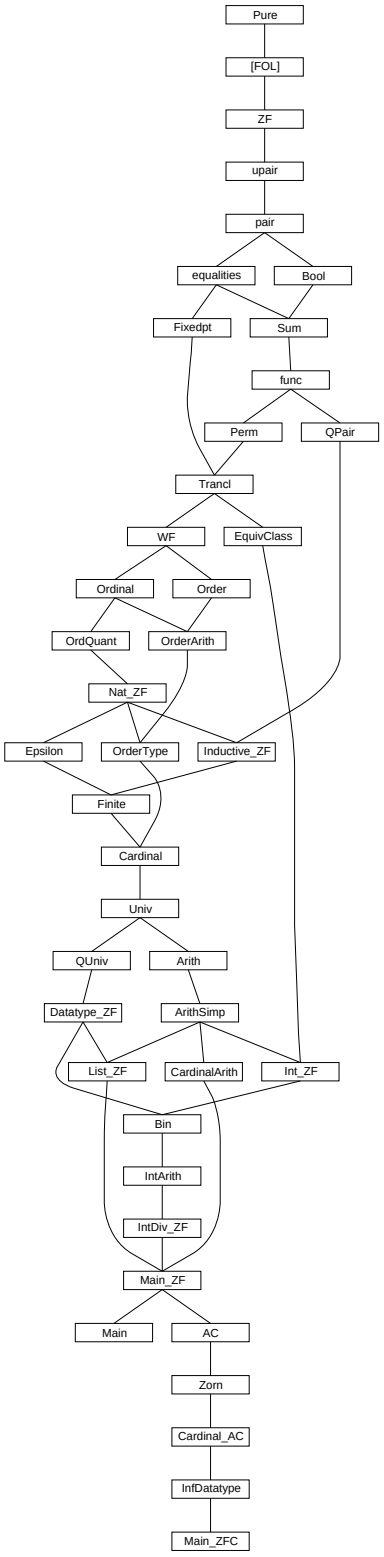
ZF

Lawrence C Paulson and others

April 19, 2009

Contents


Pure
[FOL]
ZF
upair
pair
equalities
Bool
Fixedpt
Sum
func
Perm
QPair
Trancl
WF
EquivClass
Ordinal
Order
OrdQuant
OrderArith
Nat_ZF
Epsilon
OrderType
Inductive_ZF
Finite
Cardinal
Univ
QUniv
Arith
Datatype_ZF
ArithSimp
List_ZF
CardinalArith
Int_ZF
Bin
IntArith
IntDiv_ZF
Main_ZF
Main
AC
Zorn
Cardinal_AC
InfDatatype
Main_ZFC

1 ZF: Zermelo-Fraenkel Set Theory

theory *ZF* **imports** *FOL* **begin**

⟨*ML*⟩

global

typedecl *i*
arities *i* :: *term*

consts

0 :: *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** $\sim$: *50*) — negated membership relation
where $x \sim: y == \sim (x : y)$

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

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

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

  <x, y, z>   == <x, <y, z>>
  <x, y>      == Pair(x, y)
  %<x,y,zs>.b == split(%x <y,zs>.b)
  %<x,y>.b    == 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 ($\bigcup$- [*90*] *90*) **and**
Inter ($\bigcap$- [*90*] *90*)

syntax (*xsymbols*)
@*Collect* :: [*pttrn*, *i*, *o*] => *i* (*1*{- $\in$ - ./ -})
@*Replace* :: [*pttrn*, *pttrn*, *i*, *o*] => *i* (*1*{- ./ - $\in$ -, -})
@*RepFun* :: [*i*, *pttrn*, *i*] => *i* (*1*{- ./ - $\in$ -}) [*51*,*0*,*51*])
@*UNION* :: [*pttrn*, *i*, *i*] => *i* (*3*$\bigcup$-$\in$-./ -) *10*)
@*INTER* :: [*pttrn*, *i*, *i*] => *i* (*3*$\bigcap$-$\in$-./ -) *10*)
@*PROD* :: [*pttrn*, *i*, *i*] => *i* (*3*Π-$\in$-./ -) *10*)
@*SUM* :: [*pttrn*, *i*, *i*] => *i* (*3*Σ-$\in$-./ -) *10*)
@*lam* :: [*pttrn*, *i*, *i*] => *i* (*3*λ-$\in$-./ -) *10*)
@*Ball* :: [*pttrn*, *i*, *o*] => *o* (*3*$\forall$-$\in$-./ -) *10*)
@*Bex* :: [*pttrn*, *i*, *o*] => *o* (*3*$\exists$-$\in$-./ -) *10*)
@*Tuple* :: [*i*, *is*] => *i* ($\langle$(-,/ -)$\rangle$)
@*pattern* :: *patterns* => *pttrn* ($\langle$-$\rangle$)

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 ($\bigcup$- [*90*] *90*) **and**
Inter ($\bigcap$- [*90*] *90*)

syntax (*HTML* **output**)
@*Collect* :: [*pttrn*, *i*, *o*] => *i* (*1*{- $\in$ - ./ -}))
@*Replace* :: [*pttrn*, *pttrn*, *i*, *o*] => *i* (*1*{- ./ - $\in$ -, -}))
@*RepFun* :: [*i*, *pttrn*, *i*] => *i* (*1*{- ./ - $\in$ -}) [*51*,*0*,*51*])
@*UNION* :: [*pttrn*, *i*, *i*] => *i* (*3*$\bigcup$-$\in$-./ -) *10*)
@*INTER* :: [*pttrn*, *i*, *i*] => *i* (*3*$\bigcap$-$\in$-./ -) *10*)
@*PROD* :: [*pttrn*, *i*, *i*] => *i* (*3*Π-$\in$-./ -) *10*)
@*SUM* :: [*pttrn*, *i*, *i*] => *i* (*3*Σ-$\in$-./ -) *10*)
@*lam* :: [*pttrn*, *i*, *i*] => *i* (*3*λ-$\in$-./ -) *10*)
@*Ball* :: [*pttrn*, *i*, *o*] => *o* (*3*$\forall$-$\in$-./ -) *10*)
@*Bex* :: [*pttrn*, *i*, *o*] => *o* (*3*$\exists$-$\in$-./ -) *10*)
@*Tuple* :: [*i*, *is*] => *i* ($\langle$(-,/ -)$\rangle$)
@*pattern* :: *patterns* => *pttrn* ($\langle$-$\rangle$)

finalconsts
0 Pow Inf Union PrimReplace mem

defs

Ball-def: $Ball(A,\ P) == \forall\, x.\ x{\in}A\ {-}{-}{>}\ P(x)$
Bex-def: $Bex(A,\ P) == \exists\, x.\ x{\in}A\ \&\ P(x)$

subset-def: $A <= B == \forall\, x{\in}A.\ x{\in}B$

local

axioms

extension: $A = B <{-}{>} A <= B\ \&\ B <= A$
Union-iff: $A \in Union(C) <{-}{>} (\exists\, B{\in}C.\ A{\in}B)$
Pow-iff: $A \in Pow(B) <{-}{>} A <= B$

infinity: $0{\in}Inf\ \&\ (\forall\, y{\in}Inf.\ succ(y){:}\ Inf)$

foundation: $A{=}0\ |\ (\exists\, x{\in}A.\ \forall\, y{\in}x.\ y^{\sim}{:}A)$

replacement: $(\forall\, x{\in}A.\ \forall\, y\ z.\ P(x,y)\ \&\ P(x,z)\ {-}{-}{>}\ y{=}z) ==>$
$b \in PrimReplace(A,P) <{-}{>} (\exists\, x{\in}A.\ P(x,b))$

defs

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

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

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

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

cons-def: $cons(a,A) == Upair(a,a)\ Un\ A$
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: $<a,b> == \{\{a,a\},\ \{a,b\}\}$
fst-def: $fst(p) == THE\ a.\ \exists\, b.\ p{=}<a,b>$
snd-def: $snd(p) == THE\ b.\ \exists\, a.\ p{=}<a,b>$
split-def: $split(c) == \%p.\ c(fst(p),\ snd(p))$
Sigma-def: $Sigma(A,B) == \bigcup x{\in}A.\ \bigcup y{\in}B(x).\ \{<x,y>\}$

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

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

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

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

1.1 Substitution

lemma *subst-elem*: $[|\ b{\in}A;\ \ a{=}b\ |] ==> a{\in}A$
⟨*proof*⟩

1.2 Bounded universal quantifier

lemma *ballI* [*intro*!]: [| !!x. x∈A ==> $P(x)$ |] ==> ∀ x∈A. $P(x)$
⟨*proof*⟩

lemmas *strip* = *impI allI ballI*

lemma *bspec* [*dest?*]: [| ∀ x∈A. $P(x)$; x: A |] ==> $P(x)$
⟨*proof*⟩

lemma *rev-ballE* [*elim*]:
[| ∀ x∈A. $P(x)$; $x^{\sim}$:A ==> Q; $P(x)$ ==> Q |] ==> Q
⟨*proof*⟩

lemma *ballE*: [| ∀ x∈A. $P(x)$; $P(x)$ ==> Q; $x^{\sim}$:A ==> Q |] ==> Q
⟨*proof*⟩

lemma *rev-bspec*: [| x: A; ∀ x∈A. $P(x)$ |] ==> $P(x)$
⟨*proof*⟩

lemma *ball-triv* [*simp*]: (∀ x∈A. P) <-> ((∃ x. x∈A) --> P)
⟨*proof*⟩

lemma *ball-cong* [*cong*]:
[| A=A'; !!x. x∈A' ==> $P(x)$ <-> $P'(x)$ |] ==> (∀ x∈A. $P(x)$) <-> (∀ x∈A'. $P'(x)$)
⟨*proof*⟩

lemma *atomize-ball*:
(!!x. x ∈ A ==> $P(x)$) == *Trueprop* (∀ x∈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 |] ==> ∃ x∈A. $P(x)$
⟨*proof*⟩

lemma *rev-bexI*: [| x∈A; $P(x)$ |] ==> ∃ x∈A. $P(x)$
⟨*proof*⟩

lemma *bexCI*: [| ∀ x∈A. ~$P(x)$ ==> $P(a)$; a: A |] ==> ∃ x∈A. $P(x)$

⟨*proof*⟩

lemma *bexE* [*elim*!]: [| ∃ *x*∈*A*. *P*(*x*); !!*x*. [| *x*∈*A*; *P*(*x*) |] ==> *Q* |] ==> *Q*
⟨*proof*⟩

lemma *bex-triv* [*simp*]: (∃ *x*∈*A*. *P*) <-> ((∃ *x*. *x*∈*A*) & *P*)
⟨*proof*⟩

lemma *bex-cong* [*cong*]:
[| *A*=*A*'; !!*x*. *x*∈*A*' ==> *P*(*x*) <-> *P*'(*x*) |]
==> (∃ *x*∈*A*. *P*(*x*)) <-> (∃ *x*∈*A*'. *P*'(*x*))
⟨*proof*⟩

1.4 Rules for subsets

lemma *subsetI* [*intro*!]:
(!!*x*. *x*∈*A* ==> *x*∈*B*) ==> *A* <= *B*
⟨*proof*⟩

lemma *subsetD* [*elim*]: [| *A* <= *B*; *c*∈*A* |] ==> *c*∈*B*
⟨*proof*⟩

lemma *subsetCE* [*elim*]:
[| *A* <= *B*; *c*~:*A* ==> *P*; *c*∈*B* ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *rev-subsetD*: [| *c*∈*A*; *A*<=*B* |] ==> *c*∈*B*
⟨*proof*⟩

lemma *contra-subsetD*: [| *A* <= *B*; *c* ~: *B* |] ==> *c* ~: *A*
⟨*proof*⟩

lemma *rev-contra-subsetD*: [| *c* ~: *B*; *A* <= *B* |] ==> *c* ~: *A*
⟨*proof*⟩

lemma *subset-refl* [*simp*]: *A* <= *A*
⟨*proof*⟩

lemma *subset-trans*: [| *A*<=*B*; *B*<=*C* |] ==> *A*<=*C*
⟨*proof*⟩

lemma *subset-iff*:
A<=*B* <-> (∀ *x*. *x*∈*A* --> *x*∈*B*)
⟨*proof*⟩

1.5 Rules for equality

lemma *equalityI* [*intro*]: [| $A <= B$; $B <= A$ |] ==> $A = B$
⟨*proof*⟩

lemma *equality-iffI*: (!!x. x∈A <-> x∈B) ==> $A = B$
⟨*proof*⟩

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

lemma *equalityE*: [| $A = B$; [| A<=B; B<=A |] ==> P |] ==> P
⟨*proof*⟩

lemma *equalityCE*:
[| $A = B$; [| c∈A; c∈B |] ==> P; [| $c^{\sim}$:A; $c^{\sim}$:B |] ==> P |] ==> P
⟨*proof*⟩

lemma *equality-iffD*:
$A = B$ ==> (!!x. x : A <-> x : B)
⟨*proof*⟩

1.6 Rules for Replace – the derived form of replacement

lemma *Replace-iff*:
b : {y. x∈A, $P(x,y)$} <-> (∃x∈A. $P(x,b)$ & (∀y. $P(x,y)$ --> y=b))
⟨*proof*⟩

lemma *ReplaceI* [*intro*]:
[| $P(x,b)$; x: A; !!y. $P(x,y)$ ==> y=b |] ==>
b : {y. x∈A, $P(x,y)$}
⟨*proof*⟩

lemma *ReplaceE*:
[| b : {y. x∈A, $P(x,y)$};
!!x. [| x: A; $P(x,b)$; ∀y. $P(x,y)$-->y=b |] ==> R
|] ==> R
⟨*proof*⟩

lemma *ReplaceE2* [*elim*!]:
[| b : {y. x∈A, $P(x,y)$};
!!x. [| x: A; $P(x,b)$ |] ==> R
|] ==> R
⟨*proof*⟩

lemma *Replace-cong* [*cong*]:

[| A=B; !!x y. x∈B ==> $P(x,y)$ <-> $Q(x,y)$ |] ==>
$Replace(A,P) = Replace(B,Q)$
⟨*proof*⟩

1.7 Rules for RepFun

lemma *RepFunI*: $a \in A$ ==> $f(a) : \{f(x).\ x∈A\}$
⟨*proof*⟩

lemma *RepFun-eqI* [*intro*]: [| b=$f(a)$; $a \in A$ |] ==> $b : \{f(x).\ x∈A\}$
⟨*proof*⟩

lemma *RepFunE* [*elim*!]:
[| $b : \{f(x).\ x∈A\}$;
!!x.[| $x∈A$; b=$f(x)$ |] ==> P |] ==>
P
⟨*proof*⟩

lemma *RepFun-cong* [*cong*]:
[| A=B; !!x. x∈B ==> $f(x)$=$g(x)$ |] ==> $RepFun(A,f) = RepFun(B,g)$
⟨*proof*⟩

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

lemma *triv-RepFun* [*simp*]: $\{x.\ x∈A\} = A$
⟨*proof*⟩

1.8 Rules for Collect – forming a subset by separation

lemma *separation* [*simp*]: $a : \{x∈A.\ P(x)\}$ <-> $a∈A$ & $P(a)$
⟨*proof*⟩

lemma *CollectI* [*intro*!]: [| $a∈A$; $P(a)$ |] ==> $a : \{x∈A.\ P(x)\}$
⟨*proof*⟩

lemma *CollectE* [*elim*!]: [| $a : \{x∈A.\ P(x)\}$; [| $a∈A$; $P(a)$ |] ==> R |] ==> R
⟨*proof*⟩

lemma *CollectD1*: $a : \{x∈A.\ P(x)\}$ ==> $a∈A$
⟨*proof*⟩

lemma *CollectD2*: $a : \{x∈A.\ P(x)\}$ ==> $P(a)$
⟨*proof*⟩

lemma *Collect-cong* [*cong*]:
[| A=B; !!x. x∈B ==> $P(x)$ <-> $Q(x)$ |]
==> $Collect(A, \%x.\ P(x)) = Collect(B, \%x.\ Q(x))$
⟨*proof*⟩

1.9 Rules for Unions

declare *Union-iff* [*simp*]

lemma *UnionI* [*intro*]: [| *B*: *C*; *A*: *B* |] ==> *A*: *Union*(*C*)
⟨*proof*⟩

lemma *UnionE* [*elim*!]: [| *A* ∈ *Union*(*C*); !!*B*.[| *A*: *B*; *B*: *C* |] ==> *R* |] ==> *R*
⟨*proof*⟩

1.10 Rules for Unions of families

lemma *UN-iff* [*simp*]: *b* : (⋃*x*∈*A*. *B*(*x*)) <-> (∃*x*∈*A*. *b* ∈ *B*(*x*))
⟨*proof*⟩

lemma *UN-I*: [| *a*: *A*; *b*: *B*(*a*) |] ==> *b*: (⋃*x*∈*A*. *B*(*x*))
⟨*proof*⟩

lemma *UN-E* [*elim*!]:
[| *b* : (⋃*x*∈*A*. *B*(*x*)); !!*x*.[| *x*: *A*; *b*: *B*(*x*) |] ==> *R* |] ==> *R*
⟨*proof*⟩

lemma *UN-cong*:
[| *A*=*B*; !!*x*. *x*∈*B* ==> *C*(*x*)=*D*(*x*) |] ==> (⋃*x*∈*A*. *C*(*x*)) = (⋃*x*∈*B*. *D*(*x*))
⟨*proof*⟩

1.11 Rules for the empty set

lemma *not-mem-empty* [*simp*]: *a* ~: *0*
⟨*proof*⟩

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

lemma *empty-subsetI* [*simp*]: *0* <= *A*
⟨*proof*⟩

lemma *equals0I*: [| !!*y*. *y*∈*A* ==> *False* |] ==> *A*=*0*
⟨*proof*⟩

lemma *equals0D* [*dest*]: *A*=*0* ==> *a* ~: *A*
⟨*proof*⟩

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

lemma *not-emptyI*: *a*∈*A* ==> *A* ~= *0*

⟨*proof*⟩

lemma *not-emptyE*: [| $A \sim= 0$; !!x. x∈A ==> R |] ==> R
⟨*proof*⟩

1.12 Rules for Inter

lemma *Inter-iff*: $A \in Inter(C)$ <-> (∀x∈C. A: x) & C≠0
⟨*proof*⟩

lemma *InterI* [*intro*!]:
[| !!x. x: C ==> A: x; C≠0 |] ==> $A \in Inter(C)$
⟨*proof*⟩

lemma *InterD* [*elim*]: [| $A \in Inter(C)$; $B \in C$ |] ==> $A \in B$
⟨*proof*⟩

lemma *InterE* [*elim*]:
[| $A \in Inter(C)$; $B^\sim$:C ==> R; A∈B ==> R |] ==> R
⟨*proof*⟩

1.13 Rules for Intersections of families

lemma *INT-iff*: b : (⋂x∈A. $B(x)$) <-> (∀x∈A. $b \in B(x)$) & A≠0
⟨*proof*⟩

lemma *INT-I*: [| !!x. x: A ==> b: $B(x)$; A≠0 |] ==> b: (⋂x∈A. $B(x)$)
⟨*proof*⟩

lemma *INT-E*: [| b : (⋂x∈A. $B(x)$); a: A |] ==> $b \in B(a)$
⟨*proof*⟩

lemma *INT-cong*:
[| A=B; !!x. x∈B ==> $C(x)$=$D(x)$ |] ==> (⋂x∈A. $C(x)$) = (⋂x∈B. $D(x)$)
⟨*proof*⟩

1.14 Rules for Powersets

lemma *PowI*: A <= B ==> $A \in Pow(B)$
⟨*proof*⟩

lemma *PowD*: $A \in Pow(B)$ ==> A<=B
⟨*proof*⟩

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*]:
    (!!x. x:A ==> $P(x)$) == *Trueprop* (*ALL* x:A. $P(x)$)
⟨*proof*⟩

2.1 Unordered Pairs: constant *Upair*

lemma *Upair-iff* [*simp*]: c : *Upair*(a,b) <-> (c=a | c=b)
⟨*proof*⟩

lemma *UpairI1*: a : *Upair*(a,b)
⟨*proof*⟩

lemma *UpairI2*: b : *Upair*(a,b)
⟨*proof*⟩

lemma *UpairE*: [| a : *Upair*(b,c); a=b ==> P; a=c ==> P |] ==> P
⟨*proof*⟩

2.2 Rules for Binary Union, Defined via *Upair*

lemma *Un-iff* [*simp*]: c : A *Un* B <-> (c:A | c:B)
⟨*proof*⟩

lemma *UnI1*: c : A ==> c : A *Un* B
⟨*proof*⟩

lemma *UnI2*: c : B ==> c : A *Un* B

⟨*proof*⟩

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

lemma *UnE* [*elim*!]: [| $c : A$ *Un* B; $c{:}A \Longrightarrow P$; $c{:}B \Longrightarrow P$ |] $\Longrightarrow P$
⟨*proof*⟩

lemma *UnE′*: [| $c : A$ *Un* B; $c{:}A \Longrightarrow P$; [| $c{:}B$; $c^{\sim}{:}A$ |] $\Longrightarrow P$ |] $\Longrightarrow P$
⟨*proof*⟩

lemma *UnCI* [*intro*!]: (c ~: $B \Longrightarrow c : A$) $\Longrightarrow c : A$ *Un* B
⟨*proof*⟩

2.3 Rules for Binary Intersection, Defined via *Upair*

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

lemma *IntI* [*intro*!]: [| $c : A$; $c : B$ |] $\Longrightarrow c : A$ *Int* B
⟨*proof*⟩

lemma *IntD1*: $c : A$ *Int* $B \Longrightarrow c : A$
⟨*proof*⟩

lemma *IntD2*: $c : A$ *Int* $B \Longrightarrow c : B$
⟨*proof*⟩

lemma *IntE* [*elim*!]: [| $c : A$ *Int* B; [| $c{:}A$; $c{:}B$ |] $\Longrightarrow P$ |] $\Longrightarrow P$
⟨*proof*⟩

2.4 Rules for Set Difference, Defined via *Upair*

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

lemma *DiffI* [*intro*!]: [| $c : A$; c ~: B |] $\Longrightarrow c : A - B$
⟨*proof*⟩

lemma *DiffD1*: $c : A - B \Longrightarrow c : A$
⟨*proof*⟩

lemma *DiffD2*: $c : A - B \Longrightarrow c$ ~: B
⟨*proof*⟩

lemma *DiffE* [*elim*!]: [| $c : A - B$; [| $c{:}A$; $c^{\sim}{:}B$ |] $\Longrightarrow P$ |] $\Longrightarrow P$
⟨*proof*⟩

2.5 Rules for *cons*

lemma *cons-iff* [*simp*]: $a : cons(b,A) <-> (a=b \mid a:A)$
⟨*proof*⟩

lemma *consI1* [*simp*,*TC*]: $a : cons(a,B)$
⟨*proof*⟩

lemma *consI2*: $a : B ==> a : cons(b,B)$
⟨*proof*⟩

lemma *consE* [*elim*!]: $[\mid a : cons(b,A);\ \ a=b ==> P;\ \ a:A ==> P \mid] ==> P$
⟨*proof*⟩

lemma *consE'*:
$[\mid a : cons(b,A);\ \ a=b ==> P;\ \ [\mid a:A;\ \ a^{\sim}=b \mid] ==> P \mid] ==> P$
⟨*proof*⟩

lemma *consCI* [*intro*!]: $(a^{\sim}:B ==> a=b) ==> a: cons(b,B)$
⟨*proof*⟩

lemma *cons-not-0* [*simp*]: $cons(a,B) \ ^{\sim}= 0$
⟨*proof*⟩

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

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

2.6 Singletons

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

lemma *singletonI* [*intro*!]: $a : \{a\}$
⟨*proof*⟩

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

2.7 Descriptions

lemma *the-equality* [*intro*]:
$[\mid P(a);\ \ !!x.\ P(x) ==> x=a \mid] ==> (THE\ x.\ P(x)) = a$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *theI*: *EX*! *x*. *P*(*x*) ==> *P*(*THE x*. *P*(*x*))
⟨*proof*⟩

lemma *the-0*: ~ (*EX*! *x*. *P*(*x*)) ==> (*THE x*. *P*(*x*))=*0*
⟨*proof*⟩

lemma *theI2*:
assumes *p1*: ~ *Q*(*0*) ==> *EX*! *x*. *P*(*x*)
and *p2*: !!*x*. *P*(*x*) ==> *Q*(*x*)
shows *Q*(*THE x*. *P*(*x*))
⟨*proof*⟩

lemma *the-eq-trivial* [*simp*]: (*THE x*. *x* = *a*) = *a*
⟨*proof*⟩

lemma *the-eq-trivial2* [*simp*]: (*THE x*. *a* = *x*) = *a*
⟨*proof*⟩

2.8 Conditional Terms: *if*−*then*−*else*

lemma *if-true* [*simp*]: (*if True then a else b*) = *a*
⟨*proof*⟩

lemma *if-false* [*simp*]: (*if False then a else b*) = *b*
⟨*proof*⟩

lemma *if-cong*:
[| *P*<−>*Q*; *Q* ==> *a*=*c*; ~*Q* ==> *b*=*d* |]
==> (*if P then a else b*) = (*if Q then c else d*)
⟨*proof*⟩

lemma *if-weak-cong*: *P*<−>*Q* ==> (*if P then x else y*) = (*if Q then x else y*)
⟨*proof*⟩

lemma *if-P*: *P* ==> (*if P then a else b*) = *a*
⟨*proof*⟩

lemma *if-not-P*: ~*P* ==> (*if P then a else b*) = *b*
⟨*proof*⟩

lemma *split-if* [*split*]:
    $P(\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y) <-> ((Q \dashrightarrow P(x)) \ \&\ (\sim Q \dashrightarrow P(y)))$
⟨*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: $(\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y) <-> P \ \&\ a{:}x \mid \sim P \ \&\ a{:}y$
⟨*proof*⟩

lemma *if-type* [*TC*]:
    $[\!|\ P ==> a{:}\ A;\ \sim P ==> b{:}\ A\ |\!] ==> (\mathit{if}\ P\ \mathit{then}\ a\ \mathit{else}\ b){:}\ A$
⟨*proof*⟩

lemma *split-if-asm*: $P(\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y) <-> (\sim((Q \ \&\ \sim P(x)) \mid (\sim Q \ \&\ \sim P(y))))$
⟨*proof*⟩

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

2.9 Consequences of Foundation

lemma *mem-asym*: $[\!|\ a{:}b;\ \sim P ==> b{:}a\ |\!] ==> P$
⟨*proof*⟩

lemma *mem-irrefl*: $a{:}a ==> P$
⟨*proof*⟩

lemma *mem-not-refl*: $a \sim{:}\ a$
⟨*proof*⟩

lemma *mem-imp-not-eq*: $a{:}A ==> a \sim{=} A$
⟨*proof*⟩

lemma *eq-imp-not-mem*: $a{=}A ==> a \sim{:}\ A$

⟨*proof*⟩

2.10 Rules for Successor

lemma *succ-iff*: $i : succ(j) <-> i=j \mid i:j$
⟨*proof*⟩

lemma *succI1* [*simp*]: $i : succ(i)$
⟨*proof*⟩

lemma *succI2*: $i : j ==> i : succ(j)$
⟨*proof*⟩

lemma *succE* [*elim*!]:
$[\!| i : succ(j);\ i=j ==> P;\ i:j ==> P |\!] ==> P$
⟨*proof*⟩

lemma *succCI* [*intro*!]: $(i^{\sim}:j ==> i=j) ==> i: succ(j)$
⟨*proof*⟩

lemma *succ-not-0* [*simp*]: $succ(n) \ ^{\sim}= 0$
⟨*proof*⟩

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*]: $succ(m) = succ(n) <-> m=n$
⟨*proof*⟩

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

2.11 Miniscoping of the Bounded Universal Quantifier

lemma *ball-simps1*:
$(ALL\ x{:}A.\ P(x)\ \&\ Q) \quad <-> (ALL\ x{:}A.\ P(x))\ \&\ (A=0 \mid Q)$
$(ALL\ x{:}A.\ P(x) \mid Q) \quad <-> ((ALL\ x{:}A.\ P(x)) \mid Q)$
$(ALL\ x{:}A.\ P(x) \ --> Q) <-> ((EX\ x{:}A.\ P(x)) \ --> Q)$
$(\sim(ALL\ x{:}A.\ P(x))) <-> (EX\ x{:}A.\ \sim P(x))$
$(ALL\ x{:}0.P(x)) <-> True$
$(ALL\ x{:}succ(i).P(x)) <-> P(i)\ \&\ (ALL\ x{:}i.\ P(x))$

$(ALL\ x{:}cons(a,B).P(x)) <-> P(a)\ \&\ (ALL\ x{:}B.\ P(x))$
$(ALL\ x{:}RepFun(A,f).\ P(x)) <-> (ALL\ y{:}A.\ P(f(y)))$
$(ALL\ x{:}Union(A).P(x)) <-> (ALL\ y{:}A.\ ALL\ x{:}y.\ P(x))$
⟨*proof*⟩

lemma *ball-simps2*:
$(ALL\ x{:}A.\ P\ \&\ Q(x)) <-> (A{=}0 \mid P)\ \&\ (ALL\ x{:}A.\ Q(x))$
$(ALL\ x{:}A.\ P \mid Q(x)) <-> (P \mid (ALL\ x{:}A.\ Q(x)))$
$(ALL\ x{:}A.\ P --> Q(x)) <-> (P --> (ALL\ x{:}A.\ Q(x)))$
⟨*proof*⟩

lemma *ball-simps3*:
$(ALL\ x{:}Collect(A,Q).P(x)) <-> (ALL\ x{:}A.\ Q(x) --> P(x))$
⟨*proof*⟩

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

lemma *ball-conj-distrib*:
$(ALL\ x{:}A.\ P(x)\ \&\ Q(x)) <-> ((ALL\ x{:}A.\ P(x))\ \&\ (ALL\ x{:}A.\ Q(x)))$
⟨*proof*⟩

2.12 Miniscoping of the Bounded Existential Quantifier

lemma *bex-simps1*:
$(EX\ x{:}A.\ P(x)\ \&\ Q) <-> ((EX\ x{:}A.\ P(x))\ \&\ Q)$
$(EX\ x{:}A.\ P(x) \mid Q) <-> (EX\ x{:}A.\ P(x)) \mid (A^{\sim}{=}0\ \&\ Q)$
$(EX\ x{:}A.\ P(x) --> Q) <-> ((ALL\ x{:}A.\ P(x)) --> (A^{\sim}{=}0\ \&\ Q))$
$(EX\ x{:}0.P(x)) <-> False$
$(EX\ x{:}succ(i).P(x)) <-> P(i) \mid (EX\ x{:}i.\ P(x))$
$(EX\ x{:}cons(a,B).P(x)) <-> P(a) \mid (EX\ x{:}B.\ P(x))$
$(EX\ x{:}RepFun(A,f).\ P(x)) <-> (EX\ y{:}A.\ P(f(y)))$
$(EX\ x{:}Union(A).P(x)) <-> (EX\ y{:}A.\ EX\ x{:}y.\ \ P(x))$
$(\sim(EX\ x{:}A.\ P(x))) <-> (ALL\ x{:}A.\ \sim P(x))$
⟨*proof*⟩

lemma *bex-simps2*:
$(EX\ x{:}A.\ P\ \&\ Q(x)) <-> (P\ \&\ (EX\ x{:}A.\ Q(x)))$
$(EX\ x{:}A.\ P \mid Q(x)) <-> (A^{\sim}{=}0\ \&\ P) \mid (EX\ x{:}A.\ Q(x))$
$(EX\ x{:}A.\ P --> Q(x)) <-> ((A{=}0 \mid P) --> (EX\ x{:}A.\ Q(x)))$
⟨*proof*⟩

lemma *bex-simps3*:
$(EX\ x{:}Collect(A,Q).P(x)) <-> (EX\ x{:}A.\ Q(x)\ \&\ P(x))$
⟨*proof*⟩

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

lemma *bex-disj-distrib*:
$(EX\ x{:}A.\ P(x) \mid Q(x)) <-> ((EX\ x{:}A.\ P(x)) \mid (EX\ x{:}A.\ Q(x)))$

⟨*proof*⟩

lemma *bex-triv-one-point1* [*simp*]: (*EX x*:*A*. *x*=*a*) <−> (*a*:*A*)
⟨*proof*⟩

lemma *bex-triv-one-point2* [*simp*]: (*EX x*:*A*. *a*=*x*) <−> (*a*:*A*)
⟨*proof*⟩

lemma *bex-one-point1* [*simp*]: (*EX x*:*A*. *x*=*a* & *P*(*x*)) <−> (*a*:*A* & *P*(*a*))
⟨*proof*⟩

lemma *bex-one-point2* [*simp*]: (*EX x*:*A*. *a*=*x* & *P*(*x*)) <−> (*a*:*A* & *P*(*a*))
⟨*proof*⟩

lemma *ball-one-point1* [*simp*]: (*ALL x*:*A*. *x*=*a* −−> *P*(*x*)) <−> (*a*:*A* −−> *P*(*a*))
⟨*proof*⟩

lemma *ball-one-point2* [*simp*]: (*ALL x*:*A*. *a*=*x* −−> *P*(*x*)) <−> (*a*:*A* −−> *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} = (if Q then A else 0)
RepFun(0,f) = 0
RepFun(succ(i),f) = cons(f(i), RepFun(i,f))
RepFun(cons(a,B),f) = cons(f(a), RepFun(B,f))
⟨*proof*⟩

2.14 Miniscoping of Unions

lemma *UN-simps1*:
(UN x:C. cons(a, B(x))) = (if C=0 then 0 else cons(a, UN x:C. B(x)))
(UN x:C. A(x) Un B′) = (if C=0 then 0 else (UN x:C. A(x)) Un B′)
(UN x:C. A′ Un B(x)) = (if C=0 then 0 else A′ Un (UN x:C. B(x)))
(UN x:C. A(x) Int B′) = ((UN x:C. A(x)) Int B′)
(UN x:C. A′ Int B(x)) = (A′ Int (UN x:C. B(x)))
(UN x:C. A(x) − B′) = ((UN x:C. A(x)) − B′)
(UN x:C. A′ − B(x)) = (if C=0 then 0 else A′ − (INT x:C. B(x)))
⟨*proof*⟩

lemma *UN-simps2*:
(UN x: Union(A). B(x)) = (UN y:A. UN x:y. B(x))

$(UN\ z{:}\ (UN\ x{:}A.\ B(x)).\ C(z)) = (UN\ \ x{:}A.\ UN\ z{:}\ B(x).\ C(z))$
$(UN\ x{:}\ RepFun(A,f).\ B(x)) \quad = (UN\ a{:}A.\ B(f(a)))$
⟨*proof*⟩

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

Opposite of miniscoping: pull the operator out

lemma *UN-extend-simps1*:
$(UN\ x{:}C.\ A(x))\ Un\ B \ = (if\ C{=}0\ then\ B\ else\ (UN\ x{:}C.\ A(x)\ Un\ B))$
$((UN\ x{:}C.\ A(x))\ Int\ B) = (UN\ x{:}C.\ A(x)\ Int\ B)$
$((UN\ x{:}C.\ A(x)) - B) = (UN\ x{:}C.\ A(x) - B)$
⟨*proof*⟩

lemma *UN-extend-simps2*:
$cons(a,\ UN\ x{:}C.\ B(x)) = (if\ C{=}0\ then\ \{a\}\ else\ (UN\ x{:}C.\ cons(a,\ B(x))))$
$A\ Un\ (UN\ x{:}C.\ B(x)) \ = (if\ C{=}0\ then\ A\ else\ (UN\ x{:}C.\ A\ Un\ B(x)))$
$(A\ Int\ (UN\ x{:}C.\ B(x))) = (UN\ x{:}C.\ A\ Int\ B(x))$
$A - (INT\ x{:}C.\ B(x)) \ = (if\ C{=}0\ then\ A\ else\ (UN\ x{:}C.\ A - B(x)))$
$(UN\ y{:}A.\ UN\ x{:}y.\ B(x)) = (UN\ x{:}\ Union(A).\ B(x))$
$(UN\ a{:}A.\ B(f(a))) = (UN\ x{:}\ RepFun(A,f).\ B(x))$
⟨*proof*⟩

lemma *UN-UN-extend*:
$(UN\ \ x{:}A.\ UN\ z{:}\ B(x).\ C(z)) = (UN\ z{:}\ (UN\ x{:}A.\ B(x)).\ C(z))$
⟨*proof*⟩

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

2.15 Miniscoping of Intersections

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

lemma *INT-simps2*:
$(INT\ x{:}C.\ A\ Int\ B(x)) = A\ Int\ (INT\ x{:}C.\ B(x))$
$(INT\ x{:}C.\ A - B(x)) \ = (if\ C{=}0\ then\ 0\ else\ A - (UN\ x{:}C.\ B(x)))$
$(INT\ x{:}C.\ cons(a,\ B(x))) = (if\ C{=}0\ then\ 0\ else\ cons(a,\ INT\ x{:}C.\ B(x)))$
$(INT\ x{:}C.\ A\ Un\ B(x)) \ = (if\ C{=}0\ then\ 0\ else\ A\ Un\ (INT\ x{:}C.\ B(x)))$
⟨*proof*⟩

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

Opposite of miniscoping: pull the operator out

lemma *INT-extend-simps1*:
$(INT\ x{:}C.\ A(x))\ Int\ B = (INT\ x{:}C.\ A(x)\ Int\ B)$
$(INT\ x{:}C.\ A(x)) - B = (INT\ x{:}C.\ A(x) - B)$

$(INT\ x{:}C.\ A(x))\ Un\ B\ = (if\ C{=}0\ then\ B\ else\ (INT\ x{:}C.\ A(x)\ Un\ B))$
⟨*proof*⟩

lemma *INT-extend-simps2*:
$A\ Int\ (INT\ x{:}C.\ B(x)) = (INT\ x{:}C.\ A\ Int\ B(x))$
$A - (UN\ x{:}C.\ B(x)) \ = (if\ C{=}0\ then\ A\ else\ (INT\ x{:}C.\ A - B(x)))$
$cons(a,\ INT\ x{:}C.\ B(x)) = (if\ C{=}0\ then\ \{a\}\ else\ (INT\ x{:}C.\ cons(a,\ B(x))))$
$A\ Un\ (INT\ x{:}C.\ B(x)) \ = (if\ C{=}0\ then\ A\ else\ (INT\ x{:}C.\ A\ Un\ B(x)))$
⟨*proof*⟩

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

2.16 Other simprules

lemma *misc-simps* [*simp*]:
$0\ Un\ A = A$
$A\ Un\ 0 = A$
$0\ Int\ A = 0$
$A\ Int\ 0 = 0$
$0 - A = 0$
$A - 0 = A$
$Union(0) = 0$
$Union(cons(b,A)) = b\ Un\ Union(A)$
$Inter(\{b\}) = b$
⟨*proof*⟩

end

3 pair: Ordered Pairs

theory *pair* **imports** *upair*
uses *simpdata.ML* **begin**

lemma *singleton-eq-iff* [*iff*]: $\{a\} = \{b\} <-> a{=}b$
⟨*proof*⟩

lemma *doubleton-eq-iff*: $\{a,b\} = \{c,d\} <-> (a{=}c\ \&\ b{=}d) \mid (a{=}d\ \&\ b{=}c)$
⟨*proof*⟩

lemma *Pair-iff* [*simp*]: $<a,b> = <c,d> <-> a{=}c\ \&\ b{=}d$
⟨*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*: $<a,b> \sim= 0$
⟨*proof*⟩

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

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

lemma *Pair-neq-fst*: $<a,b>=a \Longrightarrow P$
⟨*proof*⟩

lemma *Pair-neq-snd*: $<a,b>=b \Longrightarrow P$
⟨*proof*⟩

3.1 Sigma: Disjoint Union of a Family of Sets

Generalizes Cartesian product

lemma *Sigma-iff* [*simp*]: $<a,b>: Sigma(A,B) <-> a:A \;\&\; b:B(a)$
⟨*proof*⟩

lemma *SigmaI* [*TC*,*intro*!]: $[|\ a:A;\ \ b:B(a)\ |] \Longrightarrow <a,b> : Sigma(A,B)$
⟨*proof*⟩

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

lemma *SigmaE* [*elim*!]:
$[|\ c: Sigma(A,B);$
$\quad !!x\ y.[|\ x:A;\ \ y:B(x);\ \ c=<x,y>\ |] \Longrightarrow P$
$|] \Longrightarrow P$
⟨*proof*⟩

lemma *SigmaE2* [*elim*!]:
$[|\ <a,b> : Sigma(A,B);$
$\quad [|\ a:A;\ \ b:B(a)\ |] \Longrightarrow P$
$|] \Longrightarrow P$
⟨*proof*⟩

lemma *Sigma-cong*:
$[|\ A=A';\ \ !!x.\ x:A' \Longrightarrow B(x)=B'(x)\ |] \Longrightarrow$
$Sigma(A,B) = Sigma(A',B')$
⟨*proof*⟩

lemma *Sigma-empty1* [*simp*]: $Sigma(0,B) = 0$
⟨*proof*⟩

lemma *Sigma-empty2* [*simp*]: $A*0 = 0$
⟨*proof*⟩

lemma *Sigma-empty-iff*: $A*B=0 <-> A=0 \mid B=0$
⟨*proof*⟩

3.2 Projections *fst* and *snd*

lemma *fst-conv* [*simp*]: $fst(<a,b>) = a$
⟨*proof*⟩

lemma *snd-conv* [*simp*]: $snd(<a,b>) = b$
⟨*proof*⟩

lemma *fst-type* [*TC*]: p:$Sigma(A,B)$ ==> $fst(p) : A$
⟨*proof*⟩

lemma *snd-type* [*TC*]: p:$Sigma(A,B)$ ==> $snd(p) : B(fst(p))$
⟨*proof*⟩

lemma *Pair-fst-snd-eq*: a: $Sigma(A,B)$ ==> $<fst(a),snd(a)> = a$
⟨*proof*⟩

3.3 The Eliminator, *split*

lemma *split* [*simp*]: $split(\%x\ y.\ c(x,y), <a,b>) == c(a,b)$
⟨*proof*⟩

lemma *split-type* [*TC*]:
[| p:$Sigma(A,B)$;
!!$x\ y$.[| x:A; y:$B(x)$ |] ==> $c(x,y)$:$C(<x,y>)$
|] ==> $split(\%x\ y.\ c(x,y),\ p) : C(p)$
⟨*proof*⟩

lemma *expand-split*:
u: $A*B$ ==>
$R(split(c,u)) <-> (ALL\ x{:}A.\ ALL\ y{:}B.\ u = <x,y> --> R(c(x,y)))$
⟨*proof*⟩

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

lemma *splitI*: $R(a,b)$ ==> $split(R, <a,b>)$
⟨*proof*⟩

lemma *splitE*:
[| $split(R,z)$; z:$Sigma(A,B)$;
!!$x\ y$. [| $z = <x,y>$; $R(x,y)$ |] ==> P
|] ==> P
⟨*proof*⟩

lemma *splitD*: *split*(*R*,<*a*,*b*>) ==> *R*(*a*,*b*)
⟨*proof*⟩

Complex rules for Sigma.

lemma *split-paired-Bex-Sigma* [*simp*]:
(∃ *z* ∈ *Sigma*(*A*,*B*). *P*(*z*)) <-> (∃ *x* ∈ *A*. ∃ *y* ∈ *B*(*x*). *P*(<*x*,*y*>))
⟨*proof*⟩

lemma *split-paired-Ball-Sigma* [*simp*]:
(∀ *z* ∈ *Sigma*(*A*,*B*). *P*(*z*)) <-> (∀ *x* ∈ *A*. ∀ *y* ∈ *B*(*x*). *P*(<*x*,*y*>))
⟨*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*⊆*B* ==> *x*∈*A* --> *x*∈*B*
⟨*proof*⟩

lemma *the-eq-0* [*simp*]: (*THE x*. *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*: (∀ *x* ∈ *A*∪*B*. *P*(*x*)) <-> (∀ *x* ∈ *A*. *P*(*x*)) & (∀ *x* ∈ *B*. *P*(*x*))
⟨*proof*⟩

lemma *bex-Un*: (∃ *x* ∈ *A*∪*B*. *P*(*x*)) <-> (∃ *x* ∈ *A*. *P*(*x*)) | (∃ *x* ∈ *B*. *P*(*x*))
⟨*proof*⟩

lemma *ball-UN*: (∀ *z* ∈ (⋃ *x*∈*A*. *B*(*x*)). *P*(*z*)) <-> (∀ *x*∈*A*. ∀ *z* ∈ *B*(*x*). *P*(*z*))
⟨*proof*⟩

lemma *bex-UN*: (∃ *z* ∈ (⋃ *x*∈*A*. *B*(*x*)). *P*(*z*)) <-> (∃ *x*∈*A*. ∃ *z*∈*B*(*x*). *P*(*z*))
⟨*proof*⟩

4.2 Converse of a Relation

lemma *converse-iff* [*simp*]: $<a,b>\in converse(r) <-> <b,a>\in r$
⟨*proof*⟩

lemma *converseI* [*intro*!]: $<a,b>\in r ==> <b,a>\in converse(r)$
⟨*proof*⟩

lemma *converseD*: $<a,b> \in converse(r) ==> <b,a> \in r$
⟨*proof*⟩

lemma *converseE* [*elim*!]:
$[| yx \in converse(r);$
$\quad !!x\ y.\ [| yx=<y,x>;\ <x,y>\in r |] ==> P |]$
$\quad ==> P$
⟨*proof*⟩

lemma *converse-converse*: $r\subseteq Sigma(A,B) ==> converse(converse(r)) = r$
⟨*proof*⟩

lemma *converse-type*: $r\subseteq A*B ==> converse(r)\subseteq B*A$
⟨*proof*⟩

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

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

lemma *converse-subset-iff*:
$A \subseteq Sigma(X,Y) ==> converse(A) \subseteq converse(B) <-> A \subseteq B$
⟨*proof*⟩

4.3 Finite Set Constructions Using *cons*

lemma *cons-subsetI*: $[| a\in C;\ B\subseteq C |] ==> cons(a,B) \subseteq C$
⟨*proof*⟩

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

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

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

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

lemma *subset-cons-iff*: $C \subseteq cons(a,B) <-> C \subseteq B \mid (a \in C \ \&\ C - \{a\} \subseteq B)$
⟨*proof*⟩

lemma *cons-eq*: $\{a\}\ Un\ B = cons(a,B)$
⟨*proof*⟩

lemma *cons-commute*: $cons(a,\ cons(b,\ C)) = cons(b,\ cons(a,\ C))$
⟨*proof*⟩

lemma *cons-absorb*: $a: B ==> cons(a,B) = B$
⟨*proof*⟩

lemma *cons-Diff*: $a: B ==> cons(a,\ B - \{a\}) = B$
⟨*proof*⟩

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

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

lemma [*simp*]: $cons(a,cons(a,B)) = cons(a,B)$
⟨*proof*⟩

lemma *singleton-subsetI*: $a \in C ==> \{a\} \subseteq C$
⟨*proof*⟩

lemma *singleton-subsetD*: $\{a\} \subseteq C ==> a \in C$
⟨*proof*⟩

lemma *subset-succI*: $i \subseteq succ(i)$
⟨*proof*⟩

lemma *succ-subsetI*: $[\mid i \in j;\ \ i \subseteq j \mid] ==> succ(i) \subseteq j$
⟨*proof*⟩

lemma *succ-subsetE*:
$[\mid succ(i) \subseteq j;\ [\mid i \in j;\ \ i \subseteq j \mid] ==> P \mid] ==> P$
⟨*proof*⟩

lemma *succ-subset-iff*: $succ(a) \subseteq B <-> (a \subseteq B \ \&\ a \in B)$
⟨*proof*⟩

4.4 Binary Intersection

lemma *Int-subset-iff*: $C \subseteq A\ Int\ B <-> C \subseteq A\ \&\ C \subseteq B$
⟨*proof*⟩

lemma *Int-lower1*: $A\ Int\ B \subseteq A$
⟨*proof*⟩

lemma *Int-lower2*: $A\ Int\ B \subseteq B$
⟨*proof*⟩

lemma *Int-greatest*: $[|\ C \subseteq A;\ \ C \subseteq B\ |] ==> C \subseteq A\ Int\ B$
⟨*proof*⟩

lemma *Int-cons*: $cons(a,B)\ Int\ C \subseteq cons(a,\ B\ Int\ C)$
⟨*proof*⟩

lemma *Int-absorb* [*simp*]: $A\ Int\ A = A$
⟨*proof*⟩

lemma *Int-left-absorb*: $A\ Int\ (A\ Int\ B) = A\ Int\ B$
⟨*proof*⟩

lemma *Int-commute*: $A\ Int\ B = B\ Int\ A$
⟨*proof*⟩

lemma *Int-left-commute*: $A\ Int\ (B\ Int\ C) = B\ Int\ (A\ Int\ C)$
⟨*proof*⟩

lemma *Int-assoc*: $(A\ Int\ B)\ Int\ C\ =\ A\ Int\ (B\ Int\ C)$
⟨*proof*⟩

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

lemma *Int-absorb1*: $B \subseteq A ==> A \cap B = B$
⟨*proof*⟩

lemma *Int-absorb2*: $A \subseteq B ==> A \cap B = A$
⟨*proof*⟩

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

lemma *Int-Un-distrib2*: $(B\ Un\ C)\ Int\ A = (B\ Int\ A)\ Un\ (C\ Int\ A)$
⟨*proof*⟩

lemma *subset-Int-iff*: $A \subseteq B <-> A\ Int\ B = A$
⟨*proof*⟩

lemma *subset-Int-iff2*: $A \subseteq B <-> B\ Int\ A = A$
⟨*proof*⟩

lemma *Int-Diff-eq*: $C \subseteq A ==> (A-B)\ Int\ C = C-B$
⟨*proof*⟩

lemma *Int-cons-left*:
$cons(a,A)\ Int\ B = (if\ a \in B\ then\ cons(a,\ A\ Int\ B)\ else\ A\ Int\ B)$
⟨*proof*⟩

lemma *Int-cons-right*:
$A\ Int\ cons(a,\ B) = (if\ a \in A\ then\ cons(a,\ A\ Int\ B)\ else\ A\ Int\ B)$
⟨*proof*⟩

lemma *cons-Int-distrib*: $cons(x,\ A \cap B) = cons(x,\ A) \cap cons(x,\ B)$
⟨*proof*⟩

4.5 Binary Union

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

lemma *Un-upper1*: $A \subseteq A\ Un\ B$
⟨*proof*⟩

lemma *Un-upper2*: $B \subseteq A\ Un\ B$
⟨*proof*⟩

lemma *Un-least*: $[|\ A \subseteq C;\ \ B \subseteq C\ |] ==> A\ Un\ B \subseteq C$
⟨*proof*⟩

lemma *Un-cons*: $cons(a,B)\ Un\ C = cons(a,\ B\ Un\ C)$
⟨*proof*⟩

lemma *Un-absorb* [*simp*]: $A\ Un\ A = A$
⟨*proof*⟩

lemma *Un-left-absorb*: $A\ Un\ (A\ Un\ B) = A\ Un\ B$
⟨*proof*⟩

lemma *Un-commute*: $A\ Un\ B = B\ Un\ A$
⟨*proof*⟩

lemma *Un-left-commute*: $A\ Un\ (B\ Un\ C) = B\ Un\ (A\ Un\ C)$
⟨*proof*⟩

lemma *Un-assoc*: $(A\ Un\ B)\ Un\ C\ =\ A\ Un\ (B\ Un\ C)$
⟨*proof*⟩

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

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

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

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

lemma *subset-Un-iff*: $A \subseteq B <-> A\ Un\ B = B$
⟨*proof*⟩

lemma *subset-Un-iff2*: $A \subseteq B <-> B\ Un\ A = B$
⟨*proof*⟩

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

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

4.6 Set Difference

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

lemma *Diff-contains*: $[|\ C \subseteq A;\ \ C\ Int\ B = 0\ |] \Longrightarrow C \subseteq A-B$
⟨*proof*⟩

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

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

lemma *Diff-triv*: $A\ \ Int\ B = 0 \Longrightarrow A - B = A$
⟨*proof*⟩

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

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

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

lemma *Diff-cons*: $A - cons(a,B) = A - B - \{a\}$
⟨*proof*⟩

lemma *Diff-cons2*: $A - cons(a,B) = A - \{a\} - B$
⟨*proof*⟩

lemma *Diff-disjoint*: $A\ Int\ (B-A) = 0$
⟨*proof*⟩

lemma *Diff-partition*: $A \subseteq B \Longrightarrow A\ Un\ (B-A) = B$
⟨*proof*⟩

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

lemma *double-complement*: $[|\ A \subseteq B;\ B \subseteq C\ |] \Longrightarrow B-(C-A) = A$
⟨*proof*⟩

lemma *double-complement-Un*: $(A\ Un\ B) - (B-A) = A$
⟨*proof*⟩

lemma *Un-Int-crazy*:
$(A\ Int\ B)\ Un\ (B\ Int\ C)\ Un\ (C\ Int\ A) = (A\ Un\ B)\ Int\ (B\ Un\ C)\ Int\ (C\ Un\ A)$
⟨*proof*⟩

lemma *Diff-Un*: $A - (B\ Un\ C) = (A-B)\ Int\ (A-C)$
⟨*proof*⟩

lemma *Diff-Int*: $A - (B\ Int\ C) = (A-B)\ Un\ (A-C)$
⟨*proof*⟩

lemma *Un-Diff*: $(A\ Un\ B) - C = (A - C)\ Un\ (B - C)$
⟨*proof*⟩

lemma *Int-Diff*: $(A\ Int\ B) - C = A\ Int\ (B - C)$
⟨*proof*⟩

lemma *Diff-Int-distrib*: $C\ Int\ (A-B) = (C\ Int\ A) - (C\ Int\ B)$
⟨*proof*⟩

lemma *Diff-Int-distrib2*: $(A-B)\ Int\ C = (A\ Int\ C) - (B\ Int\ C)$
⟨*proof*⟩

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

4.7 Big Union and Intersection

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

lemma *Union-upper*: $B{\in}A ==> B \subseteq Union(A)$
⟨*proof*⟩

lemma *Union-least*: $[|\ !!x.\ x{\in}A ==> x{\subseteq}C\ |] ==> Union(A) \subseteq C$
⟨*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*: $Union(C)\ Int\ A = 0 <-> (\forall B{\in}C.\ B\ Int\ A = 0)$
⟨*proof*⟩

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

lemma *Int-Union2*: $Union(B)\ Int\ A = (\bigcup C{\in}B.\ C\ Int\ A)$
⟨*proof*⟩

lemma *Inter-subset-iff*: $A{\neq}0\ ==>\ C \subseteq Inter(A) <-> (\forall x{\in}A.\ C \subseteq x)$
⟨*proof*⟩

lemma *Inter-lower*: $B{\in}A ==> Inter(A) \subseteq B$
⟨*proof*⟩

lemma *Inter-greatest*: $[|\ A{\neq}0;\ \ !!x.\ x{\in}A ==> C{\subseteq}x\ |] ==> C \subseteq Inter(A)$
⟨*proof*⟩

lemma *INT-lower*: $x{\in}A ==> (\bigcap x{\in}A.\ B(x)) \subseteq B(x)$
⟨*proof*⟩

lemma *INT-greatest*: $[|\ A{\neq}0;\ \ !!x.\ x{\in}A ==> C{\subseteq}B(x)\ |] ==> C \subseteq (\bigcap x{\in}A.\ B(x))$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *Inter-Un-subset*:
[| $z \in A$; $z \in B$ |] ==> $Inter(A)\ Un\ Inter(B) \subseteq Inter(A\ Int\ B)$
⟨*proof*⟩

lemma *Inter-Un-distrib*:
[| $A \neq 0$; $B \neq 0$ |] ==> $Inter(A\ Un\ B) = Inter(A)\ Int\ Inter(B)$
⟨*proof*⟩

lemma *Union-singleton*: $Union(\{b\}) = b$
⟨*proof*⟩

lemma *Inter-singleton*: $Inter(\{b\}) = b$
⟨*proof*⟩

lemma *Inter-cons* [*simp*]:
$Inter(cons(a,B)) = (if\ B=0\ then\ a\ else\ a\ Int\ Inter(B))$
⟨*proof*⟩

4.8 Unions and Intersections of Families

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

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

lemma *UN-upper*: $x \in A$ ==> $B(x) \subseteq (\bigcup x \in A.\ B(x))$
⟨*proof*⟩

lemma *UN-least*: [| !!x. $x \in A$ ==> $B(x) \subseteq C$ |] ==> $(\bigcup x \in A.\ B(x)) \subseteq C$
⟨*proof*⟩

lemma *Union-eq-UN*: $Union(A) = (\bigcup x \in A.\ x)$
⟨*proof*⟩

lemma *Inter-eq-INT*: $Inter(A) = (\bigcap x \in A.\ x)$
⟨*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*: $(\bigcup i \in A\ Un\ B.\ C(i)) = (\bigcup i \in A.\ C(i))\ Un\ (\bigcup i \in B.\ C(i))$
⟨*proof*⟩

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

lemma *Un-INT-distrib*: $I{\neq}0 \Longrightarrow B\ Un\ (\bigcap i{\in}I.\ A(i)) = (\bigcap i{\in}I.\ B\ Un\ A(i))$
⟨*proof*⟩

lemma *Int-UN-distrib2*:
$(\bigcup i{\in}I.\ A(i))\ Int\ (\bigcup j{\in}J.\ B(j)) = (\bigcup i{\in}I.\ \bigcup j{\in}J.\ A(i)\ Int\ B(j))$
⟨*proof*⟩

lemma *Un-INT-distrib2*: $[|\ I{\neq}0;\ \ J{\neq}0\ |] \Longrightarrow$
$(\bigcap i{\in}I.\ A(i))\ Un\ (\bigcap j{\in}J.\ B(j)) = (\bigcap i{\in}I.\ \bigcap j{\in}J.\ A(i)\ Un\ B(j))$
⟨*proof*⟩

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

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

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

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

lemma *INT-Union-eq*:
$0\ {\sim}{:}\ A \Longrightarrow (\bigcap x{\in}\ Union(A).\ B(x)) = (\bigcap y{\in}A.\ \bigcap x{\in}y.\ B(x))$
⟨*proof*⟩

lemma *INT-UN-eq*:
$(\forall x{\in}A.\ B(x)\ {\sim}{=}\ 0)$
$\Longrightarrow (\bigcap z{\in}\ (\bigcup x{\in}A.\ B(x)).\ C(z)) = (\bigcap x{\in}A.\ \bigcap z{\in}\ B(x).\ C(z))$
⟨*proof*⟩

lemma *UN-Un-distrib*:
(⋃ i∈I. A(i) Un B(i)) = (⋃ i∈I. A(i)) Un (⋃ i∈I. B(i))
⟨*proof*⟩

lemma *INT-Int-distrib*:
I≠0 ==> (⋂ i∈I. A(i) Int B(i)) = (⋂ i∈I. A(i)) Int (⋂ i∈I. B(i))
⟨*proof*⟩

lemma *UN-Int-subset*:
(⋃ z∈I Int J. A(z)) ⊆ (⋃ z∈I. A(z)) Int (⋃ z∈J. A(z))
⟨*proof*⟩

lemma *Diff-UN*: I≠0 ==> B − (⋃ i∈I. A(i)) = (⋂ i∈I. B − A(i))
⟨*proof*⟩

lemma *Diff-INT*: I≠0 ==> B − (⋂ i∈I. A(i)) = (⋃ i∈I. B − A(i))
⟨*proof*⟩

lemma *Sigma-cons1*: Sigma(cons(a,B), C) = ({a}∗C(a)) Un Sigma(B,C)
⟨*proof*⟩

lemma *Sigma-cons2*: A ∗ cons(b,B) = A∗{b} Un A∗B
⟨*proof*⟩

lemma *Sigma-succ1*: Sigma(succ(A), B) = ({A}∗B(A)) Un Sigma(A,B)
⟨*proof*⟩

lemma *Sigma-succ2*: A ∗ succ(B) = A∗{B} Un A∗B
⟨*proof*⟩

lemma *SUM-UN-distrib1*:
(Σ x ∈ (⋃ y∈A. C(y)). B(x)) = (⋃ y∈A. Σ x∈C(y). B(x))
⟨*proof*⟩

lemma *SUM-UN-distrib2*:
(Σ i∈I. ⋃ j∈J. C(i,j)) = (⋃ j∈J. Σ i∈I. C(i,j))
⟨*proof*⟩

lemma *SUM-Un-distrib1*:
(Σ i∈I Un J. C(i)) = (Σ i∈I. C(i)) Un (Σ j∈J. C(j))
⟨*proof*⟩

lemma *SUM-Un-distrib2*:
(Σ i∈I. $A(i)$ *Un* $B(i)$) = (Σ i∈I. $A(i)$) *Un* (Σ i∈I. $B(i)$)
⟨*proof*⟩

lemma *prod-Un-distrib2*: I ∗ (A *Un* B) = I∗A *Un* I∗B
⟨*proof*⟩

lemma *SUM-Int-distrib1*:
(Σ i∈I *Int* J. $C(i)$) = (Σ i∈I. $C(i)$) *Int* (Σ j∈J. $C(j)$)
⟨*proof*⟩

lemma *SUM-Int-distrib2*:
(Σ i∈I. $A(i)$ *Int* $B(i)$) = (Σ i∈I. $A(i)$) *Int* (Σ i∈I. $B(i)$)
⟨*proof*⟩

lemma *prod-Int-distrib2*: I ∗ (A *Int* B) = I∗A *Int* I∗B
⟨*proof*⟩

lemma *SUM-eq-UN*: (Σ i∈I. $A(i)$) = (⋃i∈I. {i} ∗ $A(i)$)
⟨*proof*⟩

lemma *times-subset-iff*:
(A'∗B' ⊆ A∗B) <−> (A' = *0* | B' = *0* | (A'⊆A) & (B'⊆B))
⟨*proof*⟩

lemma *Int-Sigma-eq*:
(Σ x ∈ A'. $B'(x)$) *Int* (Σ x ∈ A. $B(x)$) = (Σ x ∈ A' *Int* A. $B'(x)$ *Int* $B(x)$)
⟨*proof*⟩

lemma *domain-iff*: a: *domain*(r) <−> (*EX* y. <a,y>∈ r)
⟨*proof*⟩

lemma *domainI* [*intro*]: <a,b>∈ r ==> a: *domain*(r)
⟨*proof*⟩

lemma *domainE* [*elim*!]:
[| a ∈ *domain*(r); !!y. <a,y>∈ r ==> P |] ==> P
⟨*proof*⟩

lemma *domain-subset*: *domain*(*Sigma*(A,B)) ⊆ A
⟨*proof*⟩

lemma *domain-of-prod*: b∈B ==> *domain*(A∗B) = A

⟨*proof*⟩

lemma *domain-0* [*simp*]: $domain(0) = 0$
⟨*proof*⟩

lemma *domain-cons* [*simp*]: $domain(cons(<a,b>,r)) = cons(a,\ domain(r))$
⟨*proof*⟩

lemma *domain-Un-eq* [*simp*]: $domain(A\ Un\ B) = domain(A)\ Un\ domain(B)$
⟨*proof*⟩

lemma *domain-Int-subset*: $domain(A\ Int\ B) \subseteq domain(A)\ Int\ domain(B)$
⟨*proof*⟩

