<proof>

18.2 Restricting an Ordering's Domain

lemma *part-ord-subset*:

$\llbracket \text{part-ord}(A,r); B \leq A \rrbracket \implies \text{part-ord}(B,r)$

<proof>

lemma *linear-subset*:

$\llbracket \text{linear}(A,r); B \leq A \rrbracket \implies \text{linear}(B,r)$

<proof>

lemma *tot-ord-subset*:

$\llbracket \text{tot-ord}(A,r); B \leq A \rrbracket \implies \text{tot-ord}(B,r)$

<proof>

lemma *well-ord-subset*:

$\llbracket \text{well-ord}(A,r); B \leq A \rrbracket \implies \text{well-ord}(B,r)$

<proof>

lemma *irrefl-Int-iff*: $\text{irrefl}(A,r \text{ Int } A*A) \longleftrightarrow \text{irrefl}(A,r)$

<proof>

lemma *trans-on-Int-iff*: $\text{trans}[A](r \text{ Int } A*A) \longleftrightarrow \text{trans}[A](r)$

<proof>

lemma *part-ord-Int-iff*: $\text{part-ord}(A,r \text{ Int } A*A) \longleftrightarrow \text{part-ord}(A,r)$

<proof>

lemma *linear-Int-iff*: $\text{linear}(A,r \text{ Int } A*A) \longleftrightarrow \text{linear}(A,r)$

<proof>

lemma *tot-ord-Int-iff*: $\text{tot-ord}(A,r \text{ Int } A*A) \longleftrightarrow \text{tot-ord}(A,r)$

<proof>

lemma *wf-on-Int-iff*: $\text{wf}[A](r \text{ Int } A*A) \longleftrightarrow \text{wf}[A](r)$

<proof>

lemma *well-ord-Int-iff*: $\text{well-ord}(A, r \text{ Int } A * A) \leftrightarrow \text{well-ord}(A, r)$
(proof)

18.3 Empty and Unit Domains

lemma *wf-on-any-0*: $\text{wf}[A](0)$
(proof)

18.3.1 Relations over the Empty Set

lemma *irrefl-0*: $\text{irrefl}(0, r)$
(proof)

lemma *trans-on-0*: $\text{trans}[0](r)$
(proof)

lemma *part-ord-0*: $\text{part-ord}(0, r)$
(proof)

lemma *linear-0*: $\text{linear}(0, r)$
(proof)

lemma *tot-ord-0*: $\text{tot-ord}(0, r)$
(proof)

lemma *wf-on-0*: $\text{wf}[0](r)$
(proof)

lemma *well-ord-0*: $\text{well-ord}(0, r)$
(proof)

18.3.2 The Empty Relation Well-Orders the Unit Set

by Grabczewski

lemma *tot-ord-unit*: $\text{tot-ord}(\{a\}, 0)$
(proof)

lemma *well-ord-unit*: $\text{well-ord}(\{a\}, 0)$
(proof)

18.4 Order-Isomorphisms

Suppes calls them "similarities"

lemma *mono-map-is-fun*: $f: \text{mono-map}(A, r, B, s) \implies f: A \rightarrow B$
(proof)

lemma *mono-map-is-inj*:

$\llbracket \text{linear}(A,r); \text{wf}[B](s); f: \text{mono-map}(A,r,B,s) \rrbracket \implies f: \text{inj}(A,B)$
 $\langle \text{proof} \rangle$

lemma *ord-isoI*:

$\llbracket f: \text{bij}(A, B);$
 $\quad \text{!!}x\ y. \llbracket x:A; y:A \rrbracket \implies \langle x, y \rangle : r \leftrightarrow \langle f'x, f'y \rangle : s \rrbracket$
 $\implies f: \text{ord-iso}(A,r,B,s)$
 $\langle \text{proof} \rangle$

lemma *ord-iso-is-mono-map*:

$f: \text{ord-iso}(A,r,B,s) \implies f: \text{mono-map}(A,r,B,s)$
 $\langle \text{proof} \rangle$

lemma *ord-iso-is-bij*:

$f: \text{ord-iso}(A,r,B,s) \implies f: \text{bij}(A,B)$
 $\langle \text{proof} \rangle$

lemma *ord-iso-apply*:

$\llbracket f: \text{ord-iso}(A,r,B,s); \langle x,y \rangle : r; x:A; y:A \rrbracket \implies \langle f'x, f'y \rangle : s$
 $\langle \text{proof} \rangle$

lemma *ord-iso-converse*:

$\llbracket f: \text{ord-iso}(A,r,B,s); \langle x,y \rangle : s; x:B; y:B \rrbracket$
 $\implies \langle \text{converse}(f) 'x, \text{converse}(f) 'y \rangle : r$
 $\langle \text{proof} \rangle$

lemma *ord-iso-refl*: $\text{id}(A): \text{ord-iso}(A,r,A,r)$

$\langle \text{proof} \rangle$

lemma *ord-iso-sym*: $f: \text{ord-iso}(A,r,B,s) \implies \text{converse}(f): \text{ord-iso}(B,s,A,r)$

$\langle \text{proof} \rangle$

lemma *mono-map-trans*:

$\llbracket g: \text{mono-map}(A,r,B,s); f: \text{mono-map}(B,s,C,t) \rrbracket$
 $\implies (f \circ g): \text{mono-map}(A,r,C,t)$
 $\langle \text{proof} \rangle$

lemma *ord-iso-trans*:

$\llbracket g: \text{ord-iso}(A,r,B,s); f: \text{ord-iso}(B,s,C,t) \rrbracket$
 $\implies (f \circ g): \text{ord-iso}(A,r,C,t)$
 $\langle \text{proof} \rangle$

lemma *mono-ord-isoI*:

$$\llbracket f: \text{mono-map}(A,r,B,s); g: \text{mono-map}(B,s,A,r);$$
$$f \circ g = \text{id}(B); g \circ f = \text{id}(A) \rrbracket \implies f: \text{ord-iso}(A,r,B,s)$$

<proof>

lemma *well-ord-mono-ord-isoI*:

$$\llbracket \text{well-ord}(A,r); \text{well-ord}(B,s);$$
$$f: \text{mono-map}(A,r,B,s); \text{converse}(f): \text{mono-map}(B,s,A,r) \rrbracket$$
$$\implies f: \text{ord-iso}(A,r,B,s)$$

<proof>

lemma *part-ord-ord-iso*:

$$\llbracket \text{part-ord}(B,s); f: \text{ord-iso}(A,r,B,s) \rrbracket \implies \text{part-ord}(A,r)$$

<proof>

lemma *linear-ord-iso*:

$$\llbracket \text{linear}(B,s); f: \text{ord-iso}(A,r,B,s) \rrbracket \implies \text{linear}(A,r)$$

<proof>

lemma *wf-on-ord-iso*:

$$\llbracket \text{wf}[B](s); f: \text{ord-iso}(A,r,B,s) \rrbracket \implies \text{wf}[A](r)$$

<proof>

lemma *well-ord-ord-iso*:

$$\llbracket \text{well-ord}(B,s); f: \text{ord-iso}(A,r,B,s) \rrbracket \implies \text{well-ord}(A,r)$$

<proof>

18.5 Main results of Kunen, Chapter 1 section 6

lemma *well-ord-iso-subset-lemma*:

$$\llbracket \text{well-ord}(A,r); f: \text{ord-iso}(A,r,A',r); A' \leq A; y: A \rrbracket$$
$$\implies \sim \langle f'y, y \rangle: r$$

<proof>

lemma *well-ord-iso-predE*:

$$\llbracket \text{well-ord}(A,r); f: \text{ord-iso}(A,r, \text{pred}(A,x,r), r); x:A \rrbracket \implies P$$

<proof>

lemma *well-ord-iso-pred-eq*:

$$\llbracket \text{well-ord}(A,r); f: \text{ord-iso}(\text{pred}(A,a,r), r, \text{pred}(A,c,r), r);$$
$$a:A; c:A \rrbracket \implies a=c$$

$\langle proof \rangle$

lemma *ord-iso-image-pred*:

$[[f : ord\text{-}iso(A,r,B,s); a:A]] \implies f \text{ `` } pred(A,a,r) = pred(B, f'a, s)$
 $\langle proof \rangle$

lemma *ord-iso-restrict-image*:

$[[f : ord\text{-}iso(A,r,B,s); C \leq A]] \implies restrict(f,C) : ord\text{-}iso(C, r, f''C, s)$
 $\langle proof \rangle$

lemma *ord-iso-restrict-pred*:

$[[f : ord\text{-}iso(A,r,B,s); a:A]] \implies restrict(f, pred(A,a,r)) : ord\text{-}iso(pred(A,a,r), r, pred(B, f'a, s), s)$
 $\langle proof \rangle$

lemma *well-ord-iso-preserving*:

$[[well\text{-}ord(A,r); well\text{-}ord(B,s); \langle a,c \rangle : r;$
 $f : ord\text{-}iso(pred(A,a,r), r, pred(B,b,s), s);$
 $g : ord\text{-}iso(pred(A,c,r), r, pred(B,d,s), s);$
 $a:A; c:A; b:B; d:B]] \implies \langle b,d \rangle : s$
 $\langle proof \rangle$

lemma *well-ord-iso-unique-lemma*:

$[[well\text{-}ord(A,r);$
 $f : ord\text{-}iso(A,r, B,s); g : ord\text{-}iso(A,r, B,s); y : A]] \implies \sim \langle g'y, f'y \rangle : s$
 $\langle proof \rangle$

lemma *well-ord-iso-unique*: $[[well\text{-}ord(A,r);$

$f : ord\text{-}iso(A,r, B,s); g : ord\text{-}iso(A,r, B,s)] \implies f = g$
 $\langle proof \rangle$

18.6 Towards Kunen's Theorem 6.3: Linearity of the Similarity Relation

lemma *ord-iso-map-subset*: $ord\text{-}iso\text{-}map(A,r,B,s) \leq A * B$

$\langle proof \rangle$

lemma *domain-ord-iso-map*: $domain(ord\text{-}iso\text{-}map(A,r,B,s)) \leq A$

$\langle proof \rangle$

lemma *range-ord-iso-map*: $range(ord\text{-}iso\text{-}map(A,r,B,s)) \leq B$

<proof>

lemma *converse-ord-iso-map*:

$$\text{converse}(\text{ord-iso-map}(A,r,B,s)) = \text{ord-iso-map}(B,s,A,r)$$

<proof>

lemma *function-ord-iso-map*:

$$\text{well-ord}(B,s) \implies \text{function}(\text{ord-iso-map}(A,r,B,s))$$

<proof>

lemma *ord-iso-map-fun*: $\text{well-ord}(B,s) \implies \text{ord-iso-map}(A,r,B,s)$

$$: \text{domain}(\text{ord-iso-map}(A,r,B,s)) \rightarrow \text{range}(\text{ord-iso-map}(A,r,B,s))$$

<proof>

lemma *ord-iso-map-mono-map*:

$$[[\text{well-ord}(A,r); \text{well-ord}(B,s)]]$$

$$\implies \text{ord-iso-map}(A,r,B,s)$$

$$: \text{mono-map}(\text{domain}(\text{ord-iso-map}(A,r,B,s)), r, \\ \text{range}(\text{ord-iso-map}(A,r,B,s)), s)$$

<proof>

lemma *ord-iso-map-ord-iso*:

$$[[\text{well-ord}(A,r); \text{well-ord}(B,s)]] \implies \text{ord-iso-map}(A,r,B,s)$$

$$: \text{ord-iso}(\text{domain}(\text{ord-iso-map}(A,r,B,s)), r, \\ \text{range}(\text{ord-iso-map}(A,r,B,s)), s)$$

<proof>

lemma *domain-ord-iso-map-subset*:

$$[[\text{well-ord}(A,r); \text{well-ord}(B,s);$$

$$a: A; a \sim: \text{domain}(\text{ord-iso-map}(A,r,B,s))]]$$

$$\implies \text{domain}(\text{ord-iso-map}(A,r,B,s)) \leq \text{pred}(A, a, r)$$

<proof>

lemma *domain-ord-iso-map-cases*:

$$[[\text{well-ord}(A,r); \text{well-ord}(B,s)]]$$

$$\implies \text{domain}(\text{ord-iso-map}(A,r,B,s)) = A \mid$$

$$(\text{EX } x:A. \text{domain}(\text{ord-iso-map}(A,r,B,s)) = \text{pred}(A,x,r))$$

<proof>

lemma *range-ord-iso-map-cases*:

$$[[\text{well-ord}(A,r); \text{well-ord}(B,s)]]$$

$$\implies \text{range}(\text{ord-iso-map}(A,r,B,s)) = B \mid$$

$$(\text{EX } y:B. \text{range}(\text{ord-iso-map}(A,r,B,s)) = \text{pred}(B,y,s))$$

<proof>

Kunen's Theorem 6.3: Fundamental Theorem for Well-Ordered Sets

theorem *well-ord-trichotomy*:

[[*well-ord*(A,r); *well-ord*(B,s)]]
 \implies *ord-iso-map*(A,r,B,s) : *ord-iso*(A, r, B, s) |
 (*EX* $x:A$. *ord-iso-map*(A,r,B,s) : *ord-iso*(*pred*(A,x,r), r, B, s)) |
 (*EX* $y:B$. *ord-iso-map*(A,r,B,s) : *ord-iso*($A, r, \text{pred}(B,y,s), s$))
 ⟨*proof*⟩

18.7 Miscellaneous Results by Krzysztof Grabczewski

lemma *irrefl-converse*: *irrefl*(A,r) \implies *irrefl*($A,\text{converse}(r)$)
 ⟨*proof*⟩

lemma *trans-on-converse*: *trans*[A](r) \implies *trans*[A](*converse*(r))
 ⟨*proof*⟩

lemma *part-ord-converse*: *part-ord*(A,r) \implies *part-ord*($A,\text{converse}(r)$)
 ⟨*proof*⟩

lemma *linear-converse*: *linear*(A,r) \implies *linear*($A,\text{converse}(r)$)
 ⟨*proof*⟩

lemma *tot-ord-converse*: *tot-ord*(A,r) \implies *tot-ord*($A,\text{converse}(r)$)
 ⟨*proof*⟩

lemma *first-is-elem*: *first*(b,B,r) \implies $b:B$
 ⟨*proof*⟩

lemma *well-ord-imp-ex1-first*:
 [[*well-ord*(A,r); $B \leq A$; $B \sim 0$]] \implies (*EX!* b . *first*(b,B,r))
 ⟨*proof*⟩

lemma *the-first-in*:
 [[*well-ord*(A,r); $B \leq A$; $B \sim 0$]] \implies (*THE* b . *first*(b,B,r)) : B
 ⟨*proof*⟩

18.8 Lemmas for the Reflexive Orders

lemma *subset-vimage-vimage-iff*:
 [[*Preorder*(r); $A \subseteq \text{field}(r)$; $B \subseteq \text{field}(r)$]] \implies
 $r - \llbracket A \subseteq r - \llbracket B \llbracket \iff (\text{ALL } a:A. \text{EX } b:B. \langle a, b \rangle : r)$
 ⟨*proof*⟩

lemma *subset-vimage1-vimage1-iff*:
 [[*Preorder*(r); $a : \text{field}(r)$; $b : \text{field}(r)$]] \implies
 $r - \llbracket \{a\} \subseteq r - \llbracket \{b\} \llbracket \iff \langle a, b \rangle : r$
 ⟨*proof*⟩

lemma *Refl-antisym-eq-Image1-Image1-iff*:
 $\llbracket \text{refl}(\text{field}(r), r); \text{antisym}(r); a : \text{field}(r); b : \text{field}(r) \rrbracket \implies$
 $r \text{ “ } \{a\} = r \text{ “ } \{b\} \iff a = b$
<proof>

lemma *Partial-order-eq-Image1-Image1-iff*:
 $\llbracket \text{Partial-order}(r); a : \text{field}(r); b : \text{field}(r) \rrbracket \implies$
 $r \text{ “ } \{a\} = r \text{ “ } \{b\} \iff a = b$
<proof>

lemma *Refl-antisym-eq-vimage1-vimage1-iff*:
 $\llbracket \text{refl}(\text{field}(r), r); \text{antisym}(r); a : \text{field}(r); b : \text{field}(r) \rrbracket \implies$
 $r \text{ -“ } \{a\} = r \text{ -“ } \{b\} \iff a = b$
<proof>

lemma *Partial-order-eq-vimage1-vimage1-iff*:
 $\llbracket \text{Partial-order}(r); a : \text{field}(r); b : \text{field}(r) \rrbracket \implies$
 $r \text{ -“ } \{a\} = r \text{ -“ } \{b\} \iff a = b$
<proof>

end

19 OrderArith: Combining Orderings: Foundations of Ordinal Arithmetic

theory *OrderArith* imports *Order Sum Ordinal* begin

definition

radd :: $[i, i, i, i] \Rightarrow i$ **where**
 $\text{radd}(A, r, B, s) ==$
 $\{z: (A+B) * (A+B).$
 $(EX\ x\ y. z = \langle \text{Inl}(x), \text{Inr}(y) \rangle) \mid$
 $(EX\ x'\ x. z = \langle \text{Inl}(x'), \text{Inl}(x) \rangle \ \& \ \langle x', x \rangle : r) \mid$
 $(EX\ y'\ y. z = \langle \text{Inr}(y'), \text{Inr}(y) \rangle \ \& \ \langle y', y \rangle : s)\}$

definition

rmult :: $[i, i, i, i] \Rightarrow i$ **where**
 $\text{rmult}(A, r, B, s) ==$
 $\{z: (A*B) * (A*B).$
 $EX\ x'\ y'\ x\ y. z = \langle \langle x', y' \rangle, \langle x, y \rangle \rangle \ \&$
 $(\langle x', x \rangle : r \mid (x' = x \ \& \ \langle y', y \rangle : s))\}$

definition

rvimage :: $[i, i, i] \Rightarrow i$ **where**

$rvimage(A,f,r) == \{z: A*A. EX x y. z = \langle x,y \rangle \ \& \ \langle f'x,f'y \rangle: r\}$

definition

$measure :: [i, i \Rightarrow i] \Rightarrow i$ **where**
 $measure(A,f) == \{\langle x,y \rangle: A*A. f(x) < f(y)\}$

19.1 Addition of Relations – Disjoint Sum

19.1.1 Rewrite rules. Can be used to obtain introduction rules

lemma *radd-Inl-Inr-iff* [*iff*]:

$\langle Inl(a), Inr(b) \rangle : radd(A,r,B,s) \langle - \rangle a:A \ \& \ b:B$

\langle proof \rangle

lemma *radd-Inl-iff* [*iff*]:

$\langle Inl(a'), Inl(a) \rangle : radd(A,r,B,s) \langle - \rangle a':A \ \& \ a:A \ \& \ \langle a',a \rangle: r$

\langle proof \rangle

lemma *radd-Inr-iff* [*iff*]:

$\langle Inr(b'), Inr(b) \rangle : radd(A,r,B,s) \langle - \rangle b':B \ \& \ b:B \ \& \ \langle b',b \rangle: s$

\langle proof \rangle

lemma *radd-Inr-Inl-iff* [*simp*]:

$\langle Inr(b), Inl(a) \rangle : radd(A,r,B,s) \langle - \rangle False$

\langle proof \rangle

declare *radd-Inr-Inl-iff* [*THEN iffD1, dest!*]

19.1.2 Elimination Rule

lemma *raddE*:

$[[\langle p',p \rangle : radd(A,r,B,s);$
 $!!x y. [[p'=Inl(x); x:A; p=Inr(y); y:B]] ==> Q;$
 $!!x' x. [[p'=Inl(x'); p=Inl(x); \langle x',x \rangle: r; x':A; x:A]] ==> Q;$
 $!!y' y. [[p'=Inr(y'); p=Inr(y); \langle y',y \rangle: s; y':B; y:B]] ==> Q$
 $]] ==> Q$

\langle proof \rangle

19.1.3 Type checking

lemma *radd-type*: $radd(A,r,B,s) \leq (A+B) * (A+B)$

\langle proof \rangle

lemmas *field-radd* = *radd-type* [*THEN field-rel-subset*]

19.1.4 Linearity

lemma *linear-radd*:

$[[linear(A,r); linear(B,s)]] ==> linear(A+B,radd(A,r,B,s))$

\langle proof \rangle

19.1.5 Well-foundedness

lemma *wf-on-radd*: $[[\text{wf}[A](r); \text{wf}[B](s)]] \implies \text{wf}[A+B](\text{radd}(A,r,B,s))$
<proof>

lemma *wf-radd*: $[[\text{wf}(r); \text{wf}(s)]] \implies \text{wf}(\text{radd}(\text{field}(r),r,\text{field}(s),s))$
<proof>

lemma *well-ord-radd*:
 $[[\text{well-ord}(A,r); \text{well-ord}(B,s)]] \implies \text{well-ord}(A+B, \text{radd}(A,r,B,s))$
<proof>

19.1.6 An ord-iso congruence law

lemma *sum-bij*:
 $[[f: \text{bij}(A,C); g: \text{bij}(B,D)]]$
 $\implies (\text{lam } z:A+B. \text{case}(\%x. \text{Inl}(f'x), \%y. \text{Inr}(g'y), z)) : \text{bij}(A+B, C+D)$
<proof>

lemma *sum-ord-iso-cong*:
 $[[f: \text{ord-iso}(A,r,A',r'); g: \text{ord-iso}(B,s,B',s')]] \implies$
 $(\text{lam } z:A+B. \text{case}(\%x. \text{Inl}(f'x), \%y. \text{Inr}(g'y), z))$
 $: \text{ord-iso}(A+B, \text{radd}(A,r,B,s), A'+B', \text{radd}(A',r',B',s'))$
<proof>

lemma *sum-disjoint-bij*: $A \text{ Int } B = 0 \implies$
 $(\text{lam } z:A+B. \text{case}(\%x. x, \%y. y, z)) : \text{bij}(A+B, A \text{ Un } B)$
<proof>

19.1.7 Associativity

lemma *sum-assoc-bij*:
 $(\text{lam } z:(A+B)+C. \text{case}(\text{case}(\text{Inl}, \%y. \text{Inr}(\text{Inl}(y))), \%y. \text{Inr}(\text{Inr}(y)), z))$
 $: \text{bij}((A+B)+C, A+(B+C))$
<proof>

lemma *sum-assoc-ord-iso*:
 $(\text{lam } z:(A+B)+C. \text{case}(\text{case}(\text{Inl}, \%y. \text{Inr}(\text{Inl}(y))), \%y. \text{Inr}(\text{Inr}(y)), z))$
 $: \text{ord-iso}((A+B)+C, \text{radd}(A+B, \text{radd}(A,r,B,s), C, t),$
 $A+(B+C), \text{radd}(A, r, B+C, \text{radd}(B,s,C,t)))$
<proof>

19.2 Multiplication of Relations – Lexicographic Product

19.2.1 Rewrite rule. Can be used to obtain introduction rules

lemma *rmult-iff [iff]*:
 $\langle\langle a',b' \rangle, \langle a,b \rangle\rangle : \text{rmult}(A,r,B,s) \langle\langle - \rangle\rangle$
 $(\langle a',a \rangle : r \ \& \ a':A \ \& \ a:A \ \& \ b':B \ \& \ b:B) \mid$
 $(\langle b',b \rangle : s \ \& \ a'=a \ \& \ a:A \ \& \ b':B \ \& \ b:B)$

<proof>

lemma *rmultE*:

$$\begin{aligned} & \llbracket \langle \langle a', b' \rangle, \langle a, b \rangle \rangle : \text{rmult}(A, r, B, s); \\ & \quad \llbracket \langle a', a \rangle : r; a' : A; a : A; b' : B; b : B \rrbracket \implies Q; \\ & \quad \llbracket \langle b', b \rangle : s; a : A; a' = a; b' : B; b : B \rrbracket \implies Q \\ & \rrbracket \implies Q \end{aligned}$$

<proof>

19.2.2 Type checking

lemma *rmult-type*: $\text{rmult}(A, r, B, s) \leq (A * B) * (A * B)$

<proof>

lemmas *field-rmult = rmult-type [THEN field-rel-subset]*

19.2.3 Linearity

lemma *linear-rmult*:

$$\llbracket \text{linear}(A, r); \text{linear}(B, s) \rrbracket \implies \text{linear}(A * B, \text{rmult}(A, r, B, s))$$

<proof>

19.2.4 Well-foundedness

lemma *wf-on-rmult*: $\llbracket \text{wf}[A](r); \text{wf}[B](s) \rrbracket \implies \text{wf}[A * B](\text{rmult}(A, r, B, s))$

<proof>

lemma *wf-rmult*: $\llbracket \text{wf}(r); \text{wf}(s) \rrbracket \implies \text{wf}(\text{rmult}(\text{field}(r), r, \text{field}(s), s))$

<proof>

lemma *well-ord-rmult*:

$$\llbracket \text{well-ord}(A, r); \text{well-ord}(B, s) \rrbracket \implies \text{well-ord}(A * B, \text{rmult}(A, r, B, s))$$

<proof>

19.2.5 An ord-iso congruence law

lemma *prod-bij*:

$$\begin{aligned} & \llbracket f : \text{bij}(A, C); g : \text{bij}(B, D) \rrbracket \\ & \implies (\text{lam } \langle x, y \rangle : A * B. \langle f'x, g'y \rangle) : \text{bij}(A * B, C * D) \end{aligned}$$

<proof>

lemma *prod-ord-iso-cong*:

$$\begin{aligned} & \llbracket f : \text{ord-iso}(A, r, A', r'); g : \text{ord-iso}(B, s, B', s') \rrbracket \\ & \implies (\text{lam } \langle x, y \rangle : A * B. \langle f'x, g'y \rangle) \\ & \quad : \text{ord-iso}(A * B, \text{rmult}(A, r, B, s), A' * B', \text{rmult}(A', r', B', s')) \end{aligned}$$

<proof>

lemma *singleton-prod-bij*: $(\text{lam } z : A. \langle x, z \rangle) : \text{bij}(A, \{x\} * A)$

$\langle \text{proof} \rangle$

lemma *singleton-prod-ord-iso*:

$\text{well-ord}(\{x\}, xr) \implies$
 $(\text{lam } z:A. \langle x, z \rangle) : \text{ord-iso}(A, r, \{x\} * A, \text{rmult}(\{x\}, xr, A, r))$

$\langle \text{proof} \rangle$

lemma *prod-sum-singleton-bij*:

$a \sim : C \implies$
 $(\text{lam } x:C * B + D. \text{case}(\%x. x, \%y. \langle a, y \rangle, x))$
 $: \text{bij}(C * B + D, C * B \text{ Un } \{a\} * D)$

$\langle \text{proof} \rangle$

lemma *prod-sum-singleton-ord-iso*:

$\llbracket a:A; \text{well-ord}(A, r) \rrbracket \implies$
 $(\text{lam } x:\text{pred}(A, a, r) * B + \text{pred}(B, b, s). \text{case}(\%x. x, \%y. \langle a, y \rangle, x))$
 $: \text{ord-iso}(\text{pred}(A, a, r) * B + \text{pred}(B, b, s),$
 $\text{radd}(A * B, \text{rmult}(A, r, B, s), B, s),$
 $\text{pred}(A, a, r) * B \text{ Un } \{a\} * \text{pred}(B, b, s), \text{rmult}(A, r, B, s))$

$\langle \text{proof} \rangle$

19.2.6 Distributive law

lemma *sum-prod-distrib-bij*:

$(\text{lam } \langle x, z \rangle : (A + B) * C. \text{case}(\%y. \text{Inl}(\langle y, z \rangle), \%y. \text{Inr}(\langle y, z \rangle), x))$
 $: \text{bij}((A + B) * C, (A * C) + (B * C))$

$\langle \text{proof} \rangle$

lemma *sum-prod-distrib-ord-iso*:

$(\text{lam } \langle x, z \rangle : (A + B) * C. \text{case}(\%y. \text{Inl}(\langle y, z \rangle), \%y. \text{Inr}(\langle y, z \rangle), x))$
 $: \text{ord-iso}((A + B) * C, \text{rmult}(A + B, \text{radd}(A, r, B, s), C, t),$
 $(A * C) + (B * C), \text{radd}(A * C, \text{rmult}(A, r, C, t), B * C, \text{rmult}(B, s, C, t)))$

$\langle \text{proof} \rangle$

19.2.7 Associativity

lemma *prod-assoc-bij*:

$(\text{lam } \langle \langle x, y \rangle, z \rangle : (A * B) * C. \langle x, \langle y, z \rangle \rangle) : \text{bij}((A * B) * C, A * (B * C))$

$\langle \text{proof} \rangle$

lemma *prod-assoc-ord-iso*:

$(\text{lam } \langle \langle x, y \rangle, z \rangle : (A * B) * C. \langle x, \langle y, z \rangle \rangle)$
 $: \text{ord-iso}((A * B) * C, \text{rmult}(A * B, \text{rmult}(A, r, B, s), C, t),$
 $A * (B * C), \text{rmult}(A, r, B * C, \text{rmult}(B, s, C, t)))$

$\langle \text{proof} \rangle$

19.3 Inverse Image of a Relation

19.3.1 Rewrite rule

lemma *rvimage-iff*: $\langle a, b \rangle : \text{rvimage}(A, f, r) \iff \langle f'a, f'b \rangle : r \ \& \ a:A \ \& \ b:A$
<proof>

19.3.2 Type checking

lemma *rvimage-type*: $\text{rvimage}(A, f, r) \leq A * A$
<proof>

lemmas *field-rvimage = rvimage-type* [THEN *field-rel-subset*]

lemma *rvimage-converse*: $\text{rvimage}(A, f, \text{converse}(r)) = \text{converse}(\text{rvimage}(A, f, r))$
<proof>

19.3.3 Partial Ordering Properties

lemma *irrefl-rvimage*:
 $\llbracket f : \text{inj}(A, B); \text{irrefl}(B, r) \rrbracket \implies \text{irrefl}(A, \text{rvimage}(A, f, r))$
<proof>

lemma *trans-on-rvimage*:
 $\llbracket f : \text{inj}(A, B); \text{trans}[B](r) \rrbracket \implies \text{trans}[A](\text{rvimage}(A, f, r))$
<proof>

lemma *part-ord-rvimage*:
 $\llbracket f : \text{inj}(A, B); \text{part-ord}(B, r) \rrbracket \implies \text{part-ord}(A, \text{rvimage}(A, f, r))$
<proof>

19.3.4 Linearity

lemma *linear-rvimage*:
 $\llbracket f : \text{inj}(A, B); \text{linear}(B, r) \rrbracket \implies \text{linear}(A, \text{rvimage}(A, f, r))$
<proof>

lemma *tot-ord-rvimage*:
 $\llbracket f : \text{inj}(A, B); \text{tot-ord}(B, r) \rrbracket \implies \text{tot-ord}(A, \text{rvimage}(A, f, r))$
<proof>

19.3.5 Well-foundedness

lemma *wf-rvimage* [*intro!*]: $\text{wf}(r) \implies \text{wf}(\text{rvimage}(A, f, r))$
<proof>

But note that the combination of *wf-imp-wf-on* and *wf-rvimage* gives $\text{wf}(r) \implies \text{wf}[C](\text{rvimage}(A, f, r))$

lemma *wf-on-rvimage*: $\llbracket f : A \rightarrow B; \text{wf}[B](r) \rrbracket \implies \text{wf}[A](\text{rvimage}(A, f, r))$
<proof>

lemma *well-ord-rvimage*:

$\llbracket f: \text{inj}(A,B); \text{well-ord}(B,r) \rrbracket \implies \text{well-ord}(A, \text{rvimage}(A,f,r))$
 $\langle \text{proof} \rangle$

lemma *ord-iso-rvimage*:

$f: \text{bij}(A,B) \implies f: \text{ord-iso}(A, \text{rvimage}(A,f,s), B, s)$
 $\langle \text{proof} \rangle$

lemma *ord-iso-rvimage-eq*:

$f: \text{ord-iso}(A,r, B,s) \implies \text{rvimage}(A,f,s) = r \text{ Int } A*A$
 $\langle \text{proof} \rangle$

19.4 Every well-founded relation is a subset of some inverse image of an ordinal

lemma *wf-rvimage-Ord*: $\text{Ord}(i) \implies \text{wf}(\text{rvimage}(A, f, \text{Memrel}(i)))$
 $\langle \text{proof} \rangle$

definition

$w\text{frank} :: [i,i] \implies i$ **where**
 $w\text{frank}(r,a) == w\text{frec}(r, a, \%x f. \bigcup y \in r - \{x\}. \text{succ}(f'y))$

definition

$w\text{ftype} :: i \implies i$ **where**
 $w\text{ftype}(r) == \bigcup y \in \text{range}(r). \text{succ}(w\text{frank}(r,y))$

lemma *wfrank*: $\text{wf}(r) \implies w\text{frank}(r,a) = (\bigcup y \in r - \{a\}. \text{succ}(w\text{frank}(r,y)))$
 $\langle \text{proof} \rangle$

lemma *Ord-wfrank*: $\text{wf}(r) \implies \text{Ord}(w\text{frank}(r,a))$
 $\langle \text{proof} \rangle$

lemma *wfrank-lt*: $\llbracket \text{wf}(r); \langle a,b \rangle \in r \rrbracket \implies w\text{frank}(r,a) < w\text{frank}(r,b)$
 $\langle \text{proof} \rangle$

lemma *Ord-wftype*: $\text{wf}(r) \implies \text{Ord}(w\text{ftype}(r))$
 $\langle \text{proof} \rangle$

lemma *wftypeI*: $\llbracket \text{wf}(r); x \in \text{field}(r) \rrbracket \implies w\text{frank}(r,x) \in w\text{ftype}(r)$
 $\langle \text{proof} \rangle$

lemma *wf-imp-subset-rvimage*:

$\llbracket \text{wf}(r); r \subseteq A*A \rrbracket \implies \exists i f. \text{Ord}(i) \ \& \ r \leq \text{rvimage}(A, f, \text{Memrel}(i))$
 $\langle \text{proof} \rangle$

theorem *wf-iff-subset-rvimage*:

$relation(r) ==> wf(r) <-> (\exists i f A. Ord(i) \ \& \ r \leq rvimage(A, f, Memrel(i)))$
<proof>

19.5 Other Results

lemma *wf-times*: $A \ Int \ B = 0 ==> wf(A*B)$

<proof>

Could also be used to prove *wf-radd*

lemma *wf-Un*:

$[[\ range(r) \ Int \ domain(s) = 0; \ wf(r); \ wf(s) \]] ==> wf(r \ Un \ s)$
<proof>

19.5.1 The Empty Relation

lemma *wf0*: $wf(0)$

<proof>

lemma *linear0*: $linear(0,0)$

<proof>

lemma *well-ord0*: $well-ord(0,0)$

<proof>

19.5.2 The "measure" relation is useful with wfrec

lemma *measure-eq-rvimage-Memrel*:

$measure(A,f) = rvimage(A, Lambda(A,f), Memrel(Collect(RepFun(A,f), Ord)))$
<proof>

lemma *wf-measure [iff]*: $wf(measure(A,f))$

<proof>

lemma *measure-iff [iff]*: $<x,y> : measure(A,f) <-> x:A \ \& \ y:A \ \& \ f(x) < f(y)$

<proof>

lemma *linear-measure*:

assumes *Ord**f*: $!!x. x \in A ==> Ord(f(x))$

and *inj*: $!!x \ y. [[x \in A; y \in A; f(x) = f(y) \]] ==> x=y$

shows $linear(A, measure(A,f))$

<proof>

lemma *wf-on-measure*: $wf[B](measure(A,f))$

<proof>

lemma *well-ord-measure*:

assumes *Ord**f*: $!!x. x \in A ==> Ord(f(x))$

and *inj*: $!!x \ y. [[x \in A; y \in A; f(x) = f(y) \]] ==> x=y$

shows $well-ord(A, measure(A,f))$

<proof>

lemma *measure-type*: $measure(A,f) \leq A * A$
<proof>

19.5.3 Well-foundedness of Unions

lemma *wf-on-Union*:

assumes *wfA*: $wf[A](r)$
 and *wfB*: $\forall a. a \in A \implies wf[B(a)](s)$
 and *ok*: $\forall a u v. [\langle u,v \rangle \in s; v \in B(a); a \in A]$
 $\implies (\exists a' \in A. \langle a',a \rangle \in r \ \& \ u \in B(a')) \mid u \in B(a)$
shows $wf[\bigcup_{a \in A} B(a)](s)$
<proof>

19.5.4 Bijections involving Powersets

lemma *Pow-sum-bij*:

$(\lambda Z \in Pow(A+B). \langle \{x \in A. Inl(x) \in Z\}, \{y \in B. Inr(y) \in Z\} \rangle)$
 $\in bij(Pow(A+B), Pow(A) * Pow(B))$
<proof>

As a special case, we have $bij(Pow(A \times B), A \rightarrow Pow(B))$

lemma *Pow-Sigma-bij*:

$(\lambda r \in Pow(Sigma(A,B)). \lambda x \in A. r \text{ `` } \{x\})$
 $\in bij(Pow(Sigma(A,B)), \Pi x \in A. Pow(B(x)))$
<proof>

end

20 OrderType: Order Types and Ordinal Arithmetic

theory *OrderType* **imports** *OrderArith OrdQuant Nat-ZF* **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**
 $ordermap(A,r) == lam x:A. wfrec[A](r, x, \%x f. f \text{ `` } pred(A,x,r))$

definition

ordertype :: $[i,i] \Rightarrow i$ **where**
 $ordertype(A,r) == ordermap(A,r) \text{ `` } A$

definition

Ord-alt :: $i \Rightarrow o$ **where**
Ord-alt(X) == *well-ord*(X , *Memrel*(X)) & (*ALL* $u:X$. $u = \text{pred}(X, u, \text{Memrel}(X))$)

definition

ordify :: $i \Rightarrow i$ **where**
ordify(x) == *if* *Ord*(x) *then* x *else* 0

definition

omult :: $[i, i] \Rightarrow i$ (**infixl** ** 70) **where**
 $i ** j == \text{ordertype}(j * i, \text{rmult}(j, \text{Memrel}(j), i, \text{Memrel}(i)))$

definition

raw-oadd :: $[i, i] \Rightarrow i$ **where**
raw-oadd(i, j) == *ordertype*($i + j$, *radd*($i, \text{Memrel}(i), j, \text{Memrel}(j)$))

definition

oadd :: $[i, i] \Rightarrow i$ (**infixl** ++ 65) **where**
 $i ++ j == \text{raw-oadd}(\text{ordify}(i), \text{ordify}(j))$

definition

odiff :: $[i, i] \Rightarrow i$ (**infixl** -- 65) **where**
 $i -- j == \text{ordertype}(i - j, \text{Memrel}(i))$

notation (*xsymbols*)

omult (**infixl** $\times \times$ 70)

notation (*HTML output*)

omult (**infixl** $\times \times$ 70)

20.1 Proofs needing the combination of Ordinal.thy and Order.thy

lemma *le-well-ord-Memrel*: $j \leq i \implies \text{well-ord}(j, \text{Memrel}(i))$
 <proof>

lemmas *well-ord-Memrel* = *le-refl* [THEN *le-well-ord-Memrel*]

lemma *lt-pred-Memrel*:

$j < i \implies \text{pred}(i, j, \text{Memrel}(i)) = j$
 <proof>

lemma *pred-Memrel*:

$x:A \implies \text{pred}(A, x, \text{Memrel}(A)) = A \text{ Int } x$
<proof>

lemma *Ord-iso-implies-eq-lemma*:

$[[j < i; f: \text{ord-iso}(i, \text{Memrel}(i), j, \text{Memrel}(j))]] \implies R$
<proof>

lemma *Ord-iso-implies-eq*:

$[[\text{Ord}(i); \text{Ord}(j); f: \text{ord-iso}(i, \text{Memrel}(i), j, \text{Memrel}(j))]] \implies i=j$
<proof>

20.2 Ordermap and ordertype

lemma *ordermap-type*:

$\text{ordermap}(A, r) : A \rightarrow \text{ordertype}(A, r)$
<proof>

20.2.1 Unfolding of ordermap

lemma *ordermap-eq-image*:

$[[\text{wf}[A](r); x:A]] \implies \text{ordermap}(A, r) \text{ ‘ } x = \text{ordermap}(A, r) \text{ ‘ ‘ } \text{pred}(A, x, r)$
<proof>

lemma *ordermap-pred-unfold*:

$[[\text{wf}[A](r); x:A]] \implies \text{ordermap}(A, r) \text{ ‘ } x = \{ \text{ordermap}(A, r) \text{ ‘ } y . y : \text{pred}(A, x, r) \}$
<proof>

lemmas *ordermap-unfold = ordermap-pred-unfold [simplified pred-def]*

20.2.2 Showing that ordermap, ordertype yield ordinals

lemma *Ord-ordermap*:

$[[\text{well-ord}(A, r); x:A]] \implies \text{Ord}(\text{ordermap}(A, r) \text{ ‘ } x)$
<proof>

lemma *Ord-ordertype*:

$\text{well-ord}(A, r) \implies \text{Ord}(\text{ordertype}(A, r))$
<proof>

20.2.3 ordermap preserves the orderings in both directions

lemma *ordermap-mono*:

$$\begin{aligned} & \llbracket \langle w, x \rangle : r; \text{wf}[A](r); w : A; x : A \rrbracket \\ & \implies \text{ordermap}(A, r) 'w : \text{ordermap}(A, r) 'x \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *converse-ordermap-mono*:

$$\begin{aligned} & \llbracket \text{ordermap}(A, r) 'w : \text{ordermap}(A, r) 'x; \text{well-ord}(A, r); w : A; x : A \rrbracket \\ & \implies \langle w, x \rangle : r \\ \langle \text{proof} \rangle \end{aligned}$$

lemmas *ordermap-surj* =

ordermap-type [THEN *surj-image*, *unfolded ordertype-def* [*symmetric*]]

lemma *ordermap-bij*:

$$\text{well-ord}(A, r) \implies \text{ordermap}(A, r) : \text{bij}(A, \text{ordertype}(A, r))$$

$$\langle \text{proof} \rangle$$

20.2.4 Isomorphisms involving ordertype

lemma *ordertype-ord-iso*:

$$\text{well-ord}(A, r)$$

$$\implies \text{ordermap}(A, r) : \text{ord-iso}(A, r, \text{ordertype}(A, r), \text{Memrel}(\text{ordertype}(A, r)))$$

$$\langle \text{proof} \rangle$$

lemma *ordertype-eq*:

$$\llbracket f : \text{ord-iso}(A, r, B, s); \text{well-ord}(B, s) \rrbracket$$

$$\implies \text{ordertype}(A, r) = \text{ordertype}(B, s)$$

$$\langle \text{proof} \rangle$$

lemma *ordertype-eq-imp-ord-iso*:

$$\llbracket \text{ordertype}(A, r) = \text{ordertype}(B, s); \text{well-ord}(A, r); \text{well-ord}(B, s) \rrbracket$$

$$\implies \text{EX } f. f : \text{ord-iso}(A, r, B, s)$$

$$\langle \text{proof} \rangle$$

20.2.5 Basic equalities for ordertype

lemma *le-ordertype-Memrel*: $j \text{ le } i \implies \text{ordertype}(j, \text{Memrel}(i)) = j$

$\langle \text{proof} \rangle$

lemmas *ordertype-Memrel* = *le-refl* [THEN *le-ordertype-Memrel*]

lemma *ordertype-0* [*simp*]: $\text{ordertype}(0, r) = 0$

$\langle \text{proof} \rangle$

lemmas *bij-ordertype-vimage* = *ord-iso-rvimage* [THEN *ordertype-eq*]

20.2.6 A fundamental unfolding law for ordertype.

lemma *ordermap-pred-eq-ordermap*:

$[[\text{well-ord}(A,r); y:A; z: \text{pred}(A,y,r)]]$
 $\implies \text{ordermap}(\text{pred}(A,y,r), r) \text{ ` } z = \text{ordermap}(A, r) \text{ ` } z$
(*proof*)

lemma *ordertype-unfold*:

$\text{ordertype}(A,r) = \{ \text{ordermap}(A,r) \text{ ` } y . y : A \}$
(*proof*)

Theorems by Krzysztof Grabczewski; proofs simplified by lcp

lemma *ordertype-pred-subset*: $[[\text{well-ord}(A,r); x:A]]$ \implies

$\text{ordertype}(\text{pred}(A,x,r),r) \leq \text{ordertype}(A,r)$
(*proof*)

lemma *ordertype-pred-lt*:

$[[\text{well-ord}(A,r); x:A]]$
 $\implies \text{ordertype}(\text{pred}(A,x,r),r) < \text{ordertype}(A,r)$
(*proof*)

lemma *ordertype-pred-unfold*:

$\text{well-ord}(A,r)$
 $\implies \text{ordertype}(A,r) = \{ \text{ordertype}(\text{pred}(A,x,r),r). x:A \}$
(*proof*)

20.3 Alternative definition of ordinal

lemma *Ord-is-Ord-alt*: $\text{Ord}(i) \implies \text{Ord-alt}(i)$

(*proof*)

lemma *Ord-alt-is-Ord*:

$\text{Ord-alt}(i) \implies \text{Ord}(i)$
(*proof*)

20.4 Ordinal Addition

20.4.1 Order Type calculations for radd

Addition with 0

lemma *bij-sum-0*: $(\text{lam } z:A+0. \text{case}(\%x. x, \%y. y, z)) : \text{bij}(A+0, A)$

(*proof*)

lemma *ordertype-sum-0-eq*:

$\text{well-ord}(A,r) \implies \text{ordertype}(A+0, \text{radd}(A,r,0,s)) = \text{ordertype}(A,r)$
(*proof*)

lemma *bij-0-sum*: $(\text{lam } z:0+A. \text{ case } (\%x. x, \%y. y, z)) : \text{bij}(0+A, A)$
 $\langle \text{proof} \rangle$

lemma *ordertype-0-sum-eq*:
 $\text{well-ord}(A,r) \implies \text{ordertype}(0+A, \text{radd}(0,s,A,r)) = \text{ordertype}(A,r)$
 $\langle \text{proof} \rangle$

Initial segments of *radd*. Statements by Grabczewski

lemma *pred-Inl-bij*:
 $a:A \implies (\text{lam } x:\text{pred}(A,a,r). \text{Inl}(x))$
 $: \text{bij}(\text{pred}(A,a,r), \text{pred}(A+B, \text{Inl}(a), \text{radd}(A,r,B,s)))$
 $\langle \text{proof} \rangle$

lemma *ordertype-pred-Inl-eq*:
 $[[a:A; \text{well-ord}(A,r)]]$
 $\implies \text{ordertype}(\text{pred}(A+B, \text{Inl}(a), \text{radd}(A,r,B,s)), \text{radd}(A,r,B,s)) =$
 $\text{ordertype}(\text{pred}(A,a,r), r)$
 $\langle \text{proof} \rangle$

lemma *pred-Inr-bij*:
 $b:B \implies$
 $\text{id}(A+\text{pred}(B,b,s))$
 $: \text{bij}(A+\text{pred}(B,b,s), \text{pred}(A+B, \text{Inr}(b), \text{radd}(A,r,B,s)))$
 $\langle \text{proof} \rangle$

lemma *ordertype-pred-Inr-eq*:
 $[[b:B; \text{well-ord}(A,r); \text{well-ord}(B,s)]]$
 $\implies \text{ordertype}(\text{pred}(A+B, \text{Inr}(b), \text{radd}(A,r,B,s)), \text{radd}(A,r,B,s)) =$
 $\text{ordertype}(A+\text{pred}(B,b,s), \text{radd}(A,r,\text{pred}(B,b,s),s))$
 $\langle \text{proof} \rangle$

20.4.2 ordify: trivial coercion to an ordinal

lemma *Ord-ordify* [*iff*, *TC*]: $\text{Ord}(\text{ordify}(x))$
 $\langle \text{proof} \rangle$

lemma *ordify-idem* [*simp*]: $\text{ordify}(\text{ordify}(x)) = \text{ordify}(x)$
 $\langle \text{proof} \rangle$

20.4.3 Basic laws for ordinal addition

lemma *Ord-raw-oadd*: $[[\text{Ord}(i); \text{Ord}(j)]]$ $\implies \text{Ord}(\text{raw-oadd}(i,j))$
 $\langle \text{proof} \rangle$

lemma *Ord-oadd* [*iff*, *TC*]: $\text{Ord}(i++j)$
 $\langle \text{proof} \rangle$

Ordinal addition with zero

lemma *raw-oadd-0*: $Ord(i) \implies raw-oadd(i,0) = i$
 ⟨proof⟩

lemma *oadd-0 [simp]*: $Ord(i) \implies i++0 = i$
 ⟨proof⟩

lemma *raw-oadd-0-left*: $Ord(i) \implies raw-oadd(0,i) = i$
 ⟨proof⟩

lemma *oadd-0-left [simp]*: $Ord(i) \implies 0++i = i$
 ⟨proof⟩

lemma *oadd-eq-if-raw-oadd*:
 $i++j = (if\ Ord(i)\ then\ (if\ Ord(j)\ then\ raw-oadd(i,j)\ else\ i)$
 $\quad\quad\quad else\ (if\ Ord(j)\ then\ j\ else\ 0))$
 ⟨proof⟩

lemma *raw-oadd-eq-oadd*: $[[Ord(i); Ord(j)]] \implies raw-oadd(i,j) = i++j$
 ⟨proof⟩

lemma *lt-oadd1*: $k < i \implies k < i++j$
 ⟨proof⟩

lemma *oadd-le-self*: $Ord(i) \implies i\ le\ i++j$
 ⟨proof⟩

Various other results

lemma *id-ord-iso-Memrel*: $A \leq B \implies id(A) : ord-iso(A, Memrel(A), A, Memrel(B))$
 ⟨proof⟩

lemma *subset-ord-iso-Memrel*:
 $[[f : ord-iso(A, Memrel(B), C, r); A \leq B]] \implies f : ord-iso(A, Memrel(A), C, r)$
 ⟨proof⟩

lemma *restrict-ord-iso*:
 $[[f \in ord-iso(i, Memrel(i), Order.pred(A,a,r), r); a \in A; j < i;$
 $\quad\quad\quad trans[A](r)]]$
 $\implies restrict(f,j) \in ord-iso(j, Memrel(j), Order.pred(A,f*j,r), r)$
 ⟨proof⟩

lemma *restrict-ord-iso2*:
 $[[f \in ord-iso(Order.pred(A,a,r), r, i, Memrel(i)); a \in A;$
 $\quad\quad\quad j < i; trans[A](r)]]$

$==> \text{converse}(\text{restrict}(\text{converse}(f), j))$
 $\in \text{ord-iso}(\text{Order.pred}(A, \text{converse}(f)'j, r), r, j, \text{Memrel}(j))$
 <proof>

lemma *ordertype-sum-Memrel*:
 $[[\text{well-ord}(A, r); k < j]]$
 $==> \text{ordertype}(A+k, \text{radd}(A, r, k, \text{Memrel}(j))) =$
 $\text{ordertype}(A+k, \text{radd}(A, r, k, \text{Memrel}(k)))$
 <proof>

lemma *oadd-lt-mono2*: $k < j ==> i++k < i++j$
 <proof>

lemma *oadd-lt-cancel2*: $[[i++j < i++k; \text{Ord}(j)]]$ $==> j < k$
 <proof>

lemma *oadd-lt-iff2*: $\text{Ord}(j) ==> i++j < i++k <-> j < k$
 <proof>

lemma *oadd-inject*: $[[i++j = i++k; \text{Ord}(j); \text{Ord}(k)]]$ $==> j = k$
 <proof>

lemma *lt-oadd-disj*: $k < i++j ==> k < i \mid (\exists x. k = i++x)$
 <proof>

20.4.4 Ordinal addition with successor – via associativity!

lemma *oadd-assoc*: $(i++j)++k = i++(j++k)$
 <proof>

lemma *oadd-unfold*: $[[\text{Ord}(i); \text{Ord}(j)]]$ $==> i++j = i \text{ Un } (\bigcup_{k \in j. \{i++k\})$
 <proof>

lemma *oadd-1*: $\text{Ord}(i) ==> i++1 = \text{succ}(i)$
 <proof>

lemma *oadd-succ* [*simp*]: $\text{Ord}(j) ==> i++\text{succ}(j) = \text{succ}(i++j)$
 <proof>

Ordinal addition with limit ordinals

lemma *oadd-UN*:
 $[[!!x. x:A ==> \text{Ord}(j(x)); a:A]]$
 $==> i++(\bigcup_{x \in A. j(x)}) = (\bigcup_{x \in A. i++j(x)})$
 <proof>

lemma *oadd-Limit*: $\text{Limit}(j) ==> i++j = (\bigcup_{k \in j. i++k)$
 <proof>

lemma *oadd-eq-0-iff*: $[[\text{Ord}(i); \text{Ord}(j)]]$ $==> (i++j) = 0 <-> i=0 \ \& \ j=0$

<proof>

lemma *oadd-eq-lt-iff*: $[[\text{Ord}(i); \text{Ord}(j)]] \implies 0 < (i ++ j) \iff 0 < i \mid 0 < j$
<proof>

lemma *oadd-LimitI*: $[[\text{Ord}(i); \text{Limit}(j)]] \implies \text{Limit}(i ++ j)$
<proof>

Order/monotonicity properties of ordinal addition

lemma *oadd-le-self2*: $\text{Ord}(i) \implies i \text{ le } j ++ i$
<proof>

lemma *oadd-le-mono1*: $k \text{ le } j \implies k ++ i \text{ le } j ++ i$
<proof>

lemma *oadd-lt-mono*: $[[i' \text{ le } i; j' < j]] \implies i' ++ j' < i ++ j$
<proof>

lemma *oadd-le-mono*: $[[i' \text{ le } i; j' \text{ le } j]] \implies i' ++ j' \text{ le } i ++ j$
<proof>

lemma *oadd-le-iff2*: $[[\text{Ord}(j); \text{Ord}(k)]] \implies i ++ j \text{ le } i ++ k \iff j \text{ le } k$
<proof>

lemma *oadd-lt-self*: $[[\text{Ord}(i); 0 < j]] \implies i < i ++ j$
<proof>

Every ordinal is exceeded by some limit ordinal.

lemma *Ord-imp-greater-Limit*: $\text{Ord}(i) \implies \exists k. i < k \ \& \ \text{Limit}(k)$
<proof>

lemma *Ord2-imp-greater-Limit*: $[[\text{Ord}(i); \text{Ord}(j)]] \implies \exists k. i < k \ \& \ j < k \ \& \ \text{Limit}(k)$
<proof>

20.5 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*:
 $A \leq B \implies (\text{lam } y:B. \text{if}(y:A, \text{Inl}(y), \text{Inr}(y))) : \text{bij}(B, A + (B - A))$
<proof>

lemma *ordertype-sum-Diff*:
 $i \text{ le } j \implies$
 $\text{ordertype}(i + (j - i), \text{radd}(i, \text{Memrel}(j), j - i, \text{Memrel}(j))) =$
 $\text{ordertype}(j, \text{Memrel}(j))$
<proof>

lemma *Ord-odiff* [*simp,TC*]:

$$\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \text{Ord}(i--j)$$
 $\langle \text{proof} \rangle$

lemma *raw-oadd-ordertype-Diff*:

$$i \text{ le } j \implies \text{raw-oadd}(i,j--i) = \text{ordertype}(i+(j-i), \text{radd}(i,\text{Memrel}(j),j-i,\text{Memrel}(j)))$$
 $\langle \text{proof} \rangle$

lemma *oadd-odiff-inverse*: $i \text{ le } j \implies i ++ (j--i) = j$
 $\langle \text{proof} \rangle$

lemma *odiff-oadd-inverse*: $\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies (i++j) -- i = j$
 $\langle \text{proof} \rangle$

lemma *odiff-lt-mono2*: $\llbracket i < j; k \text{ le } i \rrbracket \implies i--k < j--k$
 $\langle \text{proof} \rangle$

20.6 Ordinal Multiplication

lemma *Ord-omult* [*simp,TC*]:

$$\llbracket \text{Ord}(i); \text{Ord}(j) \rrbracket \implies \text{Ord}(i**j)$$
 $\langle \text{proof} \rangle$

20.6.1 A useful unfolding law

lemma *pred-Pair-eq*:

$$\llbracket a:A; b:B \rrbracket \implies \text{pred}(A*B, \langle a,b \rangle, \text{rmult}(A,r,B,s)) = \text{pred}(A,a,r)*B \text{ Un } (\{a\} * \text{pred}(B,b,s))$$
 $\langle \text{proof} \rangle$

lemma *ordertype-pred-Pair-eq*:

$$\llbracket a:A; b:B; \text{well-ord}(A,r); \text{well-ord}(B,s) \rrbracket \implies \text{ordertype}(\text{pred}(A*B, \langle a,b \rangle, \text{rmult}(A,r,B,s)), \text{rmult}(A,r,B,s)) = \text{ordertype}(\text{pred}(A,a,r)*B + \text{pred}(B,b,s), \text{radd}(A*B, \text{rmult}(A,r,B,s), B, s))$$
 $\langle \text{proof} \rangle$

lemma *ordertype-pred-Pair-lemma*:

$$\llbracket i' < i; j' < j \rrbracket \implies \text{ordertype}(\text{pred}(i*j, \langle i',j' \rangle, \text{rmult}(i,\text{Memrel}(i),j,\text{Memrel}(j))), \text{rmult}(i,\text{Memrel}(i),j,\text{Memrel}(j))) = \text{raw-oadd}(j**i', j')$$
 $\langle \text{proof} \rangle$

lemma *lt-omult*:

$$\llbracket \text{Ord}(i); \text{Ord}(j); k < j**i \rrbracket \implies \text{EX } j' i'. k = j**i' ++ j' \ \& \ j' < j \ \& \ i' < i$$

$\langle proof \rangle$

lemma *omult-oadd-lt*:

$\llbracket j' < j; i' < i \rrbracket \implies j ** i' ++ j' < j ** i$
 $\langle proof \rangle$

lemma *omult-unfold*:

$\llbracket Ord(i); Ord(j) \rrbracket \implies j ** i = (\bigcup j' \in j. \bigcup i' \in i. \{j ** i' ++ j'\})$
 $\langle proof \rangle$

20.6.2 Basic laws for ordinal multiplication

Ordinal multiplication by zero

lemma *omult-0* [*simp*]: $i ** 0 = 0$

$\langle proof \rangle$

lemma *omult-0-left* [*simp*]: $0 ** i = 0$

$\langle proof \rangle$

Ordinal multiplication by 1

lemma *omult-1* [*simp*]: $Ord(i) \implies i ** 1 = i$

$\langle proof \rangle$

lemma *omult-1-left* [*simp*]: $Ord(i) \implies 1 ** i = i$

$\langle proof \rangle$

Distributive law for ordinal multiplication and addition

lemma *oadd-omult-distrib*:

$\llbracket Ord(i); Ord(j); Ord(k) \rrbracket \implies i ** (j ++ k) = (i ** j) ++ (i ** k)$
 $\langle proof \rangle$

lemma *omult-succ*: $\llbracket Ord(i); Ord(j) \rrbracket \implies i ** succ(j) = (i ** j) ++ i$

$\langle proof \rangle$

Associative law

lemma *omult-assoc*:

$\llbracket Ord(i); Ord(j); Ord(k) \rrbracket \implies (i ** j) ** k = i ** (j ** k)$
 $\langle proof \rangle$

Ordinal multiplication with limit ordinals

lemma *omult-UN*:

$\llbracket Ord(i); \forall x. x:A \implies Ord(j(x)) \rrbracket$
 $\implies i ** (\bigcup x \in A. j(x)) = (\bigcup x \in A. i ** j(x))$
 $\langle proof \rangle$

lemma *omult-Limit*: $\llbracket Ord(i); Limit(j) \rrbracket \implies i ** j = (\bigcup k \in j. i ** k)$

$\langle proof \rangle$

20.6.3 Ordering/monotonicity properties of ordinal multiplication

lemma *lt-omult1*: $[[k < i; 0 < j]] ==> k < i**j$
(proof)

lemma *omult-le-self*: $[[Ord(i); 0 < j]] ==> i le i**j$
(proof)

lemma *omult-le-mono1*: $[[k le j; Ord(i)]] ==> k**i le j**i$
(proof)

lemma *omult-lt-mono2*: $[[k < j; 0 < i]] ==> i**k < i**j$
(proof)

lemma *omult-le-mono2*: $[[k le j; Ord(i)]] ==> i**k le i**j$
(proof)

lemma *omult-le-mono*: $[[i' le i; j' le j]] ==> i'**j' le i**j$
(proof)

lemma *omult-lt-mono*: $[[i' le i; j' < j; 0 < i]] ==> i'**j' < i**j$
(proof)

lemma *omult-le-self2*: $[[Ord(i); 0 < j]] ==> i le j**i$
(proof)

Further properties of ordinal multiplication

lemma *omult-inject*: $[[i**j = i**k; 0 < i; Ord(j); Ord(k)]] ==> j=k$
(proof)

20.7 The Relation *Lt*

lemma *wf-Lt*: *wf(Lt)*
(proof)

lemma *irrefl-Lt*: *irrefl(A,Lt)*
(proof)

lemma *trans-Lt*: *trans[A](Lt)*
(proof)

lemma *part-ord-Lt*: *part-ord(A,Lt)*
(proof)

lemma *linear-Lt*: *linear(nat,Lt)*
(proof)

lemma *tot-ord-Lt*: *tot-ord(nat,Lt)*
(proof)

lemma *well-ord-Lt*: *well-ord*(*nat*,*Lt*)

<proof>

end

21 Finite: Finite Powerset Operator and Finite Function Space

theory *Finite* **imports** *Inductive-ZF Epsilon Nat-ZF* **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=>i*

FiniteFun :: [*i,i*]==>*i* ((- -||>/ -) [61, 60] 60)

inductive

domains *Fin*(*A*) <= *Pow*(*A*)

intros

emptyI: *0* : *Fin*(*A*)

consI: [[*a*: *A*; *b*: *Fin*(*A*)]] ==> *cons*(*a*,*b*) : *Fin*(*A*)

type-intros *empty-subsetI cons-subsetI PowI*

type-elims *PowD [THEN revcut-rl]*

inductive

domains *FiniteFun*(*A*,*B*) <= *Fin*(*A*B*)

intros

emptyI: *0* : *A* -||> *B*

consI: [[*a*: *A*; *b*: *B*; *h*: *A* -||> *B*; *a* ~: *domain*(*h*)]]

==> *cons*(<*a*,*b*>, *h*) : *A* -||> *B*

type-intros *Fin.intros*

21.1 Finite Powerset Operator

lemma *Fin-mono*: *A*<=*B* ==> *Fin*(*A*) <= *Fin*(*B*)

<proof>

lemmas *FinD* = *Fin.dom-subset [THEN subsetD, THEN PowD, standard]*

lemma *Fin-induct* [*case-names 0 cons, induct set: Fin*]:
 [| *b*: *Fin*(*A*);
 P(0);
 !!*x y*. [| *x*: *A*; *y*: *Fin*(*A*); *x*~:*y*; *P*(*y*) |] ==> *P*(*cons*(*x*,*y*))
 |] ==> *P*(*b*)
 <*proof*>

declare *Fin.intros* [*simp*]

lemma *Fin-0*: *Fin*(0) = {0}
 <*proof*>

lemma *Fin-UnI* [*simp*]: [| *b*: *Fin*(*A*); *c*: *Fin*(*A*) |] ==> *b Un c* : *Fin*(*A*)
 <*proof*>

lemma *Fin-UnionI*: *C* : *Fin*(*Fin*(*A*)) ==> *Union*(*C*) : *Fin*(*A*)
 <*proof*>

lemma *Fin-subset-lemma* [*rule-format*]: *b*: *Fin*(*A*) ==> $\forall z. z \leq b \rightarrow z$: *Fin*(*A*)
 <*proof*>

lemma *Fin-subset*: [| *c* ≤ *b*; *b*: *Fin*(*A*) |] ==> *c*: *Fin*(*A*)
 <*proof*>

lemma *Fin-IntI1* [*intro, simp*]: *b*: *Fin*(*A*) ==> *b Int c* : *Fin*(*A*)
 <*proof*>

lemma *Fin-IntI2* [*intro, simp*]: *c*: *Fin*(*A*) ==> *b Int c* : *Fin*(*A*)
 <*proof*>

lemma *Fin-0-induct-lemma* [*rule-format*]:
 [| *c*: *Fin*(*A*); *b*: *Fin*(*A*); *P*(*b*);
 !!*x y*. [| *x*: *A*; *y*: *Fin*(*A*); *x*:*y*; *P*(*y*) |] ==> *P*(*y*−{*x*})
 |] ==> *c* ≤ *b* → *P*(*b*−*c*)
 <*proof*>

lemma *Fin-0-induct*:
 [| *b*: *Fin*(*A*);
 P(*b*);
 !!*x y*. [| *x*: *A*; *y*: *Fin*(*A*); *x*:*y*; *P*(*y*) |] ==> *P*(*y*−{*x*})
 |]

$[] ==> P(0)$
 $\langle proof \rangle$

lemma *nat-fun-subset-Fin*: $n: nat ==> n \rightarrow A \leq Fin(nat * A)$
 $\langle proof \rangle$

21.2 Finite Function Space

lemma *FiniteFun-mono*:

$[] A \leq C; B \leq D [] ==> A \multimap B \leq C \multimap D$
 $\langle proof \rangle$

lemma *FiniteFun-mono1*: $A \leq B ==> A \multimap A \leq B \multimap B$
 $\langle proof \rangle$

lemma *FiniteFun-is-fun*: $h: A \multimap B ==> h: domain(h) \rightarrow B$
 $\langle proof \rangle$

lemma *FiniteFun-domain-Fin*: $h: A \multimap B ==> domain(h) : Fin(A)$
 $\langle proof \rangle$

lemmas *FiniteFun-apply-type = FiniteFun-is-fun [THEN apply-type, standard]*

lemma *FiniteFun-subset-lemma [rule-format]*:

$b: A \multimap B ==> ALL z. z \leq b \multimap z: A \multimap B$
 $\langle proof \rangle$

lemma *FiniteFun-subset*: $[] c \leq b; b: A \multimap B [] ==> c: A \multimap B$
 $\langle proof \rangle$

lemma *fun-FiniteFunI [rule-format]*: $A: Fin(X) ==> ALL f. f: A \rightarrow B \multimap f: A \multimap B$
 $\langle proof \rangle$

lemma *lam-FiniteFun*: $A: Fin(X) ==> (lam x:A. b(x)) : A \multimap \{b(x). x:A\}$
 $\langle proof \rangle$

lemma *FiniteFun-Collect-iff*:

$f : FiniteFun(A, \{y:B. P(y)\})$
 $\langle - \rangle f : FiniteFun(A,B) \ \& \ (ALL x:domain(f). P(f'x))$
 $\langle proof \rangle$

21.3 The Contents of a Singleton Set

definition

$contents :: i ==> i$ **where**
 $contents(X) == THE x. X = \{x\}$

lemma *contents-eq* [*simp*]: *contents* ($\{x\}$) = x
(*proof*)

end

22 Cardinal: Cardinal Numbers Without the Axiom of Choice

theory *Cardinal* **imports** *OrderType Finite Nat-ZF Sum* **begin**

definition

Least :: ($i \Rightarrow o$) \Rightarrow i (**binder** *LEAST* 10) **where**
Least(P) == *THE* i . *Ord*(i) & $P(i)$ & (*ALL* j . $j < i \rightarrow \sim P(j)$)

definition

eqpoll :: $[i, i] \Rightarrow o$ (**infixl** *eqpoll* 50) **where**
 A *eqpoll* B == *EX* f . f : *bij*(A, B)

definition

lepoll :: $[i, i] \Rightarrow o$ (**infixl** *lepoll* 50) **where**
 A *lepoll* B == *EX* f . f : *inj*(A, B)

definition

lesspoll :: $[i, i] \Rightarrow o$ (**infixl** *lesspoll* 50) **where**
 A *lesspoll* B == A *lepoll* B & $\sim(A$ *eqpoll* $B)$

definition

cardinal :: $i \Rightarrow i$ (**|**-) **where**
 $|A|$ == *LEAST* i . i *eqpoll* A

definition

Finite :: $i \Rightarrow o$ **where**
Finite(A) == *EX* n :*nat*. A *eqpoll* n

definition

Card :: $i \Rightarrow o$ **where**
Card(i) == ($i = |i|$)

notation (*xsymbols*)

eqpoll (**infixl** \approx 50) **and**
lepoll (**infixl** \lesssim 50) **and**
lesspoll (**infixl** \prec 50) **and**
Least (**binder** μ 10)

notation (*HTML output*)

eqpoll (infixl ≈ 50) and
Least (binder $\mu 10$)

22.1 The Schroeder-Bernstein Theorem

See Davey and Priestly, page 106

lemma *decomp-bnd-mono*: $bnd\text{-}mono(X, \%W. X - g^{''}(Y - f^{''}W))$
 ⟨proof⟩

lemma *Banach-last-equation*:

$g: Y \rightarrow X$
 $\implies g^{''}(Y - f^{''} lfp(X, \%W. X - g^{''}(Y - f^{''}W))) =$
 $X - lfp(X, \%W. X - g^{''}(Y - f^{''}W))$
 ⟨proof⟩

lemma *decomposition*:

$[f: X \rightarrow Y; g: Y \rightarrow X] \implies$
 $EX\ XA\ XB\ YA\ YB. (XA\ Int\ XB = 0) \ \& \ (XA\ Un\ XB = X) \ \&$
 $(YA\ Int\ YB = 0) \ \& \ (YA\ Un\ YB = Y) \ \&$
 $f^{''}XA = YA \ \& \ g^{''}YB = XB$
 ⟨proof⟩

lemma *schroeder-bernstein*:

$[f: inj(X, Y); g: inj(Y, X)] \implies EX\ h. h: bij(X, Y)$
 ⟨proof⟩

lemma *bij-imp-epoll*: $f: bij(A, B) \implies A \approx B$
 ⟨proof⟩

lemmas *epoll-refl* = *id-bij* [THEN *bij-imp-epoll*, *standard*, *simp*]

lemma *epoll-sym*: $X \approx Y \implies Y \approx X$
 ⟨proof⟩

lemma *epoll-trans*:

$[X \approx Y; Y \approx Z] \implies X \approx Z$
 ⟨proof⟩

lemma *subset-imp-lepoll*: $X \leq Y \implies X \lesssim Y$
 ⟨proof⟩

lemmas *lepoll-refl* = *subset-refl* [THEN *subset-imp-lepoll*, *standard*, *simp*]

lemmas *le-imp-lepoll* = *le-imp-subset* [THEN *subset-imp-lepoll*, *standard*]

lemma *eqpoll-imp-lepoll*: $X \approx Y \implies X \lesssim Y$
 ⟨*proof*⟩

lemma *lepoll-trans*: $[X \lesssim Y; Y \lesssim Z] \implies X \lesssim Z$
 ⟨*proof*⟩

lemma *eqpollI*: $[X \lesssim Y; Y \lesssim X] \implies X \approx Y$
 ⟨*proof*⟩

lemma *eqpollE*:
 $[X \approx Y; [X \lesssim Y; Y \lesssim X] \implies P] \implies P$
 ⟨*proof*⟩

lemma *eqpoll-iff*: $X \approx Y \iff X \lesssim Y \ \& \ Y \lesssim X$
 ⟨*proof*⟩

lemma *lepoll-0-is-0*: $A \lesssim 0 \implies A = 0$
 ⟨*proof*⟩

lemmas *empty-lepollI* = *empty-subsetI* [THEN *subset-imp-lepoll*, *standard*]

lemma *lepoll-0-iff*: $A \lesssim 0 \iff A = 0$
 ⟨*proof*⟩

lemma *Un-lepoll-Un*:
 $[A \lesssim B; C \lesssim D; B \text{ Int } D = 0] \implies A \text{ Un } C \lesssim B \text{ Un } D$
 ⟨*proof*⟩

lemmas *eqpoll-0-is-0* = *eqpoll-imp-lepoll* [THEN *lepoll-0-is-0*, *standard*]

lemma *eqpoll-0-iff*: $A \approx 0 \iff A = 0$
 ⟨*proof*⟩

lemma *eqpoll-disjoint-Un*:
 $[A \approx B; C \approx D; A \text{ Int } C = 0; B \text{ Int } D = 0] \implies A \text{ Un } C \approx B \text{ Un } D$
 ⟨*proof*⟩

22.2 lesspoll: contributions by Krzysztof Grabczewski

lemma *lesspoll-not-refl*: $\sim (i \prec i)$
 ⟨*proof*⟩

lemma *lesspoll-irrefl* [*elim!*]: $i \prec i \implies P$

<proof>

lemma *lesspoll-imp-lepoll*: $A \prec B \implies A \lesssim B$
<proof>

lemma *lepoll-well-ord*: $[[A \lesssim B; \text{well-ord}(B,r)]] \implies \text{EX } s. \text{well-ord}(A,s)$
<proof>

lemma *lepoll-iff-leqpoll*: $A \lesssim B \iff A \prec B \mid A \approx B$
<proof>

lemma *inj-not-surj-succ*:
 $[[f : \text{inj}(A, \text{succ}(m)); f \sim : \text{surj}(A, \text{succ}(m))]] \implies \text{EX } f. f : \text{inj}(A,m)$
<proof>

lemma *lesspoll-trans*:
 $[[X \prec Y; Y \prec Z]] \implies X \prec Z$
<proof>

lemma *lesspoll-trans1*:
 $[[X \lesssim Y; Y \prec Z]] \implies X \prec Z$
<proof>

lemma *lesspoll-trans2*:
 $[[X \prec Y; Y \lesssim Z]] \implies X \prec Z$
<proof>

lemma *Least-equality*:
 $[[P(i); \text{Ord}(i); \forall x. x < i \implies \sim P(x)]] \implies (\text{LEAST } x. P(x)) = i$
<proof>

lemma *LeastI*: $[[P(i); \text{Ord}(i)]] \implies P(\text{LEAST } x. P(x))$
<proof>

lemma *Least-le*: $[[P(i); \text{Ord}(i)]] \implies (\text{LEAST } x. P(x)) \text{ le } i$
<proof>

lemma *less-LeastE*: $[[P(i); i < (\text{LEAST } x. P(x))]] \implies Q$
<proof>

lemma *LeastI2*:

$\llbracket P(i); \text{Ord}(i); \forall j. P(j) \implies Q(j) \rrbracket \implies Q(\text{LEAST } j. P(j))$
 <proof>

lemma *Least-0*:

$\llbracket \sim (EX i. \text{Ord}(i) \ \& \ P(i)) \rrbracket \implies (\text{LEAST } x. P(x)) = 0$
 <proof>

lemma *Ord-Least* [*intro,simp,TC*]: $\text{Ord}(\text{LEAST } x. P(x))$

<proof>

lemma *Least-cong*:

$(\forall y. P(y) \iff Q(y)) \implies (\text{LEAST } x. P(x)) = (\text{LEAST } x. Q(x))$
 <proof>

lemma *cardinal-cong*: $X \approx Y \implies |X| = |Y|$

<proof>

lemma *well-ord-cardinal-epoll*:

$\text{well-ord}(A,r) \implies |A| \approx A$
 <proof>

lemmas *Ord-cardinal-epoll = well-ord-Memrel* [*THEN well-ord-cardinal-epoll*]

lemma *well-ord-cardinal-epE*:

$\llbracket \text{well-ord}(X,r); \text{well-ord}(Y,s); |X| = |Y| \rrbracket \implies X \approx Y$
 <proof>

lemma *well-ord-cardinal-epoll-iff*:

$\llbracket \text{well-ord}(X,r); \text{well-ord}(Y,s) \rrbracket \implies |X| = |Y| \iff X \approx Y$
 <proof>

lemma *Ord-cardinal-le*: $\text{Ord}(i) \implies |i| \text{ le } i$

<proof>

lemma *Card-cardinal-eq*: $\text{Card}(K) \implies |K| = K$

<proof>

lemma *CardI*: $[| \text{Ord}(i); \forall j. j < i \implies \sim(j \approx i) |] \implies \text{Card}(i)$
(proof)

lemma *Card-is-Ord*: $\text{Card}(i) \implies \text{Ord}(i)$
(proof)

lemma *Card-cardinal-le*: $\text{Card}(K) \implies K \text{ le } |K|$
(proof)

lemma *Ord-cardinal [simp,intro!]*: $\text{Ord}(|A|)$
(proof)

lemma *Card-iff-initial*: $\text{Card}(K) \iff \text{Ord}(K) \ \& \ (\text{ALL } j. j < K \iff \sim j \approx K)$
(proof)

lemma *lt-Card-imp-lesspoll*: $[| \text{Card}(a); i < a |] \implies i \prec a$
(proof)

lemma *Card-0*: $\text{Card}(0)$
(proof)

lemma *Card-Un*: $[| \text{Card}(K); \text{Card}(L) |] \implies \text{Card}(K \text{ Un } L)$
(proof)

lemma *Card-cardinal*: $\text{Card}(|A|)$
(proof)

lemma *cardinal-eq-lemma*: $[| |i| \text{ le } j; j \text{ le } i |] \implies |j| = |i|$
(proof)

lemma *cardinal-mono*: $i \text{ le } j \implies |i| \text{ le } |j|$
(proof)

lemma *cardinal-lt-imp-lt*: $[| |i| < |j|; \text{Ord}(i); \text{Ord}(j) |] \implies i < j$
(proof)

lemma *Card-lt-imp-lt*: $[| |i| < K; \text{Ord}(i); \text{Card}(K) |] \implies i < K$
(proof)

lemma *Card-lt-iff*: $[| \text{Ord}(i); \text{Card}(K) |] \implies (|i| < K) \iff (i < K)$
(proof)

lemma *Card-le-iff*: $[| \text{Ord}(i); \text{Card}(K) |] \implies (K \text{ le } |i|) \iff (K \text{ le } i)$
(proof)

lemma *well-ord-lepoll-imp-Card-le*:

$\llbracket \text{well-ord}(B,r); A \lesssim B \rrbracket \implies |A| \text{ le } |B|$
 $\langle \text{proof} \rangle$

lemma *lepoll-cardinal-le*: $\llbracket A \lesssim i; \text{Ord}(i) \rrbracket \implies |A| \text{ le } i$
 $\langle \text{proof} \rangle$

lemma *lepoll-Ord-imp-epoll*: $\llbracket A \lesssim i; \text{Ord}(i) \rrbracket \implies |A| \approx A$
 $\langle \text{proof} \rangle$

lemma *lesspoll-imp-epoll*: $\llbracket A \prec i; \text{Ord}(i) \rrbracket \implies |A| \approx A$
 $\langle \text{proof} \rangle$

lemma *cardinal-subset-Ord*: $\llbracket |A| \leq i; \text{Ord}(i) \rrbracket \implies |A| \leq i$
 $\langle \text{proof} \rangle$

22.3 The finite cardinals

lemma *cons-lepoll-consD*:

$\llbracket \text{cons}(u,A) \lesssim \text{cons}(v,B); u \sim A; v \sim B \rrbracket \implies \text{cons}(u,A) \lesssim \text{cons}(v,B)$
 $\langle \text{proof} \rangle$

lemma *cons-epoll-consD*: $\llbracket \text{cons}(u,A) \approx \text{cons}(v,B); u \sim A; v \sim B \rrbracket \implies \text{cons}(u,A) \approx \text{cons}(v,B)$
 $\langle \text{proof} \rangle$

lemma *succ-lepoll-succD*: $\text{succ}(m) \lesssim \text{succ}(n) \implies m \lesssim n$
 $\langle \text{proof} \rangle$

lemma *nat-lepoll-imp-le* [*rule-format*]:

$m:\text{nat} \implies \text{ALL } n:\text{nat}. m \lesssim n \dashrightarrow m \text{ le } n$
 $\langle \text{proof} \rangle$

lemma *nat-epoll-iff*: $\llbracket m:\text{nat}; n:\text{nat} \rrbracket \implies m \approx n \leftrightarrow m = n$
 $\langle \text{proof} \rangle$

lemma *nat-into-Card*:

$n:\text{nat} \implies \text{Card}(n)$
 $\langle \text{proof} \rangle$

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*: $[[\text{succ}(n) \lesssim n; n:\text{nat}]] \implies P$
 ⟨proof⟩

lemma *n-lesspoll-nat*: $n \in \text{nat} \implies n \prec \text{nat}$
 ⟨proof⟩

lemma *nat-lepoll-imp-ex-epoll-n*:
 $[[n \in \text{nat}; \text{nat} \lesssim X]] \implies \exists Y. Y \subseteq X \ \& \ n \approx Y$
 ⟨proof⟩

lemma *lepoll-imp-lesspoll-succ*:
 $[[A \lesssim m; m:\text{nat}]] \implies A \prec \text{succ}(m)$
 ⟨proof⟩

lemma *lesspoll-succ-imp-lepoll*:
 $[[A \prec \text{succ}(m); m:\text{nat}]] \implies A \lesssim m$
 ⟨proof⟩

lemma *lesspoll-succ-iff*: $m:\text{nat} \implies A \prec \text{succ}(m) \iff A \lesssim m$
 ⟨proof⟩

lemma *lepoll-succ-disj*: $[[A \lesssim \text{succ}(m); m:\text{nat}]] \implies A \lesssim m \mid A \approx \text{succ}(m)$
 ⟨proof⟩

lemma *lesspoll-cardinal-lt*: $[[A \prec i; \text{Ord}(i)]] \implies |A| < i$
 ⟨proof⟩

22.4 The first infinite cardinal: Omega, or nat

lemma *lt-not-lepoll*: $[[n < i; n:\text{nat}]] \implies \sim i \lesssim n$
 ⟨proof⟩

lemma *Ord-nat-epoll-iff*: $[[\text{Ord}(i); n:\text{nat}]] \implies i \approx n \iff i = n$
 ⟨proof⟩

lemma *Card-nat*: $\text{Card}(\text{nat})$
 ⟨proof⟩

lemma *nat-le-cardinal*: $\text{nat} \text{ le } i \implies \text{nat} \text{ le } |i|$
 ⟨proof⟩

22.5 Towards Cardinal Arithmetic

lemma *cons-lepoll-cong*:
 $[[A \lesssim B; b \sim: B]] \implies \text{cons}(a,A) \lesssim \text{cons}(b,B)$
 ⟨proof⟩

lemma *cons-epoll-cong*:

$\llbracket A \approx B; a \sim: A; b \sim: B \rrbracket \implies \text{cons}(a,A) \approx \text{cons}(b,B)$
<proof>

lemma *cons-lepoll-cons-iff*:

$\llbracket a \sim: A; b \sim: B \rrbracket \implies \text{cons}(a,A) \lesssim \text{cons}(b,B) \iff A \lesssim B$
<proof>

lemma *cons-epoll-cons-iff*:

$\llbracket a \sim: A; b \sim: B \rrbracket \implies \text{cons}(a,A) \approx \text{cons}(b,B) \iff A \approx B$
<proof>

lemma *singleton-epoll-1*: $\{a\} \approx 1$

<proof>

lemma *cardinal-singleton*: $|\{a\}| = 1$

<proof>

lemma *not-0-is-lepoll-1*: $A \sim = 0 \implies 1 \lesssim A$

<proof>

lemma *succ-epoll-cong*: $A \approx B \implies \text{succ}(A) \approx \text{succ}(B)$

<proof>

lemma *sum-epoll-cong*: $\llbracket A \approx C; B \approx D \rrbracket \implies A+B \approx C+D$

<proof>

lemma *prod-epoll-cong*:

$\llbracket A \approx C; B \approx D \rrbracket \implies A*B \approx C*D$
<proof>

lemma *inj-disjoint-epoll*:

$\llbracket f: \text{inj}(A,B); A \text{ Int } B = 0 \rrbracket \implies A \text{ Un } (B - \text{range}(f)) \approx B$
<proof>

22.6 Lemmas by Krzysztof Grabczewski

lemma *Diff-sing-lepoll*:

$\llbracket a:A; A \lesssim \text{succ}(n) \rrbracket \implies A - \{a\} \lesssim n$
<proof>

lemma *lepoll-Diff-sing*:

$\llbracket \text{succ}(n) \lesssim A \rrbracket \implies n \lesssim A - \{a\}$
<proof>

lemma *Diff-sing-eqpoll*: $[[a:A; A \approx \text{succ}(n)]] \implies A - \{a\} \approx n$
 ⟨proof⟩

lemma *lepoll-1-is-sing*: $[[A \lesssim 1; a:A]] \implies A = \{a\}$
 ⟨proof⟩

lemma *Un-lepoll-sum*: $A \text{ Un } B \lesssim A+B$
 ⟨proof⟩

lemma *well-ord-Un*:
 $[[\text{well-ord}(X,R); \text{well-ord}(Y,S)]] \implies \exists X T. \text{well-ord}(X \text{ Un } Y, T)$
 ⟨proof⟩

lemma *disj-Un-eqpoll-sum*: $A \text{ Int } B = 0 \implies A \text{ Un } B \approx A + B$
 ⟨proof⟩

22.7 Finite and infinite sets

lemma *Finite-0* [*simp*]: $\text{Finite}(0)$
 ⟨proof⟩

lemma *lepoll-nat-imp-Finite*: $[[A \lesssim n; n:\text{nat}]] \implies \text{Finite}(A)$
 ⟨proof⟩

lemma *lesspoll-nat-is-Finite*:
 $A \prec \text{nat} \implies \text{Finite}(A)$
 ⟨proof⟩

lemma *lepoll-Finite*:
 $[[Y \lesssim X; \text{Finite}(X)]] \implies \text{Finite}(Y)$
 ⟨proof⟩

lemmas *subset-Finite = subset-imp-lepoll* [*THEN lepoll-Finite, standard*]

lemma *Finite-Int*: $\text{Finite}(A) \mid \text{Finite}(B) \implies \text{Finite}(A \text{ Int } B)$
 ⟨proof⟩

lemmas *Finite-Diff = Diff-subset* [*THEN subset-Finite, standard*]

lemma *Finite-cons*: $\text{Finite}(x) \implies \text{Finite}(\text{cons}(y,x))$
 ⟨proof⟩

lemma *Finite-succ*: $\text{Finite}(x) \implies \text{Finite}(\text{succ}(x))$
 ⟨proof⟩

lemma *Finite-cons-iff* [*iff*]: $\text{Finite}(\text{cons}(y,x)) \iff \text{Finite}(x)$
 ⟨proof⟩

lemma *Finite-succ-iff* [iff]: $Finite(succ(x)) \leftrightarrow Finite(x)$
 ⟨proof⟩

lemma *nat-le-infinite-Ord*:
 $[| Ord(i); \sim Finite(i) |] \implies nat\ le\ i$
 ⟨proof⟩

lemma *Finite-imp-well-ord*:
 $Finite(A) \implies \exists X\ r.\ well_ord(A,r)$
 ⟨proof⟩

lemma *succ-lepoll-imp-not-empty*: $succ(x) \lesssim y \implies y \neq 0$
 ⟨proof⟩

lemma *eqpoll-succ-imp-not-empty*: $x \approx succ(n) \implies x \neq 0$
 ⟨proof⟩

lemma *Finite-Fin-lemma* [rule-format]:
 $n \in nat \implies \forall A.\ (A \approx n \ \& \ A \subseteq X) \dashrightarrow A \in Fin(X)$
 ⟨proof⟩

lemma *Finite-Fin*: $[| Finite(A); A \subseteq X |] \implies A \in Fin(X)$
 ⟨proof⟩

lemma *eqpoll-imp-Finite-iff*: $A \approx B \implies Finite(A) \leftrightarrow Finite(B)$
 ⟨proof⟩

lemma *Fin-lemma* [rule-format]: $n : nat \implies ALL\ A.\ A \approx n \dashrightarrow A : Fin(A)$
 ⟨proof⟩

lemma *Finite-into-Fin*: $Finite(A) \implies A : Fin(A)$
 ⟨proof⟩

lemma *Fin-into-Finite*: $A : Fin(U) \implies Finite(A)$
 ⟨proof⟩

lemma *Finite-Fin-iff*: $Finite(A) \leftrightarrow A : Fin(A)$
 ⟨proof⟩

lemma *Finite-Un*: $[| Finite(A); Finite(B) |] \implies Finite(A\ Un\ B)$
 ⟨proof⟩

lemma *Finite-Un-iff* [simp]: $Finite(A\ Un\ B) \leftrightarrow (Finite(A) \ \& \ Finite(B))$
 ⟨proof⟩

The converse must hold too.

lemma *Finite-Union*: $[| ALL\ y:X.\ Finite(y); Finite(X) |] \implies Finite(Union(X))$
 ⟨proof⟩

lemma *Finite-induct* [case-names 0 cons, induct set: Finite]:

$$\llbracket \text{Finite}(A); P(0);$$

$$\quad \llbracket \forall x B. \llbracket \text{Finite}(B); x \sim: B; P(B) \rrbracket \implies P(\text{cons}(x, B)) \rrbracket$$

$$\implies P(A)$$

<proof>

lemma *Diff-sing-Finite*: $\text{Finite}(A - \{a\}) \implies \text{Finite}(A)$
<proof>

lemma *Diff-Finite* [rule-format]: $\text{Finite}(B) \implies \text{Finite}(A-B) \dashrightarrow \text{Finite}(A)$
<proof>

lemma *Finite-RepFun*: $\text{Finite}(A) \implies \text{Finite}(\text{RepFun}(A, f))$
<proof>

lemma *Finite-RepFun-iff-lemma* [rule-format]:

$$\llbracket \text{Finite}(x); \forall x y. f(x)=f(y) \implies x=y \rrbracket$$

$$\implies \forall A. x = \text{RepFun}(A, f) \dashrightarrow \text{Finite}(A)$$

<proof>

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*:

$$(\forall x y. f(x)=f(y) \dashrightarrow x=y) \implies \text{Finite}(\text{RepFun}(A, f)) \leftrightarrow \text{Finite}(A)$$

<proof>

lemma *Finite-Pow*: $\text{Finite}(A) \implies \text{Finite}(\text{Pow}(A))$
<proof>

lemma *Finite-Pow-imp-Finite*: $\text{Finite}(\text{Pow}(A)) \implies \text{Finite}(A)$
<proof>

lemma *Finite-Pow-iff* [iff]: $\text{Finite}(\text{Pow}(A)) \leftrightarrow \text{Finite}(A)$
<proof>

lemma *nat-wf-on-converse-Memrel*: $n:\text{nat} \implies \text{wf}[n](\text{converse}(\text{Memrel}(n)))$
<proof>

lemma *nat-well-ord-converse-Memrel*: $n:\text{nat} \implies \text{well-ord}(n, \text{converse}(\text{Memrel}(n)))$
<proof>

lemma *well-ord-converse*:

$$\llbracket \text{well-ord}(A,r); \text{well-ord}(\text{ordertype}(A,r), \text{converse}(\text{Memrel}(\text{ordertype}(A, r)))) \rrbracket \implies \text{well-ord}(A,\text{converse}(r))$$
 $\langle \text{proof} \rangle$

lemma *ordertype-eq-n*:

$$\llbracket \text{well-ord}(A,r); A \approx n; n:\text{nat} \rrbracket \implies \text{ordertype}(A,r)=n$$
 $\langle \text{proof} \rangle$

lemma *Finite-well-ord-converse*:

$$\llbracket \text{Finite}(A); \text{well-ord}(A,r) \rrbracket \implies \text{well-ord}(A,\text{converse}(r))$$
 $\langle \text{proof} \rangle$

lemma *nat-into-Finite*: $n:\text{nat} \implies \text{Finite}(n)$
 $\langle \text{proof} \rangle$

lemma *nat-not-Finite*: $\sim \text{Finite}(\text{nat})$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

end

23 Univ: The Cumulative Hierarchy and a Small Universe for Recursive Types

theory *Univ* **imports** *Epsilon Cardinal* **begin**

definition

$$\text{Vfrom} \quad :: [i,i] \Rightarrow i \text{ where}$$

$$\text{Vfrom}(A,i) == \text{transrec}(i, \%x f. A \text{ Un } (\bigcup_{y \in x} \text{Pow}(f'y)))$$

abbreviation

$$\text{Vset} \quad :: i \Rightarrow i \text{ where}$$

$$\text{Vset}(x) == \text{Vfrom}(\emptyset, x)$$

definition

$$\text{Vrec} \quad :: [i, [i,i] \Rightarrow i] \Rightarrow i \text{ where}$$

$$\text{Vrec}(a,H) == \text{transrec}(\text{rank}(a), \%x g. \text{lam } z: \text{Vset}(\text{succ}(x)).$$

$$H(z, \text{lam } w: \text{Vset}(x). g' \text{rank}(w)'w)) ' a$$

definition

$$\text{Vrecursor} \quad :: [[i,i] \Rightarrow i, i] \Rightarrow i \text{ where}$$

$$\text{Vrecursor}(H,a) == \text{transrec}(\text{rank}(a), \%x g. \text{lam } z: \text{Vset}(\text{succ}(x)).$$

$$H(\text{lam } w: \text{Vset}(x). g' \text{rank}(w)'w, z)) ' a$$

definition

$univ \quad :: i=>i$ **where**
 $univ(A) == Vfrom(A,nat)$

23.1 Immediate Consequences of the Definition of $Vfrom(A, i)$

NOT SUITABLE FOR REWRITING – RECURSIVE!

lemma $Vfrom$: $Vfrom(A,i) = A \ Un \ (\bigcup_{j \in i}. Pow(Vfrom(A,j)))$
 $\langle proof \rangle$

23.1.1 Monotonicity

lemma $Vfrom$ -mono [rule-format]:
 $A <= B \implies \forall j. i <= j \ \longrightarrow \ Vfrom(A,i) <= Vfrom(B,j)$
 $\langle proof \rangle$

lemma $VfromI$: $[[a \in Vfrom(A,j); j < i]] \implies a \in Vfrom(A,i)$
 $\langle proof \rangle$

23.1.2 A fundamental equality: $Vfrom$ does not require ordinals!

lemma $Vfrom$ -rank-subset1: $Vfrom(A,x) <= Vfrom(A,rank(x))$
 $\langle proof \rangle$

lemma $Vfrom$ -rank-subset2: $Vfrom(A,rank(x)) <= Vfrom(A,x)$
 $\langle proof \rangle$

lemma $Vfrom$ -rank-eq: $Vfrom(A,rank(x)) = Vfrom(A,x)$
 $\langle proof \rangle$

23.2 Basic Closure Properties

lemma zero-in- $Vfrom$: $y:x \implies 0 \in Vfrom(A,x)$
 $\langle proof \rangle$

lemma i -subset- $Vfrom$: $i <= Vfrom(A,i)$
 $\langle proof \rangle$

lemma A -subset- $Vfrom$: $A <= Vfrom(A,i)$
 $\langle proof \rangle$

lemmas A -into- $Vfrom = A$ -subset- $Vfrom$ [THEN subsetD]

lemma subset-mem- $Vfrom$: $a <= Vfrom(A,i) \implies a \in Vfrom(A,succ(i))$
 $\langle proof \rangle$

23.2.1 Finite sets and ordered pairs

lemma *singleton-in-Vfrom*: $a \in Vfrom(A,i) \implies \{a\} \in Vfrom(A,succ(i))$
(proof)

lemma *doubleton-in-Vfrom*:
[[$a \in Vfrom(A,i); b \in Vfrom(A,i)$]] $\implies \{a,b\} \in Vfrom(A,succ(i))$
(proof)

lemma *Pair-in-Vfrom*:
[[$a \in Vfrom(A,i); b \in Vfrom(A,i)$]] $\implies \langle a,b \rangle \in Vfrom(A,succ(succ(i)))$
(proof)

lemma *succ-in-Vfrom*: $a \leq Vfrom(A,i) \implies succ(a) \in Vfrom(A,succ(succ(i)))$
(proof)

23.3 0, Successor and Limit Equations for Vfrom

lemma *Vfrom-0*: $Vfrom(A,0) = A$
(proof)

lemma *Vfrom-succ-lemma*: $Ord(i) \implies Vfrom(A,succ(i)) = A \text{ Un } Pow(Vfrom(A,i))$
(proof)

lemma *Vfrom-succ*: $Vfrom(A,succ(i)) = A \text{ Un } Pow(Vfrom(A,i))$
(proof)

lemma *Vfrom-Union*: $y \in X \implies Vfrom(A,Union(X)) = (\bigcup_{y \in X} Vfrom(A,y))$
(proof)

23.4 Vfrom applied to Limit Ordinals

lemma *Limit-Vfrom-eq*:
 $Limit(i) \implies Vfrom(A,i) = (\bigcup_{y \in i} Vfrom(A,y))$
(proof)

lemma *Limit-VfromE*:
[[$a \in Vfrom(A,i); \sim R \implies Limit(i);$
!! $x. [[x < i; a \in Vfrom(A,x)]]$ $\implies R$
]] $\implies R$
(proof)

lemma *singleton-in-VLimit*:
[[$a \in Vfrom(A,i); Limit(i)$]] $\implies \{a\} \in Vfrom(A,i)$
(proof)

lemmas *Vfrom-UnI1* =
Un-upper1 [THEN subset-refl [THEN Vfrom-mono, THEN subsetD], standard]
lemmas *Vfrom-UnI2* =

Un-upper2 [THEN subset-refl [THEN Vfrom-mono, THEN subsetD], standard]

Hard work is finding a single $j:i$ such that $a,b_i = Vfrom(A,j)$

lemma *doubleton-in-VLimit*:

$[[a \in Vfrom(A,i); b \in Vfrom(A,i); Limit(i)]] ==> \{a,b\} \in Vfrom(A,i)$
 $\langle proof \rangle$

lemma *Pair-in-VLimit*:

$[[a \in Vfrom(A,i); b \in Vfrom(A,i); Limit(i)]] ==> \langle a,b \rangle \in Vfrom(A,i)$ $\langle proof \rangle$

lemma *product-VLimit*: $Limit(i) ==> Vfrom(A,i) * Vfrom(A,i) \leq Vfrom(A,i)$

$\langle proof \rangle$

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*: $[[n: nat; Limit(i)]] ==> n \in Vfrom(A,i)$

$\langle proof \rangle$

23.4.1 Closure under Disjoint Union

lemmas *zero-in-VLimit* = *Limit-has-0* [THEN *ltD*, THEN *zero-in-Vfrom*, standard]

lemma *one-in-VLimit*: $Limit(i) ==> 1 \in Vfrom(A,i)$

$\langle proof \rangle$

lemma *Inl-in-VLimit*:

$[[a \in Vfrom(A,i); Limit(i)]] ==> Inl(a) \in Vfrom(A,i)$
 $\langle proof \rangle$

lemma *Inr-in-VLimit*:

$[[b \in Vfrom(A,i); Limit(i)]] ==> Inr(b) \in Vfrom(A,i)$
 $\langle proof \rangle$

lemma *sum-VLimit*: $Limit(i) ==> Vfrom(C,i) + Vfrom(C,i) \leq Vfrom(C,i)$

$\langle proof \rangle$

lemmas *sum-subset-VLimit* = *subset-trans* [OF *sum-mono sum-VLimit*]

23.5 Properties assuming *Transset*(A)

lemma *Transset-Vfrom*: $Transset(A) ==> Transset(Vfrom(A,i))$

$\langle proof \rangle$

lemma *Transset-Vfrom-succ*:

$Transset(A) \implies Vfrom(A, succ(i)) = Pow(Vfrom(A,i))$
 ⟨proof⟩

lemma *Transset-Pair-subset*: $[[\langle a,b \rangle \leq C; Transset(C)]] \implies a: C \ \& \ b: C$
 ⟨proof⟩

lemma *Transset-Pair-subset-VLimit*:
 $[[\langle a,b \rangle \leq Vfrom(A,i); Transset(A); Limit(i)]]$
 $\implies \langle a,b \rangle \in Vfrom(A,i)$
 ⟨proof⟩

lemma *Union-in-Vfrom*:
 $[[X \in Vfrom(A,j); Transset(A)]] \implies Union(X) \in Vfrom(A, succ(j))$
 ⟨proof⟩

lemma *Union-in-VLimit*:
 $[[X \in Vfrom(A,i); Limit(i); Transset(A)]] \implies Union(X) \in Vfrom(A,i)$
 ⟨proof⟩

General theorem for membership in $Vfrom(A,i)$ when i is a limit ordinal

lemma *in-VLimit*:
 $[[a \in Vfrom(A,i); b \in Vfrom(A,i); Limit(i);$
 $!!x \ y \ j. [[j < i; 1:j; x \in Vfrom(A,j); y \in Vfrom(A,j)]]$
 $\implies \exists k. h(x,y) \in Vfrom(A,k) \ \& \ k < i]]$
 $\implies h(a,b) \in Vfrom(A,i)$ ⟨proof⟩

23.5.1 Products

lemma *prod-in-Vfrom*:
 $[[a \in Vfrom(A,j); b \in Vfrom(A,j); Transset(A)]]$
 $\implies a*b \in Vfrom(A, succ(succ(succ(j))))$
 ⟨proof⟩

lemma *prod-in-VLimit*:
 $[[a \in Vfrom(A,i); b \in Vfrom(A,i); Limit(i); Transset(A)]]$
 $\implies a*b \in Vfrom(A,i)$
 ⟨proof⟩

23.5.2 Disjoint Sums, or Quine Ordered Pairs

lemma *sum-in-Vfrom*:
 $[[a \in Vfrom(A,j); b \in Vfrom(A,j); Transset(A); 1:j]]$
 $\implies a+b \in Vfrom(A, succ(succ(succ(j))))$
 ⟨proof⟩

lemma *sum-in-VLimit*:
 $[[a \in Vfrom(A,i); b \in Vfrom(A,i); Limit(i); Transset(A)]]$
 $\implies a+b \in Vfrom(A,i)$
 ⟨proof⟩

23.5.3 Function Space!

lemma *fun-in-Vfrom*:

$$\llbracket a \in Vfrom(A,j); b \in Vfrom(A,j); Transset(A) \rrbracket \implies a \rightarrow b \in Vfrom(A, succ(succ(succ(succ(j)))))$$

<proof>

lemma *fun-in-VLimit*:

$$\llbracket a \in Vfrom(A,i); b \in Vfrom(A,i); Limit(i); Transset(A) \rrbracket \implies a \rightarrow b \in Vfrom(A,i)$$

<proof>

lemma *Pow-in-Vfrom*:

$$\llbracket a \in Vfrom(A,j); Transset(A) \rrbracket \implies Pow(a) \in Vfrom(A, succ(succ(j)))$$

<proof>

lemma *Pow-in-VLimit*:

$$\llbracket a \in Vfrom(A,i); Limit(i); Transset(A) \rrbracket \implies Pow(a) \in Vfrom(A,i)$$

<proof>

23.6 The Set $Vset(i)$

lemma *Vset*: $Vset(i) = (\bigcup_{j \in i} Pow(Vset(j)))$

<proof>

lemmas *Vset-succ = Transset-0* [*THEN Transset-Vfrom-succ, standard*]

lemmas *Transset-Vset = Transset-0* [*THEN Transset-Vfrom, standard*]

23.6.1 Characterisation of the elements of $Vset(i)$

lemma *VsetD* [*rule-format*]: $Ord(i) \implies \forall b. b \in Vset(i) \dashrightarrow rank(b) < i$

<proof>

lemma *VsetI-lemma* [*rule-format*]:

$$Ord(i) \implies \forall b. rank(b) \in i \dashrightarrow b \in Vset(i)$$

<proof>

lemma *VsetI*: $rank(x) < i \implies x \in Vset(i)$

<proof>

Merely a lemma for the next result

lemma *Vset-Ord-rank-iff*: $Ord(i) \implies b \in Vset(i) \leftrightarrow rank(b) < i$

<proof>

lemma *Vset-rank-iff* [*simp*]: $b \in Vset(a) \leftrightarrow rank(b) < rank(a)$

<proof>

This is $rank(rank(a)) = rank(a)$

declare *Ord-rank* [*THEN rank-of-Ord, simp*]

lemma *rank-Vset*: $Ord(i) ==> rank(Vset(i)) = i$
<proof>

lemma *Finite-Vset*: $i \in nat ==> Finite(Vset(i))$
<proof>

23.6.2 Reasoning about Sets in Terms of Their Elements' Ranks

lemma *arg-subset-Vset-rank*: $a \leq Vset(rank(a))$
<proof>

lemma *Int-Vset-subset*:
[[!!i. $Ord(i) ==> a \text{ Int } Vset(i) \leq b$]] ==> $a \leq b$
<proof>

23.6.3 Set Up an Environment for Simplification

lemma *rank-Inl*: $rank(a) < rank(Inl(a))$
<proof>

lemma *rank-Inr*: $rank(a) < rank(Inr(a))$
<proof>

lemmas *rank-rls* = *rank-Inl rank-Inr rank-pair1 rank-pair2*

23.6.4 Recursion over Vset Levels!

NOT SUITABLE FOR REWRITING: recursive!

lemma *Vrec*: $Vrec(a,H) = H(a, lam x: Vset(rank(a)). Vrec(x,H))$
<proof>

This form avoids giant explosions in proofs. NOTE USE OF ==

lemma *def-Vrec*:
[[!!x. $h(x) == Vrec(x,H)$]] ==>
 $h(a) = H(a, lam x: Vset(rank(a)). h(x))$
<proof>

NOT SUITABLE FOR REWRITING: recursive!

lemma *Vrecursor*:
 $Vrecursor(H,a) = H(lam x: Vset(rank(a)). Vrecursor(H,x), a)$
<proof>

This form avoids giant explosions in proofs. NOTE USE OF ==

lemma *def-Vrecursor*:
 $h == Vrecursor(H) ==> h(a) = H(lam x: Vset(rank(a)). h(x), a)$
<proof>

23.7 The Datatype Universe: $univ(A)$

lemma *univ-mono*: $A \leq B \implies univ(A) \leq univ(B)$
<proof>

lemma *Transset-univ*: $Transset(A) \implies Transset(univ(A))$
<proof>

23.7.1 The Set $univ(A)$ as a Limit

lemma *univ-eq-UN*: $univ(A) = (\bigcup i \in nat. Vfrom(A, i))$
<proof>

lemma *subset-univ-eq-Int*: $c \leq univ(A) \implies c = (\bigcup i \in nat. c \text{ Int } Vfrom(A, i))$
<proof>

lemma *univ-Int-Vfrom-subset*:
[[$a \leq univ(X)$;
!! $i. i : nat \implies a \text{ Int } Vfrom(X, i) \leq b$]]
 $\implies a \leq b$
<proof>

lemma *univ-Int-Vfrom-eq*:
[[$a \leq univ(X)$; $b \leq univ(X)$;
!! $i. i : nat \implies a \text{ Int } Vfrom(X, i) = b \text{ Int } Vfrom(X, i)$]]
 $\implies a = b$
<proof>

23.8 Closure Properties for $univ(A)$

lemma *zero-in-univ*: $0 \in univ(A)$
<proof>

lemma *zero-subset-univ*: $\{0\} \leq univ(A)$
<proof>

lemma *A-subset-univ*: $A \leq univ(A)$
<proof>

lemmas *A-into-univ* = *A-subset-univ* [THEN *subsetD*, *standard*]

23.8.1 Closure under Unordered and Ordered Pairs

lemma *singleton-in-univ*: $a : univ(A) \implies \{a\} \in univ(A)$
<proof>

lemma *doubleton-in-univ*:
[[$a : univ(A)$; $b : univ(A)$]] $\implies \{a, b\} \in univ(A)$
<proof>

lemma *Pair-in-univ*:

$\llbracket a: \text{univ}(A); b: \text{univ}(A) \rrbracket \implies \langle a, b \rangle \in \text{univ}(A)$
<proof>

lemma *Union-in-univ*:

$\llbracket X: \text{univ}(A); \text{Transset}(A) \rrbracket \implies \text{Union}(X) \in \text{univ}(A)$
<proof>

lemma *product-univ*: $\text{univ}(A) * \text{univ}(A) \leq \text{univ}(A)$

<proof>

23.8.2 The Natural Numbers

lemma *nat-subset-univ*: $\text{nat} \leq \text{univ}(A)$

<proof>

$\text{n:nat} \implies \text{n:univ}(A)$

lemmas *nat-into-univ* = *nat-subset-univ* [*THEN subsetD, standard*]

23.8.3 Instances for 1 and 2

lemma *one-in-univ*: $1 \in \text{univ}(A)$

<proof>

unused!

lemma *two-in-univ*: $2 \in \text{univ}(A)$

<proof>

lemma *bool-subset-univ*: $\text{bool} \leq \text{univ}(A)$

<proof>

lemmas *bool-into-univ* = *bool-subset-univ* [*THEN subsetD, standard*]

23.8.4 Closure under Disjoint Union

lemma *Inl-in-univ*: $a: \text{univ}(A) \implies \text{Inl}(a) \in \text{univ}(A)$

<proof>

lemma *Inr-in-univ*: $b: \text{univ}(A) \implies \text{Inr}(b) \in \text{univ}(A)$

<proof>

lemma *sum-univ*: $\text{univ}(C) + \text{univ}(C) \leq \text{univ}(C)$

<proof>

lemmas *sum-subset-univ* = *subset-trans* [*OF sum-mono sum-univ*]

lemma *Sigma-subset-univ*:

$\llbracket A \subseteq \text{univ}(D); \bigwedge x. x \in A \implies B(x) \subseteq \text{univ}(D) \rrbracket \implies \text{Sigma}(A, B) \subseteq \text{univ}(D)$
<proof>

23.9 Finite Branching Closure Properties

23.9.1 Closure under Finite Powerset

lemma *Fin-Vfrom-lemma*:

$\llbracket b: \text{Fin}(\text{Vfrom}(A,i)); \text{Limit}(i) \rrbracket \implies \exists j. b \leq \text{Vfrom}(A,j) \ \& \ j < i$
<proof>

lemma *Fin-VLimit*: $\text{Limit}(i) \implies \text{Fin}(\text{Vfrom}(A,i)) \leq \text{Vfrom}(A,i)$
<proof>

lemmas *Fin-subset-VLimit = subset-trans [OF Fin-mono Fin-VLimit]*

lemma *Fin-univ*: $\text{Fin}(\text{univ}(A)) \leq \text{univ}(A)$
<proof>

23.9.2 Closure under Finite Powers: Functions from a Natural Number

lemma *nat-fun-VLimit*:

$\llbracket n: \text{nat}; \text{Limit}(i) \rrbracket \implies n \rightarrow \text{Vfrom}(A,i) \leq \text{Vfrom}(A,i)$
<proof>

lemmas *nat-fun-subset-VLimit = subset-trans [OF Pi-mono nat-fun-VLimit]*

lemma *nat-fun-univ*: $n: \text{nat} \implies n \rightarrow \text{univ}(A) \leq \text{univ}(A)$
<proof>

23.9.3 Closure under Finite Function Space

General but seldom-used version; normally the domain is fixed

lemma *FiniteFun-VLimit1*:

$\text{Limit}(i) \implies \text{Vfrom}(A,i) \dashv\vdash \text{Vfrom}(A,i) \leq \text{Vfrom}(A,i)$
<proof>

lemma *FiniteFun-univ1*: $\text{univ}(A) \dashv\vdash \text{univ}(A) \leq \text{univ}(A)$
<proof>

Version for a fixed domain

lemma *FiniteFun-VLimit*:

$\llbracket W \leq \text{Vfrom}(A,i); \text{Limit}(i) \rrbracket \implies W \dashv\vdash \text{Vfrom}(A,i) \leq \text{Vfrom}(A,i)$
<proof>

lemma *FiniteFun-univ*:

$W \leq \text{univ}(A) \implies W \dashv\vdash \text{univ}(A) \leq \text{univ}(A)$
<proof>

lemma *FiniteFun-in-univ*:

$\llbracket f: W \dashv\vdash \text{univ}(A); W \leq \text{univ}(A) \rrbracket \implies f \in \text{univ}(A)$

<proof>

Remove $j=$ from the rule above

lemmas *FiniteFun-in-univ'* = *FiniteFun-in-univ* [*OF - subsetI*]

23.10 * For QUniv. Properties of Vfrom analogous to the "take-lemma" *

Intersecting $a*b$ with Vfrom...

This version says a, b exist one level down, in the smaller set $Vfrom(X,i)$

lemma *doubleton-in-Vfrom-D*:

$[[\{a,b\} \in Vfrom(X,succ(i)); Transset(X)]]$
 $==> a \in Vfrom(X,i) \ \& \ b \in Vfrom(X,i)$

<proof>

This weaker version says a, b exist at the same level

lemmas *Vfrom-doubleton-D* = *Transset-Vfrom* [*THEN Transset-doubleton-D, standard*]

lemma *Pair-in-Vfrom-D*:

$[[\langle a,b \rangle \in Vfrom(X,succ(i)); Transset(X)]]$
 $==> a \in Vfrom(X,i) \ \& \ b \in Vfrom(X,i)$

<proof>

lemma *product-Int-Vfrom-subset*:

Transset(X) ==>
 $(a*b) \text{ Int } Vfrom(X, succ(i)) \leq (a \text{ Int } Vfrom(X,i)) * (b \text{ Int } Vfrom(X,i))$

<proof>

<ML>

end

24 QUniv: 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**
 $quniv(A) == Pow(univ(eclose(A)))$

24.1 Properties involving Transset and Sum

lemma *Transset-includes-summands*:

$[| Transset(C); A+B \leq C |] \Rightarrow A \leq C \ \& \ B \leq C$
<proof>

lemma *Transset-sum-Int-subset*:

$Transset(C) \Rightarrow (A+B) \ Int \ C \leq (A \ Int \ C) + (B \ Int \ C)$
<proof>

24.2 Introduction and Elimination Rules

lemma *qunivI*: $X \leq univ(eclose(A)) \Rightarrow X : quniv(A)$

<proof>

lemma *qunivD*: $X : quniv(A) \Rightarrow X \leq univ(eclose(A))$

<proof>

lemma *quniv-mono*: $A \leq B \Rightarrow quniv(A) \leq quniv(B)$

<proof>

24.3 Closure Properties

lemma *univ-eclose-subset-quniv*: $univ(eclose(A)) \leq quniv(A)$

<proof>

lemma *univ-subset-quniv*: $univ(A) \leq quniv(A)$

<proof>

lemmas *univ-into-quniv = univ-subset-quniv [THEN subsetD, standard]*

lemma *Pow-univ-subset-quniv*: $Pow(univ(A)) \leq quniv(A)$

<proof>

lemmas *univ-subset-into-quniv =*

PowI [THEN Pow-univ-subset-quniv [THEN subsetD], standard]

lemmas $zero\text{-}in\text{-}quniv = zero\text{-}in\text{-}univ$ [THEN univ-into-quniv, standard]
lemmas $one\text{-}in\text{-}quniv = one\text{-}in\text{-}univ$ [THEN univ-into-quniv, standard]
lemmas $two\text{-}in\text{-}quniv = two\text{-}in\text{-}univ$ [THEN univ-into-quniv, standard]

lemmas $A\text{-}subset\text{-}quniv = subset\text{-}trans$ [OF A-subset-univ univ-subset-quniv]

lemmas $A\text{-}into\text{-}quniv = A\text{-}subset\text{-}quniv$ [THEN subsetD, standard]

lemma $QPair\text{-}subset\text{-}univ$:

$\llbracket a \leq univ(A); b \leq univ(A) \rrbracket \implies \langle a;b \rangle \leq univ(A)$
 ⟨proof⟩

24.4 Quine Disjoint Sum

lemma $QInl\text{-}subset\text{-}univ$: $a \leq univ(A) \implies QInl(a) \leq univ(A)$
 ⟨proof⟩

lemmas $naturals\text{-}subset\text{-}nat =$

$Ord\text{-}nat$ [THEN Ord-is-Transset, unfolded Transset-def, THEN bspec, standard]

lemmas $naturals\text{-}subset\text{-}univ =$

$subset\text{-}trans$ [OF naturals-subset-nat nat-subset-univ]

lemma $QInr\text{-}subset\text{-}univ$: $a \leq univ(A) \implies QInr(a) \leq univ(A)$
 ⟨proof⟩

24.5 Closure for Quine-Inspired Products and Sums

lemma $QPair\text{-}in\text{-}quniv$:

$\llbracket a : quniv(A); b : quniv(A) \rrbracket \implies \langle a;b \rangle : quniv(A)$
 ⟨proof⟩

lemma $QSigma\text{-}quniv$: $quniv(A) \langle * \rangle quniv(A) \leq quniv(A)$
 ⟨proof⟩

lemmas $QSigma\text{-}subset\text{-}quniv = subset\text{-}trans$ [OF QSigma-mono QSigma-quniv]

lemma $quniv\text{-}QPair\text{-}D$:

$\langle a;b \rangle : quniv(A) \implies a : quniv(A) \ \& \ b : quniv(A)$
 ⟨proof⟩

lemmas $quniv\text{-}QPair\text{-}E = quniv\text{-}QPair\text{-}D$ [THEN conjE, standard]

lemma $quniv\text{-}QPair\text{-}iff$: $\langle a;b \rangle : quniv(A) \iff a : quniv(A) \ \& \ b : quniv(A)$
 ⟨proof⟩

24.6 Quine Disjoint Sum

lemma *QInl-in-quniv*: $a: \text{quniv}(A) \implies \text{QInl}(a) : \text{quniv}(A)$
<proof>

lemma *QInr-in-quniv*: $b: \text{quniv}(A) \implies \text{QInr}(b) : \text{quniv}(A)$
<proof>

lemma *qsum-quniv*: $\text{quniv}(C) <+> \text{quniv}(C) \leq \text{quniv}(C)$
<proof>

lemmas *qsum-subset-quniv* = *subset-trans* [*OF* *qsum-mono* *qsum-quniv*]

24.7 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*, *standard*]

lemmas *bool-subset-quniv* = *subset-trans* [*OF* *bool-subset-univ* *univ-subset-quniv*]

lemmas *bool-into-quniv* = *bool-subset-quniv* [*THEN* *subsetD*, *standard*]

lemma *QPair-Int-Vfrom-succ-subset*:
Transset(*X*) \implies
 $\langle a; b \rangle \text{ Int Vfrom}(X, \text{succ}(i)) \leq \langle a \text{ Int Vfrom}(X, i); b \text{ Int Vfrom}(X, i) \rangle$
<proof>

24.8 "Take-Lemma" Rules

lemma *QPair-Int-Vfrom-subset*:
Transset(*X*) \implies
 $\langle a; b \rangle \text{ Int Vfrom}(X, i) \leq \langle a \text{ Int Vfrom}(X, i); b \text{ Int Vfrom}(X, i) \rangle$
<proof>

lemmas *QPair-Int-Vset-subset-trans* =
subset-trans [*OF* *Transset-0* [*THEN* *QPair-Int-Vfrom-subset*] *QPair-mono*]

lemma *QPair-Int-Vset-subset-UN*:
Ord(*i*) $\implies \langle a; b \rangle \text{ Int Vset}(i) \leq (\bigcup_{j \in i}. \langle a \text{ Int Vset}(j); b \text{ Int Vset}(j) \rangle)$
<proof>

end

25 Datatype-ZF: Datatype and CoDatatype Definitions

```
theory Datatype-ZF
imports Inductive-ZF Univ QUniv
uses Tools/datatype-package.ML
begin
```

<ML>

```
end
```

26 Arith: 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, %x. x, y)
```

definition

```
natify :: i=>i    where
  natify == Vrecursor(%f a. if a = succ(pred(a)) then succ(f*pred(a))
                      else 0)
```

consts