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

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

lemma *domain-Union*: $domain(Union(A)) = (\bigcup x{\in}A.\ domain(x))$
⟨*proof*⟩

lemma *rangeI* [*intro*]: $<a,b>{\in}\ r \Longrightarrow b \in range(r)$
⟨*proof*⟩

lemma *rangeE* [*elim*!]: $[|\ b \in range(r);\ !!x.\ <x,b>{\in}\ r \Longrightarrow P\ |] \Longrightarrow P$
⟨*proof*⟩

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

lemma *range-of-prod*: $a{\in}A \Longrightarrow range(A{*}B) = B$
⟨*proof*⟩

lemma *range-0* [*simp*]: $range(0) = 0$
⟨*proof*⟩

lemma *range-cons* [*simp*]: $range(cons(<a,b>,r)) = cons(b,\ range(r))$
⟨*proof*⟩

lemma *range-Un-eq* [*simp*]: $range(A\ Un\ B) = range(A)\ Un\ range(B)$
⟨*proof*⟩

lemma *range-Int-subset*: $range(A\ Int\ B) \subseteq range(A)\ Int\ range(B)$
⟨*proof*⟩

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

lemma *domain-converse* [*simp*]: $domain(converse(r)) = range(r)$
⟨*proof*⟩

lemma *range-converse* [*simp*]: $range(converse(r)) = domain(r)$
⟨*proof*⟩

lemma *fieldI1*: $<a,b>\in r \Longrightarrow a \in field(r)$
⟨*proof*⟩

lemma *fieldI2*: $<a,b>\in r \Longrightarrow b \in field(r)$
⟨*proof*⟩

lemma *fieldCI* [*intro*]:
$(\sim <c,a>\in r \Longrightarrow <a,b>\in r) \Longrightarrow a \in field(r)$
⟨*proof*⟩

lemma *fieldE* [*elim*!]:
$[|\ a \in field(r);$
$!!x.\ <a,x>\in r \Longrightarrow P;$
$!!x.\ <x,a>\in r \Longrightarrow P \quad |] \Longrightarrow P$
⟨*proof*⟩

lemma *field-subset*: $field(A*B) \subseteq A\ Un\ B$
⟨*proof*⟩

lemma *domain-subset-field*: $domain(r) \subseteq field(r)$
⟨*proof*⟩

lemma *range-subset-field*: $range(r) \subseteq field(r)$
⟨*proof*⟩

lemma *domain-times-range*: $r \subseteq Sigma(A,B) \Longrightarrow r \subseteq domain(r)*range(r)$
⟨*proof*⟩

lemma *field-times-field*: $r \subseteq Sigma(A,B) \Longrightarrow r \subseteq field(r)*field(r)$
⟨*proof*⟩

lemma *relation-field-times-field*: $relation(r) \Longrightarrow r \subseteq field(r)*field(r)$
⟨*proof*⟩

lemma *field-of-prod*: $field(A*A) = A$
⟨*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 Un B*) = *field*(*A*) *Un field*(*B*)
⟨*proof*⟩

lemma *field-Int-subset*: *field*(*A Int B*) ⊆ *field*(*A*) *Int field*(*B*)
⟨*proof*⟩

lemma *field-Diff-subset*: *field*(*A*) − *field*(*B*) ⊆ *field*(*A* − *B*)
⟨*proof*⟩

lemma *field-converse* [*simp*]: *field*(*converse*(*r*)) = *field*(*r*)
⟨*proof*⟩

lemma *rel-Union*: (∀ *x*∈*S*. *EX A B*. *x* ⊆ *A*∗*B*) ==>
Union(*S*) ⊆ *domain*(*Union*(*S*)) ∗ *range*(*Union*(*S*))
⟨*proof*⟩

lemma *rel-Un*: [| *r* ⊆ *A*∗*B*; *s* ⊆ *C*∗*D* |] ==> (*r Un s*) ⊆ (*A Un C*) ∗ (*B Un D*)
⟨*proof*⟩

lemma *domain-Diff-eq*: [| <*a*,*c*> ∈ *r*; *c*~=*b* |] ==> *domain*(*r*−{<*a*,*b*>}) = *domain*(*r*)
⟨*proof*⟩

lemma *range-Diff-eq*: [| <*c*,*b*> ∈ *r*; *c*~=*a* |] ==> *range*(*r*−{<*a*,*b*>}) = *range*(*r*)
⟨*proof*⟩

4.9 Image of a Set under a Function or Relation

lemma *image-iff*: *b* ∈ *r*‘‘*A* <−> (∃ *x*∈*A*. <*x*,*b*>∈*r*)
⟨*proof*⟩

lemma *image-singleton-iff*: *b* ∈ *r*‘‘{*a*} <−> <*a*,*b*>∈*r*
⟨*proof*⟩

lemma *imageI* [*intro*]: [| <*a*,*b*>∈ *r*; *a*∈*A* |] ==> *b* ∈ *r*‘‘*A*
⟨*proof*⟩

lemma *imageE* [*elim*!]:
[| *b*: *r*‘‘*A*; !!*x*.[| <*x*,*b*>∈ *r*; *x*∈*A* |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *image-subset*: $r \subseteq A*B \Longrightarrow r``C \subseteq B$
⟨*proof*⟩

lemma *image-0* [*simp*]: $r``0 = 0$
⟨*proof*⟩

lemma *image-Un* [*simp*]: $r``(A\ Un\ B) = (r``A)\ Un\ (r``B)$
⟨*proof*⟩

lemma *image-UN*: $r\ ``\ (\bigcup x{\in}A.\ B(x)) = (\bigcup x{\in}A.\ r\ ``\ B(x))$
⟨*proof*⟩

lemma *Collect-image-eq*:
$\{z \in Sigma(A,B).\ P(z)\}\ ``\ C = (\bigcup x \in A.\ \{y \in B(x).\ x \in C\ \&\ P(<x,y>)\})$
⟨*proof*⟩

lemma *image-Int-subset*: $r``(A\ Int\ B) \subseteq (r``A)\ Int\ (r``B)$
⟨*proof*⟩

lemma *image-Int-square-subset*: $(r\ Int\ A*A)``B \subseteq (r``B)\ Int\ A$
⟨*proof*⟩

lemma *image-Int-square*: $B{\subseteq}A \Longrightarrow (r\ Int\ A*A)``B = (r``B)\ Int\ A$
⟨*proof*⟩

lemma *image-0-left* [*simp*]: $0``A = 0$
⟨*proof*⟩

lemma *image-Un-left*: $(r\ Un\ s)``A = (r``A)\ Un\ (s``A)$
⟨*proof*⟩

lemma *image-Int-subset-left*: $(r\ Int\ s)``A \subseteq (r``A)\ Int\ (s``A)$
⟨*proof*⟩

4.10 Inverse Image of a Set under a Function or Relation

lemma *vimage-iff*:
$a \in r-``B \mathrel{<->} (\exists\, y{\in}B.\ <a,y>{\in}r)$
⟨*proof*⟩

lemma *vimage-singleton-iff*: $a \in r-``\{b\} \mathrel{<->} <a,b>{\in}r$
⟨*proof*⟩

lemma *vimageI* [*intro*]: $[|\ <a,b>{\in}\ r;\ \ b{\in}B\ |] \Longrightarrow a \in r-``B$
⟨*proof*⟩

lemma *vimageE* [*elim*!]:
[| *a*: *r*−‘‘*B*; !!*x*.[| <*a*,*x*>∈ *r*; *x*∈*B* |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *vimage-subset*: *r* ⊆ *A*∗*B* ==> *r*−‘‘*C* ⊆ *A*
⟨*proof*⟩

lemma *vimage-0* [*simp*]: *r*−‘‘*0* = *0*
⟨*proof*⟩

lemma *vimage-Un* [*simp*]: *r*−‘‘(*A Un B*) = (*r*−‘‘*A*) *Un* (*r*−‘‘*B*)
⟨*proof*⟩

lemma *vimage-Int-subset*: *r*−‘‘(*A Int B*) ⊆ (*r*−‘‘*A*) *Int* (*r*−‘‘*B*)
⟨*proof*⟩

lemma *vimage-eq-UN*: *f* −‘‘*B* = (⋃ *y*∈*B*. *f*−‘‘{*y*})
⟨*proof*⟩

lemma *function-vimage-Int*:
function(*f*) ==> *f*−‘‘(*A Int B*) = (*f*−‘‘*A*) *Int* (*f*−‘‘*B*)
⟨*proof*⟩

lemma *function-vimage-Diff*: *function*(*f*) ==> *f*−‘‘(*A*−*B*) = (*f*−‘‘*A*) − (*f*−‘‘*B*)
⟨*proof*⟩

lemma *function-image-vimage*: *function*(*f*) ==> *f* ‘‘ (*f*−‘‘ *A*) ⊆ *A*
⟨*proof*⟩

lemma *vimage-Int-square-subset*: (*r Int A*∗*A*)−‘‘*B* ⊆ (*r*−‘‘*B*) *Int A*
⟨*proof*⟩

lemma *vimage-Int-square*: *B*⊆*A* ==> (*r Int A*∗*A*)−‘‘*B* = (*r*−‘‘*B*) *Int A*
⟨*proof*⟩

lemma *vimage-0-left* [*simp*]: *0*−‘‘*A* = *0*
⟨*proof*⟩

lemma *vimage-Un-left*: (*r Un s*)−‘‘*A* = (*r*−‘‘*A*) *Un* (*s*−‘‘*A*)
⟨*proof*⟩

lemma *vimage-Int-subset-left*: (*r Int s*)−‘‘*A* ⊆ (*r*−‘‘*A*) *Int* (*s*−‘‘*A*)
⟨*proof*⟩

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

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

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

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

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

4.11 Powerset Operator

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

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

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

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

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

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

lemma *Union-Pow-iff*: $Union(A) \in Pow(B) <-> A \in Pow(Pow(B))$
⟨*proof*⟩

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

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

4.12 RepFun

lemma *RepFun-subset*: [| !!x. x∈A ==> $f(x)$ ∈ B |] ==> {$f(x)$. x∈A} ⊆ B
⟨*proof*⟩

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

lemma *RepFun-constant* [*simp*]: {c. x∈A} = (*if* A=0 *then* 0 *else* {c})
⟨*proof*⟩

4.13 Collect

lemma *Collect-subset*: *Collect*(A,P) ⊆ A
⟨*proof*⟩

lemma *Collect-Un*: *Collect*(A *Un* B, P) = *Collect*(A,P) *Un* *Collect*(B,P)
⟨*proof*⟩

lemma *Collect-Int*: *Collect*(A *Int* B, P) = *Collect*(A,P) *Int* *Collect*(B,P)
⟨*proof*⟩

lemma *Collect-Diff*: *Collect*(A − B, P) = *Collect*(A,P) − *Collect*(B,P)
⟨*proof*⟩

lemma *Collect-cons*: {x∈*cons*(a,B). $P(x)$} =
(*if* $P(a)$ *then* *cons*(a, {x∈B. $P(x)$}) *else* {x∈B. $P(x)$})
⟨*proof*⟩

lemma *Int-Collect-self-eq*: A *Int* *Collect*(A,P) = *Collect*(A,P)
⟨*proof*⟩

lemma *Collect-Collect-eq* [*simp*]:
Collect(*Collect*(A,P), Q) = *Collect*(A, %x. $P(x)$ & $Q(x)$)
⟨*proof*⟩

lemma *Collect-Int-Collect-eq*:
Collect(A,P) *Int* *Collect*(A,Q) = *Collect*(A, %x. $P(x)$ & $Q(x)$)
⟨*proof*⟩

lemma *Collect-Union-eq* [*simp*]:
Collect(⋃x∈A. $B(x)$, P) = (⋃x∈A. *Collect*($B(x)$, P))
⟨*proof*⟩

lemma *Collect-Int-left*: {x∈A. $P(x)$} *Int* B = {x ∈ A *Int* B. $P(x)$}
⟨*proof*⟩

lemma *Collect-Int-right*: A *Int* {x∈B. $P(x)$} = {x ∈ A *Int* B. $P(x)$}
⟨*proof*⟩

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

lemma *Collect-conj-eq*: $\{x{\in}A.\ P(x)\ \&\ Q(x)\} = Collect(A,\ P)\ Int\ Collect(A,\ Q)$
⟨*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 :: $[i,i{=>}i]{=>}o$ **where**
$bnd\text{-}mono(D,h) == h(D){<=}D\ \&\ (ALL\ W\ X.\ W{<=}X\ {-->}\ X{<=}D\ {-->}\ h(W)\ <=\ h(X))$

definition
lfp :: $[i,i{=>}i]{=>}i$ **where**
$lfp(D,h) == Inter(\{X{:}\ Pow(D).\ h(X)\ <=\ X\})$

definition
gfp :: $[i,i{=>}i]{=>}i$ **where**
$gfp(D,h) == Union(\{X{:}\ Pow(D).\ X\ <=\ h(X)\})$

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

5.1 Monotone Operators

lemma *bnd-monoI*:
$[|\ h(D){<=}D;$
$\ \ !!W\ X.\ [|\ W{<=}D;\ \ X{<=}D;\ \ W{<=}X\ |]\ ==>\ h(W)\ <=\ h(X)$
$\ |]\ ==>\ bnd\text{-}mono(D,h)$
⟨*proof*⟩

lemma *bnd-monoD1*: $bnd\text{-}mono(D,h)\ ==>\ h(D)\ <=\ D$

⟨*proof*⟩

lemma *bnd-monoD2*: [| *bnd-mono*(*D*,*h*); *W*<=*X*; *X*<=*D* |] ==> *h*(*W*) <= *h*(*X*)
⟨*proof*⟩

lemma *bnd-mono-subset*:
[| *bnd-mono*(*D*,*h*); *X*<=*D* |] ==> *h*(*X*) <= *D*
⟨*proof*⟩

lemma *bnd-mono-Un*:
[| *bnd-mono*(*D*,*h*); *A* <= *D*; *B* <= *D* |] ==> *h*(*A*) *Un* *h*(*B*) <= *h*(*A* *Un* *B*)
⟨*proof*⟩

lemma *bnd-mono-UN*:
[| *bnd-mono*(*D*,*h*); ∀*i*∈*I*. *A*(*i*) <= *D* |]
==> (⋃*i*∈*I*. *h*(*A*(*i*))) <= *h*((⋃*i*∈*I*. *A*(*i*)))
⟨*proof*⟩

lemma *bnd-mono-Int*:
[| *bnd-mono*(*D*,*h*); *A* <= *D*; *B* <= *D* |] ==> *h*(*A* *Int* *B*) <= *h*(*A*) *Int* *h*(*B*)
⟨*proof*⟩

5.2 Proof of Knaster-Tarski Theorem using *lfp*

lemma *lfp-lowerbound*:
[| *h*(*A*) <= *A*; *A*<=*D* |] ==> *lfp*(*D*,*h*) <= *A*
⟨*proof*⟩

lemma *lfp-subset*: *lfp*(*D*,*h*) <= *D*
⟨*proof*⟩

lemma *def-lfp-subset*: *A* == *lfp*(*D*,*h*) ==> *A* <= *D*
⟨*proof*⟩

lemma *lfp-greatest*:
[| *h*(*D*) <= *D*; !!*X*. [| *h*(*X*) <= *X*; *X*<=*D* |] ==> *A*<=*X* |] ==> *A* <= *lfp*(*D*,*h*)
⟨*proof*⟩

lemma *lfp-lemma1*:
[| *bnd-mono*(*D*,*h*); *h*(*A*)<=*A*; *A*<=*D* |] ==> *h*(*lfp*(*D*,*h*)) <= *A*
⟨*proof*⟩

lemma *lfp-lemma2*: *bnd-mono*(*D*,*h*) ==> *h*(*lfp*(*D*,*h*)) <= *lfp*(*D*,*h*)

⟨*proof*⟩

lemma *lfp-lemma3*:
bnd-mono(*D*,*h*) ==> *lfp*(*D*,*h*) <= *h*(*lfp*(*D*,*h*))
⟨*proof*⟩

lemma *lfp-unfold*: *bnd-mono*(*D*,*h*) ==> *lfp*(*D*,*h*) = *h*(*lfp*(*D*,*h*))
⟨*proof*⟩

lemma *def-lfp-unfold*:
[| *A*==*lfp*(*D*,*h*); *bnd-mono*(*D*,*h*) |] ==> *A* = *h*(*A*)
⟨*proof*⟩

5.3 General Induction Rule for Least Fixedpoints

lemma *Collect-is-pre-fixedpt*:
[| *bnd-mono*(*D*,*h*); !!*x*. *x* : *h*(*Collect*(*lfp*(*D*,*h*),*P*)) ==> *P*(*x*) |]
==> *h*(*Collect*(*lfp*(*D*,*h*),*P*)) <= *Collect*(*lfp*(*D*,*h*),*P*)
⟨*proof*⟩

lemma *induct*:
[| *bnd-mono*(*D*,*h*); *a* : *lfp*(*D*,*h*);
!!*x*. *x* : *h*(*Collect*(*lfp*(*D*,*h*),*P*)) ==> *P*(*x*)
|] ==> *P*(*a*)
⟨*proof*⟩

lemma *def-induct*:
[| *A* == *lfp*(*D*,*h*); *bnd-mono*(*D*,*h*); *a*:*A*;
!!*x*. *x* : *h*(*Collect*(*A*,*P*)) ==> *P*(*x*)
|] ==> *P*(*a*)
⟨*proof*⟩

lemma *lfp-Int-lowerbound*:
[| *h*(*D Int A*) <= *A*; *bnd-mono*(*D*,*h*) |] ==> *lfp*(*D*,*h*) <= *A*
⟨*proof*⟩

lemma *lfp-mono*:
assumes *hmono*: *bnd-mono*(*D*,*h*)
and *imono*: *bnd-mono*(*E*,*i*)
and *subhi*: !!*X*. *X*<=*D* ==> *h*(*X*) <= *i*(*X*)
shows *lfp*(*D*,*h*) <= *lfp*(*E*,*i*)
⟨*proof*⟩

lemma *lfp-mono2*:
[| $i(D) <= D$; !!X. $X<=D ==> h(X) <= i(X)$ |] ==> *lfp*(D,h) <= *lfp*(D,i)
⟨*proof*⟩

lemma *lfp-cong*:
[|D=D'; !!X. $X <= D' ==> h(X) = h'(X)$|] ==> *lfp*(D,h) = *lfp*(D',h')
⟨*proof*⟩

5.4 Proof of Knaster-Tarski Theorem using *gfp*

lemma *gfp-upperbound*: [| $A <= h(A)$; $A<=D$ |] ==> $A <=$ *gfp*(D,h)
⟨*proof*⟩

lemma *gfp-subset*: *gfp*(D,h) <= D
⟨*proof*⟩

lemma *def-gfp-subset*: A==*gfp*(D,h) ==> $A <= D$
⟨*proof*⟩

lemma *gfp-least*:
[| *bnd-mono*(D,h); !!X. [| $X <= h(X)$; $X<=D$ |] ==> $X<=A$ |] ==>
gfp(D,h) <= A
⟨*proof*⟩

lemma *gfp-lemma1*:
[| *bnd-mono*(D,h); $A<=h(A)$; $A<=D$ |] ==> $A <= h($*gfp*(D,h))
⟨*proof*⟩

lemma *gfp-lemma2*: *bnd-mono*(D,h) ==> *gfp*(D,h) <= $h($*gfp*(D,h))
⟨*proof*⟩

lemma *gfp-lemma3*:
bnd-mono(D,h) ==> $h($*gfp*(D,h)) <= *gfp*(D,h)
⟨*proof*⟩

lemma *gfp-unfold*: *bnd-mono*(D,h) ==> *gfp*(D,h) = $h($*gfp*(D,h))
⟨*proof*⟩

lemma *def-gfp-unfold*:
[| A==*gfp*(D,h); *bnd-mono*(D,h) |] ==> $A = h(A)$
⟨*proof*⟩

5.5 Coinduction Rules for Greatest Fixed Points

lemma *weak-coinduct*: [| a: X; $X <= h(X)$; $X <= D$ |] ==> a : *gfp*(D,h)
⟨*proof*⟩

lemma *coinduct-lemma*:

[| $X <= h(X\ Un\ gfp(D,h))$; $X <= D$; $bnd\text{-}mono(D,h)$ |] ==>
$X\ Un\ gfp(D,h) <= h(X\ Un\ gfp(D,h))$
⟨*proof*⟩

lemma *coinduct*:
[| $bnd\text{-}mono(D,h)$; a: X; $X <= h(X\ Un\ gfp(D,h))$; $X <= D$ |]
==> $a : gfp(D,h)$
⟨*proof*⟩

lemma *def-coinduct*:
[| $A == gfp(D,h)$; $bnd\text{-}mono(D,h)$; a: X; $X <= h(X\ Un\ A)$; $X <= D$ |]
==>
$a : A$
⟨*proof*⟩

lemma *def-Collect-coinduct*:
[| $A == gfp(D, \%w.\ Collect(D,P(w)))$; $bnd\text{-}mono(D, \%w.\ Collect(D,P(w)))$;

a: X; $X <= D$; !!z. z: X ==> $P(X\ Un\ A, z)$ |] ==>
$a : A$
⟨*proof*⟩

lemma *gfp-mono*:
[| $bnd\text{-}mono(D,h)$; $D <= E$;
!!X. $X<=D$ ==> $h(X) <= i(X)$ |] ==> $gfp(D,h) <= gfp(E,i)$
⟨*proof*⟩

end

6 Bool: Booleans in Zermelo-Fraenkel Set Theory

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

abbreviation
one (*1*) **where**
1 == *succ*(*0*)

abbreviation
two (*2*) **where**
2 == *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*
⟨*proof*⟩

lemma *bool-1I* [*simp*,*TC*]: *1* : *bool*
⟨*proof*⟩

lemma *bool-0I* [*simp*,*TC*]: *0* : *bool*
⟨*proof*⟩

lemma *one-not-0*: *1*~=*0*
⟨*proof*⟩

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

lemma *boolE*:
  [| *c*: *bool*; *c=1* ==> *P*; *c=0* ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *cond-1* [*simp*]: *cond*(*1*,*c*,*d*) = *c*
⟨*proof*⟩

lemma *cond-0* [*simp*]: *cond*(*0*,*c*,*d*) = *d*
⟨*proof*⟩

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*:*bool* ==> *not*(*a or b*) = *not*(*a*) *and not*(*b*)
⟨*proof*⟩

6.2 Laws About 'and'

lemma *and-absorb* [*simp*]: *a*: *bool* ==> *a and a* = *a*
⟨*proof*⟩

lemma *and-commute*: [| *a*: *bool*; *b*:*bool* |] ==> *a and b* = *b and a*
⟨*proof*⟩

lemma *and-assoc*: *a*: *bool* ==> (*a and b*) *and c* = *a and* (*b and c*)
⟨*proof*⟩

lemma *and-or-distrib*: [| *a*: *bool*; *b*:*bool*; *c*:*bool* |] ==>
(*a or b*) *and c* = (*a and c*) *or* (*b and c*)
⟨*proof*⟩

6.3 Laws About 'or'

lemma *or-absorb* [*simp*]: *a*: *bool* ==> *a or a* = *a*
⟨*proof*⟩

lemma *or-commute*: [| *a*: *bool*; *b*:*bool* |] ==> *a or b* = *b or a*
⟨*proof*⟩

lemma *or-assoc*: *a*: *bool* ==> (*a or b*) *or c* = *a or* (*b or c*)
⟨*proof*⟩

lemma *or-and-distrib*: [| *a*: *bool*; *b*: *bool*; *c*: *bool* |] ==>
(*a and b*) *or c* = (*a or c*) *and* (*b or c*)
⟨*proof*⟩

definition
bool-of-o :: *o*=>*i* **where**
bool-of-o(*P*) == (*if P then 1 else 0*)

lemma [*simp*]: *bool-of-o*(*True*) = *1*
⟨*proof*⟩

lemma [*simp*]: *bool-of-o*(*False*) = *0*
⟨*proof*⟩

lemma [*simp*,*TC*]: *bool-of-o*(*P*) ∈ *bool*
⟨*proof*⟩

lemma [*simp*]: (*bool-of-o*(*P*) = *1*) <-> *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

global

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

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

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

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

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

local

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* ==> *b*: *B*
⟨*proof*⟩

lemma *sum-iff*: *u*: *A*+*B* <-> (*EX x*. *x*:*A* & *u*=*Inl*(*x*)) | (*EX y*. *y*:*B* & *u*=*Inr*(*y*))
⟨*proof*⟩

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

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

lemma *sum-subset-iff*: *A*+*B* <= *C*+*D* <-> *A*<=*C* & *B*<=*D*
⟨*proof*⟩

lemma *sum-equal-iff*: *A*+*B* = *C*+*D* <-> *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* ==> *c*(*x*): *C*(*Inl*(*x*));
!!*y*. *y*: *B* ==> *d*(*y*): *C*(*Inr*(*y*))
|] ==> *case*(*c*,*d*,*u*) : *C*(*u*)
⟨*proof*⟩

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

lemma *case-cong*:
[| z: $A+B$;
!!x. x:A ==> $c(x)=c'(x)$;
!!y. y:B ==> $d(y)=d'(y)$
|] ==> $case(c,d,z) = case(c',d',z)$
⟨*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*: $C <= A+B$ ==> $Part(C,Inl)\ Un\ Part(C,Inr) = C$
⟨*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*:
$f: Pi(A,B) <-> function(f) \& f<=Sigma(A,B) \& A<=domain(f)$
⟨*proof*⟩

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

lemma *Pi-cong*:
$[|\ A=A';\ \ !!x.\ x{:}A' ==> B(x)=B'(x)\ |] ==> Pi(A,B) = Pi(A',B')$
⟨*proof*⟩

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

lemma *apply-0*: *a* ~: *domain*(*f*) ==> *f*‘*a* = *0*
⟨*proof*⟩

lemma *Pi-memberD*: [| *f*: *Pi*(*A*,*B*); *c*: *f* |] ==> *EX x*:*A*. *c* = <*x*,*f*‘*x*>
⟨*proof*⟩

lemma *function-apply-Pair*: [| *function*(*f*); *a* : *domain*(*f*)|] ==> <*a*,*f*‘*a*>: *f*
⟨*proof*⟩

lemma *apply-Pair*: [| *f*: *Pi*(*A*,*B*); *a*:*A* |] ==> <*a*,*f*‘*a*>: *f*
⟨*proof*⟩

lemma *apply-type* [*TC*]: [| *f*: *Pi*(*A*,*B*); *a*:*A* |] ==> *f*‘*a* : *B*(*a*)
⟨*proof*⟩

lemma *apply-funtype*: [| *f*: *A*->*B*; *a*:*A* |] ==> *f*‘*a* : *B*
⟨*proof*⟩

lemma *apply-iff*: *f*: *Pi*(*A*,*B*) ==> <*a*,*b*>: *f* <-> *a*:*A* & *f*‘*a* = *b*
⟨*proof*⟩

lemma *Pi-type*: [| *f*: *Pi*(*A*,*C*); !!*x*. *x*:*A* ==> *f*‘*x* : *B*(*x*) |] ==> *f* : *Pi*(*A*,*B*)
⟨*proof*⟩

lemma *Pi-Collect-iff*:
    (*f* : *Pi*(*A*, %*x*. {*y*:*B*(*x*). *P*(*x*,*y*)}))
     <-> *f* : *Pi*(*A*,*B*) & (*ALL x*: *A*. *P*(*x*, *f*‘*x*))
⟨*proof*⟩

lemma *Pi-weaken-type*:
      [| *f* : *Pi*(*A*,*B*); !!*x*. *x*:*A* ==> *B*(*x*)<=*C*(*x*) |] ==> *f* : *Pi*(*A*,*C*)
⟨*proof*⟩

lemma *domain-type*: [| <*a*,*b*> : *f*; *f*: *Pi*(*A*,*B*) |] ==> *a* : *A*

⟨*proof*⟩

lemma *range-type*: [| <*a*,*b*> : *f*; *f*: *Pi*(*A*,*B*) |] ==> *b* : *B*(*a*)
⟨*proof*⟩

lemma *Pair-mem-PiD*: [| <*a*,*b*>: *f*; *f*: *Pi*(*A*,*B*) |] ==> *a*:*A* & *b*:*B*(*a*) & *f*‘*a* = *b*
⟨*proof*⟩

8.3 Lambda Abstraction

lemma *lamI*: *a*:*A* ==> <*a*,*b*(*a*)> : (*lam* *x*:*A*. *b*(*x*))
⟨*proof*⟩

lemma *lamE*:
[| *p*: (*lam* *x*:*A*. *b*(*x*)); !!*x*.[| *x*:*A*; *p*=<*x*,*b*(*x*)> |] ==> *P*
|] ==> *P*
⟨*proof*⟩

lemma *lamD*: [| <*a*,*c*>: (*lam* *x*:*A*. *b*(*x*)) |] ==> *c* = *b*(*a*)
⟨*proof*⟩

lemma *lam-type* [*TC*]:
[| !!*x*. *x*:*A* ==> *b*(*x*): *B*(*x*) |] ==> (*lam* *x*:*A*. *b*(*x*)) : *Pi*(*A*,*B*)
⟨*proof*⟩

lemma *lam-funtype*: (*lam* *x*:*A*. *b*(*x*)) : *A* -> {*b*(*x*). *x*:*A*}
⟨*proof*⟩

lemma *function-lam*: *function* (*lam* *x*:*A*. *b*(*x*))
⟨*proof*⟩

lemma *relation-lam*: *relation* (*lam* *x*:*A*. *b*(*x*))
⟨*proof*⟩

lemma *beta-if* [*simp*]: (*lam* *x*:*A*. *b*(*x*)) ‘ *a* = (*if* *a* : *A* *then* *b*(*a*) *else* *0*)
⟨*proof*⟩

lemma *beta*: *a* : *A* ==> (*lam* *x*:*A*. *b*(*x*)) ‘ *a* = *b*(*a*)
⟨*proof*⟩

lemma *lam-empty* [*simp*]: (*lam* *x*:*0*. *b*(*x*)) = *0*
⟨*proof*⟩

lemma *domain-lam* [*simp*]: *domain*(*Lambda*(*A*,*b*)) = *A*
⟨*proof*⟩

lemma *lam-cong* [*cong*]:
[| *A*=*A*′; !!*x*. *x*:*A*′ ==> *b*(*x*)=*b*′(*x*) |] ==> *Lambda*(*A*,*b*) = *Lambda*(*A*′,*b*′)

⟨*proof*⟩

lemma *lam-theI*:
(!!x. x:A ==> $EX!$ y. $Q(x,y)$) ==> $EX\, f$. $ALL\ x$:A. $Q(x, f`x)$
⟨*proof*⟩

lemma *lam-eqE*: [| (*lam* x:A. $f(x)$) = (*lam* x:A. $g(x)$); a:A |] ==> $f(a)$=$g(a)$
⟨*proof*⟩

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

lemma *singleton-fun* [*simp*]: {<a,b>} : {a} -> {b}
⟨*proof*⟩

lemma *Pi-empty2* [*simp*]: (A->0) = (*if* A=0 *then* {0} *else* 0)
⟨*proof*⟩

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

8.4 Extensionality

lemma *fun-subset*:
[| f : $Pi(A,B)$; g: $Pi(C,D)$; A<=C;
!!x. x:A ==> $f`x = g`x$ |] ==> f<=g
⟨*proof*⟩

lemma *fun-extension*:
[| f : $Pi(A,B)$; g: $Pi(A,D)$;
!!x. x:A ==> $f`x = g`x$ |] ==> f=g
⟨*proof*⟩

lemma *eta* [*simp*]: f : $Pi(A,B)$ ==> (*lam* x:A. $f`x$) = f
⟨*proof*⟩

lemma *fun-extension-iff*:
[| f:$Pi(A,B)$; g:$Pi(A,C)$ |] ==> ($ALL\ a$:A. $f`a = g`a$) <-> f=g
⟨*proof*⟩

lemma *fun-subset-eq*: [| f:$Pi(A,B)$; g:$Pi(A,C)$ |] ==> f <= g <-> (f = g)
⟨*proof*⟩

lemma *Pi-lamE*:
assumes *major*: f: $Pi(A,B)$
and *minor*: !!b. [| $ALL\ x{:}A.\ b(x){:}B(x);\ f = (lam\ x{:}A.\ b(x))$ |] ==> P
shows P
⟨*proof*⟩

8.5 Images of Functions

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

lemma *Repfun-function-if*:
$function(f)$
==> $\{f`x.\ x{:}C\} = (if\ C <= domain(f)\ then\ f``C\ else\ cons(0,f``C))$
⟨*proof*⟩

lemma *image-function*:
[| $function(f)$; $C <= domain(f)$ |] ==> $f``C = \{f`x.\ x{:}C\}$
⟨*proof*⟩

lemma *image-fun*: [| $f : Pi(A,B)$; $C <= A$ |] ==> $f``C = \{f`x.\ x{:}C\}$
⟨*proof*⟩

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

lemma *Pi-image-cons*:
[| f: $Pi(A,B)$; x: A |] ==> $f\ ``\ cons(x,y) = cons(f`x,\ f``y)$
⟨*proof*⟩

8.6 Properties of *restrict*(f, A)

lemma *restrict-subset*: $restrict(f,A) <= f$
⟨*proof*⟩

lemma *function-restrictI*:
$function(f)$ ==> $function(restrict(f,A))$
⟨*proof*⟩

lemma *restrict-type2*: [| f: $Pi(C,B)$; $A<=C$ |] ==> $restrict(f,A) : Pi(A,B)$
⟨*proof*⟩

lemma *restrict*: $restrict(f,A)\ `\ a = (if\ a : A\ then\ f`a\ else\ 0)$
⟨*proof*⟩

lemma *restrict-empty* [*simp*]: $restrict(f,0) = 0$
⟨*proof*⟩

lemma *restrict-iff*: $z \in restrict(r,A) \longleftrightarrow z \in r \;\&\; (\exists\, x{\in}A.\ \exists\, y.\ z = \langle x,\ y\rangle)$
⟨*proof*⟩

lemma *restrict-restrict* [*simp*]:
$restrict(restrict(r,A),B) = restrict(r,\ A\ Int\ B)$
⟨*proof*⟩

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

lemma *restrict-idem*: $f <= Sigma(A,B) ==> restrict(f,A) = f$
⟨*proof*⟩

lemma *domain-restrict-idem*:
$[|\ domain(r) <= A;\ relation(r)\ |] ==> restrict(r,A) = r$
⟨*proof*⟩

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

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

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

lemma *fun-cons-restrict-eq*:
$f : cons(a,\ b) -> B ==> f = cons(<a,\ f\ `\ a>,\ restrict(f,\ b))$
⟨*proof*⟩

8.7 Unions of Functions

lemma *function-Union*:
$[|\ ALL\ x{:}S.\ function(x);$
$\quad ALL\ x{:}S.\ ALL\ y{:}S.\ x<=y\ |\ y<=x\ \ |]$
$==> function(Union(S))$
⟨*proof*⟩

lemma *fun-Union*:
$[|\ ALL\ f{:}S.\ EX\ C\ D.\ f{:}C{->}D;$
$\qquad ALL\ f{:}S.\ ALL\ y{:}S.\ f<=y\ |\ y<=f\ \ |] ==>$
$\quad Union(S) : domain(Union(S)) -> range(Union(S))$
⟨*proof*⟩

lemma *gen-relation-Union* [*rule-format*]:
$\forall f{\in}F.\ relation(f) \Longrightarrow relation(Union(F))$

⟨*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*−>*B*; *g*: *C*−>*D*; *A Int C* = *0* |]
==> (*f Un g*) : (*A Un C*) −> (*B Un D*)

⟨*proof*⟩

lemma *fun-disjoint-apply1*: *a* ∉ *domain*(*g*) ==> (*f Un g*)‘*a* = *f*‘*a*
⟨*proof*⟩

lemma *fun-disjoint-apply2*: *c* ∉ *domain*(*f*) ==> (*f Un g*)‘*c* = *g*‘*c*
⟨*proof*⟩

8.8 Domain and Range of a Function or Relation

lemma *domain-of-fun*: *f* : *Pi*(*A*,*B*) ==> *domain*(*f*)=*A*
⟨*proof*⟩

lemma *apply-rangeI*: [| *f* : *Pi*(*A*,*B*); *a*: *A* |] ==> *f*‘*a* : *range*(*f*)
⟨*proof*⟩

lemma *range-of-fun*: *f* : *Pi*(*A*,*B*) ==> *f* : *A*−>*range*(*f*)
⟨*proof*⟩

8.9 Extensions of Functions

lemma *fun-extend*:
[| *f*: *A*−>*B*; *c*~:*A* |] ==> *cons*(<*c*,*b*>,*f*) : *cons*(*c*,*A*) −> *cons*(*b*,*B*)
⟨*proof*⟩

lemma *fun-extend3*:
[| *f*: *A*−>*B*; *c*~:*A*; *b*: *B* |] ==> *cons*(<*c*,*b*>,*f*) : *cons*(*c*,*A*) −> *B*
⟨*proof*⟩

lemma *extend-apply*:
c ~: *domain*(*f*) ==> *cons*(<*c*,*b*>,*f*)‘*a* = (*if a*=*c then b else f*‘*a*)
⟨*proof*⟩

lemma *fun-extend-apply* [*simp*]:
[| *f*: *A*−>*B*; *c*~:*A* |] ==> *cons*(<*c*,*b*>,*f*)‘*a* = (*if a*=*c then b else f*‘*a*)
⟨*proof*⟩

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

lemma *cons-fun-eq*:
  c ~: A ==> *cons*(c,A) -> B = ($\bigcup f \in A$->B. $\bigcup b \in B$. {*cons*(<c,b>, f)})
⟨*proof*⟩

lemma *succ-fun-eq*: *succ*(n) -> B = ($\bigcup f \in n$->B. $\bigcup b \in B$. {*cons*(<n,b>, f)})
⟨*proof*⟩

8.10 Function Updates

definition
  update :: [i,i,i] => i **where**
   update(f,a,b) == *lam* x: *cons*(a, *domain*(f)). *if*(x=a, b, f‘x)

nonterminals
  updbinds *updbind*

syntax

```
  -updbind    :: [i, i] => updbind              ((2- :=/ -))
              :: updbind => updbinds             (-)
  -updbinds   :: [updbind, updbinds] => updbinds (-,/ -)
  -Update     :: [i, updbinds] => i              (-/'((-)') [900,0] 900)
```

translations
  -Update (f, *-updbinds*(b,bs)) == *-Update* (*-Update*(f,b), bs)
  f(x:=y) == *CONST update*(f,x,y)

lemma *update-apply* [*simp*]: f(x:=y) ‘ z = (*if* z=x *then* y *else* f‘z)
⟨*proof*⟩

lemma *update-idem*: [| f‘x = y; f: *Pi*(A,B); x: A |] ==> f(x:=y) = f
⟨*proof*⟩

declare *refl* [*THEN update-idem*, *simp*]

lemma *domain-update* [*simp*]: *domain*(f(x:=y)) = *cons*(x, *domain*(f))
⟨*proof*⟩

lemma *update-type*: [| f:*Pi*(A,B); x : A; y: B(x) |] ==> f(x:=y) : *Pi*(A, B)
⟨*proof*⟩

8.11 Monotonicity Theorems

8.11.1 Replacement in its Various Forms

lemma *Replace-mono*: $A<=B ==> Replace(A,P) <= Replace(B,P)$
⟨*proof*⟩

lemma *RepFun-mono*: $A<=B ==> \{f(x).\ x:A\} <= \{f(x).\ x:B\}$
⟨*proof*⟩

lemma *Pow-mono*: $A<=B ==> Pow(A) <= Pow(B)$
⟨*proof*⟩

lemma *Union-mono*: $A<=B ==> Union(A) <= Union(B)$
⟨*proof*⟩

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

lemma *Inter-anti-mono*: $[|\ A<=B;\ A\neq 0\ |] ==> Inter(B) <= Inter(A)$
⟨*proof*⟩

lemma *cons-mono*: $C<=D ==> cons(a,C) <= cons(a,D)$
⟨*proof*⟩

lemma *Un-mono*: $[|\ A<=C;\ B<=D\ |] ==> A\ Un\ B <= C\ Un\ D$
⟨*proof*⟩

lemma *Int-mono*: $[|\ A<=C;\ B<=D\ |] ==> A\ Int\ B <= C\ Int\ D$
⟨*proof*⟩

lemma *Diff-mono*: $[|\ A<=C;\ D<=B\ |] ==> A-B <= C-D$
⟨*proof*⟩

8.11.2 Standard Products, Sums and Function Spaces

lemma *Sigma-mono* [*rule-format*]:
$[|\ A<=C;\ !!x.\ x:A --> B(x) <= D(x)\ |] ==> Sigma(A,B) <= Sigma(C,D)$
⟨*proof*⟩

lemma *sum-mono*: $[|\ A<=C;\ B<=D\ |] ==> A+B <= C+D$
⟨*proof*⟩

lemma *Pi-mono*: $B<=C ==> A->B <= A->C$
⟨*proof*⟩

lemma *lam-mono*: $A<=B \Longrightarrow Lambda(A,c) <= Lambda(B,c)$
⟨*proof*⟩

8.11.3 Converse, Domain, Range, Field

lemma *converse-mono*: $r<=s \Longrightarrow converse(r) <= converse(s)$
⟨*proof*⟩

lemma *domain-mono*: $r<=s \Longrightarrow domain(r)<=domain(s)$
⟨*proof*⟩

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

lemma *range-mono*: $r<=s \Longrightarrow range(r)<=range(s)$
⟨*proof*⟩

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

lemma *field-mono*: $r<=s \Longrightarrow field(r)<=field(s)$
⟨*proof*⟩

lemma *field-rel-subset*: $r <= A*A \Longrightarrow field(r) <= A$
⟨*proof*⟩

8.11.4 Images

lemma *image-pair-mono*:
$[| \; !!\; x\; y.\; <x,y>:r \Longrightarrow <x,y>:s;\;\; A<=B \; |] \Longrightarrow r``A <= s``B$
⟨*proof*⟩

lemma *vimage-pair-mono*:
$[| \; !!\; x\; y.\; <x,y>:r \Longrightarrow <x,y>:s;\;\; A<=B \; |] \Longrightarrow r-``A <= s-``B$
⟨*proof*⟩

lemma *image-mono*: $[| \; r<=s;\;\; A<=B \; |] \Longrightarrow r``A <= s``B$
⟨*proof*⟩

lemma *vimage-mono*: $[| \; r<=s;\;\; A<=B \; |] \Longrightarrow r-``A <= s-``B$
⟨*proof*⟩

lemma *Collect-mono*:
$[| \; A<=B;\;\; !!x.\; x:A \Longrightarrow P(x) \dashrightarrow Q(x) \; |] \Longrightarrow Collect(A,P) <= Collect(B,Q)$
⟨*proof*⟩

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

lemma *bex-image-simp*:
[| $f : Pi(X,\ Y)$; $A \subseteq X$ |] ==> $(EX\ x : f``A.\ P(x)) <-> (EX\ x{:}A.\ P(f`x))$
⟨*proof*⟩

lemma *ball-image-simp*:
[| $f : Pi(X,\ Y)$; $A \subseteq X$ |] ==> $(ALL\ x : f``A.\ P(x)) <-> (ALL\ x{:}A.\ P(f`x))$
⟨*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 :: [*i*, *i*] => *i* (<(-;/ -)>) **where**
<*a*;*b*> == *a*+*b*

definition
qfst :: *i* => *i* **where**
qfst(*p*) == *THE a. EX b. p*=<*a*;*b*>

definition
qsnd :: *i* => *i* **where**
qsnd(*p*) == *THE b. EX a. p*=<*a*;*b*>

definition
qsplit :: [[*i*, *i*] => *'a*, *i*] => *'a*::{} **where**
qsplit(*c*,*p*) == *c*(*qfst*(*p*), *qsnd*(*p*))

definition
qconverse :: *i* => *i* **where**
qconverse(*r*) == {*z*. *w*:*r*, *EX x y*. *w*=<*x*;*y*> & *z*=<*y*;*x*>}

definition
QSigma :: [*i*, *i* => *i*] => *i* **where**
QSigma(*A*,*B*) == ⋃*x*∈*A*. ⋃*y*∈*B*(*x*). {<*x*;*y*>}

syntax
-QSUM :: [*idt*, *i*, *i*] => *i* ((*3QSUM* -:-./ -) *10*)
translations

QSUM x:A. B => CONST QSigma(A, %x. B)

abbreviation
qprod (**infixr** <*> *80*) **where**
A <> B == QSigma(A, %-. B)*

definition
qsum :: [*i*,*i*]=>*i* (**infixr** <+> *65*) **where**
A <+> B == ({*0*} <*> *A*) *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*: $<a;b> : QSigma(A,B) ==> a : A$
⟨*proof*⟩

lemma *QSigmaD2*: $<a;b> : QSigma(A,B) ==> b : B(a)$
⟨*proof*⟩

lemma *QSigma-cong*:
$[| A=A';\ !!x.\ x:A' ==> B(x)=B'(x) |] ==>$
$QSigma(A,B) = QSigma(A',B')$
⟨*proof*⟩

lemma *QSigma-empty1* [*simp*]: $QSigma(0,B) = 0$
⟨*proof*⟩

lemma *QSigma-empty2* [*simp*]: $A <*> 0 = 0$
⟨*proof*⟩

9.1.2 Projections: qfst, qsnd

lemma *qfst-conv* [*simp*]: $qfst(<a;b>) = a$
⟨*proof*⟩

lemma *qsnd-conv* [*simp*]: $qsnd(<a;b>) = b$
⟨*proof*⟩

lemma *qfst-type* [*TC*]: $p:QSigma(A,B) ==> qfst(p) : A$
⟨*proof*⟩

lemma *qsnd-type* [*TC*]: $p:QSigma(A,B) ==> qsnd(p) : B(qfst(p))$
⟨*proof*⟩

lemma *QPair-qfst-qsnd-eq*: $a: QSigma(A,B) ==> <qfst(a);\ qsnd(a)> = a$
⟨*proof*⟩

9.1.3 Eliminator: qsplit

lemma *qsplit* [*simp*]: $qsplit(\%x\ y.\ c(x,y), <a;b>) == c(a,b)$
⟨*proof*⟩

lemma *qsplit-type* [*elim*!]:
$[|\ \ p:QSigma(A,B);$
$\quad !!x\ y.[|\ x:A;\ y:B(x)\ |] ==> c(x,y):C(<x;y>)$

|] ==> *qsplit*(%*x y*. *c*(*x*,*y*), *p*) : *C*(*p*)
⟨*proof*⟩

lemma *expand-qsplit*:
u: *A*<*>*B* ==> *R*(*qsplit*(*c*,*u*)) <-> (*ALL x*:*A*. *ALL y*:*B*. *u* = <*x*;*y*> --> *R*(*c*(*x*,*y*)))
⟨*proof*⟩

9.1.4 qsplit for predicates: result type o

lemma *qsplitI*: *R*(*a*,*b*) ==> *qsplit*(*R*, <*a*;*b*>)
⟨*proof*⟩

lemma *qsplitE*:
[| *qsplit*(*R*,*z*); *z*:*QSigma*(*A*,*B*);
!!*x y*. [| *z* = <*x*;*y*>; *R*(*x*,*y*) |] ==> *P*
|] ==> *P*
⟨*proof*⟩

lemma *qsplitD*: *qsplit*(*R*,<*a*;*b*>) ==> *R*(*a*,*b*)
⟨*proof*⟩

9.1.5 qconverse

lemma *qconverseI* [*intro*!]: <*a*;*b*>:*r* ==> <*b*;*a*>:*qconverse*(*r*)
⟨*proof*⟩

lemma *qconverseD* [*elim*!]: <*a*;*b*> : *qconverse*(*r*) ==> <*b*;*a*> : *r*
⟨*proof*⟩

lemma *qconverseE* [*elim*!]:
[| *yx* : *qconverse*(*r*);
!!*x y*. [| *yx*=<*y*;*x*>; <*x*;*y*>:*r* |] ==> *P*
|] ==> *P*
⟨*proof*⟩

lemma *qconverse-qconverse*: *r*<=*QSigma*(*A*,*B*) ==> *qconverse*(*qconverse*(*r*)) = *r*
⟨*proof*⟩

lemma *qconverse-type*: *r* <= *A* <*> *B* ==> *qconverse*(*r*) <= *B* <*> *A*
⟨*proof*⟩

lemma *qconverse-prod*: *qconverse*(*A* <*> *B*) = *B* <*> *A*
⟨*proof*⟩

lemma *qconverse-empty*: *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 ==> QInl(a) : A <+> B*
⟨*proof*⟩

lemma *QInrI* [*intro*!]: *b : B ==> QInr(b) : A <+> B*
⟨*proof*⟩

lemma *qsumE* [*elim*!]:
[| u: A <+> B;
!!x. [| x:A; u=QInl(x) |] ==> P;
!!y. [| y:B; u=QInr(y) |] ==> P
|] ==> 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 ==> a*: *A*
⟨*proof*⟩

lemma *QInrD*: *QInr(b)*: *A<+>B ==> b*: *B*

⟨*proof*⟩

lemma *qsum-iff*:
u: A <+> B <-> (EX x. x:A & u=QInl(x)) | (EX y. y:B & u=QInr(y))
⟨*proof*⟩

lemma *qsum-subset-iff*: A <+> B <= C <+> D <-> A<=C & B<=D
⟨*proof*⟩

lemma *qsum-equal-iff*: A <+> B = C <+> D <-> 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*:
[| u: A <+> B;
!!x. x: A ==> c(x): C(QInl(x));
!!y. y: B ==> d(y): C(QInr(y))
|] ==> qcase(c,d,u) : C(u)
⟨*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 <= A <+> B ==> Part(C,QInl) Un Part(C,QInr)
= C
⟨*proof*⟩

9.2.2 Monotonicity

lemma *QPair-mono*: [| a<=c; b<=d |] ==> <a;b> <= <c;d>
⟨*proof*⟩

lemma *QSigma-mono* [*rule-format*]:
[| $A<=C$; ALL x:A. $B(x) <= D(x)$ |] ==> $QSigma(A,B) <= QSigma(C,D)$
⟨*proof*⟩

lemma *QInl-mono*: $a<=b$ ==> $QInl(a) <= QInl(b)$
⟨*proof*⟩

lemma *QInr-mono*: $a<=b$ ==> $QInr(a) <= QInr(b)$
⟨*proof*⟩

lemma *qsum-mono*: [| $A<=C$; $B<=D$ |] ==> $A <+> B <= C <+> D$
⟨*proof*⟩

end

10 Perm: Injections, Surjections, Bijections, Composition

theory *Perm* **imports** *func* **begin**

definition

comp :: [*i*,*i*]=>*i* (**infixr** *O* 60) **where**
$r\ O\ s == \{xz : domain(s)*range(r)\ .$
$EX\ x\ y\ z.\ xz=<x,z> \&\ <x,y>:s \&\ <y,z>:r\}$

definition

id :: *i*=>*i* **where**
$id(A) == (lam\ x:A.\ x)$

definition

inj :: [*i*,*i*]=>*i* **where**
$inj(A,B) == \{\ f: A->B.\ ALL\ w:A.\ ALL\ x:A.\ f`w=f`x\ --> w=x\}$

definition

surj :: [*i*,*i*]=>*i* **where**
$surj(A,B) == \{\ f: A->B\ .\ ALL\ y:B.\ EX\ x:A.\ f`x=y\}$

definition

bij :: [*i*,*i*]=>*i* **where**
$bij(A,B) == inj(A,B)\ Int\ surj(A,B)$

10.1 Surjections

lemma *surj-is-fun*: f: $surj(A,B)$ ==> f: A->B
⟨*proof*⟩

lemma *fun-is-surj*: f : $Pi(A,B)$ ==> f: $surj(A,range(f))$
⟨*proof*⟩

lemma *surj-range*: f: $surj(A,B)$ ==> $range(f)$=B
⟨*proof*⟩

lemma *f-imp-surjective*:
[| f: A->B; !!y. y:B ==> $d(y)$: A; !!y. y:B ==> f`$d(y)$ = y |]
==> f: $surj(A,B)$
⟨*proof*⟩

lemma *lam-surjective*:
[| !!x. x:A ==> $c(x)$: B;
!!y. y:B ==> $d(y)$: A;
!!y. y:B ==> $c(d(y))$ = y
|] ==> (*lam* x:A. $c(x)$) : $surj(A,B)$
⟨*proof*⟩

lemma *cantor-surj*: f ~: $surj(A,Pow(A))$
⟨*proof*⟩

10.2 Injections

lemma *inj-is-fun*: f: $inj(A,B)$ ==> f: A->B
⟨*proof*⟩

lemma *inj-equality*:
[| <a,b>:f; <c,b>:f; f: $inj(A,B)$ |] ==> a=c
⟨*proof*⟩

lemma *inj-apply-equality*: [| f:$inj(A,B)$; f`$a$=$f$`b; a:A; b:A |] ==> a=b
⟨*proof*⟩

lemma *f-imp-injective*: [| f: A->B; *ALL* x:A. $d(f$`$x)$=x |] ==> f: $inj(A,B)$
⟨*proof*⟩

lemma *lam-injective*:
[| !!x. x:A ==> $c(x)$: B;

!!x. x:A ==> $d(c(x)) = x$ |]
==> (*lam* x:A. $c(x)$) : *inj*(A,B)
⟨*proof*⟩

10.3 Bijections

lemma *bij-is-inj*: f: *bij*(A,B) ==> f: *inj*(A,B)
⟨*proof*⟩

lemma *bij-is-surj*: f: *bij*(A,B) ==> f: *surj*(A,B)
⟨*proof*⟩

lemmas *bij-is-fun* = *bij-is-inj* [*THEN inj-is-fun*, *standard*]

lemma *lam-bijective*:
[| !!x. x:A ==> $c(x)$: B;
!!y. y:B ==> $d(y)$: A;
!!x. x:A ==> $d(c(x)) = x$;
!!y. y:B ==> $c(d(y)) = y$
|] ==> (*lam* x:A. $c(x)$) : *bij*(A,B)
⟨*proof*⟩

lemma *RepFun-bijective*: (*ALL* y : x. *EX*! y'. $f(y') = f(y)$)
==> (*lam* z:{$f(y)$. y:x}. *THE* y. $f(y) = z$) : *bij*({$f(y)$. y:x}, x)
⟨*proof*⟩

10.4 Identity Function

lemma *idI* [*intro*!]: a:A ==> <a,a> : *id*(A)
⟨*proof*⟩

lemma *idE* [*elim*!]: [| p: *id*(A); !!x.[| x:A; p=<x,x> |] ==> P |] ==> P
⟨*proof*⟩

lemma *id-type*: *id*(A) : A->A
⟨*proof*⟩

lemma *id-conv* [*simp*]: x:A ==> *id*(A)‘x = x
⟨*proof*⟩

lemma *id-mono*: A<=B ==> *id*(A) <= *id*(B)
⟨*proof*⟩

lemma *id-subset-inj*: A<=B ==> *id*(A): *inj*(A,B)
⟨*proof*⟩

lemmas *id-inj* = *subset-refl* [*THEN id-subset-inj*, *standard*]

lemma *id-surj*: *id*(A): *surj*(A,A)

⟨*proof*⟩

lemma *id-bij*: *id*(*A*): *bij*(*A*,*A*)
⟨*proof*⟩

lemma *subset-iff-id*: *A* <= *B* <−> *id*(*A*) : *A*−>*B*
⟨*proof*⟩

id as the identity relation

lemma *id-iff* [*simp*]: <*x*,*y*> ∈ *id*(*A*) <−> *x*=*y* & *y* ∈ *A*
⟨*proof*⟩

10.5 Converse of a Function

lemma *inj-converse-fun*: *f*: *inj*(*A*,*B*) ==> *converse*(*f*) : *range*(*f*)−>*A*
⟨*proof*⟩

The premises are equivalent to saying that f is injective...

lemma *left-inverse-lemma*:
[| *f*: *A*−>*B*; *converse*(*f*): *C*−>*A*; *a*: *A* |] ==> *converse*(*f*)‘(*f*‘*a*) = *a*
⟨*proof*⟩

lemma *left-inverse* [*simp*]: [| *f*: *inj*(*A*,*B*); *a*: *A* |] ==> *converse*(*f*)‘(*f*‘*a*) = *a*
⟨*proof*⟩

lemma *left-inverse-eq*:
[|*f* ∈ *inj*(*A*,*B*); *f* ‘ *x* = *y*; *x* ∈ *A*|] ==> *converse*(*f*) ‘ *y* = *x*
⟨*proof*⟩

lemmas *left-inverse-bij* = *bij-is-inj* [*THEN left-inverse*, *standard*]

lemma *right-inverse-lemma*:
[| *f*: *A*−>*B*; *converse*(*f*): *C*−>*A*; *b*: *C* |] ==> *f*‘(*converse*(*f*)‘*b*) = *b*
⟨*proof*⟩

lemma *right-inverse* [*simp*]:
[| *f*: *inj*(*A*,*B*); *b*: *range*(*f*) |] ==> *f*‘(*converse*(*f*)‘*b*) = *b*
⟨*proof*⟩

lemma *right-inverse-bij*: [| *f*: *bij*(*A*,*B*); *b*: *B* |] ==> *f*‘(*converse*(*f*)‘*b*) = *b*
⟨*proof*⟩

10.6 Converses of Injections, Surjections, Bijections

lemma *inj-converse-inj*: *f*: *inj*(*A*,*B*) ==> *converse*(*f*): *inj*(*range*(*f*), *A*)
⟨*proof*⟩

lemma *inj-converse-surj*: *f*: *inj*(*A*,*B*) ==> *converse*(*f*): *surj*(*range*(*f*), *A*)

⟨*proof*⟩

lemma *bij-converse-bij* [*TC*]: *f*: *bij*(*A*,*B*) ==> *converse*(*f*): *bij*(*B*,*A*)
⟨*proof*⟩

10.7 Composition of Two Relations

lemma *compI* [*intro*]: [| <*a*,*b*>:*s*; <*b*,*c*>:*r* |] ==> <*a*,*c*> : *r* *O* *s*
⟨*proof*⟩

lemma *compE* [*elim*!]:
[| *xz* : *r* *O* *s*;
!!*x* *y* *z*. [| *xz*=<*x*,*z*>; <*x*,*y*>:*s*; <*y*,*z*>:*r* |] ==> *P* |]
==> *P*
⟨*proof*⟩

lemma *compEpair*:
[| <*a*,*c*> : *r* *O* *s*;
!!*y*. [| <*a*,*y*>:*s*; <*y*,*c*>:*r* |] ==> *P* |]
==> *P*
⟨*proof*⟩

lemma *converse-comp*: *converse*(*R* *O* *S*) = *converse*(*S*) *O* *converse*(*R*)
⟨*proof*⟩

10.8 Domain and Range – see Suppes, Section 3.1

lemma *range-comp*: *range*(*r* *O* *s*) <= *range*(*r*)
⟨*proof*⟩

lemma *range-comp-eq*: *domain*(*r*) <= *range*(*s*) ==> *range*(*r* *O* *s*) = *range*(*r*)
⟨*proof*⟩

lemma *domain-comp*: *domain*(*r* *O* *s*) <= *domain*(*s*)
⟨*proof*⟩

lemma *domain-comp-eq*: *range*(*s*) <= *domain*(*r*) ==> *domain*(*r* *O* *s*) = *domain*(*s*)
⟨*proof*⟩

lemma *image-comp*: (*r* *O* *s*)‘‘*A* = *r*‘‘(*s*‘‘*A*)
⟨*proof*⟩

10.9 Other Results

lemma *comp-mono*: [| *r*'<=*r*; *s*'<=*s* |] ==> (*r*' *O* *s*') <= (*r* *O* *s*)
⟨*proof*⟩

lemma *comp-rel*: [| $s<=A*B$; $r<=B*C$ |] ==> $(r\ O\ s) <= A*C$
⟨*proof*⟩

lemma *comp-assoc*: $(r\ O\ s)\ O\ t = r\ O\ (s\ O\ t)$
⟨*proof*⟩

lemma *left-comp-id*: $r<=A*B$ ==> $id(B)\ O\ r = r$
⟨*proof*⟩

lemma *right-comp-id*: $r<=A*B$ ==> $r\ O\ id(A) = r$
⟨*proof*⟩

10.10 Composition Preserves Functions, Injections, and Surjections

lemma *comp-function*: [| *function*(g); *function*(f) |] ==> *function*$(f\ O\ g)$
⟨*proof*⟩

lemma *comp-fun*: [| g: $A->B$; f: $B->C$ |] ==> $(f\ O\ g) : A->C$
⟨*proof*⟩

lemma *comp-fun-apply* [*simp*]:
    [| g: $A->B$; a:A |] ==> $(f\ O\ g)`a = f`(g`a)$
⟨*proof*⟩

lemma *comp-lam*:
    [| $!!x.\ x$:A ==> $b(x)$: B |]
    ==> $(lam\ y$:$B.\ c(y))\ O\ (lam\ x$:$A.\ b(x)) = (lam\ x$:$A.\ c(b(x)))$
⟨*proof*⟩

lemma *comp-inj*:
    [| g: *inj*(A,B); f: *inj*(B,C) |] ==> $(f\ O\ g) :$ *inj*(A,C)
⟨*proof*⟩

lemma *comp-surj*:
    [| g: *surj*(A,B); f: *surj*(B,C) |] ==> $(f\ O\ g) :$ *surj*(A,C)
⟨*proof*⟩

lemma *comp-bij*:
    [| g: *bij*(A,B); f: *bij*(B,C) |] ==> $(f\ O\ g) :$ *bij*(A,C)
⟨*proof*⟩

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*:
[| (*f O g*): *inj*(*A*,*C*); *g*: *A*−>*B*; *f*: *B*−>*C* |] ==> *g*: *inj*(*A*,*B*)
⟨*proof*⟩

lemma *comp-mem-injD2*:
[| (*f O g*): *inj*(*A*,*C*); *g*: *surj*(*A*,*B*); *f*: *B*−>*C* |] ==> *f*: *inj*(*B*,*C*)
⟨*proof*⟩

lemma *comp-mem-surjD1*:
[| (*f O g*): *surj*(*A*,*C*); *g*: *A*−>*B*; *f*: *B*−>*C* |] ==> *f*: *surj*(*B*,*C*)
⟨*proof*⟩

lemma *comp-mem-surjD2*:
[| (*f O g*): *surj*(*A*,*C*); *g*: *A*−>*B*; *f*: *inj*(*B*,*C*) |] ==> *g*: *surj*(*A*,*B*)
⟨*proof*⟩

10.11.1 Inverses of Composition

lemma *left-comp-inverse*: *f*: *inj*(*A*,*B*) ==> *converse*(*f*) *O f* = *id*(*A*)
⟨*proof*⟩

lemma *right-comp-inverse*:
f: *surj*(*A*,*B*) ==> *f O converse*(*f*) = *id*(*B*)
⟨*proof*⟩

10.11.2 Proving that a Function is a Bijection

lemma *comp-eq-id-iff*:
[| *f*: *A*−>*B*; *g*: *B*−>*A* |] ==> *f O g* = *id*(*B*) <−> (*ALL y*:*B*. *f*‘(*g*‘*y*)=*y*)
⟨*proof*⟩

lemma *fg-imp-bijective*:
[| *f*: *A*−>*B*; *g*: *B*−>*A*; *f O g* = *id*(*B*); *g O f* = *id*(*A*) |] ==> *f* : *bij*(*A*,*B*)
⟨*proof*⟩

lemma *nilpotent-imp-bijective*: [| *f*: *A*−>*A*; *f O f* = *id*(*A*) |] ==> *f* : *bij*(*A*,*A*)
⟨*proof*⟩

lemma *invertible-imp-bijective*:
[| *converse*(*f*): *B*−>*A*; *f*: *A*−>*B* |] ==> *f* : *bij*(*A*,*B*)
⟨*proof*⟩

10.11.3 Unions of Functions

See similar theorems in func.thy

lemma *inj-disjoint-Un*:
[| *f*: *inj*(*A*,*B*); *g*: *inj*(*C*,*D*); *B Int D* = *0* |]
==> (*lam a*: *A Un C*. *if a*:*A then f*‘*a else g*‘*a*) : *inj*(*A Un C*, *B Un D*)
⟨*proof*⟩

lemma *surj-disjoint-Un*:
[| *f*: *surj*(*A*,*B*); *g*: *surj*(*C*,*D*); *A Int C* = *0* |]
==> (*f Un g*) : *surj*(*A Un C*, *B Un D*)
⟨*proof*⟩

lemma *bij-disjoint-Un*:
[| *f*: *bij*(*A*,*B*); *g*: *bij*(*C*,*D*); *A Int C* = *0*; *B Int D* = *0* |]
==> (*f Un g*) : *bij*(*A Un C*, *B Un D*)
⟨*proof*⟩

10.11.4 Restrictions as Surjections and Bijections

lemma *surj-image*:
f: *Pi*(*A*,*B*) ==> *f*: *surj*(*A*, *f*‘‘*A*)
⟨*proof*⟩

lemma *restrict-image* [*simp*]: *restrict*(*f*,*A*) ‘‘ *B* = *f* ‘‘ (*A Int B*)
⟨*proof*⟩

lemma *restrict-inj*:
[| *f*: *inj*(*A*,*B*); *C*<=*A* |] ==> *restrict*(*f*,*C*): *inj*(*C*,*B*)
⟨*proof*⟩

lemma *restrict-surj*: [| *f*: *Pi*(*A*,*B*); *C*<=*A* |] ==> *restrict*(*f*,*C*): *surj*(*C*, *f*‘‘*C*)
⟨*proof*⟩

lemma *restrict-bij*:
[| *f*: *inj*(*A*,*B*); *C*<=*A* |] ==> *restrict*(*f*,*C*): *bij*(*C*, *f*‘‘*C*)
⟨*proof*⟩

10.11.5 Lemmas for Ramsey's Theorem

lemma *inj-weaken-type*: [| *f*: *inj*(*A*,*B*); *B*<=*D* |] ==> *f*: *inj*(*A*,*D*)
⟨*proof*⟩

lemma *inj-succ-restrict*:
[| *f*: *inj*(*succ*(*m*), *A*) |] ==> *restrict*(*f*,*m*) : *inj*(*m*, *A*−{*f*‘*m*})
⟨*proof*⟩

lemma *inj-extend*:
[| f: $inj(A,B)$; $a^{\sim}$:A; $b^{\sim}$:B |]
==> $cons(<a,b>,f)$: $inj(cons(a,A), cons(b,B))$
⟨*proof*⟩

end

11 Trancl: Relations: Their General Properties and Transitive Closure

theory *Trancl* **imports** *Fixedpt Perm* **begin**

definition
refl :: $[i,i]=>o$ **where**
$refl(A,r) == (ALL\ x{:}\ A.\ <x,x> : r)$

definition
irrefl :: $[i,i]=>o$ **where**
$irrefl(A,r) == ALL\ x{:}\ A.\ <x,x>\ ^{\sim}{:}\ r$

definition
sym :: $i=>o$ **where**
$sym(r) == ALL\ x\ y.\ <x,y>{:}\ r\ --> <y,x>{:}\ r$

definition
asym :: $i=>o$ **where**
$asym(r) == ALL\ x\ y.\ <x,y>{:}r\ --> {}^{\sim}\ <y,x>{:}r$

definition
antisym :: $i=>o$ **where**
$antisym(r) == ALL\ x\ y.<x,y>{:}r\ --> <y,x>{:}r\ --> x=y$

definition
trans :: $i=>o$ **where**
$trans(r) == ALL\ x\ y\ z.\ <x,y>{:}\ r\ --> <y,z>{:}\ r\ --> <x,z>{:}\ r$

definition
trans-on :: $[i,i]=>o$ (*trans*[-]'(-')) **where**
$trans[A](r) == ALL\ x{:}A.\ ALL\ y{:}A.\ ALL\ z{:}A.$
$<x,y>{:}\ r\ --> <y,z>{:}\ r\ --> <x,z>{:}\ r$

definition
rtrancl :: $i=>i$ ((-^*) [100] 100) **where**
$r^* == lfp(field(r)*field(r),\ \%s.\ id(field(r))\ Un\ (r\ O\ s))$

definition
trancl :: $i=>i$ ((-^+) [100] 100) **where**

$r\hat{}+ == r\ O\ r\hat{}*$

definition
equiv :: $[i,i]=>o$ **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*:
$[|\ !!x.\ x{:}A ==> <x,x> \sim{:}\ r\ |] ==> irrefl(A,r)$
⟨*proof*⟩

lemma *irreflE*: $[|\ irrefl(A,r);\ \ x{:}A\ |] ==>\ \ <x,x> \sim{:}\ r$
⟨*proof*⟩

11.1.2 symmetry

lemma *symI*:
$[|\ !!x\ y.<x,y>{:}\ r ==> <y,x>{:}\ r\ |] ==> sym(r)$
⟨*proof*⟩

lemma *symE*: $[|\ sym(r); <x,y>{:}\ r\ |]\ ==>\ \ <y,x>{:}\ r$
⟨*proof*⟩

11.1.3 antisymmetry

lemma *antisymI*:
$[|\ !!x\ y.[|\ <x,y>{:}\ r;\ \ <y,x>{:}\ r\ |] ==> x=y\ |] ==> antisym(r)$
⟨*proof*⟩

lemma *antisymE*: $[|\ antisym(r); <x,y>{:}\ r;\ \ <y,x>{:}\ r\ |]\ ==>\ \ x=y$
⟨*proof*⟩

11.1.4 transitivity

lemma *transD*: $[|\ trans(r);\ \ <a,b>{:}r;\ \ <b,c>{:}r\ |] ==> <a,c>{:}r$
⟨*proof*⟩

lemma *trans-onD*:
$[|\ trans[A](r);\ \ <a,b>{:}r;\ \ <b,c>{:}r;\ \ a{:}A;\ \ b{:}A;\ \ c{:}A\ |] ==> <a,c>{:}r$
⟨*proof*⟩

lemma *trans-imp-trans-on*: $trans(r) ==> trans[A](r)$
⟨*proof*⟩

lemma *trans-on-imp-trans*: $[|trans[A](r);\ r <= A*A|] ==> trans(r)$
⟨*proof*⟩

11.2 Transitive closure of a relation

lemma *rtrancl-bnd-mono*:
    bnd-mono(*field*(*r*)∗*field*(*r*), %*s*. *id*(*field*(*r*)) *Un* (*r O s*))
⟨*proof*⟩

lemma *rtrancl-mono*: *r*<=*s* ==> *r*^∗ <= *s*^∗
⟨*proof*⟩

lemmas *rtrancl-unfold* =
    rtrancl-bnd-mono [*THEN rtrancl-def* [*THEN def-lfp-unfold*], *standard*]

lemmas *rtrancl-type* = *rtrancl-def* [*THEN def-lfp-subset*, *standard*]

lemma *relation-rtrancl*: *relation*(*r*^∗)
⟨*proof*⟩

lemma *rtrancl-refl*: [| *a*: *field*(*r*) |] ==> <*a*,*a*> : *r*^∗
⟨*proof*⟩

lemma *rtrancl-into-rtrancl*: [| <*a*,*b*> : *r*^∗; <*b*,*c*> : *r* |] ==> <*a*,*c*> : *r*^∗
⟨*proof*⟩

lemma *r-into-rtrancl*: <*a*,*b*> : *r* ==> <*a*,*b*> : *r*^∗
⟨*proof*⟩

lemma *r-subset-rtrancl*: *relation*(*r*) ==> *r* <= *r*^∗
⟨*proof*⟩

lemma *rtrancl-field*: *field*(*r*^∗) = *field*(*r*)
⟨*proof*⟩

lemma *rtrancl-full-induct* [*case-names initial step*, *consumes 1*]:
  [| <*a*,*b*> : *r*^∗;
      !!*x*. *x*: *field*(*r*) ==> *P*(<*x*,*x*>);
      !!*x y z*.[| *P*(<*x*,*y*>); <*x*,*y*>: *r*^∗; <*y*,*z*>: *r* |] ==> *P*(<*x*,*z*>) |]
   ==> *P*(<*a*,*b*>)
⟨*proof*⟩

lemma *rtrancl-induct* [*case-names initial step*, *induct set*: *rtrancl*]:
[| $<a,b> : r\hat{}*$;
$P(a)$;
!!$y\ z$.[| $<a,y> : r\hat{}*$; $<y,z> : r$; $P(y)$ |] ==> $P(z)$
|] ==> $P(b)$

⟨*proof*⟩

lemma *trans-rtrancl*: *trans*($r\hat{}*$)
⟨*proof*⟩

lemmas *rtrancl-trans* = *trans-rtrancl* [*THEN transD*, *standard*]

lemma *rtranclE*:
[| $<a,b> : r\hat{}*$; $(a=b)$ ==> P;
!!y.[| $<a,y> : r\hat{}*$; $<y,b> : r$ |] ==> P |]
==> P
⟨*proof*⟩

lemma *trans-trancl*: *trans*($r\hat{}+$)
⟨*proof*⟩

lemmas *trans-on-trancl* = *trans-trancl* [*THEN trans-imp-trans-on*]

lemmas *trancl-trans* = *trans-trancl* [*THEN transD*, *standard*]

lemma *trancl-into-rtrancl*: $<a,b> : r\hat{}+$ ==> $<a,b> : r\hat{}*$
⟨*proof*⟩

lemma *r-into-trancl*: $<a,b> : r$ ==> $<a,b> : r\hat{}+$
⟨*proof*⟩

lemma *r-subset-trancl*: *relation*(r) ==> $r <= r\hat{}+$
⟨*proof*⟩

lemma *rtrancl-into-trancl1*: [| $<a,b> : r\hat{}*$; $<b,c> : r$ |] ==> $<a,c> : r\hat{}+$

⟨*proof*⟩

lemma *rtrancl-into-trancl2*:
[| <*a*,*b*> : *r*; <*b*,*c*> : *r* ^* |] ==> <*a*,*c*> : *r* ^+
⟨*proof*⟩

lemma *trancl-induct* [*case-names initial step*, *induct set*: *trancl*]:
[| <*a*,*b*> : *r* ^+;
!!*y*. [| <*a*,*y*> : *r* |] ==> *P*(*y*);
!!*y z*.[| <*a*,*y*> : *r* ^+; <*y*,*z*> : *r*; *P*(*y*) |] ==> *P*(*z*)
|] ==> *P*(*b*)
⟨*proof*⟩

lemma *tranclE*:
[| <*a*,*b*> : *r* ^+;
<*a*,*b*> : *r* ==> *P*;
!!*y*.[| <*a*,*y*> : *r* ^+; <*y*,*b*> : *r* |] ==> *P*
|] ==> *P*
⟨*proof*⟩

lemma *trancl-type*: *r* ^+ <= *field*(*r*)∗*field*(*r*)
⟨*proof*⟩

lemma *relation-trancl*: *relation*(*r* ^+)
⟨*proof*⟩

lemma *trancl-subset-times*: *r* ⊆ *A* ∗ *A* ==> *r* ^+ ⊆ *A* ∗ *A*
⟨*proof*⟩

lemma *trancl-mono*: *r*<=*s* ==> *r* ^+ <= *s* ^+
⟨*proof*⟩

lemma *trancl-eq-r*: [|*relation*(*r*); *trans*(*r*)|] ==> *r* ^+ = *r*
⟨*proof*⟩

lemma *rtrancl-idemp* [*simp*]: (*r* ^*) ^* = *r* ^*
⟨*proof*⟩

lemma *rtrancl-subset*: [| *R* <= *S*; *S* <= *R* ^* |] ==> *S* ^* = *R* ^*
⟨*proof*⟩

lemma *rtrancl-Un-rtrancl*:
[| *relation*(*r*); *relation*(*s*) |] ==> (*r* ^* *Un* *s* ^*) ^* = (*r* *Un* *s*) ^*

⟨*proof*⟩

lemma *rtrancl-converseD*: <*x*,*y*>:*converse*(*r*) ^* ==> <*x*,*y*>:*converse*(*r*^*)
⟨*proof*⟩

lemma *rtrancl-converseI*: <*x*,*y*>:*converse*(*r*^*) ==> <*x*,*y*>:*converse*(*r*) ^*
⟨*proof*⟩

lemma *rtrancl-converse*: *converse*(*r*) ^* = *converse*(*r*^*)
⟨*proof*⟩

lemma *trancl-converseD*: <*a*, *b*>:*converse*(*r*) ^+ ==> <*a*, *b*>:*converse*(*r*^+)
⟨*proof*⟩

lemma *trancl-converseI*: <*x*,*y*>:*converse*(*r*^+) ==> <*x*,*y*>:*converse*(*r*) ^+
⟨*proof*⟩

lemma *trancl-converse*: *converse*(*r*) ^+ = *converse*(*r*^+)
⟨*proof*⟩

lemma *converse-trancl-induct* [*case-names initial step*, *consumes 1*]:
[| <*a*, *b*>:*r*^+; !!*y*. <*y*, *b*> :*r* ==> *P*(*y*);
!!*y z*. [| <*y*, *z*> : *r*; <*z*, *b*> : *r*^+; *P*(*z*) |] ==> *P*(*y*) |]
==> *P*(*a*)
⟨*proof*⟩

end

12 WF: Well-Founded Recursion

theory *WF* **imports** *Trancl* **begin**

definition
wf :: *i*=>*o* **where**

wf(*r*) == *ALL Z*. *Z*=*0* | (*EX x*:*Z*. *ALL y*. <*y*,*x*>:*r* −−> ~ *y*:*Z*)

definition
wf-on :: [*i*,*i*]=>*o* (*wf*[-]'(-')) **where**

wf-on(*A*,*r*) == *wf*(*r Int A*∗*A*)

definition
is-recfun :: [*i*, *i*, [*i*,*i*]=>*i*, *i*] =>*o* **where**
is-recfun(*r*,*a*,*H*,*f*) == (*f* = (*lam x*: *r*−‘‘{*a*}. *H*(*x*, *restrict*(*f*, *r*−‘‘{*x*}))))

definition
the-recfun :: [*i*, *i*, [*i*,*i*]=>*i*] =>*i* **where**
the-recfun(*r*,*a*,*H*) == (*THE f*. *is-recfun*(*r*,*a*,*H*,*f*))

definition
wftrec :: [*i*, *i*, [*i*,*i*]=>*i*] =>*i* **where**
wftrec(*r*,*a*,*H*) == *H*(*a*, *the-recfun*(*r*,*a*,*H*))

definition
wfrec :: [*i*, *i*, [*i*,*i*]=>*i*] =>*i* **where**

wfrec(*r*,*a*,*H*) == *wftrec*(*r*^+, *a*, %*x f*. *H*(*x*, *restrict*(*f*,*r*−‘‘{*x*})))

definition
wfrec-on :: [*i*, *i*, *i*, [*i*,*i*]=>*i*] =>*i* (*wfrec*[-]‘(-,-,-‘)) **where**
wfrec[*A*](*r*,*a*,*H*) == *wfrec*(*r Int A*∗*A*, *a*, *H*)

12.1 Well-Founded Relations

12.1.1 Equivalences between *wf* and *wf-on*

lemma *wf-imp-wf-on*: *wf*(*r*) ==> *wf*[*A*](*r*)
⟨*proof*⟩

lemma *wf-on-imp-wf*: [|*wf*[*A*](*r*); *r* <= *A*∗*A*|] ==> *wf*(*r*)
⟨*proof*⟩

lemma *wf-on-field-imp-wf*: *wf*[*field*(*r*)](*r*) ==> *wf*(*r*)
⟨*proof*⟩

lemma *wf-iff-wf-on-field*: *wf*(*r*) <−> *wf*[*field*(*r*)](*r*)
⟨*proof*⟩

lemma *wf-on-subset-A*: [| *wf*[*A*](*r*); *B*<=*A* |] ==> *wf*[*B*](*r*)
⟨*proof*⟩

lemma *wf-on-subset-r*: [| *wf*[*A*](*r*); *s*<=*r* |] ==> *wf*[*A*](*s*)
⟨*proof*⟩

lemma *wf-subset*: [|*wf*(*s*); *r*<=*s*|] ==> *wf*(*r*)
⟨*proof*⟩

12.1.2 Introduction Rules for *wf-on*

If every non-empty subset of A has an r-minimal element then we have $wf[A](r)$.

lemma *wf-onI*:
assumes *prem*: !!*Z u*. [| *Z*<=*A*; *u*:*Z*; *ALL x*:*Z*. *EX y*:*Z*. <*y*,*x*>:*r* |] ==> *False*
shows *wf*[*A*](*r*)
⟨*proof*⟩

If r allows well-founded induction over A then we have $wf[A](r)$. Premise is equivalent to $\bigwedge B.\ \forall x{\in}A.\ (\forall y.\ \langle y,\ x\rangle \in r \longrightarrow y \in B) \longrightarrow x \in B \Longrightarrow A \subseteq B$

lemma *wf-onI2*:
assumes *prem*: !!*y B*. [| *ALL x*:*A*. (*ALL y*:*A*. <*y*,*x*>:*r* --> *y*:*B*) --> *x*:*B*; *y*:*A* |]
==> *y*:*B*
shows *wf*[*A*](*r*)
⟨*proof*⟩

12.1.3 Well-founded Induction

Consider the least z in $domain(r)$ such that $P(z)$ does not hold...

lemma *wf-induct* [*induct set*: *wf*]:
[| *wf*(*r*);
!!*x*.[| *ALL y*. <*y*,*x*>: *r* --> *P*(*y*) |] ==> *P*(*x*) |]
==> *P*(*a*)
⟨*proof*⟩

lemmas *wf-induct-rule* = *wf-induct* [*rule-format*, *induct set*: *wf*]

The form of this rule is designed to match *wfI*

lemma *wf-induct2*:
[| *wf*(*r*); *a*:*A*; *field*(*r*)<=*A*;
!!*x*.[| *x*: *A*; *ALL y*. <*y*,*x*>: *r* --> *P*(*y*) |] ==> *P*(*x*) |]
==> *P*(*a*)
⟨*proof*⟩

lemma *field-Int-square*: *field*(*r Int A*∗*A*) <= *A*
⟨*proof*⟩

lemma *wf-on-induct* [*consumes 2*, *induct set*: *wf-on*]:
[| *wf*[*A*](*r*); *a*:*A*;
!!*x*.[| *x*: *A*; *ALL y*:*A*. <*y*,*x*>: *r* --> *P*(*y*) |] ==> *P*(*x*)
|] ==> *P*(*a*)
⟨*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*:
[| *field*(*r*)<=*A*;

!!y B. [| ALL x:A. (ALL y:A. <y,x>:r --> y:B) --> x:B; y:A|]
==> y:B |]
==> $wf(r)$
⟨*proof*⟩

12.2 Basic Properties of Well-Founded Relations

lemma *wf-not-refl*: $wf(r)$ ==> <a,a> ~: r
⟨*proof*⟩

lemma *wf-not-sym* [*rule-format*]: $wf(r)$ ==> ALL x. <a,x>:r --> <x,a> ~: r
⟨*proof*⟩

lemmas *wf-asym* = *wf-not-sym* [*THEN swap*, *standard*]

lemma *wf-on-not-refl*: [| wf[A](r); a: A |] ==> <a,a> ~: r
⟨*proof*⟩

lemma *wf-on-not-sym* [*rule-format*]:
[| wf[A](r); a:A |] ==> ALL b:A. <a,b>:r --> <b,a>~:r
⟨*proof*⟩

lemma *wf-on-asym*:
[| wf[A](r); ~Z ==> <a,b> : r;
<b,a> ~: r ==> Z; ~Z ==> a : A; ~Z ==> b : A |] ==> Z
⟨*proof*⟩

lemma *wf-on-chain3*:
[| wf[A](r); <a,b>:r; <b,c>:r; <c,a>:r; a:A; b:A; c:A |] ==> P
⟨*proof*⟩

transitive closure of a WF relation is WF provided A is downward closed

lemma *wf-on-trancl*:
[| wf[A](r); r-``A <= A |] ==> wf[A](r^+)
⟨*proof*⟩

lemma *wf-trancl*: $wf(r)$ ==> $wf(r^+)$
⟨*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) ==> f: r-``{a} -> *range*(f)

⟨*proof*⟩

lemmas *is-recfun-imp-function* = *is-recfun-type* [*THEN fun-is-function*]

lemma *apply-recfun*:
[| *is-recfun*(*r*,*a*,*H*,*f*); <*x*,*a*>:*r* |] ==> *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*) |]
==> <*x*,*a*>:*r* −−> <*x*,*b*>:*r* −−> *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*); <*b*,*a*>:*r* |]
==> *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*) |] ==> *f*=*g*
⟨*proof*⟩

lemma *the-recfun-eq*:
[| *is-recfun*(*r*,*a*,*H*,*f*); *wf*(*r*); *trans*(*r*) |] ==> *the-recfun*(*r*,*a*,*H*) = *f*
⟨*proof*⟩

lemma *is-the-recfun*:
[| *is-recfun*(*r*,*a*,*H*,*f*); *wf*(*r*); *trans*(*r*) |]
==> *is-recfun*(*r*, *a*, *H*, *the-recfun*(*r*,*a*,*H*))
⟨*proof*⟩

lemma *unfold-the-recfun*:
[| *wf*(*r*); *trans*(*r*) |] ==> *is-recfun*(*r*, *a*, *H*, *the-recfun*(*r*,*a*,*H*))
⟨*proof*⟩

12.5 Unfolding *wftrec*(*r*, *a*, *H*)

lemma *the-recfun-cut*:
[| *wf*(*r*); *trans*(*r*); <*b*,*a*>:*r* |]
==> *restrict*(*the-recfun*(*r*,*a*,*H*), *r*−‘‘{*b*}) = *the-recfun*(*r*,*b*,*H*)
⟨*proof*⟩

lemma *wftrec*:
[| *wf*(*r*); *trans*(*r*) |] ==>
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*) ==> *wfrec*(*r*,*a*,*H*) = *H*(*a*, *lam x*:*r*−‘‘{*a*}. *wfrec*(*r*,*x*,*H*))
⟨*proof*⟩

lemma *def-wfrec*:
[| !!*x*. *h*(*x*)==*wfrec*(*r*,*x*,*H*); *wf*(*r*) |] ==>
h(*a*) = *H*(*a*, *lam x*: *r*−‘‘{*a*}. *h*(*x*))
⟨*proof*⟩

lemma *wfrec-type*:
[| *wf*(*r*); *a*:*A*; *field*(*r*)<=*A*;
!!*x u*. [| *x*: *A*; *u*: *Pi*(*r*−‘‘{*x*}, *B*) |] ==> *H*(*x*,*u*) : *B*(*x*)
|] ==> *wfrec*(*r*,*a*,*H*) : *B*(*a*)
⟨*proof*⟩

lemma *wfrec-on*:
[| *wf*[*A*](*r*); *a*: *A* |] ==>
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*) <−> (*ALL Q x*. *x*:*Q* −−> (*EX z*:*Q*. *ALL y*. <*y*,*z*>:*r* −−> *y*~:*Q*))
⟨*proof*⟩

end

13 Ordinal: Transitive Sets and Ordinals

theory *Ordinal* **imports** *WF Bool equalities* **begin**

definition
Memrel :: *i*=>*i* **where**
Memrel(*A*) == {*z*: *A*∗*A* . *EX x y*. *z*=<*x*,*y*> & *x*:*y* }

definition
Transset :: *i*=>*o* **where**
Transset(*i*) == *ALL x*:*i*. *x*<=*i*

definition
Ord :: *i*=>*o* **where**
Ord(*i*) == *Transset*(*i*) & (*ALL x*:*i*. *Transset*(*x*))

definition
lt :: [*i*,*i*] => *o* (**infixl** < *50*) **where**
i<*j* == *i*:*j* & *Ord*(*j*)

definition
Limit :: *i*=>*o* **where**
Limit(*i*) == *Ord*(*i*) & 0<*i* & (*ALL y*. *y*<*i* −−> *succ*(*y*)<*i*)

abbreviation
le (**infixl** *le 50*) **where**
x le y == *x* < *succ*(*y*)

notation (*xsymbols*)
le (**infixl** ≤ *50*)

notation (*HTML* **output**)
le (**infixl** ≤ *50*)

13.1 Rules for Transset

13.1.1 Three Neat Characterisations of Transset

lemma *Transset-iff-Pow*: *Transset*(*A*) <−> *A*<=*Pow*(*A*)
⟨*proof*⟩

lemma *Transset-iff-Union-succ*: *Transset*(*A*) <−> *Union*(*succ*(*A*)) = *A*
⟨*proof*⟩

lemma *Transset-iff-Union-subset*: *Transset*(*A*) <−> *Union*(*A*) <= *A*
⟨*proof*⟩

13.1.2 Consequences of Downwards Closure

lemma *Transset-doubleton-D*:
[| *Transset*(*C*); {*a*,*b*}: *C* |] ==> *a*:*C* & *b*: *C*
⟨*proof*⟩

lemma *Transset-Pair-D*:
[| *Transset*(*C*); <*a*,*b*>: *C* |] ==> *a*:*C* & *b*: *C*
⟨*proof*⟩

lemma *Transset-includes-domain*:
[| *Transset*(*C*); *A*∗*B* <= *C*; *b*: *B* |] ==> *A* <= *C*
⟨*proof*⟩

lemma *Transset-includes-range*:
[| *Transset*(*C*); *A*∗*B* <= *C*; *a*: *A* |] ==> *B* <= *C*
⟨*proof*⟩

13.1.3 Closure Properties

lemma *Transset-0*: $Transset(0)$
⟨*proof*⟩

lemma *Transset-Un*:
$[|\ Transset(i);\ \ Transset(j)\ |] ==> Transset(i\ Un\ j)$
⟨*proof*⟩

lemma *Transset-Int*:
$[|\ Transset(i);\ \ Transset(j)\ |] ==> Transset(i\ Int\ j)$
⟨*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*:
$[|\ !!i.\ i{:}A ==> Transset(i)\ |] ==> Transset(Union(A))$
⟨*proof*⟩

lemma *Transset-Inter-family*:
$[|\ !!i.\ i{:}A ==> Transset(i)\ |] ==> Transset(Inter(A))$
⟨*proof*⟩

lemma *Transset-UN*:
$(!!x.\ x \in A ==> Transset(B(x))) ==> Transset\ (\bigcup x{\in}A.\ B(x))$
⟨*proof*⟩

lemma *Transset-INT*:
$(!!x.\ x \in A ==> Transset(B(x))) ==> Transset\ (\bigcap x{\in}A.\ B(x))$
⟨*proof*⟩

13.2 Lemmas for Ordinals

lemma *OrdI*:
$[|\ Transset(i);\ \ !!x.\ x{:}i ==> Transset(x)\ |]\ \ ==>\ \ Ord(i)$
⟨*proof*⟩

lemma *Ord-is-Transset*: $Ord(i) ==> Transset(i)$
⟨*proof*⟩

lemma *Ord-contains-Transset*:
$[|\ Ord(i);\ \ j{:}i\ |] ==> Transset(j)$
⟨*proof*⟩

lemma *Ord-in-Ord*: [| *Ord*(*i*); *j*:*i* |] ==> *Ord*(*j*)
⟨*proof*⟩

lemma *Ord-in-Ord'*: [| *j*:*i*; *Ord*(*i*) |] ==> *Ord*(*j*)
⟨*proof*⟩

lemmas *Ord-succD* = *Ord-in-Ord* [*OF* - *succI1*]

lemma *Ord-subset-Ord*: [| *Ord*(*i*); *Transset*(*j*); *j*<=*i* |] ==> *Ord*(*j*)
⟨*proof*⟩

lemma *OrdmemD*: [| *j*:*i*; *Ord*(*i*) |] ==> *j*<=*i*
⟨*proof*⟩

lemma *Ord-trans*: [| *i*:*j*; *j*:*k*; *Ord*(*k*) |] ==> *i*:*k*
⟨*proof*⟩

lemma *Ord-succ-subsetI*: [| *i*:*j*; *Ord*(*j*) |] ==> *succ*(*i*) <= *j*
⟨*proof*⟩

13.3 The Construction of Ordinals: 0, succ, Union

lemma *Ord-0* [*iff*,*TC*]: *Ord*(*0*)
⟨*proof*⟩

lemma *Ord-succ* [*TC*]: *Ord*(*i*) ==> *Ord*(*succ*(*i*))
⟨*proof*⟩

lemmas *Ord-1* = *Ord-0* [*THEN Ord-succ*]

lemma *Ord-succ-iff* [*iff*]: *Ord*(*succ*(*i*)) <-> *Ord*(*i*)
⟨*proof*⟩

lemma *Ord-Un* [*intro*,*simp*,*TC*]: [| *Ord*(*i*); *Ord*(*j*) |] ==> *Ord*(*i Un j*)
⟨*proof*⟩

lemma *Ord-Int* [*TC*]: [| *Ord*(*i*); *Ord*(*j*) |] ==> *Ord*(*i Int j*)
⟨*proof*⟩

lemma *ON-class*: ~ (*ALL i*. *i*:*X* <-> *Ord*(*i*))
⟨*proof*⟩

13.4 ¡ is 'less Than' for Ordinals

lemma *ltI*: [| *i*:*j*; *Ord*(*j*) |] ==> *i*<*j*

⟨*proof*⟩

lemma *ltE*:
[| $i<j$; [| $i:j$; $Ord(i)$; $Ord(j)$ |] ==> P |] ==> P
⟨*proof*⟩

lemma *ltD*: $i<j$ ==> $i:j$
⟨*proof*⟩

lemma *not-lt0* [*simp*]: ~ $i<0$
⟨*proof*⟩

lemma *lt-Ord*: $j<i$ ==> $Ord(j)$
⟨*proof*⟩

lemma *lt-Ord2*: $j<i$ ==> $Ord(i)$
⟨*proof*⟩

lemmas *le-Ord2* = *lt-Ord2* [*THEN Ord-succD*]

lemmas *lt0E* = *not-lt0* [*THEN notE*, *elim*!]

lemma *lt-trans*: [| $i<j$; $j<k$ |] ==> $i<k$
⟨*proof*⟩

lemma *lt-not-sym*: $i<j$ ==> ~ $(j<i)$
⟨*proof*⟩

lemmas *lt-asym* = *lt-not-sym* [*THEN swap*]

lemma *lt-irrefl* [*elim*!]: $i<i$ ==> P
⟨*proof*⟩

lemma *lt-not-refl*: ~ $i<i$
⟨*proof*⟩

lemma *le-iff*: i *le* j <-> $i<j$ | $(i=j$ & $Ord(j))$
⟨*proof*⟩

lemma *leI*: $i<j$ ==> i *le* j
⟨*proof*⟩

lemma *le-eqI*: [| *i=j*; *Ord(j)* |] ==> *i le j*
⟨*proof*⟩

lemmas *le-refl* = *refl* [*THEN le-eqI*]

lemma *le-refl-iff* [*iff*]: *i le i* <-> *Ord(i)*
⟨*proof*⟩

lemma *leCI*: (~ (*i=j* & *Ord(j)*) ==> *i<j*) ==> *i le j*
⟨*proof*⟩

lemma *leE*:
[| *i le j*; *i<j* ==> *P*; [| *i=j*; *Ord(j)* |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *le-anti-sym*: [| *i le j*; *j le i* |] ==> *i=j*
⟨*proof*⟩

lemma *le0-iff* [*simp*]: *i le 0* <-> *i=0*
⟨*proof*⟩

lemmas *le0D* = *le0-iff* [*THEN iffD1*, *dest*!]

13.5 Natural Deduction Rules for Memrel

lemma *Memrel-iff* [*simp*]: <*a*,*b*> : *Memrel*(*A*) <-> *a*:*b* & *a*:*A* & *b*:*A*
⟨*proof*⟩

lemma *MemrelI* [*intro*!]: [| *a*: *b*; *a*: *A*; *b*: *A* |] ==> <*a*,*b*> : *Memrel*(*A*)
⟨*proof*⟩

lemma *MemrelE* [*elim*!]:
[| <*a*,*b*> : *Memrel*(*A*);
[| *a*: *A*; *b*: *A*; *a*:*b* |] ==> *P* |]
==> *P*
⟨*proof*⟩

lemma *Memrel-type*: *Memrel*(*A*) <= *A*∗*A*
⟨*proof*⟩

lemma *Memrel-mono*: *A*<=*B* ==> *Memrel*(*A*) <= *Memrel*(*B*)
⟨*proof*⟩

lemma *Memrel-0* [*simp*]: *Memrel*(*0*) = *0*
⟨*proof*⟩

lemma *Memrel-1* [*simp*]: *Memrel*(*1*) = *0*
⟨*proof*⟩

lemma *relation-Memrel*: *relation*(*Memrel*(*A*))
⟨*proof*⟩

lemma *wf-Memrel*: *wf*(*Memrel*(*A*))
⟨*proof*⟩

The premise *Ord*(*i*) does not suffice.

lemma *trans-Memrel*:
Ord(*i*) ==> *trans*(*Memrel*(*i*))
⟨*proof*⟩

However, the following premise is strong enough.

lemma *Transset-trans-Memrel*:
∀*j*∈*i*. *Transset*(*j*) ==> *trans*(*Memrel*(*i*))
⟨*proof*⟩

lemma *Transset-Memrel-iff*:
Transset(*A*) ==> <*a*,*b*> : *Memrel*(*A*) <-> *a*:*b* & *b*:*A*
⟨*proof*⟩

13.6 Transfinite Induction

lemma *Transset-induct*:
[| *i*: *k*; *Transset*(*k*);
!!*x*.[| *x*: *k*; *ALL y*:*x*. *P*(*y*) |] ==> *P*(*x*) |]
==> *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*]:
[| *Ord*(*i*);
!!*x*.[| *Ord*(*x*); *ALL y*:*x*. *P*(*y*) |] ==> *P*(*x*) |]
==> *P*(*i*)
⟨*proof*⟩

lemmas *trans-induct-rule* = *trans-induct* [*rule-format*, *consumes 1*]

13.6.1 Proving That ¡ is a Linear Ordering on the Ordinals

lemma *Ord-linear* [*rule-format*]:
Ord(*i*) ==> (*ALL j*. *Ord*(*j*) --> *i*:*j* | *i*=*j* | *j*:*i*)
⟨*proof*⟩

lemma *Ord-linear-lt*:
[| *Ord*(*i*); *Ord*(*j*); *i*<*j* ==> *P*; *i*=*j* ==> *P*; *j*<*i* ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *Ord-linear2*:
[| *Ord*(*i*); *Ord*(*j*); *i*<*j* ==> *P*; *j le i* ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *Ord-linear-le*:
[| *Ord*(*i*); *Ord*(*j*); *i le j* ==> *P*; *j le i* ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *le-imp-not-lt*: *j le i* ==> ~ *i*<*j*
⟨*proof*⟩

lemma *not-lt-imp-le*: [| ~ *i*<*j*; *Ord*(*i*); *Ord*(*j*) |] ==> *j le i*
⟨*proof*⟩

13.6.2 Some Rewrite Rules for ¡, le

lemma *Ord-mem-iff-lt*: *Ord*(*j*) ==> *i*:*j* <-> *i*<*j*
⟨*proof*⟩

lemma *not-lt-iff-le*: [| *Ord*(*i*); *Ord*(*j*) |] ==> ~ *i*<*j* <-> *j le i*
⟨*proof*⟩

lemma *not-le-iff-lt*: [| *Ord*(*i*); *Ord*(*j*) |] ==> ~ *i le j* <-> *j*<*i*
⟨*proof*⟩

lemma *Ord-0-le*: *Ord*(*i*) ==> *0 le i*
⟨*proof*⟩

lemma *Ord-0-lt*: [| *Ord*(*i*); *i*~=*0* |] ==> *0*<*i*
⟨*proof*⟩

lemma *Ord-0-lt-iff*: *Ord*(*i*) ==> *i*~=*0* <-> *0*<*i*
⟨*proof*⟩

13.7 Results about Less-Than or Equals

lemma *zero-le-succ-iff* [*iff*]: *0 le succ*(*x*) <-> *Ord*(*x*)
⟨*proof*⟩

lemma *subset-imp-le*: [| *j*<=*i*; *Ord*(*i*); *Ord*(*j*) |] ==> *j le i*
⟨*proof*⟩

lemma *le-imp-subset*: *i le j* ==> *i*<=*j*

⟨*proof*⟩

lemma *le-subset-iff*: *j le i <−> j<=i & Ord(i) & Ord(j)*
⟨*proof*⟩

lemma *le-succ-iff*: *i le succ(j) <−> i le j | i=succ(j) & Ord(i)*
⟨*proof*⟩

lemma *all-lt-imp-le*: [| *Ord(i)*; *Ord(j)*; !!*x*. *x<j ==> x<i* |] ==> *j le i*
⟨*proof*⟩

13.7.1 Transitivity Laws

lemma *lt-trans1*: [| *i le j*; *j<k* |] ==> *i<k*
⟨*proof*⟩

lemma *lt-trans2*: [| *i<j*; *j le k* |] ==> *i<k*
⟨*proof*⟩

lemma *le-trans*: [| *i le j*; *j le k* |] ==> *i le k*
⟨*proof*⟩

lemma *succ-leI*: *i<j ==> succ(i) le j*
⟨*proof*⟩

lemma *succ-leE*: *succ(i) le j ==> i<j*
⟨*proof*⟩

lemma *succ-le-iff* [*iff*]: *succ(i) le j <−> i<j*
⟨*proof*⟩

lemma *succ-le-imp-le*: *succ(i) le succ(j) ==> i le j*
⟨*proof*⟩

lemma *lt-subset-trans*: [| *i <= j*; *j<k*; *Ord(i)* |] ==> *i<k*
⟨*proof*⟩

lemma *lt-imp-0-lt*: *j<i ==> 0<i*
⟨*proof*⟩

lemma *succ-lt-iff*: *succ(i) < j <−> i<j & succ(i) ≠ j*
⟨*proof*⟩

lemma *Ord-succ-mem-iff*: *Ord(j) ==> succ(i) ∈ succ(j) <−> i∈j*
⟨*proof*⟩

13.7.2 Union and Intersection

lemma *Un-upper1-le*: [| *Ord*(*i*); *Ord*(*j*) |] ==> *i le i Un j*
⟨*proof*⟩

lemma *Un-upper2-le*: [| *Ord*(*i*); *Ord*(*j*) |] ==> *j le i Un j*
⟨*proof*⟩

lemma *Un-least-lt*: [| *i*<*k*; *j*<*k* |] ==> *i Un j* < *k*
⟨*proof*⟩

lemma *Un-least-lt-iff*: [| *Ord*(*i*); *Ord*(*j*) |] ==> *i Un j* < *k* <-> *i*<*k* & *j*<*k*
⟨*proof*⟩

lemma *Un-least-mem-iff*:
[| *Ord*(*i*); *Ord*(*j*); *Ord*(*k*) |] ==> *i Un j* : *k* <-> *i*:*k* & *j*:*k*
⟨*proof*⟩

lemma *Int-greatest-lt*: [| *i*<*k*; *j*<*k* |] ==> *i Int j* < *k*
⟨*proof*⟩

lemma *Ord-Un-if*:
[| *Ord*(*i*); *Ord*(*j*) |] ==> *i* ∪ *j* = (*if j*<*i then i else j*)
⟨*proof*⟩

lemma *succ-Un-distrib*:
[| *Ord*(*i*); *Ord*(*j*) |] ==> *succ*(*i* ∪ *j*) = *succ*(*i*) ∪ *succ*(*j*)
⟨*proof*⟩

lemma *lt-Un-iff*:
[| *Ord*(*i*); *Ord*(*j*) |] ==> *k* < *i* ∪ *j* <-> *k* < *i* | *k* < *j*
⟨*proof*⟩

lemma *le-Un-iff*:
[| *Ord*(*i*); *Ord*(*j*) |] ==> *k* ≤ *i* ∪ *j* <-> *k* ≤ *i* | *k* ≤ *j*
⟨*proof*⟩

lemma *Un-upper1-lt*: [|*k* < *i*; *Ord*(*j*)|] ==> *k* < *i Un j*
⟨*proof*⟩

lemma *Un-upper2-lt*: [|*k* < *j*; *Ord*(*i*)|] ==> *k* < *i Un j*
⟨*proof*⟩

lemma *Ord-Union-succ-eq*: *Ord*(*i*) ==> ⋃(*succ*(*i*)) = *i*
⟨*proof*⟩

13.8 Results about Limits

lemma *Ord-Union* [*intro*,*simp*,*TC*]: [| !!*i*. *i*:*A* ==> *Ord*(*i*) |] ==> *Ord*(*Union*(*A*))
⟨*proof*⟩

lemma *Ord-UN* [*intro*,*simp*,*TC*]:
[| !!*x*. *x*:*A* ==> *Ord*(*B*(*x*)) |] ==> *Ord*(⋃*x*∈*A*. *B*(*x*))
⟨*proof*⟩

lemma *Ord-Inter* [*intro*,*simp*,*TC*]:
[| !!*i*. *i*:*A* ==> *Ord*(*i*) |] ==> *Ord*(*Inter*(*A*))
⟨*proof*⟩

lemma *Ord-INT* [*intro*,*simp*,*TC*]:
[| !!*x*. *x*:*A* ==> *Ord*(*B*(*x*)) |] ==> *Ord*(⋂*x*∈*A*. *B*(*x*))
⟨*proof*⟩

lemma *UN-least-le*:
[| *Ord*(*i*); !!*x*. *x*:*A* ==> *b*(*x*) *le* *i* |] ==> (⋃*x*∈*A*. *b*(*x*)) *le* *i*
⟨*proof*⟩

lemma *UN-succ-least-lt*:
[| *j*<*i*; !!*x*. *x*:*A* ==> *b*(*x*)<*j* |] ==> (⋃*x*∈*A*. *succ*(*b*(*x*))) < *i*
⟨*proof*⟩

lemma *UN-upper-lt*:
[| *a*∈*A*; *i* < *b*(*a*); *Ord*(⋃*x*∈*A*. *b*(*x*)) |] ==> *i* < (⋃*x*∈*A*. *b*(*x*))
⟨*proof*⟩

lemma *UN-upper-le*:
[| *a*: *A*; *i le* *b*(*a*); *Ord*(⋃*x*∈*A*. *b*(*x*)) |] ==> *i le* (⋃*x*∈*A*. *b*(*x*))
⟨*proof*⟩

lemma *lt-Union-iff*: ∀*i*∈*A*. *Ord*(*i*) ==> (*j* < ⋃(*A*)) <-> (∃*i*∈*A*. *j*<*i*)
⟨*proof*⟩

lemma *Union-upper-le*:
[| *j*: *J*; *i*≤*j*; *Ord*(⋃(*J*)) |] ==> *i* ≤ ⋃*J*
⟨*proof*⟩

lemma *le-implies-UN-le-UN*:
[| !!*x*. *x*:*A* ==> *c*(*x*) *le* *d*(*x*) |] ==> (⋃*x*∈*A*. *c*(*x*)) *le* (⋃*x*∈*A*. *d*(*x*))
⟨*proof*⟩

lemma *Ord-equality*: *Ord*(*i*) ==> (⋃*y*∈*i*. *succ*(*y*)) = *i*
⟨*proof*⟩

lemma *Ord-Union-subset*: *Ord*(*i*) ==> *Union*(*i*) <= *i*
⟨*proof*⟩

13.9 Limit Ordinals – General Properties

lemma *Limit-Union-eq*: *Limit*(*i*) ==> *Union*(*i*) = *i*
⟨*proof*⟩

lemma *Limit-is-Ord*: *Limit*(*i*) ==> *Ord*(*i*)
⟨*proof*⟩

lemma *Limit-has-0*: *Limit*(*i*) ==> 0 < *i*
⟨*proof*⟩

lemma *Limit-nonzero*: *Limit*(*i*) ==> *i* ~= 0
⟨*proof*⟩

lemma *Limit-has-succ*: [| *Limit*(*i*); *j*<*i* |] ==> *succ*(*j*) < *i*
⟨*proof*⟩

lemma *Limit-succ-lt-iff* [*simp*]: *Limit*(*i*) ==> *succ*(*j*) < *i* <-> (*j*<*i*)
⟨*proof*⟩

lemma *zero-not-Limit* [*iff*]: ~ *Limit*(0)
⟨*proof*⟩

lemma *Limit-has-1*: *Limit*(*i*) ==> 1 < *i*
⟨*proof*⟩

lemma *increasing-LimitI*: [| 0<*l*; ∀ *x*∈*l*. ∃ *y*∈*l*. *x*<*y* |] ==> *Limit*(*l*)
⟨*proof*⟩

lemma *non-succ-LimitI*:
[| 0<*i*; ALL *y*. *succ*(*y*) ~= *i* |] ==> *Limit*(*i*)
⟨*proof*⟩

lemma *succ-LimitE* [*elim*!]: *Limit*(*succ*(*i*)) ==> *P*
⟨*proof*⟩

lemma *not-succ-Limit* [*simp*]: ~ *Limit*(*succ*(*i*))
⟨*proof*⟩

lemma *Limit-le-succD*: [| *Limit*(*i*); *i le succ*(*j*) |] ==> *i le j*
⟨*proof*⟩

13.9.1 Traditional 3-Way Case Analysis on Ordinals

lemma *Ord-cases-disj*: *Ord*(*i*) ==> *i*=0 | (EX *j*. *Ord*(*j*) & *i*=*succ*(*j*)) | *Limit*(*i*)
⟨*proof*⟩

lemma *Ord-cases*:
[| $Ord(i)$;
$i=0$ ==> P;
!!j. [| $Ord(j)$; $i=succ(j)$ |] ==> P;
$Limit(i)$ ==> P
|] ==> P
⟨*proof*⟩

lemma *trans-induct3* [*case-names 0 succ limit*, *consumes 1*]:
[| $Ord(i)$;
$P(0)$;
!!x. [| $Ord(x)$; $P(x)$ |] ==> $P(succ(x))$;
!!x. [| $Limit(x)$; ALL y:x. $P(y)$ |] ==> $P(x)$
|] ==> $P(i)$
⟨*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*:
$\forall i{\in}I.\ Ord(i) ==> I=0 \vee \bigcup(I) \in I \vee (\bigcup(I) \notin I \wedge Limit(\bigcup(I)))$
⟨*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*: [|$\forall x{\in}X.\ Ord(x);\ \bigcup X = succ(j)$|] ==> $succ(j) \in X$
⟨*proof*⟩

lemma *Limit-Union* [*rule-format*]: [| $I \neq 0$; $\forall i{\in}I.\ Limit(i)$ |] ==> $Limit(\bigcup I)$
⟨*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*: ⋃*x*∈*i*. *B*(*x*). *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* ((*3*∀ -<-./ -) *10*)
@*oex* :: [*idt*, *i*, *o*] => *o* ((*3*∃ -<-./ -) *10*)
@*OUNION* :: [*idt*, *i*, *i*] => *i* ((*3*⋃ -<-./ -) *10*)
syntax (*HTML* **output**)
@*oall* :: [*idt*, *i*, *o*] => *o* ((*3*∀ -<-./ -) *10*)
@*oex* :: [*idt*, *i*, *o*] => *o* ((*3*∃ -<-./ -) *10*)
@*OUNION* :: [*idt*, *i*, *i*] => *i* ((*3*⋃ -<-./ -) *10*)

14.1.1 simplification of the new quantifiers

lemma [*simp*]: (*ALL x*<*0*. *P*(*x*))
⟨*proof*⟩

lemma [*simp*]: ~(*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*]: (*EX x*<*succ*(*i*). *P*(*x*)) <-> (*Ord*(*i*) & (*P*(*i*) | (*EX x*<*i*. *P*(*x*))))
⟨*proof*⟩

14.1.2 Union over ordinals

lemma *Ord-OUN* [*intro*,*simp*]:
[| !!*x*. *x*<*A* ==> *Ord*(*B*(*x*)) |] ==> *Ord*(⋃*x*<*A*. *B*(*x*))
⟨*proof*⟩

lemma *OUN-upper-lt*:

[| $a<A$; $i < b(a)$; $Ord(\bigcup x<A.\ b(x))$ |] ==> $i < (\bigcup x<A.\ b(x))$
⟨*proof*⟩

lemma *OUN-upper-le*:
[| $a<A$; $i \leq b(a)$; $Ord(\bigcup x<A.\ b(x))$ |] ==> $i \leq (\bigcup x<A.\ b(x))$
⟨*proof*⟩

lemma *Limit-OUN-eq*: $Limit(i)$ ==> $(\bigcup x<i.\ x) = i$
⟨*proof*⟩

lemma *OUN-least*:
(!!x. $x<A$ ==> $B(x) \subseteq C$) ==> $(\bigcup x<A.\ B(x)) \subseteq C$
⟨*proof*⟩

lemma *OUN-least-le*:
[| $Ord(i)$; !!x. $x<A$ ==> $b(x) \leq i$ |] ==> $(\bigcup x<A.\ b(x)) \leq i$
⟨*proof*⟩

lemma *le-implies-OUN-le-OUN*:
[| !!x. $x<A$ ==> $c(x) \leq d(x)$ |] ==> $(\bigcup x<A.\ c(x)) \leq (\bigcup x<A.\ d(x))$
⟨*proof*⟩

lemma *OUN-UN-eq*:
(!!x. x:A ==> $Ord(B(x))$)
==> $(\bigcup z < (\bigcup x\in A.\ B(x)).\ C(z)) = (\bigcup x\in A.\ \bigcup z < B(x).\ C(z))$
⟨*proof*⟩

lemma *OUN-Union-eq*:
(!!x. x:X ==> $Ord(x)$)
==> $(\bigcup z < Union(X).\ C(z)) = (\bigcup x\in X.\ \bigcup z < x.\ C(z))$
⟨*proof*⟩

lemma *atomize-oall* [*symmetric*, *rulify*]:
(!!x. $x<A$ ==> $P(x)$) == *Trueprop* (*ALL* $x<A$. $P(x)$)
⟨*proof*⟩

14.1.3 universal quantifier for ordinals

lemma *oallI* [*intro*!]:
[| !!x. $x<A$ ==> $P(x)$ |] ==> *ALL* $x<A$. $P(x)$
⟨*proof*⟩

lemma *ospec*: [| *ALL* $x<A$. $P(x)$; $x<A$ |] ==> $P(x)$
⟨*proof*⟩

lemma *oallE*:

[| $ALL\ x{<}A.\ P(x)$; $P(x) ==> Q$; $\sim x{<}A ==> Q$ |] $==> Q$
⟨*proof*⟩

lemma *rev-oallE* [*elim*]:
[| $ALL\ x{<}A.\ P(x)$; $\sim x{<}A ==> Q$; $P(x) ==> Q$ |] $==> Q$
⟨*proof*⟩

lemma *oall-simp* [*simp*]: $(ALL\ x{<}a.\ True) <-> True$
⟨*proof*⟩

lemma *oall-cong* [*cong*]:
[| $a{=}a'$; $!!x.\ x{<}a' ==> P(x) <-> P'(x)$ |]
$==> oall(a, \%x.\ P(x)) <-> oall(a', \%x.\ P'(x))$
⟨*proof*⟩

14.1.4 existential quantifier for ordinals

lemma *oexI* [*intro*]:
[| $P(x)$; $x{<}A$ |] $==> EX\ x{<}A.\ P(x)$
⟨*proof*⟩

lemma *oexCI*:
[| $ALL\ x{<}A.\ \sim P(x) ==> P(a)$; $a{<}A$ |] $==> EX\ x{<}A.\ P(x)$
⟨*proof*⟩

lemma *oexE* [*elim*!]:
[| $EX\ x{<}A.\ P(x)$; $!!x.$ [| $x{<}A$; $P(x)$ |] $==> Q$ |] $==> Q$
⟨*proof*⟩

lemma *oex-cong* [*cong*]:
[| $a{=}a'$; $!!x.\ x{<}a' ==> P(x) <-> P'(x)$ |]
$==> oex(a, \%x.\ P(x)) <-> oex(a', \%x.\ P'(x))$
⟨*proof*⟩

14.1.5 Rules for Ordinal-Indexed Unions

lemma *OUN-I* [*intro*]: [| $a{<}i$; $b{:}\ B(a)$ |] $==> b{:}\ (\bigcup z{<}i.\ B(z))$
⟨*proof*⟩

lemma *OUN-E* [*elim*!]:
[| $b : (\bigcup z{<}i.\ B(z))$; $!!a.$[| $b{:}\ B(a)$; $a{<}i$ |] $==> R$ |] $==> R$
⟨*proof*⟩

lemma *OUN-iff*: $b : (\bigcup x{<}i.\ B(x)) <-> (EX\ x{<}i.\ b : B(x))$
⟨*proof*⟩

lemma *OUN-cong* [*cong*]:
[| i=j; !!x. x<j ==> $C(x)$=$D(x)$ |] ==> ($\bigcup x{<}i.\ C(x)$) = ($\bigcup x{<}j.\ D(x)$)
⟨*proof*⟩

lemma *lt-induct*:
[| i<k; !!x.[| x<k; ALL y<x. $P(y)$ |] ==> $P(x)$ |] ==> $P(i)$
⟨*proof*⟩

14.2 Quantification over a class

definition
rall :: [i=>o, i=>o] => o **where**
rall(M, P) == ALL x. $M(x)$ --> $P(x)$

definition
rex :: [i=>o, i=>o] => o **where**
rex(M, P) == EX x. $M(x)$ & $P(x)$

syntax
@*rall* :: [*pttrn*, i=>o, o] => o ((3ALL -[-]./ -) 10)
@*rex* :: [*pttrn*, i=>o, o] => o ((3EX -[-]./ -) 10)

syntax (*xsymbols*)
@*rall* :: [*pttrn*, i=>o, o] => o ((3∀ -[-]./ -) 10)
@*rex* :: [*pttrn*, i=>o, o] => o ((3∃ -[-]./ -) 10)
syntax (*HTML* **output**)
@*rall* :: [*pttrn*, i=>o, o] => o ((3∀ -[-]./ -) 10)
@*rex* :: [*pttrn*, i=>o, o] => o ((3∃ -[-]./ -) 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. $M(x)$ ==> $P(x)$ |] ==> ALL x[M]. $P(x)$
⟨*proof*⟩

lemma *rspec*: [| ALL x[M]. $P(x)$; $M(x)$ |] ==> $P(x)$
⟨*proof*⟩

lemma *rev-rallE* [*elim*]:
[| ALL x[M]. $P(x)$; ~ $M(x)$ ==> Q; $P(x)$ ==> Q |] ==> Q
⟨*proof*⟩

lemma *rallE*: [| ALL x[M]. $P(x)$; $P(x)$ ==> Q; ~ $M(x)$ ==> Q |] ==> Q
⟨*proof*⟩

lemma *rall-triv* [*simp*]: $(ALL\ x[M].\ P) <-> ((EX\ x.\ M(x)) --> P)$
⟨*proof*⟩

lemma *rall-cong* [*cong*]:
$(!!x.\ M(x) ==> P(x) <-> P'(x)) ==> (ALL\ x[M].\ P(x)) <-> (ALL\ x[M].\ P'(x))$
⟨*proof*⟩

14.2.2 Relativized existential quantifier

lemma *rexI* [*intro*]: $[|\ P(x);\ M(x)\ |] ==> EX\ x[M].\ P(x)$
⟨*proof*⟩

lemma *rev-rexI*: $[|\ M(x);\ \ P(x)\ |] ==> EX\ x[M].\ P(x)$
⟨*proof*⟩

lemma *rexCI*: $[|\ ALL\ x[M].\ ~P(x) ==> P(a);\ M(a)\ |] ==> EX\ x[M].\ P(x)$
⟨*proof*⟩

lemma *rexE* [*elim*!]: $[|\ EX\ x[M].\ P(x);\ \ !!x.\ [|\ M(x);\ P(x)\ |] ==> Q\ |] ==> Q$
⟨*proof*⟩

lemma *rex-triv* [*simp*]: $(EX\ x[M].\ P) <-> ((EX\ x.\ M(x)) \&\ P)$
⟨*proof*⟩

lemma *rex-cong* [*cong*]:
$(!!x.\ M(x) ==> P(x) <-> P'(x)) ==> (EX\ x[M].\ P(x)) <-> (EX\ x[M].\ P'(x))$
⟨*proof*⟩

lemma *rall-is-ball* [*simp*]: $(\forall x[\%z.\ z{\in}A].\ P(x)) <-> (\forall x{\in}A.\ P(x))$
⟨*proof*⟩

lemma *rex-is-bex* [*simp*]: $(\exists x[\%z.\ z{\in}A].\ P(x)) <-> (\exists x{\in}A.\ P(x))$
⟨*proof*⟩

lemma *atomize-rall*: $(!!x.\ M(x) ==> P(x)) == Trueprop\ (ALL\ x[M].\ P(x))$
⟨*proof*⟩

declare *atomize-rall* [*symmetric*, *rulify*]

lemma *rall-simps1*:
$(ALL\ x[M].\ P(x) \&\ Q) \ \ <-> (ALL\ x[M].\ P(x)) \&\ ((ALL\ x[M].\ False) \mid Q)$
$(ALL\ x[M].\ P(x) \mid Q) \ \ <-> ((ALL\ x[M].\ P(x)) \mid Q)$
$(ALL\ x[M].\ P(x) --> Q) <-> ((EX\ x[M].\ P(x)) --> Q)$

$(\sim(ALL\ x[M].\ P(x))) <-> (EX\ x[M].\ \sim P(x))$
⟨*proof*⟩

lemma *rall-simps2*:
$(ALL\ x[M].\ P\ \&\ Q(x)) <-> ((ALL\ x[M].\ False) \mid P)\ \&\ (ALL\ x[M].\ Q(x))$
$(ALL\ x[M].\ P \mid Q(x)) <-> (P \mid (ALL\ x[M].\ Q(x)))$
$(ALL\ x[M].\ P --> Q(x)) <-> (P --> (ALL\ x[M].\ Q(x)))$
⟨*proof*⟩