```
raw-add :: [i,i]=>i
raw-diff :: [i,i]=>i
raw-mult :: [i,i]=>i
```

primrec

```
raw-add (0, n) = n
raw-add (succ(m), n) = succ(raw-add(m, n))
```

primrec

```
raw-diff-0:   raw-diff(m, 0) = m
raw-diff-succ: raw-diff(m, succ(n)) =
               nat-case(0, %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]=>i          (infixl #+ 65) where
```

$m \#+ n == \text{raw-add } (\text{natify}(m), \text{natify}(n))$

definition

$\text{diff} :: [i, i] => i$ (infixl #- 65) **where**
 $m \#- n == \text{raw-diff } (\text{natify}(m), \text{natify}(n))$

definition

$\text{mult} :: [i, i] => i$ (infixl #* 70) **where**
 $m \#* n == \text{raw-mult } (\text{natify}(m), \text{natify}(n))$

definition

$\text{raw-div} :: [i, i] => i$ **where**
 $\text{raw-div } (m, n) ==$
 $\text{transrec}(m, \%j f. \text{if } j < n \mid n=0 \text{ then } 0 \text{ else succ}(f^{\text{!}}(j \#- n)))$

definition

$\text{raw-mod} :: [i, i] => i$ **where**
 $\text{raw-mod } (m, n) ==$
 $\text{transrec}(m, \%j f. \text{if } j < n \mid n=0 \text{ then } j \text{ else } f^{\text{!}}(j \#- n))$

definition

$\text{div} :: [i, i] => i$ (infixl div 70) **where**
 $m \text{ div } n == \text{raw-div } (\text{natify}(m), \text{natify}(n))$

definition

$\text{mod} :: [i, i] => i$ (infixl mod 70) **where**
 $m \text{ mod } n == \text{raw-mod } (\text{natify}(m), \text{natify}(n))$

notation (*xsymbols*)

mult (infixr #× 70)

notation (*HTML output*)

mult (infixr #× 70)

declare *rec-type* [*simp*]

nat-0-le [*simp*]

lemma *zero-lt-lemma*: [$0 < k; k \in \text{nat}$] ==> $\exists j \in \text{nat}. k = \text{succ}(j)$
(*proof*)

lemmas *zero-lt-natE* = *zero-lt-lemma* [*THEN* *bexE*, *standard*]

26.1 *natify*, the Coercion to *nat*

lemma *pred-succ-eq* [*simp*]: $\text{pred}(\text{succ}(y)) = y$
(*proof*)

lemma *natify-succ*: $\text{natify}(\text{succ}(x)) = \text{succ}(\text{natify}(x))$
<proof>

lemma *natify-0* [*simp*]: $\text{natify}(0) = 0$
<proof>

lemma *natify-non-succ*: $\forall z. x \sim = \text{succ}(z) \implies \text{natify}(x) = 0$
<proof>

lemma *natify-in-nat* [*iff, TC*]: $\text{natify}(x) \in \text{nat}$
<proof>

lemma *natify-ident* [*simp*]: $n \in \text{nat} \implies \text{natify}(n) = n$
<proof>

lemma *natify-eqE*: $[\text{natify}(x) = y; x \in \text{nat}] \implies x = y$
<proof>

lemma *natify-idem* [*simp*]: $\text{natify}(\text{natify}(x)) = \text{natify}(x)$
<proof>

lemma *add-natify1* [*simp*]: $\text{natify}(m) \# + n = m \# + n$
<proof>

lemma *add-natify2* [*simp*]: $m \# + \text{natify}(n) = m \# + n$
<proof>

lemma *mult-natify1* [*simp*]: $\text{natify}(m) \# * n = m \# * n$
<proof>

lemma *mult-natify2* [*simp*]: $m \# * \text{natify}(n) = m \# * n$
<proof>

lemma *diff-natify1* [*simp*]: $\text{natify}(m) \# - n = m \# - n$
<proof>

lemma *diff-natify2* [*simp*]: $m \# - \text{natify}(n) = m \# - n$
<proof>

lemma *mod-natify1* [*simp*]: $\text{natify}(m) \text{ mod } n = m \text{ mod } n$
<proof>

lemma *mod-natify2* [*simp*]: $m \text{ mod } \text{natify}(n) = m \text{ mod } n$
<proof>

lemma *div-natify1* [*simp*]: $\text{natify}(m) \text{ div } n = m \text{ div } n$
<proof>

lemma *div-natify2* [*simp*]: $m \text{ div } \text{natify}(n) = m \text{ div } n$
<proof>

26.2 Typing rules

lemma *raw-add-type*: $[| m \in \text{nat}; n \in \text{nat} |] \implies \text{raw-add } (m, n) \in \text{nat}$
<proof>

lemma *add-type* [*iff, TC*]: $m \# + n \in \text{nat}$
<proof>

lemma *raw-mult-type*: $[| m \in \text{nat}; n \in \text{nat} |] \implies \text{raw-mult } (m, n) \in \text{nat}$
<proof>

lemma *mult-type* [*iff, TC*]: $m \# * n \in \text{nat}$
<proof>

lemma *raw-diff-type*: $[| m \in \text{nat}; n \in \text{nat} |] \implies \text{raw-diff } (m, n) \in \text{nat}$
<proof>

lemma *diff-type* [*iff, TC*]: $m \# - n \in \text{nat}$
<proof>

lemma *diff-0-eq-0* [*simp*]: $0 \# - n = 0$
<proof>

lemma *diff-succ-succ* [*simp*]: $\text{succ}(m) \# - \text{succ}(n) = m \# - n$
<proof>

declare *raw-diff-succ* [*simp del*]

lemma *diff-0* [*simp*]: $m \#- 0 = \text{nativy}(m)$
(*proof*)

lemma *diff-le-self*: $m \in \text{nat} \implies (m \#- n) \text{ le } m$
(*proof*)

26.3 Addition

lemma *add-0-nativy* [*simp*]: $0 \#+ m = \text{nativy}(m)$
(*proof*)

lemma *add-succ* [*simp*]: $\text{succ}(m) \#+ n = \text{succ}(m \#+ n)$
(*proof*)

lemma *add-0*: $m \in \text{nat} \implies 0 \#+ m = m$
(*proof*)

lemma *add-assoc*: $(m \#+ n) \#+ k = m \#+ (n \#+ k)$
(*proof*)

lemma *add-0-right-nativy* [*simp*]: $m \#+ 0 = \text{nativy}(m)$
(*proof*)

lemma *add-succ-right* [*simp*]: $m \#+ \text{succ}(n) = \text{succ}(m \#+ n)$
(*proof*)

lemma *add-0-right*: $m \in \text{nat} \implies m \#+ 0 = m$
(*proof*)

lemma *add-commute*: $m \#+ n = n \#+ m$
(*proof*)

lemma *add-left-commute*: $m \#+ (n \#+ k) = n \#+ (m \#+ k)$
(*proof*)

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

lemma *raw-add-left-cancel*:
[[*raw-add*(k, m) = *raw-add*(k, n); $k \in \text{nat}$]] $\implies m = n$
(*proof*)

lemma *add-left-cancel-natify*: $k \# + m = k \# + n \implies \text{natify}(m) = \text{natify}(n)$
 ⟨*proof*⟩

lemma *add-left-cancel*:
 $[[i = j; i \# + m = j \# + n; m \in \text{nat}; n \in \text{nat}]] \implies m = n$
 ⟨*proof*⟩

lemma *add-le-elim1-natify*: $k \# + m \text{ le } k \# + n \implies \text{natify}(m) \text{ le } \text{natify}(n)$
 ⟨*proof*⟩

lemma *add-le-elim1*: $[[k \# + m \text{ le } k \# + n; m \in \text{nat}; n \in \text{nat}]] \implies m \text{ le } n$
 ⟨*proof*⟩

lemma *add-lt-elim1-natify*: $k \# + m < k \# + n \implies \text{natify}(m) < \text{natify}(n)$
 ⟨*proof*⟩

lemma *add-lt-elim1*: $[[k \# + m < k \# + n; m \in \text{nat}; n \in \text{nat}]] \implies m < n$
 ⟨*proof*⟩

lemma *zero-less-add*: $[[n \in \text{nat}; m \in \text{nat}]] \implies 0 < m \# + n \iff (0 < m \mid 0 < n)$
 ⟨*proof*⟩

26.4 Monotonicity of Addition

lemma *add-lt-mono1*: $[[i < j; j \in \text{nat}]] \implies i \# + k < j \# + k$
 ⟨*proof*⟩

strict, in second argument

lemma *add-lt-mono2*: $[[i < j; j \in \text{nat}]] \implies k \# + i < k \# + j$
 ⟨*proof*⟩

A [clumsy] way of lifting \leq monotonicity to \leq monotonicity

lemma *Ord-lt-mono-imp-le-mono*:
assumes *lt-mono*: $!!i j. [[i < j; j:k]] \implies f(i) < f(j)$
and *ford*: $!!i. i:k \implies \text{Ord}(f(i))$
and *leij*: $i \text{ le } j$
and *jink*: $j:k$
shows $f(i) \text{ le } f(j)$
 ⟨*proof*⟩

\leq monotonicity, 1st argument

lemma *add-le-mono1*: $[[i \text{ le } j; j \in \text{nat}]] \implies i \# + k \text{ le } j \# + k$
 ⟨*proof*⟩

\leq monotonicity, both arguments

lemma *add-le-mono*: $[[i \text{ le } j; k \text{ le } l; j \in \text{nat}; l \in \text{nat}]] \implies i \# + k \text{ le } j \# + l$

<proof>

Combinations of less-than and less-than-or-equals

lemma *add-lt-le-mono*: $[[i < j; k \leq l; j \in \text{nat}; l \in \text{nat}]] \implies i \# + k < j \# + l$
<proof>

lemma *add-le-lt-mono*: $[[i \leq j; k < l; j \in \text{nat}; l \in \text{nat}]] \implies i \# + k < j \# + l$
<proof>

Less-than: in other words, strict in both arguments

lemma *add-lt-mono*: $[[i < j; k < l; j \in \text{nat}; l \in \text{nat}]] \implies i \# + k < j \# + l$
<proof>

lemma *diff-add-inverse*: $(n \# + m) \# - n = \text{nativify}(m)$
<proof>

lemma *diff-add-inverse2*: $(m \# + n) \# - n = \text{nativify}(m)$
<proof>

lemma *diff-cancel*: $(k \# + m) \# - (k \# + n) = m \# - n$
<proof>

lemma *diff-cancel2*: $(m \# + k) \# - (n \# + k) = m \# - n$
<proof>

lemma *diff-add-0*: $n \# - (n \# + m) = 0$
<proof>

lemma *pred-0* [*simp*]: $\text{pred}(0) = 0$
<proof>

lemma *eq-succ-imp-eq-m1*: $[[i = \text{succ}(j); i \in \text{nat}]] \implies j = i \# - 1 \ \& \ j \in \text{nat}$
<proof>

lemma *pred-Un-distrib*:
 $[[i \in \text{nat}; j \in \text{nat}]] \implies \text{pred}(i \text{ Un } j) = \text{pred}(i) \text{ Un } \text{pred}(j)$
<proof>

lemma *pred-type* [*TC, simp*]:
 $i \in \text{nat} \implies \text{pred}(i) \in \text{nat}$
<proof>

lemma *nat-diff-pred*: $[[i \in \text{nat}; j \in \text{nat}]] \implies i \# - \text{succ}(j) = \text{pred}(i \# - j)$
<proof>

lemma *diff-succ-eq-pred*: $i \# - \text{succ}(j) = \text{pred}(i \# - j)$
<proof>

lemma *nat-diff-Un-distrib*:

$[[i \in \text{nat}; j \in \text{nat}; k \in \text{nat}]] \implies (i \text{ Un } j) \# - k = (i \# - k) \text{ Un } (j \# - k)$
<proof>

lemma *diff-Un-distrib*:

$[[i \in \text{nat}; j \in \text{nat}]] \implies (i \text{ Un } j) \# - k = (i \# - k) \text{ Un } (j \# - k)$
<proof>

We actually prove $i \# - j \# - k = i \# - (j \# + k)$

lemma *diff-diff-left [simplified]*:

$\text{nativify}(i) \# - \text{nativify}(j) \# - k = \text{nativify}(i) \# - (\text{nativify}(j) \# + k)$
<proof>

lemma *eq-add-iff*: $(u \# + m = u \# + n) \iff (0 \# + m = \text{nativify}(n))$
<proof>

lemma *less-add-iff*: $(u \# + m < u \# + n) \iff (0 \# + m < \text{nativify}(n))$
<proof>

lemma *diff-add-eq*: $((u \# + m) \# - (u \# + n)) = ((0 \# + m) \# - n)$
<proof>

lemma *eq-cong2*: $u = u' \implies (t == u) == (t == u')$
<proof>

lemma *iff-cong2*: $u \iff u' \implies (t == u) == (t == u')$
<proof>

26.5 Multiplication

lemma *mult-0 [simp]*: $0 \# * m = 0$
<proof>

lemma *mult-succ [simp]*: $\text{succ}(m) \# * n = n \# + (m \# * n)$
<proof>

lemma *mult-0-right [simp]*: $m \# * 0 = 0$
<proof>

lemma *mult-succ-right [simp]*: $m \# * \text{succ}(n) = m \# + (m \# * n)$
<proof>

lemma *mult-1-natify* [simp]: $1 \#* n = \text{natify}(n)$
<proof>

lemma *mult-1-right-natify* [simp]: $n \#* 1 = \text{natify}(n)$
<proof>

lemma *mult-1*: $n \in \text{nat} \implies 1 \#* n = n$
<proof>

lemma *mult-1-right*: $n \in \text{nat} \implies n \#* 1 = n$
<proof>

lemma *mult-commute*: $m \#* n = n \#* m$
<proof>

lemma *add-mult-distrib*: $(m \#+ n) \#* k = (m \#* k) \#+ (n \#* k)$
<proof>

lemma *add-mult-distrib-left*: $k \#* (m \#+ n) = (k \#* m) \#+ (k \#* n)$
<proof>

lemma *mult-assoc*: $(m \#* n) \#* k = m \#* (n \#* k)$
<proof>

lemma *mult-left-commute*: $m \#* (n \#* k) = n \#* (m \#* k)$
<proof>

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

lemma *lt-succ-eq-0-disj*:
[[$m \in \text{nat}; n \in \text{nat}$]]
 $\implies (m < \text{succ}(n)) \iff (m = 0 \mid (\exists j \in \text{nat}. m = \text{succ}(j) \ \& \ j < n))$
<proof>

lemma *less-diff-conv* [rule-format]:
[[$j \in \text{nat}; k \in \text{nat}$]] $\implies \forall i \in \text{nat}. (i < j \#- k) \iff (i \#+ k < j)$
<proof>

lemmas *nat-typechecks = rec-type nat-0I nat-1I nat-succI Ord-nat*

end

27 ArithSimp: Arithmetic with simplification

```
theory ArithSimp
imports Arith
uses ~~/src/Provers/Arith/cancel-numerals.ML
     ~~/src/Provers/Arith/combine-numerals.ML
     arith-data.ML
begin
```

27.1 Difference

```
lemma diff-self-eq-0 [simp]:  $m \#- m = 0$ 
⟨proof⟩
```

```
lemma add-diff-inverse: [ $n \text{ le } m$ ;  $m:\text{nat}$ ] ==>  $n \#+ (m \#- n) = m$ 
⟨proof⟩
```

```
lemma add-diff-inverse2: [ $n \text{ le } m$ ;  $m:\text{nat}$ ] ==>  $(m \#- n) \#+ n = m$ 
⟨proof⟩
```

```
lemma diff-succ: [ $n \text{ le } m$ ;  $m:\text{nat}$ ] ==>  $\text{succ}(m) \#- n = \text{succ}(m \#- n)$ 
⟨proof⟩
```

```
lemma zero-less-diff [simp]:
  [ $m:\text{nat}$ ;  $n:\text{nat}$ ] ==>  $0 < (n \#- m) \iff m < n$ 
⟨proof⟩
```

```
lemma diff-mult-distrib:  $(m \#- n) \#* k = (m \#* k) \#- (n \#* k)$ 
⟨proof⟩
```

```
lemma diff-mult-distrib2:  $k \#* (m \#- n) = (k \#* m) \#- (k \#* n)$ 
⟨proof⟩
```

27.2 Remainder

```
lemma div-termination: [ $0 < n$ ;  $n \text{ le } m$ ;  $m:\text{nat}$ ] ==>  $m \#- n < m$ 
⟨proof⟩
```

```
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*: $[[m:\text{nat}; n:\text{nat}]] \implies \text{raw-mod } (m, n) : \text{nat}$
<proof>

lemma *mod-type* $[TC,iff]$: $m \text{ mod } n : \text{nat}$
<proof>

lemma *DIVISION-BY-ZERO-DIV*: $a \text{ div } 0 = 0$
<proof>

lemma *DIVISION-BY-ZERO-MOD*: $a \text{ mod } 0 = \text{nativify}(a)$
<proof>

lemma *raw-mod-less*: $m < n \implies \text{raw-mod } (m, n) = m$
<proof>

lemma *mod-less* $[simp]$: $[[m < n; n : \text{nat}]] \implies m \text{ mod } n = m$
<proof>

lemma *raw-mod-geq*:
 $[[0 < n; n \text{ le } m; m : \text{nat}]] \implies \text{raw-mod } (m, n) = \text{raw-mod } (m\#-n, n)$
<proof>

lemma *mod-geq*: $[[n \text{ le } m; m : \text{nat}]] \implies m \text{ mod } n = (m\#-n) \text{ mod } n$
<proof>

27.3 Division

lemma *raw-div-type*: $[[m:\text{nat}; n:\text{nat}]] \implies \text{raw-div } (m, n) : \text{nat}$
<proof>

lemma *div-type* $[TC,iff]$: $m \text{ div } n : \text{nat}$
<proof>

lemma *raw-div-less*: $m < n \implies \text{raw-div } (m, n) = 0$
<proof>

lemma *div-less* $[simp]$: $[[m < n; n : \text{nat}]] \implies m \text{ div } n = 0$
<proof>

lemma *raw-div-geq*: $[[0 < n; n \text{ le } m; m : \text{nat}]] \implies \text{raw-div}(m, n) = \text{succ}(\text{raw-div}(m\#-n, n))$
<proof>

lemma *div-geq* $[simp]$:

$\llbracket 0 < n; n \leq m; m : \text{nat} \rrbracket \implies m \text{ div } n = \text{succ } ((m \# - n) \text{ div } n)$
 $\langle \text{proof} \rangle$

declare *div-less* [simp] *div-geq* [simp]

lemma *mod-div-lemma*: $\llbracket m : \text{nat}; n : \text{nat} \rrbracket \implies (m \text{ div } n) \# * n \# + m \text{ mod } n = m$
 $\langle \text{proof} \rangle$

lemma *mod-div-equality-natify*: $(m \text{ div } n) \# * n \# + m \text{ mod } n = \text{natify}(m)$
 $\langle \text{proof} \rangle$

lemma *mod-div-equality*: $m : \text{nat} \implies (m \text{ div } n) \# * n \# + m \text{ mod } n = m$
 $\langle \text{proof} \rangle$

27.4 Further Facts about Remainder

(mainly for mutilated chess board)

lemma *mod-succ-lemma*:
 $\llbracket 0 < n; m : \text{nat}; n : \text{nat} \rrbracket$
 $\implies \text{succ}(m) \text{ mod } n = (\text{if } \text{succ}(m \text{ mod } n) = n \text{ then } 0 \text{ else } \text{succ}(m \text{ mod } n))$
 $\langle \text{proof} \rangle$

lemma *mod-succ*:
 $n : \text{nat} \implies \text{succ}(m) \text{ mod } n = (\text{if } \text{succ}(m \text{ mod } n) = n \text{ then } 0 \text{ else } \text{succ}(m \text{ mod } n))$
 $\langle \text{proof} \rangle$

lemma *mod-less-divisor*: $\llbracket 0 < n; n : \text{nat} \rrbracket \implies m \text{ mod } n < n$
 $\langle \text{proof} \rangle$

lemma *mod-1-eq* [simp]: $m \text{ mod } 1 = 0$
 $\langle \text{proof} \rangle$

lemma *mod2-cases*: $b < 2 \implies k \text{ mod } 2 = b \mid k \text{ mod } 2 = (\text{if } b = 1 \text{ then } 0 \text{ else } 1)$
 $\langle \text{proof} \rangle$

lemma *mod2-succ-succ* [simp]: $\text{succ}(\text{succ}(m)) \text{ mod } 2 = m \text{ mod } 2$
 $\langle \text{proof} \rangle$

lemma *mod2-add-more* [simp]: $(m \# + m \# + n) \text{ mod } 2 = n \text{ mod } 2$
 $\langle \text{proof} \rangle$

lemma *mod2-add-self* [simp]: $(m \# + m) \text{ mod } 2 = 0$
 $\langle \text{proof} \rangle$

27.5 Additional theorems about \leq

lemma *add-le-self*: $m:\text{nat} \implies m \text{ le } (m \# + n)$
<proof>

lemma *add-le-self2*: $m:\text{nat} \implies m \text{ le } (n \# + m)$
<proof>

lemma *mult-le-mono1*: $[[i \text{ le } j; j:\text{nat}]] \implies (i \# * k) \text{ le } (j \# * k)$
<proof>

lemma *mult-le-mono*: $[[i \text{ le } j; k \text{ le } l; j:\text{nat}; l:\text{nat}]] \implies i \# * k \text{ le } j \# * l$
<proof>

lemma *mult-lt-mono2*: $[[i < j; 0 < k; j:\text{nat}; k:\text{nat}]] \implies k \# * i < k \# * j$
<proof>

lemma *mult-lt-mono1*: $[[i < j; 0 < k; j:\text{nat}; k:\text{nat}]] \implies i \# * k < j \# * k$
<proof>

lemma *add-eq-0-iff* [*iff*]: $m \# + n = 0 \iff \text{nativify}(m) = 0 \ \& \ \text{nativify}(n) = 0$
<proof>

lemma *zero-lt-mult-iff* [*iff*]: $0 < m \# * n \iff 0 < \text{nativify}(m) \ \& \ 0 < \text{nativify}(n)$
<proof>

lemma *mult-eq-1-iff* [*iff*]: $m \# * n = 1 \iff \text{nativify}(m) = 1 \ \& \ \text{nativify}(n) = 1$
<proof>

lemma *mult-is-zero*: $[[m:\text{nat}; n:\text{nat}]] \implies (m \# * n = 0) \iff (m = 0 \mid n = 0)$
<proof>

lemma *mult-is-zero-nativify* [*iff*]:
 $(m \# * n = 0) \iff (\text{nativify}(m) = 0 \mid \text{nativify}(n) = 0)$
<proof>

27.6 Cancellation Laws for Common Factors in Comparisons

lemma *mult-less-cancel-lemma*:
 $[[k:\text{nat}; m:\text{nat}; n:\text{nat}]] \implies (m \# * k < n \# * k) \iff (0 < k \ \& \ m < n)$
<proof>

lemma *mult-less-cancel2* [*simp*]:
 $(m \# * k < n \# * k) \iff (0 < \text{nativify}(k) \ \& \ \text{nativify}(m) < \text{nativify}(n))$

$\langle proof \rangle$

lemma *mult-less-cancel1* [simp]:

$$(k \# * m < k \# * n) <-> (0 < natify(k) \& natify(m) < natify(n))$$

$\langle proof \rangle$

lemma *mult-le-cancel2* [simp]: $(m \# * k \text{ le } n \# * k) <-> (0 < natify(k) \dashrightarrow natify(m) \text{ le } natify(n))$

$\langle proof \rangle$

lemma *mult-le-cancel1* [simp]: $(k \# * m \text{ le } k \# * n) <-> (0 < natify(k) \dashrightarrow natify(m) \text{ le } natify(n))$

$\langle proof \rangle$

lemma *mult-le-cancel-le1*: $k : nat ==> k \# * m \text{ le } k \longleftrightarrow (0 < k \longrightarrow natify(m) \text{ le } 1)$

$\langle proof \rangle$

lemma *Ord-eq-iff-le*: $[| Ord(m); Ord(n) |] ==> m = n <-> (m \text{ le } n \& n \text{ le } m)$

$\langle proof \rangle$

lemma *mult-cancel2-lemma*:

$$[| k : nat; m : nat; n : nat |] ==> (m \# * k = n \# * k) <-> (m = n \mid k = 0)$$

$\langle proof \rangle$

lemma *mult-cancel2* [simp]:

$$(m \# * k = n \# * k) <-> (natify(m) = natify(n) \mid natify(k) = 0)$$

$\langle proof \rangle$

lemma *mult-cancel1* [simp]:

$$(k \# * m = k \# * n) <-> (natify(m) = natify(n) \mid natify(k) = 0)$$

$\langle proof \rangle$

lemma *div-cancel-raw*:

$$[| 0 < n; 0 < k; k : nat; m : nat; n : nat |] ==> (k \# * m) \text{ div } (k \# * n) = m \text{ div } n$$

$\langle proof \rangle$

lemma *div-cancel*:

$$[| 0 < natify(n); 0 < natify(k) |] ==> (k \# * m) \text{ div } (k \# * n) = m \text{ div } n$$

$\langle proof \rangle$

27.7 More Lemmas about Remainder

lemma *mult-mod-distrib-raw*:

$$[| k : nat; m : nat; n : nat |] ==> (k \# * m) \text{ mod } (k \# * n) = k \# * (m \text{ mod } n)$$

$\langle proof \rangle$

lemma *mod-mult-distrib2*: $k \#* (m \text{ mod } n) = (k\#*m) \text{ mod } (k\#*n)$
 ⟨*proof*⟩

lemma *mult-mod-distrib*: $(m \text{ mod } n) \#* k = (m\#*k) \text{ mod } (n\#*k)$
 ⟨*proof*⟩

lemma *mod-add-self2-raw*: $n \in \text{nat} \implies (m \#+ n) \text{ mod } n = m \text{ mod } n$
 ⟨*proof*⟩

lemma *mod-add-self2* [*simp*]: $(m \#+ n) \text{ mod } n = m \text{ mod } n$
 ⟨*proof*⟩

lemma *mod-add-self1* [*simp*]: $(n\#+m) \text{ mod } n = m \text{ mod } n$
 ⟨*proof*⟩

lemma *mod-mult-self1-raw*: $k \in \text{nat} \implies (m \#+ k\#*n) \text{ mod } n = m \text{ mod } n$
 ⟨*proof*⟩

lemma *mod-mult-self1* [*simp*]: $(m \#+ k\#*n) \text{ mod } n = m \text{ mod } n$
 ⟨*proof*⟩

lemma *mod-mult-self2* [*simp*]: $(m \#+ n\#*k) \text{ mod } n = m \text{ mod } n$
 ⟨*proof*⟩

lemma *mult-eq-self-implies-10*: $m = m\#*n \implies \text{natisfy}(n)=1 \mid m=0$
 ⟨*proof*⟩

lemma *less-imp-succ-add* [*rule-format*]:
 $\llbracket m < n; n: \text{nat} \rrbracket \implies \text{EX } k: \text{nat}. n = \text{succ}(m\#+k)$
 ⟨*proof*⟩

lemma *less-iff-succ-add*:
 $\llbracket m: \text{nat}; n: \text{nat} \rrbracket \implies (m < n) \iff (\text{EX } k: \text{nat}. n = \text{succ}(m\#+k))$
 ⟨*proof*⟩

lemma *add-lt-elim2*:
 $\llbracket a \#+ d = b \#+ c; a < b; b \in \text{nat}; c \in \text{nat}; d \in \text{nat} \rrbracket \implies c < d$
 ⟨*proof*⟩

lemma *add-le-elim2*:
 $\llbracket a \#+ d = b \#+ c; a \text{ le } b; b \in \text{nat}; c \in \text{nat}; d \in \text{nat} \rrbracket \implies c \text{ le } d$
 ⟨*proof*⟩

27.7.1 More Lemmas About Difference

lemma *diff-is-0-lemma*:
 $\llbracket m: \text{nat}; n: \text{nat} \rrbracket \implies m \#- n = 0 \iff m \text{ le } n$

<proof>

lemma *diff-is-0-iff*: $m \#- n = 0 \leftrightarrow \text{natify}(m) = \text{natify}(n)$

<proof>

lemma *nat-lt-imp-diff-eq-0*:

$[[a:\text{nat}; b:\text{nat}; a < b]] \implies a \#- b = 0$

<proof>

lemma *raw-nat-diff-split*:

$[[a:\text{nat}; b:\text{nat}]] \implies$

$(P(a \#- b)) \leftrightarrow ((a < b \implies P(0)) \& (\text{ALL } d:\text{nat}. a = b \#+ d \implies$

$P(d)))$

<proof>

lemma *nat-diff-split*:

$(P(a \#- b)) \leftrightarrow$

$(\text{natify}(a) < \text{natify}(b) \implies P(0)) \& (\text{ALL } d:\text{nat}. \text{natify}(a) = b \#+ d \implies$

$P(d))$

<proof>

Difference and less-than

lemma *diff-lt-imp-lt*: $[[(k \#- i) < (k \#- j); i \in \text{nat}; j \in \text{nat}; k \in \text{nat}]] \implies j < i$

<proof>

lemma *lt-imp-diff-lt*: $[[j < i; i \leq k; k \in \text{nat}]] \implies (k \#- i) < (k \#- j)$

<proof>

lemma *diff-lt-iff-lt*: $[[i \leq k; j \in \text{nat}; k \in \text{nat}]] \implies (k \#- i) < (k \#- j) \leftrightarrow j < i$

<proof>

end

28 List-ZF: Lists in Zermelo-Fraenkel Set Theory

theory *List-ZF* **imports** *Datatype-ZF ArithSimp* **begin**

consts

list $:: i \implies i$

datatype

list(*A*) = *Nil* | *Cons* (*a*:*A*, *l*: *list*(*A*))

syntax

-*Nil* $:: i$ ($[]$)

-*List* $:: is \implies i$ ($[(-)]$)

translations

$$\begin{aligned} [x, xs] &== \text{CONST Cons}(x, [xs]) \\ [x] &== \text{CONST Cons}(x, []) \\ [] &== \text{CONST Nil} \end{aligned}$$
consts

$$\begin{aligned} \text{length} &:: i \Rightarrow i \\ \text{hd} &:: i \Rightarrow i \\ \text{tl} &:: i \Rightarrow i \end{aligned}$$
primrec

$$\begin{aligned} \text{length}([]) &= 0 \\ \text{length}(\text{Cons}(a,l)) &= \text{succ}(\text{length}(l)) \end{aligned}$$
primrec

$$\begin{aligned} \text{hd}([]) &= 0 \\ \text{hd}(\text{Cons}(a,l)) &= a \end{aligned}$$
primrec

$$\begin{aligned} \text{tl}([]) &= [] \\ \text{tl}(\text{Cons}(a,l)) &= l \end{aligned}$$
consts

$$\begin{aligned} \text{map} &:: [i \Rightarrow i, i] \Rightarrow i \\ \text{set-of-list} &:: i \Rightarrow i \\ \text{app} &:: [i, i] \Rightarrow i \quad (\text{infixr } @ \ 60) \end{aligned}$$
primrec

$$\begin{aligned} \text{map}(f, []) &= [] \\ \text{map}(f, \text{Cons}(a,l)) &= \text{Cons}(f(a), \text{map}(f,l)) \end{aligned}$$
primrec

$$\begin{aligned} \text{set-of-list}([]) &= 0 \\ \text{set-of-list}(\text{Cons}(a,l)) &= \text{cons}(a, \text{set-of-list}(l)) \end{aligned}$$
primrec

$$\begin{aligned} \text{app-Nil}: [] @ ys &= ys \\ \text{app-Cons}: (\text{Cons}(a,l)) @ ys &= \text{Cons}(a, l @ ys) \end{aligned}$$
consts

$$\begin{aligned} \text{rev} &:: i \Rightarrow i \\ \text{flat} &:: i \Rightarrow i \\ \text{list-add} &:: i \Rightarrow i \end{aligned}$$

primrec

$$\begin{aligned} rev(\[]) &= [] \\ rev(Cons(a,l)) &= rev(l) @ [a] \end{aligned}$$
primrec

$$\begin{aligned} flat(\[]) &= [] \\ flat(Cons(l,ls)) &= l @ flat(ls) \end{aligned}$$
primrec

$$\begin{aligned} list-add(\[]) &= 0 \\ list-add(Cons(a,l)) &= a \# + list-add(l) \end{aligned}$$
consts

$$drop \quad :: [i,i] \Rightarrow i$$
primrec

$$\begin{aligned} drop-0: \quad drop(0,l) &= l \\ drop-succ: \quad drop(succ(i), l) &= tl (drop(i,l)) \end{aligned}$$
definition

$$\begin{aligned} take \quad &:: [i,i] \Rightarrow i \text{ \textbf{where}} \\ take(n, as) &== list-rec(lam n:nat. [], \\ &\quad \%a l r. lam n:nat. nat-case([], \%m. Cons(a, r'm), n), as) 'n \end{aligned}$$
definition

$$\begin{aligned} nth \quad &:: [i, i] \Rightarrow i \text{ \textbf{where}} \\ \text{---} &\text{ returns the (n+1)th element of a list, or 0 if the list is too short.} \\ nth(n, as) &== list-rec(lam n:nat. 0, \\ &\quad \%a l r. lam n:nat. nat-case(a, \%m. r'm, n), as) 'n \end{aligned}$$
definition

$$\begin{aligned} list-update \quad &:: [i, i, i] \Rightarrow i \text{ \textbf{where}} \\ list-update(xs, i, v) &== list-rec(lam n:nat. Nil, \\ &\quad \%u us vs. lam n:nat. nat-case(Cons(v, us), \%m. Cons(u, vs'm), n), xs) 'i \end{aligned}$$
consts

$$\begin{aligned} filter \quad &:: [i \Rightarrow o, i] \Rightarrow i \\ upt \quad &:: [i, i] \Rightarrow i \end{aligned}$$
primrec

$$\begin{aligned} filter(P, Nil) &= Nil \\ filter(P, Cons(x, xs)) &= \\ &\quad (if P(x) then Cons(x, filter(P, xs)) else filter(P, xs)) \end{aligned}$$
primrec

$upt(i, 0) = Nil$
 $upt(i, succ(j)) = (if\ i\ le\ j\ then\ upt(i, j)@[j]\ else\ Nil)$

definition

$min :: [i, i] => i$ **where**
 $min(x, y) == (if\ x\ le\ y\ then\ x\ else\ y)$

definition

$max :: [i, i] => i$ **where**
 $max(x, y) == (if\ x\ le\ y\ then\ y\ else\ x)$

declare *list.intros* [*simp, TC*]

inductive-cases *ConsE*: $Cons(a, l) : list(A)$

lemma *Cons-type-iff* [*simp*]: $Cons(a, l) \in list(A) \leftrightarrow a \in A \ \& \ l \in list(A)$
 $\langle proof \rangle$

lemma *Cons-iff*: $Cons(a, l) = Cons(a', l') \leftrightarrow a = a' \ \& \ l = l'$
 $\langle proof \rangle$

lemma *Nil-Cons-iff*: $\sim Nil = Cons(a, l)$
 $\langle proof \rangle$

lemma *list-unfold*: $list(A) = \{0\} + (A * list(A))$
 $\langle proof \rangle$

lemma *list-mono*: $A \leq B \implies list(A) \leq list(B)$
 $\langle proof \rangle$

lemma *list-univ*: $list(univ(A)) \leq univ(A)$
 $\langle proof \rangle$

lemmas *list-subset-univ* = *subset-trans* [*OF list-mono list-univ*]

lemma *list-into-univ*: $[[\ l : list(A); \ A \leq univ(B) \]] \implies l : univ(B)$
 $\langle proof \rangle$

lemma *list-case-type*:
 $[[\ l : list(A);$

$c: C(\text{Nil});$
 $!!x y. [! x: A; y: \text{list}(A)] \implies h(x,y): C(\text{Cons}(x,y))$
 $[!] \implies \text{list-case}(c,h,l) : C(l)$
 $\langle \text{proof} \rangle$

lemma *list-0-triv*: $\text{list}(0) = \{\text{Nil}\}$
 $\langle \text{proof} \rangle$

lemma *tl-type*: $l: \text{list}(A) \implies \text{tl}(l) : \text{list}(A)$
 $\langle \text{proof} \rangle$

lemma *drop-Nil* [*simp*]: $i: \text{nat} \implies \text{drop}(i, \text{Nil}) = \text{Nil}$
 $\langle \text{proof} \rangle$

lemma *drop-succ-Cons* [*simp*]: $i: \text{nat} \implies \text{drop}(\text{succ}(i), \text{Cons}(a,l)) = \text{drop}(i,l)$
 $\langle \text{proof} \rangle$

lemma *drop-type* [*simp, TC*]: $[! i: \text{nat}; l: \text{list}(A)] \implies \text{drop}(i,l) : \text{list}(A)$
 $\langle \text{proof} \rangle$

declare *drop-succ* [*simp del*]

lemma *list-rec-type* [*TC*]:
 $[! l: \text{list}(A);$
 $c: C(\text{Nil});$
 $!!x y r. [! x: A; y: \text{list}(A); r: C(y)] \implies h(x,y,r): C(\text{Cons}(x,y))$
 $[!] \implies \text{list-rec}(c,h,l) : C(l)$
 $\langle \text{proof} \rangle$

lemma *map-type* [*TC*]:
 $[! l: \text{list}(A); !!x. x: A \implies h(x): B] \implies \text{map}(h,l) : \text{list}(B)$
 $\langle \text{proof} \rangle$

lemma *map-type2* [*TC*]: $l: \text{list}(A) \implies \text{map}(h,l) : \text{list}(\{h(u). u:A\})$
 $\langle \text{proof} \rangle$

lemma *length-type* [*TC*]: $l: \text{list}(A) \implies \text{length}(l) : \text{nat}$

<proof>

lemma *lt-length-in-nat*:

$[[x < \text{length}(xs); xs \in \text{list}(A)]] \implies x \in \text{nat}$
<proof>

lemma *app-type* [TC]: $[[xs: \text{list}(A); ys: \text{list}(A)]] \implies xs@ys : \text{list}(A)$
<proof>

lemma *rev-type* [TC]: $xs: \text{list}(A) \implies \text{rev}(xs) : \text{list}(A)$
<proof>

lemma *flat-type* [TC]: $ls: \text{list}(\text{list}(A)) \implies \text{flat}(ls) : \text{list}(A)$
<proof>

lemma *set-of-list-type* [TC]: $l: \text{list}(A) \implies \text{set-of-list}(l) : \text{Pow}(A)$
<proof>

lemma *set-of-list-append*:

$xs: \text{list}(A) \implies \text{set-of-list}(xs@ys) = \text{set-of-list}(xs) \cup \text{set-of-list}(ys)$
<proof>

lemma *list-add-type* [TC]: $xs: \text{list}(\text{nat}) \implies \text{list-add}(xs) : \text{nat}$
<proof>

lemma *map-ident* [simp]: $l: \text{list}(A) \implies \text{map}(\%u. u, l) = l$
<proof>

lemma *map-compose*: $l: \text{list}(A) \implies \text{map}(h, \text{map}(j, l)) = \text{map}(\%u. h(j(u)), l)$
<proof>

lemma *map-app-distrib*: $xs: \text{list}(A) \implies \text{map}(h, xs@ys) = \text{map}(h, xs) @ \text{map}(h, ys)$
<proof>

lemma *map-flat*: $ls: list(list(A)) \implies map(h, flat(ls)) = flat(map(map(h),ls))$
 ⟨proof⟩

lemma *list-rec-map*:
 $l: list(A) \implies$
 $list-rec(c, d, map(h,l)) =$
 $list-rec(c, \forall x xs r. d(h(x), map(h,xs), r), l)$
 ⟨proof⟩

lemmas *list-CollectD* = *Collect-subset* [THEN *list-mono*, THEN *subsetD*, *standard*]

lemma *map-list-Collect*: $l: list(\{x:A. h(x)=j(x)\}) \implies map(h,l) = map(j,l)$
 ⟨proof⟩

lemma *length-map* [*simp*]: $xs: list(A) \implies length(map(h,xs)) = length(xs)$
 ⟨proof⟩

lemma *length-app* [*simp*]:
 $[[xs: list(A); ys: list(A)]]$
 $\implies length(xs@ys) = length(xs) \# + length(ys)$
 ⟨proof⟩

lemma *length-rev* [*simp*]: $xs: list(A) \implies length(rev(xs)) = length(xs)$
 ⟨proof⟩

lemma *length-flat*:
 $ls: list(list(A)) \implies length(flat(ls)) = list-add(map(length,ls))$
 ⟨proof⟩

lemma *drop-length-Cons* [*rule-format*]:
 $xs: list(A) \implies$
 $\forall x. EX z zs. drop(length(xs), Cons(x,xs)) = Cons(z,zs)$
 ⟨proof⟩

lemma *drop-length* [*rule-format*]:
 $l: list(A) \implies \forall i \in length(l). (EX z zs. drop(i,l) = Cons(z,zs))$
 ⟨proof⟩

lemma *app-right-Nil* [*simp*]: $xs: list(A) ==> xs@Nil=xs$
 ⟨*proof*⟩

lemma *app-assoc*: $xs: list(A) ==> (xs@ys)@zs = xs@(ys@zs)$
 ⟨*proof*⟩

lemma *flat-app-distrib*: $ls: list(list(A)) ==> flat(ls@ms) = flat(ls)@flat(ms)$
 ⟨*proof*⟩

lemma *rev-map-distrib*: $l: list(A) ==> rev(map(h,l)) = map(h,rev(l))$
 ⟨*proof*⟩

lemma *rev-app-distrib*:
 $[[xs: list(A); ys: list(A)]] ==> rev(xs@ys) = rev(ys)@rev(xs)$
 ⟨*proof*⟩

lemma *rev-rev-ident* [*simp*]: $l: list(A) ==> rev(rev(l))=l$
 ⟨*proof*⟩

lemma *rev-flat*: $ls: list(list(A)) ==> rev(flat(ls)) = flat(map(rev,rev(ls)))$
 ⟨*proof*⟩

lemma *list-add-app*:
 $[[xs: list(nat); ys: list(nat)]] ==> list-add(xs@ys) = list-add(ys) \#+ list-add(xs)$
 ⟨*proof*⟩

lemma *list-add-rev*: $l: list(nat) ==> list-add(rev(l)) = list-add(l)$
 ⟨*proof*⟩

lemma *list-add-flat*:
 $ls: list(list(nat)) ==> list-add(flat(ls)) = list-add(map(list-add,ls))$
 ⟨*proof*⟩

lemma *list-append-induct* [*case-names Nil snoc, consumes 1*]:
 $[[l: list(A); P(Nil); !!x y. [[x: A; y: list(A); P(y)]] ==> P(y @ [x])]] ==> P(l)$

<proof>

lemma *list-complete-induct-lemma* [rule-format]:

assumes *ih*:

$\bigwedge l. [\![l \in \text{list}(A);$
 $\quad \forall l' \in \text{list}(A). \text{length}(l') < \text{length}(l) \longrightarrow P(l')]\!]$
 $\implies P(l)$

shows $n \in \text{nat} \implies \forall l \in \text{list}(A). \text{length}(l) < n \longrightarrow P(l)$
<proof>

theorem *list-complete-induct*:

$[\![l \in \text{list}(A);$
 $\quad \bigwedge l. [\![l \in \text{list}(A);$
 $\quad \quad \forall l' \in \text{list}(A). \text{length}(l') < \text{length}(l) \longrightarrow P(l')]\!]$
 $\quad \implies P(l)$
 $\!]\!]$ $\implies P(l)$
<proof>

lemma *min-sym*: $[\![i:\text{nat}; j:\text{nat}]\!] \implies \text{min}(i,j)=\text{min}(j,i)$
<proof>

lemma *min-type* [simp,TC]: $[\![i:\text{nat}; j:\text{nat}]\!] \implies \text{min}(i,j):\text{nat}$
<proof>

lemma *min-0* [simp]: $i:\text{nat} \implies \text{min}(0,i) = 0$
<proof>

lemma *min-02* [simp]: $i:\text{nat} \implies \text{min}(i, 0) = 0$
<proof>

lemma *lt-min-iff*: $[\![i:\text{nat}; j:\text{nat}; k:\text{nat}]\!] \implies i < \text{min}(j,k) \longleftrightarrow i < j \ \& \ i < k$
<proof>

lemma *min-succ-succ* [simp]:
 $[\![i:\text{nat}; j:\text{nat}]\!] \implies \text{min}(\text{succ}(i), \text{succ}(j)) = \text{succ}(\text{min}(i, j))$
<proof>

lemma *filter-append* [simp]:
 $xs:\text{list}(A) \implies \text{filter}(P, xs@ys) = \text{filter}(P, xs) @ \text{filter}(P, ys)$
<proof>

lemma *filter-type* [simp,TC]: $xs:list(A) \implies filter(P, xs):list(A)$
 <proof>

lemma *length-filter*: $xs:list(A) \implies length(filter(P, xs)) \leq length(xs)$
 <proof>

lemma *filter-is-subset*: $xs:list(A) \implies set-of-list(filter(P, xs)) \subseteq set-of-list(xs)$
 <proof>

lemma *filter-False* [simp]: $xs:list(A) \implies filter(\%p. False, xs) = Nil$
 <proof>

lemma *filter-True* [simp]: $xs:list(A) \implies filter(\%p. True, xs) = xs$
 <proof>

lemma *length-is-0-iff* [simp]: $xs:list(A) \implies length(xs)=0 \iff xs=Nil$
 <proof>

lemma *length-is-0-iff2* [simp]: $xs:list(A) \implies 0 = length(xs) \iff xs=Nil$
 <proof>

lemma *length-tl* [simp]: $xs:list(A) \implies length(tl(xs)) = length(xs) - 1$
 <proof>

lemma *length-greater-0-iff*: $xs:list(A) \implies 0 < length(xs) \iff xs \neq Nil$
 <proof>

lemma *length-succ-iff*: $xs:list(A) \implies length(xs)=succ(n) \iff (EX y ys. xs=Cons(y, ys) \ \& \ length(ys)=n)$
 <proof>

lemma *append-is-Nil-iff* [simp]:
 $xs:list(A) \implies (xs@ys = Nil) \iff (xs=Nil \ \& \ ys = Nil)$
 <proof>

lemma *append-is-Nil-iff2* [simp]:
 $xs:list(A) \implies (Nil = xs@ys) \iff (xs=Nil \ \& \ ys = Nil)$
 <proof>

lemma *append-left-is-self-iff* [simp]:
 $xs:list(A) \implies (xs@ys = xs) \iff (ys = Nil)$
 <proof>

lemma *append-left-is-self-iff2* [simp]:

$xs:list(A) ==> (xs = xs@ys) <-> (ys = Nil)$
 ⟨proof⟩

lemma *append-left-is-Nil-iff* [rule-format]:

$[| xs:list(A); ys:list(A); zs:list(A) |] ==>$
 $length(ys)=length(zs) \dashrightarrow (xs@ys=zs <-> (xs=Nil \& ys=zs))$
 ⟨proof⟩

lemma *append-left-is-Nil-iff2* [rule-format]:

$[| xs:list(A); ys:list(A); zs:list(A) |] ==>$
 $length(ys)=length(zs) \dashrightarrow (zs=ys@xs <-> (xs=Nil \& ys=zs))$
 ⟨proof⟩

lemma *append-eq-append-iff* [rule-format,simp]:

$xs:list(A) ==> \forall ys \in list(A).$
 $length(xs)=length(ys) \dashrightarrow (xs@us = ys@us) <-> (xs=ys \& us=us)$
 ⟨proof⟩

lemma *append-eq-append* [rule-format]:

$xs:list(A) ==>$
 $\forall ys \in list(A). \forall us \in list(A). \forall vs \in list(A).$
 $length(us) = length(vs) \dashrightarrow (xs@us = ys@vs) \dashrightarrow (xs=ys \& us=vs)$
 ⟨proof⟩

lemma *append-eq-append-iff2* [simp]:

$[| xs:list(A); ys:list(A); us:list(A); vs:list(A); length(us)=length(vs) |]$
 $==> xs@us = ys@vs <-> (xs=ys \& us=vs)$
 ⟨proof⟩

lemma *append-self-iff* [simp]:

$[| xs:list(A); ys:list(A); zs:list(A) |] ==> xs@ys=xs@zs <-> ys=zs$
 ⟨proof⟩

lemma *append-self-iff2* [simp]:

$[| xs:list(A); ys:list(A); zs:list(A) |] ==> ys@xs=zs@xs <-> ys=zs$
 ⟨proof⟩

lemma *append1-eq-iff* [rule-format,simp]:

$xs:list(A) ==> \forall ys \in list(A). xs@[x] = ys@[y] <-> (xs = ys \& x=y)$
 ⟨proof⟩

lemma *append-right-is-self-iff* [simp]:

$[| xs:list(A); ys:list(A) |] ==> (xs@ys = ys) <-> (xs=Nil)$
 ⟨proof⟩

lemma *append-right-is-self-iff2* [*simp*]:
 $[[xs:list(A); ys:list(A)]] ==> (ys = xs@ys) <-> (xs=Nil)$
 ⟨*proof*⟩

lemma *hd-append* [*rule-format,simp*]:
 $xs:list(A) ==> xs \sim Nil \dashrightarrow hd(xs @ ys) = hd(xs)$
 ⟨*proof*⟩

lemma *tl-append* [*rule-format,simp*]:
 $xs:list(A) ==> xs \sim Nil \dashrightarrow tl(xs @ ys) = tl(xs)@ys$
 ⟨*proof*⟩

lemma *rev-is-Nil-iff* [*simp*]: $xs:list(A) ==> (rev(xs) = Nil <-> xs = Nil)$
 ⟨*proof*⟩

lemma *Nil-is-rev-iff* [*simp*]: $xs:list(A) ==> (Nil = rev(xs) <-> xs = Nil)$
 ⟨*proof*⟩

lemma *rev-is-rev-iff* [*rule-format,simp*]:
 $xs:list(A) ==> \forall ys \in list(A). rev(xs)=rev(ys) <-> xs=ys$
 ⟨*proof*⟩

lemma *rev-list-elim* [*rule-format*]:
 $xs:list(A) ==>$
 $(xs=Nil \dashrightarrow P) \dashrightarrow (\forall ys \in list(A). \forall y \in A. xs = ys@[y] \dashrightarrow P) \dashrightarrow P$
 ⟨*proof*⟩

lemma *length-drop* [*rule-format,simp*]:
 $n:nat ==> \forall xs \in list(A). length(drop(n, xs)) = length(xs) \#- n$
 ⟨*proof*⟩

lemma *drop-all* [*rule-format,simp*]:
 $n:nat ==> \forall xs \in list(A). length(xs) le n \dashrightarrow drop(n, xs)=Nil$
 ⟨*proof*⟩

lemma *drop-append* [*rule-format*]:
 $n:nat ==>$
 $\forall xs \in list(A). drop(n, xs@ys) = drop(n,xs) @ drop(n \#- length(xs), ys)$
 ⟨*proof*⟩

lemma *drop-drop*:
 $m:nat ==> \forall xs \in list(A). \forall n \in nat. drop(n, drop(m, xs))=drop(n \#+ m,$
 $xs)$
 ⟨*proof*⟩

lemma *take-0* [*simp*]: $xs: \text{list}(A) \implies \text{take}(0, xs) = \text{Nil}$
 ⟨*proof*⟩

lemma *take-succ-Cons* [*simp*]:
 $n: \text{nat} \implies \text{take}(\text{succ}(n), \text{Cons}(a, xs)) = \text{Cons}(a, \text{take}(n, xs))$
 ⟨*proof*⟩

lemma *take-Nil* [*simp*]: $n: \text{nat} \implies \text{take}(n, \text{Nil}) = \text{Nil}$
 ⟨*proof*⟩

lemma *take-all* [*rule-format, simp*]:
 $n: \text{nat} \implies \forall xs \in \text{list}(A). \text{length}(xs) \text{ le } n \implies \text{take}(n, xs) = xs$
 ⟨*proof*⟩

lemma *take-type* [*rule-format, simp, TC*]:
 $xs: \text{list}(A) \implies \forall n \in \text{nat}. \text{take}(n, xs): \text{list}(A)$
 ⟨*proof*⟩

lemma *take-append* [*rule-format, simp*]:
 $xs: \text{list}(A) \implies$
 $\forall ys \in \text{list}(A). \forall n \in \text{nat}. \text{take}(n, xs @ ys) =$
 $\text{take}(n, xs) @ \text{take}(n \#- \text{length}(xs), ys)$
 ⟨*proof*⟩

lemma *take-take* [*rule-format*]:
 $m : \text{nat} \implies$
 $\forall xs \in \text{list}(A). \forall n \in \text{nat}. \text{take}(n, \text{take}(m, xs)) = \text{take}(\min(n, m), xs)$
 ⟨*proof*⟩

lemma *nth-0* [*simp*]: $\text{nth}(0, \text{Cons}(a, l)) = a$
 ⟨*proof*⟩

lemma *nth-Cons* [*simp*]: $n: \text{nat} \implies \text{nth}(\text{succ}(n), \text{Cons}(a, l)) = \text{nth}(n, l)$
 ⟨*proof*⟩

lemma *nth-empty* [*simp*]: $\text{nth}(n, \text{Nil}) = 0$
 ⟨*proof*⟩

lemma *nth-type* [*rule-format, simp, TC*]:
 $xs: \text{list}(A) \implies \forall n. n < \text{length}(xs) \implies \text{nth}(n, xs) : A$
 ⟨*proof*⟩

lemma *nth-eq-0* [*rule-format*]:
 $xs: \text{list}(A) \implies \forall n \in \text{nat}. \text{length}(xs) \text{ le } n \implies \text{nth}(n, xs) = 0$

$\langle proof \rangle$

lemma *nth-append* [rule-format]:

$xs: list(A) ==>$
 $\forall n \in nat. nth(n, xs @ ys) = (if\ n < length(xs)\ then\ nth(n, xs)$
 $\quad\quad\quad else\ nth(n \# - length(xs), ys))$

$\langle proof \rangle$

lemma *set-of-list-conv-nth*:

$xs: list(A)$
 $==> set-of-list(xs) = \{x:A. EX\ i:nat. i < length(xs) \ \&\ x = nth(i, xs)\}$

$\langle proof \rangle$

lemma *nth-take-lemma* [rule-format]:

$k:nat ==>$
 $\forall xs \in list(A). (\forall ys \in list(A). k \leq length(xs) \ \rightarrow\ k \leq length(ys) \ \rightarrow$
 $\quad (\forall i \in nat. i < k \ \rightarrow\ nth(i, xs) = nth(i, ys)) \ \rightarrow\ take(k, xs) = take(k, ys))$
 $\langle proof \rangle$

lemma *nth-equalityI* [rule-format]:

$[[\ xs: list(A);\ ys: list(A);\ length(xs) = length(ys);$
 $\quad \forall i \in nat. i < length(xs) \ \rightarrow\ nth(i, xs) = nth(i, ys) \]]$
 $==> xs = ys$

$\langle proof \rangle$

lemma *take-equalityI* [rule-format]:

$[[\ xs: list(A);\ ys: list(A);\ (\forall i \in nat. take(i, xs) = take(i, ys)) \]]$
 $==> xs = ys$

$\langle proof \rangle$

lemma *nth-drop* [rule-format]:

$n:nat ==> \forall i \in nat. \forall xs \in list(A). nth(i, drop(n, xs)) = nth(n \# + i, xs)$
 $\langle proof \rangle$

lemma *take-succ* [rule-format]:

$xs \in list(A)$
 $==> \forall i. i < length(xs) \ \rightarrow\ take(succ(i), xs) = take(i, xs) @ [nth(i, xs)]$
 $\langle proof \rangle$

lemma *take-add* [rule-format]:

$[[xs \in list(A);\ j \in nat]]$
 $==> \forall i \in nat. take(i \# + j, xs) = take(i, xs) @ take(j, drop(i, xs))$
 $\langle proof \rangle$

lemma *length-take*:

$l \in \text{list}(A) \implies \forall n \in \text{nat}. \text{length}(\text{take}(n, l)) = \min(n, \text{length}(l))$
 <proof>

28.1 The function zip

Crafty definition to eliminate a type argument

consts

$\text{zip-aux} \quad :: [i, i] \Rightarrow i$

primrec

$\text{zip-aux}(B, []) =$
 $(\lambda ys \in \text{list}(B). \text{list-case}([], \%y l. [], ys))$

$\text{zip-aux}(B, \text{Cons}(x, l)) =$
 $(\lambda ys \in \text{list}(B). \text{list-case}(\text{Nil}, \%y zs. \text{Cons}(\langle x, y \rangle, \text{zip-aux}(B, l) 'zs), ys))$

definition

$\text{zip} :: [i, i] \Rightarrow i$ **where**
 $\text{zip}(xs, ys) == \text{zip-aux}(\text{set-of-list}(ys), xs) 'ys$

lemma *list-on-set-of-list*: $xs \in \text{list}(A) \implies xs \in \text{list}(\text{set-of-list}(xs))$
 <proof>

lemma *zip-Nil* [*simp*]: $ys : \text{list}(A) \implies \text{zip}(\text{Nil}, ys) = \text{Nil}$
 <proof>

lemma *zip-Nil2* [*simp*]: $xs : \text{list}(A) \implies \text{zip}(xs, \text{Nil}) = \text{Nil}$
 <proof>

lemma *zip-aux-unique* [*rule-format*]:

$[[B \leq C; xs \in \text{list}(A)]]$
 $\implies \forall ys \in \text{list}(B). \text{zip-aux}(C, xs) 'ys = \text{zip-aux}(B, xs) 'ys$
 <proof>

lemma *zip-Cons-Cons* [*simp*]:

$[[xs : \text{list}(A); ys : \text{list}(B); x : A; y : B]] \implies$
 $\text{zip}(\text{Cons}(x, xs), \text{Cons}(y, ys)) = \text{Cons}(\langle x, y \rangle, \text{zip}(xs, ys))$
 <proof>

lemma *zip-type* [*rule-format, simp, TC*]:

$xs : \text{list}(A) \implies \forall ys \in \text{list}(B). \text{zip}(xs, ys) : \text{list}(A * B)$
 <proof>

lemma *length-zip* [*rule-format, simp*]:

$$xs:list(A) ==> \forall ys \in list(B). length(zip(xs,ys)) = \min(length(xs), length(ys))$$

<proof>

lemma *zip-append1* [rule-format]:

$$\begin{aligned} &[[ys:list(A); zs:list(B)]] ==> \\ &\forall xs \in list(A). zip(xs @ ys, zs) = \\ &\quad zip(xs, take(length(xs), zs)) @ zip(ys, drop(length(xs),zs)) \end{aligned}$$

<proof>

lemma *zip-append2* [rule-format]:

$$\begin{aligned} &[[xs:list(A); zs:list(B)]] ==> \forall ys \in list(B). zip(xs, ys@zs) = \\ &\quad zip(take(length(ys), xs), ys) @ zip(drop(length(ys), xs), zs) \end{aligned}$$

<proof>

lemma *zip-append* [simp]:

$$\begin{aligned} &[[length(xs) = length(us); length(ys) = length(vs); \\ &\quad xs:list(A); us:list(B); ys:list(A); vs:list(B)]] \\ &==> zip(xs@ys,us@vs) = zip(xs, us) @ zip(ys, vs) \end{aligned}$$

<proof>

lemma *zip-rev* [rule-format,simp]:

$$\begin{aligned} &ys:list(B) ==> \forall xs \in list(A). \\ &\quad length(xs) = length(ys) --> zip(rev(xs), rev(ys)) = rev(zip(xs, ys)) \end{aligned}$$

<proof>

lemma *nth-zip* [rule-format,simp]:

$$\begin{aligned} &ys:list(B) ==> \forall i \in nat. \forall xs \in list(A). \\ &\quad i < length(xs) --> i < length(ys) --> \\ &\quad nth(i,zip(xs, ys)) = <nth(i,xs),nth(i, ys)> \end{aligned}$$

<proof>

lemma *set-of-list-zip* [rule-format]:

$$\begin{aligned} &[[xs:list(A); ys:list(B); i:nat]] \\ &==> set-of-list(zip(xs, ys)) = \\ &\quad \{<x, y>:A*B. EX i:nat. i < \min(length(xs), length(ys)) \\ &\quad \& x = nth(i, xs) \& y = nth(i, ys)\} \end{aligned}$$

<proof>

lemma *list-update-Nil* [simp]: $i:nat ==> list-update(Nil, i, v) = Nil$

<proof>

lemma *list-update-Cons-0* [simp]: $list-update(Cons(x, xs), 0, v) = Cons(v, xs)$

<proof>

lemma *list-update-Cons-succ* [simp]:

$n:\text{nat} \implies$
 $\text{list-update}(\text{Cons}(x, xs), \text{succ}(n), v) = \text{Cons}(x, \text{list-update}(xs, n, v))$
 $\langle \text{proof} \rangle$

lemma *list-update-type* [rule-format,simp,TC]:
 $[[xs:\text{list}(A); v:A]] \implies \forall n \in \text{nat}. \text{list-update}(xs, n, v):\text{list}(A)$
 $\langle \text{proof} \rangle$

lemma *length-list-update* [rule-format,simp]:
 $xs:\text{list}(A) \implies \forall i \in \text{nat}. \text{length}(\text{list-update}(xs, i, v)) = \text{length}(xs)$
 $\langle \text{proof} \rangle$

lemma *nth-list-update* [rule-format]:
 $[[xs:\text{list}(A)]] \implies \forall i \in \text{nat}. \forall j \in \text{nat}. i < \text{length}(xs) \dashrightarrow$
 $\text{nth}(j, \text{list-update}(xs, i, x)) = (\text{if } i=j \text{ then } x \text{ else } \text{nth}(j, xs))$
 $\langle \text{proof} \rangle$

lemma *nth-list-update-eq* [simp]:
 $[[i < \text{length}(xs); xs:\text{list}(A)]] \implies \text{nth}(i, \text{list-update}(xs, i, x)) = x$
 $\langle \text{proof} \rangle$

lemma *nth-list-update-neq* [rule-format,simp]:
 $xs:\text{list}(A) \implies$
 $\forall i \in \text{nat}. \forall j \in \text{nat}. i \sim = j \dashrightarrow \text{nth}(j, \text{list-update}(xs, i, x)) = \text{nth}(j, xs)$
 $\langle \text{proof} \rangle$

lemma *list-update-overwrite* [rule-format,simp]:
 $xs:\text{list}(A) \implies \forall i \in \text{nat}. i < \text{length}(xs)$
 $\dashrightarrow \text{list-update}(\text{list-update}(xs, i, x), i, y) = \text{list-update}(xs, i, y)$
 $\langle \text{proof} \rangle$

lemma *list-update-same-conv* [rule-format]:
 $xs:\text{list}(A) \implies$
 $\forall i \in \text{nat}. i < \text{length}(xs) \dashrightarrow$
 $(\text{list-update}(xs, i, x) = xs) \leftrightarrow (\text{nth}(i, xs) = x)$
 $\langle \text{proof} \rangle$

lemma *update-zip* [rule-format]:
 $ys:\text{list}(B) \implies$
 $\forall i \in \text{nat}. \forall xy \in A*B. \forall xs \in \text{list}(A).$
 $\text{length}(xs) = \text{length}(ys) \dashrightarrow$
 $\text{list-update}(\text{zip}(xs, ys), i, xy) = \text{zip}(\text{list-update}(xs, i, \text{fst}(xy)),$
 $\text{list-update}(ys, i, \text{snd}(xy)))$
 $\langle \text{proof} \rangle$

lemma *set-update-subset-cons* [rule-format]:
 $xs:\text{list}(A) \implies$
 $\forall i \in \text{nat}. \text{set-of-list}(\text{list-update}(xs, i, x)) \leq \text{cons}(x, \text{set-of-list}(xs))$

<proof>

lemma *set-of-list-update-subsetI*:

$[[\text{set-of-list}(xs) \leq A; xs:\text{list}(A); x:A; i:\text{nat}]]$
 $\implies \text{set-of-list}(\text{list-update}(xs, i, x)) \leq A$
<proof>

lemma *upt-rec*:

$j:\text{nat} \implies \text{upt}(i, j) = (\text{if } i < j \text{ then } \text{Cons}(i, \text{upt}(\text{succ}(i), j)) \text{ else } \text{Nil})$
<proof>

lemma *upt-conv-Nil* [*simp*]: $[[j \text{ le } i; j:\text{nat}]]$ $\implies \text{upt}(i, j) = \text{Nil}$
<proof>

lemma *upt-succ-append*:

$[[i \text{ le } j; j:\text{nat}]]$ $\implies \text{upt}(i, \text{succ}(j)) = \text{upt}(i, j) @ [j]$
<proof>

lemma *upt-conv-Cons*:

$[[i < j; j:\text{nat}]]$ $\implies \text{upt}(i, j) = \text{Cons}(i, \text{upt}(\text{succ}(i), j))$
<proof>

lemma *upt-type* [*simp, TC*]: $j:\text{nat} \implies \text{upt}(i, j) : \text{list}(\text{nat})$
<proof>

lemma *upt-add-eq-append*:

$[[i \text{ le } j; j:\text{nat}; k:\text{nat}]]$ $\implies \text{upt}(i, j \# + k) = \text{upt}(i, j) @ \text{upt}(j, j \# + k)$
<proof>

lemma *length-upt* [*simp*]: $[[i:\text{nat}; j:\text{nat}]]$ $\implies \text{length}(\text{upt}(i, j)) = j \# - i$
<proof>

lemma *nth-upt* [*rule-format, simp*]:

$[[i:\text{nat}; j:\text{nat}; k:\text{nat}]]$ $\implies i \# + k < j \dashrightarrow \text{nth}(k, \text{upt}(i, j)) = i \# + k$
<proof>

lemma *take-upt* [*rule-format, simp*]:

$[[m:\text{nat}; n:\text{nat}]]$ \implies
 $\forall i \in \text{nat}. i \# + m \text{ le } n \dashrightarrow \text{take}(m, \text{upt}(i, n)) = \text{upt}(i, i \# + m)$
<proof>

lemma *map-succ-upt*:

$[[m:\text{nat}; n:\text{nat}]]$ $\implies \text{map}(\text{succ}, \text{upt}(m, n)) = \text{upt}(\text{succ}(m), \text{succ}(n))$
<proof>

lemma *nth-map* [*rule-format,simp*]:

$xs:list(A) ==>$
 $\forall n \in nat. n < length(xs) \dashrightarrow nth(n, map(f, xs)) = f(nth(n, xs))$
(*proof*)

lemma *nth-map-upt* [*rule-format*]:

$[| m:nat; n:nat |] ==>$
 $\forall i \in nat. i < n \#- m \dashrightarrow nth(i, map(f, upt(m,n))) = f(m \#+ i)$
(*proof*)

definition

sublist :: [*i, i*] => *i* **where**
sublist(*xs, A*) ==
map(*fst, (filter(%p. snd(p): A, zip(xs, upt(0,length(xs))))*)

lemma *sublist-0* [*simp*]: $xs:list(A) ==> sublist(xs, 0) = Nil$
(*proof*)

lemma *sublist-Nil* [*simp*]: $sublist(Nil, A) = Nil$
(*proof*)

lemma *sublist-shift-lemma*:

$[| xs:list(B); i:nat |] ==>$
 $map(fst, filter(%p. snd(p):A, zip(xs, upt(i,i \#+ length(xs)))) =$
 $map(fst, filter(%p. snd(p):nat \& snd(p) \#+ i:A, zip(xs,upt(0,length(xs))))$
(*proof*)

lemma *sublist-type* [*simp,TC*]:

$xs:list(B) ==> sublist(xs, A):list(B)$
(*proof*)

lemma *upt-add-eq-append2*:

$[| i:nat; j:nat |] ==> upt(0, i \#+ j) = upt(0, i) @ upt(i, i \#+ j)$
(*proof*)

lemma *sublist-append*:

$[| xs:list(B); ys:list(B) |] ==>$
 $sublist(xs@ys, A) = sublist(xs, A) @ sublist(ys, \{j:nat. j \#+ length(xs): A\})$
(*proof*)

lemma *sublist-Cons*:

$[| xs:list(B); x:B |] ==>$
 $sublist(Cons(x, xs), A) =$
 $(if 0:A then [x] else []) @ sublist(xs, \{j:nat. succ(j) : A\})$
(*proof*)

lemma *sublist-singleton* [*simp*]:
 $sublist([x], A) = (if\ 0 : A\ then\ [x]\ else\ [])$
 ⟨*proof*⟩

lemma *sublist-upt-eq-take* [*rule-format, simp*]:
 $xs : list(A) ==> ALL\ n : nat.\ sublist(xs, n) = take(n, xs)$
 ⟨*proof*⟩

lemma *sublist-Int-eq*:
 $xs : list(B) ==> sublist(xs, A \cap nat) = sublist(xs, A)$
 ⟨*proof*⟩

Repetition of a List Element

consts *repeat* :: $[i, i] => i$
primrec
 $repeat(a, 0) = []$
 $repeat(a, succ(n)) = Cons(a, repeat(a, n))$

lemma *length-repeat*: $n \in nat ==> length(repeat(a, n)) = n$
 ⟨*proof*⟩

lemma *repeat-succ-app*: $n \in nat ==> repeat(a, succ(n)) = repeat(a, n) @ [a]$
 ⟨*proof*⟩

lemma *repeat-type* [*TC*]: $[a \in A; n \in nat] ==> repeat(a, n) \in list(A)$
 ⟨*proof*⟩

end

29 EquivClass: Equivalence Relations

theory *EquivClass* **imports** *Trancl Perm* **begin**

definition
 $quotient :: [i, i] => i$ (**infixl** $'/'$ 90) **where**
 $A / r == \{r''\{x\} . x:A\}$

definition
 $congruent :: [i, i => i] => o$ **where**
 $congruent(r, b) == ALL\ y\ z.\ <y, z> : r --> b(y) = b(z)$

definition
 $congruent2 :: [i, i, [i, i] => i] => o$ **where**
 $congruent2(r1, r2, b) == ALL\ y1\ z1\ y2\ z2.\$
 $<y1, z1> : r1 --> <y2, z2> : r2 --> b(y1, y2) = b(z1, z2)$

abbreviation

RESPECTS :: $[i=>i, i] => o$ (**infixr respects 80**) **where**
f respects r == congruent(r,f)

abbreviation

RESPECTS2 :: $[i=>i=>i, i] => o$ (**infixr respects2 80**) **where**
f respects2 r == congruent2(r,r,f)

— Abbreviation for the common case where the relations are identical

29.1 Suppes, Theorem 70: r is an equiv relation iff $\text{converse}(r)$

$O r = r$

lemma *sym-trans-comp-subset*:

$[[\text{sym}(r); \text{trans}(r)]] ==> \text{converse}(r) O r <= r$
<proof>

lemma *refl-comp-subset*:

$[[\text{refl}(A,r); r <= A*A]] ==> r <= \text{converse}(r) O r$
<proof>

lemma *equiv-comp-eq*:

$\text{equiv}(A,r) ==> \text{converse}(r) O r = r$
<proof>

lemma *comp-equivI*:

$[[\text{converse}(r) O r = r; \text{domain}(r) = A]] ==> \text{equiv}(A,r)$
<proof>

lemma *equiv-class-subset*:

$[[\text{sym}(r); \text{trans}(r); <a,b>: r]] ==> r''\{a\} <= r''\{b\}$
<proof>

lemma *equiv-class-eq*:

$[[\text{equiv}(A,r); <a,b>: r]] ==> r''\{a\} = r''\{b\}$
<proof>

lemma *equiv-class-self*:

$[[\text{equiv}(A,r); a: A]] ==> a: r''\{a\}$
<proof>

lemma *subset-equiv-class*:

$[[\text{equiv}(A,r); r''\{b\} <= r''\{a\}; b: A]] ==> <a,b>: r$
<proof>

lemma *eq-equiv-class*: $[[r''\{a\} = r''\{b\}; \text{equiv}(A,r); b: A]] ==> <a,b>: r$

$\langle proof \rangle$

lemma *equiv-class-nondisjoint*:

$\llbracket equiv(A,r); x: (r''\{a} \text{ Int } r''\{b}) \rrbracket \implies \langle a,b \rangle: r$
 $\langle proof \rangle$

lemma *equiv-type*: $equiv(A,r) \implies r \leq A * A$

$\langle proof \rangle$

lemma *equiv-class-iff*:

$equiv(A,r) \implies \langle x,y \rangle: r \iff r''\{x} = r''\{y} \ \& \ x:A \ \& \ y:A$
 $\langle proof \rangle$

lemma *eq-equiv-class-iff*:

$\llbracket equiv(A,r); x: A; y: A \rrbracket \implies r''\{x} = r''\{y} \iff \langle x,y \rangle: r$
 $\langle proof \rangle$

lemma *quotientI* [TC]: $x:A \implies r''\{x}: A//r$

$\langle proof \rangle$

lemma *quotientE*:

$\llbracket X: A//r; !!x. \llbracket X = r''\{x}; x:A \rrbracket \implies P \rrbracket \implies P$
 $\langle proof \rangle$

lemma *Union-quotient*:

$equiv(A,r) \implies Union(A//r) = A$
 $\langle proof \rangle$

lemma *quotient-disj*:

$\llbracket equiv(A,r); X: A//r; Y: A//r \rrbracket \implies X=Y \mid (X \text{ Int } Y \leq 0)$
 $\langle proof \rangle$

29.2 Defining Unary Operations upon Equivalence Classes

lemma *UN-equiv-class*:

$\llbracket equiv(A,r); b \text{ respects } r; a: A \rrbracket \implies (UN x:r''\{a}. b(x)) = b(a)$
 $\langle proof \rangle$

lemma *UN-equiv-class-type*:

$\llbracket equiv(A,r); b \text{ respects } r; X: A//r; !!x. x: A \rrbracket \implies b(x): B$
 $\implies (UN x:X. b(x)): B$
 $\langle proof \rangle$

lemma *UN-equiv-class-inject*:

$\llbracket \text{equiv}(A,r); \text{ b respects } r; \\ (UN\ x:X. b(x))=(UN\ y:Y. b(y)); X: A//r; Y: A//r; \\ !!x\ y. \llbracket x:A; y:A; b(x)=b(y) \rrbracket \implies \langle x,y \rangle : r \rrbracket \\ \implies X=Y$
 $\langle \text{proof} \rangle$

29.3 Defining Binary Operations upon Equivalence Classes

lemma *congruent2-implies-congruent*:

$\llbracket \text{equiv}(A,r1); \text{ congruent2}(r1,r2,b); a: A \rrbracket \implies \text{congruent}(r2,b(a))$
 $\langle \text{proof} \rangle$

lemma *congruent2-implies-congruent-UN*:

$\llbracket \text{equiv}(A1,r1); \text{ equiv}(A2,r2); \text{ congruent2}(r1,r2,b); a: A2 \rrbracket \implies \\ \text{congruent}(r1, \%x1. \bigcup x2 \in r2 \text{ ``}\{a\}. b(x1,x2))$
 $\langle \text{proof} \rangle$

lemma *UN-equiv-class2*:

$\llbracket \text{equiv}(A1,r1); \text{ equiv}(A2,r2); \text{ congruent2}(r1,r2,b); a1: A1; a2: A2 \rrbracket \\ \implies (\bigcup x1 \in r1 \text{ ``}\{a1\}. \bigcup x2 \in r2 \text{ ``}\{a2\}. b(x1,x2)) = b(a1,a2)$
 $\langle \text{proof} \rangle$

lemma *UN-equiv-class-type2*:

$\llbracket \text{equiv}(A,r); \text{ b respects2 } r; \\ X1: A//r; X2: A//r; \\ !!x1\ x2. \llbracket x1: A; x2: A \rrbracket \implies b(x1,x2) : B \\ \rrbracket \implies (UN\ x1:X1. UN\ x2:X2. b(x1,x2)) : B$
 $\langle \text{proof} \rangle$

lemma *congruent2I*:

$\llbracket \text{equiv}(A1,r1); \text{ equiv}(A2,r2); \\ !!\ y\ z\ w. \llbracket w \in A2; \langle y,z \rangle \in r1 \rrbracket \implies b(y,w) = b(z,w); \\ !!\ y\ z\ w. \llbracket w \in A1; \langle y,z \rangle \in r2 \rrbracket \implies b(w,y) = b(w,z) \\ \rrbracket \implies \text{congruent2}(r1,r2,b)$
 $\langle \text{proof} \rangle$

lemma *congruent2-commuteI*:

assumes *equivA*: $\text{equiv}(A,r)$
and *commute*: $!!\ y\ z. \llbracket y: A; z: A \rrbracket \implies b(y,z) = b(z,y)$
and *cong*: $!!\ y\ z\ w. \llbracket w: A; \langle y,z \rangle : r \rrbracket \implies b(w,y) = b(w,z)$
shows *b respects2 r*
 $\langle \text{proof} \rangle$

```

lemma congruent-commuteI:
  [| equiv(A,r); Z: A//r;
    !!w. [| w: A |] ==> congruent(r, %z. b(w,z));
    !!x y. [| x: A; y: A |] ==> b(y,x) = b(x,y)
  |] ==> congruent(r, %w. UN z: Z. b(w,z))
<proof>

end

```

30 Int-ZF: The Integers as Equivalence Classes Over Pairs of Natural Numbers

```

theory Int-ZF imports EquivClass ArithSimp begin

```

definition

```

intrel :: i where
  intrel == {p : (nat*nat)*(nat*nat).
    ∃ x1 y1 x2 y2. p=<<x1,y1>,<x2,y2>> & x1#+y2 = x2#+y1}

```

definition

```

int :: i where
  int == (nat*nat)//intrel

```

definition

```

int-of :: i=>i — coercion from nat to int    ($# - [80] 80) where
  $# m == intrel “ {<natify(m), 0>}

```

definition

```

intify :: i=>i — coercion from ANYTHING to int where
  intify(m) == if m : int then m else $#0

```

definition

```

raw-zminus :: i=>i where
  raw-zminus(z) == ∪ <x,y>∈z. intrel“{<y,x>}

```

definition

```

zminus :: i=>i    ($- - [80] 80) where
  $- z == raw-zminus (intify(z))

```

definition

```

znegative :: i=>o where
  znegative(z) == ∃ x y. x<y & y∈nat & <x,y>∈z

```

definition

```

iszero :: i=>o where
  iszero(z) == z = $# 0

```


notation (*xsymbols*)
 $zmult$ (**infixl** \times 70) **and**
 zle (**infixl** \leq 50) — less than or equals

notation (*HTML output*)
 $zmult$ (**infixl** \times 70) **and**
 zle (**infixl** \leq 50)

declare *quotientE* [*elim!*]

30.1 Proving that *intrel* is an equivalence relation

lemma *intrel-iff* [*simp*]:

$\langle\langle x1, y1 \rangle, \langle x2, y2 \rangle\rangle : intrel \leftrightarrow$
 $x1 \in nat \ \& \ y1 \in nat \ \& \ x2 \in nat \ \& \ y2 \in nat \ \& \ x1 \# + y2 = x2 \# + y1$
<proof>

lemma *intrelI* [*intro!*]:

$[[x1 \# + y2 = x2 \# + y1; x1 \in nat; y1 \in nat; x2 \in nat; y2 \in nat]]$
 $\implies \langle\langle x1, y1 \rangle, \langle x2, y2 \rangle\rangle : intrel$
<proof>

lemma *intrelE* [*elim!*]:

$[[p : intrel;$
 $!!x1 \ y1 \ x2 \ y2. [[p = \langle\langle x1, y1 \rangle, \langle x2, y2 \rangle\rangle; x1 \# + y2 = x2 \# + y1;$
 $x1 \in nat; y1 \in nat; x2 \in nat; y2 \in nat]] \implies Q]]$
 $\implies Q$
<proof>

lemma *int-trans-lemma*:

$[[x1 \# + y2 = x2 \# + y1; x2 \# + y3 = x3 \# + y2]] \implies x1 \# + y3 = x3 \# + y1$
<proof>

lemma *equiv-intrel*: *equiv*(*nat***nat*, *intrel*)

<proof>

lemma *image-intrel-int*: $[[m \in nat; n \in nat]] \implies intrel \ \{ \langle m, n \rangle \} : int$

<proof>

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*]: $\$ \# m : int$

<proof>

lemma *int-of-eq* [iff]: ($\$ \# m = \$ \# n$) \leftrightarrow $\text{nativy}(m) = \text{nativy}(n)$
<proof>

lemma *int-of-inject*: [$\$ \# m = \$ \# n$; $m \in \text{nat}$; $n \in \text{nat}$] $\implies m = n$
<proof>

lemma *intify-in-int* [iff,TC]: $\text{intify}(x) : \text{int}$
<proof>

lemma *intify-ident* [simp]: $n : \text{int} \implies \text{intify}(n) = n$
<proof>

30.2 Collapsing rules: to remove *intify* from arithmetic expressions

lemma *intify-idem* [simp]: $\text{intify}(\text{intify}(x)) = \text{intify}(x)$
<proof>

lemma *int-of-nativy* [simp]: $\$ \# (\text{nativy}(m)) = \$ \# m$
<proof>

lemma *zminus-intify* [simp]: $\$ - (\text{intify}(m)) = \$ - m$
<proof>

lemma *zadd-intify1* [simp]: $\text{intify}(x) \$ + y = x \$ + y$
<proof>

lemma *zadd-intify2* [simp]: $x \$ + \text{intify}(y) = x \$ + y$
<proof>

lemma *zdiff-intify1* [simp]: $\text{intify}(x) \$ - y = x \$ - y$
<proof>

lemma *zdiff-intify2* [simp]: $x \$ - \text{intify}(y) = x \$ - y$
<proof>

lemma *zmult-intify1* [simp]: $\text{intify}(x) \$ * y = x \$ * y$
<proof>

lemma *zmult-intify2* [*simp*]: $x \$* \text{intify}(y) = x \$* y$
 ⟨*proof*⟩

lemma *zless-intify1* [*simp*]: $\text{intify}(x) \$< y \leftrightarrow x \$< y$
 ⟨*proof*⟩

lemma *zless-intify2* [*simp*]: $x \$< \text{intify}(y) \leftrightarrow x \$< y$
 ⟨*proof*⟩

lemma *zle-intify1* [*simp*]: $\text{intify}(x) \$\leq y \leftrightarrow x \$\leq y$
 ⟨*proof*⟩

lemma *zle-intify2* [*simp*]: $x \$\leq \text{intify}(y) \leftrightarrow x \$\leq y$
 ⟨*proof*⟩

30.3 *zminus*: unary negation on *int*

lemma *zminus-congruent*: $(\%<x,y>. \text{intrel}''\{<y,x>\})$ respects *intrel*
 ⟨*proof*⟩

lemma *raw-zminus-type*: $z : \text{int} \implies \text{raw-zminus}(z) : \text{int}$
 ⟨*proof*⟩

lemma *zminus-type* [*TC,iff*]: $\$-z : \text{int}$
 ⟨*proof*⟩

lemma *raw-zminus-inject*:
 $[\text{raw-zminus}(z) = \text{raw-zminus}(w); z : \text{int}; w : \text{int}] \implies z = w$
 ⟨*proof*⟩

lemma *zminus-inject-intify* [*dest!*]: $\$-z = \$-w \implies \text{intify}(z) = \text{intify}(w)$
 ⟨*proof*⟩

lemma *zminus-inject*: $[\$-z = \$-w; z : \text{int}; w : \text{int}] \implies z = w$
 ⟨*proof*⟩

lemma *raw-zminus*:
 $[\text{intrel}''\{<x,y>\} \implies \text{raw-zminus}(\text{intrel}''\{<x,y>\}) = \text{intrel}''\{<y,x>\}$
 ⟨*proof*⟩

lemma *zminus*:
 $[\text{intrel}''\{<x,y>\} \implies \$-(\text{intrel}''\{<x,y>\}) = \text{intrel}''\{<y,x>\}$
 ⟨*proof*⟩

lemma *raw-zminus-zminus*: $z : \text{int} \implies \text{raw-zminus}(\text{raw-zminus}(z)) = z$

<proof>

lemma *zminus-zminus-intify* [simp]: $\$- (\$- z) = \text{intify}(z)$
<proof>

lemma *zminus-int0* [simp]: $\$- (\$ \# 0) = \$ \# 0$
<proof>

lemma *zminus-zminus*: $z : \text{int} \implies \$- (\$- z) = z$
<proof>

30.4 *znegative*: the test for negative integers

lemma *znegative*: $[[x \in \text{nat}; y \in \text{nat}]] \implies \text{znegative}(\text{intrel}\{\langle x, y \rangle\}) \iff x < y$
<proof>

lemma *not-znegative-int-of* [iff]: $\sim \text{znegative}(\$ \# n)$
<proof>

lemma *znegative-zminus-int-of* [simp]: $\text{znegative}(\$- \$ \# \text{succ}(n))$
<proof>

lemma *not-znegative-imp-zero*: $\sim \text{znegative}(\$- \$ \# n) \implies \text{nativify}(n) = 0$
<proof>

30.5 *nat-of*: Coercion of an Integer to a Natural Number

lemma *nat-of-intify* [simp]: $\text{nat-of}(\text{intify}(z)) = \text{nat-of}(z)$
<proof>

lemma *nat-of-congruent*: $(\lambda x. (\lambda \langle x, y \rangle. x \# - y)(x))$ respects *intrel*
<proof>

lemma *raw-nat-of*:
 $[[x \in \text{nat}; y \in \text{nat}]] \implies \text{raw-nat-of}(\text{intrel}\{\langle x, y \rangle\}) = x \# - y$
<proof>

lemma *raw-nat-of-int-of*: $\text{raw-nat-of}(\$ \# n) = \text{nativify}(n)$
<proof>

lemma *nat-of-int-of* [simp]: $\text{nat-of}(\$ \# n) = \text{nativify}(n)$
<proof>

lemma *raw-nat-of-type*: $\text{raw-nat-of}(z) \in \text{nat}$
<proof>

lemma *nat-of-type* [iff, TC]: $\text{nat-of}(z) \in \text{nat}$
<proof>

30.6 zmagnitude: magnitide of an integer, as a natural number

lemma *zmagnitude-int-of* [*simp*]: $zmagnitude(\$# n) = natify(n)$
 ⟨*proof*⟩

lemma *natify-int-of-eq*: $natify(x) = n \implies \$#x = \$# n$
 ⟨*proof*⟩

lemma *zmagnitude-zminus-int-of* [*simp*]: $zmagnitude(\$- \$# n) = natify(n)$
 ⟨*proof*⟩

lemma *zmagnitude-type* [*iff, TC*]: $zmagnitude(z) \in nat$
 ⟨*proof*⟩

lemma *not-zneg-int-of*:
 $[| z : int; \sim znegative(z) |] \implies \exists n \in nat. z = \$# n$
 ⟨*proof*⟩

lemma *not-zneg-mag* [*simp*]:
 $[| z : int; \sim znegative(z) |] \implies \$# (zmagnitude(z)) = z$
 ⟨*proof*⟩

lemma *zneg-int-of*:
 $[| znegative(z); z : int |] \implies \exists n \in nat. z = \$- (\$# succ(n))$
 ⟨*proof*⟩

lemma *zneg-mag* [*simp*]:
 $[| znegative(z); z : int |] \implies \$# (zmagnitude(z)) = \$- z$
 ⟨*proof*⟩

lemma *int-cases*: $z : int \implies \exists n \in nat. z = \$# n \mid z = \$- (\$# succ(n))$
 ⟨*proof*⟩

lemma *not-zneg-raw-nat-of*:
 $[| \sim znegative(z); z : int |] \implies \$# (raw-nat-of(z)) = z$
 ⟨*proof*⟩

lemma *not-zneg-nat-of-intify*:
 $\sim znegative(intify(z)) \implies \$# (nat-of(z)) = intify(z)$
 ⟨*proof*⟩

lemma *not-zneg-nat-of*: $[| \sim znegative(z); z : int |] \implies \$# (nat-of(z)) = z$
 ⟨*proof*⟩

lemma *zneg-nat-of* [*simp*]: $znegative(intify(z)) \implies nat-of(z) = 0$
 ⟨*proof*⟩

30.7 *op* \$+: addition on int

Congruence Property for Addition

lemma *zadd-congruent2*:

(%z1 z2. let <x1,y1>=z1; <x2,y2>=z2
in intrel“{<x1#+x2, y1#+y2>}”
respects2 intrel
<proof>

lemma *raw-zadd-type*: [| z: int; w: int |] ==> raw-zadd(z,w) : int
<proof>

lemma *zadd-type [iff,TC]*: z \$+ w : int
<proof>

lemma *raw-zadd*:

[| x1∈nat; y1∈nat; x2∈nat; y2∈nat |]
==> raw-zadd (intrel“{<x1,y1>}”, intrel“{<x2,y2>}”) =
intrel “ {<x1#+x2, y1#+y2>}”
<proof>

lemma *zadd*:

[| x1∈nat; y1∈nat; x2∈nat; y2∈nat |]
==> (intrel“{<x1,y1>}”) \$+ (intrel“{<x2,y2>}”) =
intrel “ {<x1#+x2, y1#+y2>}”
<proof>

lemma *raw-zadd-int0*: z : int ==> raw-zadd (\$#0,z) = z
<proof>

lemma *zadd-int0-intify [simp]*: \$#0 \$+ z = intify(z)
<proof>

lemma *zadd-int0*: z: int ==> \$#0 \$+ z = z
<proof>

lemma *raw-zminus-zadd-distrib*:

[| z: int; w: int |] ==> \$- raw-zadd(z,w) = raw-zadd(\$- z, \$- w)
<proof>

lemma *zminus-zadd-distrib [simp]*: \$- (z \$+ w) = \$- z \$+ \$- w
<proof>

lemma *raw-zadd-commute*:

[| z: int; w: int |] ==> raw-zadd(z,w) = raw-zadd(w,z)
<proof>

lemma *zadd-commute*: z \$+ w = w \$+ z
<proof>

lemma *raw-zadd-assoc*:

$\llbracket z1 : \text{int}; z2 : \text{int}; z3 : \text{int} \rrbracket$
 $\implies \text{raw-zadd} (\text{raw-zadd}(z1, z2), z3) = \text{raw-zadd}(z1, \text{raw-zadd}(z2, z3))$
 $\langle \text{proof} \rangle$

lemma *zadd-assoc*: $(z1 \ \$+ \ z2) \ \$+ \ z3 = z1 \ \$+ \ (z2 \ \$+ \ z3)$
 $\langle \text{proof} \rangle$

lemma *zadd-left-commute*: $z1 \ \$+ \ (z2 \ \$+ \ z3) = z2 \ \$+ \ (z1 \ \$+ \ z3)$
 $\langle \text{proof} \rangle$

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

lemma *int-of-add*: $\$# (m \ \#+ \ n) = (\$#m) \ \$+ \ (\$#n)$
 $\langle \text{proof} \rangle$

lemma *int-succ-int-1*: $\$# \text{succ}(m) = \$# \ 1 \ \$+ \ (\$# \ m)$
 $\langle \text{proof} \rangle$

lemma *int-of-diff*:
 $\llbracket m \in \text{nat}; n \ \text{le} \ m \rrbracket \implies \$# (m \ \#- \ n) = (\$#m) \ \$- \ (\$#n)$
 $\langle \text{proof} \rangle$

lemma *raw-zadd-zminus-inverse*: $z : \text{int} \implies \text{raw-zadd} (z, \$- \ z) = \$\#0$
 $\langle \text{proof} \rangle$

lemma *zadd-zminus-inverse [simp]*: $z \ \$+ \ (\$- \ z) = \$\#0$
 $\langle \text{proof} \rangle$

lemma *zadd-zminus-inverse2 [simp]*: $(\$- \ z) \ \$+ \ z = \$\#0$
 $\langle \text{proof} \rangle$

lemma *zadd-int0-right-intify [simp]*: $z \ \$+ \ \$\#0 = \text{intify}(z)$
 $\langle \text{proof} \rangle$

lemma *zadd-int0-right*: $z : \text{int} \implies z \ \$+ \ \$\#0 = z$
 $\langle \text{proof} \rangle$

30.8 *op* $\$ \times$: Integer Multiplication

Congruence property for multiplication

lemma *zmult-congruent2*:

$(\%p1 \ p2. \ \text{split}(\%x1 \ y1. \ \text{split}(\%x2 \ y2. \ \text{intrel}''\{\langle x1 \ \#* \ x2 \ \#+ \ y1 \ \#* \ y2, \ x1 \ \#* \ y2 \ \#+ \ y1 \ \#* \ x2 \rangle\}, p2), p1))$
respects2 intrel
 $\langle \text{proof} \rangle$

lemma *raw-zmult-type*: $[[z: int; w: int]] \implies \text{raw-zmult}(z,w) : int$
 ⟨proof⟩

lemma *zmult-type [iff,TC]*: $z \text{ \$* } w : int$
 ⟨proof⟩

lemma *raw-zmult*:
 $[[x1 \in nat; y1 \in nat; x2 \in nat; y2 \in nat]]$
 $\implies \text{raw-zmult}(\text{intrel}\{\langle x1,y1 \rangle\}, \text{intrel}\{\langle x2,y2 \rangle\}) =$
 $\text{intrel}\{\langle x1\#*x2 \#+ y1\#*y2, x1\#*y2 \#+ y1\#*x2 \rangle\}$
 ⟨proof⟩

lemma *zmult*:
 $[[x1 \in nat; y1 \in nat; x2 \in nat; y2 \in nat]]$
 $\implies (\text{intrel}\{\langle x1,y1 \rangle\}) \text{ \$* } (\text{intrel}\{\langle x2,y2 \rangle\}) =$
 $\text{intrel}\{\langle x1\#*x2 \#+ y1\#*y2, x1\#*y2 \#+ y1\#*x2 \rangle\}$
 ⟨proof⟩

lemma *raw-zmult-int0*: $z : int \implies \text{raw-zmult}(\$#0,z) = \$#0$
 ⟨proof⟩

lemma *zmult-int0 [simp]*: $\$#0 \text{ \$* } z = \$#0$
 ⟨proof⟩

lemma *raw-zmult-int1*: $z : int \implies \text{raw-zmult}(\$#1,z) = z$
 ⟨proof⟩

lemma *zmult-int1-intify [simp]*: $\$#1 \text{ \$* } z = \text{intify}(z)$
 ⟨proof⟩

lemma *zmult-int1*: $z : int \implies \$#1 \text{ \$* } z = z$
 ⟨proof⟩

lemma *raw-zmult-commute*:
 $[[z: int; w: int]] \implies \text{raw-zmult}(z,w) = \text{raw-zmult}(w,z)$
 ⟨proof⟩

lemma *zmult-commute*: $z \text{ \$* } w = w \text{ \$* } z$
 ⟨proof⟩

lemma *raw-zmult-zminus*:
 $[[z: int; w: int]] \implies \text{raw-zmult}(\$- z, w) = \$- \text{raw-zmult}(z, w)$
 ⟨proof⟩

lemma *zmult-zminus [simp]*: $(\$- z) \text{ \$* } w = \$- (z \text{ \$* } w)$
 ⟨proof⟩

lemma *zmult-zminus-right [simp]*: $w \ \$* (\$- z) = \$- (w \ \$* z)$
 ⟨proof⟩

lemma *raw-zmult-assoc*:
 [| $z1: int$; $z2: int$; $z3: int$ |]
 $\implies raw-zmult (raw-zmult(z1, z2), z3) = raw-zmult(z1, raw-zmult(z2, z3))$
 ⟨proof⟩

lemma *zmult-assoc*: $(z1 \ \$* z2) \ \$* z3 = z1 \ \$* (z2 \ \$* z3)$
 ⟨proof⟩

lemma *zmult-left-commute*: $z1 \ \$*(z2 \ \$* z3) = z2 \ \$*(z1 \ \$* z3)$
 ⟨proof⟩

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

lemma *raw-zadd-zmult-distrib*:
 [| $z1: int$; $z2: int$; $w: int$ |]
 $\implies raw-zmult(raw-zadd(z1, z2), w) =$
 $raw-zadd (raw-zmult(z1, w), raw-zmult(z2, w))$
 ⟨proof⟩

lemma *zadd-zmult-distrib*: $(z1 \ \$+ z2) \ \$* w = (z1 \ \$* w) \ \$+ (z2 \ \$* w)$
 ⟨proof⟩

lemma *zadd-zmult-distrib2*: $w \ \$* (z1 \ \$+ z2) = (w \ \$* z1) \ \$+ (w \ \$* z2)$
 ⟨proof⟩

lemmas *int-typechecks =*
int-of-type zminus-type zmagnitude-type zadd-type zmult-type

lemma *zdiff-type [iff, TC]*: $z \ \$- w : int$
 ⟨proof⟩

lemma *zminus-zdiff-eq [simp]*: $\$- (z \ \$- y) = y \ \$- z$
 ⟨proof⟩

lemma *zdiff-zmult-distrib*: $(z1 \ \$- z2) \ \$* w = (z1 \ \$* w) \ \$- (z2 \ \$* w)$
 ⟨proof⟩

lemma *zdiff-zmult-distrib2*: $w \ \$* (z1 \ \$- z2) = (w \ \$* z1) \ \$- (w \ \$* z2)$
 ⟨proof⟩

lemma *zadd-zdiff-eq*: $x \ \$+ (y \ \$- z) = (x \ \$+ y) \ \$- z$

<proof>

lemma *zdifff-zadd-eq*: $(x \$- y) \$+ z = (x \$+ z) \$- y$
<proof>

30.9 The "Less Than" Relation

lemma *zless-linear-lemma*:
[[$z: int; w: int$]] ==> $z \$< w \mid z = w \mid w \$< z$
<proof>

lemma *zless-linear*: $z \$< w \mid intify(z) = intify(w) \mid w \$< z$
<proof>

lemma *zless-not-refl [iff]*: $\sim (z \$< z)$
<proof>

lemma *neq-iff-zless*: [[$x: int; y: int$]] ==> $(x \sim = y) <-> (x \$< y \mid y \$< x)$
<proof>

lemma *zless-imp-intify-neq*: $w \$< z ==> intify(w) \sim = intify(z)$
<proof>

lemma *zless-imp-succ-zadd-lemma*:
[[$w \$< z; w: int; z: int$]] ==> $(\exists n \in nat. z = w \$+ \$\#(succ(n)))$
<proof>

lemma *zless-imp-succ-zadd*:
 $w \$< z ==> (\exists n \in nat. w \$+ \$\#(succ(n)) = intify(z))$
<proof>

lemma *zless-succ-zadd-lemma*:
 $w : int ==> w \$< w \$+ \$\# succ(n)$
<proof>

lemma *zless-succ-zadd*: $w \$< w \$+ \$\# succ(n)$
<proof>

lemma *zless-iff-succ-zadd*:
 $w \$< z <-> (\exists n \in nat. w \$+ \$\#(succ(n)) = intify(z))$
<proof>

lemma *zless-int-of [simp]*: [[$m \in nat; n \in nat$]] ==> $(\$ \# m \$< \$ \# n) <-> (m < n)$
<proof>

lemma *zless-trans-lemma*:
[[$x \$< y; y \$< z; x: int; y: int; z: int$]] ==> $x \$< z$
<proof>

lemma *zless-trans*: $[[x \$< y; y \$< z]] ==> x \$< z$
<proof>

lemma *zless-not-sym*: $z \$< w ==> \sim (w \$< z)$
<proof>

lemmas *zless-asm* = *zless-not-sym* [*THEN swap, standard*]

lemma *zless-imp-zle*: $z \$< w ==> z \$\leq w$
<proof>

lemma *zle-linear*: $z \$\leq w \mid w \$\leq z$
<proof>

30.10 Less Than or Equals

lemma *zle-refl*: $z \$\leq z$
<proof>

lemma *zle-eq-refl*: $x=y ==> x \$\leq y$
<proof>

lemma *zle-anti-sym-intify*: $[[x \$\leq y; y \$\leq x]] ==> \text{intify}(x) = \text{intify}(y)$
<proof>

lemma *zle-anti-sym*: $[[x \$\leq y; y \$\leq x; x: \text{int}; y: \text{int}]] ==> x=y$
<proof>

lemma *zle-trans-lemma*:
 $[[x: \text{int}; y: \text{int}; z: \text{int}; x \$\leq y; y \$\leq z]] ==> x \$\leq z$
<proof>

lemma *zle-trans*: $[[x \$\leq y; y \$\leq z]] ==> x \$\leq z$
<proof>

lemma *zle-zless-trans*: $[[i \$\leq j; j \$< k]] ==> i \$< k$
<proof>

lemma *zless-zle-trans*: $[[i \$< j; j \$\leq k]] ==> i \$< k$
<proof>

lemma *not-zless-iff-zle*: $\sim (z \$< w) <-> (w \$\leq z)$
<proof>

lemma *not-zle-iff-zless*: $\sim (z \$\leq w) <-> (w \$< z)$
<proof>

30.11 More subtraction laws (for *zcompare-rls*)

lemma *zdiff-zdiff-eq*: $(x \$- y) \$- z = x \$- (y \$+ z)$
 $\langle proof \rangle$

lemma *zdiff-zdiff-eq2*: $x \$- (y \$- z) = (x \$+ z) \$- y$
 $\langle proof \rangle$

lemma *zdiff-zless-iff*: $(x \$- y \$< z) <-> (x \$< z \$+ y)$
 $\langle proof \rangle$

lemma *zless-zdiff-iff*: $(x \$< z \$- y) <-> (x \$+ y \$< z)$
 $\langle proof \rangle$

lemma *zdiff-eq-iff*: $[| x: int; z: int |] ==> (x \$- y = z) <-> (x = z \$+ y)$
 $\langle proof \rangle$

lemma *eq-zdiff-iff*: $[| x: int; z: int |] ==> (x = z \$- y) <-> (x \$+ y = z)$
 $\langle proof \rangle$

lemma *zdiff-zle-iff-lemma*:
 $[| x: int; z: int |] ==> (x \$- y \$<= z) <-> (x \$<= z \$+ y)$
 $\langle proof \rangle$

lemma *zdiff-zle-iff*: $(x \$- y \$<= z) <-> (x \$<= z \$+ y)$
 $\langle proof \rangle$

lemma *zle-zdiff-iff-lemma*:
 $[| x: int; z: int |] ==> (x \$<= z \$- y) <-> (x \$+ y \$<= z)$
 $\langle proof \rangle$

lemma *zle-zdiff-iff*: $(x \$<= z \$- y) <-> (x \$+ y \$<= z)$
 $\langle proof \rangle$

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*

30.12 Monotonicity and Cancellation Results for Instantiation of the CancelNumerals Simprocs

lemma *zadd-left-cancel*:
 $[| w: int; w': int |] ==> (z \$+ w' = z \$+ w) <-> (w' = w)$
 $\langle proof \rangle$

lemma *zadd-left-cancel-intify* [*simp*]:

$(z \$+ w' = z \$+ w) \langle - \rangle \text{intify}(w') = \text{intify}(w)$
 $\langle \text{proof} \rangle$

lemma *zadd-right-cancel*:

$[[w: \text{int}; w': \text{int}]] \implies (w' \$+ z = w \$+ z) \langle - \rangle (w' = w)$
 $\langle \text{proof} \rangle$

lemma *zadd-right-cancel-intify* [*simp*]:

$(w' \$+ z = w \$+ z) \langle - \rangle \text{intify}(w') = \text{intify}(w)$
 $\langle \text{proof} \rangle$

lemma *zadd-right-cancel-zless* [*simp*]: $(w' \$+ z \$< w \$+ z) \langle - \rangle (w' \$< w)$
 $\langle \text{proof} \rangle$

lemma *zadd-left-cancel-zless* [*simp*]: $(z \$+ w' \$< z \$+ w) \langle - \rangle (w' \$< w)$
 $\langle \text{proof} \rangle$

lemma *zadd-right-cancel-zle* [*simp*]: $(w' \$+ z \$\leq w \$+ z) \langle - \rangle w' \$\leq w$
 $\langle \text{proof} \rangle$

lemma *zadd-left-cancel-zle* [*simp*]: $(z \$+ w' \$\leq z \$+ w) \langle - \rangle w' \$\leq w$
 $\langle \text{proof} \rangle$

lemmas *zadd-zless-mono1* = *zadd-right-cancel-zless* [*THEN iffD2, standard*]

lemmas *zadd-zless-mono2* = *zadd-left-cancel-zless* [*THEN iffD2, standard*]

lemmas *zadd-zle-mono1* = *zadd-right-cancel-zle* [*THEN iffD2, standard*]

lemmas *zadd-zle-mono2* = *zadd-left-cancel-zle* [*THEN iffD2, standard*]

lemma *zadd-zle-mono*: $[[w' \$\leq w; z' \$\leq z]] \implies w' \$+ z' \$\leq w \$+ z$
 $\langle \text{proof} \rangle$

lemma *zadd-zless-mono*: $[[w' \$< w; z' \$\leq z]] \implies w' \$+ z' \$< w \$+ z$
 $\langle \text{proof} \rangle$

30.13 Comparison laws

lemma *zminus-zless-zminus* [*simp*]: $(\$- x \$< \$- y) \langle - \rangle (y \$< x)$
 $\langle \text{proof} \rangle$

lemma *zminus-zle-zminus* [*simp*]: $(\$- x \$\leq \$- y) \langle - \rangle (y \$\leq x)$
 $\langle \text{proof} \rangle$

30.13.1 More inequality lemmas

lemma *equation-zminus*: $[[x: int; y: int]] ==> (x = \$- y) <-> (y = \$- x)$
<proof>

lemma *zminus-equation*: $[[x: int; y: int]] ==> (\$- x = y) <-> (\$- y = x)$
<proof>

lemma *equation-zminus-intify*: $(intify(x) = \$- y) <-> (intify(y) = \$- x)$
<proof>

lemma *zminus-equation-intify*: $(\$- x = intify(y)) <-> (\$- y = intify(x))$
<proof>

30.13.2 The next several equations are permutative: watch out!

lemma *zless-zminus*: $(x \$< \$- y) <-> (y \$< \$- x)$
<proof>

lemma *zminus-zless*: $(\$- x \$< y) <-> (\$- y \$< x)$
<proof>

lemma *zle-zminus*: $(x \$<= \$- y) <-> (y \$<= \$- x)$
<proof>

lemma *zminus-zle*: $(\$- x \$<= y) <-> (\$- y \$<= x)$
<proof>

end

31 Bin: Arithmetic on Binary Integers

```
theory Bin
imports Int-ZF Datatype-ZF
uses (Tools/numeral-syntax.ML)
begin

consts bin :: i
datatype
  bin = Pls
      | Min
      | Bit (w: bin, b: bool)    (infixl BIT 90)

consts
  integ-of :: i=>i
  NCons    :: [i,i]=>i
  bin-succ :: i=>i
  bin-pred :: i=>i
```

$bin-minus :: i => i$
 $bin-adder :: i => i$
 $bin-mult :: [i, i] => i$

primrec

$integ-of-Pls: integ-of (Pls) = \#\ 0$
 $integ-of-Min: integ-of (Min) = \#-\ (\#\ 1)$
 $integ-of-BIT: integ-of (w BIT b) = \#\ b \ \$+ integ-of(w) \ \$+ integ-of(w)$

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

$bin-minus-Pls:$
 $bin-minus (Pls) = Pls$
 $bin-minus-Min:$
 $bin-minus (Min) = Pls BIT 1$
 $bin-minus-BIT:$
 $bin-minus (w BIT b) = cond(b, bin-pred(NCons(bin-minus(w), 0)), bin-minus(w) BIT 0)$

primrec

$bin-adder-Pls:$
 $bin-adder (Pls) = (lam w:bin. w)$
 $bin-adder-Min:$
 $bin-adder (Min) = (lam w:bin. bin-pred(w))$
 $bin-adder-BIT:$
 $bin-adder (v BIT x) =$
 $(lam w:bin.$
 $bin-case (v BIT x, bin-pred(v BIT x),$
 $\%w y. NCons(bin-adder (v) ' cond(x and y, bin-succ(w), w),$
 $x xor y),$
 $w))$

definition

$bin-add :: [i,i] \Rightarrow i$ **where**
 $bin-add(v,w) == bin-adder(v)w$

primrec

$bin-mult-Pls$:
 $bin-mult(Pls,w) = Pls$
 $bin-mult-Min$:
 $bin-mult(Min,w) = bin-minus(w)$
 $bin-mult-BIT$:
 $bin-mult(v BIT b,w) = cond(b, bin-add(NCons(bin-mult(v,w),0),w),$
 $NCons(bin-mult(v,w),0))$

syntax

$-Int :: xnum \Rightarrow i \quad (-)$

$\langle ML \rangle$

declare $bin.intros [simp,TC]$

lemma $NCons-Pls-0$: $NCons(Pls,0) = Pls$
 $\langle proof \rangle$

lemma $NCons-Pls-1$: $NCons(Pls,1) = Pls BIT 1$
 $\langle proof \rangle$

lemma $NCons-Min-0$: $NCons(Min,0) = Min BIT 0$
 $\langle proof \rangle$

lemma $NCons-Min-1$: $NCons(Min,1) = Min$
 $\langle proof \rangle$

lemma $NCons-BIT$: $NCons(w BIT x,b) = w BIT x BIT b$
 $\langle proof \rangle$

lemmas $NCons-simps [simp] =$
 $NCons-Pls-0 NCons-Pls-1 NCons-Min-0 NCons-Min-1 NCons-BIT$

lemma $integ-of-type [TC]$: $w: bin \Rightarrow integ-of(w) : int$
 $\langle proof \rangle$

lemma $NCons-type [TC]$: $[| w: bin; b: bool |] \Rightarrow NCons(w,b) : bin$

<proof>

lemma *bin-succ-type* [TC]: $w: \text{bin} \implies \text{bin-succ}(w) : \text{bin}$
<proof>

lemma *bin-pred-type* [TC]: $w: \text{bin} \implies \text{bin-pred}(w) : \text{bin}$
<proof>

lemma *bin-minus-type* [TC]: $w: \text{bin} \implies \text{bin-minus}(w) : \text{bin}$
<proof>

lemma *bin-add-type* [rule-format,TC]:
 $w: \text{bin} \implies \text{ALL } w: \text{bin}. \text{bin-add}(v,w) : \text{bin}$
<proof>

lemma *bin-mult-type* [TC]: $[[v: \text{bin}; w: \text{bin}]] \implies \text{bin-mult}(v,w) : \text{bin}$
<proof>

31.0.3 The Carry and Borrow Functions, *bin-succ* and *bin-pred*

lemma *integ-of-NCons* [simp]:
 $[[w: \text{bin}; b: \text{bool}]] \implies \text{integ-of}(NCons(w,b)) = \text{integ-of}(w \text{ BIT } b)$
<proof>

lemma *integ-of-succ* [simp]:
 $w: \text{bin} \implies \text{integ-of}(\text{bin-succ}(w)) = \$\#1 \ \$+ \ \text{integ-of}(w)$
<proof>

lemma *integ-of-pred* [simp]:
 $w: \text{bin} \implies \text{integ-of}(\text{bin-pred}(w)) = \$- \ (\#\#1) \ \$+ \ \text{integ-of}(w)$
<proof>

31.0.4 *bin-minus*: Unary Negation of Binary Integers

lemma *integ-of-minus*: $w: \text{bin} \implies \text{integ-of}(\text{bin-minus}(w)) = \$- \ \text{integ-of}(w)$
<proof>

31.0.5 *bin-add*: Binary Addition

lemma *bin-add-Pls* [simp]: $w: \text{bin} \implies \text{bin-add}(Pls,w) = w$
<proof>

lemma *bin-add-Pls-right*: $w: \text{bin} \implies \text{bin-add}(w,Pls) = w$
<proof>

lemma *bin-add-Min* [simp]: $w: \text{bin} \implies \text{bin-add}(Min,w) = \text{bin-pred}(w)$
<proof>

lemma *bin-add-Min-right*: $w: \text{bin} \implies \text{bin-add}(w,Min) = \text{bin-pred}(w)$

<proof>

lemma *bin-add-BIT-Pls* [*simp*]: $\text{bin-add}(v \text{ BIT } x, \text{Pls}) = v \text{ BIT } x$
<proof>

lemma *bin-add-BIT-Min* [*simp*]: $\text{bin-add}(v \text{ BIT } x, \text{Min}) = \text{bin-pred}(v \text{ BIT } x)$
<proof>

lemma *bin-add-BIT-BIT* [*simp*]:
[[$w: \text{bin}; y: \text{bool}$]]
 $\implies \text{bin-add}(v \text{ BIT } x, w \text{ BIT } y) =$
 $\text{NCons}(\text{bin-add}(v, \text{cond}(x \text{ and } y, \text{bin-succ}(w), w)), x \text{ xor } y)$
<proof>

lemma *integ-of-add* [*rule-format*]:
 $v: \text{bin} \implies$
 $\text{ALL } w: \text{bin}. \text{integ-of}(\text{bin-add}(v, w)) = \text{integ-of}(v) \$+ \text{integ-of}(w)$
<proof>

lemma *diff-integ-of-eq*:
[[$v: \text{bin}; w: \text{bin}$]]
 $\implies \text{integ-of}(v) \$- \text{integ-of}(w) = \text{integ-of}(\text{bin-add}(v, \text{bin-minus}(w)))$
<proof>

31.0.6 *bin-mult*: Binary Multiplication

lemma *integ-of-mult*:
[[$v: \text{bin}; w: \text{bin}$]]
 $\implies \text{integ-of}(\text{bin-mult}(v, w)) = \text{integ-of}(v) \$* \text{integ-of}(w)$
<proof>

31.1 Computations

lemma *bin-succ-1*: $\text{bin-succ}(w \text{ BIT } 1) = \text{bin-succ}(w) \text{ BIT } 0$
<proof>

lemma *bin-succ-0*: $\text{bin-succ}(w \text{ BIT } 0) = \text{NCons}(w, 1)$
<proof>

lemma *bin-pred-1*: $\text{bin-pred}(w \text{ BIT } 1) = \text{NCons}(w, 0)$
<proof>

lemma *bin-pred-0*: $\text{bin-pred}(w \text{ BIT } 0) = \text{bin-pred}(w) \text{ BIT } 1$
<proof>

lemma *bin-minus-1*: $\text{bin-minus}(w \text{ BIT } 1) = \text{bin-pred}(\text{NCons}(\text{bin-minus}(w), 0))$
<proof>

lemma *bin-minus-0*: $\text{bin-minus}(w \text{ BIT } 0) = \text{bin-minus}(w) \text{ BIT } 0$
<proof>

lemma *bin-add-BIT-11*: $w: \text{bin} \implies \text{bin-add}(v \text{ BIT } 1, w \text{ BIT } 1) =$
 $\text{NCons}(\text{bin-add}(v, \text{bin-succ}(w)), 0)$
<proof>

lemma *bin-add-BIT-10*: $w: \text{bin} \implies \text{bin-add}(v \text{ BIT } 1, w \text{ BIT } 0) =$
 $\text{NCons}(\text{bin-add}(v, w), 1)$
<proof>

lemma *bin-add-BIT-0*: $[[w: \text{bin}; y: \text{bool}]]$
 $\implies \text{bin-add}(v \text{ BIT } 0, w \text{ BIT } y) = \text{NCons}(\text{bin-add}(v, w), y)$
<proof>

lemma *bin-mult-1*: $\text{bin-mult}(v \text{ BIT } 1, w) = \text{bin-add}(\text{NCons}(\text{bin-mult}(v, w), 0), w)$
<proof>

lemma *bin-mult-0*: $\text{bin-mult}(v \text{ BIT } 0, w) = \text{NCons}(\text{bin-mult}(v, w), 0)$
<proof>

lemma *int-of-0*: $\$ \# 0 = \# 0$
<proof>

lemma *int-of-succ*: $\$ \# \text{succ}(n) = \# 1 \$ + \$ \# n$
<proof>

lemma *zminus-0 [simp]*: $\$ - \# 0 = \# 0$
<proof>

lemma *zadd-0-intify [simp]*: $\# 0 \$ + z = \text{intify}(z)$
<proof>

lemma *zadd-0-right-intify [simp]*: $z \$ + \# 0 = \text{intify}(z)$
<proof>

lemma *zmult-1-intify [simp]*: $\# 1 \$ * z = \text{intify}(z)$
<proof>

lemma *zmult-1-right-intify [simp]*: $z \$ * \# 1 = \text{intify}(z)$
<proof>

lemma *zmult-0* [*simp*]: $\#0 \ \$* \ z = \#0$
<proof>

lemma *zmult-0-right* [*simp*]: $z \ \$* \ \#0 = \#0$
<proof>

lemma *zmult-minus1* [*simp*]: $\#-1 \ \$* \ z = \$-z$
<proof>

lemma *zmult-minus1-right* [*simp*]: $z \ \$* \ \#-1 = \$-z$
<proof>

31.2 Simplification Rules for Comparison of Binary Numbers

Thanks to Norbert Voelker

lemma *eq-integ-of-eq*:
[[*v*: *bin*; *w*: *bin*]]
==> ((*integ-of*(*v*)) = *integ-of*(*w*)) <->
 iszero (*integ-of* (*bin-add* (*v*, *bin-minus*(*w*))))
<proof>

lemma *iszero-integ-of-Pls*: *iszero* (*integ-of*(*Pls*))
<proof>

lemma *nonzero-integ-of-Min*: \sim *iszero* (*integ-of*(*Min*))
<proof>

lemma *iszero-integ-of-BIT*:
[[*w*: *bin*; *x*: *bool*]]
==> *iszero* (*integ-of* (*w BIT x*)) <-> (*x=0* & *iszero* (*integ-of*(*w*)))
<proof>

lemma *iszero-integ-of-0*:
w: *bin* ==> *iszero* (*integ-of* (*w BIT 0*)) <-> *iszero* (*integ-of*(*w*))
<proof>

lemma *iszero-integ-of-1*: *w*: *bin* ==> \sim *iszero* (*integ-of* (*w BIT 1*))
<proof>

lemma *less-integ-of-eq-neg*:
[[*v*: *bin*; *w*: *bin*]]
==> *integ-of*(*v*) $\$<$ *integ-of*(*w*)

$\langle \text{proof} \rangle \quad \langle - \rangle \text{znegative } (\text{integ-of } (\text{bin-add } (v, \text{bin-minus}(w))))$

lemma *not-neg-integ-of-Pls*: $\sim \text{znegative } (\text{integ-of } (Pls))$
 $\langle \text{proof} \rangle$

lemma *neg-integ-of-Min*: $\text{znegative } (\text{integ-of } (Min))$
 $\langle \text{proof} \rangle$

lemma *neg-integ-of-BIT*:
[[w : *bin*; x : *bool*]]
 $\implies \text{znegative } (\text{integ-of } (w \text{ BIT } x)) \langle - \rangle \text{znegative } (\text{integ-of } (w))$
 $\langle \text{proof} \rangle$

lemma *le-integ-of-eq-not-less*:
 $(\text{integ-of } (x) \$\leq (\text{integ-of } (w))) \langle - \rangle \sim (\text{integ-of } (w) \$\langle (\text{integ-of } (x)))$
 $\langle \text{proof} \rangle$

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*]:
[[*v*: *bin*; *w*: *bin*]]
 $\implies \text{integ-of}(v) \$+ (\text{integ-of}(w) \$+ z) = (\text{integ-of}(\text{bin-add}(v,w)) \$+ z)$
<*proof*>

lemma *mult-integ-of-left* [*simp*]:
[[*v*: *bin*; *w*: *bin*]]
 $\implies \text{integ-of}(v) \$* (\text{integ-of}(w) \$* z) = (\text{integ-of}(\text{bin-mult}(v,w)) \$* z)$
<*proof*>

lemma *add-integ-of-diff1* [*simp*]:
[[*v*: *bin*; *w*: *bin*]]
 $\implies \text{integ-of}(v) \$+ (\text{integ-of}(w) \$- c) = \text{integ-of}(\text{bin-add}(v,w)) \$- (c)$
<*proof*>

lemma *add-integ-of-diff2* [*simp*]:
[[*v*: *bin*; *w*: *bin*]]
 $\implies \text{integ-of}(v) \$+ (c \$- \text{integ-of}(w)) =$
 $\text{integ-of}(\text{bin-add}(v, \text{bin-minus}(w))) \$+ (c)$
<*proof*>

declare *int-of-0* [*simp*] *int-of-succ* [*simp*]

lemma *zdiff0* [*simp*]: $\#0 \$- x = \$-x$
<*proof*>

lemma *zdiff0-right* [*simp*]: $x \text{ \$- } \#0 = \text{intify}(x)$
<*proof*>

lemma *zdiff-self* [*simp*]: $x \text{ \$- } x = \#0$
<*proof*>

lemma *znegative-iff-zless-0*: $k: \text{int} \implies \text{znegative}(k) \leftrightarrow k \text{ \$< } \#0$
<*proof*>

lemma *zero-zless-imp-znegative-zminus*: $[\#0 \text{ \$< } k; k: \text{int}] \implies \text{znegative}(\text{\$-}k)$
<*proof*>

lemma *zero-zle-int-of* [*simp*]: $\#0 \text{ \$<=} \text{\$# } n$
<*proof*>

lemma *nat-of-0* [*simp*]: $\text{nat-of}(\#0) = 0$
<*proof*>

lemma *nat-le-int0-lemma*: $[z \text{ \$<=} \text{\$#}0; z: \text{int}] \implies \text{nat-of}(z) = 0$
<*proof*>

lemma *nat-le-int0*: $z \text{ \$<=} \text{\$#}0 \implies \text{nat-of}(z) = 0$
<*proof*>

lemma *int-of-eq-0-imp-natify-eq-0*: $\text{\$# } n = \#0 \implies \text{natify}(n) = 0$
<*proof*>

lemma *nat-of-zminus-int-of*: $\text{nat-of}(\text{\$- } \text{\$# } n) = 0$
<*proof*>

lemma *int-of-nat-of*: $\#0 \text{ \$<=} z \implies \text{\$# } \text{nat-of}(z) = \text{intify}(z)$
<*proof*>

declare *int-of-nat-of* [*simp*] *nat-of-zminus-int-of* [*simp*]

lemma *int-of-nat-of-if*: $\text{\$# } \text{nat-of}(z) = (\text{if } \#0 \text{ \$<=} z \text{ then } \text{intify}(z) \text{ else } \#0)$
<*proof*>

lemma *zless-nat-iff-int-zless*: $[m: \text{nat}; z: \text{int}] \implies (m < \text{nat-of}(z)) \leftrightarrow (\text{\$#}m \text{ \$< } z)$
<*proof*>

lemma *zless-nat-conj-lemma*: $\text{\$#}0 \text{ \$< } z \implies (\text{nat-of}(w) < \text{nat-of}(z)) \leftrightarrow (w \text{ \$< } z)$

<proof>

lemma *zless-nat-conj*: $(\text{nat-of}(w) < \text{nat-of}(z)) \leftrightarrow (\$ \neq 0 \ \$ < z \ \& \ w \ \$ < z)$
<proof>

lemma *integ-of-minus-reorient* [*simp*]:
 $(\text{integ-of}(w) = \$ - x) \leftrightarrow (\$ - x = \text{integ-of}(w))$
<proof>

lemma *integ-of-add-reorient* [*simp*]:
 $(\text{integ-of}(w) = x \ \$ + y) \leftrightarrow (x \ \$ + y = \text{integ-of}(w))$
<proof>

lemma *integ-of-diff-reorient* [*simp*]:
 $(\text{integ-of}(w) = x \ \$ - y) \leftrightarrow (x \ \$ - y = \text{integ-of}(w))$
<proof>

lemma *integ-of-mult-reorient* [*simp*]:
 $(\text{integ-of}(w) = x \ \$ * y) \leftrightarrow (x \ \$ * y = \text{integ-of}(w))$
<proof>

end

theory *IntArith* **imports** *Bin*
uses (*int-arith.ML*)
begin

lemmas [*simp*] =
zminus-equation [**where** $y = \text{integ-of}(w)$, *standard*]
equation-zminus [**where** $x = \text{integ-of}(w)$, *standard*]

lemmas [*iff*] =
zminus-zless [**where** $y = \text{integ-of}(w)$, *standard*]
zless-zminus [**where** $x = \text{integ-of}(w)$, *standard*]

lemmas [*iff*] =
zminus-zle [**where** $y = \text{integ-of}(w)$, *standard*]
zle-zminus [**where** $x = \text{integ-of}(w)$, *standard*]

lemmas [*simp*] =
Let-def [**where** $s = \text{integ-of}(w)$, *standard*]

lemma *zless-iff-zdiff-zless-0*: $(x \text{ \$< } y) \text{ <-> } (x \text{ \$-} y \text{ \$< } \#0)$
 ⟨*proof*⟩

lemma *eq-iff-zdiff-eq-0*: $([x: \text{int}; y: \text{int}] \text{ ==> } (x = y) \text{ <-> } (x \text{ \$-} y = \#0))$
 ⟨*proof*⟩

lemma *zle-iff-zdiff-zle-0*: $(x \text{ \$<=} y) \text{ <-> } (x \text{ \$-} y \text{ \$<=} \#0)$
 ⟨*proof*⟩

lemma *left-zadd-zmult-distrib*: $i \text{ \$*} u \text{ \$+ } (j \text{ \$*} u \text{ \$+ } k) = (i \text{ \$+} j) \text{ \$*} u \text{ \$+ } k$
 ⟨*proof*⟩

lemmas *rel-iff-rel-0-rls* =
zless-iff-zdiff-zless-0 [where $y = u \text{ \$+ } v$, *standard*]
eq-iff-zdiff-eq-0 [where $y = u \text{ \$+ } v$, *standard*]
zle-iff-zdiff-zle-0 [where $y = u \text{ \$+ } v$, *standard*]
zless-iff-zdiff-zless-0 [where $y = n$]
eq-iff-zdiff-eq-0 [where $y = n$]
zle-iff-zdiff-zle-0 [where $y = n$]

lemma *eq-add-iff1*: $(i \text{ \$*} u \text{ \$+ } m = j \text{ \$*} u \text{ \$+ } n) \text{ <-> } ((i \text{ \$-} j) \text{ \$*} u \text{ \$+ } m = \text{intify}(n))$
 ⟨*proof*⟩

lemma *eq-add-iff2*: $(i \text{ \$*} u \text{ \$+ } m = j \text{ \$*} u \text{ \$+ } n) \text{ <-> } (\text{intify}(m) = (j \text{ \$-} i) \text{ \$*} u \text{ \$+ } n)$
 ⟨*proof*⟩

lemma *less-add-iff1*: $(i \text{ \$*} u \text{ \$+ } m \text{ \$< } j \text{ \$*} u \text{ \$+ } n) \text{ <-> } ((i \text{ \$-} j) \text{ \$*} u \text{ \$+ } m \text{ \$< } n)$
 ⟨*proof*⟩

lemma *less-add-iff2*: $(i \text{ \$*} u \text{ \$+ } m \text{ \$< } j \text{ \$*} u \text{ \$+ } n) \text{ <-> } (m \text{ \$< } (j \text{ \$-} i) \text{ \$*} u \text{ \$+ } n)$
 ⟨*proof*⟩

lemma *le-add-iff1*: $(i \text{ \$*} u \text{ \$+ } m \text{ \$<=} j \text{ \$*} u \text{ \$+ } n) \text{ <-> } ((i \text{ \$-} j) \text{ \$*} u \text{ \$+ } m \text{ \$<=} n)$
 ⟨*proof*⟩

lemma *le-add-iff2*: $(i \text{ \$*} u \text{ \$+ } m \text{ \$<=} j \text{ \$*} u \text{ \$+ } n) \text{ <-> } (m \text{ \$<=} (j \text{ \$-} i) \text{ \$*} u \text{ \$+ } n)$

<proof>
 <ML>
 end

32 IntDiv-ZF: The Division Operators Div and Mod

theory IntDiv-ZF imports IntArith OrderArith begin

definition

$quorem :: [i, i] \Rightarrow o$ **where**
 $quorem == \%<a, b> <q, r>.$
 $a = b * q + r \ \&$
 $(\#0 < b \ \& \ \#0 \leq r \ \& \ r < b \mid \sim(\#0 < b) \ \& \ b < r \ \& \ r \leq \#0)$

definition

$adjust :: [i, i] \Rightarrow i$ **where**
 $adjust(b) == \%<q, r>.$ *if* $\#0 \leq r - b$ *then* $<\#2 * q + \#1, r - b>$
else $<\#2 * q, r>$

definition

$posDivAlg :: i \Rightarrow i$ **where**

$posDivAlg(ab) ==$
 $wfrec(measure(int * int, \%<a, b>. nat-of (a - b + \#1)),$
 $ab,$
 $\%<a, b> f. \text{if } (a < b \mid b \leq \#0) \text{ then } <\#0, a>$
 $\text{else } adjust(b, f ' <a, \#2 * b>))$

definition

$negDivAlg :: i \Rightarrow i$ **where**

$negDivAlg(ab) ==$
 $wfrec(measure(int * int, \%<a, b>. nat-of (\$ - a - b)),$
 $ab,$
 $\%<a, b> f. \text{if } (\#0 \leq a + b \mid b \leq \#0) \text{ then } <\# - 1, a + b>$
 $\text{else } adjust(b, f ' <a, \#2 * b>))$

definition

```
negateSnd :: i => i where
  negateSnd == %<q,r>. <q, $-r>
```

definition

```
divAlg :: i => i where
  divAlg ==
    %<a,b>. if #0 $<= a then
      if #0 $<= b then posDivAlg (<a,b>)
      else if a=#0 then <#0,#0>
      else negateSnd (negDivAlg (<$-a,$-b>))
    else
      if #0$<b then negDivAlg (<a,b>)
      else negateSnd (posDivAlg (<$-a,$-b>))
```

definition

```
zdiv :: [i,i]=>i (infixl zdiv 70) where
  a zdiv b == fst (divAlg (<intify(a), intify(b)>))
```

definition

```
zmod :: [i,i]=>i (infixl zmod 70) where
  a zmod b == snd (divAlg (<intify(a), intify(b)>))
```

lemma *zspos-add-zspos-imp-zspos*: $[\#0 \$< x; \#0 \$< y] \implies \#0 \$< x \$+ y$
 <proof>

lemma *zpos-add-zpos-imp-zpos*: $[\#0 \$<= x; \#0 \$<= y] \implies \#0 \$<= x \$+ y$
 <proof>

lemma *zneg-add-zneg-imp-zneg*: $[x \$< \#0; y \$< \#0] \implies x \$+ y \$< \#0$
 <proof>

lemma *zneg-or-0-add-zneg-or-0-imp-zneg-or-0*:
 $[x \$<= \#0; y \$<= \#0] \implies x \$+ y \$<= \#0$
 <proof>

lemma *zero-lt-zmagnitude*: $[\#0 \$< k; k \in \text{int}] \implies 0 < \text{zmagnitude}(k)$
 <proof>

lemma *zless-add-succ-iff*:

$(w \leq z + \# succ(m)) \leftrightarrow (w \leq z + \#m \mid \text{intify}(w) = z + \#m)$
<proof>

lemma *zadd-succ-lemma*:

$z \in \text{int} \implies (w + \# succ(m) \leq z) \leftrightarrow (w + \#m \leq z)$
<proof>

lemma *zadd-succ-zle-iff*: $(w + \# succ(m) \leq z) \leftrightarrow (w + \#m \leq z)$

<proof>

lemma *zless-add1-iff-zle*: $(w \leq z + \#1) \leftrightarrow (w \leq z)$

<proof>

lemma *add1-zle-iff*: $(w + \#1 \leq z) \leftrightarrow (w \leq z)$

<proof>

lemma *add1-left-zle-iff*: $(\#1 + w \leq z) \leftrightarrow (w \leq z)$

<proof>

lemma *zmult-mono-lemma*: $k \in \text{nat} \implies i \leq j \implies i * \#k \leq j * \#k$

<proof>

lemma *zmult-zle-mono1*: $[[i \leq j; \#0 \leq k]] \implies i * k \leq j * k$

<proof>

lemma *zmult-zle-mono1-neg*: $[[i \leq j; k \leq \#0]] \implies j * k \leq i * k$

<proof>

lemma *zmult-zle-mono2*: $[[i \leq j; \#0 \leq k]] \implies k * i \leq k * j$

<proof>

lemma *zmult-zle-mono2-neg*: $[[i \leq j; k \leq \#0]] \implies k * j \leq k * i$

<proof>

lemma *zmult-zle-mono*:

$[[i \leq j; k \leq l; \#0 \leq j; \#0 \leq k]] \implies i * k \leq j * l$
<proof>

lemma *zmult-zless-mono2-lemma* [*rule-format*]:

$\llbracket i < j; k \in \text{nat} \rrbracket \implies 0 < k \dashrightarrow \#k * i < \#k * j$
 <proof>

lemma *zmult-zless-mono2*: $\llbracket i < j; \#0 < k \rrbracket \implies k * i < k * j$
 <proof>

lemma *zmult-zless-mono1*: $\llbracket i < j; \#0 < k \rrbracket \implies i * k < j * k$
 <proof>

lemma *zmult-zless-mono*:

$\llbracket i < j; k < l; \#0 < j; \#0 < k \rrbracket \implies i * k < j * l$
 <proof>

lemma *zmult-zless-mono1-neg*: $\llbracket i < j; k < \#0 \rrbracket \implies j * k < i * k$
 <proof>

lemma *zmult-zless-mono2-neg*: $\llbracket i < j; k < \#0 \rrbracket \implies k * j < k * i$
 <proof>

lemma *zmult-eq-lemma*:

$\llbracket m \in \text{int}; n \in \text{int} \rrbracket \implies (m = \#0 \mid n = \#0) \leftrightarrow (m * n = \#0)$
 <proof>

lemma *zmult-eq-0-iff* [iff]: $(m * n = \#0) \leftrightarrow (\text{intify}(m) = \#0 \mid \text{intify}(n) = \#0)$
 <proof>

lemma *zmult-zless-lemma*:

$\llbracket k \in \text{int}; m \in \text{int}; n \in \text{int} \rrbracket$
 $\implies (m * k < n * k) \leftrightarrow ((\#0 < k \ \& \ m < n) \mid (k < \#0 \ \& \ n < m))$
 <proof>

lemma *zmult-zless-cancel2*:

$(m * k < n * k) \leftrightarrow ((\#0 < k \ \& \ m < n) \mid (k < \#0 \ \& \ n < m))$
 <proof>

lemma *zmult-zless-cancel1*:

$(k * m < k * n) \leftrightarrow ((\#0 < k \ \& \ m < n) \mid (k < \#0 \ \& \ n < m))$
 <proof>

lemma *zmult-zle-cancel2*:

$(m * k \leq n * k) \leftrightarrow ((\#0 < k \dashrightarrow m \leq n) \ \& \ (k < \#0 \dashrightarrow$

$n \leq m$)
 <proof>

lemma *zmult-zle-cancel1*:

$(k * m \leq k * n) \leftrightarrow ((\#0 < k \rightarrow m \leq n) \& (k < \#0 \rightarrow n \leq m))$
 <proof>

lemma *int-eq-iff-zle*: $[[m \in \text{int}; n \in \text{int}]] \implies m = n \leftrightarrow (m \leq n \& n \leq m)$
 <proof>

lemma *zmult-cancel2-lemma*:

$[[k \in \text{int}; m \in \text{int}; n \in \text{int}]] \implies (m * k = n * k) \leftrightarrow (k \neq \#0 \mid m = n)$
 <proof>

lemma *zmult-cancel2 [simp]*:

$(m * k = n * k) \leftrightarrow (\text{intify}(k) \neq \#0 \mid \text{intify}(m) = \text{intify}(n))$
 <proof>

lemma *zmult-cancel1 [simp]*:

$(k * m = k * n) \leftrightarrow (\text{intify}(k) \neq \#0 \mid \text{intify}(m) = \text{intify}(n))$
 <proof>

32.1 Uniqueness and monotonicity of quotients and remainders

lemma *unique-quotient-lemma*:

$[[b * q' \leq r \& r \leq b * q \& r \neq \#0 \& b \neq \#0 \& r < b]] \implies q' \leq q$
 <proof>

lemma *unique-quotient-lemma-neg*:

$[[b * q' \leq r \& r \leq b * q \& r \neq \#0 \& b \neq \#0 \& b < r']] \implies q \leq q'$
 <proof>

lemma *unique-quotient*:

$[[\text{quorem} \langle a, b \rangle, \langle q, r \rangle; \text{quorem} \langle a, b \rangle, \langle q', r' \rangle; b \in \text{int}; b \neq \#0; q \in \text{int}; q' \in \text{int}]] \implies q = q'$
 <proof>

lemma *unique-remainder*:

$[[\text{quorem} \langle a, b \rangle, \langle q, r \rangle; \text{quorem} \langle a, b \rangle, \langle q', r' \rangle; b \in \text{int}; b \neq \#0; q \in \text{int}; q' \in \text{int}; r \in \text{int}; r' \in \text{int}]] \implies r = r'$
 <proof>

32.2 Correctness of posDivAlg, the Division Algorithm for $a \geq 0$ and $b > 0$

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

$$\text{adjust}(b, \langle q, r \rangle) = (\text{let } \text{diff} = r - b \text{ in} \\ \text{if } \#0 \leq \text{diff} \text{ then } \langle \#2 * q + \#1, \text{diff} \rangle \\ \text{else } \langle \#2 * q, r \rangle)$$

<proof>

lemma *posDivAlg-termination*:

$$[\#0 \leq b; \sim a \leq b] \\ \implies \text{nat-of}(a - \#2 * b) < \text{nat-of}(a - b)$$

<proof>

lemmas *posDivAlg-unfold* = *def-wfrec* [*OF posDivAlg-def wf-measure*]

lemma *posDivAlg-eqn*:

$$[\#0 \leq b; a \in \text{int}; b \in \text{int}] \implies \\ \text{posDivAlg}(\langle a, b \rangle) = \\ (\text{if } a \leq b \text{ then } \langle \#0, a \rangle \text{ else } \text{adjust}(b, \text{posDivAlg}(\langle a, \#2 * b \rangle)))$$

<proof>

lemma *posDivAlg-induct-lemma* [*rule-format*]:

assumes *prem*:

$$!!a \ b. [\#0 \leq a; \#0 \leq b; \\ \sim (a \leq b \mid b \leq \#0) \implies P(\langle a, \#2 * b \rangle)] \implies P(\langle a, b \rangle)$$

shows $\langle u, v \rangle \in \text{int} * \text{int} \implies P(\langle u, v \rangle)$

<proof>

lemma *posDivAlg-induct* [*consumes 2*]:

assumes *u-int*: $u \in \text{int}$

and *v-int*: $v \in \text{int}$

and *ih*: $!!a \ b. [\#0 \leq a; \#0 \leq b; \\ \sim (a \leq b \mid b \leq \#0) \implies P(a, \#2 * b)] \implies P(a, b)$

shows $P(u, v)$

<proof>

lemma *intify-eq-0-iff-zle*: $\text{intify}(m) = \#0 \iff (m \leq \#0 \ \& \ \#0 \leq m)$

<proof>

32.3 Some convenient biconditionals for products of signs

lemma *zmult-pos*: $[\#0 \leq i; \#0 \leq j] \implies \#0 \leq i * j$

<proof>

lemma *zmult-neg*: $[i \leq \#0; j \leq \#0] \implies \#0 \leq i * j$

<proof>

lemma *zmult-pos-neg*: $[[\#0 \ \$< \ i; \ j \ \$< \ \#0]] \implies i \ \$* \ j \ \$< \ \#0$
 $\langle proof \rangle$

lemma *int-0-less-lemma*:
 $[[x \in \text{int}; y \in \text{int}]]$
 $\implies (\#0 \ \$< \ x \ \$* \ y) \ \langle - \rangle \ (\#0 \ \$< \ x \ \& \ \#0 \ \$< \ y \ | \ x \ \$< \ \#0 \ \& \ y \ \$< \ \#0)$
 $\langle proof \rangle$

lemma *int-0-less-mult-iff*:
 $(\#0 \ \$< \ x \ \$* \ y) \ \langle - \rangle \ (\#0 \ \$< \ x \ \& \ \#0 \ \$< \ y \ | \ x \ \$< \ \#0 \ \& \ y \ \$< \ \#0)$
 $\langle proof \rangle$

lemma *int-0-le-lemma*:
 $[[x \in \text{int}; y \in \text{int}]]$
 $\implies (\#0 \ \$\leq \ x \ \$* \ y) \ \langle - \rangle \ (\#0 \ \$\leq \ x \ \& \ \#0 \ \$\leq \ y \ | \ x \ \$\leq \ \#0 \ \& \ y$
 $\ \$\leq \ \#0)$
 $\langle proof \rangle$

lemma *int-0-le-mult-iff*:
 $(\#0 \ \$\leq \ x \ \$* \ y) \ \langle - \rangle \ ((\#0 \ \$\leq \ x \ \& \ \#0 \ \$\leq \ y) \ | \ (x \ \$\leq \ \#0 \ \& \ y \ \$\leq \$
 $\ \#0))$
 $\langle proof \rangle$

lemma *zmult-less-0-iff*:
 $(x \ \$* \ y \ \$< \ \#0) \ \langle - \rangle \ (\#0 \ \$< \ x \ \& \ y \ \$< \ \#0 \ | \ x \ \$< \ \#0 \ \& \ \#0 \ \$< \ y)$
 $\langle proof \rangle$

lemma *zmult-le-0-iff*:
 $(x \ \$* \ y \ \$\leq \ \#0) \ \langle - \rangle \ (\#0 \ \$\leq \ x \ \& \ y \ \$\leq \ \#0 \ | \ x \ \$\leq \ \#0 \ \& \ \#0 \ \$\leq \ y)$
 $\langle proof \rangle$

lemma *posDivAlg-type* [*rule-format*]:
 $[[a \in \text{int}; b \in \text{int}]] \implies \text{posDivAlg}(\langle a, b \rangle) \in \text{int} * \text{int}$
 $\langle proof \rangle$

lemma *posDivAlg-correct* [*rule-format*]:
 $[[a \in \text{int}; b \in \text{int}]]$
 $\implies \#0 \ \$\leq \ a \ \dashrightarrow \ \#0 \ \$< \ b \ \dashrightarrow \ \text{quorem}(\langle a, b \rangle, \text{posDivAlg}(\langle a, b \rangle))$
 $\langle proof \rangle$

32.4 Correctness of `negDivAlg`, the division algorithm for `a;0` and `b;0`

lemma *negDivAlg-termination*:

$[[\#0 \ \$< \ b; \ a \ \$+ \ b \ \$< \ \#0 \]]$
 $\implies \text{nat-of}(\$- \ a \ \$- \ \#2 \ \$* \ b) < \text{nat-of}(\$- \ a \ \$- \ b)$
<proof>

lemmas *negDivAlg-unfold = def-wfrec* [*OF negDivAlg-def wf-measure*]

lemma *negDivAlg-eqn*:

$[[\#0 \ \$< \ b; \ a : \text{int}; \ b : \text{int} \]] \implies$
 $\text{negDivAlg}(\langle a, b \rangle) =$
(if $\#0 \ \$\leq a \ \$+ b$ *then* $\langle \#-1, a \ \$+ b \rangle$
else $\text{adjust}(b, \text{negDivAlg}(\langle a, \#2 \ \$* b \rangle))$ *)*
<proof>

lemma *negDivAlg-induct-lemma* [*rule-format*]:

assumes *prem*:
 $!!a \ b. [[a \in \text{int}; \ b \in \text{int};$
 $\sim (\#0 \ \$\leq a \ \$+ b \mid b \ \$\leq \#0) \ \longrightarrow \ P(\langle a, \#2 \ \$* b \rangle)]]$
 $\implies P(\langle a, b \rangle)$
shows $\langle u, v \rangle \in \text{int} * \text{int} \ \longrightarrow \ P(\langle u, v \rangle)$
<proof>

lemma *negDivAlg-induct* [*consumes 2*]:

assumes *u-int*: $u \in \text{int}$
and *v-int*: $v \in \text{int}$
and *ih*: $!!a \ b. [[a \in \text{int}; \ b \in \text{int};$
 $\sim (\#0 \ \$\leq a \ \$+ b \mid b \ \$\leq \#0) \ \longrightarrow \ P(a, \#2 \ \$* b)]]$
 $\implies P(a, b)$
shows $P(u, v)$
<proof>

lemma *negDivAlg-type*:

$[[a \in \text{int}; \ b \in \text{int} \]] \implies \text{negDivAlg}(\langle a, b \rangle) \in \text{int} * \text{int}$
<proof>

lemma *negDivAlg-correct* [*rule-format*]:

$[[a \in \text{int}; \ b \in \text{int} \]]$
 $\implies a \ \$< \ \#0 \ \longrightarrow \ \#0 \ \$< \ b \ \longrightarrow \ \text{quorem}(\langle a, b \rangle, \text{negDivAlg}(\langle a, b \rangle))$
<proof>

32.5 Existence shown by proving the division algorithm to be correct

lemma *quorem-0*: $[[b \neq \#0; b \in \text{int}]] \implies \text{quorem} (\langle \#0, b \rangle, \langle \#0, \#0 \rangle)$
 $\langle \text{proof} \rangle$

lemma *posDivAlg-zero-divisor*: $\text{posDivAlg}(\langle a, \#0 \rangle) = \langle \#0, a \rangle$
 $\langle \text{proof} \rangle$

lemma *posDivAlg-0* [simp]: $\text{posDivAlg} (\langle \#0, b \rangle) = \langle \#0, \#0 \rangle$
 $\langle \text{proof} \rangle$

lemma *linear-arith-lemma*: $\sim (\#0 \ \$\leq \ \#-1 \ \$+ \ b) \implies (b \ \$\leq \ \#0)$
 $\langle \text{proof} \rangle$

lemma *negDivAlg-minus1* [simp]: $\text{negDivAlg} (\langle \#-1, b \rangle) = \langle \#-1, b \ \$- \ \#1 \rangle$
 $\langle \text{proof} \rangle$

lemma *negateSnd-eq* [simp]: $\text{negateSnd} (\langle q, r \rangle) = \langle q, \ \$-r \rangle$
 $\langle \text{proof} \rangle$

lemma *negateSnd-type*: $qr \in \text{int} * \text{int} \implies \text{negateSnd} (qr) \in \text{int} * \text{int}$
 $\langle \text{proof} \rangle$

lemma *quorem-neg*:
 $[[\text{quorem} (\langle \ \$-a, \ \$-b \rangle, qr); a \in \text{int}; b \in \text{int}; qr \in \text{int} * \text{int}]]$
 $\implies \text{quorem} (\langle a, b \rangle, \text{negateSnd}(qr))$
 $\langle \text{proof} \rangle$

lemma *divAlg-correct*:
 $[[b \neq \#0; a \in \text{int}; b \in \text{int}]] \implies \text{quorem} (\langle a, b \rangle, \text{divAlg}(\langle a, b \rangle))$
 $\langle \text{proof} \rangle$

lemma *divAlg-type*: $[[a \in \text{int}; b \in \text{int}]] \implies \text{divAlg}(\langle a, b \rangle) \in \text{int} * \text{int}$
 $\langle \text{proof} \rangle$

lemma *zdiv-intify1* [simp]: $\text{intify}(x) \text{ zdiv } y = x \text{ zdiv } y$
 $\langle \text{proof} \rangle$

lemma *zdiv-intify2* [simp]: $x \text{ zdiv } \text{intify}(y) = x \text{ zdiv } y$
 $\langle \text{proof} \rangle$

lemma *zdiv-type* [iff, TC]: $z \text{ zdiv } w \in \text{int}$
 $\langle \text{proof} \rangle$

lemma *zmod-intify1* [*simp*]: $\text{intify}(x) \text{ zmod } y = x \text{ zmod } y$
(*proof*)

lemma *zmod-intify2* [*simp*]: $x \text{ zmod } \text{intify}(y) = x \text{ zmod } y$
(*proof*)

lemma *zmod-type* [*iff, TC*]: $z \text{ zmod } w \in \text{int}$
(*proof*)

lemma *DIVISION-BY-ZERO-ZDIV*: $a \text{ zdiv } \#0 = \#0$
(*proof*)

lemma *DIVISION-BY-ZERO-ZMOD*: $a \text{ zmod } \#0 = \text{intify}(a)$
(*proof*)

lemma *raw-zmod-zdiv-equality*:
[[$a \in \text{int}; b \in \text{int}$]] $\implies a = b \$* (a \text{ zdiv } b) \$+ (a \text{ zmod } b)$
(*proof*)

lemma *zmod-zdiv-equality*: $\text{intify}(a) = b \$* (a \text{ zdiv } b) \$+ (a \text{ zmod } b)$
(*proof*)

lemma *pos-mod*: $\#0 \$< b \implies \#0 \$<= a \text{ zmod } b \ \& \ a \text{ zmod } b \$< b$
(*proof*)

lemmas *pos-mod-sign* = *pos-mod* [*THEN conjunct1, standard*]
and *pos-mod-bound* = *pos-mod* [*THEN conjunct2, standard*]

lemma *neg-mod*: $b \$< \#0 \implies a \text{ zmod } b \$<= \#0 \ \& \ b \$< a \text{ zmod } b$
(*proof*)

lemmas *neg-mod-sign* = *neg-mod* [*THEN conjunct1, standard*]
and *neg-mod-bound* = *neg-mod* [*THEN conjunct2, standard*]

lemma *quorem-div-mod*:
[[$b \neq \#0; a \in \text{int}; b \in \text{int}$]]
 $\implies \text{quorem } \langle a, b \rangle, \langle a \text{ zdiv } b, a \text{ zmod } b \rangle$
(*proof*)

lemma *quorem-div*:

$[[\text{quorem}(\langle a, b \rangle, \langle q, r \rangle); b \neq \#0; a \in \text{int}; b \in \text{int}; q \in \text{int}]]$
 $\implies a \text{ zdiv } b = q$

<proof>

lemma *quorem-mod*:

$[[\text{quorem}(\langle a, b \rangle, \langle q, r \rangle); b \neq \#0; a \in \text{int}; b \in \text{int}; q \in \text{int}; r \in \text{int}]]$
 $\implies a \text{ zmod } b = r$

<proof>

lemma *zdiv-pos-pos-trivial-raw*:

$[[a \in \text{int}; b \in \text{int}; \#0 \leq a; a < b]]$ $\implies a \text{ zdiv } b = \#0$

<proof>

lemma *zdiv-pos-pos-trivial*: $[[\#0 \leq a; a < b]]$ $\implies a \text{ zdiv } b = \#0$

<proof>

lemma *zdiv-neg-neg-trivial-raw*:

$[[a \in \text{int}; b \in \text{int}; a \leq \#0; b < a]]$ $\implies a \text{ zdiv } b = \#0$

<proof>

lemma *zdiv-neg-neg-trivial*: $[[a \leq \#0; b < a]]$ $\implies a \text{ zdiv } b = \#0$

<proof>

lemma *zadd-le-0-lemma*: $[[a+b \leq \#0; \#0 < a; \#0 < b]]$ $\implies \text{False}$

<proof>

lemma *zdiv-pos-neg-trivial-raw*:

$[[a \in \text{int}; b \in \text{int}; \#0 < a; a+b \leq \#0]]$ $\implies a \text{ zdiv } b = \#-1$

<proof>

lemma *zdiv-pos-neg-trivial*: $[[\#0 < a; a+b \leq \#0]]$ $\implies a \text{ zdiv } b = \#-1$

<proof>

lemma *zmod-pos-pos-trivial-raw*:

$[[a \in \text{int}; b \in \text{int}; \#0 \leq a; a < b]]$ $\implies a \text{ zmod } b = a$

<proof>

lemma *zmod-pos-pos-trivial*: $[[\#0 \leq a; a < b]]$ $\implies a \text{ zmod } b = \text{intify}(a)$

<proof>

lemma *zmod-neg-neg-trivial-raw*:

$[[a \in \text{int}; b \in \text{int}; a \leq \#0; b < a]]$ $\implies a \text{ zmod } b = a$

<proof>

lemma *zmod-neg-neg-trivial*: $[[a \leq 0; b < a]] \implies a \text{ zmod } b = \text{intify}(a)$
 <proof>

lemma *zmod-pos-neg-trivial-raw*:
 $[[a \in \text{int}; b \in \text{int}; \#0 \leq a; a+b \leq \#0]] \implies a \text{ zmod } b = a+b$
 <proof>

lemma *zmod-pos-neg-trivial*: $[[\#0 < a; a+b \leq \#0]] \implies a \text{ zmod } b = a+b$
 <proof>

lemma *zdiv-zminus-zminus-raw*:
 $[[a \in \text{int}; b \in \text{int}]] \implies (\$-a) \text{ zdiv } (\$-b) = a \text{ zdiv } b$
 <proof>

lemma *zdiv-zminus-zminus [simp]*: $(\$-a) \text{ zdiv } (\$-b) = a \text{ zdiv } b$
 <proof>

lemma *zmod-zminus-zminus-raw*:
 $[[a \in \text{int}; b \in \text{int}]] \implies (\$-a) \text{ zmod } (\$-b) = \$- (a \text{ zmod } b)$
 <proof>

lemma *zmod-zminus-zminus [simp]*: $(\$-a) \text{ zmod } (\$-b) = \$- (a \text{ zmod } b)$
 <proof>

32.6 division of a number by itself

lemma *self-quotient-aux1*: $[[\#0 < a; a = r + a*q; r < a]] \implies \#1 \leq q$
 <proof>

lemma *self-quotient-aux2*: $[[\#0 < a; a = r + a*q; \#0 \leq r]] \implies q \leq \#1$
 <proof>

lemma *self-quotient*:
 $[[\text{quorem}(\langle a, a \rangle, \langle q, r \rangle); a \in \text{int}; q \in \text{int}; a \neq \#0]] \implies q = \#1$
 <proof>

lemma *self-remainder*:
 $[[\text{quorem}(\langle a, a \rangle, \langle q, r \rangle); a \in \text{int}; q \in \text{int}; r \in \text{int}; a \neq \#0]] \implies r = \#0$
 <proof>

lemma *zdiv-self-raw*: $[a \neq \#0; a \in \text{int}] \implies a \text{ zdiv } a = \#1$
 ⟨*proof*⟩

lemma *zdiv-self* [*simp*]: $\text{intify}(a) \neq \#0 \implies a \text{ zdiv } a = \#1$
 ⟨*proof*⟩

lemma *zmod-self-raw*: $a \in \text{int} \implies a \text{ zmod } a = \#0$
 ⟨*proof*⟩

lemma *zmod-self* [*simp*]: $a \text{ zmod } a = \#0$
 ⟨*proof*⟩

32.7 Computation of division and remainder

lemma *zdiv-zero* [*simp*]: $\#0 \text{ zdiv } b = \#0$
 ⟨*proof*⟩

lemma *zdiv-eq-minus1*: $\#0 \ \$< b \implies \#-1 \text{ zdiv } b = \#-1$
 ⟨*proof*⟩

lemma *zmod-zero* [*simp*]: $\#0 \text{ zmod } b = \#0$
 ⟨*proof*⟩

lemma *zdiv-minus1*: $\#0 \ \$< b \implies \#-1 \text{ zdiv } b = \#-1$
 ⟨*proof*⟩

lemma *zmod-minus1*: $\#0 \ \$< b \implies \#-1 \text{ zmod } b = b \ \$- \#1$
 ⟨*proof*⟩

lemma *zdiv-pos-pos*: $[\#0 \ \$< a; \#0 \ \$\leq b]$
 $\implies a \text{ zdiv } b = \text{fst } (\text{posDivAlg}(\langle \text{intify}(a), \text{intify}(b) \rangle))$
 ⟨*proof*⟩

lemma *zmod-pos-pos*:
 $[\#0 \ \$< a; \#0 \ \$\leq b]$
 $\implies a \text{ zmod } b = \text{snd } (\text{posDivAlg}(\langle \text{intify}(a), \text{intify}(b) \rangle))$
 ⟨*proof*⟩

lemma *zdiv-neg-pos*:
 $[a \ \$< \#0; \#0 \ \$< b]$
 $\implies a \text{ zdiv } b = \text{fst } (\text{negDivAlg}(\langle \text{intify}(a), \text{intify}(b) \rangle))$
 ⟨*proof*⟩

lemma *zmod-neg-pos*:

$$\begin{aligned} & [[a \neq 0; \neq 0 \leq b]] \\ & \implies a \text{ zmod } b = \text{snd } (\text{negDivAlg}(\langle \text{intify}(a), \text{intify}(b) \rangle)) \end{aligned}$$
 <proof>

lemma *zdiv-pos-neg*:

$$\begin{aligned} & [[\neq 0 \leq a; b \leq \neq 0]] \\ & \implies a \text{ zdiv } b = \text{fst } (\text{negateSnd}(\text{negDivAlg}(\langle \text{\$-}a, \text{\$-}b \rangle))) \end{aligned}$$
 <proof>

lemma *zmod-pos-neg*:

$$\begin{aligned} & [[\neq 0 \leq a; b \leq \neq 0]] \\ & \implies a \text{ zmod } b = \text{snd } (\text{negateSnd}(\text{negDivAlg}(\langle \text{\$-}a, \text{\$-}b \rangle))) \end{aligned}$$
 <proof>

lemma *zdiv-neg-neg*:

$$\begin{aligned} & [[a \leq \neq 0; b \leq \neq 0]] \\ & \implies a \text{ zdiv } b = \text{fst } (\text{negateSnd}(\text{posDivAlg}(\langle \text{\$-}a, \text{\$-}b \rangle))) \end{aligned}$$
 <proof>

lemma *zmod-neg-neg*:

$$\begin{aligned} & [[a \leq \neq 0; b \leq \neq 0]] \\ & \implies a \text{ zmod } b = \text{snd } (\text{negateSnd}(\text{posDivAlg}(\langle \text{\$-}a, \text{\$-}b \rangle))) \end{aligned}$$
 <proof>

declare *zdiv-pos-pos* [of integ-of (v) integ-of (w), standard, simp]
declare *zdiv-neg-pos* [of integ-of (v) integ-of (w), standard, simp]
declare *zdiv-pos-neg* [of integ-of (v) integ-of (w), standard, simp]
declare *zdiv-neg-neg* [of integ-of (v) integ-of (w), standard, simp]
declare *zmod-pos-pos* [of integ-of (v) integ-of (w), standard, simp]
declare *zmod-neg-pos* [of integ-of (v) integ-of (w), standard, simp]
declare *zmod-pos-neg* [of integ-of (v) integ-of (w), standard, simp]
declare *zmod-neg-neg* [of integ-of (v) integ-of (w), standard, simp]
declare *posDivAlg-eqn* [of concl: integ-of (v) integ-of (w), standard, simp]
declare *negDivAlg-eqn* [of concl: integ-of (v) integ-of (w), standard, simp]

lemma *zmod-1* [simp]: $a \text{ zmod } \#1 = \#0$
 <proof>

lemma *zdiv-1* [simp]: $a \text{ zdiv } \#1 = \text{intify}(a)$
 <proof>

lemma *zmod-minus1-right* [simp]: $a \text{ zmod } \#-1 = \#0$

<proof>

lemma *zdiv-minus1-right-raw*: $a \in \text{int} \implies a \text{ zdiv } \#-1 = \$-a$
<proof>

lemma *zdiv-minus1-right*: $a \text{ zdiv } \#-1 = \$-a$
<proof>

declare *zdiv-minus1-right* [*simp*]

32.8 Monotonicity in the first argument (divisor)

lemma *zdiv-mono1*: $[[a \leq a'; \#0 < b]] \implies a \text{ zdiv } b \leq a' \text{ zdiv } b$
<proof>

lemma *zdiv-mono1-neg*: $[[a \leq a'; b < \#0]] \implies a' \text{ zdiv } b \leq a \text{ zdiv } b$
<proof>

32.9 Monotonicity in the second argument (dividend)

lemma *q-pos-lemma*:

$[[\#0 \leq b * q' + r'; r' < b'; \#0 < b']] \implies \#0 \leq q'$
<proof>

lemma *zdiv-mono2-lemma*:

$[[b * q + r = b * q' + r'; \#0 \leq b * q' + r';$
 $r' < b'; \#0 \leq r; \#0 < b'; b' \leq b]]$
 $\implies q \leq q'$
<proof>

lemma *zdiv-mono2-raw*:

$[[\#0 \leq a; \#0 < b'; b' \leq b; a \in \text{int}]]$
 $\implies a \text{ zdiv } b \leq a \text{ zdiv } b'$
<proof>

lemma *zdiv-mono2*:

$[[\#0 \leq a; \#0 < b'; b' \leq b]]$
 $\implies a \text{ zdiv } b \leq a \text{ zdiv } b'$
<proof>

lemma *q-neg-lemma*:

$[[b * q' + r' < \#0; \#0 \leq r'; \#0 < b']] \implies q' < \#0$
<proof>

lemma *zdiv-mono2-neg-lemma*:

$[[b * q + r = b * q' + r'; b * q' + r' < \#0;$
 $r < b; \#0 \leq r'; \#0 < b'; b' \leq b]]$
 $\implies q' \leq q$

<proof>

lemma *zdiv-mono2-neg-raw*:

$$\begin{aligned} & [[a \ \$< \ #0; \ #0 \ \$< \ b'; \ b' \ \$\leq \ b; \ a \in \ int \]] \\ & \implies a \ zdiv \ b' \ \$\leq \ a \ zdiv \ b \end{aligned}$$

<proof>

lemma *zdiv-mono2-neg*: $[[a \ \$< \ #0; \ #0 \ \$< \ b'; \ b' \ \$\leq \ b \]]$

$$\implies a \ zdiv \ b' \ \$\leq \ a \ zdiv \ b$$

<proof>

32.10 More algebraic laws for zdiv and zmod

lemma *zmult1-lemma*:

$$\begin{aligned} & [[\text{quorem}(\langle b, c \rangle, \langle q, r \rangle); \ c \in \ int; \ c \neq \ #0 \]] \\ & \implies \text{quorem}(\langle a\$*b, c \rangle, \langle a\$*q \ \$+ \ (a\$*r) \ zdiv \ c, \ (a\$*r) \ zmod \ c \rangle) \end{aligned}$$

<proof>

lemma *zdiv-zmult1-eq-raw*:

$$\begin{aligned} & [[b \in \ int; \ c \in \ int \]] \\ & \implies (a\$*b) \ zdiv \ c = a\$*(b \ zdiv \ c) \ \$+ \ a\$*(b \ zmod \ c) \ zdiv \ c \end{aligned}$$

<proof>

lemma *zdiv-zmult1-eq*: $(a\$*b) \ zdiv \ c = a\$*(b \ zdiv \ c) \ \$+ \ a\$*(b \ zmod \ c) \ zdiv \ c$

<proof>

lemma *zmod-zmult1-eq-raw*:

$$[[b \in \ int; \ c \in \ int \]] \implies (a\$*b) \ zmod \ c = a\$*(b \ zmod \ c) \ zmod \ c$$

<proof>

lemma *zmod-zmult1-eq*: $(a\$*b) \ zmod \ c = a\$*(b \ zmod \ c) \ zmod \ c$

<proof>

lemma *zmod-zmult1-eq'*: $(a\$*b) \ zmod \ c = ((a \ zmod \ c) \ \$* \ b) \ zmod \ c$

<proof>

lemma *zmod-zmult-distrib*: $(a\$*b) \ zmod \ c = ((a \ zmod \ c) \ \$* \ (b \ zmod \ c)) \ zmod \ c$

<proof>

lemma *zdiv-zmult-self1* [*simp*]: $\text{intify}(b) \neq \ #0 \implies (a\$*b) \ zdiv \ b = \text{intify}(a)$

<proof>

lemma *zdiv-zmult-self2* [*simp*]: $\text{intify}(b) \neq \ #0 \implies (b\$*a) \ zdiv \ b = \text{intify}(a)$

<proof>

lemma *zmod-zmult-self1* [*simp*]: $(a\$*b) \ zmod \ b = \ #0$

<proof>

lemma *zmod-zmult-self2* [*simp*]: $(b\$*a) \ zmod \ b = \ #0$

$\langle proof \rangle$

lemma *zadd1-lemma*:

$[[\text{quorem}(\langle a, c \rangle, \langle aq, ar \rangle); \text{quorem}(\langle b, c \rangle, \langle bq, br \rangle);$
 $c \in \text{int}; c \neq \#0]]$

$\implies \text{quorem}(\langle a\$+b, c \rangle, \langle aq \$+ bq \$+ (ar\$+br) \text{zdiv } c, (ar\$+br) \text{zmod } c \rangle)$
 $\langle proof \rangle$

lemma *zdiv-zadd1-eq-raw*:

$[[a \in \text{int}; b \in \text{int}; c \in \text{int}]] \implies$

$(a\$+b) \text{zdiv } c = a \text{zdiv } c \$+ b \text{zdiv } c \$+ ((a \text{zmod } c \$+ b \text{zmod } c) \text{zdiv } c)$
 $\langle proof \rangle$

lemma *zdiv-zadd1-eq*:

$(a\$+b) \text{zdiv } c = a \text{zdiv } c \$+ b \text{zdiv } c \$+ ((a \text{zmod } c \$+ b \text{zmod } c) \text{zdiv } c)$
 $\langle proof \rangle$

lemma *zmod-zadd1-eq-raw*:

$[[a \in \text{int}; b \in \text{int}; c \in \text{int}]]$

$\implies (a\$+b) \text{zmod } c = (a \text{zmod } c \$+ b \text{zmod } c) \text{zmod } c$
 $\langle proof \rangle$

lemma *zmod-zadd1-eq*: $(a\$+b) \text{zmod } c = (a \text{zmod } c \$+ b \text{zmod } c) \text{zmod } c$

$\langle proof \rangle$

lemma *zmod-div-trivial-raw*:

$[[a \in \text{int}; b \in \text{int}]] \implies (a \text{zmod } b) \text{zdiv } b = \#0$

$\langle proof \rangle$

lemma *zmod-div-trivial [simp]*: $(a \text{zmod } b) \text{zdiv } b = \#0$

$\langle proof \rangle$

lemma *zmod-mod-trivial-raw*:

$[[a \in \text{int}; b \in \text{int}]] \implies (a \text{zmod } b) \text{zmod } b = a \text{zmod } b$

$\langle proof \rangle$

lemma *zmod-mod-trivial [simp]*: $(a \text{zmod } b) \text{zmod } b = a \text{zmod } b$

$\langle proof \rangle$

lemma *zmod-zadd-left-eq*: $(a\$+b) \text{zmod } c = ((a \text{zmod } c) \$+ b) \text{zmod } c$

$\langle proof \rangle$

lemma *zmod-zadd-right-eq*: $(a\$+b) \text{zmod } c = (a \$+ (b \text{zmod } c)) \text{zmod } c$

$\langle proof \rangle$

lemma *zdiv-zadd-self1* [*simp*]:

$\text{intify}(a) \neq \#0 \implies (a+b) \text{zdiv } a = b \text{zdiv } a \text{ } \$+ \#1$
<proof>

lemma *zdiv-zadd-self2* [*simp*]:

$\text{intify}(a) \neq \#0 \implies (b+a) \text{zdiv } a = b \text{zdiv } a \text{ } \$+ \#1$
<proof>

lemma *zmod-zadd-self1* [*simp*]: $(a+b) \text{zmod } a = b \text{zmod } a$

<proof>

lemma *zmod-zadd-self2* [*simp*]: $(b+a) \text{zmod } a = b \text{zmod } a$

<proof>

32.11 proving $a \text{zdiv } (b*c) = (a \text{zdiv } b) \text{zdiv } c$

lemma *zdiv-zmult2-aux1*:

$[[\#0 \text{ } \$< c; b \text{ } \$< r; r \text{ } \$\leq \#0]] \implies b*c \text{ } \$< b*(q \text{zmod } c) \text{ } \$+ r$
<proof>

lemma *zdiv-zmult2-aux2*:

$[[\#0 \text{ } \$< c; b \text{ } \$< r; r \text{ } \$\leq \#0]] \implies b \text{ } \$* (q \text{zmod } c) \text{ } \$+ r \text{ } \$\leq \#0$
<proof>

lemma *zdiv-zmult2-aux3*:

$[[\#0 \text{ } \$< c; \#0 \text{ } \$\leq r; r \text{ } \$< b]] \implies \#0 \text{ } \$\leq b \text{ } \$* (q \text{zmod } c) \text{ } \$+ r$
<proof>

lemma *zdiv-zmult2-aux4*:

$[[\#0 \text{ } \$< c; \#0 \text{ } \$\leq r; r \text{ } \$< b]] \implies b \text{ } \$* (q \text{zmod } c) \text{ } \$+ r \text{ } \$< b \text{ } \$* c$
<proof>

lemma *zdiv-zmult2-lemma*:

$[[\text{quorem } (\langle a, b \rangle, \langle q, r \rangle); a \in \text{int}; b \in \text{int}; b \neq \#0; \#0 \text{ } \$< c]]$
 $\implies \text{quorem } (\langle a, b*c \rangle, \langle q \text{zdiv } c, b*(q \text{zmod } c) \text{ } \$+ r \rangle)$
<proof>

lemma *zdiv-zmult2-eq-raw*:

$[[\#0 \text{ } \$< c; a \in \text{int}; b \in \text{int}]] \implies a \text{zdiv } (b*c) = (a \text{zdiv } b) \text{zdiv } c$
<proof>

lemma *zdiv-zmult2-eq*: $\#0 \text{ } \$< c \implies a \text{zdiv } (b*c) = (a \text{zdiv } b) \text{zdiv } c$

<proof>

lemma *zmod-zmult2-eq-raw*:

$[[\#0 \text{ } \$< c; a \in \text{int}; b \in \text{int}]]$
 $\implies a \text{zmod } (b*c) = b*(a \text{zdiv } b \text{zmod } c) \text{ } \$+ a \text{zmod } b$

<proof>

lemma *zmod-zmult2-eq*:

$\#0 \ \$< c \implies a \ zmod \ (b\$*c) = b\$*(a \ zdiv \ b \ zmod \ c) \ \$+ \ a \ zmod \ b$
<proof>

32.12 Cancellation of common factors in "zdiv"

lemma *zdiv-zmult-zmult1-aux1*:

$[[\#0 \ \$< b; \ intify(c) \neq \#0 \]] \implies (c\$*a) \ zdiv \ (c\$*b) = a \ zdiv \ b$
<proof>

lemma *zdiv-zmult-zmult1-aux2*:

$[[b \ \$< \#0; \ intify(c) \neq \#0 \]] \implies (c\$*a) \ zdiv \ (c\$*b) = a \ zdiv \ b$
<proof>

lemma *zdiv-zmult-zmult1-raw*:

$[[intify(c) \neq \#0; \ b \in \ int \]] \implies (c\$*a) \ zdiv \ (c\$*b) = a \ zdiv \ b$
<proof>

lemma *zdiv-zmult-zmult1*: $intify(c) \neq \#0 \implies (c\$*a) \ zdiv \ (c\$*b) = a \ zdiv \ b$
<proof>

lemma *zdiv-zmult-zmult2*: $intify(c) \neq \#0 \implies (a\$*c) \ zdiv \ (b\$*c) = a \ zdiv \ b$
<proof>

32.13 Distribution of factors over "zmod"

lemma *zmod-zmult-zmult1-aux1*:

$[[\#0 \ \$< b; \ intify(c) \neq \#0 \]] \implies (c\$*a) \ zmod \ (c\$*b) = c \ \$* \ (a \ zmod \ b)$
<proof>

lemma *zmod-zmult-zmult1-aux2*:

$[[b \ \$< \#0; \ intify(c) \neq \#0 \]] \implies (c\$*a) \ zmod \ (c\$*b) = c \ \$* \ (a \ zmod \ b)$
<proof>

lemma *zmod-zmult-zmult1-raw*:

$[[b \in \ int; \ c \in \ int \]] \implies (c\$*a) \ zmod \ (c\$*b) = c \ \$* \ (a \ zmod \ b)$
<proof>

lemma *zmod-zmult-zmult1*: $(c\$*a) \ zmod \ (c\$*b) = c \ \$* \ (a \ zmod \ b)$
<proof>

lemma *zmod-zmult-zmult2*: $(a\$*c) \ zmod \ (b\$*c) = (a \ zmod \ b) \ \$* \ c$
<proof>

lemma *zdiv-neg-pos-less0*: $[| a \mathbb{Z} < \#0; \#0 \mathbb{Z} < b |] \implies a \text{ zdiv } b \mathbb{Z} < \#0$
<proof>

lemma *zdiv-nonneg-neg-le0*: $[| \#0 \mathbb{Z} \leq a; b \mathbb{Z} < \#0 |] \implies a \text{ zdiv } b \mathbb{Z} \leq \#0$
<proof>

lemma *pos-imp-zdiv-nonneg-iff*: $\#0 \mathbb{Z} < b \implies (\#0 \mathbb{Z} \leq a \text{ zdiv } b) \iff (\#0 \mathbb{Z} \leq a)$
<proof>

lemma *neg-imp-zdiv-nonneg-iff*: $b \mathbb{Z} < \#0 \implies (\#0 \mathbb{Z} \leq a \text{ zdiv } b) \iff (a \mathbb{Z} \leq \#0)$
<proof>

lemma *pos-imp-zdiv-neg-iff*: $\#0 \mathbb{Z} < b \implies (a \text{ zdiv } b \mathbb{Z} < \#0) \iff (a \mathbb{Z} < \#0)$
<proof>

lemma *neg-imp-zdiv-neg-iff*: $b \mathbb{Z} < \#0 \implies (a \text{ zdiv } b \mathbb{Z} < \#0) \iff (\#0 \mathbb{Z} < a)$
<proof>

end

33 CardinalArith: Cardinal Arithmetic Without the Axiom of Choice

theory *CardinalArith* **imports** *Cardinal OrderArith ArithSimp Finite* **begin**

definition

InfCard $:: i \implies o$ **where**
InfCard(*i*) == *Card*(*i*) & *nat le i*

definition

cmult $:: [i, i] \implies i$ (**infixl** $|*|$ 70) **where**
 $i |*| j == |i*j|$

definition

cadd $:: [i, i] \implies i$ (**infixl** $|+|$ 65) **where**
 $i |+| j == |i+j|$

definition

csquare-rel $:: i \implies i$ **where**

$csquare-rel(K) ==$
 $rvimage(K * K,$
 $lam <x,y>:K * K. <x Un y, x, y>,$
 $rmult(K, Memrel(K), K * K, rmult(K, Memrel(K), K, Memrel(K))))$

definition

$jump-cardinal :: i ==> i$ **where**

— This def is more complex than Kunen’s but it more easily proved to be a cardinal

$jump-cardinal(K) ==$
 $\bigcup X \in Pow(K). \{z. r: Pow(K * K), well-ord(X, r) \ \& \ z = ordertype(X, r)\}$

definition

$csucc :: i ==> i$ **where**

— needed because $jump-cardinal(K)$ might not be the successor of K

$csucc(K) == LEAST L. Card(L) \ \& \ K < L$

notation (*xsymbols output*)

$cadd$ (**infixl** \oplus 65) **and**

$cmult$ (**infixl** \otimes 70)

notation (*HTML output*)

$cadd$ (**infixl** \oplus 65) **and**

$cmult$ (**infixl** \otimes 70)

lemma *Card-Union* [*simp,intro,TC*]: $(ALL \ x:A. Card(x)) ==> Card(Union(A))$
 $\langle proof \rangle$

lemma *Card-UN*: $(!!x. x:A ==> Card(K(x))) ==> Card(\bigcup x \in A. K(x))$
 $\langle proof \rangle$

lemma *Card-OUN* [*simp,intro,TC*]:

$(!!x. x:A ==> Card(K(x))) ==> Card(\bigcup x < A. K(x))$

$\langle proof \rangle$

lemma *n-lesspoll-nat*: $n \in nat ==> n < nat$

$\langle proof \rangle$

lemma *in-Card-imp-lesspoll*: $[| Card(K); b \in K |] ==> b < K$

$\langle proof \rangle$

lemma *lesspoll-lemma*: $[| \sim A < B; C < B |] ==> A - C \neq 0$

$\langle proof \rangle$

33.1 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.