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

lemma *rall-conj-distrib*:
$(ALL\ x[M].\ P(x)\ \&\ Q(x)) <-> ((ALL\ x[M].\ P(x))\ \&\ (ALL\ x[M].\ Q(x)))$
⟨*proof*⟩

lemma *rex-simps1*:
$(EX\ x[M].\ P(x)\ \&\ Q) <-> ((EX\ x[M].\ P(x))\ \&\ Q)$
$(EX\ x[M].\ P(x) \mid Q) <-> (EX\ x[M].\ P(x)) \mid ((EX\ x[M].\ True)\ \&\ Q)$
$(EX\ x[M].\ P(x) --> Q) <-> ((ALL\ x[M].\ P(x)) --> ((EX\ x[M].\ True)\ \&\ Q))$
$(\sim(EX\ x[M].\ P(x))) <-> (ALL\ x[M].\ \sim P(x))$
⟨*proof*⟩

lemma *rex-simps2*:
$(EX\ x[M].\ P\ \&\ Q(x)) <-> (P\ \&\ (EX\ x[M].\ Q(x)))$
$(EX\ x[M].\ P \mid Q(x)) <-> ((EX\ x[M].\ True)\ \&\ P) \mid (EX\ x[M].\ Q(x))$
$(EX\ x[M].\ P --> Q(x)) <-> (((ALL\ x[M].\ False) \mid P) --> (EX\ x[M].\ Q(x)))$
⟨*proof*⟩

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

lemma *rex-disj-distrib*:
$(EX\ x[M].\ P(x) \mid Q(x)) <-> ((EX\ x[M].\ P(x)) \mid (EX\ x[M].\ Q(x)))$
⟨*proof*⟩

14.2.3 One-point rule for bounded quantifiers

lemma *rex-triv-one-point1* [*simp*]: $(EX\ x[M].\ x=a) <-> (\ M(a))$
⟨*proof*⟩

lemma *rex-triv-one-point2* [*simp*]: $(EX\ x[M].\ a=x) <-> (\ M(a))$
⟨*proof*⟩

lemma *rex-one-point1* [*simp*]: $(EX\ x[M].\ x=a\ \&\ P(x)) <-> (\ M(a)\ \&\ P(a))$
⟨*proof*⟩

lemma *rex-one-point2* [*simp*]: $(EX\ x[M].\ a=x\ \&\ P(x)) <-> (\ M(a)\ \&\ P(a))$
⟨*proof*⟩

lemma *rall-one-point1* [*simp*]: $(ALL\ x[M].\ x{=}a\ {-}{-}{>}\ P(x))\ {<}{-}{>}\ (\ M(a)\ {-}{-}{>}\ P(a))$
⟨*proof*⟩

lemma *rall-one-point2* [*simp*]: $(ALL\ x[M].\ a{=}x\ {-}{-}{>}\ P(x))\ {<}{-}{>}\ (\ M(a)\ {-}{-}{>}\ P(a))$
⟨*proof*⟩

14.2.4 Sets as Classes

definition
setclass :: [*i*,*i*] => *o* (##- [40] 40) **where**
setclass(*A*) == %*x*. *x* : *A*

lemma *setclass-iff* [*simp*]: *setclass*(*A*,*x*) <-> *x* : *A*
⟨*proof*⟩

lemma *rall-setclass-is-ball* [*simp*]: $(\forall\, x[\#\#A].\ P(x))\ {<}{-}{>}\ (\forall\, x{\in}A.\ P(x))$
⟨*proof*⟩

lemma *rex-setclass-is-bex* [*simp*]: $(\exists\, x[\#\#A].\ P(x))\ {<}{-}{>}\ (\exists\, x{\in}A.\ P(x))$
⟨*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
quasinat :: *i* => *o* **where**
quasinat(*n*) == *n*=*0* | (∃ *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* : *nat*
⟨*proof*⟩

lemma *nat-succI* [*intro*!,*TC*]: *n* : *nat* ==> *succ*(*n*) : *nat*
⟨*proof*⟩

lemma *nat-1I* [*iff*,*TC*]: *1* : *nat*
⟨*proof*⟩

lemma *nat-2I* [*iff*, *TC*]: *2* : *nat*
⟨*proof*⟩

lemma *bool-subset-nat*: *bool* <= *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*]:
[| *n*: *nat*; *P*(*0*); !!*x*. [| *x*: *nat*; *P*(*x*) |] ==> *P*(*succ*(*x*)) |] ==> *P*(*n*)
⟨*proof*⟩

lemma *natE*:
[| *n*: *nat*; *n*=*0* ==> *P*; !!*x*. [| *x*: *nat*; *n*=*succ*(*x*) |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *nat-into-Ord* [*simp*]: *n*: *nat* ==> *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*]: *Ord*(*nat*)
⟨*proof*⟩

lemma *Limit-nat* [*iff*]: *Limit*(*nat*)
⟨*proof*⟩

lemma *naturals-not-limit*: *a* ∈ *nat* ==> ~ *Limit*(*a*)
⟨*proof*⟩

lemma *succ-natD*: *succ*(*i*): *nat* ==> *i*: *nat*
⟨*proof*⟩

lemma *nat-succ-iff* [*iff*]: *succ*(*n*): *nat* <-> *n*: *nat*
⟨*proof*⟩

lemma *nat-le-Limit*: *Limit*(*i*) ==> *nat le i*
⟨*proof*⟩

lemmas *succ-in-naturalD* = *Ord-trans* [*OF succI1 - nat-into-Ord*]

lemma *lt-nat-in-nat*: [| *m*<*n*; *n*: *nat* |] ==> *m*: *nat*

⟨*proof*⟩

lemma *le-in-nat*: [| *m le n*; *n*:*nat* |] ==> *m*:*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*: *nat*; *m*: *nat*;
!!*x*. [| *x*: *nat*; *m le x*; *P*(*x*) |] ==> *P*(*succ*(*x*)) |]
==> *m le n* --> *P*(*m*) --> *P*(*n*)
⟨*proof*⟩

lemma *nat-induct-from*:
[| *m le n*; *m*: *nat*; *n*: *nat*;
P(*m*);
!!*x*. [| *x*: *nat*; *m le x*; *P*(*x*) |] ==> *P*(*succ*(*x*)) |]
==> *P*(*n*)
⟨*proof*⟩

lemma *diff-induct* [*case-names 0 0-succ succ-succ*, *consumes 2*]:
[| *m*: *nat*; *n*: *nat*;
!!*x*. *x*: *nat* ==> *P*(*x*,*0*);
!!*y*. *y*: *nat* ==> *P*(*0*,*succ*(*y*));
!!*x y*. [| *x*: *nat*; *y*: *nat*; *P*(*x*,*y*) |] ==> *P*(*succ*(*x*),*succ*(*y*)) |]
==> *P*(*m*,*n*)
⟨*proof*⟩

lemma *succ-lt-induct-lemma* [*rule-format*]:
m: *nat* ==> *P*(*m*,*succ*(*m*)) --> (*ALL x*: *nat*. *P*(*m*,*x*) --> *P*(*m*,*succ*(*x*)))
-->
(*ALL n*:*nat*. *m*<*n* --> *P*(*m*,*n*))
⟨*proof*⟩

lemma *succ-lt-induct*:
[| *m*<*n*; *n*: *nat*;
P(*m*,*succ*(*m*));

!!x. [| x: nat; $P(m,x)$ |] ==> $P(m,succ(x))$ |]
==> $P(m,n)$
⟨*proof*⟩

15.3 quasinat: to allow a case-split rule for *nat-case*

True if the argument is zero or any successor

lemma [*iff*]: $quasinat(0)$
⟨*proof*⟩

lemma [*iff*]: $quasinat(succ(x))$
⟨*proof*⟩

lemma *nat-imp-quasinat*: $n \in nat$ ==> $quasinat(n)$
⟨*proof*⟩

lemma *non-nat-case*: ~ $quasinat(x)$ ==> $nat\text{-}case(a,b,x) = 0$
⟨*proof*⟩

lemma *nat-cases-disj*: $k=0$ | $(\exists y.\ k = succ(y))$ | ~ $quasinat(k)$
⟨*proof*⟩

lemma *nat-cases*:
[|$k=0$ ==> P; !!y. $k = succ(y)$ ==> P; ~ $quasinat(k)$ ==> P|] ==> P
⟨*proof*⟩

lemma *nat-case-0* [*simp*]: $nat\text{-}case(a,b,0) = a$
⟨*proof*⟩

lemma *nat-case-succ* [*simp*]: $nat\text{-}case(a,b,succ(n)) = b(n)$
⟨*proof*⟩

lemma *nat-case-type* [*TC*]:
[| n: nat; a: $C(0)$; !!m. m: nat ==> $b(m)$: $C(succ(m))$ |]
==> $nat\text{-}case(a,b,n)$: $C(n)$
⟨*proof*⟩

lemma *split-nat-case*:
$P(nat\text{-}case(a,b,k))$ <->
$((k=0$ --> $P(a))$ & $(\forall x.\ k=succ(x)$ --> $P(b(x)))$ & (~ $quasinat(k)$ ⟶ $P(0)))$
⟨*proof*⟩

15.4 Recursion on the Natural Numbers

lemma *nat-rec-0*: $nat\text{-}rec(0,a,b) = a$
⟨*proof*⟩

lemma *nat-rec-succ*: m: nat ==> $nat\text{-}rec(succ(m),a,b) = b(m,\ nat\text{-}rec(m,a,b))$
⟨*proof*⟩

lemma *Un-nat-type* [*TC*]: [| i: nat; j: nat |] ==> $i\ Un\ j$: nat
⟨*proof*⟩

lemma *Int-nat-type* [*TC*]: [| i: nat; j: nat |] ==> $i\ Int\ j$: nat
⟨*proof*⟩

lemma *nat-nonempty* [*simp*]: $nat\ ^{\sim}= 0$
⟨*proof*⟩

A natural number is the set of its predecessors

lemma *nat-eq-Collect-lt*: $i \in nat$ ==> $\{j{\in}nat.\ j{<}i\} = i$
⟨*proof*⟩

lemma *Le-iff* [*iff*]: $<x,y> : Le <-> x\ le\ y\ \&\ x : nat\ \&\ y : nat$
⟨*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*) == ⋃*n*∈*nat*. *nat-rec*(*n*, *A*, %*m r*. *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*. ⋃*y*∈*x*. *succ*(*f*‘*y*))

definition
  transrec2 :: [*i*, *i*, [*i*,*i*]=>*i*] =>*i* **where**
    transrec2(*k*, *a*, *b*) ==
      transrec(*k*,
            %*i r*. *if*(*i*=*0*, *a*,
                  if(*EX j*. *i*=*succ*(*j*),
                    b(*THE j*. *i*=*succ*(*j*), *r*‘(*THE j*. *i*=*succ*(*j*))),
                    ⋃*j*<*i*. *r*‘*j*))))

definition
  recursor :: [*i*, [*i*,*i*]=>*i*, *i*]=>*i* **where**
    recursor(*a*,*b*,*k*) == *transrec*(*k*, %*n f*. *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* <= *eclose*(*A*)
⟨*proof*⟩

lemmas *arg-into-eclose* = *arg-subset-eclose* [*THEN subsetD*, *standard*]

lemma *Transset-eclose*: *Transset*(*eclose*(*A*))
⟨*proof*⟩

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*:
[| !!x. *ALL* $y{:}x$. $P(y)$ ==> $P(x)$ |] ==> $P(a)$
⟨*proof*⟩

17.2 Leastness of *eclose*

lemma *eclose-least-lemma*:
[| *Transset*(X); A<=X; n: *nat* |] ==> *nat-rec*(n, A, %m r. *Union*(r)) <= X
⟨*proof*⟩

lemma *eclose-least*:
[| *Transset*(X); A<=X |] ==> *eclose*(A) <= X
⟨*proof*⟩

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*: *Transset*(X) ==> *eclose*(X) = X
⟨*proof*⟩

A transitive set either is empty or contains the empty set.

lemma *Transset-0-lemma* [*rule-format*]: *Transset*(A) ==> x∈A --> 0∈A
⟨*proof*⟩

lemma *Transset-0-disj*: *Transset*(A) ==> A=0 | 0∈A
⟨*proof*⟩

17.3 Epsilon Recursion

lemma *mem-eclose-trans*: [| A: *eclose*(B); B: *eclose*(C) |] ==> A: *eclose*(C)

⟨*proof*⟩

lemma *mem-eclose-sing-trans*:
[| A: *eclose*({B}); B: *eclose*({C}) |] ==> A: *eclose*({C})
⟨*proof*⟩

lemma *under-Memrel*: [| *Transset*(i); j:i |] ==> *Memrel*(i)−‘‘{j} = j
⟨*proof*⟩

lemma *lt-Memrel*: $j < i$ ==> *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:*eclose*({j}); j:*eclose*({i}) |] ==>
wfrec(*Memrel*(*eclose*({i})), k, H) = *wfrec*(*Memrel*(*eclose*({j})), k, H)
⟨*proof*⟩

lemma *wfrec-eclose-eq2*:
k: i ==> *wfrec*(*Memrel*(*eclose*({i})),k,H) = *wfrec*(*Memrel*(*eclose*({k})),k,H)
⟨*proof*⟩

lemma *transrec*: *transrec*(a,H) = H(a, *lam* x:a. *transrec*(x,H))
⟨*proof*⟩

lemma *def-transrec*:
[| !!x. f(x)==*transrec*(x,H) |] ==> f(a) = H(a, *lam* x:a. f(x))
⟨*proof*⟩

lemma *transrec-type*:
[| !!x u. [| x:*eclose*({a}); u: *Pi*(x,B) |] ==> H(x,u) : B(x) |]
==> *transrec*(a,H) : B(a)
⟨*proof*⟩

lemma *eclose-sing-Ord*: *Ord*(i) ==> *eclose*({i}) <= *succ*(i)
⟨*proof*⟩

lemma *succ-subset-eclose-sing*: *succ*(i) <= *eclose*({i})
⟨*proof*⟩

lemma *eclose-sing-Ord-eq*: *Ord*(i) ==> *eclose*({i}) = *succ*(i)
⟨*proof*⟩

lemma *Ord-transrec-type*:
assumes *jini*: j: i
and *ordi*: $Ord(i)$
and *minor*: $!!x\ u.\ [|\ x{:}\ i;\ \ u{:}\ Pi(x,B)\ |] ==> H(x,u) : B(x)$
shows $transrec(j,H) : B(j)$
⟨*proof*⟩

17.4 Rank

lemma *rank*: $rank(a) = (\bigcup y{\in}a.\ succ(rank(y)))$
⟨*proof*⟩

lemma *Ord-rank* [*simp*]: $Ord(rank(a))$
⟨*proof*⟩

lemma *rank-of-Ord*: $Ord(i) ==> rank(i) = i$
⟨*proof*⟩

lemma *rank-lt*: $a{:}b ==> rank(a) < rank(b)$
⟨*proof*⟩

lemma *eclose-rank-lt*: $a{:}\ eclose(b) ==> rank(a) < rank(b)$
⟨*proof*⟩

lemma *rank-mono*: $a<=b ==> rank(a)\ le\ rank(b)$
⟨*proof*⟩

lemma *rank-Pow*: $rank(Pow(a)) = succ(rank(a))$
⟨*proof*⟩

lemma *rank-0* [*simp*]: $rank(0) = 0$
⟨*proof*⟩

lemma *rank-succ* [*simp*]: $rank(succ(x)) = succ(rank(x))$
⟨*proof*⟩

lemma *rank-Union*: $rank(\ Union(A)) = (\bigcup x{\in}A.\ rank(x))$
⟨*proof*⟩

lemma *rank-eclose*: $rank(eclose(a)) = rank(a)$
⟨*proof*⟩

lemma *rank-pair1*: $rank(a) < rank(<a,b>)$
⟨*proof*⟩

lemma *rank-pair2*: $rank(b) < rank(<a,b>)$
⟨*proof*⟩

lemma *the-equality-if*:
$P(a) ==> (THE\ x.\ P(x)) = (if\ (EX!x.\ P(x))\ then\ a\ else\ 0)$
⟨*proof*⟩

lemma *rank-apply*: $[|i : domain(f); function(f)|] ==> rank(f`i) < rank(f)$
⟨*proof*⟩

17.5 Corollaries of Leastness

lemma *mem-eclose-subset*: $A:B ==> eclose(A)<=eclose(B)$
⟨*proof*⟩

lemma *eclose-mono*: $A<=B ==> eclose(A) <= eclose(B)$
⟨*proof*⟩

lemma *eclose-idem*: $eclose(eclose(A)) = eclose(A)$
⟨*proof*⟩

lemma *transrec2-0* [*simp*]: $transrec2(0,a,b) = a$
⟨*proof*⟩

lemma *transrec2-succ* [*simp*]: $transrec2(succ(i),a,b) = b(i,\ transrec2(i,a,b))$
⟨*proof*⟩

lemma *transrec2-Limit*:
$Limit(i) ==> transrec2(i,a,b) = (\bigcup j<i.\ transrec2(j,a,b))$
⟨*proof*⟩

lemma *def-transrec2*:
$(!!x.\ f(x)==transrec2(x,a,b))$
$==> f(0) = a\ \&$
$f(succ(i)) = b(i,\ f(i))\ \&$
$(Limit(K) --> f(K) = (\bigcup j<K.\ f(j)))$
⟨*proof*⟩

lemmas *recursor-lemma* = *recursor-def* [*THEN def-transrec*, *THEN trans*]

lemma *recursor-0*: $recursor(a,b,0) = a$
⟨*proof*⟩

lemma *recursor-succ*: $recursor(a,b,succ(m)) = b(m,\ recursor(a,b,m))$
⟨*proof*⟩

lemma *rec-0* [*simp*]: $rec(0,a,b) = a$
⟨*proof*⟩

lemma *rec-succ* [*simp*]: $rec(succ(m),a,b) = b(m,\ rec(m,a,b))$
⟨*proof*⟩

lemma *rec-type*:
$[|\ n{:}\ nat;$
$a{:}\ C(0);$
$!!m\ z.\ [|\ m{:}\ nat;\ \ z{:}\ C(m)\ |] ==> b(m,z){:}\ C(succ(m))\ |]$
$==> rec(n,a,b) : C(n)$
⟨*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$(A,r) == (ALL\ x{:}A.\ ALL\ y{:}A.\ <x,y>{:}r \mid x=y \mid <y,x>{:}r)$

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$(A,r,B,s) ==$
$\{f$: $A->B$. *ALL* x:A. *ALL* y:A. $<x,y>$:r $-->$ $<f`x,f`y>$:$s\}$

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. $<x,y>$:r $<->$ $<f`x,f`y>$:$s\}$

definition
pred :: $[i,i,i]=>i$ **where**
pred$(A,x,r) == \{y$:A. $<y,x>$:$r\}$

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). $\{<x,y>\}$

definition
first :: $[i, i, i] => o$ **where**
first$(u, X, R) == u$:X & $(ALL\ v$:$X.\ v^{\sim}=u\ -->\ <u,v> : R)$

notation (*xsymbols*)
ord-iso $((\langle$-, -$\rangle \cong/ \langle$-, -$\rangle)$ *51*)

18.1 Immediate Consequences of the Definitions

lemma *part-ord-Imp-asym*:
part-ord$(A,r) ==>$ *asym*$(r$ *Int* $A*A)$
⟨*proof*⟩

lemma *linearE*:
[| *linear*(A,r); x:A; y:A;
$<x,y>$:$r ==> P$; $x=y ==> P$; $<y,x>$:$r ==> P$ |]
$==> P$
⟨*proof*⟩

lemma *well-ordI*:
[| *wf*[*A*](*r*); *linear*(*A*,*r*) |] ==> *well-ord*(*A*,*r*)
⟨*proof*⟩

lemma *well-ord-is-wf*:
well-ord(*A*,*r*) ==> *wf*[*A*](*r*)
⟨*proof*⟩

lemma *well-ord-is-trans-on*:
well-ord(*A*,*r*) ==> *trans*[*A*](*r*)
⟨*proof*⟩

lemma *well-ord-is-linear*: *well-ord*(*A*,*r*) ==> *linear*(*A*,*r*)
⟨*proof*⟩

lemma *pred-iff*: *y* : *pred*(*A*,*x*,*r*) <-> <*y*,*x*>:*r* & *y*:*A*
⟨*proof*⟩

lemmas *predI* = *conjI* [*THEN pred-iff* [*THEN iffD2*]]

lemma *predE*: [| *y*: *pred*(*A*,*x*,*r*); [| *y*:*A*; <*y*,*x*>:*r* |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *pred-subset-under*: *pred*(*A*,*x*,*r*) <= *r* -`` {*x*}
⟨*proof*⟩

lemma *pred-subset*: *pred*(*A*,*x*,*r*) <= *A*
⟨*proof*⟩

lemma *pred-pred-eq*:
pred(*pred*(*A*,*x*,*r*), *y*, *r*) = *pred*(*A*,*x*,*r*) *Int* *pred*(*A*,*y*,*r*)
⟨*proof*⟩

lemma *trans-pred-pred-eq*:
[| *trans*[*A*](*r*); <*y*,*x*>:*r*; *x*:*A*; *y*:*A* |]
==> *pred*(*pred*(*A*,*x*,*r*), *y*, *r*) = *pred*(*A*,*y*,*r*)
⟨*proof*⟩

18.2 Restricting an Ordering's Domain

lemma *part-ord-subset*:
[| *part-ord*(*A*,*r*); *B*<=*A* |] ==> *part-ord*(*B*,*r*)

⟨*proof*⟩

lemma *linear-subset*:
[| *linear*(*A*,*r*); *B*<=*A* |] ==> *linear*(*B*,*r*)
⟨*proof*⟩

lemma *tot-ord-subset*:
[| *tot-ord*(*A*,*r*); *B*<=*A* |] ==> *tot-ord*(*B*,*r*)
⟨*proof*⟩

lemma *well-ord-subset*:
[| *well-ord*(*A*,*r*); *B*<=*A* |] ==> *well-ord*(*B*,*r*)
⟨*proof*⟩

lemma *irrefl-Int-iff*: *irrefl*(*A*,*r Int A*∗*A*) <−> *irrefl*(*A*,*r*)
⟨*proof*⟩

lemma *trans-on-Int-iff*: *trans*[*A*](*r Int A*∗*A*) <−> *trans*[*A*](*r*)
⟨*proof*⟩

lemma *part-ord-Int-iff*: *part-ord*(*A*,*r Int A*∗*A*) <−> *part-ord*(*A*,*r*)
⟨*proof*⟩

lemma *linear-Int-iff*: *linear*(*A*,*r Int A*∗*A*) <−> *linear*(*A*,*r*)
⟨*proof*⟩

lemma *tot-ord-Int-iff*: *tot-ord*(*A*,*r Int A*∗*A*) <−> *tot-ord*(*A*,*r*)
⟨*proof*⟩

lemma *wf-on-Int-iff*: *wf*[*A*](*r Int A*∗*A*) <−> *wf*[*A*](*r*)
⟨*proof*⟩

lemma *well-ord-Int-iff*: *well-ord*(*A*,*r Int A*∗*A*) <−> *well-ord*(*A*,*r*)
⟨*proof*⟩

18.3 Empty and Unit Domains

lemma *wf-on-any-0*: *wf*[*A*](*0*)
⟨*proof*⟩

18.3.1 Relations over the Empty Set

lemma *irrefl-0*: *irrefl*(*0*,*r*)
⟨*proof*⟩

lemma *trans-on-0*: *trans*[*0*](*r*)
⟨*proof*⟩

lemma *part-ord-0*: *part-ord*(*0*,*r*)
⟨*proof*⟩

lemma *linear-0*: *linear*(*0*,*r*)
⟨*proof*⟩

lemma *tot-ord-0*: *tot-ord*(*0*,*r*)
⟨*proof*⟩

lemma *wf-on-0*: *wf*[*0*](*r*)
⟨*proof*⟩

lemma *well-ord-0*: *well-ord*(*0*,*r*)
⟨*proof*⟩

18.3.2 The Empty Relation Well-Orders the Unit Set

by Grabczewski

lemma *tot-ord-unit*: *tot-ord*({*a*},*0*)
⟨*proof*⟩

lemma *well-ord-unit*: *well-ord*({*a*},*0*)
⟨*proof*⟩

18.4 Order-Isomorphisms

Suppes calls them "similarities"

lemma *mono-map-is-fun*: *f*: *mono-map*(*A*,*r*,*B*,*s*) ==> *f*: *A*->*B*
⟨*proof*⟩

lemma *mono-map-is-inj*:
[| *linear*(*A*,*r*); *wf*[*B*](*s*); *f*: *mono-map*(*A*,*r*,*B*,*s*) |] ==> *f*: *inj*(*A*,*B*)
⟨*proof*⟩

lemma *ord-isoI*:
[| *f*: *bij*(*A*, *B*);
!!*x y*. [| *x*:*A*; *y*:*A* |] ==> <*x*, *y*> : *r* <-> <*f*‘*x*, *f*‘*y*> : *s* |]
==> *f*: *ord-iso*(*A*,*r*,*B*,*s*)
⟨*proof*⟩

lemma *ord-iso-is-mono-map*:
f: *ord-iso*(*A*,*r*,*B*,*s*) ==> *f*: *mono-map*(*A*,*r*,*B*,*s*)
⟨*proof*⟩

lemma *ord-iso-is-bij*:
f: *ord-iso*(*A*,*r*,*B*,*s*) ==> *f*: *bij*(*A*,*B*)
⟨*proof*⟩

lemma *ord-iso-apply*:
[| f: *ord-iso*(A,r,B,s); <x,y>: r; x:A; y:A |] ==> <f‘x, f‘y> : s
⟨*proof*⟩

lemma *ord-iso-converse*:
[| f: *ord-iso*(A,r,B,s); <x,y>: s; x:B; y:B |]
==> <*converse*(f) ‘ x, *converse*(f) ‘ y> : r
⟨*proof*⟩

lemma *ord-iso-refl*: *id*(A): *ord-iso*(A,r,A,r)
⟨*proof*⟩

lemma *ord-iso-sym*: f: *ord-iso*(A,r,B,s) ==> *converse*(f): *ord-iso*(B,s,A,r)
⟨*proof*⟩

lemma *mono-map-trans*:
[| g: *mono-map*(A,r,B,s); f: *mono-map*(B,s,C,t) |]
==> (f *O* g): *mono-map*(A,r,C,t)
⟨*proof*⟩

lemma *ord-iso-trans*:
[| g: *ord-iso*(A,r,B,s); f: *ord-iso*(B,s,C,t) |]
==> (f *O* g): *ord-iso*(A,r,C,t)
⟨*proof*⟩

lemma *mono-ord-isoI*:
[| f: *mono-map*(A,r,B,s); g: *mono-map*(B,s,A,r);
f *O* g = *id*(B); g *O* f = *id*(A) |] ==> f: *ord-iso*(A,r,B,s)
⟨*proof*⟩

lemma *well-ord-mono-ord-isoI*:
[| *well-ord*(A,r); *well-ord*(B,s);
f: *mono-map*(A,r,B,s); *converse*(f): *mono-map*(B,s,A,r) |]
==> f: *ord-iso*(A,r,B,s)
⟨*proof*⟩

lemma *part-ord-ord-iso*:
[| *part-ord*(*B*,*s*); *f*: *ord-iso*(*A*,*r*,*B*,*s*) |] ==> *part-ord*(*A*,*r*)
⟨*proof*⟩

lemma *linear-ord-iso*:
[| *linear*(*B*,*s*); *f*: *ord-iso*(*A*,*r*,*B*,*s*) |] ==> *linear*(*A*,*r*)
⟨*proof*⟩

lemma *wf-on-ord-iso*:
[| *wf*[*B*](*s*); *f*: *ord-iso*(*A*,*r*,*B*,*s*) |] ==> *wf*[*A*](*r*)
⟨*proof*⟩

lemma *well-ord-ord-iso*:
[| *well-ord*(*B*,*s*); *f*: *ord-iso*(*A*,*r*,*B*,*s*) |] ==> *well-ord*(*A*,*r*)
⟨*proof*⟩

18.5 Main results of Kunen, Chapter 1 section 6

lemma *well-ord-iso-subset-lemma*:
[| *well-ord*(*A*,*r*); *f*: *ord-iso*(*A*,*r*, *A*′,*r*); *A*′<= *A*; *y*: *A* |]
==> ~ <*f*‘*y*, *y*>: *r*
⟨*proof*⟩

lemma *well-ord-iso-predE*:
[| *well-ord*(*A*,*r*); *f* : *ord-iso*(*A*, *r*, *pred*(*A*,*x*,*r*), *r*); *x*:*A* |] ==> *P*
⟨*proof*⟩

lemma *well-ord-iso-pred-eq*:
[| *well-ord*(*A*,*r*); *f* : *ord-iso*(*pred*(*A*,*a*,*r*), *r*, *pred*(*A*,*c*,*r*), *r*);
a:*A*; *c*:*A* |] ==> *a*=*c*
⟨*proof*⟩

lemma *ord-iso-image-pred*:
[|*f* : *ord-iso*(*A*,*r*,*B*,*s*); *a*:*A*|] ==> *f* `` *pred*(*A*,*a*,*r*) = *pred*(*B*, *f*‘*a*, *s*)
⟨*proof*⟩

lemma *ord-iso-restrict-image*:
[| *f* : *ord-iso*(*A*,*r*,*B*,*s*); *C*<=*A* |]
==> *restrict*(*f*,*C*) : *ord-iso*(*C*, *r*, *f*``*C*, *s*)
⟨*proof*⟩

lemma *ord-iso-restrict-pred*:
[| *f* : *ord-iso*(*A*,*r*,*B*,*s*); *a*:*A* |]
==> *restrict*(*f*, *pred*(*A*,*a*,*r*)) : *ord-iso*(*pred*(*A*,*a*,*r*), *r*, *pred*(*B*, *f*‘*a*, *s*), *s*)

⟨*proof*⟩

lemma *well-ord-iso-preserving*:
[| *well-ord*(*A*,*r*); *well-ord*(*B*,*s*); <*a*,*c*>: *r*;
f : *ord-iso*(*pred*(*A*,*a*,*r*), *r*, *pred*(*B*,*b*,*s*), *s*);
g : *ord-iso*(*pred*(*A*,*c*,*r*), *r*, *pred*(*B*,*d*,*s*), *s*);
a:*A*; *c*:*A*; *b*:*B*; *d*:*B* |] ==> <*b*,*d*>: *s*
⟨*proof*⟩

lemma *well-ord-iso-unique-lemma*:
[| *well-ord*(*A*,*r*);
f: *ord-iso*(*A*,*r*, *B*,*s*); *g*: *ord-iso*(*A*,*r*, *B*,*s*); *y*: *A* |]
==> ~ <*g*‘*y*, *f*‘*y*> : *s*
⟨*proof*⟩

lemma *well-ord-iso-unique*: [| *well-ord*(*A*,*r*);
f: *ord-iso*(*A*,*r*, *B*,*s*); *g*: *ord-iso*(*A*,*r*, *B*,*s*) |] ==> *f* = *g*
⟨*proof*⟩

18.6 Towards Kunen's Theorem 6.3: Linearity of the Similarity Relation

lemma *ord-iso-map-subset*: *ord-iso-map*(*A*,*r*,*B*,*s*) <= *A*∗*B*
⟨*proof*⟩

lemma *domain-ord-iso-map*: *domain*(*ord-iso-map*(*A*,*r*,*B*,*s*)) <= *A*
⟨*proof*⟩

lemma *range-ord-iso-map*: *range*(*ord-iso-map*(*A*,*r*,*B*,*s*)) <= *B*
⟨*proof*⟩

lemma *converse-ord-iso-map*:
converse(*ord-iso-map*(*A*,*r*,*B*,*s*)) = *ord-iso-map*(*B*,*s*,*A*,*r*)
⟨*proof*⟩

lemma *function-ord-iso-map*:
well-ord(*B*,*s*) ==> *function*(*ord-iso-map*(*A*,*r*,*B*,*s*))
⟨*proof*⟩

lemma *ord-iso-map-fun*: *well-ord*(*B*,*s*) ==> *ord-iso-map*(*A*,*r*,*B*,*s*)
: *domain*(*ord-iso-map*(*A*,*r*,*B*,*s*)) -> *range*(*ord-iso-map*(*A*,*r*,*B*,*s*))
⟨*proof*⟩

lemma *ord-iso-map-mono-map*:
[| *well-ord*(*A*,*r*); *well-ord*(*B*,*s*) |]

$==>$ *ord-iso-map*(A,r,B,s)
: *mono-map*(*domain*(*ord-iso-map*(A,r,B,s)), r,
range(*ord-iso-map*(A,r,B,s)), s)
⟨*proof*⟩

lemma *ord-iso-map-ord-iso*:
[| *well-ord*(A,r); *well-ord*(B,s) |] $==>$ *ord-iso-map*(A,r,B,s)
: *ord-iso*(*domain*(*ord-iso-map*(A,r,B,s)), r,
range(*ord-iso-map*(A,r,B,s)), s)
⟨*proof*⟩

lemma *domain-ord-iso-map-subset*:
[| *well-ord*(A,r); *well-ord*(B,s);
a: A; a ~: *domain*(*ord-iso-map*(A,r,B,s)) |]
$==>$ *domain*(*ord-iso-map*(A,r,B,s)) $<=$ *pred*(A, a, r)
⟨*proof*⟩

lemma *domain-ord-iso-map-cases*:
[| *well-ord*(A,r); *well-ord*(B,s) |]
$==>$ *domain*(*ord-iso-map*(A,r,B,s)) $= A$ |
(*EX* x:A. *domain*(*ord-iso-map*(A,r,B,s)) $=$ *pred*(A,x,r))
⟨*proof*⟩

lemma *range-ord-iso-map-cases*:
[| *well-ord*(A,r); *well-ord*(B,s) |]
$==>$ *range*(*ord-iso-map*(A,r,B,s)) $= B$ |
(*EX* y:B. *range*(*ord-iso-map*(A,r,B,s)) $=$ *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) |]
$==>$ *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*$(A, r,$ *pred*(B,y,s), s))
⟨*proof*⟩

18.7 Miscellaneous Results by Krzysztof Grabczewski

lemma *irrefl-converse*: *irrefl*(A,r) $==>$ *irrefl*$(A,$*converse*$(r))$
⟨*proof*⟩

lemma *trans-on-converse*: *trans*$[A](r)$ $==>$ *trans*$[A]($*converse*$(r))$
⟨*proof*⟩

lemma *part-ord-converse*: *part-ord*(*A*,*r*) ==> *part-ord*(*A*,*converse*(*r*))
⟨*proof*⟩

lemma *linear-converse*: *linear*(*A*,*r*) ==> *linear*(*A*,*converse*(*r*))
⟨*proof*⟩

lemma *tot-ord-converse*: *tot-ord*(*A*,*r*) ==> *tot-ord*(*A*,*converse*(*r*))
⟨*proof*⟩

lemma *first-is-elem*: *first*(*b*,*B*,*r*) ==> *b*:*B*
⟨*proof*⟩

lemma *well-ord-imp-ex1-first*:
[| *well-ord*(*A*,*r*); *B*<=*A*; *B*^~=*0* |] ==> (*EX*! *b*. *first*(*b*,*B*,*r*))
⟨*proof*⟩

lemma *the-first-in*:
[| *well-ord*(*A*,*r*); *B*<=*A*; *B*^~=*0* |] ==> (*THE b*. *first*(*b*,*B*,*r*)) : *B*
⟨*proof*⟩

18.8 Lemmas for the Reflexive Orders

lemma *subset-vimage-vimage-iff*:
[| *Preorder*(*r*); *A* ⊆ *field*(*r*); *B* ⊆ *field*(*r*) |] ==>
r −‘‘ *A* ⊆ *r* −‘‘ *B* <-> (*ALL a*:*A*. *EX b*:*B*. <*a*, *b*> : *r*)
⟨*proof*⟩

lemma *subset-vimage1-vimage1-iff*:
[| *Preorder*(*r*); *a* : *field*(*r*); *b* : *field*(*r*) |] ==>
r −‘‘ {*a*} ⊆ *r* −‘‘ {*b*} <-> <*a*, *b*> : *r*
⟨*proof*⟩

lemma *Refl-antisym-eq-Image1-Image1-iff*:
[| *refl*(*field*(*r*), *r*); *antisym*(*r*); *a* : *field*(*r*); *b* : *field*(*r*) |] ==>
r ‘‘ {*a*} = *r* ‘‘ {*b*} <-> *a* = *b*
⟨*proof*⟩

lemma *Partial-order-eq-Image1-Image1-iff*:
[| *Partial-order*(*r*); *a* : *field*(*r*); *b* : *field*(*r*) |] ==>
r ‘‘ {*a*} = *r* ‘‘ {*b*} <-> *a* = *b*
⟨*proof*⟩

lemma *Refl-antisym-eq-vimage1-vimage1-iff*:
[| *refl*(*field*(*r*), *r*); *antisym*(*r*); *a* : *field*(*r*); *b* : *field*(*r*) |] ==>
r −‘‘ {*a*} = *r* −‘‘ {*b*} <-> *a* = *b*
⟨*proof*⟩

lemma *Partial-order-eq-vimage1-vimage1-iff*:
[| *Partial-order*(*r*); *a* : *field*(*r*); *b* : *field*(*r*) |] ==>
r −‘‘ {*a*} = *r* −‘‘ {*b*} <−> *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*]=>*i* **where**
radd(*A*,*r*,*B*,*s*) ==
{*z*: (*A*+*B*) ∗ (*A*+*B*).
(*EX x y*. *z* = <*Inl*(*x*), *Inr*(*y*)>) |
(*EX x′ x*. *z* = <*Inl*(*x′*), *Inl*(*x*)> & <*x′*,*x*>:*r*) |
(*EX y′ y*. *z* = <*Inr*(*y′*), *Inr*(*y*)> & <*y′*,*y*>:*s*)}

definition

rmult :: [*i*,*i*,*i*,*i*]=>*i* **where**
rmult(*A*,*r*,*B*,*s*) ==
{*z*: (*A*∗*B*) ∗ (*A*∗*B*).
EX x′ y′ x y. *z* = <<*x′*,*y′*>, <*x*,*y*>> &
(<*x′*,*x*>: *r* | (*x′*=*x* & <*y′*,*y*>: *s*))}

definition

rvimage :: [*i*,*i*,*i*]=>*i* **where**
rvimage(*A*,*f*,*r*) == {*z*: *A*∗*A*. *EX x y*. *z* = <*x*,*y*> & <*f*‘*x*,*f*‘*y*>: *r*}

definition
measure :: [*i*, *i*⇒*i*] ⇒ *i* **where**
measure(*A*,*f*) == {<*x*,*y*>: *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*]:
<*Inl*(*a*), *Inr*(*b*)> : *radd*(*A*,*r*,*B*,*s*) <−> *a*:*A* & *b*:*B*
⟨*proof*⟩

lemma *radd-Inl-iff* [*iff*]:

$<Inl(a'),\ Inl(a)> : radd(A,r,B,s) \ <-> \ a':A \ \& \ a:A \ \& \ <a',a>:r$
⟨*proof*⟩

lemma *radd-Inr-iff* [*iff*]:
$<Inr(b'),\ Inr(b)> : radd(A,r,B,s) \ <-> \ b':B \ \& \ b:B \ \& \ <b',b>:s$
⟨*proof*⟩

lemma *radd-Inr-Inl-iff* [*simp*]:
$<Inr(b),\ Inl(a)> : radd(A,r,B,s) \ <-> \ False$
⟨*proof*⟩

declare *radd-Inr-Inl-iff* [*THEN iffD1*, *dest*!]

19.1.2 Elimination Rule

lemma *raddE*:
$[|\ <p',p> : radd(A,r,B,s);$
$\quad !!x\ y.\ [|\ p'=Inl(x);\ x:A;\ p=Inr(y);\ y:B\ |] ==> Q;$
$\quad !!x'\ x.\ [|\ p'=Inl(x');\ p=Inl(x);\ <x',x>:\ r;\ x':A;\ x:A\ |] ==> Q;$
$\quad !!y'\ y.\ [|\ p'=Inr(y');\ p=Inr(y);\ <y',y>:\ s;\ y':B;\ y:B\ |] ==> Q$
$\ |] ==> Q$
⟨*proof*⟩

19.1.3 Type checking

lemma *radd-type*: $radd(A,r,B,s) <= (A+B) * (A+B)$
⟨*proof*⟩

lemmas *field-radd* = *radd-type* [*THEN field-rel-subset*]

19.1.4 Linearity

lemma *linear-radd*:
$[|\ linear(A,r);\ \ linear(B,s)\ |] ==> linear(A+B,radd(A,r,B,s))$
⟨*proof*⟩

19.1.5 Well-foundedness

lemma *wf-on-radd*: $[|\ wf[A](r);\ \ wf[B](s)\ |] ==> wf[A+B](radd(A,r,B,s))$
⟨*proof*⟩

lemma *wf-radd*: $[|\ wf(r);\ \ wf(s)\ |] ==> wf(radd(field(r),r,field(s),s))$
⟨*proof*⟩

lemma *well-ord-radd*:
$[|\ well\text{-}ord(A,r);\ \ well\text{-}ord(B,s)\ |] ==> well\text{-}ord(A+B,\ radd(A,r,B,s))$
⟨*proof*⟩

19.1.6 An *ord-iso* congruence law

lemma *sum-bij*:
[| f: $bij(A,C)$; g: $bij(B,D)$ |]
==> $(lam\ z{:}A+B.\ case(\%x.\ Inl(f`x),\ \%y.\ Inr(g`y),\ z)) : bij(A+B,\ C+D)$
⟨*proof*⟩

lemma *sum-ord-iso-cong*:
[| f: $ord\text{-}iso(A,r,A',r')$; g: $ord\text{-}iso(B,s,B',s')$ |] ==>
$(lam\ z{:}A+B.\ case(\%x.\ Inl(f`x),\ \%y.\ Inr(g`y),\ z))$
: $ord\text{-}iso(A+B,\ radd(A,r,B,s),\ A'+B',\ radd(A',r',B',s'))$
⟨*proof*⟩

lemma *sum-disjoint-bij*: $A\ Int\ B = 0$ ==>
$(lam\ z{:}A+B.\ case(\%x.\ x,\ \%y.\ y,\ z)) : bij(A+B,\ A\ Un\ B)$
⟨*proof*⟩

19.1.7 Associativity

lemma *sum-assoc-bij*:
$(lam\ z{:}(A+B)+C.\ case(case(Inl,\ \%y.\ Inr(Inl(y))),\ \%y.\ Inr(Inr(y)),\ z))$
: $bij((A+B)+C,\ A+(B+C))$
⟨*proof*⟩

lemma *sum-assoc-ord-iso*:
$(lam\ z{:}(A+B)+C.\ case(case(Inl,\ \%y.\ Inr(Inl(y))),\ \%y.\ Inr(Inr(y)),\ z))$
: $ord\text{-}iso((A+B)+C,\ radd(A+B,\ radd(A,r,B,s),\ C,\ t),$
$A+(B+C),\ radd(A,\ r,\ B+C,\ radd(B,s,C,t)))$
⟨*proof*⟩

19.2 Multiplication of Relations – Lexicographic Product

19.2.1 Rewrite rule. Can be used to obtain introduction rules

lemma *rmult-iff* [*iff*]:
$<<a',b'>,<a,b>> : rmult(A,r,B,s)$ <->
$(<a',a>: r\ \&\ a'{:}A\ \&\ a{:}A\ \&\ b'{:}\ B\ \&\ b{:}\ B)\ |$
$(<b',b>: s\ \&\ a'{=}a\ \&\ a{:}A\ \&\ b'{:}\ B\ \&\ b{:}\ B)$

⟨*proof*⟩

lemma *rmultE*:
[| $<<a',b'>,<a,b>> : rmult(A,r,B,s)$;
[| $<a',a>: r$; $a'{:}A$; $a{:}A$; $b'{:}B$; $b{:}B$ |] ==> Q;
[| $<b',b>: s$; $a{:}A$; $a'{=}a$; $b'{:}B$; $b{:}B$ |] ==> Q
|] ==> Q
⟨*proof*⟩

19.2.2 Type checking

lemma *rmult-type*: $rmult(A,r,B,s) <= (A*B) * (A*B)$
⟨*proof*⟩

lemmas *field-rmult* = *rmult-type* [*THEN field-rel-subset*]

19.2.3 Linearity

lemma *linear-rmult*:
$[| linear(A,r);\ \ linear(B,s) |] ==> linear(A*B,rmult(A,r,B,s))$
⟨*proof*⟩

19.2.4 Well-foundedness

lemma *wf-on-rmult*: $[| wf[A](r);\ \ wf[B](s) |] ==> wf[A*B](rmult(A,r,B,s))$
⟨*proof*⟩

lemma *wf-rmult*: $[| wf(r);\ \ wf(s) |] ==> wf(rmult(field(r),r,field(s),s))$
⟨*proof*⟩

lemma *well-ord-rmult*:
$[| well\text{-}ord(A,r);\ \ well\text{-}ord(B,s) |] ==> well\text{-}ord(A*B,\ rmult(A,r,B,s))$
⟨*proof*⟩

19.2.5 An *ord-iso* congruence law

lemma *prod-bij*:
$[| f: bij(A,C);\ \ g: bij(B,D) |]$
$==> (lam\ <x,y>:A*B.\ <f`x,\ g`y>) : bij(A*B,\ C*D)$
⟨*proof*⟩

lemma *prod-ord-iso-cong*:
$[| f: ord\text{-}iso(A,r,A',r');\ \ g: ord\text{-}iso(B,s,B',s') |]$
$==> (lam\ <x,y>:A*B.\ <f`x,\ g`y>)$
$: ord\text{-}iso(A*B,\ rmult(A,r,B,s),\ A'*B',\ rmult(A',r',B',s'))$
⟨*proof*⟩

lemma *singleton-prod-bij*: $(lam\ z:A.\ <x,z>) : bij(A,\ \{x\}*A)$
⟨*proof*⟩

lemma *singleton-prod-ord-iso*:
$well\text{-}ord(\{x\},xr) ==>$
$(lam\ z:A.\ <x,z>) : ord\text{-}iso(A,\ r,\ \{x\}*A,\ rmult(\{x\},\ xr,\ A,\ r))$
⟨*proof*⟩

lemma *prod-sum-singleton-bij*:

$a^{\sim}$:C ==>
(*lam* x:$C*B + D$. *case*(%x. x, %y.<a,y>, x))
: *bij*($C*B + D$, $C*B$ *Un* {a}$*D$)
⟨*proof*⟩

lemma *prod-sum-singleton-ord-iso*:
[| a:A; *well-ord*(A,r) |] ==>
(*lam* x:*pred*(A,a,r)$*B + pred*($B$,$b$,$s$). *case*(%$x$. x, %y.<a,y>, x))
: *ord-iso*(*pred*(A,a,r)$*B + pred*($B$,$b$,$s$),
radd($A*B$, *rmult*(A,r,B,s), B, s),
pred(A,a,r)$*B$ *Un* {a}$*pred*($B$,$b$,$s$), *rmult*($A$,$r$,$B$,$s$))
⟨*proof*⟩

19.2.6 Distributive law

lemma *sum-prod-distrib-bij*:
(*lam* <x,z>:(A+B)$*C$. *case*(%y. *Inl*(<y,z>), %y. *Inr*(<y,z>), x))
: *bij*((A+B)$*C$, ($A*C$)+($B*C$))
⟨*proof*⟩

lemma *sum-prod-distrib-ord-iso*:
(*lam* <x,z>:(A+B)$*C$. *case*(%y. *Inl*(<y,z>), %y. *Inr*(<y,z>), x))
: *ord-iso*((A+B)$*C$, *rmult*(A+B, *radd*(A,r,B,s), C, t),
($A*C$)+($B*C$), *radd*($A*C$, *rmult*(A,r,C,t), $B*C$, *rmult*(B,s,C,t)))
⟨*proof*⟩

19.2.7 Associativity

lemma *prod-assoc-bij*:
(*lam* <<x,y>, z>:($A*B$)$*C$. <x,<y,z>>) : *bij*(($A*B$)$*C$, $A*(B*C)$)
⟨*proof*⟩

lemma *prod-assoc-ord-iso*:
(*lam* <<x,y>, z>:($A*B$)$*C$. <x,<y,z>>)
: *ord-iso*(($A*B$)$*C$, *rmult*($A*B$, *rmult*(A,r,B,s), C, t),
$A*(B*C)$, *rmult*(A, r, $B*C$, *rmult*(B,s,C,t)))
⟨*proof*⟩

19.3 Inverse Image of a Relation

19.3.1 Rewrite rule

lemma *rvimage-iff*: <a,b> : *rvimage*(A,f,r) <-> <f`$a$,$f$`b>: r & a:A & b:A
⟨*proof*⟩

19.3.2 Type checking

lemma *rvimage-type*: *rvimage*(A,f,r) <= $A*A$
⟨*proof*⟩

lemmas *field-rvimage* = *rvimage-type* [*THEN field-rel-subset*]

lemma *rvimage-converse*: *rvimage*(*A*,*f*, *converse*(*r*)) = *converse*(*rvimage*(*A*,*f*,*r*))
⟨*proof*⟩

19.3.3 Partial Ordering Properties

lemma *irrefl-rvimage*:
[| *f*: *inj*(*A*,*B*); *irrefl*(*B*,*r*) |] ==> *irrefl*(*A*, *rvimage*(*A*,*f*,*r*))
⟨*proof*⟩

lemma *trans-on-rvimage*:
[| *f*: *inj*(*A*,*B*); *trans*[*B*](*r*) |] ==> *trans*[*A*](*rvimage*(*A*,*f*,*r*))
⟨*proof*⟩

lemma *part-ord-rvimage*:
[| *f*: *inj*(*A*,*B*); *part-ord*(*B*,*r*) |] ==> *part-ord*(*A*, *rvimage*(*A*,*f*,*r*))
⟨*proof*⟩

19.3.4 Linearity

lemma *linear-rvimage*:
[| *f*: *inj*(*A*,*B*); *linear*(*B*,*r*) |] ==> *linear*(*A*,*rvimage*(*A*,*f*,*r*))
⟨*proof*⟩

lemma *tot-ord-rvimage*:
[| *f*: *inj*(*A*,*B*); *tot-ord*(*B*,*r*) |] ==> *tot-ord*(*A*, *rvimage*(*A*,*f*,*r*))
⟨*proof*⟩

19.3.5 Well-foundedness

lemma *wf-rvimage* [*intro*!]: *wf*(*r*) ==> *wf*(*rvimage*(*A*,*f*,*r*))
⟨*proof*⟩

But note that the combination of *wf-imp-wf-on* and *wf-rvimage* gives *wf*(*r*) ⟹ *wf*[*C*](*rvimage*(*A*, *f*, *r*))

lemma *wf-on-rvimage*: [| *f*: *A*->*B*; *wf*[*B*](*r*) |] ==> *wf*[*A*](*rvimage*(*A*,*f*,*r*))
⟨*proof*⟩

lemma *well-ord-rvimage*:
[| *f*: *inj*(*A*,*B*); *well-ord*(*B*,*r*) |] ==> *well-ord*(*A*, *rvimage*(*A*,*f*,*r*))
⟨*proof*⟩

lemma *ord-iso-rvimage*:
f: *bij*(*A*,*B*) ==> *f*: *ord-iso*(*A*, *rvimage*(*A*,*f*,*s*), *B*, *s*)
⟨*proof*⟩

lemma *ord-iso-rvimage-eq*:
f: *ord-iso*(*A*,*r*, *B*,*s*) ==> *rvimage*(*A*,*f*,*s*) = *r Int A*∗*A*

⟨*proof*⟩

19.4 Every well-founded relation is a subset of some inverse image of an ordinal

lemma *wf-rvimage-Ord*: $Ord(i) \Longrightarrow wf(rvimage(A, f, Memrel(i)))$
⟨*proof*⟩

definition
$wfrank :: [i,i]=>i$ **where**
$wfrank(r,a) == wfrec(r, a, \%x\, f.\, \bigcup y \in r-``\{x\}.\, succ(f`y))$

definition
$wftype :: i=>i$ **where**
$wftype(r) == \bigcup y \in range(r).\, succ(wfrank(r,y))$

lemma *wfrank*: $wf(r) ==> wfrank(r,a) = (\bigcup y \in r-``\{a\}.\, succ(wfrank(r,y)))$
⟨*proof*⟩

lemma *Ord-wfrank*: $wf(r) ==> Ord(wfrank(r,a))$
⟨*proof*⟩

lemma *wfrank-lt*: $[|wf(r); <a,b> \in r|] ==> wfrank(r,a) < wfrank(r,b)$
⟨*proof*⟩

lemma *Ord-wftype*: $wf(r) ==> Ord(wftype(r))$
⟨*proof*⟩

lemma *wftypeI*: $[\![wf(r);\ \ x \in field(r)]\!] \Longrightarrow wfrank(r,x) \in wftype(r)$
⟨*proof*⟩

lemma *wf-imp-subset-rvimage*:
$[|wf(r); r \subseteq A*A|] ==> \exists i\, f.\, Ord(i)\ \&\ r <= rvimage(A, f, Memrel(i))$
⟨*proof*⟩

theorem *wf-iff-subset-rvimage*:
$relation(r) ==> wf(r) <-> (\exists i\, f\, A.\, Ord(i)\ \&\ r <= rvimage(A, f, Memrel(i)))$
⟨*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 *Ordf*: !!*x*. *x* ∈ *A* ==> *Ord*(*f*(*x*))
  and *inj*: !!*x y*. [|*x* ∈ *A*; *y* ∈ *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 *Ordf*: !!*x*. *x* ∈ *A* ==> *Ord*(*f*(*x*))
  and *inj*: !!*x y*. [|*x* ∈ *A*; *y* ∈ *A*; *f*(*x*) = *f*(*y*) |] ==> *x*=*y*
 shows *well-ord*(*A*, *measure*(*A*,*f*))
⟨*proof*⟩

lemma *measure-type*: *measure*(*A*,*f*) <= *A*∗*A*
⟨*proof*⟩

19.5.3 Well-foundedness of Unions

lemma *wf-on-Union*:
 assumes *wfA*: *wf*[*A*](*r*)
  and *wfB*: !!*a*. *a*∈*A* ==> *wf*[*B*(*a*)](*s*)
  and *ok*: !!*a u v*. [|<*u*,*v*> ∈ *s*; *v* ∈ *B*(*a*); *a* ∈ *A*|]
        ==> (∃ *a′*∈*A*. <*a′*,*a*> ∈ *r* & *u* ∈ *B*(*a′*)) | *u* ∈ *B*(*a*)

shows $wf[\bigcup a{\in}A.\ B(a)](s)$
⟨*proof*⟩

19.5.4 Bijections involving Powersets

lemma *Pow-sum-bij*:
$(\lambda Z \in Pow(A+B).\ <\{x \in A.\ Inl(x) \in Z\},\ \{y \in B.\ Inr(y) \in Z\}>)$
$\in bij(Pow(A+B),\ Pow(A)*Pow(B))$
⟨*proof*⟩

As a special case, we have $bij(Pow(A \times B),\ A \to Pow(B))$

lemma *Pow-Sigma-bij*:
$(\lambda r \in Pow(Sigma(A,B)).\ \lambda x \in A.\ r``\{x\})$
$\in bij(Pow(Sigma(A,B)),\ \Pi\ x \in A.\ Pow(B(x)))$
⟨*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 :: $[i,i]=>i$ **where**
$ordermap(A,r) == lam\ x{:}A.\ wfrec[A](r,\ x,\ \%x\ f.\ f\ ``\ pred(A,x,r))$

definition
ordertype :: $[i,i]=>i$ **where**
$ordertype(A,r) == ordermap(A,r)``A$

definition

Ord-alt :: $i => o$ **where**
$Ord\text{-}alt(X) == well\text{-}ord(X,\ Memrel(X))\ \&\ (ALL\ u{:}X.\ u{=}pred(X,\ u,\ Memrel(X)))$

definition

ordify :: $i=>i$ **where**
$ordify(x) == if\ Ord(x)\ then\ x\ else\ 0$

definition

omult :: [*i*,*i*]=>*i* (**infixl** ** 70*) **where**
i ** *j* == *ordertype*(*j***i*, *rmult*(*j*,*Memrel*(*j*),*i*,*Memrel*(*i*)))

definition

raw-oadd :: [*i*,*i*]=>*i* **where**
raw-oadd(*i*,*j*) == *ordertype*(*i*+*j*, *radd*(*i*,*Memrel*(*i*),*j*,*Memrel*(*j*)))

definition
oadd :: [*i*,*i*]=>*i* (**infixl** ++ *65*) **where**
i ++ *j* == *raw-oadd*(*ordify*(*i*),*ordify*(*j*))

definition

odiff :: [*i*,*i*]=>*i* (**infixl** -- *65*) **where**
i -- *j* == *ordertype*(*i*-*j*, *Memrel*(*i*))

notation (*xsymbols*)
omult (**infixl** ×× *70*)

notation (*HTML* **output**)
omult (**infixl** ×× *70*)

20.1 Proofs needing the combination of Ordinal.thy and Order.thy

lemma *le-well-ord-Memrel*: *j le i* ==> *well-ord*(*j*, *Memrel*(*i*))
⟨*proof*⟩

lemmas *well-ord-Memrel* = *le-refl* [*THEN le-well-ord-Memrel*]

lemma *lt-pred-Memrel*:
j<*i* ==> *pred*(*i*, *j*, *Memrel*(*i*)) = *j*
⟨*proof*⟩

lemma *pred-Memrel*:
x:*A* ==> *pred*(*A*, *x*, *Memrel*(*A*)) = *A Int x*
⟨*proof*⟩

lemma *Ord-iso-implies-eq-lemma*:
[| *j*<*i*; *f*: *ord-iso*(*i*,*Memrel*(*i*),*j*,*Memrel*(*j*)) |] ==> *R*
⟨*proof*⟩

lemma *Ord-iso-implies-eq*:
[| *Ord*(*i*); *Ord*(*j*); *f*: *ord-iso*(*i*,*Memrel*(*i*),*j*,*Memrel*(*j*)) |]

$\Longrightarrow i{=}j$
⟨*proof*⟩

20.2 Ordermap and ordertype

lemma *ordermap-type*:
$ordermap(A,r) : A \to ordertype(A,r)$
⟨*proof*⟩

20.2.1 Unfolding of ordermap

lemma *ordermap-eq-image*:
$[\!| \ wf[A](r); \ \ x{:}A \ |\!]$
$\Longrightarrow ordermap(A,r)\ `x = ordermap(A,r)\ ``pred(A,x,r)$
⟨*proof*⟩

lemma *ordermap-pred-unfold*:
$[\!| \ wf[A](r); \ \ x{:}A \ |\!]$
$\Longrightarrow ordermap(A,r)\ `x = \{ordermap(A,r)`y \ .\ y : pred(A,x,r)\}$
⟨*proof*⟩

lemmas *ordermap-unfold* = *ordermap-pred-unfold* [*simplified pred-def*]

20.2.2 Showing that ordermap, ordertype yield ordinals

lemma *Ord-ordermap*:
$[\!| \ well\text{-}ord(A,r); \ \ x{:}A \ |\!] \Longrightarrow Ord(ordermap(A,r)\ `x)$
⟨*proof*⟩

lemma *Ord-ordertype*:
$well\text{-}ord(A,r) \Longrightarrow Ord(ordertype(A,r))$
⟨*proof*⟩

20.2.3 ordermap preserves the orderings in both directions

lemma *ordermap-mono*:
$[\!| \ <w,x>{:}\ r; \ \ wf[A](r); \ \ w{:}\ A; \ x{:}\ A \ |\!]$
$\Longrightarrow ordermap(A,r)`w : ordermap(A,r)`x$
⟨*proof*⟩

lemma *converse-ordermap-mono*:
$[\!| \ ordermap(A,r)`w : ordermap(A,r)`x; \ \ well\text{-}ord(A,r); \ w{:}\ A; \ x{:}\ A \ |\!]$
$\Longrightarrow <w,x>{:}\ r$
⟨*proof*⟩

lemmas *ordermap-surj* =
ordermap-type [*THEN surj-image*, *unfolded ordertype-def* [*symmetric*]]

lemma *ordermap-bij*:
well-ord(*A*,*r*) ==> *ordermap*(*A*,*r*) : *bij*(*A*, *ordertype*(*A*,*r*))
⟨*proof*⟩

20.2.4 Isomorphisms involving ordertype

lemma *ordertype-ord-iso*:
well-ord(*A*,*r*)
==> *ordermap*(*A*,*r*) : *ord-iso*(*A*,*r*, *ordertype*(*A*,*r*), *Memrel*(*ordertype*(*A*,*r*)))
⟨*proof*⟩

lemma *ordertype-eq*:
[| *f*: *ord-iso*(*A*,*r*,*B*,*s*); *well-ord*(*B*,*s*) |]
==> *ordertype*(*A*,*r*) = *ordertype*(*B*,*s*)
⟨*proof*⟩

lemma *ordertype-eq-imp-ord-iso*:
[| *ordertype*(*A*,*r*) = *ordertype*(*B*,*s*); *well-ord*(*A*,*r*); *well-ord*(*B*,*s*) |]
==> *EX f*. *f*: *ord-iso*(*A*,*r*,*B*,*s*)
⟨*proof*⟩

20.2.5 Basic equalities for ordertype

lemma *le-ordertype-Memrel*: *j le i* ==> *ordertype*(*j*,*Memrel*(*i*)) = *j*
⟨*proof*⟩

lemmas *ordertype-Memrel* = *le-refl* [*THEN le-ordertype-Memrel*]

lemma *ordertype-0* [*simp*]: *ordertype*(*0*,*r*) = *0*
⟨*proof*⟩

lemmas *bij-ordertype-vimage* = *ord-iso-rvimage* [*THEN ordertype-eq*]

20.2.6 A fundamental unfolding law for ordertype.

lemma *ordermap-pred-eq-ordermap*:
[| *well-ord*(*A*,*r*); *y*:*A*; *z*: *pred*(*A*,*y*,*r*) |]
==> *ordermap*(*pred*(*A*,*y*,*r*), *r*) ‘ *z* = *ordermap*(*A*, *r*) ‘ *z*
⟨*proof*⟩

lemma *ordertype-unfold*:
ordertype(*A*,*r*) = {*ordermap*(*A*,*r*)‘*y* . *y* : *A*}
⟨*proof*⟩

Theorems by Krzysztof Grabczewski; proofs simplified by lcp

lemma *ordertype-pred-subset*: [| *well-ord*(*A*,*r*); *x*:*A* |] ==>
ordertype(*pred*(*A*,*x*,*r*),*r*) <= *ordertype*(*A*,*r*)

⟨*proof*⟩

lemma *ordertype-pred-lt*:
[| *well-ord*(*A*,*r*); *x*:*A* |]
==> *ordertype*(*pred*(*A*,*x*,*r*),*r*) < *ordertype*(*A*,*r*)
⟨*proof*⟩

lemma *ordertype-pred-unfold*:
well-ord(*A*,*r*)
==> *ordertype*(*A*,*r*) = {*ordertype*(*pred*(*A*,*x*,*r*),*r*). *x*:*A*}
⟨*proof*⟩

20.3 Alternative definition of ordinal

lemma *Ord-is-Ord-alt*: *Ord*(*i*) ==> *Ord-alt*(*i*)
⟨*proof*⟩

lemma *Ord-alt-is-Ord*:
Ord-alt(*i*) ==> *Ord*(*i*)
⟨*proof*⟩

20.4 Ordinal Addition

20.4.1 Order Type calculations for radd

Addition with 0

lemma *bij-sum-0*: (*lam z*:*A*+*0*. *case*(%*x*. *x*, %*y*. *y*, *z*)) : *bij*(*A*+*0*, *A*)
⟨*proof*⟩

lemma *ordertype-sum-0-eq*:
well-ord(*A*,*r*) ==> *ordertype*(*A*+*0*, *radd*(*A*,*r*,*0*,*s*)) = *ordertype*(*A*,*r*)
⟨*proof*⟩

lemma *bij-0-sum*: (*lam z*:*0*+*A*. *case*(%*x*. *x*, %*y*. *y*, *z*)) : *bij*(*0*+*A*, *A*)
⟨*proof*⟩

lemma *ordertype-0-sum-eq*:
well-ord(*A*,*r*) ==> *ordertype*(*0*+*A*, *radd*(*0*,*s*,*A*,*r*)) = *ordertype*(*A*,*r*)
⟨*proof*⟩

Initial segments of radd. Statements by Grabczewski

lemma *pred-Inl-bij*:
a:*A* ==> (*lam x*:*pred*(*A*,*a*,*r*). *Inl*(*x*))
: *bij*(*pred*(*A*,*a*,*r*), *pred*(*A*+*B*, *Inl*(*a*), *radd*(*A*,*r*,*B*,*s*)))
⟨*proof*⟩

lemma *ordertype-pred-Inl-eq*:

[| a:A; $well\text{-}ord(A,r)$ |]
==> $ordertype(pred(A+B,\ Inl(a),\ radd(A,r,B,s)),\ radd(A,r,B,s)) =$
$ordertype(pred(A,a,r),\ r)$
⟨*proof*⟩

lemma *pred-Inr-bij*:
b:B ==>
$id(A+pred(B,b,s))$
: $bij(A+pred(B,b,s),\ pred(A+B,\ Inr(b),\ radd(A,r,B,s)))$
⟨*proof*⟩

lemma *ordertype-pred-Inr-eq*:
[| b:B; $well\text{-}ord(A,r)$; $well\text{-}ord(B,s)$ |]
==> $ordertype(pred(A+B,\ Inr(b),\ radd(A,r,B,s)),\ radd(A,r,B,s)) =$
$ordertype(A+pred(B,b,s),\ radd(A,r,pred(B,b,s),s))$
⟨*proof*⟩

20.4.2 ordify: trivial coercion to an ordinal

lemma *Ord-ordify* [*iff*, *TC*]: $Ord(ordify(x))$
⟨*proof*⟩

lemma *ordify-idem* [*simp*]: $ordify(ordify(x)) = ordify(x)$
⟨*proof*⟩

20.4.3 Basic laws for ordinal addition

lemma *Ord-raw-oadd*: [|$Ord(i)$; $Ord(j)$|] ==> $Ord(raw\text{-}oadd(i,j))$
⟨*proof*⟩

lemma *Ord-oadd* [*iff*,*TC*]: $Ord(i{+}{+}j)$
⟨*proof*⟩

Ordinal addition with zero

lemma *raw-oadd-0*: $Ord(i)$ ==> $raw\text{-}oadd(i,0) = i$
⟨*proof*⟩

lemma *oadd-0* [*simp*]: $Ord(i)$ ==> $i{+}{+}0 = i$
⟨*proof*⟩

lemma *raw-oadd-0-left*: $Ord(i)$ ==> $raw\text{-}oadd(0,i) = i$
⟨*proof*⟩

lemma *oadd-0-left* [*simp*]: $Ord(i)$ ==> $0{+}{+}i = i$
⟨*proof*⟩

lemma *oadd-eq-if-raw-oadd*:
$i{+}{+}j = (if\ Ord(i)\ then\ (if\ Ord(j)\ then\ raw\text{-}oadd(i,j)\ else\ i)$

else (if Ord(j) then j else 0))
⟨*proof*⟩

lemma *raw-oadd-eq-oadd*: [|*Ord(i)*; *Ord(j)*|] ==> *raw-oadd(i,j)* = *i++j*
⟨*proof*⟩

lemma *lt-oadd1*: *k<i* ==> *k* < *i++j*
⟨*proof*⟩

lemma *oadd-le-self*: *Ord(i)* ==> *i le i++j*
⟨*proof*⟩

Various other results

lemma *id-ord-iso-Memrel*: *A<=B* ==> *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<=B* |] ==> *f*: *ord-iso*(*A*,*Memrel*(*A*),*C*,*r*)
⟨*proof*⟩

lemma *restrict-ord-iso*:
[| *f* ∈ *ord-iso*(*i*, *Memrel*(*i*), *Order.pred*(*A*,*a*,*r*), *r*); *a* ∈ *A*; *j* < *i*;
trans[*A*](*r*) |]
==> *restrict*(*f*,*j*) ∈ *ord-iso*(*j*, *Memrel*(*j*), *Order.pred*(*A*,*f*‘*j*,*r*), *r*)
⟨*proof*⟩

lemma *restrict-ord-iso2*:
[| *f* ∈ *ord-iso*(*Order.pred*(*A*,*a*,*r*), *r*, *i*, *Memrel*(*i*)); *a* ∈ *A*;
j < *i*; *trans*[*A*](*r*) |]
==> *converse*(*restrict*(*converse*(*f*), *j*))
∈ *ord-iso*(*Order.pred*(*A*, *converse*(*f*)‘*j*, *r*), *r*, *j*, *Memrel*(*j*))
⟨*proof*⟩

lemma *ordertype-sum-Memrel*:
[| *well-ord*(*A*,*r*); *k<j* |]
==> *ordertype*(*A*+*k*, *radd*(*A*, *r*, *k*, *Memrel*(*j*))) =
ordertype(*A*+*k*, *radd*(*A*, *r*, *k*, *Memrel*(*k*)))
⟨*proof*⟩

lemma *oadd-lt-mono2*: *k<j* ==> *i++k* < *i++j*
⟨*proof*⟩

lemma *oadd-lt-cancel2*: [| *i++j* < *i++k*; *Ord(j)* |] ==> *j<k*
⟨*proof*⟩

lemma *oadd-lt-iff2*: $Ord(j) \Longrightarrow i{+}{+}j < i{+}{+}k \longleftrightarrow j<k$
⟨*proof*⟩

lemma *oadd-inject*: $[\!|\ i{+}{+}j = i{+}{+}k;\ \ Ord(j);\ Ord(k)\ |\!] \Longrightarrow j=k$
⟨*proof*⟩

lemma *lt-oadd-disj*: $k < i{+}{+}j \Longrightarrow k<i \mid (EX\ l{:}j.\ k = i{+}{+}l\ )$
⟨*proof*⟩

20.4.4 Ordinal addition with successor – via associativity!

lemma *oadd-assoc*: $(i{+}{+}j){+}{+}k = i{+}{+}(j{+}{+}k)$
⟨*proof*⟩

lemma *oadd-unfold*: $[\!|\ Ord(i);\ \ Ord(j)\ |\!] \Longrightarrow i{+}{+}j = i\ Un\ (\bigcup k{\in}j.\ \{i{+}{+}k\})$
⟨*proof*⟩

lemma *oadd-1*: $Ord(i) \Longrightarrow i{+}{+}1 = succ(i)$
⟨*proof*⟩

lemma *oadd-succ* [*simp*]: $Ord(j) \Longrightarrow i{+}{+}succ(j) = succ(i{+}{+}j)$
⟨*proof*⟩

Ordinal addition with limit ordinals

lemma *oadd-UN*:
$[\!|\ !!x.\ x{:}A \Longrightarrow Ord(j(x));\ \ a{:}A\ |\!]$
$\Longrightarrow i\ {+}{+}\ (\bigcup x{\in}A.\ j(x)) = (\bigcup x{\in}A.\ i{+}{+}j(x))$
⟨*proof*⟩

lemma *oadd-Limit*: $Limit(j) \Longrightarrow i{+}{+}j = (\bigcup k{\in}j.\ i{+}{+}k)$
⟨*proof*⟩

lemma *oadd-eq-0-iff*: $[\!|\ Ord(i);\ Ord(j)\ |\!] \Longrightarrow (i\ {+}{+}\ j) = 0 \longleftrightarrow i=0\ \&\ j=0$
⟨*proof*⟩

lemma *oadd-eq-lt-iff*: $[\!|\ Ord(i);\ Ord(j)\ |\!] \Longrightarrow 0 < (i\ {+}{+}\ j) \longleftrightarrow 0<i \mid 0<j$
⟨*proof*⟩

lemma *oadd-LimitI*: $[\!|\ Ord(i);\ Limit(j)\ |\!] \Longrightarrow Limit(i\ {+}{+}\ j)$
⟨*proof*⟩

Order/monotonicity properties of ordinal addition

lemma *oadd-le-self2*: $Ord(i) \Longrightarrow i\ le\ j{+}{+}i$
⟨*proof*⟩

lemma *oadd-le-mono1*: $k\ le\ j \Longrightarrow k{+}{+}i\ le\ j{+}{+}i$
⟨*proof*⟩

lemma *oadd-lt-mono*: [| i' *le* i; $j'<j$ |] ==> $i'++j' < i++j$
⟨*proof*⟩

lemma *oadd-le-mono*: [| i' *le* i; j' *le* j |] ==> $i'++j'$ *le* $i++j$
⟨*proof*⟩

lemma *oadd-le-iff2*: [| *Ord*(j); *Ord*(k) |] ==> $i++j$ *le* $i++k$ <-> j *le* k
⟨*proof*⟩

lemma *oadd-lt-self*: [| *Ord*(i); $0<j$ |] ==> $i < i++j$
⟨*proof*⟩

Every ordinal is exceeded by some limit ordinal.

lemma *Ord-imp-greater-Limit*: *Ord*(i) ==> ∃k. $i<k$ & *Limit*(k)
⟨*proof*⟩

lemma *Ord2-imp-greater-Limit*: [|*Ord*(i); *Ord*(j)|] ==> ∃k. $i<k$ & $j<k$ & *Limit*(k)
⟨*proof*⟩

20.5 Ordinal Subtraction

The difference is *ordertype*(j − i, *Memrel*(j)). It's probably simpler to define the difference recursively!

lemma *bij-sum-Diff*:
$A<=B$ ==> (*lam* y:B. *if*(y:A, *Inl*(y), *Inr*(y))) : *bij*(B, A+(B−A))
⟨*proof*⟩

lemma *ordertype-sum-Diff*:
i le j ==>
ordertype(i+(j−i), *radd*(i,*Memrel*(j),j−i,*Memrel*(j))) =
ordertype(j, *Memrel*(j))
⟨*proof*⟩

lemma *Ord-odiff* [*simp*,*TC*]:
[| *Ord*(i); *Ord*(j) |] ==> *Ord*(i−−j)
⟨*proof*⟩

lemma *raw-oadd-ordertype-Diff*:
i le j
==> *raw-oadd*(i,j−−i) = *ordertype*(i+(j−i), *radd*(i,*Memrel*(j),j−i,*Memrel*(j)))
⟨*proof*⟩

lemma *oadd-odiff-inverse*: *i le j* ==> i ++ (j−−i) = j
⟨*proof*⟩

lemma *odiff-oadd-inverse*: [| *Ord*(i); *Ord*(j) |] ==> (i++j) −− i = j

⟨*proof*⟩

lemma *odiff-lt-mono2*: [| $i<j$; k *le* i |] ==> $i--k < j--k$
⟨*proof*⟩

20.6 Ordinal Multiplication

lemma *Ord-omult* [*simp*,*TC*]:
[| *Ord*(i); *Ord*(j) |] ==> *Ord*(i**j)
⟨*proof*⟩

20.6.1 A useful unfolding law

lemma *pred-Pair-eq*:
[| a:A; b:B |] ==> *pred*(A*B, <a,b>, *rmult*(A,r,B,s)) =
pred(A,a,r)*B *Un* ({a} * *pred*(B,b,s))
⟨*proof*⟩

lemma *ordertype-pred-Pair-eq*:
[| a:A; b:B; *well-ord*(A,r); *well-ord*(B,s) |] ==>
ordertype(*pred*(A*B, <a,b>, *rmult*(A,r,B,s)), *rmult*(A,r,B,s)) =
ordertype(*pred*(A,a,r)*B + *pred*(B,b,s),
radd(A*B, *rmult*(A,r,B,s), B, s))
⟨*proof*⟩

lemma *ordertype-pred-Pair-lemma*:
[| $i'<i$; $j'<j$ |]
==> *ordertype*(*pred*(i*j, <i',j'>, *rmult*(i,*Memrel*(i),j,*Memrel*(j))),
rmult(i,*Memrel*(i),j,*Memrel*(j))) =
raw-oadd (j**i', j')
⟨*proof*⟩

lemma *lt-omult*:
[| *Ord*(i); *Ord*(j); $k<j$**i |]
==> *EX* j' i'. k = j**i' ++ j' & $j'<j$ & $i'<i$
⟨*proof*⟩

lemma *omult-oadd-lt*:
[| $j'<j$; $i'<i$ |] ==> j**i' ++ j' < j**i
⟨*proof*⟩

lemma *omult-unfold*:
[| *Ord*(i); *Ord*(j) |] ==> j**i = ($\bigcup j' \in j.\ \bigcup i' \in i.\ \{j**i' ++ j'\}$)
⟨*proof*⟩

20.6.2 Basic laws for ordinal multiplication

Ordinal multiplication by zero

lemma *omult-0* [*simp*]: i**0 = 0*

⟨*proof*⟩

lemma *omult-0-left* [*simp*]: $0{**}i = 0$
⟨*proof*⟩

Ordinal multiplication by 1

lemma *omult-1* [*simp*]: $Ord(i) \Longrightarrow i{**}1 = i$
⟨*proof*⟩

lemma *omult-1-left* [*simp*]: $Ord(i) \Longrightarrow 1{**}i = i$
⟨*proof*⟩

Distributive law for ordinal multiplication and addition

lemma *oadd-omult-distrib*:
$[\!| \ Ord(i); \ \ Ord(j); \ \ Ord(k) \ |\!] \Longrightarrow i{**}(j{+\!+}k) = (i{**}j){+\!+}(i{**}k)$
⟨*proof*⟩

lemma *omult-succ*: $[\!| \ Ord(i); \ \ Ord(j) \ |\!] \Longrightarrow i{**}succ(j) = (i{**}j){+\!+}i$
⟨*proof*⟩

Associative law

lemma *omult-assoc*:
$[\!| \ Ord(i); \ \ Ord(j); \ \ Ord(k) \ |\!] \Longrightarrow (i{**}j){**}k = i{**}(j{**}k)$
⟨*proof*⟩

Ordinal multiplication with limit ordinals

lemma *omult-UN*:
$[\!| \ Ord(i); \ \ !!x.\ x{:}A \Longrightarrow Ord(j(x)) \ |\!]$
$\Longrightarrow i \mathbin{**} (\bigcup x{\in}A.\ j(x)) = (\bigcup x{\in}A.\ i{**}j(x))$
⟨*proof*⟩

lemma *omult-Limit*: $[\!| \ Ord(i); \ \ Limit(j) \ |\!] \Longrightarrow i{**}j = (\bigcup k{\in}j.\ i{**}k)$
⟨*proof*⟩

20.6.3 Ordering/monotonicity properties of ordinal multiplication

lemma *lt-omult1*: $[\!| \ k{<}i; \ \ 0{<}j \ |\!] \Longrightarrow k < i{**}j$
⟨*proof*⟩

lemma *omult-le-self*: $[\!| \ Ord(i); \ \ 0{<}j \ |\!] \Longrightarrow i \ le \ i{**}j$
⟨*proof*⟩

lemma *omult-le-mono1*: $[\!| \ k \ le \ j; \ \ Ord(i) \ |\!] \Longrightarrow k{**}i \ le \ j{**}i$
⟨*proof*⟩

lemma *omult-lt-mono2*: $[\!| \ k{<}j; \ \ 0{<}i \ |\!] \Longrightarrow i{**}k < i{**}j$
⟨*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 :: $i \Rightarrow i$
  FiniteFun :: $[i,i] \Rightarrow i$ ((- −||>/ -) [61, 60] 60)

inductive
  domains $Fin(A) \leq Pow(A)$
  intros
    emptyI: $0 : Fin(A)$
    consI: $[\!|\ a: A;\ b: Fin(A)\ |\!] \Longrightarrow cons(a,b) : Fin(A)$
  type-intros *empty-subsetI cons-subsetI PowI*
  type-elims *PowD* [*THEN revcut-rl*]

inductive
  domains $FiniteFun(A,B) \leq Fin(A*B)$
  intros
    emptyI: $0 : A -||> B$
    consI: $[\!|\ a: A;\ b: B;\ h: A -||> B;\ a \sim: domain(h)\ |\!]$
      $\Longrightarrow cons(\langle a,b\rangle,h) : A -||> B$
  type-intros *Fin.intros*

21.1 Finite Powerset Operator

lemma *Fin-mono*: $A \leq B \Longrightarrow Fin(A) \leq Fin(B)$
⟨*proof*⟩

lemmas *FinD* = *Fin.dom-subset* [*THEN subsetD*, *THEN PowD*, *standard*]

lemma *Fin-induct* [*case-names 0 cons*, *induct set*: *Fin*]:
  $[\!|\ b: Fin(A);$
    $P(0);$
    $!!x\ y.\ [\!|\ x: A;\ y: Fin(A);\ x \sim: y;\ P(y)\ |\!] \Longrightarrow P(cons(x,y))$
  $|\!] \Longrightarrow P(b)$
⟨*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: $Fin(A) ==> \forall z.\ z<=b --> z$: $Fin(A)$
⟨*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; $P(y)$ |] ==> $P(y-\{x\})$
|] ==> $P(0)$
⟨*proof*⟩

lemma *nat-fun-subset-Fin*: n: $nat ==> n->A <= Fin(nat*A)$
⟨*proof*⟩

21.2 Finite Function Space

lemma *FiniteFun-mono*:
[| $A<=C$; $B<=D$ |] ==> $A\ -||>\ B\ <=\ C\ -||>\ D$
⟨*proof*⟩

lemma *FiniteFun-mono1*: $A<=B ==> A -||> A <= B -||> B$
⟨*proof*⟩

lemma *FiniteFun-is-fun*: $h: A -||>B ==> h: domain(h) -> B$
⟨*proof*⟩

lemma *FiniteFun-domain-Fin*: $h: A -||>B ==> domain(h) : Fin(A)$
⟨*proof*⟩

lemmas *FiniteFun-apply-type* = *FiniteFun-is-fun* [*THEN apply-type*, *standard*]

lemma *FiniteFun-subset-lemma* [*rule-format*]:
$b: A-||>B ==> ALL\ z.\ z<=b --> z: A-||>B$
⟨*proof*⟩

lemma *FiniteFun-subset*: $[|\ c<=b;\ b: A-||>B\ |] ==> c: A-||>B$
⟨*proof*⟩

lemma *fun-FiniteFunI* [*rule-format*]: $A:Fin(X) ==> ALL f.\ f:A->B --> f:A-||>B$
⟨*proof*⟩

lemma *lam-FiniteFun*: $A: Fin(X) ==> (lam\ x:A.\ b(x)) : A -||> \{b(x).\ x:A\}$
⟨*proof*⟩

lemma *FiniteFun-Collect-iff*:
$f : FiniteFun(A, \{y:B.\ P(y)\})$
$<-> f : FiniteFun(A,B)\ \&\ (ALL\ x:domain(f).\ P(f`x))$
⟨*proof*⟩

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 :: $(i{=}{>}o) \Rightarrow i$ (**binder** *LEAST* *10*) **where**
Least$(P) == THE\ i.\ Ord(i)\ \&\ P(i)\ \&\ (ALL\ j.\ j{<}i\ {-}{-}{>}\ {\sim}P(j))$

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 :: $i{=}{>}i$ (|-|) **where**
$|A| == LEAST\ i.\ i\ eqpoll\ A$

definition
Finite :: $i{=}{>}o$ **where**
$Finite(A) == EX\ n{:}nat.\ A\ eqpoll\ n$

definition
Card :: $i{=}{>}o$ **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** $\approx$ *50*) **and**
Least (**binder** μ *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*:
$g{:}\ Y{->}X$
$\Longrightarrow g``(Y - f``\ lfp(X, \%W.\ X - g``(Y - f``W))) =$

$X - lfp(X, \%W.\ X - g``(Y - f``W))$
⟨*proof*⟩

lemma *decomposition*:
$[|\ f\colon X->Y;\ \ g\colon Y->X\ |] ==>$
$EX\ XA\ XB\ YA\ YB.\ (XA\ Int\ XB = 0)\ \&\ (XA\ Un\ XB = X)\ \&$
$(YA\ Int\ YB = 0)\ \&\ (YA\ Un\ YB = Y)\ \&$
$f``XA=YA\ \&\ g``YB=XB$
⟨*proof*⟩

lemma *schroeder-bernstein*:
$[|\ f\colon inj(X,Y);\ \ g\colon inj(Y,X)\ |] ==> EX\ h.\ h\colon bij(X,Y)$
⟨*proof*⟩

lemma *bij-imp-eqpoll*: $f\colon bij(A,B) ==> A \approx B$
⟨*proof*⟩

lemmas *eqpoll-refl* = *id-bij* [*THEN bij-imp-eqpoll*, *standard*, *simp*]

lemma *eqpoll-sym*: $X \approx Y ==> Y \approx X$
⟨*proof*⟩

lemma *eqpoll-trans*:
$[|\ X \approx Y;\ \ Y \approx Z\ |] ==> X \approx Z$
⟨*proof*⟩

lemma *subset-imp-lepoll*: $X<=Y ==> X \lesssim Y$
⟨*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*: $X \approx Y ==> X \lesssim Y$
⟨*proof*⟩

lemma *lepoll-trans*: $[|\ X \lesssim Y;\ \ Y \lesssim Z\ |] ==> X \lesssim Z$
⟨*proof*⟩

lemma *eqpollI*: $[|\ X \lesssim Y;\ \ Y \lesssim X\ |] ==> X \approx Y$
⟨*proof*⟩

lemma *eqpollE*:
[| $X \approx Y$; [| $X \lesssim Y$; $Y \lesssim X$ |] ==> P |] ==> P
⟨*proof*⟩

lemma *eqpoll-iff*: $X \approx Y <-> X \lesssim Y \ \& \ Y \lesssim X$
⟨*proof*⟩

lemma *lepoll-0-is-0*: $A \lesssim 0 ==> A = 0$
⟨*proof*⟩

lemmas *empty-lepollI* = *empty-subsetI* [*THEN subset-imp-lepoll*, *standard*]

lemma *lepoll-0-iff*: $A \lesssim 0 <-> A=0$
⟨*proof*⟩

lemma *Un-lepoll-Un*:
[| $A \lesssim B$; $C \lesssim D$; $B \ Int \ D = 0$ |] ==> $A \ Un \ C \lesssim B \ Un \ D$
⟨*proof*⟩

lemmas *eqpoll-0-is-0* = *eqpoll-imp-lepoll* [*THEN lepoll-0-is-0*, *standard*]

lemma *eqpoll-0-iff*: $A \approx 0 <-> A=0$
⟨*proof*⟩

lemma *eqpoll-disjoint-Un*:
[| $A \approx B$; $C \approx D$; $A \ Int \ C = 0$; $B \ Int \ D = 0$ |]
==> $A \ Un \ C \approx B \ Un \ D$
⟨*proof*⟩

22.2 lesspoll: contributions by Krzysztof Grabczewski

lemma *lesspoll-not-refl*: $\sim (i \prec i)$
⟨*proof*⟩

lemma *lesspoll-irrefl* [*elim*!]: $i \prec i ==> P$
⟨*proof*⟩

lemma *lesspoll-imp-lepoll*: $A \prec B ==> A \lesssim B$
⟨*proof*⟩

lemma *lepoll-well-ord*: [| $A \lesssim B$; *well-ord*(B,r) |] ==> *EX s. well-ord*(A,s)
⟨*proof*⟩

lemma *lepoll-iff-leqpoll*: $A \lesssim B <-> A \prec B \mid A \approx B$
⟨*proof*⟩

lemma *inj-not-surj-succ*:

[| f : $inj(A,\ succ(m))$; f ~: $surj(A,\ succ(m))$ |] ==> $EX\ f.\ f$:$inj(A,m)$
⟨*proof*⟩

lemma *lesspoll-trans*:
[| $X \prec Y$; $Y \prec Z$ |] ==> $X \prec Z$
⟨*proof*⟩

lemma *lesspoll-trans1*:
[| $X \lesssim Y$; $Y \prec Z$ |] ==> $X \prec Z$
⟨*proof*⟩

lemma *lesspoll-trans2*:
[| $X \prec Y$; $Y \lesssim Z$ |] ==> $X \prec Z$
⟨*proof*⟩

lemma *Least-equality*:
[| $P(i)$; $Ord(i)$; !!x. x<i ==> ~$P(x)$ |] ==> $(LEAST\ x.\ P(x)) = i$
⟨*proof*⟩

lemma *LeastI*: [| $P(i)$; $Ord(i)$ |] ==> $P(LEAST\ x.\ P(x))$
⟨*proof*⟩

lemma *Least-le*: [| $P(i)$; $Ord(i)$ |] ==> $(LEAST\ x.\ P(x))\ le\ i$
⟨*proof*⟩

lemma *less-LeastE*: [| $P(i)$; $i < (LEAST\ x.\ P(x))$ |] ==> Q
⟨*proof*⟩

lemma *LeastI2*:
[| $P(i)$; $Ord(i)$; !!j. $P(j)$ ==> $Q(j)$ |] ==> $Q(LEAST\ j.\ P(j))$
⟨*proof*⟩

lemma *Least-0*:
[| ~ $(EX\ i.\ Ord(i)\ \&\ P(i))$ |] ==> $(LEAST\ x.\ P(x)) = 0$
⟨*proof*⟩

lemma *Ord-Least* [*intro*,*simp*,*TC*]: $Ord(LEAST\ x.\ P(x))$
⟨*proof*⟩

lemma *Least-cong*:
(!!y. $P(y)$ <-> $Q(y)$) ==> $(LEAST\ x.\ P(x)) = (LEAST\ x.\ Q(x))$
⟨*proof*⟩

lemma *cardinal-cong*: $X \approx Y$ ==> $|X| = |Y|$
⟨*proof*⟩

lemma *well-ord-cardinal-eqpoll*:
well-ord(A,r) ==> $|A| \approx A$
⟨*proof*⟩

lemmas *Ord-cardinal-eqpoll* = *well-ord-Memrel* [*THEN well-ord-cardinal-eqpoll*]

lemma *well-ord-cardinal-eqE*:
[| *well-ord*(X,r); *well-ord*(Y,s); $|X| = |Y|$ |] ==> $X \approx Y$
⟨*proof*⟩

lemma *well-ord-cardinal-eqpoll-iff*:
[| *well-ord*(X,r); *well-ord*(Y,s) |] ==> $|X| = |Y|$ <-> $X \approx Y$
⟨*proof*⟩

lemma *Ord-cardinal-le*: *Ord*(i) ==> $|i|$ *le* i
⟨*proof*⟩

lemma *Card-cardinal-eq*: *Card*(K) ==> $|K| = K$
⟨*proof*⟩

lemma *CardI*: [| *Ord*(i); !!j. $j<i$ ==> ~($j \approx i$) |] ==> *Card*(i)
⟨*proof*⟩

lemma *Card-is-Ord*: *Card*(i) ==> *Ord*(i)
⟨*proof*⟩

lemma *Card-cardinal-le*: *Card*(K) ==> K *le* $|K|$
⟨*proof*⟩

lemma *Ord-cardinal* [*simp*,*intro*!]: *Ord*($|A|$)
⟨*proof*⟩

lemma *Card-iff-initial*: $Card(K) <-> Ord(K)$ & $(ALL\ j.\ j<K\ -->\ \sim\ j \approx K)$
⟨*proof*⟩

lemma *lt-Card-imp-lesspoll*: [| $Card(a)$; $i<a$ |] ==> $i \prec a$
⟨*proof*⟩

lemma *Card-0*: $Card(0)$
⟨*proof*⟩

lemma *Card-Un*: [| $Card(K)$; $Card(L)$ |] ==> $Card(K\ Un\ L)$
⟨*proof*⟩

lemma *Card-cardinal*: $Card(|A|)$
⟨*proof*⟩

lemma *cardinal-eq-lemma*: [| $|i|\ le\ j$; $j\ le\ i$ |] ==> $|j| = |i|$
⟨*proof*⟩

lemma *cardinal-mono*: $i\ le\ j$ ==> $|i|\ le\ |j|$
⟨*proof*⟩

lemma *cardinal-lt-imp-lt*: [| $|i| < |j|$; $Ord(i)$; $Ord(j)$ |] ==> $i < j$
⟨*proof*⟩

lemma *Card-lt-imp-lt*: [| $|i| < K$; $Ord(i)$; $Card(K)$ |] ==> $i < K$
⟨*proof*⟩

lemma *Card-lt-iff*: [| $Ord(i)$; $Card(K)$ |] ==> $(|i| < K) <-> (i < K)$
⟨*proof*⟩

lemma *Card-le-iff*: [| $Ord(i)$; $Card(K)$ |] ==> $(K\ le\ |i|) <-> (K\ le\ i)$
⟨*proof*⟩

lemma *well-ord-lepoll-imp-Card-le*:
[| *well-ord*(B,r); $A \lesssim B$ |] ==> $|A|\ le\ |B|$
⟨*proof*⟩

lemma *lepoll-cardinal-le*: [| $A \lesssim i$; $Ord(i)$ |] ==> $|A|\ le\ i$
⟨*proof*⟩

lemma *lepoll-Ord-imp-eqpoll*: [| $A \lesssim i$; $Ord(i)$ |] ==> $|A| \approx A$
⟨*proof*⟩

lemma *lesspoll-imp-eqpoll*: [| $A \prec i$; $Ord(i)$ |] ==> $|A| \approx A$
⟨*proof*⟩

lemma *cardinal-subset-Ord*: [|$A<=i$; $Ord(i)$|] ==> $|A| <= i$
⟨*proof*⟩

22.3 The finite cardinals

lemma *cons-lepoll-consD*:
[| $cons(u,A) \lesssim cons(v,B)$; $u^{\sim}:A$; $v^{\sim}:B$ |] ==> $A \lesssim B$
⟨*proof*⟩

lemma *cons-eqpoll-consD*: [| $cons(u,A) \approx cons(v,B)$; $u^{\sim}:A$; $v^{\sim}:B$ |] ==> $A \approx B$
⟨*proof*⟩

lemma *succ-lepoll-succD*: $succ(m) \lesssim succ(n)$ ==> $m \lesssim n$
⟨*proof*⟩

lemma *nat-lepoll-imp-le* [*rule-format*]:
m:*nat* ==> *ALL* n: *nat*. $m \lesssim n$ --> m *le* n
⟨*proof*⟩

lemma *nat-eqpoll-iff*: [| m:*nat*; n: *nat* |] ==> $m \approx n$ <-> $m = n$
⟨*proof*⟩

lemma *nat-into-Card*:
n: *nat* ==> $Card(n)$
⟨*proof*⟩

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*: [| $succ(n) \lesssim n$; n:*nat* |] ==> P
⟨*proof*⟩

lemma *n-lesspoll-nat*: $n \in nat$ ==> $n \prec nat$
⟨*proof*⟩

lemma *nat-lepoll-imp-ex-eqpoll-n*:
[| $n \in nat$; $nat \lesssim X$ |] ==> $\exists Y.\ Y \subseteq X\ \&\ n \approx Y$
⟨*proof*⟩

lemma *lepoll-imp-lesspoll-succ*:
[| $A \lesssim m$; m:*nat* |] ==> $A \prec succ(m)$
⟨*proof*⟩

lemma *lesspoll-succ-imp-lepoll*:
[| $A \prec succ(m)$; m:*nat* |] ==> $A \lesssim m$
⟨*proof*⟩

lemma *lesspoll-succ-iff*: m:*nat* ==> $A \prec succ(m) <-> A \lesssim m$
⟨*proof*⟩

lemma *lepoll-succ-disj*: [| $A \lesssim succ(m)$; m:*nat* |] ==> $A \lesssim m \mid A \approx succ(m)$
⟨*proof*⟩

lemma *lesspoll-cardinal-lt*: [| $A \prec i$; *Ord*(i) |] ==> $|A| < i$
⟨*proof*⟩

22.4 The first infinite cardinal: Omega, or nat

lemma *lt-not-lepoll*: [| $n<i$; n:*nat* |] ==> $\sim i \lesssim n$
⟨*proof*⟩

lemma *Ord-nat-eqpoll-iff*: [| *Ord*(i); n:*nat* |] ==> $i \approx n <-> i=n$
⟨*proof*⟩

lemma *Card-nat*: *Card*(*nat*)
⟨*proof*⟩

lemma *nat-le-cardinal*: *nat le* i ==> *nat le* $|i|$
⟨*proof*⟩

22.5 Towards Cardinal Arithmetic

lemma *cons-lepoll-cong*:
[| $A \lesssim B$; $b \sim: B$ |] ==> *cons*(a,A) $\lesssim$ *cons*(b,B)
⟨*proof*⟩

lemma *cons-eqpoll-cong*:
[| $A \approx B$; $a \sim: A$; $b \sim: B$ |] ==> *cons*(a,A) $\approx$ *cons*(b,B)
⟨*proof*⟩

lemma *cons-lepoll-cons-iff*:
[| $a \sim: A$; $b \sim: B$ |] ==> *cons*(a,A) $\lesssim$ *cons*(b,B) <-> $A \lesssim B$
⟨*proof*⟩

lemma *cons-eqpoll-cons-iff*:
[| $a \sim: A$; $b \sim: B$ |] ==> *cons*(a,A) $\approx$ *cons*(b,B) <-> $A \approx B$
⟨*proof*⟩

lemma *singleton-eqpoll-1*: $\{a\} \approx 1$
⟨*proof*⟩

lemma *cardinal-singleton*: $|\{a\}| = 1$
⟨*proof*⟩

lemma *not-0-is-lepoll-1*: $A \sim= 0 \Longrightarrow 1 \lesssim A$
⟨*proof*⟩

lemma *succ-eqpoll-cong*: $A \approx B \Longrightarrow succ(A) \approx succ(B)$
⟨*proof*⟩

lemma *sum-eqpoll-cong*: $[\!|\ A \approx C;\ \ B \approx D\ |\!] \Longrightarrow A+B \approx C+D$
⟨*proof*⟩

lemma *prod-eqpoll-cong*:
$[\!|\ A \approx C;\ \ B \approx D\ |\!] \Longrightarrow A*B \approx C*D$
⟨*proof*⟩

lemma *inj-disjoint-eqpoll*:
$[\!|\ f: inj(A,B);\ \ A\ Int\ B = 0\ |\!] \Longrightarrow A\ Un\ (B - range(f)) \approx B$
⟨*proof*⟩

22.6 Lemmas by Krzysztof Grabczewski

lemma *Diff-sing-lepoll*:
$[\!|\ a{:}A;\ \ A \lesssim succ(n)\ |\!] \Longrightarrow A - \{a\} \lesssim n$
⟨*proof*⟩

lemma *lepoll-Diff-sing*:
$[\!|\ succ(n) \lesssim A\ |\!] \Longrightarrow n \lesssim A - \{a\}$
⟨*proof*⟩

lemma *Diff-sing-eqpoll*: $[\!|\ a{:}A;\ A \approx succ(n)\ |\!] \Longrightarrow A - \{a\} \approx n$
⟨*proof*⟩

lemma *lepoll-1-is-sing*: $[\!|\ A \lesssim 1;\ a{:}A\ |\!] \Longrightarrow A = \{a\}$
⟨*proof*⟩

lemma *Un-lepoll-sum*: $A\ Un\ B \lesssim A+B$
⟨*proof*⟩

lemma *well-ord-Un*:
$[\!|\ well\text{-}ord(X,R);\ well\text{-}ord(Y,S)\ |\!] \Longrightarrow EX\ T.\ well\text{-}ord(X\ Un\ Y,\ T)$

⟨*proof*⟩

lemma *disj-Un-eqpoll-sum*: $A\ Int\ B = 0 \Longrightarrow A\ Un\ B \approx A + B$
⟨*proof*⟩

22.7 Finite and infinite sets

lemma *Finite-0* [*simp*]: $Finite(0)$
⟨*proof*⟩

lemma *lepoll-nat-imp-Finite*: $[|\ A \lesssim n;\ \ n{:}nat\ |] \Longrightarrow Finite(A)$
⟨*proof*⟩

lemma *lesspoll-nat-is-Finite*:
$A \prec nat \Longrightarrow Finite(A)$
⟨*proof*⟩

lemma *lepoll-Finite*:
$[|\ Y \lesssim X;\ \ Finite(X)\ |] \Longrightarrow Finite(Y)$
⟨*proof*⟩

lemmas *subset-Finite* = *subset-imp-lepoll* [*THEN lepoll-Finite*, *standard*]

lemma *Finite-Int*: $Finite(A) \mid Finite(B) \Longrightarrow Finite(A\ Int\ B)$
⟨*proof*⟩

lemmas *Finite-Diff* = *Diff-subset* [*THEN subset-Finite*, *standard*]

lemma *Finite-cons*: $Finite(x) \Longrightarrow Finite(cons(y,x))$
⟨*proof*⟩

lemma *Finite-succ*: $Finite(x) \Longrightarrow Finite(succ(x))$
⟨*proof*⟩

lemma *Finite-cons-iff* [*iff*]: $Finite(cons(y,x)) <-> Finite(x)$
⟨*proof*⟩

lemma *Finite-succ-iff* [*iff*]: $Finite(succ(x)) <-> Finite(x)$
⟨*proof*⟩

lemma *nat-le-infinite-Ord*:
$[|\ Ord(i);\ \ ^\sim\ Finite(i)\ |] \Longrightarrow nat\ le\ i$
⟨*proof*⟩

lemma *Finite-imp-well-ord*:
$Finite(A) \Longrightarrow EX\ r.\ well\text{-}ord(A,r)$
⟨*proof*⟩

lemma *succ-lepoll-imp-not-empty*: $succ(x) \lesssim y \Longrightarrow y \neq 0$
⟨*proof*⟩

lemma *eqpoll-succ-imp-not-empty*: $x \approx succ(n) \Longrightarrow x \neq 0$
⟨*proof*⟩

lemma *Finite-Fin-lemma* [*rule-format*]:
$n \in nat \Longrightarrow \forall A.\ (A \approx n\ \&\ A \subseteq X) \dashrightarrow A \in Fin(X)$
⟨*proof*⟩

lemma *Finite-Fin*: $[|\ Finite(A);\ A \subseteq X\ |] \Longrightarrow A \in Fin(X)$
⟨*proof*⟩

lemma *eqpoll-imp-Finite-iff*: $A \approx B \Longrightarrow Finite(A) \longleftrightarrow Finite(B)$
⟨*proof*⟩

lemma *Fin-lemma* [*rule-format*]: n: $nat \Longrightarrow ALL\ A.\ A \approx n \dashrightarrow A : Fin(A)$
⟨*proof*⟩

lemma *Finite-into-Fin*: $Finite(A) \Longrightarrow A : Fin(A)$
⟨*proof*⟩

lemma *Fin-into-Finite*: $A : Fin(U) \Longrightarrow Finite(A)$
⟨*proof*⟩

lemma *Finite-Fin-iff*: $Finite(A) \longleftrightarrow A : Fin(A)$
⟨*proof*⟩

lemma *Finite-Un*: $[|\ Finite(A);\ Finite(B)\ |] \Longrightarrow Finite(A\ Un\ B)$
⟨*proof*⟩

lemma *Finite-Un-iff* [*simp*]: $Finite(A\ Un\ B) \longleftrightarrow (Finite(A)\ \&\ Finite(B))$
⟨*proof*⟩

The converse must hold too.

lemma *Finite-Union*: $[|\ ALL\ y{:}X.\ Finite(y);\ Finite(X)\ |] \Longrightarrow Finite(Union(X))$
⟨*proof*⟩

lemma *Finite-induct* [*case-names 0 cons*, *induct set*: *Finite*]:
$[|\ Finite(A);\ P(0);$
$\quad !!\ x\ B.\quad [|\ Finite(B);\ x \sim: B;\ P(B)\ |] \Longrightarrow P(cons(x,\ B))\ |]$
$\Longrightarrow P(A)$
⟨*proof*⟩

lemma *Diff-sing-Finite*: $Finite(A - \{a\}) \Longrightarrow Finite(A)$
⟨*proof*⟩

lemma *Diff-Finite* [*rule-format*]: *Finite*(*B*) ==> *Finite*(*A*−*B*) −−> *Finite*(*A*)
⟨*proof*⟩

lemma *Finite-RepFun*: *Finite*(*A*) ==> *Finite*(*RepFun*(*A*,*f*))
⟨*proof*⟩

lemma *Finite-RepFun-iff-lemma* [*rule-format*]:
[|*Finite*(*x*); !!*x y*. *f*(*x*)=*f*(*y*) ==> *x*=*y*|]
==> ∀ *A*. *x* = *RepFun*(*A*,*f*) −−> *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*:
(∀ *x y*. *f*(*x*)=*f*(*y*) −−> *x*=*y*) ==> *Finite*(*RepFun*(*A*,*f*)) <−> *Finite*(*A*)
⟨*proof*⟩

lemma *Finite-Pow*: *Finite*(*A*) ==> *Finite*(*Pow*(*A*))
⟨*proof*⟩

lemma *Finite-Pow-imp-Finite*: *Finite*(*Pow*(*A*)) ==> *Finite*(*A*)
⟨*proof*⟩

lemma *Finite-Pow-iff* [*iff*]: *Finite*(*Pow*(*A*)) <−> *Finite*(*A*)
⟨*proof*⟩

lemma *nat-wf-on-converse-Memrel*: *n*:*nat* ==> *wf*[*n*](*converse*(*Memrel*(*n*)))
⟨*proof*⟩

lemma *nat-well-ord-converse-Memrel*: *n*:*nat* ==> *well-ord*(*n*,*converse*(*Memrel*(*n*)))
⟨*proof*⟩

lemma *well-ord-converse*:
[|*well-ord*(*A*,*r*);
well-ord(*ordertype*(*A*,*r*), *converse*(*Memrel*(*ordertype*(*A*, *r*)))) |]
==> *well-ord*(*A*,*converse*(*r*))
⟨*proof*⟩

lemma *ordertype-eq-n*:
[| *well-ord*(*A*,*r*); *A* ≈ *n*; *n*:*nat* |] ==> *ordertype*(*A*,*r*)=*n*
⟨*proof*⟩

lemma *Finite-well-ord-converse*:
[| *Finite*(*A*); *well-ord*(*A*,*r*) |] ==> *well-ord*(*A*,*converse*(*r*))

⟨*proof*⟩

lemma *nat-into-Finite*: *n*:*nat* ==> *Finite*(*n*)
⟨*proof*⟩

lemma *nat-not-Finite*: ~*Finite*(*nat*)
⟨*proof*⟩

⟨*ML*⟩

end

23 Univ: The Cumulative Hierarchy and a Small Universe for Recursive Types

theory *Univ* **imports** *Epsilon Cardinal* **begin**

definition
Vfrom :: [*i*,*i*]=>*i* **where**
Vfrom(*A*,*i*) == *transrec*(*i*, %*x f*. *A Un* (⋃*y*∈*x*. *Pow*(*f*‘*y*)))

abbreviation
Vset :: *i*=>*i* **where**
Vset(*x*) == *Vfrom*(*0*,*x*)

definition
Vrec :: [*i*, [*i*,*i*]=>*i*] =>*i* **where**
Vrec(*a*,*H*) == *transrec*(*rank*(*a*), %*x g*. *lam z*: *Vset*(*succ*(*x*)).
H(*z*, *lam w*:*Vset*(*x*). *g*‘*rank*(*w*)‘*w*)) ‘ *a*

definition
Vrecursor :: [[*i*,*i*]=>*i*, *i*] =>*i* **where**
Vrecursor(*H*,*a*) == *transrec*(*rank*(*a*), %*x g*. *lam z*: *Vset*(*succ*(*x*)).
H(*lam w*:*Vset*(*x*). *g*‘*rank*(*w*)‘*w*, *z*)) ‘ *a*

definition
univ :: *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* (⋃*j*∈*i*. *Pow*(*Vfrom*(*A*,*j*)))
⟨*proof*⟩

23.1.1 Monotonicity

lemma *Vfrom-mono* [*rule-format*]:
    $A<=B ==> \forall j.\ i<=j --> Vfrom(A,i) <= Vfrom(B,j)$
⟨*proof*⟩

lemma *VfromI*: [| $a \in Vfrom(A,j)$; $j<i$ |] ==> $a \in Vfrom(A,i)$
⟨*proof*⟩

23.1.2 A fundamental equality: Vfrom does not require ordinals!

lemma *Vfrom-rank-subset1*: $Vfrom(A,x) <= Vfrom(A,rank(x))$
⟨*proof*⟩

lemma *Vfrom-rank-subset2*: $Vfrom(A,rank(x)) <= Vfrom(A,x)$
⟨*proof*⟩

lemma *Vfrom-rank-eq*: $Vfrom(A,rank(x)) = Vfrom(A,x)$
⟨*proof*⟩

23.2 Basic Closure Properties

lemma *zero-in-Vfrom*: $y:x ==> 0 \in Vfrom(A,x)$
⟨*proof*⟩

lemma *i-subset-Vfrom*: $i <= Vfrom(A,i)$
⟨*proof*⟩

lemma *A-subset-Vfrom*: $A <= Vfrom(A,i)$
⟨*proof*⟩

lemmas *A-into-Vfrom* = *A-subset-Vfrom* [*THEN subsetD*]

lemma *subset-mem-Vfrom*: $a <= Vfrom(A,i) ==> a \in Vfrom(A,succ(i))$
⟨*proof*⟩

23.2.1 Finite sets and ordered pairs

lemma *singleton-in-Vfrom*: $a \in Vfrom(A,i) ==> \{a\} \in Vfrom(A,succ(i))$
⟨*proof*⟩

lemma *doubleton-in-Vfrom*:
    [| $a \in Vfrom(A,i)$; $b \in Vfrom(A,i)$ |] ==> $\{a,b\} \in Vfrom(A,succ(i))$
⟨*proof*⟩

lemma *Pair-in-Vfrom*:
    [| $a \in Vfrom(A,i)$; $b \in Vfrom(A,i)$ |] ==> $<a,b> \in Vfrom(A,succ(succ(i)))$
⟨*proof*⟩

lemma *succ-in-Vfrom*: $a <= Vfrom(A,i) ==> succ(a) \in Vfrom(A,succ(succ(i)))$

⟨*proof*⟩

23.3 0, Successor and Limit Equations for *Vfrom*

lemma *Vfrom-0*: $Vfrom(A,0) = A$
⟨*proof*⟩

lemma *Vfrom-succ-lemma*: $Ord(i) ==> Vfrom(A,succ(i)) = A\ Un\ Pow(Vfrom(A,i))$
⟨*proof*⟩

lemma *Vfrom-succ*: $Vfrom(A,succ(i)) = A\ Un\ Pow(Vfrom(A,i))$
⟨*proof*⟩

lemma *Vfrom-Union*: $y{:}X ==> Vfrom(A,Union(X)) = (\bigcup y{\in}X.\ Vfrom(A,y))$
⟨*proof*⟩

23.4 *Vfrom* applied to Limit Ordinals

lemma *Limit-Vfrom-eq*:
$Limit(i) ==> Vfrom(A,i) = (\bigcup y{\in}i.\ Vfrom(A,y))$
⟨*proof*⟩

lemma *Limit-VfromE*:
$[\!|\ a \in Vfrom(A,i);\ \sim R ==> Limit(i);$
$\quad !!x.\ [\!|\ x<i;\ a \in Vfrom(A,x)\ |\!] ==> R$
$\ |\!] ==> R$
⟨*proof*⟩

lemma *singleton-in-VLimit*:
$[\!|\ a \in Vfrom(A,i);\ Limit(i)\ |\!] ==> \{a\} \in Vfrom(A,i)$
⟨*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¡=Vfrom(A,j)

lemma *doubleton-in-VLimit*:
$[\!|\ a \in Vfrom(A,i);\ b \in Vfrom(A,i);\ Limit(i)\ |\!] ==> \{a,b\} \in Vfrom(A,i)$
⟨*proof*⟩

lemma *Pair-in-VLimit*:
$[\!|\ a \in Vfrom(A,i);\ b \in Vfrom(A,i);\ Limit(i)\ |\!] ==> <a,b> \in Vfrom(A,i)$⟨*proof*⟩

lemma *product-VLimit*: $Limit(i) ==> Vfrom(A,i) * Vfrom(A,i) <= Vfrom(A,i)$
⟨*proof*⟩

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* ∈ *Vfrom*(*A*,*i*)
⟨*proof*⟩

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* ∈ *Vfrom*(*A*,*i*)
⟨*proof*⟩

lemma *Inl-in-VLimit*:
[| *a* ∈ *Vfrom*(*A*,*i*); *Limit*(*i*) |] ==> *Inl*(*a*) ∈ *Vfrom*(*A*,*i*)
⟨*proof*⟩

lemma *Inr-in-VLimit*:
[| *b* ∈ *Vfrom*(*A*,*i*); *Limit*(*i*) |] ==> *Inr*(*b*) ∈ *Vfrom*(*A*,*i*)
⟨*proof*⟩

lemma *sum-VLimit*: *Limit*(*i*) ==> *Vfrom*(*C*,*i*)+*Vfrom*(*C*,*i*) <= *Vfrom*(*C*,*i*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *Transset-Vfrom-succ*:
Transset(*A*) ==> *Vfrom*(*A*, *succ*(*i*)) = *Pow*(*Vfrom*(*A*,*i*))
⟨*proof*⟩

lemma *Transset-Pair-subset*: [| <*a*,*b*> <= *C*; *Transset*(*C*) |] ==> *a*: *C* & *b*: *C*
⟨*proof*⟩

lemma *Transset-Pair-subset-VLimit*:
[| <*a*,*b*> <= *Vfrom*(*A*,*i*); *Transset*(*A*); *Limit*(*i*) |]
==> <*a*,*b*> ∈ *Vfrom*(*A*,*i*)
⟨*proof*⟩

lemma *Union-in-Vfrom*:
[| *X* ∈ *Vfrom*(*A*,*j*); *Transset*(*A*) |] ==> *Union*(*X*) ∈ *Vfrom*(*A*, *succ*(*j*))
⟨*proof*⟩

lemma *Union-in-VLimit*:
[| $X \in Vfrom(A,i)$; $Limit(i)$; $Transset(A)$ |] ==> $Union(X) \in Vfrom(A,i)$
⟨*proof*⟩

General theorem for membership in Vfrom(A,i) when i is a limit ordinal

lemma *in-VLimit*:
[| $a \in Vfrom(A,i)$; $b \in Vfrom(A,i)$; $Limit(i)$;
!!$x\ y\ j$. [| $j<i$; $1:j$; $x \in Vfrom(A,j)$; $y \in Vfrom(A,j)$ |]
==> $EX\ k.\ h(x,y) \in Vfrom(A,k)\ \&\ k<i$ |]
==> $h(a,b) \in Vfrom(A,i)$⟨*proof*⟩

23.5.1 Products

lemma *prod-in-Vfrom*:
[| $a \in Vfrom(A,j)$; $b \in Vfrom(A,j)$; $Transset(A)$ |]
==> $a*b \in Vfrom(A, succ(succ(succ(j))))$
⟨*proof*⟩

lemma *prod-in-VLimit*:
[| $a \in Vfrom(A,i)$; $b \in Vfrom(A,i)$; $Limit(i)$; $Transset(A)$ |]
==> $a*b \in Vfrom(A,i)$
⟨*proof*⟩

23.5.2 Disjoint Sums, or Quine Ordered Pairs

lemma *sum-in-Vfrom*:
[| $a \in Vfrom(A,j)$; $b \in Vfrom(A,j)$; $Transset(A)$; $1:j$ |]
==> $a+b \in Vfrom(A, succ(succ(succ(j))))$
⟨*proof*⟩

lemma *sum-in-VLimit*:
[| $a \in Vfrom(A,i)$; $b \in Vfrom(A,i)$; $Limit(i)$; $Transset(A)$ |]
==> $a+b \in Vfrom(A,i)$
⟨*proof*⟩

23.5.3 Function Space!

lemma *fun-in-Vfrom*:
[| $a \in Vfrom(A,j)$; $b \in Vfrom(A,j)$; $Transset(A)$ |] ==>
$a->b \in Vfrom(A, succ(succ(succ(succ(j)))))$
⟨*proof*⟩

lemma *fun-in-VLimit*:
[| $a \in Vfrom(A,i)$; $b \in Vfrom(A,i)$; $Limit(i)$; $Transset(A)$ |]
==> $a->b \in Vfrom(A,i)$
⟨*proof*⟩

lemma *Pow-in-Vfrom*:
[| $a \in Vfrom(A,j)$; $Transset(A)$ |] ==> $Pow(a) \in Vfrom(A, succ(succ(j)))$

⟨*proof*⟩

lemma *Pow-in-VLimit*:
[| $a \in Vfrom(A,i)$; $Limit(i)$; $Transset(A)$ |] ==> $Pow(a) \in Vfrom(A,i)$
⟨*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*]: $Ord(i) ==> \forall b.\ b \in Vset(i) --> rank(b) < i$
⟨*proof*⟩

lemma *VsetI-lemma* [*rule-format*]:
$Ord(i) ==> \forall b.\ rank(b) \in i --> b \in Vset(i)$
⟨*proof*⟩

lemma *VsetI*: $rank(x)<i ==> x \in Vset(i)$
⟨*proof*⟩

Merely a lemma for the next result

lemma *Vset-Ord-rank-iff*: $Ord(i) ==> b \in Vset(i) <-> rank(b) < i$
⟨*proof*⟩

lemma *Vset-rank-iff* [*simp*]: $b \in Vset(a) <-> rank(b) < rank(a)$
⟨*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*: $a <= Vset(rank(a))$
⟨*proof*⟩

lemma *Int-Vset-subset*:
[| !!i. $Ord(i) ==> a\ Int\ Vset(i) <= b$ |] ==> $a <= b$
⟨*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*: $A<=B ==> univ(A) <= univ(B)$
⟨*proof*⟩

lemma *Transset-univ*: $Transset(A) ==> Transset(univ(A))$
⟨*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*: $c <= univ(A) ==> c = (\bigcup i{\in}nat.\ c\ Int\ Vfrom(A,i))$
⟨*proof*⟩

lemma *univ-Int-Vfrom-subset*:

[| $a <= univ(X)$;
!!i. i:nat ==> a *Int* $Vfrom(X,i) <= b$ |]
==> $a <= b$
⟨*proof*⟩

lemma *univ-Int-Vfrom-eq*:
[| $a <= univ(X)$; $b <= univ(X)$;
!!i. i:nat ==> a *Int* $Vfrom(X,i) = b$ *Int* $Vfrom(X,i)$
|] ==> $a = b$
⟨*proof*⟩

23.8 Closure Properties for $univ(A)$

lemma *zero-in-univ*: $0 \in univ(A)$
⟨*proof*⟩

lemma *zero-subset-univ*: $\{0\} <= univ(A)$
⟨*proof*⟩

lemma *A-subset-univ*: $A <= univ(A)$
⟨*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: $univ(A)$ ==> $\{a\} \in univ(A)$
⟨*proof*⟩

lemma *doubleton-in-univ*:
[| a: $univ(A)$; b: $univ(A)$ |] ==> $\{a,b\} \in univ(A)$
⟨*proof*⟩

lemma *Pair-in-univ*:
[| a: $univ(A)$; b: $univ(A)$ |] ==> $<a,b> \in univ(A)$
⟨*proof*⟩

lemma *Union-in-univ*:
[| X: $univ(A)$; *Transset*(A) |] ==> $Union(X) \in univ(A)$
⟨*proof*⟩

lemma *product-univ*: $univ(A)*univ(A) <= univ(A)$
⟨*proof*⟩

23.8.2 The Natural Numbers

lemma *nat-subset-univ*: $nat <= univ(A)$
⟨*proof*⟩

n:nat ==¿ 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* ∈ *univ*(*A*)
⟨*proof*⟩

unused!

lemma *two-in-univ*: *2* ∈ *univ*(*A*)
⟨*proof*⟩

lemma *bool-subset-univ*: *bool* <= *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*: *univ*(*A*) ==> *Inl*(*a*) ∈ *univ*(*A*)
⟨*proof*⟩

lemma *Inr-in-univ*: *b*: *univ*(*A*) ==> *Inr*(*b*) ∈ *univ*(*A*)
⟨*proof*⟩

lemma *sum-univ*: *univ*(*C*)+*univ*(*C*) <= *univ*(*C*)
⟨*proof*⟩

lemmas *sum-subset-univ* = *subset-trans* [*OF sum-mono sum-univ*]

lemma *Sigma-subset-univ*:
[|*A* ⊆ *univ*(*D*); ⋀*x*. *x* ∈ *A* ⟹ *B*(*x*) ⊆ *univ*(*D*)|] ==> *Sigma*(*A*,*B*) ⊆ *univ*(*D*)
⟨*proof*⟩

23.9 Finite Branching Closure Properties

23.9.1 Closure under Finite Powerset

lemma *Fin-Vfrom-lemma*:
[| *b*: *Fin*(*Vfrom*(*A*,*i*)); *Limit*(*i*) |] ==> *EX j*. *b* <= *Vfrom*(*A*,*j*) & *j*<*i*
⟨*proof*⟩

lemma *Fin-VLimit*: *Limit*(*i*) ==> *Fin*(*Vfrom*(*A*,*i*)) <= *Vfrom*(*A*,*i*)
⟨*proof*⟩

lemmas *Fin-subset-VLimit* = *subset-trans* [*OF Fin-mono Fin-VLimit*]

lemma *Fin-univ*: *Fin*(*univ*(*A*)) <= *univ*(*A*)
⟨*proof*⟩

23.9.2 Closure under Finite Powers: Functions from a Natural Number

lemma *nat-fun-VLimit*:
[| *n*: *nat*; *Limit*(*i*) |] ==> *n* -> *Vfrom*(*A*,*i*) <= *Vfrom*(*A*,*i*)
⟨*proof*⟩

lemmas *nat-fun-subset-VLimit* = *subset-trans* [*OF Pi-mono nat-fun-VLimit*]

lemma *nat-fun-univ*: *n*: *nat* ==> *n* -> *univ*(*A*) <= *univ*(*A*)
⟨*proof*⟩

23.9.3 Closure under Finite Function Space

General but seldom-used version; normally the domain is fixed

lemma *FiniteFun-VLimit1*:
Limit(*i*) ==> *Vfrom*(*A*,*i*) -||> *Vfrom*(*A*,*i*) <= *Vfrom*(*A*,*i*)
⟨*proof*⟩

lemma *FiniteFun-univ1*: *univ*(*A*) -||> *univ*(*A*) <= *univ*(*A*)
⟨*proof*⟩

Version for a fixed domain

lemma *FiniteFun-VLimit*:
[| *W* <= *Vfrom*(*A*,*i*); *Limit*(*i*) |] ==> *W* -||> *Vfrom*(*A*,*i*) <= *Vfrom*(*A*,*i*)
⟨*proof*⟩

lemma *FiniteFun-univ*:
W <= *univ*(*A*) ==> *W* -||> *univ*(*A*) <= *univ*(*A*)
⟨*proof*⟩

lemma *FiniteFun-in-univ*:
[| *f*: *W* -||> *univ*(*A*); *W* <= *univ*(*A*) |] ==> *f* ∈ *univ*(*A*)
⟨*proof*⟩