33.1.1 Cardinal addition is commutative

lemma *sum-commute-epoll*: $A+B \approx B+A$
<proof>

lemma *cadd-commute*: $i \mid+ j = j \mid+ i$
<proof>

33.1.2 Cardinal addition is associative

lemma *sum-assoc-epoll*: $(A+B)+C \approx A+(B+C)$
<proof>

lemma *well-ord-cadd-assoc*:
 $\llbracket \text{well-ord}(i,ri); \text{well-ord}(j,rj); \text{well-ord}(k,rk) \rrbracket$
 $\implies (i \mid+ j) \mid+ k = i \mid+ (j \mid+ k)$
<proof>

33.1.3 0 is the identity for addition

lemma *sum-0-epoll*: $0+A \approx A$
<proof>

lemma *cadd-0 [simp]*: $\text{Card}(K) \implies 0 \mid+ K = K$
<proof>

33.1.4 Addition by another cardinal

lemma *sum-lepoll-self*: $A \lesssim A+B$
<proof>

lemma *cadd-le-self*:
 $\llbracket \text{Card}(K); \text{Ord}(L) \rrbracket \implies K \text{ le } (K \mid+ L)$
<proof>

33.1.5 Monotonicity of addition

lemma *sum-lepoll-mono*:
 $\llbracket A \lesssim C; B \lesssim D \rrbracket \implies A+B \lesssim C+D$
<proof>

lemma *cadd-le-mono*:
 $\llbracket K' \text{ le } K; L' \text{ le } L \rrbracket \implies (K' \mid+ L') \text{ le } (K \mid+ L)$
<proof>

33.1.6 Addition of finite cardinals is "ordinary" addition

lemma *sum-succ-epoll*: $\text{succ}(A)+B \approx \text{succ}(A+B)$

<proof>

lemma *cadd-succ-lemma*:

$[| \text{Ord}(m); \text{Ord}(n) |] \implies \text{succ}(m) \mid + \mid n = \mid \text{succ}(m \mid + \mid n) \mid$

<proof>

lemma *nat-cadd-ep-add*: $[| m: \text{nat}; n: \text{nat} |] \implies m \mid + \mid n = m \# + n$

<proof>

33.2 Cardinal multiplication

33.2.1 Cardinal multiplication is commutative

lemma *prod-commute-epoll*: $A*B \approx B*A$

<proof>

lemma *cmult-commute*: $i \mid * \mid j = j \mid * \mid i$

<proof>

33.2.2 Cardinal multiplication is associative

lemma *prod-assoc-epoll*: $(A*B)*C \approx A*(B*C)$

<proof>

lemma *well-ord-cmult-assoc*:

$[| \text{well-ord}(i,ri); \text{well-ord}(j,rj); \text{well-ord}(k,rk) |]$

$\implies (i \mid * \mid j) \mid * \mid k = i \mid * \mid (j \mid * \mid k)$

<proof>

33.2.3 Cardinal multiplication distributes over addition

lemma *sum-prod-distrib-epoll*: $(A+B)*C \approx (A*C)+(B*C)$

<proof>

lemma *well-ord-cadd-cmult-distrib*:

$[| \text{well-ord}(i,ri); \text{well-ord}(j,rj); \text{well-ord}(k,rk) |]$

$\implies (i \mid + \mid j) \mid * \mid k = (i \mid * \mid k) \mid + \mid (j \mid * \mid k)$

<proof>

33.2.4 Multiplication by 0 yields 0

lemma *prod-0-epoll*: $0*A \approx 0$

<proof>

lemma *cmult-0 [simp]*: $0 \mid * \mid i = 0$

<proof>

33.2.5 1 is the identity for multiplication

lemma *prod-singleton-epoll*: $\{x\} * A \approx A$

<proof>

lemma *cmult-1 [simp]*: $\text{Card}(K) ==> 1 |*| K = K$

<proof>

33.3 Some inequalities for multiplication

lemma *prod-square-lepoll*: $A \lesssim A * A$

<proof>

lemma *cmult-square-le*: $\text{Card}(K) ==> K \text{ le } K |*| K$

<proof>

33.3.1 Multiplication by a non-zero cardinal

lemma *prod-lepoll-self*: $b: B ==> A \lesssim A * B$

<proof>

lemma *cmult-le-self*:

$[[\text{Card}(K); \text{Ord}(L); 0 < L]] ==> K \text{ le } (K |*| L)$

<proof>

33.3.2 Monotonicity of multiplication

lemma *prod-lepoll-mono*:

$[[A \lesssim C; B \lesssim D]] ==> A * B \lesssim C * D$

<proof>

lemma *cmult-le-mono*:

$[[K' \text{ le } K; L' \text{ le } L]] ==> (K' |*| L') \text{ le } (K |*| L)$

<proof>

33.4 Multiplication of finite cardinals is "ordinary" multiplication

lemma *prod-succ-epoll*: $\text{succ}(A) * B \approx B + A * B$

<proof>

lemma *cmult-succ-lemma*:

$[[\text{Ord}(m); \text{Ord}(n)]] ==> \text{succ}(m) |*| n = n |+| (m |*| n)$

<proof>

lemma *nat-cmult-eq-mult*: $[| m: \text{nat}; n: \text{nat} |] \implies m \mid * \mid n = m \# * n$
 ⟨proof⟩

lemma *cmult-2*: $\text{Card}(n) \implies 2 \mid * \mid n = n \mid + \mid n$
 ⟨proof⟩

lemma *sum-lepoll-prod*: $2 \lesssim C \implies B+B \lesssim C*B$
 ⟨proof⟩

lemma *lepoll-imp-sum-lepoll-prod*: $[| A \lesssim B; 2 \lesssim A |] \implies A+B \lesssim A*B$
 ⟨proof⟩

33.5 Infinite Cardinals are Limit Ordinals

lemma *nat-cons-lepoll*: $\text{nat} \lesssim A \implies \text{cons}(u,A) \lesssim A$
 ⟨proof⟩

lemma *nat-cons-epoll*: $\text{nat} \lesssim A \implies \text{cons}(u,A) \approx A$
 ⟨proof⟩

lemma *nat-succ-epoll*: $\text{nat} \leq A \implies \text{succ}(A) \approx A$
 ⟨proof⟩

lemma *InfCard-nat*: $\text{InfCard}(\text{nat})$
 ⟨proof⟩

lemma *InfCard-is-Card*: $\text{InfCard}(K) \implies \text{Card}(K)$
 ⟨proof⟩

lemma *InfCard-Un*:
 $[| \text{InfCard}(K); \text{Card}(L) |] \implies \text{InfCard}(K \text{ Un } L)$
 ⟨proof⟩

lemma *InfCard-is-Limit*: $\text{InfCard}(K) \implies \text{Limit}(K)$
 ⟨proof⟩

lemma *ordermap-epoll-pred*:
 $[| \text{well-ord}(A,r); x:A |] \implies \text{ordermap}(A,r) \text{ ` } x \approx \text{Order.pred}(A,x,r)$
 ⟨proof⟩

33.5.1 Establishing the well-ordering

lemma *csquare-lam-inj*:
 $\text{Ord}(K) \implies (\text{lam } \langle x,y \rangle : K*K. \langle x \text{ Un } y, x, y \rangle) : \text{inj}(K*K, K*K*K)$

<proof>

lemma *well-ord-csquare*: $\text{Ord}(K) \implies \text{well-ord}(K * K, \text{csquare-rel}(K))$
<proof>

33.5.2 Characterising initial segments of the well-ordering

lemma *csquareD*:

$\llbracket \langle x, y \rangle, \langle z, z \rangle \rrbracket : \text{csquare-rel}(K); x < K; y < K; z < K \rrbracket \implies x \text{ le } z \ \& \ y \text{ le } z$
<proof>

lemma *pred-csquare-subset*:

$z < K \implies \text{Order.pred}(K * K, \langle z, z \rangle, \text{csquare-rel}(K)) \leq \text{succ}(z) * \text{succ}(z)$
<proof>

lemma *csquare-ltI*:

$\llbracket x < z; y < z; z < K \rrbracket \implies \langle x, y \rangle, \langle z, z \rangle : \text{csquare-rel}(K)$
<proof>

lemma *csquare-or-eqI*:

$\llbracket x \text{ le } z; y \text{ le } z; z < K \rrbracket \implies \langle x, y \rangle, \langle z, z \rangle : \text{csquare-rel}(K) \mid x = z \ \& \ y = z$
<proof>

33.5.3 The cardinality of initial segments

lemma *ordermap-z-lt*:

$\llbracket \text{Limit}(K); x < K; y < K; z = \text{succ}(x \text{ Un } y) \rrbracket \implies$
 $\text{ordermap}(K * K, \text{csquare-rel}(K)) \text{ ' } \langle x, y \rangle <$
 $\text{ordermap}(K * K, \text{csquare-rel}(K)) \text{ ' } \langle z, z \rangle$
<proof>

lemma *ordermap-csquare-le*:

$\llbracket \text{Limit}(K); x < K; y < K; z = \text{succ}(x \text{ Un } y) \rrbracket$
 $\implies \mid \text{ordermap}(K * K, \text{csquare-rel}(K)) \text{ ' } \langle x, y \rangle \mid \text{le} \mid \text{succ}(z) \mid * \mid \text{succ}(z) \mid$
<proof>

lemma *ordertype-csquare-le*:

$\llbracket \text{InfCard}(K); \text{ALL } y : K. \text{InfCard}(y) \dashrightarrow y \mid * \mid y = y \rrbracket$
 $\implies \text{ordertype}(K * K, \text{csquare-rel}(K)) \text{ le } K$
<proof>

lemma *InfCard-csquare-eq*: $\text{InfCard}(K) \implies K \mid * \mid K = K$

<proof>

lemma *well-ord-InfCard-square-eq*:
 $\llbracket \text{well-ord}(A,r); \text{InfCard}(|A|) \rrbracket \implies A * A \approx A$
<proof>

lemma *InfCard-square-eqpoll*: $\text{InfCard}(K) \implies K \times K \approx K$
<proof>

lemma *Inf-Card-is-InfCard*: $\llbracket \sim \text{Finite}(i); \text{Card}(i) \rrbracket \implies \text{InfCard}(i)$
<proof>

33.5.4 Toward's Kunen's Corollary 10.13 (1)

lemma *InfCard-le-cmult-eq*: $\llbracket \text{InfCard}(K); L \text{ le } K; 0 < L \rrbracket \implies K |*| L = K$
<proof>

lemma *InfCard-cmult-eq*: $\llbracket \text{InfCard}(K); \text{InfCard}(L) \rrbracket \implies K |*| L = K \text{ Un } L$
<proof>

lemma *InfCard-cdouble-eq*: $\text{InfCard}(K) \implies K |+| K = K$
<proof>

lemma *InfCard-le-cadd-eq*: $\llbracket \text{InfCard}(K); L \text{ le } K \rrbracket \implies K |+| L = K$
<proof>

lemma *InfCard-cadd-eq*: $\llbracket \text{InfCard}(K); \text{InfCard}(L) \rrbracket \implies K |+| L = K \text{ Un } L$
<proof>

33.6 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*: $\text{Ord}(\text{jump-cardinal}(K))$
<proof>

lemma *jump-cardinal-iff*:
 $i : \text{jump-cardinal}(K) \iff$
 $(\exists X \ r \ X. \ r \leq K * K \ \& \ X \leq K \ \& \ \text{well-ord}(X,r) \ \& \ i = \text{ordertype}(X,r))$
<proof>

lemma *K-lt-jump-cardinal*: $\text{Ord}(K) \implies K < \text{jump-cardinal}(K)$
<proof>

lemma *Card-jump-cardinal-lemma*:
 $\llbracket \text{well-ord}(X,r); \ r \leq K * K; \ X \leq K; \rrbracket$

$f : \text{bij}(\text{ordertype}(X,r), \text{jump-cardinal}(K)) \parallel$
 $\implies \text{jump-cardinal}(K) : \text{jump-cardinal}(K)$
 ⟨proof⟩

lemma *Card-jump-cardinal*: $\text{Card}(\text{jump-cardinal}(K))$
 ⟨proof⟩

33.7 Basic Properties of Successor Cardinals

lemma *csucc-basic*: $\text{Ord}(K) \implies \text{Card}(\text{csucc}(K)) \ \& \ K < \text{csucc}(K)$
 ⟨proof⟩

lemmas *Card-csucc = csucc-basic* [THEN conjunct1, standard]

lemmas *lt-csucc = csucc-basic* [THEN conjunct2, standard]

lemma *Ord-0-lt-csucc*: $\text{Ord}(K) \implies 0 < \text{csucc}(K)$
 ⟨proof⟩

lemma *csucc-le*: $\parallel \text{Card}(L); K < L \parallel \implies \text{csucc}(K) \text{ le } L$
 ⟨proof⟩

lemma *lt-csucc-iff*: $\parallel \text{Ord}(i); \text{Card}(K) \parallel \implies i < \text{csucc}(K) \iff |i| \text{ le } K$
 ⟨proof⟩

lemma *Card-lt-csucc-iff*:
 $\parallel \text{Card}(K'); \text{Card}(K) \parallel \implies K' < \text{csucc}(K) \iff K' \text{ le } K$
 ⟨proof⟩

lemma *InfCard-csucc*: $\text{InfCard}(K) \implies \text{InfCard}(\text{csucc}(K))$
 ⟨proof⟩

33.7.1 Removing elements from a finite set decreases its cardinality

lemma *Fin-imp-not-cons-lepoll*: $A : \text{Fin}(U) \implies x \sim : A \dashrightarrow \sim \text{cons}(x,A) \lesssim A$
 ⟨proof⟩

lemma *Finite-imp-cardinal-cons* [simp]:
 $\parallel \text{Finite}(A); a \sim : A \parallel \implies |\text{cons}(a,A)| = \text{succ}(|A|)$
 ⟨proof⟩

lemma *Finite-imp-succ-cardinal-Diff*:
 $\parallel \text{Finite}(A); a : A \parallel \implies \text{succ}(|A - \{a\}|) = |A|$
 ⟨proof⟩

lemma *Finite-imp-cardinal-Diff*: $\parallel \text{Finite}(A); a : A \parallel \implies |A - \{a\}| < |A|$
 ⟨proof⟩

lemma *Finite-cardinal-in-nat* [*simp*]: $Finite(A) ==> |A| : nat$
 ⟨*proof*⟩

lemma *card-Un-Int*:
 $[|Finite(A); Finite(B)|] ==> |A| \# + |B| = |A \ Un \ B| \# + |A \ Int \ B|$
 ⟨*proof*⟩

lemma *card-Un-disjoint*:
 $[|Finite(A); Finite(B); A \ Int \ B = 0|] ==> |A \ Un \ B| = |A| \# + |B|$
 ⟨*proof*⟩

lemma *card-partition* [*rule-format*]:
 $Finite(C) ==>$
 $Finite(\bigcup C) --->$
 $(\forall c \in C. |c| = k) --->$
 $(\forall c1 \in C. \forall c2 \in C. c1 \neq c2 ---> c1 \cap c2 = 0) --->$
 $k \# * |C| = |\bigcup C|$
 ⟨*proof*⟩

33.7.2 Theorems by Krzysztof Grabczewski, proofs by lcp

lemmas *nat-implies-well-ord = nat-into-Ord* [*THEN well-ord-Memrel, standard*]

lemma *nat-sum-egpoll-sum*: $[| m:nat; n:nat |] ==> m + n \approx m \# + n$
 ⟨*proof*⟩

lemma *Ord-subset-natD* [*rule-format*]: $Ord(i) ==> i <= nat ---> i : nat \ | \ i=nat$
 ⟨*proof*⟩

lemma *Ord-nat-subset-into-Card*: $[| Ord(i); i <= nat |] ==> Card(i)$
 ⟨*proof*⟩

lemma *Finite-Diff-sing-eq-diff-1*: $[| Finite(A); x:A |] ==> |A - \{x\}| = |A| \# - 1$
 ⟨*proof*⟩

lemma *cardinal-lt-imp-Diff-not-0* [*rule-format*]:
 $Finite(B) ==> ALL A. |B| < |A| ---> A - B \sim = 0$
 ⟨*proof*⟩

⟨*ML*⟩

end

34 Main-ZF: Theory Main: Everything Except AC

theory *Main-ZF* imports *List-ZF IntDiv-ZF CardinalArith* begin

34.1 Iteration of the function F

consts *iterates* :: $[i=>i,i,i] => i \quad ((\hat{-} \text{'(-)}) [60,1000,1000] 60)$

primrec

$F^{\wedge}0 (x) = x$
 $F^{\wedge}(\text{succ}(n)) (x) = F(F^{\wedge}n (x))$

definition

iterates-omega :: $[i=>i,i] => i$ **where**
iterates-omega(F,x) == $\bigcup n \in \text{nat}. F^{\wedge}n (x)$

notation (*xsymbols*)

iterates-omega $((\hat{-} \omega \text{'(-)}) [60,1000] 60)$

notation (*HTML output*)

iterates-omega $((\hat{-} \omega \text{'(-)}) [60,1000] 60)$

lemma *iterates-triv*:

$[[n \in \text{nat}; F(x) = x]] ==> F^{\wedge}n (x) = x$
 $\langle \text{proof} \rangle$

lemma *iterates-type* [*TC*]:

$[[n:\text{nat}; a:A; !!x. x:A ==> F(x) : A]]$
 $==> F^{\wedge}n (a) : A$
 $\langle \text{proof} \rangle$

lemma *iterates-omega-triv*:

$F(x) = x ==> F^{\wedge}\omega (x) = x$
 $\langle \text{proof} \rangle$

lemma *Ord-iterates* [*simp*]:

$[[n \in \text{nat}; !!i. \text{Ord}(i) ==> \text{Ord}(F(i)); \text{Ord}(x)]]$
 $==> \text{Ord}(F^{\wedge}n (x))$
 $\langle \text{proof} \rangle$

lemma *iterates-commute*: $n \in \text{nat} ==> F(F^{\wedge}n (x)) = F^{\wedge}n (F(x))$

$\langle \text{proof} \rangle$

34.2 Transfinite Recursion

Transfinite recursion for definitions based on the three cases of ordinals

definition

transrec3 :: $[i, i, [i,i]=>i, [i,i]=>i] => i$ **where**
transrec3(k, a, b, c) ==
transrec($k, \lambda x r.$
if $x=0$ *then* a
else if *Limit*(x) *then* $c(x, \lambda y \in x. r'y$)
else $b(\text{Arith.pred}(x), r \text{' Arith.pred}(x))$)

lemma *transrec3-0* [*simp*]: $\text{transrec3}(0, a, b, c) = a$
 ⟨*proof*⟩

lemma *transrec3-succ* [*simp*]:
 $\text{transrec3}(\text{succ}(i), a, b, c) = b(i, \text{transrec3}(i, a, b, c))$
 ⟨*proof*⟩

lemma *transrec3-Limit*:
 $\text{Limit}(i) \implies$
 $\text{transrec3}(i, a, b, c) = c(i, \lambda j \in i. \text{transrec3}(j, a, b, c))$
 ⟨*proof*⟩

⟨*ML*⟩

end

theory *Main*
imports *Main-ZF*
begin

end

35 AC: The Axiom of Choice

theory *AC* **imports** *Main-ZF* **begin**

This definition comes from Halmos (1960), page 59.

axiomatization **where**

$AC: [\![a: A; \!\!]x. x:A \implies (EX y. y:B(x)) \!\!] \implies EX z. z : Pi(A, B)$

lemma *AC-Pi*: $[\![\!\!]x. x \in A \implies (\exists y. y \in B(x)) \!\!] \implies \exists z. z \in Pi(A, B)$
 ⟨*proof*⟩

lemma *AC-ball-Pi*: $\forall x \in A. \exists y. y \in B(x) \implies \exists y. y \in Pi(A, B)$
 ⟨*proof*⟩

lemma *AC-Pi-Pow*: $\exists f. f \in (\Pi X \in Pow(C) - \{0\}. X)$
 ⟨*proof*⟩

lemma *AC-func*:

$[\![\!\!]x. x \in A \implies (\exists y. y \in x) \!\!] \implies \exists f \in A \rightarrow Union(A). \forall x \in A. f'x \in x$
 ⟨*proof*⟩

lemma *non-empty-family*: $[\![0 \notin A; x \in A \!\!] \implies \exists y. y \in x$
 ⟨*proof*⟩

lemma *AC-func0*: $0 \notin A \implies \exists f \in A \rightarrow \text{Union}(A). \forall x \in A. f'x \in x$
 ⟨*proof*⟩

lemma *AC-func-Pow*: $\exists f \in (\text{Pow}(C) - \{0\}) \rightarrow C. \forall x \in \text{Pow}(C) - \{0\}. f'x \in x$
 ⟨*proof*⟩

lemma *AC-Pi0*: $0 \notin A \implies \exists f. f \in (\prod x \in A. x)$
 ⟨*proof*⟩

end

36 Zorn: Zorn's Lemma

theory *Zorn* imports *OrderArith AC Inductive-ZF* **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**
Subset-rel(A) == $\{z \in A * A . \exists x y. z = \langle x, y \rangle \ \& \ x \leq y \ \& \ x \neq y\}$

definition

chain :: $i \Rightarrow i$ **where**
chain(A) == $\{F \in \text{Pow}(A). \forall X \in F. \forall Y \in F. X \leq Y \mid Y \leq X\}$

definition

super :: $[i, i] \Rightarrow i$ **where**
super(A, c) == $\{d \in \text{chain}(A). c \leq d \ \& \ c \neq d\}$

definition

maxchain :: $i \Rightarrow i$ **where**
maxchain(A) == $\{c \in \text{chain}(A). \text{super}(A, c) = 0\}$

definition

increasing :: $i \Rightarrow i$ **where**
increasing(A) == $\{f \in \text{Pow}(A) \rightarrow \text{Pow}(A). \forall x. x \leq A \ \dashrightarrow \ x \leq f'x\}$

Lemma for the inductive definition below

lemma *Union-in-Pow*: $Y \in \text{Pow}(\text{Pow}(A)) \implies \text{Union}(Y) \in \text{Pow}(A)$
 ⟨*proof*⟩

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 $TFin(S,next) \leq Pow(S)$

intros

$nextI: \quad [[x \in TFin(S,next); next \in increasing(S)]]$
 $\Rightarrow next'x \in TFin(S,next)$

$Pow-UnionI: Y \in Pow(TFin(S,next)) \Rightarrow Union(Y) \in TFin(S,next)$

monos $Pow-mono$

con-defs $increasing-def$

type-intros $CollectD1 [THEN apply-funtype] Union-in-Pow$

36.1 Mathematical Preamble

lemma $Union-lemma0: (\forall x \in C. x \leq A \mid B \leq x) \Rightarrow Union(C) \leq A \mid B \leq Union(C)$
 $\langle proof \rangle$

lemma $Inter-lemma0:$

$[[c \in C; \forall x \in C. A \leq x \mid x \leq B]] \Rightarrow A \leq Inter(C) \mid Inter(C) \leq B$
 $\langle proof \rangle$

36.2 The Transfinite Construction

lemma $increasingD1: f \in increasing(A) \Rightarrow f \in Pow(A) \rightarrow Pow(A)$
 $\langle proof \rangle$

lemma $increasingD2: [[f \in increasing(A); x \leq A]] \Rightarrow x \leq f'x$
 $\langle proof \rangle$

lemmas $TFin-UnionI = PowI [THEN TFin.Pow-UnionI, standard]$

lemmas $TFin-is-subset = TFin.dom-subset [THEN subsetD, THEN PowD, standard]$

Structural induction on $TFin(S, next)$

lemma $TFin-induct:$

$[[n \in TFin(S,next);$
 $\quad !!x. [[x \in TFin(S,next); P(x); next \in increasing(S)]] \Rightarrow P(next'x);$
 $\quad !!Y. [[Y \leq TFin(S,next); \forall y \in Y. P(y)]] \Rightarrow P(Union(Y))$
 $]] \Rightarrow P(n)$
 $\langle proof \rangle$

36.3 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*:

$$\begin{aligned} & [[n \in TFin(S,next); m \in TFin(S,next); \\ & \quad \forall x \in TFin(S,next) . x \leq m \leftrightarrow x = m \mid next'x \leq m]] \\ & \implies n \leq m \mid next'm \leq n \end{aligned}$$

<proof>

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*:

$$\begin{aligned} & [[m \in TFin(S,next); next \in increasing(S)]] \\ & \implies \forall n \in TFin(S,next) . n \leq m \leftrightarrow n = m \mid next'n \leq m \end{aligned}$$

<proof>

a more convenient form for Lemma 2

lemma *TFin-subsetD*:

$$\begin{aligned} & [[n \leq m; m \in TFin(S,next); n \in TFin(S,next); next \in increasing(S)]] \\ & \implies n = m \mid next'n \leq m \end{aligned}$$

<proof>

Consequences from section 3.3 – Property 3.2, the ordering is total

lemma *TFin-subset-linear*:

$$\begin{aligned} & [[m \in TFin(S,next); n \in TFin(S,next); next \in increasing(S)]] \\ & \implies n \leq m \mid m \leq n \end{aligned}$$

<proof>

Lemma 3 of section 3.3

lemma *equal-next-upper*:

$$[[n \in TFin(S,next); m \in TFin(S,next); m = next'm]] \implies n \leq m$$

<proof>

Property 3.3 of section 3.3

lemma *equal-next-Union*:

$$\begin{aligned} & [[m \in TFin(S,next); next \in increasing(S)]] \\ & \implies m = next'm \leftrightarrow m = Union(TFin(S,next)) \end{aligned}$$

<proof>

36.4 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*: $chain(A) \leq Pow(A)$

<proof>

lemma *super-subset-chain*: $super(A,c) \leq chain(A)$

<proof>

lemma *maxchain-subset-chain*: $\text{maxchain}(A) \leq \text{chain}(A)$
 ⟨proof⟩

lemma *choice-super*:
 [| $ch \in (\Pi X \in \text{Pow}(\text{chain}(S)) - \{0\}). X$; $X \in \text{chain}(S)$; $X \notin \text{maxchain}(S)$ |]
 ==> $ch \text{ ' } \text{super}(S, X) \in \text{super}(S, X)$
 ⟨proof⟩

lemma *choice-not-equals*:
 [| $ch \in (\Pi X \in \text{Pow}(\text{chain}(S)) - \{0\}). X$; $X \in \text{chain}(S)$; $X \notin \text{maxchain}(S)$ |]
 ==> $ch \text{ ' } \text{super}(S, X) \neq X$
 ⟨proof⟩

This justifies Definition 4.4

lemma *Hausdorff-next-exists*:
 $ch \in (\Pi X \in \text{Pow}(\text{chain}(S)) - \{0\}). X ==>$
 $\exists \text{next} \in \text{increasing}(S). \forall X \in \text{Pow}(S).$
 $\text{next}'X = \text{if}(X \in \text{chain}(S) - \text{maxchain}(S), \text{ch}'\text{super}(S, X), X)$
 ⟨proof⟩

Lemma 4

lemma *TFin-chain-lemma4*:
 [| $c \in \text{TFin}(S, \text{next})$;
 $ch \in (\Pi X \in \text{Pow}(\text{chain}(S)) - \{0\}). X$;
 $\text{next} \in \text{increasing}(S)$;
 $\forall X \in \text{Pow}(S). \text{next}'X =$
 $\text{if}(X \in \text{chain}(S) - \text{maxchain}(S), \text{ch}'\text{super}(S, X), X)$ |]
 ==> $c \in \text{chain}(S)$
 ⟨proof⟩

theorem *Hausdorff*: $\exists c. c \in \text{maxchain}(S)$
 ⟨proof⟩

36.5 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*:
 [| $c \in \text{chain}(A)$; $z \in A$; $\forall x \in c. x \leq z$ |] ==> $\text{cons}(z, c) \in \text{chain}(A)$
 ⟨proof⟩

lemma *Zorn*: $\forall c \in \text{chain}(S). \text{Union}(c) \in S ==> \exists y \in S. \forall z \in S. y \leq z \text{ --> } y = z$
 ⟨proof⟩

Alternative version of Zorn's Lemma

theorem *Zorn2*:

$\forall c \in \text{chain}(S). \exists y \in S. \forall x \in c. x \leq y \implies \exists y \in S. \forall z \in S. y \leq z \implies y=z$
 $\langle \text{proof} \rangle$

36.6 Zermelo's Theorem: Every Set can be Well-Ordered

Lemma 5

lemma *TFin-well-lemma5*:

$\llbracket n \in \text{TFin}(S, \text{next}); Z \leq \text{TFin}(S, \text{next}); z:Z; \sim \text{Inter}(Z) \in Z \rrbracket$
 $\implies \forall m \in Z. n \leq m$
 $\langle \text{proof} \rangle$

Well-ordering of $\text{TFin}(S, \text{next})$

lemma *well-ord-TFin-lemma*: $\llbracket Z \leq \text{TFin}(S, \text{next}); z \in Z \rrbracket \implies \text{Inter}(Z) \in Z$
 $\langle \text{proof} \rangle$

This theorem just packages the previous result

lemma *well-ord-TFin*:

$\text{next} \in \text{increasing}(S)$
 $\implies \text{well-ord}(\text{TFin}(S, \text{next}), \text{Subset-rel}(\text{TFin}(S, \text{next})))$
 $\langle \text{proof} \rangle$

* Defining the "next" operation for Zermelo's Theorem *

lemma *choice-Diff*:

$\llbracket \text{ch} \in (\prod X \in \text{Pow}(S) - \{0\}. X); X \subseteq S; X \neq S \rrbracket \implies \text{ch}'(S-X) \in S-X$
 $\langle \text{proof} \rangle$

This justifies Definition 6.1

lemma *Zermelo-next-exists*:

$\text{ch} \in (\prod X \in \text{Pow}(S) - \{0\}. X) \implies$
 $\exists \text{next} \in \text{increasing}(S). \forall X \in \text{Pow}(S).$
 $\text{next}'X = (\text{if } X=S \text{ then } S \text{ else } \text{cons}(\text{ch}'(S-X), X))$
 $\langle \text{proof} \rangle$

The construction of the injection

lemma *choice-imp-injection*:

$\llbracket \text{ch} \in (\prod X \in \text{Pow}(S) - \{0\}. X);$
 $\text{next} \in \text{increasing}(S);$
 $\forall X \in \text{Pow}(S). \text{next}'X = \text{if}(X=S, S, \text{cons}(\text{ch}'(S-X), X)) \rrbracket$
 $\implies (\lambda x \in S. \text{Union}(\{y \in \text{TFin}(S, \text{next}). x \notin y\}))$
 $\in \text{inj}(S, \text{TFin}(S, \text{next}) - \{S\})$
 $\langle \text{proof} \rangle$

The wellordering theorem

theorem *AC-well-ord*: $\exists r. \text{well-ord}(S, r)$

$\langle \text{proof} \rangle$

36.7 Zorn's Lemma for Partial Orders

Reimported from HOL by Clemens Ballarin.

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

$Chain(r) = \{A : Pow(field(r)). ALL a:A. ALL b:A. \langle a, b \rangle : r \mid \langle b, a \rangle : r\}$

lemma *mono-Chain*:

$r \subseteq s \Rightarrow Chain(r) \subseteq Chain(s)$

<proof>

theorem *Zorn-po*:

assumes *po*: *Partial-order*(r)

and u : $ALL C:Chain(r). EX u:field(r). ALL a:C. \langle a, u \rangle : r$

shows $EX m:field(r). ALL a:field(r). \langle m, a \rangle : r \longrightarrow a = m$

<proof>

end

37 Cardinal-AC: Cardinal Arithmetic Using AC

theory *Cardinal-AC* **imports** *CardinalArith Zorn* **begin**

37.1 Strengthened Forms of Existing Theorems on Cardinals

lemma *cardinal-epoll*: $|A| \text{ epoll } A$

<proof>

The theorem $||A|| = |A|$

lemmas *cardinal-idem* = *cardinal-epoll* [*THEN* *cardinal-cong*, *standard*, *simp*]

lemma *cardinal-eqE*: $|X| = |Y| \Rightarrow X \text{ epoll } Y$

<proof>

lemma *cardinal-epoll-iff*: $|X| = |Y| \Leftrightarrow X \text{ epoll } Y$

<proof>

lemma *cardinal-disjoint-Un*:

$[| |A|=|B|; |C|=|D|; A \text{ Int } C = 0; B \text{ Int } D = 0 \]$

$\Rightarrow |A \text{ Un } C| = |B \text{ Un } D|$

<proof>

lemma *lepoll-imp-Card-le*: $A \text{ lepoll } B \Rightarrow |A| \text{ le } |B|$

<proof>

lemma *cadd-assoc*: $(i \text{ |+| } j) \text{ |+| } k = i \text{ |+| } (j \text{ |+| } k)$

<proof>

lemma *cmult-assoc*: $(i \mid * \mid j) \mid * \mid k = i \mid * \mid (j \mid * \mid k)$
 ⟨proof⟩

lemma *cadd-cmult-distrib*: $(i \mid + \mid j) \mid * \mid k = (i \mid * \mid k) \mid + \mid (j \mid * \mid k)$
 ⟨proof⟩

lemma *InfCard-square-eq*: $\text{InfCard}(|A|) \implies A * A \text{ eqpoll } A$
 ⟨proof⟩

37.2 The relationship between cardinality and le-pollence

lemma *Card-le-imp-lepoll*: $|A| \text{ le } |B| \implies A \text{ lepoll } B$
 ⟨proof⟩

lemma *le-Card-iff*: $\text{Card}(K) \implies |A| \text{ le } K \iff A \text{ lepoll } K$
 ⟨proof⟩

lemma *cardinal-0-iff-0* [simp]: $|A| = 0 \iff A = 0$
 ⟨proof⟩

lemma *cardinal-lt-iff-lesspoll*: $\text{Ord}(i) \implies i < |A| \iff i \text{ lesspoll } A$
 ⟨proof⟩

lemma *cardinal-le-imp-lepoll*: $i \leq |A| \implies i \lesssim A$
 ⟨proof⟩

37.3 Other Applications of AC

lemma *surj-implies-inj*: $f: \text{surj}(X, Y) \implies \exists X \text{ g. } g: \text{inj}(Y, X)$
 ⟨proof⟩

lemma *surj-implies-cardinal-le*: $f: \text{surj}(X, Y) \implies |Y| \text{ le } |X|$
 ⟨proof⟩

lemma *cardinal-UN-le*:
 $[| \text{InfCard}(K); \text{ ALL } i:K. |X(i)| \text{ le } K |] \implies |\bigcup i \in K. X(i)| \text{ le } K$
 ⟨proof⟩

lemma *cardinal-UN-lt-csucc*:
 $[| \text{InfCard}(K); \text{ ALL } i:K. |X(i)| < \text{csucc}(K) |]$
 $\implies |\bigcup i \in K. X(i)| < \text{csucc}(K)$
 ⟨proof⟩

lemma *cardinal-UN-Ord-lt-csucc*:
 $[| \text{InfCard}(K); \text{ ALL } i:K. j(i) < \text{csucc}(K) |]$

$\implies (\bigcup i \in K. j(i)) < csucc(K)$
 <proof>

lemma *inj-UN-subset*:

$[[f: inj(A,B); a:A]] \implies$
 $(\bigcup x \in A. C(x)) \leq (\bigcup y \in B. C(\text{if } y: \text{range}(f) \text{ then } \text{converse}(f)'y \text{ else } a))$
 <proof>

lemma *le-UN-Ord-lt-csucc*:

$[[InfCard(K); |W| le K; ALL w:W. j(w) < csucc(K)]]$
 $\implies (\bigcup w \in W. j(w)) < csucc(K)$
 <proof>

<ML>

end

38 InfDatatype: Infinite-Branching Datatype Definitions

theory *InfDatatype* **imports** *Datatype-ZF Univ Finite Cardinal-AC* **begin**

lemmas *fun-Limit-VfromE* =

Limit-VfromE [OF apply-funtype InfCard-csucc [THEN InfCard-is-Limit]]

lemma *fun-Vcsucc-lemma*:

$[[f: D \rightarrow Vfrom(A, csucc(K)); |D| le K; InfCard(K)]]$
 $\implies EX j. f: D \rightarrow Vfrom(A, j) \ \& \ j < csucc(K)$
 <proof>

lemma *subset-Vcsucc*:

$[[D \leq Vfrom(A, csucc(K)); |D| le K; InfCard(K)]]$
 $\implies EX j. D \leq Vfrom(A, j) \ \& \ j < csucc(K)$
 <proof>

lemma *fun-Vcsucc*:

$[[|D| le K; InfCard(K); D \leq Vfrom(A, csucc(K))]] \implies$
 $D \rightarrow Vfrom(A, csucc(K)) \leq Vfrom(A, csucc(K))$
 <proof>

lemma *fun-in-Vcsucc*:

$$\begin{aligned} & \llbracket f: D \rightarrow V\text{from}(A, \text{csucc}(K)); |D| \text{ le } K; \text{InfCard}(K); \\ & \quad D \leq V\text{from}(A, \text{csucc}(K)) \rrbracket \\ & \implies f: V\text{from}(A, \text{csucc}(K)) \end{aligned}$$
 <proof>

lemmas *fun-in-Vcsucc'* = *fun-in-Vcsucc* [OF - - - subsetI]

lemma *Card-fun-Vcsucc*:

$$\text{InfCard}(K) \implies K \rightarrow V\text{from}(A, \text{csucc}(K)) \leq V\text{from}(A, \text{csucc}(K))$$
 <proof>

lemma *Card-fun-in-Vcsucc*:

$$\llbracket f: K \rightarrow V\text{from}(A, \text{csucc}(K)); \text{InfCard}(K) \rrbracket \implies f: V\text{from}(A, \text{csucc}(K))$$
 <proof>

lemma *Limit-csucc*: *InfCard*(K) \implies *Limit*(*csucc*(K))

<proof>

lemmas *Pair-in-Vcsucc* = *Pair-in-VLimit* [OF - - *Limit-csucc*]

lemmas *Inl-in-Vcsucc* = *Inl-in-VLimit* [OF - *Limit-csucc*]

lemmas *Inr-in-Vcsucc* = *Inr-in-VLimit* [OF - *Limit-csucc*]

lemmas *zero-in-Vcsucc* = *Limit-csucc* [THEN *zero-in-VLimit*]

lemmas *nat-into-Vcsucc* = *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-Card-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-Vcsucc Inl-in-Vcsucc Inr-in-Vcsucc

zero-in-Vcsucc A-into-Vfrom nat-into-Vcsucc

Card-fun-in-Vcsucc fun-in-Vcsucc' UN-I

end

theory *Main-ZFC* **imports** *Main-ZF InfDatatype*
begin

end