Remove ¡= 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*} ∈ *Vfrom*(*X*,*succ*(*i*)); *Transset*(*X*) |]
==> *a* ∈ *Vfrom*(*X*,*i*) & *b* ∈ *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*:
[| <*a*,*b*> ∈ *Vfrom*(*X*,*succ*(*i*)); *Transset*(*X*) |]
==> *a* ∈ *Vfrom*(*X*,*i*) & *b* ∈ *Vfrom*(*X*,*i*)
⟨*proof*⟩

lemma *product-Int-Vfrom-subset*:
Transset(*X*) ==>
(*a*∗*b*) *Int* *Vfrom*(*X*, *succ*(*i*)) <= (*a* *Int* *Vfrom*(*X*,*i*)) ∗ (*b* *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* => *i* **where**
quniv(*A*) == *Pow*(*univ*(*eclose*(*A*)))

24.1 Properties involving Transset and Sum

lemma *Transset-includes-summands*:
[| *Transset*(*C*); *A*+*B* <= *C* |] ==> *A* <= *C* & *B* <= *C*
⟨*proof*⟩

lemma *Transset-sum-Int-subset*:
Transset(*C*) ==> (*A*+*B*) *Int C* <= (*A Int C*) + (*B Int C*)
⟨*proof*⟩

24.2 Introduction and Elimination Rules

lemma *qunivI*: *X* <= *univ*(*eclose*(*A*)) ==> *X* : *quniv*(*A*)
⟨*proof*⟩

lemma *qunivD*: *X* : *quniv*(*A*) ==> *X* <= *univ*(*eclose*(*A*))
⟨*proof*⟩

lemma *quniv-mono*: *A*<=*B* ==> *quniv*(*A*) <= *quniv*(*B*)
⟨*proof*⟩

24.3 Closure Properties

lemma *univ-eclose-subset-quniv*: *univ*(*eclose*(*A*)) <= *quniv*(*A*)
⟨*proof*⟩

lemma *univ-subset-quniv*: *univ*(*A*) <= *quniv*(*A*)
⟨*proof*⟩

lemmas *univ-into-quniv* = *univ-subset-quniv* [*THEN subsetD*, *standard*]

lemma *Pow-univ-subset-quniv*: *Pow*(*univ*(*A*)) <= *quniv*(*A*)
⟨*proof*⟩

lemmas *univ-subset-into-quniv* =
PowI [*THEN Pow-univ-subset-quniv* [*THEN subsetD*], *standard*]

lemmas *zero-in-quniv* = *zero-in-univ* [*THEN univ-into-quniv*, *standard*]
lemmas *one-in-quniv* = *one-in-univ* [*THEN univ-into-quniv*, *standard*]
lemmas *two-in-quniv* = *two-in-univ* [*THEN univ-into-quniv*, *standard*]

lemmas *A-subset-quniv* = *subset-trans* [*OF A-subset-univ univ-subset-quniv*]

lemmas *A-into-quniv* = *A-subset-quniv* [*THEN subsetD*, *standard*]

lemma *QPair-subset-univ*:
[| *a* <= *univ*(*A*); *b* <= *univ*(*A*) |] ==> <*a*;*b*> <= *univ*(*A*)
⟨*proof*⟩

24.4 Quine Disjoint Sum

lemma *QInl-subset-univ*: $a <= univ(A) ==> QInl(a) <= univ(A)$
⟨*proof*⟩

lemmas *naturals-subset-nat* =
Ord-nat [*THEN Ord-is-Transset*, *unfolded Transset-def*, *THEN bspec*, *standard*]

lemmas *naturals-subset-univ* =
subset-trans [*OF naturals-subset-nat nat-subset-univ*]

lemma *QInr-subset-univ*: $a <= univ(A) ==> QInr(a) <= univ(A)$
⟨*proof*⟩

24.5 Closure for Quine-Inspired Products and Sums

lemma *QPair-in-quniv*:
$[| a: quniv(A); b: quniv(A) |] ==> <a;b> : quniv(A)$
⟨*proof*⟩

lemma *QSigma-quniv*: $quniv(A) <*> quniv(A) <= quniv(A)$
⟨*proof*⟩

lemmas *QSigma-subset-quniv* = *subset-trans* [*OF QSigma-mono QSigma-quniv*]

lemma *quniv-QPair-D*:
$<a;b> : quniv(A) ==> a: quniv(A) \& b: quniv(A)$
⟨*proof*⟩

lemmas *quniv-QPair-E* = *quniv-QPair-D* [*THEN conjE*, *standard*]

lemma *quniv-QPair-iff*: $<a;b> : quniv(A) <-> a: quniv(A) \& b: quniv(A)$
⟨*proof*⟩

24.6 Quine Disjoint Sum

lemma *QInl-in-quniv*: $a: quniv(A) ==> QInl(a) : quniv(A)$
⟨*proof*⟩

lemma *QInr-in-quniv*: $b: quniv(A) ==> QInr(b) : quniv(A)$
⟨*proof*⟩

lemma *qsum-quniv*: $quniv(C) <+> quniv(C) <= quniv(C)$
⟨*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*) ==>
<*a*;*b*> *Int Vfrom*(*X*, *succ*(*i*)) <= <*a Int Vfrom*(*X*,*i*); *b Int Vfrom*(*X*,*i*)>
⟨*proof*⟩

24.8 "Take-Lemma" Rules

lemma *QPair-Int-Vfrom-subset*:
Transset(*X*) ==>
<*a*;*b*> *Int Vfrom*(*X*,*i*) <= <*a Int Vfrom*(*X*,*i*); *b Int Vfrom*(*X*,*i*)>
⟨*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*) ==> <*a*;*b*> *Int Vset*(*i*) <= (⋃*j*∈*i*. <*a Int Vset*(*j*); *b Int Vset*(*j*)>)
⟨*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* == *raw-add* (*natify*(*m*), *natify*(*n*))

definition
  diff :: [*i*,*i*]=>*i* (**infixl** #− *65*) **where**
    m #− *n* == *raw-diff* (*natify*(*m*), *natify*(*n*))

definition
  mult :: [*i*,*i*]=>*i* (**infixl** #∗ *70*) **where**
    m #∗ *n* == *raw-mult* (*natify*(*m*), *natify*(*n*))

definition
  raw-div :: [*i*,*i*]=>*i* **where**
    raw-div (*m*, *n*) ==
      transrec(*m*, %*j f*. *if j*<*n* | *n*=*0 then 0 else succ*(*f*‘(*j*#−*n*)))

definition
raw-mod :: [*i*,*i*]=>*i* **where**
raw-mod (*m*, *n*) ==
transrec(*m*, %*j f*. *if j*<*n* | *n*=*0 then j else f*‘(*j*#−*n*))

definition
div :: [*i*,*i*]=>*i* (**infixl** *div* *70*) **where**
m div n == *raw-div* (*natify*(*m*), *natify*(*n*))

definition
mod :: [*i*,*i*]=>*i* (**infixl** *mod* *70*) **where**
m mod n == *raw-mod* (*natify*(*m*), *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* ∈ *nat* |] ==> ∃*j*∈*nat*. *k* = *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*]: *pred*(*succ*(*y*)) = *y*
⟨*proof*⟩

lemma *natify-succ*: *natify*(*succ*(*x*)) = *succ*(*natify*(*x*))
⟨*proof*⟩

lemma *natify-0* [*simp*]: *natify*(*0*) = *0*
⟨*proof*⟩

lemma *natify-non-succ*: ∀*z*. *x* ~= *succ*(*z*) ==> *natify*(*x*) = *0*
⟨*proof*⟩

lemma *natify-in-nat* [*iff*,*TC*]: *natify*(*x*) ∈ *nat*
⟨*proof*⟩

lemma *natify-ident* [*simp*]: *n* ∈ *nat* ==> *natify*(*n*) = *n*
⟨*proof*⟩

lemma *natify-eqE*: $[|natify(x) = y;\ \ x \in nat|] ==> x{=}y$
⟨*proof*⟩

lemma *natify-idem* [*simp*]: $natify(natify(x)) = natify(x)$
⟨*proof*⟩

lemma *add-natify1* [*simp*]: $natify(m) \#+ n = m \#+ n$
⟨*proof*⟩

lemma *add-natify2* [*simp*]: $m \#+ natify(n) = m \#+ n$
⟨*proof*⟩

lemma *mult-natify1* [*simp*]: $natify(m) \#* n = m \#* n$
⟨*proof*⟩

lemma *mult-natify2* [*simp*]: $m \#* natify(n) = m \#* n$
⟨*proof*⟩

lemma *diff-natify1* [*simp*]: $natify(m) \#- n = m \#- n$
⟨*proof*⟩

lemma *diff-natify2* [*simp*]: $m \#- natify(n) = m \#- n$
⟨*proof*⟩

lemma *mod-natify1* [*simp*]: $natify(m)\ mod\ n = m\ mod\ n$
⟨*proof*⟩

lemma *mod-natify2* [*simp*]: $m\ mod\ natify(n) = m\ mod\ n$
⟨*proof*⟩

lemma *div-natify1* [*simp*]: $natify(m)\ div\ n = m\ div\ n$
⟨*proof*⟩

lemma *div-natify2* [*simp*]: $m\ div\ natify(n) = m\ div\ n$

⟨*proof*⟩

26.2 Typing rules

lemma *raw-add-type*: [| $m \in nat$; $n \in nat$ |] ==> *raw-add* $(m, n) \in nat$
⟨*proof*⟩

lemma *add-type* [*iff*, *TC*]: $m \#+ n \in nat$
⟨*proof*⟩

lemma *raw-mult-type*: [| $m \in nat$; $n \in nat$ |] ==> *raw-mult* $(m, n) \in nat$
⟨*proof*⟩

lemma *mult-type* [*iff*, *TC*]: $m \#* n \in nat$
⟨*proof*⟩

lemma *raw-diff-type*: [| $m \in nat$; $n \in nat$ |] ==> *raw-diff* $(m, n) \in nat$
⟨*proof*⟩

lemma *diff-type* [*iff*, *TC*]: $m \#- n \in nat$
⟨*proof*⟩

lemma *diff-0-eq-0* [*simp*]: $0 \#- n = 0$
⟨*proof*⟩

lemma *diff-succ-succ* [*simp*]: $succ(m) \#- succ(n) = m \#- n$
⟨*proof*⟩

declare *raw-diff-succ* [*simp del*]

lemma *diff-0* [*simp*]: $m \#- 0 = natify(m)$
⟨*proof*⟩

lemma *diff-le-self*: $m \in nat$ ==> $(m \#- n)$ *le* m
⟨*proof*⟩

26.3 Addition

lemma *add-0-natify* [*simp*]: $0 \#+ m = natify(m)$
⟨*proof*⟩

lemma *add-succ* [*simp*]: $succ(m) \#+ n = succ(m \#+ n)$

⟨*proof*⟩

lemma *add-0*: $m \in nat \Longrightarrow 0 \mathbin{\#+} m = m$
⟨*proof*⟩

lemma *add-assoc*: $(m \mathbin{\#+} n) \mathbin{\#+} k = m \mathbin{\#+} (n \mathbin{\#+} k)$
⟨*proof*⟩

lemma *add-0-right-natify* [*simp*]: $m \mathbin{\#+} 0 = natify(m)$
⟨*proof*⟩

lemma *add-succ-right* [*simp*]: $m \mathbin{\#+} succ(n) = succ(m \mathbin{\#+} n)$
⟨*proof*⟩

lemma *add-0-right*: $m \in nat \Longrightarrow m \mathbin{\#+} 0 = m$
⟨*proof*⟩

lemma *add-commute*: $m \mathbin{\#+} n = n \mathbin{\#+} m$
⟨*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*:
$[\!| raw\text{-}add(k, m) = raw\text{-}add(k, n);\ k \in nat |\!] \Longrightarrow m=n$
⟨*proof*⟩

lemma *add-left-cancel-natify*: $k \mathbin{\#+} m = k \mathbin{\#+} n \Longrightarrow natify(m) = natify(n)$
⟨*proof*⟩

lemma *add-left-cancel*:
$[\!| i = j;\ i \mathbin{\#+} m = j \mathbin{\#+} n;\ m \in nat;\ n \in nat |\!] \Longrightarrow m = n$
⟨*proof*⟩

lemma *add-le-elim1-natify*: $k\#+m\ le\ k\#+n \Longrightarrow natify(m)\ le\ natify(n)$
⟨*proof*⟩

lemma *add-le-elim1*: $[\!| k\#+m\ le\ k\#+n;\ m \in nat;\ n \in nat |\!] \Longrightarrow m\ le\ n$
⟨*proof*⟩

lemma *add-lt-elim1-natify*: $k\#+m < k\#+n \Longrightarrow natify(m) < natify(n)$
⟨*proof*⟩

lemma *add-lt-elim1*: $[\!| k\#+m < k\#+n; m \in nat; n \in nat |\!] \Longrightarrow m < n$
⟨*proof*⟩

lemma *zero-less-add*: $[\!| n \in nat; m \in nat |\!] \Longrightarrow 0 < m \#+ n \longleftrightarrow (0<m \mid 0<n)$
⟨*proof*⟩

26.4 Monotonicity of Addition

lemma *add-lt-mono1*: $[\!| i<j; j \in nat |\!] \Longrightarrow i\#+k < j\#+k$
⟨*proof*⟩

strict, in second argument

lemma *add-lt-mono2*: $[\!| i<j; j \in nat |\!] \Longrightarrow k\#+i < k\#+j$
⟨*proof*⟩

A [clumsy] way of lifting ¡ monotonicity to $\leq$ monotonicity

lemma *Ord-lt-mono-imp-le-mono*:
assumes *lt-mono*: $!!i\ j.\ [\!| i<j; j:k |\!] \Longrightarrow f(i) < f(j)$
and *ford*: $!!i.\ i:k \Longrightarrow Ord(f(i))$
and *leij*: $i\ le\ j$
and *jink*: $j:k$
shows $f(i)\ le\ f(j)$
⟨*proof*⟩

$\leq$ monotonicity, 1st argument

lemma *add-le-mono1*: $[\!| i\ le\ j; j \in nat |\!] \Longrightarrow i\#+k\ le\ j\#+k$
⟨*proof*⟩

$\leq$ monotonicity, both arguments

lemma *add-le-mono*: $[\!| i\ le\ j; k\ le\ l; j \in nat; l \in nat |\!] \Longrightarrow i\#+k\ le\ j\#+l$
⟨*proof*⟩

Combinations of less-than and less-than-or-equals

lemma *add-lt-le-mono*: $[\!| i<j; k\leq l; j \in nat; l \in nat |\!] \Longrightarrow i\#+k < j\#+l$
⟨*proof*⟩

lemma *add-le-lt-mono*: $[\!| i\leq j; k<l; j \in nat; l \in nat |\!] \Longrightarrow i\#+k < j\#+l$
⟨*proof*⟩

Less-than: in other words, strict in both arguments

lemma *add-lt-mono*: $[\!| i<j; k<l; j \in nat; l \in nat |\!] \Longrightarrow i\#+k < j\#+l$
⟨*proof*⟩

lemma *diff-add-inverse*: $(n\#+m)\ \#-\ n = natify(m)$
⟨*proof*⟩

lemma *diff-add-inverse2*: $(m\#+n)\ \#-\ n = natify(m)$
⟨*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*]: $pred(0) = 0$
⟨*proof*⟩

lemma *eq-succ-imp-eq-m1*: $[|i = succ(j);\ i{\in}nat|] ==> j = i\ \#-\ 1\ \&\ j\ {\in}nat$
⟨*proof*⟩

lemma *pred-Un-distrib*:
$[|i{\in}nat;\ j{\in}nat|] ==> pred(i\ Un\ j) = pred(i)\ Un\ pred(j)$
⟨*proof*⟩

lemma *pred-type* [*TC*,*simp*]:
$i \in nat ==> pred(i) \in nat$
⟨*proof*⟩

lemma *nat-diff-pred*: $[|i{\in}nat;\ j{\in}nat|] ==> i\ \#-\ succ(j) = pred(i\ \#-\ j)$
⟨*proof*⟩

lemma *diff-succ-eq-pred*: $i\ \#-\ succ(j) = pred(i\ \#-\ j)$
⟨*proof*⟩

lemma *nat-diff-Un-distrib*:
$[|i{\in}nat;\ j{\in}nat;\ k{\in}nat|] ==> (i\ Un\ j)\ \#-\ k = (i\#-k)\ Un\ (j\#-k)$
⟨*proof*⟩

lemma *diff-Un-distrib*:
$[|i{\in}nat;\ j{\in}nat|] ==> (i\ Un\ j)\ \#-\ k = (i\#-k)\ Un\ (j\#-k)$
⟨*proof*⟩

We actually prove $i\ \#-\ j\ \#-\ k = i\ \#-\ (j\ \#+\ k)$

lemma *diff-diff-left* [*simplified*]:
$natify(i)\#-natify(j)\#-k = natify(i)\ \#-\ (natify(j)\#+k)$
⟨*proof*⟩

lemma *eq-add-iff*: $(u \#+ m = u \#+ n) <-> (0 \#+ m = natify(n))$
⟨*proof*⟩

lemma *less-add-iff*: $(u \#+ m < u \#+ n) <-> (0 \#+ m < natify(n))$
⟨*proof*⟩

lemma *diff-add-eq*: $((u \#+ m) \#- (u \#+ n)) = ((0 \#+ m) \#- n)$
⟨*proof*⟩

lemma *eq-cong2*: $u = u' ==> (t==u) == (t==u')$
⟨*proof*⟩

lemma *iff-cong2*: $u <-> u' ==> (t==u) == (t==u')$
⟨*proof*⟩

26.5 Multiplication

lemma *mult-0* [*simp*]: $0 \#* m = 0$
⟨*proof*⟩

lemma *mult-succ* [*simp*]: $succ(m) \#* n = n \#+ (m \#* n)$
⟨*proof*⟩

lemma *mult-0-right* [*simp*]: $m \#* 0 = 0$
⟨*proof*⟩

lemma *mult-succ-right* [*simp*]: $m \#* succ(n) = m \#+ (m \#* n)$
⟨*proof*⟩

lemma *mult-1-natify* [*simp*]: $1 \#* n = natify(n)$
⟨*proof*⟩

lemma *mult-1-right-natify* [*simp*]: $n \#* 1 = natify(n)$
⟨*proof*⟩

lemma *mult-1*: $n \in nat ==> 1 \#* n = n$
⟨*proof*⟩

lemma *mult-1-right*: $n \in nat ==> n \#* 1 = n$
⟨*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*:
[| $m \in nat$; $n \in nat$ |]
==> $(m < succ(n)) <-> (m = 0 \mid (\exists j \in nat.\ m = succ(j)\ \&\ j < n))$
⟨*proof*⟩

lemma *less-diff-conv* [*rule-format*]:
[| $j \in nat$; $k \in nat$ |] ==> $\forall i \in nat.\ (i < j \#- k) <-> (i \#+ k < j)$
⟨*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 le m*; *m*:*nat* |] ==> *n* #+ (*m*#−*n*) = *m*
⟨*proof*⟩

lemma *add-diff-inverse2*: [| *n le m*; *m*:*nat* |] ==> (*m*#−*n*) #+ *n* = *m*
⟨*proof*⟩

lemma *diff-succ*: [| *n le m*; *m*:*nat* |] ==> *succ*(*m*) #− *n* = *succ*(*m*#−*n*)
⟨*proof*⟩

lemma *zero-less-diff* [*simp*]:
[| *m*: *nat*; *n*: *nat* |] ==> *0* < (*n* #− *m*) <−> *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 le m*; *m*:*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*:*nat*; *n*:*nat* |] ==> *raw-mod* (*m*, *n*) : *nat*
⟨*proof*⟩

lemma *mod-type* [*TC*,*iff*]: *m mod n* : *nat*
⟨*proof*⟩

lemma *DIVISION-BY-ZERO-DIV*: *a div 0* = *0*
⟨*proof*⟩

lemma *DIVISION-BY-ZERO-MOD*: *a mod 0 = natify(a)*
⟨*proof*⟩

lemma *raw-mod-less*: *m<n ==> raw-mod (m,n) = m*
⟨*proof*⟩

lemma *mod-less* [*simp*]: [| *m<n*; *n : nat* |] *==> m mod n = m*
⟨*proof*⟩

lemma *raw-mod-geq*:
[| *0<n*; *n le m*; *m:nat* |] *==> raw-mod (m, n) = raw-mod (m#−n, n)*
⟨*proof*⟩

lemma *mod-geq*: [| *n le m*; *m:nat* |] *==> m mod n = (m#−n) mod n*
⟨*proof*⟩

27.3 Division

lemma *raw-div-type*: [| *m:nat*; *n:nat* |] *==> raw-div (m, n) : nat*
⟨*proof*⟩

lemma *div-type* [*TC*,*iff*]: *m div n : nat*
⟨*proof*⟩

lemma *raw-div-less*: *m<n ==> raw-div (m,n) = 0*
⟨*proof*⟩

lemma *div-less* [*simp*]: [| *m<n*; *n : nat* |] *==> m div n = 0*
⟨*proof*⟩

lemma *raw-div-geq*: [| *0<n*; *n le m*; *m:nat* |] *==> raw-div(m,n) = succ(raw-div(m#−n, n))*
⟨*proof*⟩

lemma *div-geq* [*simp*]:
[| *0<n*; *n le m*; *m:nat* |] *==> m div n = succ ((m#−n) div n)*
⟨*proof*⟩

declare *div-less* [*simp*] *div-geq* [*simp*]

lemma *mod-div-lemma*: [| *m*: *nat*; *n*: *nat* |] *==> (m div n)#∗n #+ m mod n = m*
⟨*proof*⟩

lemma *mod-div-equality-natify*: *(m div n)#∗n #+ m mod n = natify(m)*
⟨*proof*⟩

lemma *mod-div-equality*: *m*: *nat* ==> (*m div n*)#∗*n* #+ *m mod n* = *m*
⟨*proof*⟩

27.4 Further Facts about Remainder

(mainly for mutilated chess board)

lemma *mod-succ-lemma*:
[| *0*<*n*; *m*:*nat*; *n*:*nat* |]
==> *succ*(*m*) *mod n* = (*if succ*(*m mod n*) = *n then 0 else succ*(*m mod n*))
⟨*proof*⟩

lemma *mod-succ*:
n:*nat* ==> *succ*(*m*) *mod n* = (*if succ*(*m mod n*) = *n then 0 else succ*(*m mod n*))
⟨*proof*⟩

lemma *mod-less-divisor*: [| *0*<*n*; *n*:*nat* |] ==> *m mod n* < *n*
⟨*proof*⟩

lemma *mod-1-eq* [*simp*]: *m mod 1* = *0*
⟨*proof*⟩

lemma *mod2-cases*: *b*<*2* ==> *k mod 2* = *b* | *k mod 2* = (*if b*=*1 then 0 else 1*)
⟨*proof*⟩

lemma *mod2-succ-succ* [*simp*]: *succ*(*succ*(*m*)) *mod 2* = *m mod 2*
⟨*proof*⟩

lemma *mod2-add-more* [*simp*]: (*m*#+*m*#+*n*) *mod 2* = *n mod 2*
⟨*proof*⟩

lemma *mod2-add-self* [*simp*]: (*m*#+*m*) *mod 2* = *0*
⟨*proof*⟩

27.5 Additional theorems about ≤

lemma *add-le-self*: *m*:*nat* ==> *m le* (*m* #+ *n*)
⟨*proof*⟩

lemma *add-le-self2*: *m*:*nat* ==> *m le* (*n* #+ *m*)
⟨*proof*⟩

lemma *mult-le-mono1*: [| *i le j*; *j*:*nat* |] ==> (*i*#∗*k*) *le* (*j*#∗*k*)
⟨*proof*⟩

lemma *mult-le-mono*: [| *i le j*; *k le l*; *j*:*nat*; *l*:*nat* |] ==> *i*#∗*k le j*#∗*l*

⟨*proof*⟩

lemma *mult-lt-mono2*: [| *i*<*j*; *0*<*k*; *j*:*nat*; *k*:*nat* |] ==> *k*#∗*i* < *k*#∗*j*
⟨*proof*⟩

lemma *mult-lt-mono1*: [| *i*<*j*; *0*<*k*; *j*:*nat*; *k*:*nat* |] ==> *i*#∗*k* < *j*#∗*k*
⟨*proof*⟩

lemma *add-eq-0-iff* [*iff*]: *m*#+*n* = *0* <−> *natify*(*m*)=*0* & *natify*(*n*)=*0*
⟨*proof*⟩

lemma *zero-lt-mult-iff* [*iff*]: *0* < *m*#∗*n* <−> *0* < *natify*(*m*) & *0* < *natify*(*n*)
⟨*proof*⟩

lemma *mult-eq-1-iff* [*iff*]: *m*#∗*n* = *1* <−> *natify*(*m*)=*1* & *natify*(*n*)=*1*
⟨*proof*⟩

lemma *mult-is-zero*: [|*m*: *nat*; *n*: *nat*|] ==> (*m* #∗ *n* = *0*) <−> (*m* = *0* | *n* = *0*)
⟨*proof*⟩

lemma *mult-is-zero-natify* [*iff*]:
(*m* #∗ *n* = *0*) <−> (*natify*(*m*) = *0* | *natify*(*n*) = *0*)
⟨*proof*⟩

27.6 Cancellation Laws for Common Factors in Comparisons

lemma *mult-less-cancel-lemma*:
[| *k*: *nat*; *m*: *nat*; *n*: *nat* |] ==> (*m*#∗*k* < *n*#∗*k*) <−> (*0*<*k* & *m*<*n*)
⟨*proof*⟩

lemma *mult-less-cancel2* [*simp*]:
(*m*#∗*k* < *n*#∗*k*) <−> (*0* < *natify*(*k*) & *natify*(*m*) < *natify*(*n*))
⟨*proof*⟩

lemma *mult-less-cancel1* [*simp*]:
(*k*#∗*m* < *k*#∗*n*) <−> (*0* < *natify*(*k*) & *natify*(*m*) < *natify*(*n*))
⟨*proof*⟩

lemma *mult-le-cancel2* [*simp*]: (*m*#∗*k* *le* *n*#∗*k*) <−> (*0* < *natify*(*k*) −−> *natify*(*m*) *le* *natify*(*n*))
⟨*proof*⟩

lemma *mult-le-cancel1* [*simp*]: (*k*#∗*m* *le* *k*#∗*n*) <−> (*0* < *natify*(*k*) −−> *natify*(*m*) *le* *natify*(*n*))
⟨*proof*⟩

lemma *mult-le-cancel-le1*: $k : nat ==> k \#* m \ le \ k \longleftrightarrow (0 < k \longrightarrow natify(m) \ le \ 1)$
⟨*proof*⟩

lemma *Ord-eq-iff-le*: [| $Ord(m)$; $Ord(n)$ |] ==> $m=n <-> (m \ le \ n \ \& \ n \ le \ m)$
⟨*proof*⟩

lemma *mult-cancel2-lemma*:
[| k: nat; m: nat; n: nat |] ==> $(m\#*k = n\#*k) <-> (m=n \mid k=0)$
⟨*proof*⟩

lemma *mult-cancel2* [*simp*]:
$(m\#*k = n\#*k) <-> (natify(m) = natify(n) \mid natify(k) = 0)$
⟨*proof*⟩

lemma *mult-cancel1* [*simp*]:
$(k\#*m = k\#*n) <-> (natify(m) = natify(n) \mid natify(k) = 0)$
⟨*proof*⟩

lemma *div-cancel-raw*:
[| $0<n$; $0<k$; k:nat; m:nat; n:nat |] ==> $(k\#*m) \ div \ (k\#*n) = m \ div \ n$
⟨*proof*⟩

lemma *div-cancel*:
[| $0 < natify(n)$; $0 < natify(k)$ |] ==> $(k\#*m) \ div \ (k\#*n) = m \ div \ n$
⟨*proof*⟩

27.7 More Lemmas about Remainder

lemma *mult-mod-distrib-raw*:
[| k:nat; m:nat; n:nat |] ==> $(k\#*m) \ mod \ (k\#*n) = k \#* (m \ mod \ n)$
⟨*proof*⟩

lemma *mod-mult-distrib2*: $k \#* (m \ mod \ n) = (k\#*m) \ mod \ (k\#*n)$
⟨*proof*⟩

lemma *mult-mod-distrib*: $(m \ mod \ n) \#* k = (m\#*k) \ mod \ (n\#*k)$
⟨*proof*⟩

lemma *mod-add-self2-raw*: $n \in nat ==> (m \#+ n) \ mod \ n = m \ mod \ n$
⟨*proof*⟩

lemma *mod-add-self2* [*simp*]: $(m \#+ n) \ mod \ n = m \ mod \ n$
⟨*proof*⟩

lemma *mod-add-self1* [*simp*]: $(n\#+m) \ mod \ n = m \ mod \ n$

⟨*proof*⟩

lemma *mod-mult-self1-raw*: $k \in nat \Longrightarrow (m \#+ k\#*n)\ mod\ n = m\ mod\ n$
⟨*proof*⟩

lemma *mod-mult-self1* [*simp*]: $(m \#+ k\#*n)\ mod\ n = m\ mod\ n$
⟨*proof*⟩

lemma *mod-mult-self2* [*simp*]: $(m \#+ n\#*k)\ mod\ n = m\ mod\ n$
⟨*proof*⟩

lemma *mult-eq-self-implies-10*: $m = m\#*n \Longrightarrow natify(n)=1 \mid m=0$
⟨*proof*⟩

lemma *less-imp-succ-add* [*rule-format*]:
$[\!| m<n; n: nat |\!] \Longrightarrow EX\ k: nat.\ n = succ(m\#+k)$
⟨*proof*⟩

lemma *less-iff-succ-add*:
$[\!| m: nat; n: nat |\!] \Longrightarrow (m<n) <-> (EX\ k: nat.\ n = succ(m\#+k))$
⟨*proof*⟩

lemma *add-lt-elim2*:
$[\![a \#+ d = b \#+ c; a < b; b \in nat; c \in nat; d \in nat]\!] \Longrightarrow c < d$
⟨*proof*⟩

lemma *add-le-elim2*:
$[\![a \#+ d = b \#+ c; a\ le\ b; b \in nat; c \in nat; d \in nat]\!] \Longrightarrow c\ le\ d$
⟨*proof*⟩

27.7.1 More Lemmas About Difference

lemma *diff-is-0-lemma*:
$[\!| m: nat; n: nat |\!] \Longrightarrow m \#- n = 0 <-> m\ le\ n$
⟨*proof*⟩

lemma *diff-is-0-iff*: $m \#- n = 0 <-> natify(m)\ le\ natify(n)$
⟨*proof*⟩

lemma *nat-lt-imp-diff-eq-0*:
$[\!| a:nat; b:nat; a<b |\!] \Longrightarrow a \#- b = 0$
⟨*proof*⟩

lemma *raw-nat-diff-split*:
$[\!| a:nat; b:nat |\!] \Longrightarrow$
$(P(a \#- b)) <-> ((a < b --> P(0)) \&\ (ALL\ d:nat.\ a = b \#+ d --> P(d)))$
⟨*proof*⟩

lemma *nat-diff-split*:
(*P*(*a* #− *b*)) <−>
(*natify*(*a*) < *natify*(*b*) −−>*P*(*0*)) & (*ALL d*:*nat*. *natify*(*a*) = *b* #+ *d* −−> *P*(*d*))
⟨*proof*⟩

Difference and less-than

lemma *diff-lt-imp-lt*: [|(*k*#−*i*) < (*k*#−*j*); *i*∈*nat*; *j*∈*nat*; *k*∈*nat*|] ==> *j*<*i*
⟨*proof*⟩

lemma *lt-imp-diff-lt*: [|*j*<*i*; *i*≤*k*; *k*∈*nat*|] ==> (*k*#−*i*) < (*k*#−*j*)
⟨*proof*⟩

lemma *diff-lt-iff-lt*: [|*i*≤*k*; *j*∈*nat*; *k*∈*nat*|] ==> (*k*#−*i*) < (*k*#−*j*) <−> *j*<*i*
⟨*proof*⟩

end

28 List-ZF: Lists in Zermelo-Fraenkel Set Theory

theory *List-ZF* **imports** *Datatype-ZF ArithSimp* **begin**

consts
list :: *i*=>*i*

datatype
list(*A*) = *Nil* | *Cons* (*a*:*A*, *l*: *list*(*A*))

syntax
[] :: *i* ([])
@*List* :: *is* => *i* ([(-)])

translations
[*x*, *xs*] == *Cons*(*x*, [*xs*])
[*x*] == *Cons*(*x*, [])
[] == *Nil*

consts
length :: *i*=>*i*
hd :: *i*=>*i*
tl :: *i*=>*i*

primrec
length([]) = *0*

$length(Cons(a,l)) = succ(length(l))$

primrec
$hd([]) = 0$
$hd(Cons(a,l)) = a$

primrec
$tl([]) = []$
$tl(Cons(a,l)) = l$

consts
map :: $[i=>i,\ i] => i$
$set\text{-}of\text{-}list$ :: $i=>i$
app :: $[i,i]=>i$ (**infixr** @ *60*)

primrec
$map(f,[]) = []$
$map(f,Cons(a,l)) = Cons(f(a),\ map(f,l))$

primrec
$set\text{-}of\text{-}list([]) = 0$
$set\text{-}of\text{-}list(Cons(a,l)) = cons(a,\ set\text{-}of\text{-}list(l))$

primrec
$app\text{-}Nil$: $[] @ ys = ys$
$app\text{-}Cons$: $(Cons(a,l)) @ ys = Cons(a,\ l @ ys)$

consts
rev :: $i=>i$
$flat$ :: $i=>i$
$list\text{-}add$ :: $i=>i$

primrec
$rev([]) = []$
$rev(Cons(a,l)) = rev(l) @ [a]$

primrec
$flat([]) = []$
$flat(Cons(l,ls)) = l @ flat(ls)$

primrec
$list\text{-}add([]) = 0$
$list\text{-}add(Cons(a,l)) = a \#+ list\text{-}add(l)$

consts
$drop$ :: $[i,i]=>i$

primrec
drop-0: $drop(0,l) = l$
drop-succ: $drop(succ(i),\ l) = tl\ (drop(i,l))$

definition

$take :: [i,i]=>i$ **where**
$take(n,\ as) == list\text{-}rec(lam\ n{:}nat.\ [],$
$\quad \%a\ l\ r.\ lam\ n{:}nat.\ nat\text{-}case([],\ \%m.\ Cons(a,\ r`m),\ n),\ as)`n$

definition
$nth :: [i,\ i]=>i$ **where**
— returns the (n+1)th element of a list, or 0 if the list is too short.
$nth(n,\ as) == list\text{-}rec(lam\ n{:}nat.\ 0,$
$\quad \%a\ l\ r.\ lam\ n{:}nat.\ nat\text{-}case(a,\ \%m.\ r`m,\ n),\ as)\ `\ n$

definition
$list\text{-}update :: [i,\ i,\ i]=>i$ **where**
$list\text{-}update(xs,\ i,\ v) == list\text{-}rec(lam\ n{:}nat.\ Nil,$
$\quad \%u\ us\ vs.\ lam\ n{:}nat.\ nat\text{-}case(Cons(v,\ us),\ \%m.\ Cons(u,\ vs`m),\ n),\ xs)`i$

consts
$filter :: [i=>o,\ i] => i$
$upt :: [i,\ i] =>i$

primrec
$filter(P,\ Nil) = Nil$
$filter(P,\ Cons(x,\ xs)) =$
$\quad (if\ P(x)\ then\ Cons(x,\ filter(P,\ xs))\ else\ filter(P,\ xs))$

primrec
$upt(i,\ 0) = Nil$
$upt(i,\ succ(j)) = (if\ i\ le\ j\ then\ upt(i,\ j)@[j]\ else\ Nil)$

definition
$min :: [i,i] =>i$ **where**
$\quad min(x,\ y) == (if\ x\ le\ y\ then\ x\ else\ y)$

definition
$max :: [i,\ i] =>i$ **where**
$\quad max(x,\ y) == (if\ x\ le\ y\ then\ y\ else\ x)$

declare $list.intros\ [simp,TC]$

inductive-cases *ConsE*: $Cons(a,l) : list(A)$

lemma *Cons-type-iff* [*simp*]: $Cons(a,l) \in list(A) <-> a \in A \;\&\; l \in list(A)$
⟨*proof*⟩

lemma *Cons-iff*: $Cons(a,l)=Cons(a',l') <-> a=a' \;\&\; l=l'$
⟨*proof*⟩

lemma *Nil-Cons-iff*: $\sim Nil=Cons(a,l)$
⟨*proof*⟩

lemma *list-unfold*: $list(A) = \{0\} + (A * list(A))$
⟨*proof*⟩

lemma *list-mono*: $A<=B ==> list(A) <= list(B)$
⟨*proof*⟩

lemma *list-univ*: $list(univ(A)) <= univ(A)$
⟨*proof*⟩

lemmas *list-subset-univ* = *subset-trans* [*OF list-mono list-univ*]

lemma *list-into-univ*: $[|\ l: list(A);\ \ A <= univ(B)\ |] ==> l: univ(B)$
⟨*proof*⟩

lemma *list-case-type*:
$[|\ l: list(A);$
$c: C(Nil);$
$!!x\ y.\ [|\ x: A;\ \ y: list(A)\ |] ==> h(x,y): C(Cons(x,y))$
$|] ==> list\text{-}case(c,h,l) : C(l)$
⟨*proof*⟩

lemma *list-0-triv*: $list(0) = \{Nil\}$
⟨*proof*⟩

lemma *tl-type*: $l: list(A) ==> tl(l) : list(A)$
⟨*proof*⟩

lemma *drop-Nil* [*simp*]: i:*nat* ==> *drop*(i, *Nil*) = *Nil*
⟨*proof*⟩

lemma *drop-succ-Cons* [*simp*]: i:*nat* ==> *drop*(*succ*(i), *Cons*(a,l)) = *drop*(i,l)
⟨*proof*⟩

lemma *drop-type* [*simp*,*TC*]: [| i:*nat*; l: *list*(A) |] ==> *drop*(i,l) : *list*(A)
⟨*proof*⟩

declare *drop-succ* [*simp del*]

lemma *list-rec-type* [*TC*]:
[| l: *list*(A);
c: C(*Nil*);
!!x y r. [| x:A; y: *list*(A); r: C(y) |] ==> h(x,y,r): C(*Cons*(x,y))
|] ==> *list-rec*(c,h,l) : C(l)
⟨*proof*⟩

lemma *map-type* [*TC*]:
[| l: *list*(A); !!x. x: A ==> h(x): B |] ==> *map*(h,l) : *list*(B)
⟨*proof*⟩

lemma *map-type2* [*TC*]: l: *list*(A) ==> *map*(h,l) : *list*({h(u). u:A})
⟨*proof*⟩

lemma *length-type* [*TC*]: l: *list*(A) ==> *length*(l) : *nat*
⟨*proof*⟩

lemma *lt-length-in-nat*:
[|x < *length*(xs); xs ∈ *list*(A)|] ==> x ∈ *nat*
⟨*proof*⟩

lemma *app-type* [*TC*]: [| xs: *list*(A); ys: *list*(A) |] ==> xs@ys : *list*(A)
⟨*proof*⟩

lemma *rev-type* [*TC*]: xs: *list*(A) ==> *rev*(xs) : *list*(A)

⟨*proof*⟩

lemma *flat-type* [*TC*]: *ls*: *list*(*list*(*A*)) ==> *flat*(*ls*) : *list*(*A*)
⟨*proof*⟩

lemma *set-of-list-type* [*TC*]: *l*: *list*(*A*) ==> *set-of-list*(*l*) : *Pow*(*A*)
⟨*proof*⟩

lemma *set-of-list-append*:
xs: *list*(*A*) ==> *set-of-list* (*xs*@*ys*) = *set-of-list*(*xs*) *Un* *set-of-list*(*ys*)
⟨*proof*⟩

lemma *list-add-type* [*TC*]: *xs*: *list*(*nat*) ==> *list-add*(*xs*) : *nat*
⟨*proof*⟩

lemma *map-ident* [*simp*]: *l*: *list*(*A*) ==> *map*(%*u*. *u*, *l*) = *l*
⟨*proof*⟩

lemma *map-compose*: *l*: *list*(*A*) ==> *map*(*h*, *map*(*j*,*l*)) = *map*(%*u*. *h*(*j*(*u*)), *l*)
⟨*proof*⟩

lemma *map-app-distrib*: *xs*: *list*(*A*) ==> *map*(*h*, *xs*@*ys*) = *map*(*h*,*xs*) @ *map*(*h*,*ys*)
⟨*proof*⟩

lemma *map-flat*: *ls*: *list*(*list*(*A*)) ==> *map*(*h*, *flat*(*ls*)) = *flat*(*map*(*map*(*h*),*ls*))
⟨*proof*⟩

lemma *list-rec-map*:
l: *list*(*A*) ==>
list-rec(*c*, *d*, *map*(*h*,*l*)) =
list-rec(*c*, %*x xs r*. *d*(*h*(*x*), *map*(*h*,*xs*), *r*), *l*)
⟨*proof*⟩

lemmas *list-CollectD* = *Collect-subset* [*THEN* *list-mono*, *THEN* *subsetD*, *stan-*

dard]

lemma *map-list-Collect*: *l*: *list*({*x*:*A*. *h*(*x*)=*j*(*x*)}) ==> *map*(*h*,*l*) = *map*(*j*,*l*)
⟨*proof*⟩

lemma *length-map* [*simp*]: *xs*: *list*(*A*) ==> *length*(*map*(*h*,*xs*)) = *length*(*xs*)
⟨*proof*⟩

lemma *length-app* [*simp*]:
[| *xs*: *list*(*A*); *ys*: *list*(*A*) |]
==> *length*(*xs*@*ys*) = *length*(*xs*) #+ *length*(*ys*)
⟨*proof*⟩

lemma *length-rev* [*simp*]: *xs*: *list*(*A*) ==> *length*(*rev*(*xs*)) = *length*(*xs*)
⟨*proof*⟩

lemma *length-flat*:
ls: *list*(*list*(*A*)) ==> *length*(*flat*(*ls*)) = *list-add*(*map*(*length*,*ls*))
⟨*proof*⟩

lemma *drop-length-Cons* [*rule-format*]:
xs: *list*(*A*) ==>
∀ *x*. *EX z zs*. *drop*(*length*(*xs*), *Cons*(*x*,*xs*)) = *Cons*(*z*,*zs*)
⟨*proof*⟩

lemma *drop-length* [*rule-format*]:
l: *list*(*A*) ==> ∀ *i* ∈ *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*:
⋀*l*. [| *l* ∈ *list*(*A*);
∀ *l′* ∈ *list*(*A*). *length*(*l′*) < *length*(*l*) −−> *P*(*l′*)|]
==> *P*(*l*)
shows *n* ∈ *nat* ==> ∀ *l* ∈ *list*(*A*). *length*(*l*) < *n* −−> *P*(*l*)
⟨*proof*⟩

theorem *list-complete-induct*:
[| *l* ∈ *list*(*A*);
⋀*l*. [| *l* ∈ *list*(*A*);
∀ *l′* ∈ *list*(*A*). *length*(*l′*) < *length*(*l*) −−> *P*(*l′*)|]

$==> P(l)$
$|] ==> P(l)$
⟨*proof*⟩

lemma *min-sym*: $[|\ i{:}nat;\ j{:}nat\ |] ==> min(i,j){=}min(j,i)$
⟨*proof*⟩

lemma *min-type* [*simp*,*TC*]: $[|\ i{:}nat;\ j{:}nat\ |] ==> min(i,j){:}nat$
⟨*proof*⟩

lemma *min-0* [*simp*]: $i{:}nat ==> min(0,i) = 0$
⟨*proof*⟩

lemma *min-02* [*simp*]: $i{:}nat ==> min(i,\ 0) = 0$
⟨*proof*⟩

lemma *lt-min-iff*: $[|\ i{:}nat;\ j{:}nat;\ k{:}nat\ |] ==> i{<}min(j,k) <-> i{<}j\ \&\ i{<}k$
⟨*proof*⟩

lemma *min-succ-succ* [*simp*]:
$[|\ i{:}nat;\ j{:}nat\ |] ==>\ min(succ(i),\ succ(j)){=}\ succ(min(i,\ j))$
⟨*proof*⟩

lemma *filter-append* [*simp*]:
$xs{:}list(A) ==>$ *filter*$(P,\ xs@ys) =$ *filter*$(P,\ xs)\ @$ *filter*$(P,\ ys)$
⟨*proof*⟩

lemma *filter-type* [*simp*,*TC*]: $xs{:}list(A) ==>$ *filter*$(P,\ xs){:}list(A)$
⟨*proof*⟩

lemma *length-filter*: $xs{:}list(A) ==>$ *length*(*filter*$(P,\ xs))$ *le length*(xs)
⟨*proof*⟩

lemma *filter-is-subset*: $xs{:}list(A) ==>$ *set-of-list*(*filter*$(P,xs)) <=$ *set-of-list*(xs)
⟨*proof*⟩

lemma *filter-False* [*simp*]: $xs{:}list(A) ==>$ *filter*$(\%p.$ *False*$,\ xs) = Nil$
⟨*proof*⟩

lemma *filter-True* [*simp*]: $xs{:}list(A) ==>$ *filter*$(\%p.$ *True*$,\ xs) = xs$

⟨*proof*⟩

lemma *length-is-0-iff* [*simp*]: *xs*:*list*(*A*) ==> *length*(*xs*)=*0* <−> *xs*=*Nil*
⟨*proof*⟩

lemma *length-is-0-iff2* [*simp*]: *xs*:*list*(*A*) ==> *0* = *length*(*xs*) <−> *xs*=*Nil*
⟨*proof*⟩

lemma *length-tl* [*simp*]: *xs*:*list*(*A*) ==> *length*(*tl*(*xs*)) = *length*(*xs*) #− *1*
⟨*proof*⟩

lemma *length-greater-0-iff*: *xs*:*list*(*A*) ==> *0*<*length*(*xs*) <−> *xs* ~= *Nil*
⟨*proof*⟩

lemma *length-succ-iff*: *xs*:*list*(*A*) ==> *length*(*xs*)=*succ*(*n*) <−> (*EX y ys*. *xs*=*Cons*(*y*, *ys*) & *length*(*ys*)=*n*)
⟨*proof*⟩

lemma *append-is-Nil-iff* [*simp*]:
xs:*list*(*A*) ==> (*xs*@*ys* = *Nil*) <−> (*xs*=*Nil* & *ys* = *Nil*)
⟨*proof*⟩

lemma *append-is-Nil-iff2* [*simp*]:
xs:*list*(*A*) ==> (*Nil* = *xs*@*ys*) <−> (*xs*=*Nil* & *ys* = *Nil*)
⟨*proof*⟩

lemma *append-left-is-self-iff* [*simp*]:
xs:*list*(*A*) ==> (*xs*@*ys* = *xs*) <−> (*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*) −−> (*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*) −−> (*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) --> (xs@us = ys@vs) <-> (xs=ys & us=vs)
⟨*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) --> (xs@us = ys@vs) --> (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 ~= *Nil* --> *hd*(xs @ ys) = *hd*(xs)
⟨*proof*⟩

lemma *tl-append* [*rule-format*,*simp*]:
xs:*list*(A) ==> xs~=*Nil* --> *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*) ==> ∀ *ys* ∈ *list*(*A*). *rev*(*xs*)=*rev*(*ys*) <-> *xs*=*ys*
⟨*proof*⟩

lemma *rev-list-elim* [*rule-format*]:
xs:*list*(*A*) ==>
(*xs*=*Nil* --> *P*) --> (∀ *ys* ∈ *list*(*A*). ∀ *y* ∈ *A*. *xs* =*ys*@[*y*] -->*P*)-->*P*
⟨*proof*⟩

lemma *length-drop* [*rule-format*,*simp*]:
n:*nat* ==> ∀ *xs* ∈ *list*(*A*). *length*(*drop*(*n*, *xs*)) = *length*(*xs*) #− *n*
⟨*proof*⟩

lemma *drop-all* [*rule-format*,*simp*]:
n:*nat* ==> ∀ *xs* ∈ *list*(*A*). *length*(*xs*) *le* *n* --> *drop*(*n*, *xs*)=*Nil*
⟨*proof*⟩

lemma *drop-append* [*rule-format*]:
n:*nat* ==>
∀ *xs* ∈ *list*(*A*). *drop*(*n*, *xs*@*ys*) = *drop*(*n*,*xs*) @ *drop*(*n* #− *length*(*xs*), *ys*)
⟨*proof*⟩

lemma *drop-drop*:
m:*nat* ==> ∀ *xs* ∈ *list*(*A*). ∀ *n* ∈ *nat*. *drop*(*n*, *drop*(*m*, *xs*))=*drop*(*n* #+ *m*, *xs*)
⟨*proof*⟩

lemma *take-0* [*simp*]: *xs*:*list*(*A*) ==> *take*(*0*, *xs*) = *Nil*
⟨*proof*⟩

lemma *take-succ-Cons* [*simp*]:
n:*nat* ==> *take*(*succ*(*n*), *Cons*(*a*, *xs*)) = *Cons*(*a*, *take*(*n*, *xs*))
⟨*proof*⟩

lemma *take-Nil* [*simp*]: *n*:*nat* ==> *take*(*n*, *Nil*) = *Nil*
⟨*proof*⟩

lemma *take-all* [*rule-format*,*simp*]:

n:nat ==> $\forall\, xs \in list(A).\ length(xs)\ le\ n$ --> $take(n,\ xs) = xs$
⟨*proof*⟩

lemma *take-type* [*rule-format*,*simp*,*TC*]:
xs:$list(A)$ ==> $\forall\, n \in nat.\ take(n,\ xs)$:$list(A)$
⟨*proof*⟩

lemma *take-append* [*rule-format*,*simp*]:
xs:$list(A)$ ==>
$\forall\, ys \in list(A).\ \forall\, n \in nat.\ take(n,\ xs\ @\ ys) =$
$take(n,\ xs)\ @\ take(n\ \#-\ length(xs),\ ys)$
⟨*proof*⟩

lemma *take-take* [*rule-format*]:
m : nat ==>
$\forall\, xs \in list(A).\ \forall\, n \in nat.\ take(n,\ take(m,xs)) = take(min(n,\ m),\ xs)$
⟨*proof*⟩

lemma *nth-0* [*simp*]: $nth(0,\ Cons(a,\ l)) = a$
⟨*proof*⟩

lemma *nth-Cons* [*simp*]: n:nat ==> $nth(succ(n),\ Cons(a,l)) = nth(n,l)$
⟨*proof*⟩

lemma *nth-empty* [*simp*]: $nth(n,\ Nil) = 0$
⟨*proof*⟩

lemma *nth-type* [*rule-format*,*simp*,*TC*]:
xs:$list(A)$ ==> $\forall\, n.\ n < length(xs)$ --> $nth(n,xs)$: A
⟨*proof*⟩

lemma *nth-eq-0* [*rule-format*]:
xs:$list(A)$ ==> $\forall\, n \in nat.\ length(xs)\ le\ n$ --> $nth(n,xs) = 0$
⟨*proof*⟩

lemma *nth-append* [*rule-format*]:
xs:$list(A)$ ==>
$\forall\, n \in nat.\ nth(n,\ xs\ @\ ys) = (if\ n < length(xs)\ then\ nth(n,xs)$
$else\ nth(n\ \#-\ length(xs),\ ys))$
⟨*proof*⟩

lemma *set-of-list-conv-nth*:
xs:$list(A)$
==> $set\text{-}of\text{-}list(xs) = \{x$:$A.\ EX\ i$:$nat.\ i<length(xs)\ \&\ x = nth(i,xs)\}$
⟨*proof*⟩

lemma *nth-take-lemma* [*rule-format*]:
k:*nat* ==>
$\forall xs \in list(A).\ (\forall ys \in list(A).\ k\ le\ length(xs) \dashrightarrow k\ le\ length(ys) \dashrightarrow$
$(\forall i \in nat.\ i<k \dashrightarrow nth(i,xs) = nth(i,ys)) \dashrightarrow take(k,xs) = take(k,ys))$
⟨*proof*⟩

lemma *nth-equalityI* [*rule-format*]:
$[|\ xs$:*list*$(A);\ ys$:*list*$(A);\ length(xs) = length(ys);$
$\forall i \in nat.\ i < length(xs) \dashrightarrow nth(i,xs) = nth(i,ys)\ |]$
==> $xs = ys$
⟨*proof*⟩

lemma *take-equalityI* [*rule-format*]:
$[|\ xs$:*list*$(A);\ ys$:*list*$(A);\ (\forall i \in nat.\ take(i,\ xs) = take(i,ys))\ |]$
==> $xs = ys$
⟨*proof*⟩

lemma *nth-drop* [*rule-format*]:
n:*nat* ==> $\forall i \in nat.\ \forall xs \in list(A).\ nth(i,\ drop(n,\ xs)) = nth(n\ \#+\ i,\ xs)$
⟨*proof*⟩

lemma *take-succ* [*rule-format*]:
$xs \in list(A)$
==> $\forall i.\ i < length(xs) \dashrightarrow take(succ(i),\ xs) = take(i,xs)\ @\ [nth(i,\ xs)]$
⟨*proof*⟩

lemma *take-add* [*rule-format*]:
$[|xs \in list(A);\ j \in nat|]$
==> $\forall i \in nat.\ take(i\ \#+\ j,\ xs) = take(i,xs)\ @\ take(j,\ drop(i,xs))$
⟨*proof*⟩

lemma *length-take*:
$l \in list(A)$ ==> $\forall n \in nat.\ length(take(n,l)) = min(n,\ length(l))$
⟨*proof*⟩

28.1 The function zip

Crafty definition to eliminate a type argument

consts
zip-aux :: $[i,i]=>i$

primrec
zip-aux$(B,[]) =$
$(\lambda ys \in list(B).\ list\text{-}case([],\ \%y\ l.\ [],\ ys))$

zip-aux$(B,Cons(x,l)) =$

$(\lambda ys \in list(B).$
$\quad list\text{-}case(Nil,\ \%y\ zs.\ Cons(<x,y>,\ zip\text{-}aux(B,l)`zs),\ ys))$

definition
$zip :: [i,\ i]=>i$ **where**
$zip(xs,\ ys) == zip\text{-}aux(set\text{-}of\text{-}list(ys),xs)`ys$

lemma *list-on-set-of-list*: $xs \in list(A) ==> xs \in list(set\text{-}of\text{-}list(xs))$
⟨*proof*⟩

lemma *zip-Nil* [*simp*]: $ys{:}list(A) ==> zip(Nil,\ ys)=Nil$
⟨*proof*⟩

lemma *zip-Nil2* [*simp*]: $xs{:}list(A) ==> zip(xs,\ Nil)=Nil$
⟨*proof*⟩

lemma *zip-aux-unique* [*rule-format*]:
$[|B<=C;\ \ xs \in list(A)|]$
$\quad ==> \forall ys \in list(B).\ zip\text{-}aux(C,xs)\ `\ ys = zip\text{-}aux(B,xs)\ `\ ys$
⟨*proof*⟩

lemma *zip-Cons-Cons* [*simp*]:
$[|\ xs{:}list(A);\ ys{:}list(B);\ x{:}A;\ y{:}B\ |] ==>$
$\quad zip(Cons(x,xs),\ Cons(y,\ ys)) = Cons(<x,y>,\ zip(xs,\ ys))$
⟨*proof*⟩

lemma *zip-type* [*rule-format*,*simp*,*TC*]:
$xs{:}list(A) ==> \forall ys \in list(B).\ zip(xs,\ ys){:}list(A*B)$
⟨*proof*⟩

lemma *length-zip* [*rule-format*,*simp*]:
$xs{:}list(A) ==> \forall ys \in list(B).\ length(zip(xs,ys)) =$
$\qquad\qquad\qquad\qquad min(length(xs),\ length(ys))$
⟨*proof*⟩

lemma *zip-append1* [*rule-format*]:
$[|\ ys{:}list(A);\ zs{:}list(B)\ |] ==>$
$\forall xs \in list(A).\ zip(xs\ @\ ys,\ zs) =$
$\qquad zip(xs,\ take(length(xs),\ zs))\ @\ zip(ys,\ drop(length(xs),zs))$
⟨*proof*⟩

lemma *zip-append2* [*rule-format*]:
$[|\ xs{:}list(A);\ zs{:}list(B)\ |] ==> \forall ys \in list(B).\ zip(xs,\ ys@zs) =$
$\quad zip(take(length(ys),\ xs),\ ys)\ @\ zip(drop(length(ys),\ xs),\ zs)$
⟨*proof*⟩

lemma *zip-append* [*simp*]:
[| *length*(*xs*) = *length*(*us*); *length*(*ys*) = *length*(*vs*);
xs:*list*(*A*); *us*:*list*(*B*); *ys*:*list*(*A*); *vs*:*list*(*B*) |]
==> *zip*(*xs*@*ys*,*us*@*vs*) = *zip*(*xs*, *us*) @ *zip*(*ys*, *vs*)
⟨*proof*⟩

lemma *zip-rev* [*rule-format*,*simp*]:
ys:*list*(*B*) ==> ∀ *xs* ∈ *list*(*A*).
length(*xs*) = *length*(*ys*) −−> *zip*(*rev*(*xs*), *rev*(*ys*)) = *rev*(*zip*(*xs*, *ys*))
⟨*proof*⟩

lemma *nth-zip* [*rule-format*,*simp*]:
ys:*list*(*B*) ==> ∀ *i* ∈ *nat*. ∀ *xs* ∈ *list*(*A*).
i < *length*(*xs*) −−> *i* < *length*(*ys*) −−>
nth(*i*,*zip*(*xs*, *ys*)) = <*nth*(*i*,*xs*),*nth*(*i*, *ys*)>
⟨*proof*⟩

lemma *set-of-list-zip* [*rule-format*]:
[| *xs*:*list*(*A*); *ys*:*list*(*B*); *i*:*nat* |]
==> *set-of-list*(*zip*(*xs*, *ys*)) =
{<*x*, *y*>:*A*∗*B*. *EX* *i*:*nat*. *i* < *min*(*length*(*xs*), *length*(*ys*))
& *x* = *nth*(*i*, *xs*) & *y* = *nth*(*i*, *ys*)}
⟨*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:*nat* ==>
list-update(*Cons*(*x*, *xs*), *succ*(*n*), *v*)= *Cons*(*x*, *list-update*(*xs*, *n*, *v*))
⟨*proof*⟩

lemma *list-update-type* [*rule-format*,*simp*,*TC*]:
[| *xs*:*list*(*A*); *v*:*A* |] ==> ∀ *n* ∈ *nat*. *list-update*(*xs*, *n*, *v*):*list*(*A*)
⟨*proof*⟩

lemma *length-list-update* [*rule-format*,*simp*]:
xs:*list*(*A*) ==> ∀ *i* ∈ *nat*. *length*(*list-update*(*xs*, *i*, *v*))=*length*(*xs*)
⟨*proof*⟩

lemma *nth-list-update* [*rule-format*]:
[| *xs*:*list*(*A*) |] ==> ∀ *i* ∈ *nat*. ∀ *j* ∈ *nat*. *i* < *length*(*xs*) −−>

$nth(j,\ list\text{-}update(xs,\ i,\ x)) = (if\ i{=}j\ then\ x\ else\ nth(j,\ xs))$
⟨*proof*⟩

lemma *nth-list-update-eq* [*simp*]:
$[|\ i < length(xs);\ xs{:}list(A)\ |] ==> nth(i,\ list\text{-}update(xs,\ i{,}x)) = x$
⟨*proof*⟩

lemma *nth-list-update-neq* [*rule-format*,*simp*]:
$xs{:}list(A) ==>$
$\forall i \in nat.\ \forall j \in nat.\ i \sim= j --> nth(j,\ list\text{-}update(xs{,}i{,}x)) = nth(j{,}xs)$
⟨*proof*⟩

lemma *list-update-overwrite* [*rule-format*,*simp*]:
$xs{:}list(A) ==> \forall i \in nat.\ i < length(xs)$
$--> list\text{-}update(list\text{-}update(xs,\ i,\ x),\ i,\ y) = list\text{-}update(xs,\ i{,}y)$
⟨*proof*⟩

lemma *list-update-same-conv* [*rule-format*]:
$xs{:}list(A) ==>$
$\forall i \in nat.\ i < length(xs) -->$
$(list\text{-}update(xs,\ i,\ x) = xs) <-> (nth(i,\ xs) = x)$
⟨*proof*⟩

lemma *update-zip* [*rule-format*]:
$ys{:}list(B) ==>$
$\forall i \in nat.\ \forall xy \in A{*}B.\ \forall xs \in list(A).$
$length(xs) = length(ys) -->$
$list\text{-}update(zip(xs,\ ys),\ i,\ xy) = zip(list\text{-}update(xs,\ i,\ fst(xy)),$
$list\text{-}update(ys,\ i,\ snd(xy)))$
⟨*proof*⟩

lemma *set-update-subset-cons* [*rule-format*]:
$xs{:}list(A) ==>$
$\forall i \in nat.\ set\text{-}of\text{-}list(list\text{-}update(xs,\ i,\ x)) <= cons(x,\ set\text{-}of\text{-}list(xs))$
⟨*proof*⟩

lemma *set-of-list-update-subsetI*:
$[|\ set\text{-}of\text{-}list(xs) <= A;\ xs{:}list(A);\ x{:}A;\ i{:}nat|]$
$==> set\text{-}of\text{-}list(list\text{-}update(xs,\ i{,}x)) <= A$
⟨*proof*⟩

lemma *upt-rec*:
$j{:}nat ==> upt(i{,}j) = (if\ i{<}j\ then\ Cons(i,\ upt(succ(i),\ j))\ else\ Nil)$
⟨*proof*⟩

lemma *upt-conv-Nil* [*simp*]: $[|\ j\ le\ i;\ j{:}nat\ |] ==> upt(i{,}j) = Nil$

⟨*proof*⟩

lemma *upt-succ-append*:
[| *i le j*; *j*:*nat* |] ==> *upt*(*i*,*succ*(*j*)) = *upt*(*i*, *j*)@[*j*]
⟨*proof*⟩

lemma *upt-conv-Cons*:
[| *i*<*j*; *j*:*nat* |] ==> *upt*(*i*,*j*) = *Cons*(*i*,*upt*(*succ*(*i*),*j*))
⟨*proof*⟩

lemma *upt-type* [*simp*,*TC*]: *j*:*nat* ==> *upt*(*i*,*j*):*list*(*nat*)
⟨*proof*⟩

lemma *upt-add-eq-append*:
[| *i le j*; *j*:*nat*; *k*:*nat* |] ==> *upt*(*i*, *j* #+*k*) = *upt*(*i*,*j*)@*upt*(*j*,*j*#+*k*)
⟨*proof*⟩

lemma *length-upt* [*simp*]: [| *i*:*nat*; *j*:*nat* |] ==>*length*(*upt*(*i*,*j*)) = *j* #− *i*
⟨*proof*⟩

lemma *nth-upt* [*rule-format*,*simp*]:
[| *i*:*nat*; *j*:*nat*; *k*:*nat* |] ==> *i* #+ *k* < *j* −−> *nth*(*k*, *upt*(*i*,*j*)) = *i* #+ *k*
⟨*proof*⟩

lemma *take-upt* [*rule-format*,*simp*]:
[| *m*:*nat*; *n*:*nat* |] ==>
∀ *i* ∈ *nat*. *i* #+ *m le n* −−> *take*(*m*, *upt*(*i*,*n*)) = *upt*(*i*,*i*#+*m*)
⟨*proof*⟩

lemma *map-succ-upt*:
[| *m*:*nat*; *n*:*nat* |] ==> *map*(*succ*, *upt*(*m*,*n*))= *upt*(*succ*(*m*), *succ*(*n*))
⟨*proof*⟩

lemma *nth-map* [*rule-format*,*simp*]:
xs:*list*(*A*) ==>
∀ *n* ∈ *nat*. *n* < *length*(*xs*) −−> *nth*(*n*, *map*(*f*, *xs*)) = *f*(*nth*(*n*, *xs*))
⟨*proof*⟩

lemma *nth-map-upt* [*rule-format*]:
[| *m*:*nat*; *n*:*nat* |] ==>
∀ *i* ∈ *nat*. *i* < *n* #− *m* −−> *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* ∩ *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* ∈ *nat* ==> *length*(*repeat*(*a*,*n*)) = *n*
⟨*proof*⟩

lemma *repeat-succ-app*: *n* ∈ *nat* ==> *repeat*(*a*,*succ*(*n*)) = *repeat*(*a*,*n*) @ [*a*]
⟨*proof*⟩

lemma *repeat-type* [*TC*]: [|*a* ∈ *A*; *n* ∈ *nat*|] ==> *repeat*(*a*,*n*) ∈ *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 *converse*(*r*) *O r* = *r*

lemma *sym-trans-comp-subset*:
[| *sym*(*r*); *trans*(*r*) |] ==> *converse*(*r*) *O r* <= *r*
⟨*proof*⟩

lemma *refl-comp-subset*:
[| *refl*(*A*,*r*); *r* <= *A*∗*A* |] ==> *r* <= *converse*(*r*) *O* *r*
⟨*proof*⟩

lemma *equiv-comp-eq*:
equiv(*A*,*r*) ==> *converse*(*r*) *O* *r* = *r*
⟨*proof*⟩

lemma *comp-equivI*:
[| *converse*(*r*) *O* *r* = *r*; *domain*(*r*) = *A* |] ==> *equiv*(*A*,*r*)
⟨*proof*⟩

lemma *equiv-class-subset*:
[| *sym*(*r*); *trans*(*r*); <*a*,*b*>: *r* |] ==> *r*‘‘{*a*} <= *r*‘‘{*b*}
⟨*proof*⟩

lemma *equiv-class-eq*:
[| *equiv*(*A*,*r*); <*a*,*b*>: *r* |] ==> *r*‘‘{*a*} = *r*‘‘{*b*}
⟨*proof*⟩

lemma *equiv-class-self*:
[| *equiv*(*A*,*r*); *a*: *A* |] ==> *a*: *r*‘‘{*a*}
⟨*proof*⟩

lemma *subset-equiv-class*:
[| *equiv*(*A*,*r*); *r*‘‘{*b*} <= *r*‘‘{*a*}; *b*: *A* |] ==> <*a*,*b*>: *r*
⟨*proof*⟩

lemma *eq-equiv-class*: [| *r*‘‘{*a*} = *r*‘‘{*b*}; *equiv*(*A*,*r*); *b*: *A* |] ==> <*a*,*b*>: *r*
⟨*proof*⟩

lemma *equiv-class-nondisjoint*:
[| *equiv*(*A*,*r*); *x*: (*r*‘‘{*a*} *Int* *r*‘‘{*b*}) |] ==> <*a*,*b*>: *r*
⟨*proof*⟩

lemma *equiv-type*: *equiv*(*A*,*r*) ==> *r* <= *A*∗*A*
⟨*proof*⟩

lemma *equiv-class-eq-iff*:
equiv(*A*,*r*) ==> <*x*,*y*>: *r* <-> *r*‘‘{*x*} = *r*‘‘{*y*} & *x*:*A* & *y*:*A*
⟨*proof*⟩

lemma *eq-equiv-class-iff*:
[| *equiv*(*A*,*r*); *x*: *A*; *y*: *A* |] ==> *r*``{*x*} = *r*``{*y*} <-> <*x*,*y*>: *r*
⟨*proof*⟩

lemma *quotientI* [*TC*]: *x*:*A* ==> *r*``{*x*}: *A*//*r*
⟨*proof*⟩

lemma *quotientE*:
[| *X*: *A*//*r*; !!*x*. [| *X* = *r*``{*x*}; *x*:*A* |] ==> *P* |] ==> *P*
⟨*proof*⟩

lemma *Union-quotient*:
equiv(*A*,*r*) ==> *Union*(*A*//*r*) = *A*
⟨*proof*⟩

lemma *quotient-disj*:
[| *equiv*(*A*,*r*); *X*: *A*//*r*; *Y*: *A*//*r* |] ==> *X*=*Y* | (*X Int Y* <= *0*)
⟨*proof*⟩

29.2 Defining Unary Operations upon Equivalence Classes

lemma *UN-equiv-class*:
[| *equiv*(*A*,*r*); *b respects r*; *a*: *A* |] ==> (*UN x*:*r*``{*a*}. *b*(*x*)) = *b*(*a*)
⟨*proof*⟩

lemma *UN-equiv-class-type*:
[| *equiv*(*A*,*r*); *b respects r*; *X*: *A*//*r*; !!*x*. *x* : *A* ==> *b*(*x*) : *B* |]
==> (*UN x*:*X*. *b*(*x*)) : *B*
⟨*proof*⟩

lemma *UN-equiv-class-inject*:
[| *equiv*(*A*,*r*); *b respects r*;
(*UN x*:*X*. *b*(*x*))=(*UN y*:*Y*. *b*(*y*)); *X*: *A*//*r*; *Y*: *A*//*r*;
!!*x y*. [| *x*:*A*; *y*:*A*; *b*(*x*)=*b*(*y*) |] ==> <*x*,*y*>:*r* |]
==> *X*=*Y*
⟨*proof*⟩

29.3 Defining Binary Operations upon Equivalence Classes

lemma *congruent2-implies-congruent*:
[| *equiv*(*A*,*r1*); *congruent2*(*r1*,*r2*,*b*); *a*: *A* |] ==> *congruent*(*r2*,*b*(*a*))
⟨*proof*⟩

lemma *congruent2-implies-congruent-UN*:

[| *equiv*(*A1*,*r1*); *equiv*(*A2*,*r2*); *congruent2*(*r1*,*r2*,*b*); *a*: *A2* |] ==>
congruent(*r1*, %*x1*. ⋃ *x2* ∈ *r2*``{*a*}. *b*(*x1*,*x2*))
⟨*proof*⟩

lemma *UN-equiv-class2*:
[| *equiv*(*A1*,*r1*); *equiv*(*A2*,*r2*); *congruent2*(*r1*,*r2*,*b*); *a1*: *A1*; *a2*: *A2* |]
==> (⋃ *x1* ∈ *r1*``{*a1*}. ⋃ *x2* ∈ *r2*``{*a2*}. *b*(*x1*,*x2*)) = *b*(*a1*,*a2*)
⟨*proof*⟩

lemma *UN-equiv-class-type2*:
[| *equiv*(*A*,*r*); *b respects2 r*;
X1: *A*//*r*; *X2*: *A*//*r*;
!!*x1 x2*. [| *x1*: *A*; *x2*: *A* |] ==> *b*(*x1*,*x2*) : *B*
|] ==> (*UN x1*:*X1*. *UN x2*:*X2*. *b*(*x1*,*x2*)) : *B*
⟨*proof*⟩

lemma *congruent2I*:
[| *equiv*(*A1*,*r1*); *equiv*(*A2*,*r2*);
!! *y z w*. [| *w* ∈ *A2*; <*y*,*z*> ∈ *r1* |] ==> *b*(*y*,*w*) = *b*(*z*,*w*);
!! *y z w*. [| *w* ∈ *A1*; <*y*,*z*> ∈ *r2* |] ==> *b*(*w*,*y*) = *b*(*w*,*z*)
|] ==> *congruent2*(*r1*,*r2*,*b*)
⟨*proof*⟩

lemma *congruent2-commuteI*:
assumes *equivA*: *equiv*(*A*,*r*)
and *commute*: !! *y z*. [| *y*: *A*; *z*: *A* |] ==> *b*(*y*,*z*) = *b*(*z*,*y*)
and *congt*: !! *y z w*. [| *w*: *A*; <*y*,*z*>: *r* |] ==> *b*(*w*,*y*) = *b*(*w*,*z*)
shows *b respects2 r*
⟨*proof*⟩

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*

definition
raw-nat-of :: *i*=>*i* **where**
raw-nat-of(*z*) == *natify* (⋃<*x*,*y*>∈*z*. *x*#−*y*)

definition
nat-of :: *i*=>*i* **where**
nat-of(*z*) == *raw-nat-of* (*intify*(*z*))

definition
zmagnitude :: *i*=>*i* **where**
— could be replaced by an absolute value function from int to int?
zmagnitude(*z*) ==
THE m. *m*∈*nat* & ((~ *znegative*(*z*) & *z* = $# *m*) |
(*znegative*(*z*) & $− *z* = $# *m*))

definition
raw-zmult :: $[i,i]=>i$ **where**

raw-zmult(*z1*,*z2*) ==
$\bigcup p1 \in z1.\ \bigcup p2 \in z2.$ *split*(%*x1 y1*. *split*(%*x2 y2*.
intrel``{<*x1*#∗*x2* #+ *y1*#∗*y2*, *x1*#∗*y2* #+ *y1*#∗*x2*>}, *p2*), *p1*)

definition
zmult :: $[i,i]=>i$ (**infixl** $∗ *70*) **where**
z1 $∗ *z2* == *raw-zmult* (*intify*(*z1*),*intify*(*z2*))

definition
raw-zadd :: $[i,i]=>i$ **where**
raw-zadd (*z1*, *z2*) ==
$\bigcup z1 \in z1.\ \bigcup z2 \in z2.$ *let* <*x1*,*y1*>=*z1*; <*x2*,*y2*>=*z2*
in intrel``{<*x1*#+*x2*, *y1*#+*y2*>}

definition
zadd :: $[i,i]=>i$ (**infixl** $+ *65*) **where**
z1 $+ *z2* == *raw-zadd* (*intify*(*z1*),*intify*(*z2*))

definition
zdiff :: $[i,i]=>i$ (**infixl** $− *65*) **where**
z1 $− *z2* == *z1* $+ *zminus*(*z2*)

definition
zless :: $[i,i]=>o$ (**infixl** $< *50*) **where**
z1 $< *z2* == *znegative*(*z1* $− *z2*)

definition
zle :: $[i,i]=>o$ (**infixl** $<= *50*) **where**
z1 $<= *z2* == *z1* $< *z2* | *intify*(*z1*)=*intify*(*z2*)

notation (*xsymbols*)
zmult (**infixl** $× *70*) **and**
zle (**infixl** $≤ *50*) — less than or equals

notation (*HTML* **output**)
zmult (**infixl** $× *70*) **and**
zle (**infixl** $≤ *50*)

declare *quotientE* [*elim*!]

30.1 Proving that *intrel* is an equivalence relation

lemma *intrel-iff* [*simp*]:
<<*x1*,*y1*>,<*x2*,*y2*>>: *intrel* <−>

$x1 \in nat$ & $y1 \in nat$ & $x2 \in nat$ & $y2 \in nat$ & $x1 \# + y2 = x2 \# + y1$
⟨*proof*⟩

lemma *intrelI* [*intro*!]:
[| $x1 \# + y2 = x2 \# + y1$; $x1 \in nat$; $y1 \in nat$; $x2 \in nat$; $y2 \in nat$ |]
==> $<<x1,y1>,<x2,y2>>$: *intrel*
⟨*proof*⟩

lemma *intrelE* [*elim*!]:
[| *p*: *intrel*;
!!*x1 y1 x2 y2*. [| $p = <<x1,y1>,<x2,y2>>$; $x1 \# + y2 = x2 \# + y1$;
$x1 \in nat$; $y1 \in nat$; $x2 \in nat$; $y2 \in nat$ |] ==> *Q* |]
==> *Q*
⟨*proof*⟩

lemma *int-trans-lemma*:
[| $x1 \# + y2 = x2 \# + y1$; $x2 \# + y3 = x3 \# + y2$ |] ==> $x1 \# + y3 = x3 \# + y1$
⟨*proof*⟩

lemma *equiv-intrel*: *equiv*($nat * nat$, *intrel*)
⟨*proof*⟩

lemma *image-intrel-int*: [| $m \in nat$; $n \in nat$ |] ==> *intrel* `` $\{<m,n>\}$: *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*]: ($\$\# m = \$\# n$) <-> *natify*(*m*)=*natify*(*n*)
⟨*proof*⟩

lemma *int-of-inject*: [| $\$\# m = \$\# n$; $m \in nat$; $n \in nat$ |] ==> *m*=*n*
⟨*proof*⟩

lemma *intify-in-int* [*iff*,*TC*]: *intify*(*x*) : *int*
⟨*proof*⟩

lemma *intify-ident* [*simp*]: *n* : *int* ==> *intify*(*n*) = *n*

⟨*proof*⟩

30.2 Collapsing rules: to remove *intify* from arithmetic expressions

lemma *intify-idem* [*simp*]: *intify*(*intify*(*x*)) = *intify*(*x*)
⟨*proof*⟩

lemma *int-of-natify* [*simp*]: $# (*natify*(*m*)) = $# *m*
⟨*proof*⟩

lemma *zminus-intify* [*simp*]: $− (*intify*(*m*)) = $− *m*
⟨*proof*⟩

lemma *zadd-intify1* [*simp*]: *intify*(*x*) $+ *y* = *x* $+ *y*
⟨*proof*⟩

lemma *zadd-intify2* [*simp*]: *x* $+ *intify*(*y*) = *x* $+ *y*
⟨*proof*⟩

lemma *zdiff-intify1* [*simp*]:*intify*(*x*) $− *y* = *x* $− *y*
⟨*proof*⟩

lemma *zdiff-intify2* [*simp*]:*x* $− *intify*(*y*) = *x* $− *y*
⟨*proof*⟩

lemma *zmult-intify1* [*simp*]:*intify*(*x*) $∗ *y* = *x* $∗ *y*
⟨*proof*⟩

lemma *zmult-intify2* [*simp*]:*x* $∗ *intify*(*y*) = *x* $∗ *y*
⟨*proof*⟩

lemma *zless-intify1* [*simp*]:*intify*(*x*) $< *y* <−> *x* $< *y*
⟨*proof*⟩

lemma *zless-intify2* [*simp*]:*x* $< *intify*(*y*) <−> *x* $< *y*
⟨*proof*⟩

lemma *zle-intify1* [*simp*]:*intify*(*x*) $<= *y* <−> *x* $<= *y*
⟨*proof*⟩

lemma *zle-intify2* [*simp*]:*x* $<= *intify*(*y*) <-> *x* $<= *y*
⟨*proof*⟩

30.3 *zminus*: unary negation on *int*

lemma *zminus-congruent*: (%<*x*,*y*>. *intrel*``{<*y*,*x*>}) *respects intrel*
⟨*proof*⟩

lemma *raw-zminus-type*: *z* : *int* ==> *raw-zminus*(*z*) : *int*
⟨*proof*⟩

lemma *zminus-type* [*TC*,*iff*]: $−*z* : *int*
⟨*proof*⟩

lemma *raw-zminus-inject*:
[| *raw-zminus*(*z*) = *raw-zminus*(*w*); *z*: *int*; *w*: *int* |] ==> *z*=*w*
⟨*proof*⟩

lemma *zminus-inject-intify* [*dest*!]: $−*z* = $−*w* ==> *intify*(*z*) = *intify*(*w*)
⟨*proof*⟩

lemma *zminus-inject*: [| $−*z* = $−*w*; *z*: *int*; *w*: *int* |] ==> *z*=*w*
⟨*proof*⟩

lemma *raw-zminus*:
[| *x*∈*nat*; *y*∈*nat* |] ==> *raw-zminus*(*intrel*``{<*x*,*y*>}) = *intrel* `` {<*y*,*x*>}
⟨*proof*⟩

lemma *zminus*:
[| *x*∈*nat*; *y*∈*nat* |]
==> $− (*intrel*``{<*x*,*y*>}) = *intrel* `` {<*y*,*x*>}
⟨*proof*⟩

lemma *raw-zminus-zminus*: *z* : *int* ==> *raw-zminus* (*raw-zminus*(*z*)) = *z*
⟨*proof*⟩

lemma *zminus-zminus-intify* [*simp*]: $− ($− *z*) = *intify*(*z*)
⟨*proof*⟩

lemma *zminus-int0* [*simp*]: $− ($#*0*) = $#*0*
⟨*proof*⟩

lemma *zminus-zminus*: *z* : *int* ==> $− ($− *z*) = *z*
⟨*proof*⟩

30.4 *znegative*: the test for negative integers

lemma *znegative*: [| *x*∈*nat*; *y*∈*nat* |] ==> *znegative*(*intrel*``{<*x*,*y*>}) <-> *x*<*y*
⟨*proof*⟩

lemma *not-znegative-int-of* [*iff*]: ~ *znegative*($# *n*)
⟨*proof*⟩

lemma *znegative-zminus-int-of* [*simp*]: *znegative*($− $# *succ*(*n*))
⟨*proof*⟩

lemma *not-znegative-imp-zero*: ~ *znegative*($− $# *n*) ==> *natify*(*n*)=*0*
⟨*proof*⟩

30.5 *nat-of*: Coercion of an Integer to a Natural Number

lemma *nat-of-intify* [*simp*]: *nat-of*(*intify*(*z*)) = *nat-of*(*z*)
⟨*proof*⟩

lemma *nat-of-congruent*: (λ*x*. (λ⟨*x*,*y*⟩. *x* #− *y*)(*x*)) *respects intrel*
⟨*proof*⟩

lemma *raw-nat-of*:
[| *x*∈*nat*; *y*∈*nat* |] ==> *raw-nat-of*(*intrel*``{<*x*,*y*>}) = *x*#−*y*
⟨*proof*⟩

lemma *raw-nat-of-int-of*: *raw-nat-of*($# *n*) = *natify*(*n*)
⟨*proof*⟩

lemma *nat-of-int-of* [*simp*]: *nat-of*($# *n*) = *natify*(*n*)
⟨*proof*⟩

lemma *raw-nat-of-type*: *raw-nat-of*(*z*) ∈ *nat*
⟨*proof*⟩

lemma *nat-of-type* [*iff*,*TC*]: *nat-of*(*z*) ∈ *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* ==> $#*x* = $# *n*
⟨*proof*⟩

lemma *zmagnitude-zminus-int-of* [*simp*]: *zmagnitude*($− $# *n*) = *natify*(*n*)
⟨*proof*⟩

lemma *zmagnitude-type* [*iff*,*TC*]: *zmagnitude*(*z*)∈*nat*
⟨*proof*⟩

lemma *not-zneg-int-of*:
[| z: *int*; ~ *znegative*(z) |] ==> $\exists\, n{\in}nat.\ z = \$\#\ n$
⟨*proof*⟩

lemma *not-zneg-mag* [*simp*]:
[| z: *int*; ~ *znegative*(z) |] ==> $\$\#\ (zmagnitude(z)) = z$
⟨*proof*⟩

lemma *zneg-int-of*:
[| *znegative*(z); z: *int* |] ==> $\exists\, n{\in}nat.\ z = \$-\ (\$\#\ succ(n))$
⟨*proof*⟩

lemma *zneg-mag* [*simp*]:
[| *znegative*(z); z: *int* |] ==> $\$\#\ (zmagnitude(z)) = \$-\ z$
⟨*proof*⟩

lemma *int-cases*: z : *int* ==> $\exists\, n{\in}nat.\ z = \$\#\ n \mid z = \$-\ (\$\#\ succ(n))$
⟨*proof*⟩

lemma *not-zneg-raw-nat-of*:
[| ~ *znegative*(z); z: *int* |] ==> $\$\#\ (raw\text{-}nat\text{-}of(z)) = z$
⟨*proof*⟩

lemma *not-zneg-nat-of-intify*:
~ *znegative*(*intify*(z)) ==> $\$\#\ (nat\text{-}of(z)) = intify(z)$
⟨*proof*⟩

lemma *not-zneg-nat-of*: [| ~ *znegative*(z); z: *int* |] ==> $\$\#\ (nat\text{-}of(z)) = z$
⟨*proof*⟩

lemma *zneg-nat-of* [*simp*]: *znegative*(*intify*(z)) ==> *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*:

[| $x1 \in nat$; $y1 \in nat$; $x2 \in nat$; $y2 \in nat$ |]
==> *raw-zadd* (*intrel*``{<*x1*,*y1*>}, *intrel*``{<*x2*,*y2*>}) =
intrel `` {<*x1*#+*x2*, *y1*#+*y2*>}
⟨*proof*⟩

lemma *zadd*:
[| $x1 \in nat$; $y1 \in nat$; $x2 \in nat$; $y2 \in nat$ |]
==> (*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*:
[| $z1: int$; $z2: int$; $z3: int$ |]
==> *raw-zadd* (*raw-zadd*(*z1*,*z2*),*z3*) = *raw-zadd*(*z1*,*raw-zadd*(*z2*,*z3*))
⟨*proof*⟩

lemma *zadd-assoc*: $(z1 $+ z2) $+ z3 = z1 $+ (z2 $+ z3)$
⟨*proof*⟩

lemma *zadd-left-commute*: $z1$+(z2$+z3) = z2$+(z1$+z3)$
⟨*proof*⟩

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

lemma *int-of-add*: $# (*m* #+ *n*) = ($#*m*) $+ ($#*n*)
⟨*proof*⟩

lemma *int-succ-int-1*: $# *succ*(*m*) = $# *1* $+ ($# *m*)
⟨*proof*⟩

lemma *int-of-diff*:
[| *m*∈*nat*; *n le m* |] ==> $# (*m* #− *n*) = ($#*m*) $− ($#*n*)
⟨*proof*⟩

lemma *raw-zadd-zminus-inverse*: *z* : *int* ==> *raw-zadd* (*z*, $− *z*) = $#*0*
⟨*proof*⟩

lemma *zadd-zminus-inverse* [*simp*]: *z* $+ ($− *z*) = $#*0*
⟨*proof*⟩

lemma *zadd-zminus-inverse2* [*simp*]: ($− *z*) $+ *z* = $#*0*
⟨*proof*⟩

lemma *zadd-int0-right-intify* [*simp*]: *z* $+ $#*0* = *intify*(*z*)
⟨*proof*⟩

lemma *zadd-int0-right*: *z*:*int* ==> *z* $+ $#*0* = *z*
⟨*proof*⟩

30.8 *op* $×: Integer Multiplication

Congruence property for multiplication

lemma *zmult-congruent2*:
(%*p1 p2*. *split*(%*x1 y1*. *split*(%*x2 y2*.
intrel``{<*x1*#∗*x2* #+ *y1*#∗*y2*, *x1*#∗*y2* #+ *y1*#∗*x2*>}, *p2*), *p1*))
respects2 intrel
⟨*proof*⟩

lemma *raw-zmult-type*: [| *z*: *int*; *w*: *int* |] ==> *raw-zmult*(*z*,*w*) : *int*
⟨*proof*⟩

lemma *zmult-type* [*iff*,*TC*]: *z* $∗ *w* : *int*
⟨*proof*⟩

lemma *raw-zmult*:
[| *x1*∈*nat*; *y1*∈*nat*; *x2*∈*nat*; *y2*∈*nat* |]
==> *raw-zmult*(*intrel*``{<*x1*,*y1*>}, *intrel*``{<*x2*,*y2*>}) =
intrel `` {<*x1*#∗*x2* #+ *y1*#∗*y2*, *x1*#∗*y2* #+ *y1*#∗*x2*>}
⟨*proof*⟩

lemma *zmult*:
[| *x1*∈*nat*; *y1*∈*nat*; *x2*∈*nat*; *y2*∈*nat* |]

$==>$ (*intrel*``{<*x1*,*y1*>}) $* (*intrel*``{<*x2*,*y2*>}) =
intrel `` {<*x1*#∗*x2* #+ *y1*#∗*y2*, *x1*#∗*y2* #+ *y1*#∗*x2*>}
⟨*proof*⟩

lemma *raw-zmult-int0*: *z* : *int* ==> *raw-zmult* ($#*0*,*z*) = $#*0*
⟨*proof*⟩

lemma *zmult-int0* [*simp*]: $#*0* $∗ *z* = $#*0*
⟨*proof*⟩

lemma *raw-zmult-int1*: *z* : *int* ==> *raw-zmult* ($#*1*,*z*) = *z*
⟨*proof*⟩

lemma *zmult-int1-intify* [*simp*]: $#*1* $∗ *z* = *intify*(*z*)
⟨*proof*⟩

lemma *zmult-int1*: *z* : *int* ==> $#*1* $∗ *z* = *z*
⟨*proof*⟩

lemma *raw-zmult-commute*:
[| *z*: *int*; *w*: *int* |] ==> *raw-zmult*(*z*,*w*) = *raw-zmult*(*w*,*z*)
⟨*proof*⟩

lemma *zmult-commute*: *z* $∗ *w* = *w* $∗ *z*
⟨*proof*⟩

lemma *raw-zmult-zminus*:
[| *z*: *int*; *w*: *int* |] ==> *raw-zmult*($− *z*, *w*) = $− *raw-zmult*(*z*, *w*)
⟨*proof*⟩

lemma *zmult-zminus* [*simp*]: ($− *z*) $∗ *w* = $− (*z* $∗ *w*)
⟨*proof*⟩

lemma *zmult-zminus-right* [*simp*]: *w* $∗ ($− *z*) = $− (*w* $∗ *z*)
⟨*proof*⟩

lemma *raw-zmult-assoc*:
[| *z1*: *int*; *z2*: *int*; *z3*: *int* |]
==> *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* |]
==> *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 *zdiff-zadd-eq*: (*x* $− *y*) $+ *z* = (*x* $+ *z*) $− *y*
⟨*proof*⟩

30.9 The "Less Than" Relation

lemma *zless-linear-lemma*:
[| *z*: *int*; *w*: *int* |] ==> *z*$<*w* | *z*=*w* | *w*$<*z*
⟨*proof*⟩

lemma *zless-linear*: *z*$<*w* | *intify*(*z*)=*intify*(*w*) | *w*$<*z*
⟨*proof*⟩

lemma *zless-not-refl* [*iff*]: ~ (*z*$<*z*)
⟨*proof*⟩

lemma *neq-iff-zless*: [| x: *int*; y: *int* |] ==> (x ~= y) <-> (x $< y | y $< x)
⟨*proof*⟩

lemma *zless-imp-intify-neq*: w $< z ==> *intify*(w) ~= *intify*(z)
⟨*proof*⟩

lemma *zless-imp-succ-zadd-lemma*:
[| w $< z; w: *int*; z: *int* |] ==> (∃ n∈*nat*. z = w $+ $#(*succ*($n$)))
⟨*proof*⟩

lemma *zless-imp-succ-zadd*:
w $< z ==> (∃ n∈*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 <-> (∃ n∈*nat*. w $+ $#(*succ*($n$)) = *intify*($z$))
⟨*proof*⟩

lemma *zless-int-of* [*simp*]: [| m∈*nat*; n∈*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 ==> ~ (w $< z)
⟨*proof*⟩

lemmas *zless-asym* = *zless-not-sym* [*THEN swap*, *standard*]

lemma *zless-imp-zle*: z $< w ==> z $<= w
⟨*proof*⟩

lemma *zle-linear*: z $<= w | w $<= z
⟨*proof*⟩

30.10 Less Than or Equals

lemma *zle-refl*: $z \ \$<= \ z$
⟨*proof*⟩

lemma *zle-eq-refl*: $x{=}y \Longrightarrow x \ \$<= \ y$
⟨*proof*⟩

lemma *zle-anti-sym-intify*: $[\!| \ x \ \$<= \ y; \ y \ \$<= \ x \ |\!] \Longrightarrow intify(x) = intify(y)$
⟨*proof*⟩

lemma *zle-anti-sym*: $[\!| \ x \ \$<= \ y; \ y \ \$<= \ x; \ x{:} \ int; \ y{:} \ int \ |\!] \Longrightarrow x{=}y$
⟨*proof*⟩

lemma *zle-trans-lemma*:
$[\!| \ x{:} \ int; \ y{:} \ int; \ z{:} \ int; \ x \ \$<= \ y; \ y \ \$<= \ z \ |\!] \Longrightarrow x \ \$<= \ z$
⟨*proof*⟩

lemma *zle-trans*: $[\!| \ x \ \$<= \ y; \ y \ \$<= \ z \ |\!] \Longrightarrow x \ \$<= \ z$
⟨*proof*⟩

lemma *zle-zless-trans*: $[\!| \ i \ \$<= \ j; \ j \ \$< \ k \ |\!] \Longrightarrow i \ \$< \ k$
⟨*proof*⟩

lemma *zless-zle-trans*: $[\!| \ i \ \$< \ j; \ j \ \$<= \ k \ |\!] \Longrightarrow i \ \$< \ k$
⟨*proof*⟩

lemma *not-zless-iff-zle*: $\sim (z \ \$< \ w) <-> (w \ \$<= \ z)$
⟨*proof*⟩

lemma *not-zle-iff-zless*: $\sim (z \ \$<= \ w) <-> (w \ \$< \ z)$
⟨*proof*⟩

30.11 More subtraction laws (for *zcompare-rls*)

lemma *zdiff-zdiff-eq*: $(x \ \$- \ y) \ \$- \ z = x \ \$- \ (y \ \$+ \ z)$
⟨*proof*⟩

lemma *zdiff-zdiff-eq2*: $x \ \$- \ (y \ \$- \ z) = (x \ \$+ \ z) \ \$- \ y$
⟨*proof*⟩

lemma *zdiff-zless-iff*: $(x\${-}y \ \$< \ z) <-> (x \ \$< \ z \ \$+ \ y)$
⟨*proof*⟩

lemma *zless-zdiff-iff*: $(x \ \$< \ z\${-}y) <-> (x \ \$+ \ y \ \$< \ z)$
⟨*proof*⟩

lemma *zdiff-eq-iff*: $[\!| \ x{:} \ int; \ z{:} \ int \ |\!] \Longrightarrow (x\${-}y = z) <-> (x = z \ \$+ \ y)$
⟨*proof*⟩

lemma *eq-zdiff-iff*: [| x: *int*; z: *int* |] ==> $(x = z\$-y) <-> (x \$+ y = z)$
⟨*proof*⟩

lemma *zdiff-zle-iff-lemma*:
[| x: *int*; z: *int* |] ==> $(x\$-y \$<= z) <-> (x \$<= z \$+ y)$
⟨*proof*⟩

lemma *zdiff-zle-iff*: $(x\$-y \$<= z) <-> (x \$<= z \$+ y)$
⟨*proof*⟩

lemma *zle-zdiff-iff-lemma*:
[| x: *int*; z: *int* |] ==>$(x \$<= z\$-y) <-> (x \$+ y \$<= z)$
⟨*proof*⟩

lemma *zle-zdiff-iff*: $(x \$<= z\$-y) <-> (x \$+ y \$<= z)$
⟨*proof*⟩

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* |] ==> $(z \$+ w' = z \$+ w) <-> (w' = w)$
⟨*proof*⟩

lemma *zadd-left-cancel-intify* [*simp*]:
$(z \$+ w' = z \$+ w) <-> intify(w') = intify(w)$
⟨*proof*⟩

lemma *zadd-right-cancel*:
[| w: *int*; w': *int* |] ==> $(w' \$+ z = w \$+ z) <-> (w' = w)$
⟨*proof*⟩

lemma *zadd-right-cancel-intify* [*simp*]:
$(w' \$+ z = w \$+ z) <-> intify(w') = intify(w)$
⟨*proof*⟩

lemma *zadd-right-cancel-zless* [*simp*]: $(w' \$+ z \$< w \$+ z) <-> (w' \$< w)$
⟨*proof*⟩

lemma *zadd-left-cancel-zless* [*simp*]: $(z \$+ w' \$< z \$+ w) <-> (w' \$< w)$
⟨*proof*⟩

lemma *zadd-right-cancel-zle* [*simp*]: $(w' \$+ z \$<= w \$+ z) <-> w' \$<= w$
⟨*proof*⟩

lemma *zadd-left-cancel-zle* [*simp*]: $(z \$+ w' \$<= z \$+ w) <-> w' \$<= w$
⟨*proof*⟩

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*: [| $w' \$<= w$; $z' \$<= z$ |] ==> $w' \$+ z' \$<= w \$+ z$
⟨*proof*⟩

lemma *zadd-zless-mono*: [| $w' \$< w$; $z' \$<= z$ |] ==> $w' \$+ z' \$< w \$+ z$
⟨*proof*⟩

30.13 Comparison laws

lemma *zminus-zless-zminus* [*simp*]: $(\$- x \$< \$- y) <-> (y \$< x)$
⟨*proof*⟩

lemma *zminus-zle-zminus* [*simp*]: $(\$- x \$<= \$- y) <-> (y \$<= x)$
⟨*proof*⟩

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*)

⟨*ML*⟩

syntax
-Int :: *xnum* => *i* (-)

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]=>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))

⟨*ML*⟩

declare *bin.intros* [*simp*,*TC*]

lemma *NCons-Pls-0*: *NCons*(*Pls*,0) = *Pls*
⟨*proof*⟩

lemma *NCons-Pls-1*: *NCons*(*Pls*,1) = *Pls BIT 1*
⟨*proof*⟩

lemma *NCons-Min-0*: *NCons*(*Min*,0) = *Min BIT 0*
⟨*proof*⟩

lemma *NCons-Min-1*: *NCons*(*Min*,1) = *Min*
⟨*proof*⟩

lemma *NCons-BIT*: *NCons*(*w BIT x*,*b*) = *w BIT x BIT b*
⟨*proof*⟩

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* ==> *integ-of*(*w*) : *int*
⟨*proof*⟩

lemma *NCons-type* [*TC*]: [| *w*: *bin*; *b*: *bool* |] ==> *NCons*(*w*,*b*) : *bin*
⟨*proof*⟩

lemma *bin-succ-type* [*TC*]: *w*: *bin* ==> *bin-succ*(*w*) : *bin*
⟨*proof*⟩

lemma *bin-pred-type* [*TC*]: *w*: *bin* ==> *bin-pred*(*w*) : *bin*
⟨*proof*⟩

lemma *bin-minus-type* [*TC*]: *w*: *bin* ==> *bin-minus*(*w*) : *bin*
⟨*proof*⟩

lemma *bin-add-type* [*rule-format*,*TC*]:
v: *bin* ==> *ALL w*: *bin*. *bin-add*(*v*,*w*) : *bin*
⟨*proof*⟩

lemma *bin-mult-type* [*TC*]: [| *v*: *bin*; *w*: *bin* |] ==> *bin-mult*(*v*,*w*) : *bin*
⟨*proof*⟩

31.0.3 The Carry and Borrow Functions, *bin-succ* and *bin-pred*

lemma *integ-of-NCons* [*simp*]:
[| *w*: *bin*; *b*: *bool* |] ==> *integ-of*(*NCons*(*w*,*b*)) = *integ-of*(*w BIT b*)
⟨*proof*⟩

lemma *integ-of-succ* [*simp*]:
w: *bin* ==> *integ-of*(*bin-succ*(*w*)) = $#*1* $+ *integ-of*(*w*)
⟨*proof*⟩

lemma *integ-of-pred* [*simp*]:
w: *bin* ==> *integ-of*(*bin-pred*(*w*)) = $− ($#*1*) $+ *integ-of*(*w*)
⟨*proof*⟩

31.0.4 *bin-minus*: Unary Negation of Binary Integers

lemma *integ-of-minus*: *w*: *bin* ==> *integ-of*(*bin-minus*(*w*)) = $− *integ-of*(*w*)
⟨*proof*⟩

31.0.5 *bin-add*: Binary Addition

lemma *bin-add-Pls* [*simp*]: *w*: *bin* ==> *bin-add*(*Pls*,*w*) = *w*
⟨*proof*⟩

lemma *bin-add-Pls-right*: *w*: *bin* ==> *bin-add*(*w*,*Pls*) = *w*
⟨*proof*⟩

lemma *bin-add-Min* [*simp*]: *w*: *bin* ==> *bin-add*(*Min*,*w*) = *bin-pred*(*w*)
⟨*proof*⟩

lemma *bin-add-Min-right*: *w*: *bin* ==> *bin-add*(*w*,*Min*) = *bin-pred*(*w*)
⟨*proof*⟩

lemma *bin-add-BIT-Pls* [*simp*]: *bin-add*(*v BIT x*,*Pls*) = *v BIT x*
⟨*proof*⟩

lemma *bin-add-BIT-Min* [*simp*]: *bin-add*(*v BIT x*,*Min*) = *bin-pred*(*v BIT x*)
⟨*proof*⟩

lemma *bin-add-BIT-BIT* [*simp*]:
[| *w*: *bin*; *y*: *bool* |]
==> *bin-add*(*v BIT x*, *w BIT y*) =
NCons(*bin-add*(*v*, *cond*(*x and y*, *bin-succ*(*w*), *w*)), *x xor y*)

⟨*proof*⟩

lemma *integ-of-add* [*rule-format*]:
v: *bin* ==>
ALL w: *bin*. *integ-of*(*bin-add*(*v*,*w*)) = *integ-of*(*v*) $+ *integ-of*(*w*)
⟨*proof*⟩

lemma *diff-integ-of-eq*:
[| *v*: *bin*; *w*: *bin* |]
==> *integ-of*(*v*) $− *integ-of*(*w*) = *integ-of*(*bin-add* (*v*, *bin-minus*(*w*)))
⟨*proof*⟩

31.0.6 *bin-mult*: Binary Multiplication

lemma *integ-of-mult*:
[| *v*: *bin*; *w*: *bin* |]
==> *integ-of*(*bin-mult*(*v*,*w*)) = *integ-of*(*v*) $∗ *integ-of*(*w*)
⟨*proof*⟩

31.1 Computations

lemma *bin-succ-1*: *bin-succ*(*w BIT 1*) = *bin-succ*(*w*) *BIT 0*
⟨*proof*⟩

lemma *bin-succ-0*: *bin-succ*(*w BIT 0*) = *NCons*(*w*,*1*)
⟨*proof*⟩

lemma *bin-pred-1*: *bin-pred*(*w BIT 1*) = *NCons*(*w*,*0*)
⟨*proof*⟩

lemma *bin-pred-0*: *bin-pred*(*w BIT 0*) = *bin-pred*(*w*) *BIT 1*
⟨*proof*⟩

lemma *bin-minus-1*: *bin-minus*(*w BIT 1*) = *bin-pred*(*NCons*(*bin-minus*(*w*), *0*))
⟨*proof*⟩

lemma *bin-minus-0*: *bin-minus*(*w BIT 0*) = *bin-minus*(*w*) *BIT 0*
⟨*proof*⟩

lemma *bin-add-BIT-11*: *w*: *bin* ==> *bin-add*(*v BIT 1*, *w BIT 1*) =
NCons(*bin-add*(*v*, *bin-succ*(*w*)), *0*)
⟨*proof*⟩

lemma *bin-add-BIT-10*: *w*: *bin* ==> *bin-add*(*v BIT 1*, *w BIT 0*) =
NCons(*bin-add*(*v*,*w*), *1*)

⟨*proof*⟩

lemma *bin-add-BIT-0*: [| *w*: *bin*; *y*: *bool* |]
    ==> *bin-add*(*v BIT 0*, *w BIT y*) = *NCons*(*bin-add*(*v*,*w*), *y*)
⟨*proof*⟩

lemma *bin-mult-1*: *bin-mult*(*v BIT 1*, *w*) = *bin-add*(*NCons*(*bin-mult*(*v*,*w*),*0*), *w*)
⟨*proof*⟩

lemma *bin-mult-0*: *bin-mult*(*v BIT 0*, *w*) = *NCons*(*bin-mult*(*v*,*w*),*0*)
⟨*proof*⟩

lemma *int-of-0*: $#*0* = #*0*
⟨*proof*⟩

lemma *int-of-succ*: $# *succ*(*n*) = #*1* $+ $#*n*
⟨*proof*⟩

lemma *zminus-0* [*simp*]: $− #*0* = #*0*
⟨*proof*⟩

lemma *zadd-0-intify* [*simp*]: #*0* $+ *z* = *intify*(*z*)
⟨*proof*⟩

lemma *zadd-0-right-intify* [*simp*]: *z* $+ #*0* = *intify*(*z*)
⟨*proof*⟩

lemma *zmult-1-intify* [*simp*]: #*1* $∗ *z* = *intify*(*z*)
⟨*proof*⟩

lemma *zmult-1-right-intify* [*simp*]: *z* $∗ #*1* = *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*: ~ *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* ==> ~ *iszero* (*integ-of* (*w BIT 1*))
⟨*proof*⟩

lemma *less-integ-of-eq-neg*:
[| *v*: *bin*; *w*: *bin* |]
==> *integ-of*(*v*) $< *integ-of*(*w*)
<-> *znegative* (*integ-of* (*bin-add* (*v*, *bin-minus*(*w*))))
⟨*proof*⟩

lemma *not-neg-integ-of-Pls*: ~ *znegative* (*integ-of*(*Pls*))
⟨*proof*⟩

lemma *neg-integ-of-Min*: *znegative* (*integ-of*(*Min*))
⟨*proof*⟩

lemma *neg-integ-of-BIT*:
[| *w*: *bin*; *x*: *bool* |]
==> *znegative* (*integ-of* (*w BIT x*)) <-> *znegative* (*integ-of*(*w*))
⟨*proof*⟩

lemma *le-integ-of-eq-not-less*:
(*integ-of*(*x*) $<= (*integ-of*(*w*))) <-> ~ (*integ-of*(*w*) $< (*integ-of*(*x*)))
⟨*proof*⟩

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* |]
==> *integ-of*(*v*) $+ (*integ-of*(*w*) $+ *z*) = (*integ-of*(*bin-add*(*v*,*w*)) $+ *z*)
⟨*proof*⟩

lemma *mult-integ-of-left* [*simp*]:
[| *v*: *bin*; *w*: *bin* |]
==> *integ-of*(*v*) $* (*integ-of*(*w*) $* *z*) = (*integ-of*(*bin-mult*(*v*,*w*)) $* *z*)
⟨*proof*⟩

lemma *add-integ-of-diff1* [*simp*]:
[| *v*: *bin*; *w*: *bin* |]
==> *integ-of*(*v*) $+ (*integ-of*(*w*) $− *c*) = *integ-of*(*bin-add*(*v*,*w*)) $− (*c*)
⟨*proof*⟩

lemma *add-integ-of-diff2* [*simp*]:
[| *v*: *bin*; *w*: *bin* |]
==> *integ-of*(*v*) $+ (*c* $− *integ-of*(*w*)) =
integ-of (*bin-add* (*v*, *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* $− #0 = *intify*(*x*)
⟨*proof*⟩

lemma *zdiff-self* [*simp*]: *x* $− *x* = #0
⟨*proof*⟩

lemma *znegative-iff-zless-0*: *k*: *int* ==> *znegative*(*k*) <−> *k* $< #0
⟨*proof*⟩

lemma *zero-zless-imp-znegative-zminus*: [|#0 $< *k*; *k*: *int*|] ==> *znegative*($−*k*)
⟨*proof*⟩

lemma *zero-zle-int-of* [*simp*]: *#0 $<= $# n*
⟨*proof*⟩

lemma *nat-of-0* [*simp*]: *nat-of*(*#0*) = *0*
⟨*proof*⟩

lemma *nat-le-int0-lemma*: [| *z $<= $#0*; *z*: *int* |] ==> *nat-of*(*z*) = *0*
⟨*proof*⟩

lemma *nat-le-int0*: *z $<= $#0 ==> nat-of*(*z*) = *0*
⟨*proof*⟩

lemma *int-of-eq-0-imp-natify-eq-0*: *$# n = #0 ==> natify*(*n*) = *0*
⟨*proof*⟩

lemma *nat-of-zminus-int-of*: *nat-of*(*$− $# n*) = *0*
⟨*proof*⟩

lemma *int-of-nat-of*: *#0 $<= z ==> $# nat-of*(*z*) = *intify*(*z*)
⟨*proof*⟩

declare *int-of-nat-of* [*simp*] *nat-of-zminus-int-of* [*simp*]

lemma *int-of-nat-of-if*: *$# nat-of*(*z*) = (*if #0 $<= z then intify*(*z*) *else #0*)
⟨*proof*⟩

lemma *zless-nat-iff-int-zless*: [| *m*: *nat*; *z*: *int* |] ==> (*m* < *nat-of*(*z*)) <−> (*$#m $< z*)
⟨*proof*⟩

lemma *zless-nat-conj-lemma*: *$#0 $< z ==> (nat-of*(*w*) < *nat-of*(*z*)) <−> (*w $< z*)
⟨*proof*⟩

lemma *zless-nat-conj*: (*nat-of*(*w*) < *nat-of*(*z*)) <−> (*$#0 $< z & w $< z*)
⟨*proof*⟩

lemma *integ-of-minus-reorient* [*simp*]:
(*integ-of*(*w*) = *$− x*) <−> (*$− x = integ-of*(*w*))
⟨*proof*⟩

lemma *integ-of-add-reorient* [*simp*]:
(*integ-of*(*w*) = *x $+ y*) <−> (*x $+ y = integ-of*(*w*))

⟨*proof*⟩

lemma *integ-of-diff-reorient* [*simp*]:
(*integ-of*(*w*) = *x* $− *y*) <−> (*x* $− *y* = *integ-of*(*w*))
⟨*proof*⟩

lemma *integ-of-mult-reorient* [*simp*]:
(*integ-of*(*w*) = *x* $∗ *y*) <−> (*x* $∗ *y* = *integ-of*(*w*))
⟨*proof*⟩

end

theory *IntArith* **imports** *Bin*
uses (*int-arith.ML*)
begin

lemmas [*simp*] =
zminus-equation [**where** *y* = *integ-of*(*w*), *standard*]
equation-zminus [**where** *x* = *integ-of*(*w*), *standard*]

lemmas [*iff*] =
zminus-zless [**where** *y* = *integ-of*(*w*), *standard*]
zless-zminus [**where** *x* = *integ-of*(*w*), *standard*]

lemmas [*iff*] =
zminus-zle [**where** *y* = *integ-of*(*w*), *standard*]
zle-zminus [**where** *x* = *integ-of*(*w*), *standard*]

lemmas [*simp*] =
Let-def [**where** *s* = *integ-of*(*w*), *standard*]

lemma *zless-iff-zdiff-zless-0*: (*x* $< *y*) <−> (*x*$−*y* $< #0)
⟨*proof*⟩

lemma *eq-iff-zdiff-eq-0*: [| *x*: *int*; *y*: *int* |] ==> (*x* = *y*) <−> (*x*$−*y* = #0)
⟨*proof*⟩

lemma *zle-iff-zdiff-zle-0*: (*x* $<= *y*) <−> (*x*$−*y* $<= #0)
⟨*proof*⟩

lemma *left-zadd-zmult-distrib*: $i\$*u\ \$+\ (j\$*u\ \$+\ k) = (i\$+j)\$*u\ \$+\ k$
⟨*proof*⟩

lemmas *rel-iff-rel-0-rls* =
zless-iff-zdiff-zless-0 [**where** $y = u\ \$+\ v$, *standard*]
eq-iff-zdiff-eq-0 [**where** $y = u\ \$+\ v$, *standard*]
zle-iff-zdiff-zle-0 [**where** $y = u\ \$+\ v$, *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*: $(i\$*u\ \$+\ m = j\$*u\ \$+\ n) <-> ((i\$-j)\$*u\ \$+\ m = intify(n))$
⟨*proof*⟩

lemma *eq-add-iff2*: $(i\$*u\ \$+\ m = j\$*u\ \$+\ n) <-> (intify(m) = (j\$-i)\$*u\ \$+\ n)$
⟨*proof*⟩

lemma *less-add-iff1*: $(i\$*u\ \$+\ m\ \$<\ j\$*u\ \$+\ n) <-> ((i\$-j)\$*u\ \$+\ m\ \$<\ n)$
⟨*proof*⟩

lemma *less-add-iff2*: $(i\$*u\ \$+\ m\ \$<\ j\$*u\ \$+\ n) <-> (m\ \$<\ (j\$-i)\$*u\ \$+\ n)$
⟨*proof*⟩

lemma *le-add-iff1*: $(i\$*u\ \$+\ m\ \$<=\ j\$*u\ \$+\ n) <-> ((i\$-j)\$*u\ \$+\ m\ \$<=\ n)$
⟨*proof*⟩

lemma *le-add-iff2*: $(i\$*u\ \$+\ m\ \$<=\ j\$*u\ \$+\ n) <-> (m\ \$<=\ (j\$-i)\$*u\ \$+\ n)$
⟨*proof*⟩

⟨*ML*⟩

end

32 IntDiv-ZF: The Division Operators Div and Mod

theory *IntDiv-ZF* **imports** *IntArith OrderArith* **begin**

definition
quorem :: [*i*,*i*] => *o* **where**
quorem == %<*a*,*b*> <*q*,*r*>.
a = *b*$∗*q* $+ *r* &
(#0$<*b* & #0$<=*r* & *r*$<*b* | ~(#0$<*b*) & *b*$<*r* & *r* $<= #0)

definition
adjust :: [*i*,*i*] => *i* **where**
adjust(*b*) == %<*q*,*r*>. *if* #0 $<= *r*$−*b* *then* <#2$∗*q* $+ #1,*r*$−*b*>
else <#2$∗*q*,*r*>

definition
posDivAlg :: *i* => *i* **where**

posDivAlg(*ab*) ==
wfrec(*measure*(*int*∗*int*, %<*a*,*b*>. *nat-of* (*a* $− *b* $+ #1)),
ab,
%<*a*,*b*> *f*. *if* (*a*$<*b* | *b*$<=#0) *then* <#0,*a*>
else adjust(*b*, *f* ‘ <*a*,#2$∗*b*>))

definition
negDivAlg :: *i* => *i* **where**

negDivAlg(*ab*) ==
wfrec(*measure*(*int*∗*int*, %<*a*,*b*>. *nat-of* ($− *a* $− *b*)),
ab,
%<*a*,*b*> *f*. *if* (#0 $<= *a*$+*b* | *b*$<=#0) *then* <#−1,*a*$+*b*>
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* |] ==> *#0 $< x $+ y*
⟨*proof*⟩

lemma *zpos-add-zpos-imp-zpos*: [| *#0 $<= x; #0 $<= y* |] ==> *#0 $<= x $+ y*
⟨*proof*⟩

lemma *zneg-add-zneg-imp-zneg*: [| *x $< #0; y $< #0* |] ==> *x $+ y $< #0*
⟨*proof*⟩

lemma *zneg-or-0-add-zneg-or-0-imp-zneg-or-0*:
[| *x $<= #0; y $<= #0* |] ==> *x $+ y $<= #0*
⟨*proof*⟩

lemma *zero-lt-zmagnitude*: [| *#0 $< k; k ∈ int* |] ==> *0 < zmagnitude(k)*
⟨*proof*⟩

lemma *zless-add-succ-iff*:
(*w $< z $+ $# succ(m)*) <−> (*w $< z $+ $#m | intify(w) = z $+ $#m*)
⟨*proof*⟩

lemma *zadd-succ-lemma*:
z ∈ int ==> (*w $+ $# succ(m) $<= z*) <−> (*w $+ $#m $< z*)
⟨*proof*⟩

lemma *zadd-succ-zle-iff*: (*w $+ $# succ(m) $<= z*) <−> (*w $+ $#m $< z*)
⟨*proof*⟩

lemma *zless-add1-iff-zle*: $(w \$< z \$+ \#1) <-> (w\$<=z)$
⟨*proof*⟩

lemma *add1-zle-iff*: $(w \$+ \#1 \$<= z) <-> (w \$< z)$
⟨*proof*⟩

lemma *add1-left-zle-iff*: $(\#1 \$+ w \$<= z) <-> (w \$< z)$
⟨*proof*⟩

lemma *zmult-mono-lemma*: $k \in nat ==> i \$<= j ==> i \$* \$\#k \$<= j \$* \$\#k$
⟨*proof*⟩

lemma *zmult-zle-mono1*: $[| i \$<= j;\ \#0 \$<= k |] ==> i\$*k \$<= j\$*k$
⟨*proof*⟩

lemma *zmult-zle-mono1-neg*: $[| i \$<= j;\ k \$<= \#0 |] ==> j\$*k \$<= i\$*k$
⟨*proof*⟩

lemma *zmult-zle-mono2*: $[| i \$<= j;\ \#0 \$<= k |] ==> k\$*i \$<= k\$*j$
⟨*proof*⟩

lemma *zmult-zle-mono2-neg*: $[| i \$<= j;\ k \$<= \#0 |] ==> k\$*j \$<= k\$*i$
⟨*proof*⟩

lemma *zmult-zle-mono*:
$[| i \$<= j;\ k \$<= l;\ \#0 \$<= j;\ \#0 \$<= k |] ==> i\$*k \$<= j\$*l$
⟨*proof*⟩

lemma *zmult-zless-mono2-lemma* [*rule-format*]:
$[| i\$<j;\ k \in nat |] ==> 0<k --> \$\#k \$* i \$< \$\#k \$* j$
⟨*proof*⟩

lemma *zmult-zless-mono2*: $[| i\$<j;\ \#0 \$< k |] ==> k\$*i \$< k\$*j$
⟨*proof*⟩

lemma *zmult-zless-mono1*: $[| i\$<j;\ \#0 \$< k |] ==> i\$*k \$< j\$*k$
⟨*proof*⟩

lemma *zmult-zless-mono*:
$[| i \$< j;\ k \$< l;\ \#0 \$< j;\ \#0 \$< k |] ==> i\$*k \$< j\$*l$
⟨*proof*⟩

lemma *zmult-zless-mono1-neg*: [| $i \ \$< j; \ \ k \ \$< \#0$ |] ==> $j\$*k \ \$< i\$*k$
⟨*proof*⟩

lemma *zmult-zless-mono2-neg*: [| $i \ \$< j; \ \ k \ \$< \#0$ |] ==> $k\$*j \ \$< k\$*i$
⟨*proof*⟩

lemma *zmult-eq-lemma*:
$[| \ m \in int; \ n \in int \ |] ==> (m = \#0 \mid n = \#0) <-> (m\$*n = \#0)$
⟨*proof*⟩

lemma *zmult-eq-0-iff* [*iff*]: $(m\$*n = \#0) <-> (intify(m) = \#0 \mid intify(n) = \#0)$
⟨*proof*⟩

lemma *zmult-zless-lemma*:
$[| \ k \in int; \ m \in int; \ n \in int \ |]$
$==> (m\$*k \ \$< n\$*k) <-> ((\#0 \ \$< k \ \& \ m\$<n) \mid (k \ \$< \#0 \ \& \ n\$<m))$
⟨*proof*⟩

lemma *zmult-zless-cancel2*:
$(m\$*k \ \$< n\$*k) <-> ((\#0 \ \$< k \ \& \ m\$<n) \mid (k \ \$< \#0 \ \& \ n\$<m))$
⟨*proof*⟩

lemma *zmult-zless-cancel1*:
$(k\$*m \ \$< k\$*n) <-> ((\#0 \ \$< k \ \& \ m\$<n) \mid (k \ \$< \#0 \ \& \ n\$<m))$
⟨*proof*⟩

lemma *zmult-zle-cancel2*:
$(m\$*k \ \$<= n\$*k) <-> ((\#0 \ \$< k \ --> \ m\$<=n) \ \& \ (k \ \$< \#0 \ --> n\$<=m))$
⟨*proof*⟩

lemma *zmult-zle-cancel1*:
$(k\$*m \ \$<= k\$*n) <-> ((\#0 \ \$< k \ --> \ m\$<=n) \ \& \ (k \ \$< \#0 \ --> n\$<=m))$
⟨*proof*⟩

lemma *int-eq-iff-zle*: $[| \ m \in int; \ n \in int \ |] ==> m=n <-> (m \ \$<= n \ \& \ n \ \$<= m)$
⟨*proof*⟩

lemma *zmult-cancel2-lemma*:

[| $k \in int$; $m \in int$; $n \in int$ |] ==> $(m\$*k = n\$*k) <-> (k=\#0 \mid m=n)$
⟨*proof*⟩

lemma *zmult-cancel2* [*simp*]:
$(m\$*k = n\$*k) <-> (intify(k) = \#0 \mid intify(m) = intify(n))$
⟨*proof*⟩

lemma *zmult-cancel1* [*simp*]:
$(k\$*m = k\$*n) <-> (intify(k) = \#0 \mid intify(m) = intify(n))$
⟨*proof*⟩

32.1 Uniqueness and monotonicity of quotients and remainders

lemma *unique-quotient-lemma*:
[| $b\$*q' \$+ r' \$<= b\$*q \$+ r$; $\#0 \$<= r'$; $\#0 \$< b$; $r \$< b$ |]
==> $q' \$<= q$
⟨*proof*⟩

lemma *unique-quotient-lemma-neg*:
[| $b\$*q' \$+ r' \$<= b\$*q \$+ r$; $r \$<= \#0$; $b \$< \#0$; $b \$< r'$ |]
==> $q \$<= q'$
⟨*proof*⟩

lemma *unique-quotient*:
[| *quorem* (<a,b>, <q,r>); *quorem* (<a,b>, <q′,r′>); $b \in int$; $b \sim= \#0$;
$q \in int$; $q' \in int$ |] ==> $q = q'$
⟨*proof*⟩

lemma *unique-remainder*:
[| *quorem* (<a,b>, <q,r>); *quorem* (<a,b>, <q′,r′>); $b \in int$; $b \sim= \#0$;
$q \in int$; $q' \in int$;
$r \in int$; $r' \in int$ |] ==> $r = r'$
⟨*proof*⟩

32.2 Correctness of posDivAlg, the Division Algorithm for $a \geq 0$ and $b > 0$

lemma *adjust-eq* [*simp*]:
adjust(b, <q,r>) = (*let diff* = r$−b *in*
if #0 $<= *diff then* <#2$*q $+ #1,*diff*>
else <#2$*q,r>)
⟨*proof*⟩

lemma *posDivAlg-termination*:
[| $\#0 \$< b$; $\sim a \$< b$ |]
==> *nat-of*$(a \$- \#2 \$\times b \$+ \#1) < nat\text{-}of(a \$- b \$+ \#1)$

⟨*proof*⟩

lemmas *posDivAlg-unfold* = *def-wfrec* [*OF posDivAlg-def wf-measure*]

lemma *posDivAlg-eqn*:
    [| #0 $< b; a ∈ int; b ∈ int |] ==>
     posDivAlg(<a,b>) =
      (*if* a$<b *then* <#0,a> *else adjust*(b, *posDivAlg* (<a, #2$*b>)))
⟨*proof*⟩

lemma *posDivAlg-induct-lemma* [*rule-format*]:
  assumes *prem*:
       !!a b. [| a ∈ int; b ∈ int;
                ~ (a $< b | b $<= #0) --> P(<a, #2 $* b>) |] ==> P(<a,b>)
  shows <u,v> ∈ int*int --> P(<u,v>)
⟨*proof*⟩

lemma *posDivAlg-induct* [*consumes 2*]:
  assumes *u-int*: u ∈ int
      and *v-int*: v ∈ int
      and *ih*: !!a b. [| a ∈ int; b ∈ int;
                    ~ (a $< b | b $<= #0) --> P(a, #2 $* b) |] ==> P(a,b)
  shows P(u,v)
⟨*proof*⟩

lemma *intify-eq-0-iff-zle*: *intify*(m) = #0 <-> (m $<= #0 & #0 $<= m)
⟨*proof*⟩

32.3 Some convenient biconditionals for products of signs

lemma *zmult-pos*: [| #0 $< i; #0 $< j |] ==> #0 $< i $* j
⟨*proof*⟩

lemma *zmult-neg*: [| i $< #0; j $< #0 |] ==> #0 $< i $* j
⟨*proof*⟩

lemma *zmult-pos-neg*: [| #0 $< i; j $< #0 |] ==> i $* j $< #0
⟨*proof*⟩

lemma *int-0-less-lemma*:
    [| x ∈ int; y ∈ int |]
    ==> (#0 $< x $* y) <-> (#0 $< x & #0 $< y | x $< #0 & y $< #0)
⟨*proof*⟩

lemma *int-0-less-mult-iff*:

(#0 $< x $* y) <-> (#0 $< x & #0 $< y | x $< #0 & y $< #0)
⟨proof⟩

lemma *int-0-le-lemma*:
[| x ∈ int; y ∈ int |]
==> (#0 $<= x $* y) <-> (#0 $<= x & #0 $<= y | x $<= #0 & y $<= #0)
⟨proof⟩

lemma *int-0-le-mult-iff*:
(#0 $<= x $* y) <-> ((#0 $<= x & #0 $<= y) | (x $<= #0 & y $<= #0))
⟨proof⟩

lemma *zmult-less-0-iff*:
(x $* y $< #0) <-> (#0 $< x & y $< #0 | x $< #0 & #0 $< y)
⟨proof⟩

lemma *zmult-le-0-iff*:
(x $* y $<= #0) <-> (#0 $<= x & y $<= #0 | x $<= #0 & #0 $<= y)
⟨proof⟩

lemma *posDivAlg-type* [*rule-format*]:
[| a ∈ int; b ∈ int |] ==> posDivAlg(<a,b>) ∈ int * int
⟨proof⟩

lemma *posDivAlg-correct* [*rule-format*]:
[| a ∈ int; b ∈ int |]
==> #0 $<= a --> #0 $< b --> quorem (<a,b>, posDivAlg(<a,b>))
⟨proof⟩

32.4 Correctness of negDivAlg, the division algorithm for a¡0 and b¿0

lemma *negDivAlg-termination*:
[| #0 $< b; a $+ b $< #0 |]
==> nat-of($- a $- #2 $* b) < nat-of($- a $- b)
⟨proof⟩

lemmas *negDivAlg-unfold* = *def-wfrec* [*OF negDivAlg-def wf-measure*]

lemma *negDivAlg-eqn*:
[| #0 $< b; a : int; b : int |] ==>
negDivAlg(<a,b>) =
(if #0 $<= a$+b then <#-1,a$+b>
else adjust(b, negDivAlg (<a, #2$*b>)))

⟨*proof*⟩

lemma *negDivAlg-induct-lemma* [*rule-format*]:
assumes *prem*:
!!*a b*. [| *a* ∈ *int*; *b* ∈ *int*;
~ (#0 $<= *a* $+ *b* | *b* $<= #0) −−> *P*(<*a*, #2 $∗ *b*>) |]
==> *P*(<*a*,*b*>)
shows <*u*,*v*> ∈ *int*∗*int* −−> *P*(<*u*,*v*>)
⟨*proof*⟩

lemma *negDivAlg-induct* [*consumes 2*]:
assumes *u-int*: *u* ∈ *int*
and *v-int*: *v* ∈ *int*
and *ih*: !!*a b*. [| *a* ∈ *int*; *b* ∈ *int*;
~ (#0 $<= *a* $+ *b* | *b* $<= #0) −−> *P*(*a*, #2 $∗ *b*) |]
==> *P*(*a*,*b*)
shows *P*(*u*,*v*)
⟨*proof*⟩

lemma *negDivAlg-type*:
[| *a* ∈ *int*; *b* ∈ *int* |] ==> *negDivAlg*(<*a*,*b*>) ∈ *int* ∗ *int*
⟨*proof*⟩

lemma *negDivAlg-correct* [*rule-format*]:
[| *a* ∈ *int*; *b* ∈ *int* |]
==> *a* $< #0 −−> #0 $< *b* −−> *quorem* (<*a*,*b*>, *negDivAlg*(<*a*,*b*>))
⟨*proof*⟩

32.5 Existence shown by proving the division algorithm to be correct

lemma *quorem-0*: [|*b* ≠ #0; *b* ∈ *int*|] ==> *quorem* (<#0,*b*>, <#0,#0>)
⟨*proof*⟩

lemma *posDivAlg-zero-divisor*: *posDivAlg*(<*a*,#0>) = <#0,*a*>
⟨*proof*⟩

lemma *posDivAlg-0* [*simp*]: *posDivAlg* (<#0,*b*>) = <#0,#0>
⟨*proof*⟩

lemma *linear-arith-lemma*: ~ (#0 $<= #−1 $+ *b*) ==> (*b* $<= #0)
⟨*proof*⟩

lemma *negDivAlg-minus1* [*simp*]: *negDivAlg* (<#−1,*b*>) = <#−1, *b*$−#1>
⟨*proof*⟩

lemma *negateSnd-eq* [*simp*]: *negateSnd* (<*q*,*r*>) = <*q*, $−*r*>
⟨*proof*⟩

lemma *negateSnd-type*: *qr* ∈ *int* ∗ *int* ==> *negateSnd* (*qr*) ∈ *int* ∗ *int*
⟨*proof*⟩

lemma *quorem-neg*:
[|*quorem* (<$−*a*,$−*b*>, *qr*); *a* ∈ *int*; *b* ∈ *int*; *qr* ∈ *int* ∗ *int*|]
==> *quorem* (<*a*,*b*>, *negateSnd*(*qr*))
⟨*proof*⟩

lemma *divAlg-correct*:
[|*b* ≠ #0; *a* ∈ *int*; *b* ∈ *int*|] ==> *quorem* (<*a*,*b*>, *divAlg*(<*a*,*b*>))
⟨*proof*⟩

lemma *divAlg-type*: [|*a* ∈ *int*; *b* ∈ *int*|] ==> *divAlg*(<*a*,*b*>) ∈ *int* ∗ *int*
⟨*proof*⟩

lemma *zdiv-intify1* [*simp*]: *intify*(*x*) *zdiv y* = *x zdiv y*
⟨*proof*⟩

lemma *zdiv-intify2* [*simp*]: *x zdiv intify*(*y*) = *x zdiv y*
⟨*proof*⟩

lemma *zdiv-type* [*iff*, *TC*]: *z zdiv w* ∈ *int*
⟨*proof*⟩

lemma *zmod-intify1* [*simp*]: *intify*(*x*) *zmod y* = *x zmod y*
⟨*proof*⟩

lemma *zmod-intify2* [*simp*]: *x zmod intify*(*y*) = *x zmod y*
⟨*proof*⟩

lemma *zmod-type* [*iff*, *TC*]: *z zmod w* ∈ *int*
⟨*proof*⟩

lemma *DIVISION-BY-ZERO-ZDIV*: *a zdiv* #0 = #0
⟨*proof*⟩

lemma *DIVISION-BY-ZERO-ZMOD*: *a zmod* #0 = *intify*(*a*)

⟨*proof*⟩

lemma *raw-zmod-zdiv-equality*:
[| *a* ∈ *int*; *b* ∈ *int* |] ==> *a* = *b* $∗ (*a zdiv b*) $+ (*a zmod b*)
⟨*proof*⟩

lemma *zmod-zdiv-equality*: *intify*(*a*) = *b* $∗ (*a zdiv b*) $+ (*a zmod b*)
⟨*proof*⟩

lemma *pos-mod*: #0 $< *b* ==> #0 $<= *a zmod b* & *a 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 ==> *a zmod b* $<= #0 & *b* $< *a 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* ≠ #0; *a* ∈ *int*; *b* ∈ *int* |]
==> *quorem* (<*a*,*b*>, <*a zdiv b*, *a zmod b*>)
⟨*proof*⟩

lemma *quorem-div*:
[| *quorem*(<*a*,*b*>,<*q*,*r*>); *b* ≠ #0; *a* ∈ *int*; *b* ∈ *int*; *q* ∈ *int* |]
==> *a zdiv b* = *q*
⟨*proof*⟩

lemma *quorem-mod*:
[| *quorem*(<*a*,*b*>,<*q*,*r*>); *b* ≠ #0; *a* ∈ *int*; *b* ∈ *int*; *q* ∈ *int*; *r* ∈ *int* |]
==> *a zmod b* = *r*
⟨*proof*⟩

lemma *zdiv-pos-pos-trivial-raw*:
[| *a* ∈ *int*; *b* ∈ *int*; #0 $<= *a*; *a* $< *b* |] ==> *a zdiv b* = #0
⟨*proof*⟩

lemma *zdiv-pos-pos-trivial*: [| #0 $<= *a*; *a* $< *b* |] ==> *a zdiv b* = #0

⟨*proof*⟩

lemma *zdiv-neg-neg-trivial-raw*:
[| $a \in int$; $b \in int$; $a \$<= \#0$; $b \$< a$ |] ==> $a\ zdiv\ b = \#0$
⟨*proof*⟩

lemma *zdiv-neg-neg-trivial*: [| $a \$<= \#0$; $b \$< a$ |] ==> $a\ zdiv\ b = \#0$
⟨*proof*⟩

lemma *zadd-le-0-lemma*: [| $a\$+b \$<= \#0$; $\#0 \$< a$; $\#0 \$< b$ |] ==> *False*
⟨*proof*⟩

lemma *zdiv-pos-neg-trivial-raw*:
[| $a \in int$; $b \in int$; $\#0 \$< a$; $a\$+b \$<= \#0$ |] ==> $a\ zdiv\ b = \#-1$
⟨*proof*⟩

lemma *zdiv-pos-neg-trivial*: [| $\#0 \$< a$; $a\$+b \$<= \#0$ |] ==> $a\ zdiv\ b = \#-1$
⟨*proof*⟩

lemma *zmod-pos-pos-trivial-raw*:
[| $a \in int$; $b \in int$; $\#0 \$<= a$; $a \$< b$ |] ==> $a\ zmod\ b = a$
⟨*proof*⟩

lemma *zmod-pos-pos-trivial*: [| $\#0 \$<= a$; $a \$< b$ |] ==> $a\ zmod\ b = intify(a)$
⟨*proof*⟩

lemma *zmod-neg-neg-trivial-raw*:
[| $a \in int$; $b \in int$; $a \$<= \#0$; $b \$< a$ |] ==> $a\ zmod\ b = a$
⟨*proof*⟩

lemma *zmod-neg-neg-trivial*: [| $a \$<= \#0$; $b \$< a$ |] ==> $a\ zmod\ b = intify(a)$
⟨*proof*⟩

lemma *zmod-pos-neg-trivial-raw*:
[| $a \in int$; $b \in int$; $\#0 \$< a$; $a\$+b \$<= \#0$ |] ==> $a\ zmod\ b = a\$+b$
⟨*proof*⟩

lemma *zmod-pos-neg-trivial*: [| $\#0 \$< a$; $a\$+b \$<= \#0$ |] ==> $a\ zmod\ b = a\$+b$
⟨*proof*⟩

lemma *zdiv-zminus-zminus-raw*:
[|$a \in int$; $b \in int$|] ==> ($-a) *zdiv* ($-b) = a *zdiv* b
⟨*proof*⟩

lemma *zdiv-zminus-zminus* [*simp*]: ($-a) *zdiv* ($-b) = a *zdiv* b
⟨*proof*⟩

lemma *zmod-zminus-zminus-raw*:
[|$a \in int$; $b \in int$|] ==> ($-a) *zmod* ($-b) = $- ($a$ *zmod* b)
⟨*proof*⟩

lemma *zmod-zminus-zminus* [*simp*]: ($-a) *zmod* ($-b) = $- ($a$ *zmod* b)
⟨*proof*⟩

32.6 division of a number by itself

lemma *self-quotient-aux1*: [| #0 $< a; $a = r$ $+ a*q; r $< a |] ==> #1 $<= q
⟨*proof*⟩

lemma *self-quotient-aux2*: [| #0 $< a; $a = r$ $+ a*q; #0 $<= r |] ==> q $<= #1
⟨*proof*⟩

lemma *self-quotient*:
[| *quorem*(<a,a>,<q,r>); $a \in int$; $q \in int$; $a \neq$ #0|] ==> q = #1
⟨*proof*⟩

lemma *self-remainder*:
[|*quorem*(<a,a>,<q,r>); $a \in int$; $q \in int$; $r \in int$; $a \neq$ #0|] ==> r = #0
⟨*proof*⟩

lemma *zdiv-self-raw*: [|$a \neq$ #0; $a \in int$|] ==> a *zdiv* a = #1
⟨*proof*⟩

lemma *zdiv-self* [*simp*]: *intify*(a) $\neq$ #0 ==> a *zdiv* a = #1
⟨*proof*⟩

lemma *zmod-self-raw*: $a \in int$ ==> a *zmod* a = #0
⟨*proof*⟩

lemma *zmod-self* [*simp*]: a *zmod* a = #0
⟨*proof*⟩

32.7 Computation of division and remainder

lemma *zdiv-zero* [*simp*]: #0 *zdiv* b = #0
⟨*proof*⟩

lemma *zdiv-eq-minus1*: *#0 $< b ==> #−1 zdiv b = #−1*
⟨*proof*⟩

lemma *zmod-zero* [*simp*]: *#0 zmod b = #0*
⟨*proof*⟩

lemma *zdiv-minus1*: *#0 $< b ==> #−1 zdiv b = #−1*
⟨*proof*⟩

lemma *zmod-minus1*: *#0 $< b ==> #−1 zmod b = b $− #1*
⟨*proof*⟩

lemma *zdiv-pos-pos*: [| *#0 $< a*; *#0 $<= b* |]
==> a zdiv b = fst (*posDivAlg*(<*intify*(*a*), *intify*(*b*)>))
⟨*proof*⟩

lemma *zmod-pos-pos*:
[| *#0 $< a*; *#0 $<= b* |]
==> a zmod b = snd (*posDivAlg*(<*intify*(*a*), *intify*(*b*)>))
⟨*proof*⟩

lemma *zdiv-neg-pos*:
[| *a $< #0*; *#0 $< b* |]
==> a zdiv b = fst (*negDivAlg*(<*intify*(*a*), *intify*(*b*)>))
⟨*proof*⟩

lemma *zmod-neg-pos*:
[| *a $< #0*; *#0 $< b* |]
==> a zmod b = snd (*negDivAlg*(<*intify*(*a*), *intify*(*b*)>))
⟨*proof*⟩

lemma *zdiv-pos-neg*:
[| *#0 $< a*; *b $< #0* |]
==> a zdiv b = fst (*negateSnd*(*negDivAlg* (<$−*a*, $−*b*>)))
⟨*proof*⟩

lemma *zmod-pos-neg*:
[| *#0 $< a*; *b $< #0* |]
==> a zmod b = snd (*negateSnd*(*negDivAlg* (<$−*a*, $−*b*>)))
⟨*proof*⟩

lemma *zdiv-neg-neg*:
[| *a* $< #0; *b* $<= #0 |]
==> *a zdiv b* = *fst* (*negateSnd*(*posDivAlg*(<$−*a*, $−*b*>)))
⟨*proof*⟩

lemma *zmod-neg-neg*:
[| *a* $< #0; *b* $<= #0 |]
==> *a zmod b* = *snd* (*negateSnd*(*posDivAlg*(<$−*a*, $−*b*>)))
⟨*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 zmod #1* = *#0*
⟨*proof*⟩

lemma *zdiv-1* [*simp*]: *a zdiv #1* = *intify*(*a*)
⟨*proof*⟩

lemma *zmod-minus1-right* [*simp*]: *a zmod #−1* = *#0*
⟨*proof*⟩

lemma *zdiv-minus1-right-raw*: *a* ∈ *int* ==> *a zdiv #−1* = $−*a*
⟨*proof*⟩

lemma *zdiv-minus1-right*: *a zdiv #−1* = $−*a*
⟨*proof*⟩
declare *zdiv-minus1-right* [*simp*]

32.8 Monotonicity in the first argument (divisor)

lemma *zdiv-mono1*: [| *a* $<= *a′*; *#0* $< *b* |] ==> *a zdiv b* $<= *a′ zdiv b*
⟨*proof*⟩

lemma *zdiv-mono1-neg*: [| *a* $<= *a′*; *b* $< *#0* |] ==> *a′ zdiv b* $<= *a zdiv b*
⟨*proof*⟩

32.9 Monotonicity in the second argument (dividend)

lemma *q-pos-lemma*:
[| #0 $<= b'$*q' $+ r'; r' $< b'; #0 $< b' |] ==> #0 $<= q'
⟨*proof*⟩

lemma *zdiv-mono2-lemma*:
[| b$*q $+ r = b'$*q' $+ r'; #0 $<= b'$*q' $+ r';
r' $< b'; #0 $<= r; #0 $< b'; b' $<= b |]
==> q $<= q'
⟨*proof*⟩

lemma *zdiv-mono2-raw*:
[| #0 $<= a; #0 $< b'; b' $<= b; a ∈ int |]
==> a zdiv b $<= a zdiv b'
⟨*proof*⟩

lemma *zdiv-mono2*:
[| #0 $<= a; #0 $< b'; b' $<= b |]
==> a zdiv b $<= a zdiv b'
⟨*proof*⟩

lemma *q-neg-lemma*:
[| b'$*q' $+ r' $< #0; #0 $<= r'; #0 $< b' |] ==> q' $< #0
⟨*proof*⟩

lemma *zdiv-mono2-neg-lemma*:
[| b$*q $+ r = b'$*q' $+ r'; b'$*q' $+ r' $< #0;
r $< b; #0 $<= r'; #0 $< b'; b' $<= b |]
==> q' $<= q
⟨*proof*⟩

lemma *zdiv-mono2-neg-raw*:
[| a $< #0; #0 $< b'; b' $<= b; a ∈ int |]
==> a zdiv b' $<= a zdiv b
⟨*proof*⟩

lemma *zdiv-mono2-neg*: [| a $< #0; #0 $< b'; b' $<= b |]
==> a zdiv b' $<= a zdiv b
⟨*proof*⟩

32.10 More algebraic laws for zdiv and zmod

lemma *zmult1-lemma*:
[| quorem(<b,c>, <q,r>); c ∈ int; c ≠ #0 |]
==> quorem (<a$*b, c>, <a$*q $+ (a$*r) zdiv c, (a$*r) zmod c>)
⟨*proof*⟩

lemma *zdiv-zmult1-eq-raw*:
[|$b \in int$; $c \in int$|]
==> $(a\$*b)\ zdiv\ c = a\$*(b\ zdiv\ c)\ \$+\ a\$*(b\ zmod\ c)\ zdiv\ c$
⟨*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*:
[|$b \in int$; $c \in int$|] ==> $(a\$*b)\ zmod\ c = a\$*(b\ zmod\ c)\ zmod\ c$
⟨*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*]: $intify(b) \neq \#0$ ==> $(a\$*b)\ zdiv\ b = intify(a)$
⟨*proof*⟩

lemma *zdiv-zmult-self2* [*simp*]: $intify(b) \neq \#0$ ==> $(b\$*a)\ zdiv\ b = intify(a)$
⟨*proof*⟩

lemma *zmod-zmult-self1* [*simp*]: $(a\$*b)\ zmod\ b = \#0$
⟨*proof*⟩

lemma *zmod-zmult-self2* [*simp*]: $(b\$*a)\ zmod\ b = \#0$
⟨*proof*⟩

lemma *zadd1-lemma*:
[| $quorem(<a,c>, <aq,ar>)$; $quorem(<b,c>, <bq,br>)$;
$c \in int$; $c \neq \#0$ |]
==> $quorem\ (<a\$+b,\ c>, <aq\ \$+\ bq\ \$+\ (ar\$+br)\ zdiv\ c,\ (ar\$+br)\ zmod\ c>)$
⟨*proof*⟩

lemma *zdiv-zadd1-eq-raw*:
[|$a \in int$; $b \in int$; $c \in int$|] ==>
$(a\$+b)\ zdiv\ c = a\ zdiv\ c\ \$+\ b\ zdiv\ c\ \$+\ ((a\ zmod\ c\ \$+\ b\ zmod\ c)\ zdiv\ c)$
⟨*proof*⟩

lemma *zdiv-zadd1-eq*:
(*a*$+*b*) *zdiv c* = *a zdiv c* $+ *b zdiv c* $+ ((*a zmod c* $+ *b zmod c*) *zdiv c*)
⟨*proof*⟩

lemma *zmod-zadd1-eq-raw*:
[|*a* ∈ *int*; *b* ∈ *int*; *c* ∈ *int*|]
==> (*a*$+*b*) *zmod c* = (*a zmod c* $+ *b zmod c*) *zmod c*
⟨*proof*⟩

lemma *zmod-zadd1-eq*: (*a*$+*b*) *zmod c* = (*a zmod c* $+ *b zmod c*) *zmod c*
⟨*proof*⟩

lemma *zmod-div-trivial-raw*:
[|*a* ∈ *int*; *b* ∈ *int*|] ==> (*a zmod b*) *zdiv b* = #0
⟨*proof*⟩

lemma *zmod-div-trivial* [*simp*]: (*a zmod b*) *zdiv b* = #0
⟨*proof*⟩

lemma *zmod-mod-trivial-raw*:
[|*a* ∈ *int*; *b* ∈ *int*|] ==> (*a zmod b*) *zmod b* = *a zmod b*
⟨*proof*⟩

lemma *zmod-mod-trivial* [*simp*]: (*a zmod b*) *zmod b* = *a zmod b*
⟨*proof*⟩

lemma *zmod-zadd-left-eq*: (*a*$+*b*) *zmod c* = ((*a zmod c*) $+ *b*) *zmod c*
⟨*proof*⟩

lemma *zmod-zadd-right-eq*: (*a*$+*b*) *zmod c* = (*a* $+ (*b zmod c*)) *zmod c*
⟨*proof*⟩

lemma *zdiv-zadd-self1* [*simp*]:
intify(*a*) ≠ #0 ==> (*a*$+*b*) *zdiv a* = *b zdiv a* $+ #1
⟨*proof*⟩

lemma *zdiv-zadd-self2* [*simp*]:
intify(*a*) ≠ #0 ==> (*b*$+*a*) *zdiv a* = *b zdiv a* $+ #1
⟨*proof*⟩

lemma *zmod-zadd-self1* [*simp*]: (*a*$+*b*) *zmod a* = *b zmod a*
⟨*proof*⟩

lemma *zmod-zadd-self2* [*simp*]: (*b*$+*a*) *zmod a* = *b zmod a*
⟨*proof*⟩

32.11 proving a zdiv (b*c) = (a zdiv b) zdiv c

lemma *zdiv-zmult2-aux1*:
[| #0 $< c; b $< r; r $<= #0 |] ==> b$$*c $< b$$*(q zmod c) $+ r
⟨*proof*⟩

lemma *zdiv-zmult2-aux2*:
[| #0 $< c; b $< r; r $<= #0 |] ==> b $* (q zmod c) $+ r $<= #0
⟨*proof*⟩

lemma *zdiv-zmult2-aux3*:
[| #0 $< c; #0 $<= r; r $< b |] ==> #0 $<= b $* (q zmod c) $+ r
⟨*proof*⟩

lemma *zdiv-zmult2-aux4*:
[| #0 $< c; #0 $<= r; r $< b |] ==> b $* (q zmod c) $+ r $< b $* c
⟨*proof*⟩

lemma *zdiv-zmult2-lemma*:
[| quorem (<a,b>, <q,r>); a ∈ int; b ∈ int; b ≠ #0; #0 $< c |]
==> quorem (<a,b$$*c>, <q zdiv c, b$$*(q zmod c) $+ r>)
⟨*proof*⟩

lemma *zdiv-zmult2-eq-raw*:
[|#0 $< c; a ∈ int; b ∈ int|] ==> a zdiv (b$$*c) = (a zdiv b) zdiv c
⟨*proof*⟩

lemma *zdiv-zmult2-eq*: #0 $< c ==> a zdiv (b$$*c) = (a zdiv b) zdiv c
⟨*proof*⟩

lemma *zmod-zmult2-eq-raw*:
[|#0 $< c; a ∈ int; b ∈ int|]
==> a zmod (b$$*c) = b$$*(a zdiv b zmod c) $+ a zmod b
⟨*proof*⟩

lemma *zmod-zmult2-eq*:
#0 $< c ==> 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) ≠ #0 |] ==> (c$$*a) zdiv (c$$*b) = a zdiv b
⟨*proof*⟩

lemma *zdiv-zmult-zmult1-aux2*:
[| b $< #0; intify(c) ≠ #0 |] ==> (c$$*a) zdiv (c$$*b) = a zdiv b
⟨*proof*⟩

lemma *zdiv-zmult-zmult1-raw*:

[|*intify*(*c*) ≠ #*0*; *b* ∈ *int*|] ==> (*c*$∗*a*) *zdiv* (*c*$∗*b*) = *a zdiv b*
⟨*proof*⟩

lemma *zdiv-zmult-zmult1*: *intify*(*c*) ≠ #*0* ==> (*c*$∗*a*) *zdiv* (*c*$∗*b*) = *a zdiv b*
⟨*proof*⟩

lemma *zdiv-zmult-zmult2*: *intify*(*c*) ≠ #*0* ==> (*a*$∗*c*) *zdiv* (*b*$∗*c*) = *a zdiv b*
⟨*proof*⟩

32.13 Distribution of factors over "zmod"

lemma *zmod-zmult-zmult1-aux1*:
[| #*0* $< *b*; *intify*(*c*) ≠ #*0* |]
==> (*c*$∗*a*) *zmod* (*c*$∗*b*) = *c* $∗ (*a zmod b*)
⟨*proof*⟩

lemma *zmod-zmult-zmult1-aux2*:
[| *b* $< #*0*; *intify*(*c*) ≠ #*0* |]
==> (*c*$∗*a*) *zmod* (*c*$∗*b*) = *c* $∗ (*a zmod b*)
⟨*proof*⟩

lemma *zmod-zmult-zmult1-raw*:
[|*b* ∈ *int*; *c* ∈ *int*|] ==> (*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* $< #*0*; #*0* $< *b* |] ==> *a zdiv b* $< #*0*
⟨*proof*⟩

lemma *zdiv-nonneg-neg-le0*: [| #*0* $<= *a*; *b* $< #*0* |] ==> *a zdiv b* $<= #*0*
⟨*proof*⟩

lemma *pos-imp-zdiv-nonneg-iff*: #*0* $< *b* ==> (#*0* $<= *a zdiv b*) <-> (#*0* $<= *a*)
⟨*proof*⟩

lemma *neg-imp-zdiv-nonneg-iff*: *b* $< #*0* ==> (#*0* $<= *a zdiv b*) <-> (*a* $<= #*0*)
⟨*proof*⟩

lemma *pos-imp-zdiv-neg-iff*: $\#0 \ \$< \ b \Longrightarrow (a \ zdiv \ b \ \$< \ \#0) <-> (a \ \$< \ \#0)$
⟨*proof*⟩

lemma *neg-imp-zdiv-neg-iff*: $b \ \$< \ \#0 \Longrightarrow (a \ zdiv \ b \ \$< \ \#0) <-> (\#0 \ \$< \ a)$
⟨*proof*⟩

end

33 CardinalArith: Cardinal Arithmetic Without the Axiom of Choice

theory *CardinalArith* **imports** *Cardinal OrderArith ArithSimp Finite* **begin**

definition
InfCard :: $i => o$ **where**
$InfCard(i) == Card(i) \ \& \ nat \ le \ i$

definition
cmult :: $[i,i] => i$ (**infixl** $|*|$ *70*) **where**
$i \ |*| \ j == |i*j|$

definition
cadd :: $[i,i] => i$ (**infixl** $|+|$ *65*) **where**
$i \ |+| \ j == |i+j|$

definition
csquare-rel :: $i => i$ **where**
$csquare\text{-}rel(K) ==$
$rvimage(K*K,$
$lam \ <x,y>:K*K. \ <x \ Un \ y, \ x, \ y>,$
$rmult(K,Memrel(K), \ K*K, \ rmult(K,Memrel(K), \ K,Memrel(K))))$

definition
jump-cardinal :: $i => i$ **where**
— This def is more complex than Kunen's but it more easily proved to be a cardinal
$jump\text{-}cardinal(K) ==$
$\bigcup X \in Pow(K). \ \{z. \ r: Pow(K*K), \ well\text{-}ord(X,r) \ \& \ z = ordertype(X,r)\}$

definition
csucc :: $i => i$ **where**
— needed because $jump\text{-}cardinal(K)$ 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))
⟨*proof*⟩

lemma *Card-UN*: (!!x. x:A ==> *Card*(K(x))) ==> *Card*($\bigcup x{\in}A$. K(x))
⟨*proof*⟩

lemma *Card-OUN* [*simp*,*intro*,*TC*]:
(!!x. x:A ==> *Card*(K(x))) ==> *Card*($\bigcup x{<}A$. K(x))
⟨*proof*⟩

lemma *n-lesspoll-nat*: $n \in nat$ ==> $n \prec nat$
⟨*proof*⟩

lemma *in-Card-imp-lesspoll*: [| *Card*(K); $b \in K$ |] ==> $b \prec K$
⟨*proof*⟩

lemma *lesspoll-lemma*: [| $\sim A \prec B$; $C \prec B$ |] ==> $A - C \neq 0$
⟨*proof*⟩

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-eqpoll*: $A+B \approx B+A$
⟨*proof*⟩

lemma *cadd-commute*: $i \mid+\mid j = j \mid+\mid i$
⟨*proof*⟩

33.1.2 Cardinal addition is associative

lemma *sum-assoc-eqpoll*: $(A+B)+C \approx A+(B+C)$
⟨*proof*⟩

lemma *well-ord-cadd-assoc*:
[| *well-ord*(*i*,*ri*); *well-ord*(*j*,*rj*); *well-ord*(*k*,*rk*) |]
==> (*i* |+| *j*) |+| *k* = *i* |+| (*j* |+| *k*)
⟨*proof*⟩

33.1.3 0 is the identity for addition

lemma *sum-0-eqpoll*: 0+*A* ≈ *A*
⟨*proof*⟩

lemma *cadd-0* [*simp*]: *Card*(*K*) ==> 0 |+| *K* = *K*
⟨*proof*⟩

33.1.4 Addition by another cardinal

lemma *sum-lepoll-self*: *A* ≲ *A*+*B*
⟨*proof*⟩

lemma *cadd-le-self*:
[| *Card*(*K*); *Ord*(*L*) |] ==> *K le* (*K* |+| *L*)
⟨*proof*⟩

33.1.5 Monotonicity of addition

lemma *sum-lepoll-mono*:
[| *A* ≲ *C*; *B* ≲ *D* |] ==> *A* + *B* ≲ *C* + *D*
⟨*proof*⟩

lemma *cadd-le-mono*:
[| *K*′ *le* *K*; *L*′ *le* *L* |] ==> (*K*′ |+| *L*′) *le* (*K* |+| *L*)
⟨*proof*⟩

33.1.6 Addition of finite cardinals is "ordinary" addition

lemma *sum-succ-eqpoll*: *succ*(*A*)+*B* ≈ *succ*(*A*+*B*)
⟨*proof*⟩

lemma *cadd-succ-lemma*:
[| *Ord*(*m*); *Ord*(*n*) |] ==> *succ*(*m*) |+| *n* = |*succ*(*m* |+| *n*)|
⟨*proof*⟩

lemma *nat-cadd-eq-add*: [| *m*: *nat*; *n*: *nat* |] ==> *m* |+| *n* = *m*#+*n*
⟨*proof*⟩

33.2 Cardinal multiplication

33.2.1 Cardinal multiplication is commutative

lemma *prod-commute-eqpoll*: $A*B \approx B*A$
⟨*proof*⟩

lemma *cmult-commute*: $i \mid*\mid j = j \mid*\mid i$
⟨*proof*⟩

33.2.2 Cardinal multiplication is associative

lemma *prod-assoc-eqpoll*: $(A*B)*C \approx A*(B*C)$
⟨*proof*⟩

lemma *well-ord-cmult-assoc*:
$[\!|\ \textit{well-ord}(i,ri);\ \textit{well-ord}(j,rj);\ \textit{well-ord}(k,rk)\ |\!]$
$\Longrightarrow (i \mid*\mid j) \mid*\mid k = i \mid*\mid (j \mid*\mid k)$
⟨*proof*⟩

33.2.3 Cardinal multiplication distributes over addition

lemma *sum-prod-distrib-eqpoll*: $(A+B)*C \approx (A*C)+(B*C)$
⟨*proof*⟩

lemma *well-ord-cadd-cmult-distrib*:
$[\!|\ \textit{well-ord}(i,ri);\ \textit{well-ord}(j,rj);\ \textit{well-ord}(k,rk)\ |\!]$
$\Longrightarrow (i \mid+\mid j) \mid*\mid k = (i \mid*\mid k) \mid+\mid (j \mid*\mid k)$
⟨*proof*⟩

33.2.4 Multiplication by 0 yields 0

lemma *prod-0-eqpoll*: $0*A \approx 0$
⟨*proof*⟩

lemma *cmult-0* [*simp*]: $0 \mid*\mid i = 0$
⟨*proof*⟩

33.2.5 1 is the identity for multiplication

lemma *prod-singleton-eqpoll*: $\{x\}*A \approx A$
⟨*proof*⟩

lemma *cmult-1* [*simp*]: $\textit{Card}(K) \Longrightarrow 1 \mid*\mid K = K$
⟨*proof*⟩

33.3 Some inequalities for multiplication

lemma *prod-square-lepoll*: $A \lesssim A*A$
⟨*proof*⟩

lemma *cmult-square-le*: $Card(K) ==> K\ le\ K\ |*|\ K$
⟨*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*:
$[|\ Card(K);\ \ Ord(L);\ \ 0<L\ |] ==> K\ le\ (K\ |*|\ L)$
⟨*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*:
$[|\ K'\ le\ K;\ \ L'\ le\ L\ |] ==> (K'\ |*|\ L')\ le\ (K\ |*|\ L)$
⟨*proof*⟩

33.4 Multiplication of finite cardinals is "ordinary" multiplication

lemma *prod-succ-eqpoll*: $succ(A)*B \approx B + A*B$
⟨*proof*⟩

lemma *cmult-succ-lemma*:
$[|\ Ord(m);\ \ Ord(n)\ |] ==> succ(m)\ |*|\ n = n\ |+|\ (m\ |*|\ n)$
⟨*proof*⟩

lemma *nat-cmult-eq-mult*: [| *m*: *nat*; *n*: *nat* |] ==> $m\ |*|\ n = m\#*n$
⟨*proof*⟩

lemma *cmult-2*: $Card(n) ==> 2\ |*|\ n = n\ |+|\ n$
⟨*proof*⟩

lemma *sum-lepoll-prod*: $2 \lesssim C ==> B+B \lesssim C*B$
⟨*proof*⟩

lemma *lepoll-imp-sum-lepoll-prod*: $[|\ A \lesssim B;\ 2 \lesssim A\ |] ==> A+B \lesssim A*B$
⟨*proof*⟩

33.5 Infinite Cardinals are Limit Ordinals

lemma *nat-cons-lepoll*: $nat \lesssim A ==> cons(u,A) \lesssim A$
⟨*proof*⟩

lemma *nat-cons-eqpoll*: $nat \lesssim A ==> cons(u,A) \approx A$
⟨*proof*⟩

lemma *nat-succ-eqpoll*: $nat <= A ==> succ(A) \approx A$
⟨*proof*⟩

lemma *InfCard-nat*: *InfCard*(*nat*)
⟨*proof*⟩

lemma *InfCard-is-Card*: $InfCard(K) ==> Card(K)$
⟨*proof*⟩

lemma *InfCard-Un*:
$[| InfCard(K);\ Card(L) |] ==> InfCard(K\ Un\ L)$
⟨*proof*⟩

lemma *InfCard-is-Limit*: $InfCard(K) ==> Limit(K)$
⟨*proof*⟩

lemma *ordermap-eqpoll-pred*:
$[| well\text{-}ord(A,r);\ x{:}A |] ==> ordermap(A,r)`x \approx Order.pred(A,x,r)$
⟨*proof*⟩

33.5.1 Establishing the well-ordering

lemma *csquare-lam-inj*:
$Ord(K) ==> (lam\ <x,y>{:}K*K.\ <x\ Un\ y,\ x,\ y>) : inj(K*K,\ K*K*K)$
⟨*proof*⟩

lemma *well-ord-csquare*: $Ord(K) ==> well\text{-}ord(K*K,\ csquare\text{-}rel(K))$
⟨*proof*⟩

33.5.2 Characterising initial segments of the well-ordering

lemma *csquareD*:
$[| <<x,y>, <z,z>> : csquare\text{-}rel(K);\ x<K;\ y<K;\ z<K |] ==> x\ le\ z\ \&\ y\ le\ z$
⟨*proof*⟩

lemma *pred-csquare-subset*:
$z<K \Longrightarrow Order.pred(K*K, <z,z>, csquare\text{-}rel(K)) <= succ(z)*succ(z)$
⟨*proof*⟩

lemma *csquare-ltI*:
$[\!| x<z;\ y<z;\ z<K |\!] \Longrightarrow\ <<x,y>, <z,z>> : csquare\text{-}rel(K)$
⟨*proof*⟩

lemma *csquare-or-eqI*:
$[\!| x\ le\ z;\ y\ le\ z;\ z<K |\!] \Longrightarrow <<x,y>, <z,z>> : csquare\text{-}rel(K) \mid x=z\ \&\ y=z$
⟨*proof*⟩

33.5.3 The cardinality of initial segments

lemma *ordermap-z-lt*:
$[\!| Limit(K);\ x<K;\ y<K;\ z=succ(x\ Un\ y) |\!] \Longrightarrow$
$ordermap(K*K, csquare\text{-}rel(K))\ `\ <x,y> <$
$ordermap(K*K, csquare\text{-}rel(K))\ `\ <z,z>$
⟨*proof*⟩

lemma *ordermap-csquare-le*:
$[\!| Limit(K);\ x<K;\ y<K;\ z=succ(x\ Un\ y) |\!]$
$\Longrightarrow | ordermap(K*K, csquare\text{-}rel(K))\ `\ <x,y> | le\ |succ(z)| \ |*|\ |succ(z)|$
⟨*proof*⟩

lemma *ordertype-csquare-le*:
$[\!| InfCard(K);\ ALL\ y{:}K.\ InfCard(y) --> y\ |*|\ y = y |\!]$
$\Longrightarrow ordertype(K*K, csquare\text{-}rel(K))\ le\ K$
⟨*proof*⟩

lemma *InfCard-csquare-eq*: $InfCard(K) \Longrightarrow K\ |*|\ K = K$
⟨*proof*⟩

lemma *well-ord-InfCard-square-eq*:
$[\!| well\text{-}ord(A,r);\ InfCard(|A|) |\!] \Longrightarrow A*A \approx A$
⟨*proof*⟩

lemma *InfCard-square-eqpoll*: $InfCard(K) \Longrightarrow K \times K \approx K$
⟨*proof*⟩

lemma *Inf-Card-is-InfCard*: $[\!| \sim Finite(i);\ Card(i) |\!] \Longrightarrow InfCard(i)$
⟨*proof*⟩

33.5.4 Toward's Kunen's Corollary 10.13 (1)

lemma *InfCard-le-cmult-eq*: [| *InfCard*(*K*); *L le K*; *0<L* |] ==> *K* |*| *L* = *K*
⟨*proof*⟩

lemma *InfCard-cmult-eq*: [| *InfCard*(*K*); *InfCard*(*L*) |] ==> *K* |*| *L* = *K Un L*
⟨*proof*⟩

lemma *InfCard-cdouble-eq*: *InfCard*(*K*) ==> *K* |+| *K* = *K*
⟨*proof*⟩

lemma *InfCard-le-cadd-eq*: [| *InfCard*(*K*); *L le K* |] ==> *K* |+| *L* = *K*
⟨*proof*⟩

lemma *InfCard-cadd-eq*: [| *InfCard*(*K*); *InfCard*(*L*) |] ==> *K* |+| *L* = *K 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*: *Ord*(*jump-cardinal*(*K*))
⟨*proof*⟩

lemma *jump-cardinal-iff*:
i : *jump-cardinal*(*K*) <->
(*EX r X*. *r* <= *K***K* & *X* <= *K* & *well-ord*(*X*,*r*) & *i* = *ordertype*(*X*,*r*))
⟨*proof*⟩

lemma *K-lt-jump-cardinal*: *Ord*(*K*) ==> *K* < *jump-cardinal*(*K*)
⟨*proof*⟩

lemma *Card-jump-cardinal-lemma*:
[| *well-ord*(*X*,*r*); *r* <= *K* * *K*; *X* <= *K*;
f : *bij*(*ordertype*(*X*,*r*), *jump-cardinal*(*K*)) |]
==> *jump-cardinal*(*K*) : *jump-cardinal*(*K*)
⟨*proof*⟩

lemma *Card-jump-cardinal*: *Card*(*jump-cardinal*(*K*))
⟨*proof*⟩

33.7 Basic Properties of Successor Cardinals

lemma *csucc-basic*: $Ord(K) ==> Card(csucc(K)) \ \& \ K < csucc(K)$
⟨*proof*⟩

lemmas *Card-csucc* = *csucc-basic* [*THEN conjunct1*, *standard*]

lemmas *lt-csucc* = *csucc-basic* [*THEN conjunct2*, *standard*]

lemma *Ord-0-lt-csucc*: $Ord(K) ==> 0 < csucc(K)$
⟨*proof*⟩

lemma *csucc-le*: $[| \ Card(L); \ K<L \ |] ==> csucc(K) \ le \ L$
⟨*proof*⟩

lemma *lt-csucc-iff*: $[| \ Ord(i); \ Card(K) \ |] ==> i < csucc(K) <-> |i| \ le \ K$
⟨*proof*⟩

lemma *Card-lt-csucc-iff*:
$[| \ Card(K'); \ Card(K) \ |] ==> K' < csucc(K) <-> K' \ le \ K$
⟨*proof*⟩

lemma *InfCard-csucc*: $InfCard(K) ==> InfCard(csucc(K))$
⟨*proof*⟩

33.7.1 Removing elements from a finite set decreases its cardinality

lemma *Fin-imp-not-cons-lepoll*: A: $Fin(U) ==> x^{\sim}:A \ --> \ ^{\sim} \ cons(x,A) \lesssim A$
⟨*proof*⟩

lemma *Finite-imp-cardinal-cons* [*simp*]:
$[| \ Finite(A); \ \ a^{\sim}:A \ |] ==> |cons(a,A)| = succ(|A|)$
⟨*proof*⟩

lemma *Finite-imp-succ-cardinal-Diff*:
$[| \ Finite(A); \ \ a:A \ |] ==> succ(|A-\{a\}|) = |A|$
⟨*proof*⟩

lemma *Finite-imp-cardinal-Diff*: $[| \ Finite(A); \ \ a:A \ |] ==> |A-\{a\}| < |A|$
⟨*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 (⋃ *C*) −−>
(∀ *c*∈*C*. |*c*| = *k*) −−>
(∀ *c1* ∈ *C*. ∀ *c2* ∈ *C*. *c1* ≠ *c2* −−> *c1* ∩ *c2* = *0*) −−>
k #∗ |*C*| = |⋃ *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-eqpoll-sum*: [| *m*:*nat*; *n*:*nat* |] ==> *m* + *n* ≈ *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* ~= *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* ((-^- '(-')) [*60*,*1000*,*1000*] *60*)

primrec
F^*0* (*x*) = *x*
F^(*succ*(*n*)) (*x*) = *F*(*F*^*n* (*x*))

definition
iterates-omega :: $[i=>i,i] => i$ **where**
iterates-omega$(F,x) == \bigcup n{\in}nat.\ F\hat{}n\ (x)$

notation (*xsymbols*)
iterates-omega ((-$\hat{}\omega$ $'$(-$'$)) [60,1000] 60)
notation (*HTML* **output**)
iterates-omega ((-$\hat{}\omega$ $'$(-$'$)) [60,1000] 60)

lemma *iterates-triv*:
$[|\ n{\in}nat;\ \ F(x) = x\ |] ==> F\hat{}n\ (x) = x$
⟨*proof*⟩

lemma *iterates-type* [*TC*]:
$[|\ n{:}nat;\ \ a{:}\ A;\ !!x.\ x{:}A ==> F(x) : A\ |]$
$==> F\hat{}n\ (a) : A$
⟨*proof*⟩

lemma *iterates-omega-triv*:
$F(x) = x ==> F\hat{}\omega\ (x) = x$
⟨*proof*⟩

lemma *Ord-iterates* [*simp*]:
$[|\ n{\in}nat;\ \ !!i.\ Ord(i) ==> Ord(F(i));\ \ Ord(x)\ |]$
$==> Ord(F\hat{}n\ (x))$
⟨*proof*⟩

lemma *iterates-commute*: $n \in nat ==> F(F\hat{}n\ (x)) = F\hat{}n\ (F(x))$
⟨*proof*⟩

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 $b($*Arith.pred*$(x),\ r\ `\ $*Arith.pred*$(x)))$

lemma *transrec3-0* [*simp*]: *transrec3*$(0,a,b,c) = a$
⟨*proof*⟩

lemma *transrec3-succ* [*simp*]:
transrec3$(succ(i),a,b,c) = b(i,\ $*transrec3*$(i,a,b,c))$
⟨*proof*⟩

lemma *transrec3-Limit*:
    Limit(*i*) ==>
     transrec3(*i*,*a*,*b*,*c*) = *c*(*i*, λ*j*∈*i*. *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* ==> (*EX y*. *y*:*B*(*x*)) |] ==> *EX z*. *z* : *Pi*(*A*,*B*)

lemma *AC-Pi*: [| !!*x*. *x* ∈ *A* ==> (∃ *y*. *y* ∈ *B*(*x*)) |] ==> ∃ *z*. *z* ∈ *Pi*(*A*,*B*)
⟨*proof*⟩

lemma *AC-ball-Pi*: ∀ *x* ∈ *A*. ∃ *y*. *y* ∈ *B*(*x*) ==> ∃ *y*. *y* ∈ *Pi*(*A*,*B*)
⟨*proof*⟩

lemma *AC-Pi-Pow*: ∃ *f*. *f* ∈ (Π *X* ∈ *Pow*(*C*)−{*0*}. *X*)
⟨*proof*⟩

lemma *AC-func*:
    [| !!*x*. *x* ∈ *A* ==> (∃ *y*. *y* ∈ *x*) |] ==> ∃ *f* ∈ *A*−>*Union*(*A*). ∀ *x* ∈ *A*. *f*‘*x* ∈ *x*
⟨*proof*⟩

lemma *non-empty-family*: [| *0* ∉ *A*; *x* ∈ *A* |] ==> ∃ *y*. *y* ∈ *x*
⟨*proof*⟩

lemma *AC-func0*: *0* ∉ *A* ==> ∃ *f* ∈ *A*−>*Union*(*A*). ∀ *x* ∈ *A*. *f*‘*x* ∈ *x*
⟨*proof*⟩

lemma *AC-func-Pow*: ∃ *f* ∈ (*Pow*(*C*)−{*0*}) −> *C*. ∀ *x* ∈ *Pow*(*C*)−{*0*}. *f*‘*x* ∈ *x*

⟨*proof*⟩

lemma *AC-Pi0*: $0 \notin A \Longrightarrow \exists f.\ f \in (\Pi\ x \in A.\ x)$
⟨*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**
   $Subset\text{-}rel(A) == \{z \in A*A .\ \exists x\ y.\ z=\langle x,y\rangle\ \&\ x\leq y\ \&\ x\neq y\}$

definition
  chain :: $i \Rightarrow i$ **where**
   $chain(A) == \{F \in Pow(A).\ \forall X \in F.\ \forall Y \in F.\ X \leq Y \mid Y \leq X\}$

definition
  super :: $[i,i] \Rightarrow i$ **where**
   $super(A,c) == \{d \in chain(A).\ c \leq d\ \&\ c \neq d\}$

definition
  maxchain :: $i \Rightarrow i$ **where**
   $maxchain(A) == \{c \in chain(A).\ super(A,c)=0\}$

definition
  increasing :: $i \Rightarrow i$ **where**
    $increasing(A) == \{f \in Pow(A) \to Pow(A).\ \forall x.\ x \leq A \longrightarrow x \leq f`x\}$

Lemma for the inductive definition below

lemma *Union-in-Pow*: $Y \in Pow(Pow(A)) \Longrightarrow Union(Y) \in Pow(A)$
⟨*proof*⟩

We could make the inductive definition conditional on $next \in increasing(S)$ 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: [| $x \in TFin(S,next)$; $next \in increasing(S)$ |] ==> $next`x \in TFin(S,next)$

Pow-UnionI: $Y \in Pow(TFin(S,next)) ==> Union(Y) \in TFin(S,next)$

monos *Pow-mono*
con-defs *increasing-def*
type-intros *CollectD1* [*THEN apply-funtype*] *Union-in-Pow*

36.1 Mathematical Preamble

lemma *Union-lemma0*: $(\forall x \in C.\ x <= A \mid B <= x) ==> Union(C) <= A \mid B <= Union(C)$
⟨*proof*⟩

lemma *Inter-lemma0*:
[| $c \in C$; $\forall x \in C.\ A <= x \mid x <= B$ |] ==> $A <= Inter(C) \mid Inter(C) <= B$
⟨*proof*⟩

36.2 The Transfinite Construction

lemma *increasingD1*: $f \in increasing(A) ==> f \in Pow(A) -> Pow(A)$
⟨*proof*⟩

lemma *increasingD2*: [| $f \in increasing(A)$; $x <= A$ |] ==> $x <= f`x$
⟨*proof*⟩

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*:
[| $n \in TFin(S,next)$;
!!x. [| $x \in TFin(S,next)$; $P(x)$; $next \in increasing(S)$ |] ==> $P(next`x)$;
!!Y. [| $Y <= TFin(S,next)$; $\forall y \in Y.\ P(y)$ |] ==> $P(Union(Y))$
|] ==> $P(n)$
⟨*proof*⟩

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*:
[| $n \in TFin(S,next)$; $m \in TFin(S,next)$;
$\forall x \in TFin(S,next)\ .\ x <= m --> x = m \mid next`x <= m$ |]
==> $n <= m \mid next`m <= n$

⟨*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*:
[| $m \in TFin(S,next)$; $next \in increasing(S)$ |]
==> $\forall n \in TFin(S,next).\ n<=m \dashrightarrow n=m \mid next`n <= m$
⟨*proof*⟩

a more convenient form for Lemma 2

lemma *TFin-subsetD*:
[| $n<=m$; $m \in TFin(S,next)$; $n \in TFin(S,next)$; $next \in increasing(S)$ |]
==> $n=m \mid next`n <= m$
⟨*proof*⟩

Consequences from section 3.3 – Property 3.2, the ordering is total

lemma *TFin-subset-linear*:
[| $m \in TFin(S,next)$; $n \in TFin(S,next)$; $next \in increasing(S)$ |]
==> $n <= m \mid m<=n$
⟨*proof*⟩

Lemma 3 of section 3.3

lemma *equal-next-upper*:
[| $n \in TFin(S,next)$; $m \in TFin(S,next)$; $m = next`m$ |] ==> $n <= m$
⟨*proof*⟩

Property 3.3 of section 3.3

lemma *equal-next-Union*:
[| $m \in TFin(S,next)$; $next \in increasing(S)$ |]
==> $m = next`m \dashrightarrow m = Union(TFin(S,next))$
⟨*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*: $chain(A) <= Pow(A)$
⟨*proof*⟩

lemma *super-subset-chain*: $super(A,c) <= chain(A)$
⟨*proof*⟩

lemma *maxchain-subset-chain*: $maxchain(A) <= chain(A)$
⟨*proof*⟩

lemma *choice-super*:
[| $ch \in (\Pi\ X \in Pow(chain(S)) - \{0\}.\ X)$; $X \in chain(S)$; $X \notin maxchain(S)$ |]
==> $ch\ `\ super(S,X) \in super(S,X)$
⟨*proof*⟩

lemma *choice-not-equals*:
[| $ch \in (\Pi\ X \in Pow(chain(S)) - \{0\}.\ X)$; $X \in chain(S)$; $X \notin maxchain(S)$ |]
==> $ch\ `\ super(S,X) \neq X$
⟨*proof*⟩

This justifies Definition 4.4

lemma *Hausdorff-next-exists*:
$ch \in (\Pi\ X \in Pow(chain(S))-\{0\}.\ X)$ ==>
$\exists next \in increasing(S).\ \forall X \in Pow(S).$
$next`X = if(X \in chain(S)-maxchain(S),\ ch`super(S,X),\ X)$
⟨*proof*⟩

Lemma 4

lemma *TFin-chain-lemma4*:
[| $c \in TFin(S,next)$;
$ch \in (\Pi\ X \in Pow(chain(S))-\{0\}.\ X)$;
$next \in increasing(S)$;
$\forall X \in Pow(S).\ next`X =$
$if(X \in chain(S)-maxchain(S),\ ch`super(S,X),\ X)$ |]
==> $c \in chain(S)$
⟨*proof*⟩

theorem *Hausdorff*: $\exists c.\ c \in maxchain(S)$
⟨*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*:
[| $c \in chain(A)$; $z \in A$; $\forall x \in c.\ x<=z$ |] ==> $cons(z,c) \in chain(A)$
⟨*proof*⟩

lemma *Zorn*: $\forall c \in chain(S).\ Union(c) \in S$ ==> $\exists y \in S.\ \forall z \in S.\ y<=z$ --> $y=z$
⟨*proof*⟩

Alternative version of Zorn's Lemma

theorem *Zorn2*:
$\forall c \in chain(S).\ \exists y \in S.\ \forall x \in c.\ x <= y$ ==> $\exists y \in S.\ \forall z \in S.\ y<=z$ --> $y=z$
⟨*proof*⟩

36.6 Zermelo's Theorem: Every Set can be Well-Ordered

Lemma 5

lemma *TFin-well-lemma5*:
[| $n \in TFin(S,next)$; $Z <= TFin(S,next)$; $z{:}Z$; $\sim Inter(Z) \in Z$ |]
==> $\forall m \in Z.\ n <= m$
⟨*proof*⟩

Well-ordering of $TFin(S,\ next)$

lemma *well-ord-TFin-lemma*: [| $Z <= TFin(S,next)$; $z \in Z$ |] ==> $Inter(Z) \in Z$
⟨*proof*⟩

This theorem just packages the previous result

lemma *well-ord-TFin*:
$next \in increasing(S)$
==> $well\text{-}ord(TFin(S,next),\ Subset\text{-}rel(TFin(S,next)))$
⟨*proof*⟩

* Defining the "next" operation for Zermelo's Theorem *

lemma *choice-Diff*:
[| $ch \in (\Pi\ X \in Pow(S) - \{0\}.\ X)$; $X \subseteq S$; $X \neq S$ |] ==> $ch\ `\ (S-X) \in S-X$
⟨*proof*⟩

This justifies Definition 6.1

lemma *Zermelo-next-exists*:
$ch \in (\Pi\ X \in Pow(S)-\{0\}.\ X)$ ==>
$\exists next \in increasing(S).\ \forall X \in Pow(S).$
$next`X = (if\ X=S\ then\ S\ else\ cons(ch`(S-X),\ X))$
⟨*proof*⟩

The construction of the injection

lemma *choice-imp-injection*:
[| $ch \in (\Pi\ X \in Pow(S)-\{0\}.\ X)$;
$next \in increasing(S)$;
$\forall X \in Pow(S).\ next`X = if(X=S,\ S,\ cons(ch`(S-X),\ X))$ |]
==> $(\lambda\ x \in S.\ Union(\{y \in TFin(S,next).\ x \notin y\}))$
$\in inj(S,\ TFin(S,next) - \{S\})$
⟨*proof*⟩

The wellordering theorem

theorem *AC-well-ord*: $\exists r.\ well\text{-}ord(S,r)$
⟨*proof*⟩

36.7 Zorn's Lemma for Partial Orders

Reimported from HOL by Clemens Ballarin.

definition *Chain* :: i => i **where**
  $Chain(r) = \{A : Pow(field(r)).\ ALL\ a{:}A.\ ALL\ b{:}A.\ <a,\ b> : r\ |\ <b,\ a> : r\}$

lemma *mono-Chain*:
  $r \subseteq s$ ==> $Chain(r) \subseteq Chain(s)$
  ⟨*proof*⟩

theorem *Zorn-po*:
  assumes *po*: $Partial\text{-}order(r)$
    and *u*: $ALL\ C{:}Chain(r).\ EX\ u{:}field(r).\ ALL\ a{:}C.\ <a,\ u> : r$
  shows $EX\ m{:}field(r).\ ALL\ a{:}field(r).\ <m,\ a> : r$ --> $a = m$
⟨*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-eqpoll*: $|A|$ *eqpoll* A
⟨*proof*⟩

The theorem $||A|| = |A|$

lemmas *cardinal-idem* = *cardinal-eqpoll* [*THEN cardinal-cong*, *standard*, *simp*]

lemma *cardinal-eqE*: $|X| = |Y|$ ==> X *eqpoll* Y
⟨*proof*⟩

lemma *cardinal-eqpoll-iff*: $|X| = |Y|$ <-> X *eqpoll* Y
⟨*proof*⟩

lemma *cardinal-disjoint-Un*:
    [| $|A|=|B|$; $|C|=|D|$; $A\ Int\ C = 0$; $B\ Int\ D = 0$ |]
     ==> $|A\ Un\ C| = |B\ Un\ D|$
⟨*proof*⟩

lemma *lepoll-imp-Card-le*: A *lepoll* B ==> $|A|$ *le* $|B|$
⟨*proof*⟩

lemma *cadd-assoc*: $(i\ |+|\ j)\ |+|\ k = i\ |+|\ (j\ |+|\ k)$
⟨*proof*⟩

lemma *cmult-assoc*: $(i\ |*|\ j)\ |*|\ k = i\ |*|\ (j\ |*|\ k)$
⟨*proof*⟩

lemma *cadd-cmult-distrib*: $(i\ |+|\ j)\ |*|\ k = (i\ |*|\ k)\ |+|\ (j\ |*|\ k)$

⟨*proof*⟩

lemma *InfCard-square-eq*: *InfCard*(|*A*|) ==> *A*∗*A eqpoll A*
⟨*proof*⟩

37.2 The relationship between cardinality and le-pollence

lemma *Card-le-imp-lepoll*: |*A*| *le* |*B*| ==> *A lepoll B*
⟨*proof*⟩

lemma *le-Card-iff*: *Card*(*K*) ==> |*A*| *le K* <−> *A lepoll K*
⟨*proof*⟩

lemma *cardinal-0-iff-0* [*simp*]: |*A*| = *0* <−> *A* = *0*
⟨*proof*⟩

lemma *cardinal-lt-iff-lesspoll*: *Ord*(*i*) ==> *i* < |*A*| <−> *i lesspoll A*
⟨*proof*⟩

lemma *cardinal-le-imp-lepoll*: *i* ≤ |*A*| ==> *i* ≲ *A*
⟨*proof*⟩

37.3 Other Applications of AC

lemma *surj-implies-inj*: *f*: *surj*(*X*,*Y*) ==> *EX g*. *g*: *inj*(*Y*,*X*)
⟨*proof*⟩

lemma *surj-implies-cardinal-le*: *f*: *surj*(*X*,*Y*) ==> |*Y*| *le* |*X*|
⟨*proof*⟩

lemma *cardinal-UN-le*:
[| *InfCard*(*K*); *ALL i*:*K*. |*X*(*i*)| *le K* |] ==> |⋃*i*∈*K*. *X*(*i*)| *le K*
⟨*proof*⟩

lemma *cardinal-UN-lt-csucc*:
[| *InfCard*(*K*); *ALL i*:*K*. |*X*(*i*)| < *csucc*(*K*) |]
==> |⋃*i*∈*K*. *X*(*i*)| < *csucc*(*K*)
⟨*proof*⟩

lemma *cardinal-UN-Ord-lt-csucc*:
[| *InfCard*(*K*); *ALL i*:*K*. *j*(*i*) < *csucc*(*K*) |]
==> (⋃*i*∈*K*. *j*(*i*)) < *csucc*(*K*)
⟨*proof*⟩

lemma *inj-UN-subset*:
[| f: $inj(A,B)$; a:A |] ==>
($\bigcup x{\in}A.\ C(x)$) <= ($\bigcup y{\in}B.\ C(\mathit{if}\ y{:}\ range(f)\ \mathit{then}\ converse(f)`y\ \mathit{else}\ a)$)
⟨*proof*⟩

lemma *le-UN-Ord-lt-csucc*:
[| $InfCard(K)$; $|W|$ *le* K; *ALL* $w{:}W.\ j(w) < csucc(K)$ |]
==> ($\bigcup w{\in}W.\ j(w)$) < $csucc(K)$
⟨*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 -> $Vfrom(A,csucc(K))$; $|D|$ *le* K; $InfCard(K)$ |]
==> *EX* $j.\ f{:}\ D$ -> $Vfrom(A,j)$ & j < $csucc(K)$
⟨*proof*⟩

lemma *subset-Vcsucc*:
[| D <= $Vfrom(A,csucc(K))$; $|D|$ *le* K; $InfCard(K)$ |]
==> *EX* $j.\ D$ <= $Vfrom(A,j)$ & j < $csucc(K)$
⟨*proof*⟩

lemma *fun-Vcsucc*:
[| $|D|$ *le* K; $InfCard(K)$; D <= $Vfrom(A,csucc(K))$ |] ==>
D -> $Vfrom(A,csucc(K))$ <= $Vfrom(A,csucc(K))$
⟨*proof*⟩

lemma *fun-in-Vcsucc*:
[| f: D -> $Vfrom(A,\ csucc(K))$; $|D|$ *le* K; $InfCard(K)$;
D <= $Vfrom(A,csucc(K))$ |]
==> f: $Vfrom(A,csucc(K))$
⟨*proof*⟩

lemmas *fun-in-Vcsucc′ = fun-in-Vcsucc [OF - - - subsetI]*

lemma *Card-fun-Vcsucc*:
InfCard(K) ==> K −> Vfrom(A,csucc(K)) <= Vfrom(A,csucc(K))
⟨*proof*⟩

lemma *Card-fun-in-Vcsucc*:
[| f: K −> Vfrom(A, csucc(K)); InfCard(K) |] ==> f: Vfrom(A,csucc(K))
⟨*proof*⟩

lemma *Limit-csucc*: *InfCard(K) ==> 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